Chapter 1

Manual for the Kria KV260

The Kria KV260 Vision artificial intelligence (AI) Starter Kit by Advanced Micro Devices (AMD) [1], used for this study, is a kit that includes an field programmable gate array (FPGA) with a pre-configured system on module (SOM) with a deep learning/data processing unit (DPU), carrier card and thermal solution. The tutorials that were mainly followed for the usage of the FPGA were the Getting Started with Kria KV260 Vision AI Starter Kit [2] and the Quick Start Guide for Zynq UltraScale+ by Xilinx [3], as well as the Vitis AI User Guide [4], but they have many faults and missing information. So, in this manual, we hope to clear any doubts, reduce the amount of trial and error needed in future work and make the usage of the FPGA smoother in general.

1.1 Versions

This section aims to list the versions of the software used, such that the reader can trace back any errors by up/down-grading to these:

- Ubuntu 20.04 long term support (LTS) for the host machine,
- Vitis AI v3.0 for use in the host machine,
- Vitis AI pre-build image for the Kria KV260, including the B4096 DPU architecture.

1.2 Start-up

The first step to working with the Kria KV260 is preparing an image from which it can boot. The module can leverage its own operating system (OS). Still, in this study, we used an SD card image pre-built specifically for the Kria KV260 for use with Vitis AI [5], as this version already comes prepared for use with Vitis AI and can compile in the B4096 architecture, the latest and faster (at the time of writing) architecture for the KV26 SOM, which, for example, an Ubuntu 20.04 LTS version-based system cannot.

The image should be burned on a microSD card with at least 8GB of memory, but at least 32GB is recommended. The Balena Etcher [6] tool can be used to do this.

Once the microSD card is ready, we can start booting up the Kria KV260. For this, the following materials are needed:

1. MicroSD card burned with the Vitis AI pre-built image
2. USB keyboard and mouse
3. Monitor with a HMDI or DisplayPort (DP) cable
4. Ethernet cable
5. Power adapter

Those should be connected, in order, to the following ports in the Kria KV260 as shown in Figure 1.1:

1. J11 MicroSD
2. 2x USB 3.0
3. J5 HDMI or J6 DisplayPort
4. J10 RJ-45 Ethernet
5. J12 DC Jack

After a couple of minutes, the monitor should show the FPGA booting. The default username and password are both *root*.

To run in the FPGA more smoothly, one can connect via secure shell protocol (SSH) [7] from the host machine. This is also useful to pass files between the host and the target (the FPGA). For this, first one needs to know the IP address of the target, which can be found with the following command:

```
ifconfig eth0
```

Listing 1.1: Code snippet to get IP address. Run on the target.

If you are using a monitor, you can do this directly on the FPGA by using the USB mouse and keyboard to reach the terminal. However, it is also possible to do this without a monitor, keyboard or mouse, by connecting a micro-USB to USB cable that allows for data transfer between the target and the host and accessing the FPGA terminal via UART connection.

To connect via SSH from the host to the target, one can use the following command:

```
ssh -X root@[TARGET_IP_ADDRESS]
```

Listing 1.2: Code snippet to connect via SSH. Run on the host.

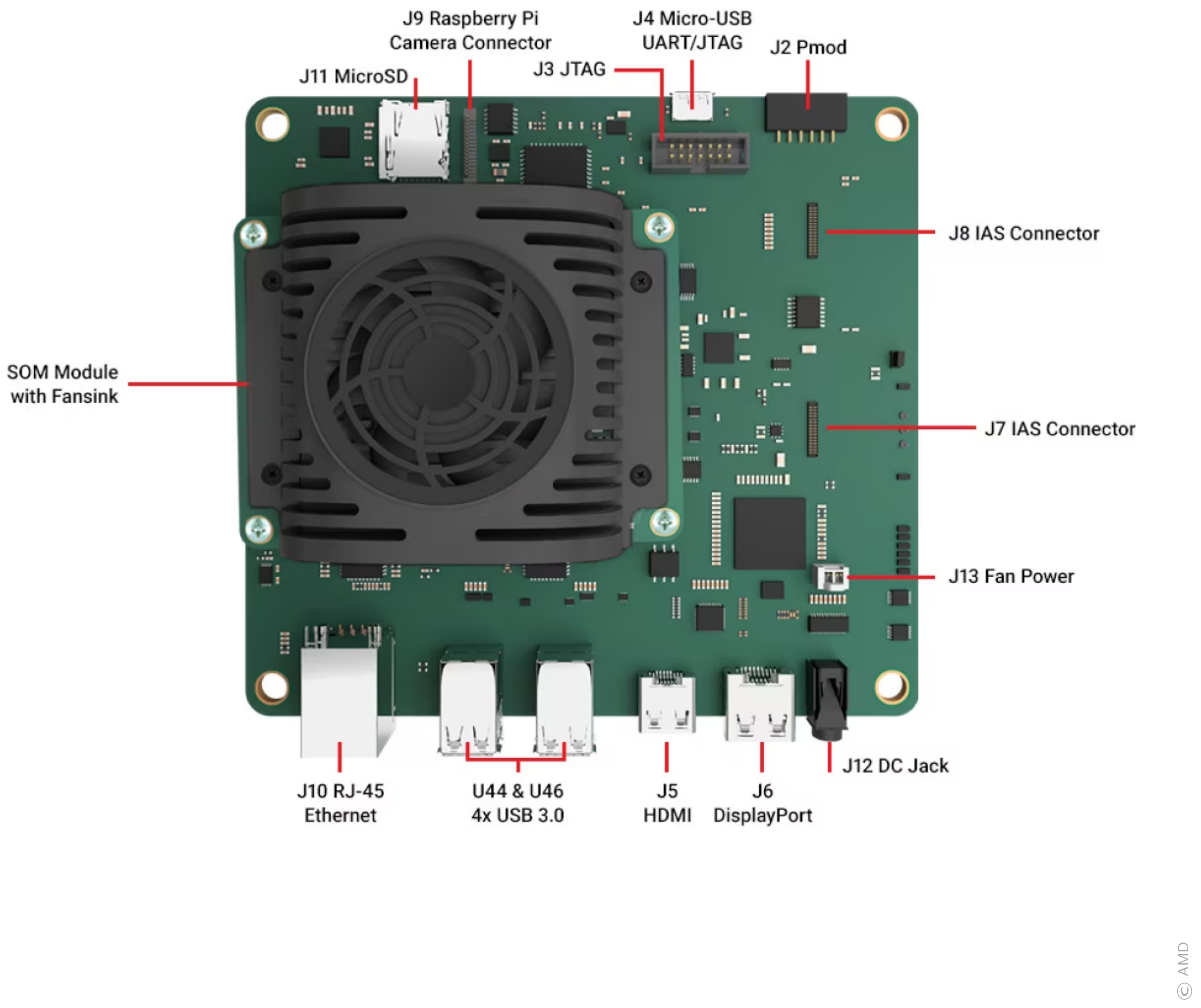

Figure 1.1: Kria KV260 board with labelled inputs [2].

The terminal on your host can now control the target's terminal and run whatever you want from there. Note that running anything on that specific terminal will run it on the FPGA, not on the host, so if you need to use the terminal on your own machine, open a new one. Now, the monitor, keyboard and mouse are no longer needed.

In order to pass files between the host and the target, one can use:

```
scp [FILE] root@[TARGET_IP_ADDRESS]:~/
```

Listing 1.3: Code snippet to pass files via SSH to the home directory of the target. Run on the host.

Note that you might need to add the flag `-O` (capital o) when using `scp` if you have a recent OpenSSH release installed on your machine.

The FPGA is ready for the deployment of a neural network (NN) model.

1.3 Preparing the neural network for the field programmable gate array: Vitis AI

1.3.1 Preparation

To prepare the NN trained and tested in a graphical processing unit (GPU) for the FPGA, external software needs to be leveraged. Vitis AI [8] was used for this study. At the time of writing, the latest stable version for the Kria KV260 is v3.0, so this one was used.

To install Vitis AI v3.0, clone the Vitis AI GitHub repository [9] into a personal machine and checkout into the desired version by using the following on a terminal:

```
git clone https://github.com/Xilinx/Vitis-AI.git
git checkout v3.0
```

Listing 1.4: Code snippet to install Vitis AI. Run on the host.

This study used a machine with the Ubuntu 20.04 LTS version as the host.

1.3.2 Docker

Docker [10] will also be needed to use Vitis AI. To install docker in the Ubuntu 20.04 LTS version, the next steps should be followed, as described in the official Docker documentation [11]:

```
# to uninstall any conflicting packages:
for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd
    ↪ runc; do sudo apt-get remove $pkg; done

# add docker's official gpg key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# add the repository to apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://
      ↪ download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

# install the latest version of docker
```

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
    ↪ compose-plugin

# check if the installation is successful; a hello world message should be printed on the
    ↪ terminal after running this command
sudo docker run hello-world
```

Listing 1.5: Code snippet to install Docker. Run on host.

To be able to run Docker freely, non-root users should be added to the Docker group, like so [12]:

```
# create the docker group; this command may return saying that the group already exists,
    ↪ continue to the next step
sudo groupadd docker

# add your user to the docker group
sudo usermod -aG docker $USER

# activate the changes
newgrp docker

# verify that the command docker works without the command sudo
docker run hello-world
```

Listing 1.6: Code snippet to add current user to the Docker group. Run on host.

Now, we can leverage the pre-built Docker containers by Xilinx. If you have access to a compute unified device architecture (CUDA)-enabled GPU [13], you can leverage a container that allows for GPU usage and the processes will run much faster. To verify this, you may check the list of CUDA GPUs [14] or look up your own GPU online to check if it is CUDA-enabled (for example, the NVIDIA GeForce MX330 GPU was used in this study [15] – you can see the CUDA version is 6.1). If you do not know if you have access to a GPU or which one you have, you can run the following on a terminal:

```
sudo lshw -C display
```

Listing 1.7: Code snippet to verify which, if any, GPU is installed. Run on host.

1.3.2.1 If you have access to a GPU

Prepare the host for the installation of the CUDA-powered Docker container by running the following to uninstall conflicting libraries and installing dependencies:

```
sudo apt purge nvidia* libnvidia*
sudo apt install nvidia-driver-510
sudo apt install nvidia-container-toolkit
```

Listing 1.8: Code snippet to install the NVIDIA driver and the Container Toolkit. Run on host.

You may install another version of the driver by switching out *nvidia-driver-510* with *nvidia-driver-[VERSION]*, but the version 510 is recent and stable at the time of writing. To confirm that the installation was successful, run:

```
nvidia-smi
```

Listing 1.9: Code snippet to check driver status. Run on host.

If the installation was successful, you should be able to see the Driver and CUDA version.

To build a CUDA-powered Docker container, first navigate to the *docker* folder inside the Vitis AI folder:

```
cd [VITIS_AI_INSTALLATION_PATH]/Vitis-AI/docker
```

Listing 1.10: Code snippet to navigate to the pre-built docker containers folder. Run on host.

There are various options to build Docker containers. In this study, we used PyTorch [16] and allowed for the possibility of using the Vitis AI Optimizer, in which case the target framework is *opt_pytorch*. If you do not have a license for the Vitis AI Optimizer, the target framework is simply *pytorch*.

```
./docker_build.sh -t gpu -f [TARGET_FRAMEWORK (e.g. opt_pytorch)]
```

Listing 1.11: Code snippet to build the pre-built docker container for a specific target framework on the GPU. Run on host.

The containers with access to a GPU have some faults, namely that the Conda [17] environments that are already set up on them do not have all of the libraries needed to run some of the programs for quantizing and compiling the model. However, most of them are fairly easy to resolve by searching on the internet for the error code (and most will be resolved by using *pip install [LIBRARY]*). Still, if you find yourself stuck, you can always rely on the CPU-based docker image, as it does the same things; it will simply take longer.

1.3.2.2 If you do not have access to a GPU

Installing the CPU-based Docker is much easier and can be done with the simple command:

```
docker pull xilinx/vitis-ai-pytorch-cpu:latest
```

Listing 1.12: Code snippet to build the pre-built docker container for PyTorch [16] on the CPU. Run on host.

1.3.2.3 Running the Docker container

After the containers are built (it might take a while), you are ready to run them. To do this, make sure you are back in the main Vitis AI folder:

```
cd [VITIS_AI_INSTALLATION_PATH]/Vitis-AI
```

Listing 1.13: Code snippet to go to the Vitis AI main folder. Run on host.

Afterwards, use the file *docker_run.sh* to run your container followed by the container you chose to leverage. If you have forgotten the name/keywords for your container, you may find your installed containers by running the following command:

```
docker images -a
```

Listing 1.14: Code snippet to list built Docker containers. Run on host.

Select the one you want to work with. This manual will use the CPU-based one from here on out for simplicity, except when mentioned otherwise, but everything is also applicable to the GPU-based applications. To run the container, type out the following:

```
./docker_run.sh xilinx/vitis-ai-pytorch-cpu:latest
```

Listing 1.15: Code snippet to run the CPU-based Docker container for PyTorch [16]. Run on host.

If this was successful, you should get a message saying "Vitis AI" in big letters and you should be logged in as *vitis-ai-user*

1.3.3 Conda environment

After you are inside the Docker container, we need to activate the Conda [17] environment already built inside this Docker container. To find the name of the environment, run:

```
conda info --envs
```

Listing 1.16: Code snippet to list the pre-built Conda environments. Run on host inside Docker.

If you are using the CPU-based Docker and the PyTorch framework, activating the container will be something along the lines:

```
conda activate vitis-ai-pytorch
```

Listing 1.17: Code snippet to activate the pre-built Conda environment. Run on host inside Docker.

If this is successful, the prompt will show that you are inside the Conda environment.

1.3.4 Model inspector

Vitis AI provides a program that can inspect your model to see whether it will run on the DPU. Some layers are currently not supported by the DPU and some layer orderings are not optimal, and the model inspector [18] can point those out. The supported layers are mentioned in the Vitis AI User Guide [19], as well as the supported sizes [20]. Some popular layer types, like Conv1d, are not possible without modifications to the hardware.

In order to run the model inspector, you can use the code provided by Xilinx [21] or use the slightly modified version that was used for this study [22].

To open the Jupyter Notebook inside the Docker container and inside the the Conda environment, navigate to where your *model_inspector.ipynb* file is and run it as follows:

```
jupyter notebook model_inspector ipynb
```

Listing 1.18: Code snippet to run the model inspector. Run on host inside Docker inside Conda environment.

The output of the Jupyter Notebook will be an image indicating where the layers of the NN will be able to run in the FPGA (CPU or DPU), as well as a text file with the same information.

In order to take best advantage of the FPGA and its DPU, all layers should be able to run in the DPU, and this is what will make the most difference in the latency of your model. The inspector will give tips on how to optimize the model for the DPU, such as layer orderings, but it might also just throw an error if you are using layers that are not supported [23], which should be easily fixed by searching online.

1.3.5 Quantizer

In this study, we used post-training quantization (PTQ) and, hence, this is the method presented in this manual.

In order to quantize your pre-trained model, you need to use the quantizer provided with the pytorch_nndct library [24], which is pre-installed in the conda environment inside the docker container.

Xilinx provides a pre-made script for quantization, but once again it is very specific for vision-based applications. As such, a more general script can be found in the GitHub for this study [22].

In order to be able to run the quantizer, the Python file of the model (.py) and the trained model file (.pt) should be in a folder. In this case, it was a folder called "model", so we will refer to it as that, but this is possible to change without touching the script, via the arguments passed to it. More on that later. Additionally, a folder with validation samples should also be made, with between 100 and 1000 samples. This folder was called "dataset".

You can evaluate the accuracy of your initial floating point model, but this is an optional step in the quantization procedure. If you wish to do so, you can use the following command:

```
python quantize.py --quant_mode float --data_dir dataset --model_dir model --model_name
    ↪ GregNet2D --decision_threshold 0.86
```

Listing 1.19: Code snippet to evaluate the accuracy of the floating point model. Run on host inside Docker inside Conda environment.

The arguments passed to the scrip include:

- **quant_mode**: the mode in which the script is being run,
- **data_dir**: the name of the folder where the validation samples are,

- **model_dir**: the name of the folder where the .py and .pt model files are,
- **model_name**: the name of the model you want to run the script for, if you have more than one,
- **decision_theshold**: the decision threshold that should be used to evaluate the accuracy.

These are used in a similar way for the rest of the commands.

In order to start quantization, run the following command:

```
python quantize.py --quant_mode calib --data_dir dataset --model_dir model --model_name [
    ↪ MODEL_NAME] --decision_threshold [DECISION_THRESHOLD]
```

Listing 1.20: Code snippet to start quantization and generate the quantized vai_q_pytorch format model and the quantization steps of tensors. Run on host inside Docker inside Conda environment.

This command will generate a new folder called "quantize_result", that includes a quantized vai_q_pytorch format model (.py) and a file with the quantization steps of tensors information (Quant_info.json).

In order to optionally evaluate the accuracy of the quantized model, you can use the following command:

```
python quantize.py --quant_mode test --data_dir dataset --model_dir model --model_name [
    ↪ MODEL_NAME] --decision_threshold [DECISION_THRESHOLD]
```

Listing 1.21: Code snippet to evaluate the accuracy of the quantized model. Run on host inside Docker inside Conda environment.

In order to generate the .xmodel quantized file, needed for the compilation for the FPGA, you can use the following command:

```
python quantize.py --quant_mode test --data_dir dataset --model_dir model --model_name [
    ↪ MODEL_NAME] --decision_threshold [DECISION_THRESHOLD] --subset_len 1 --batch_size 1 --
    ↪ deploy
```

Listing 1.22: Code snippet to generate the .xmodel file. Run on host inside Docker inside Conda environment.

The quantization step is finished, and we can now move on to compiling the quantized INT8.xmodel into a deployable DPU.xmodel, fit for the particular architecture being used.

1.3.6 Compiler

In order to compile the model for the correct architecture, check and modify the arch.json file inside the folder /opt/vitis_ai/compiler/arch/DPUCZDX8G/KV260. The default is already the B4096 architecture, so nothing needs to be done if you are using this one.

To compile the model, run the following command:

```
vai_c_xir -x quantize_result/[MODEL_NAME]_int.xmodel -a /opt/vitis_ai/compiler/arch/DPUCZDX8G
    ↪ /KV260/arch.json -o [MODEL_NAME] -n [MODEL_NAME]
```

Listing 1.23: Code snippet to compile the model. Run on host inside Docker inside Conda environment.

If the compilation is successful, a new folder with the name of your model will be generated containing the .xmodel file, generated according to the specified DPU architecture.

Additionally, it is advised to create a .prototxt file with configurations of the model. An example is given in the GitHub [22].

1.4 Model deployment

In order to deploy the model, we first need to copy the necessary files to the FPGA. We start with copying the model folder, which contains the `.xmodel`, the `md5sum.txt` and the `meta.json` files. We can do that with the following command, from the main folder where you run Vitis AI:

```
scp -r [MODEL_NAME] root@[TARGET_IP_ADDRESS]:/usr/share/vitis_ai_library/models/
```

Listing 1.24: Code snippet to pass the model files via SSH to a specified location on the target. Run on the host.

The location can be whatever you prefer, but this is the default one according to the Linux folder structure and what the OS already comes prepared with from Vitis AI.

Gather test samples in a folder and similarly copy them to the FPGA. We chose to make a new folder inside the Vitis-AI folder to keep the program and data together:

```
scp -r dataset root@[TARGET_IP_ADDRESS]:~/Vitis-AI/[PROJECT_NAME]
```

Listing 1.25: Code snippet to pass the samples folder via SSH to a specified location on the target. Run on the host.

An example of a program used to deploy the model can be found in the project GitHub [22]. To copy it to the organised directory in the target, run the following command:

```
scp -r app_mt.py root@[TARGET_IP_ADDRESS]:~/Vitis-AI/[PROJECT_NAME]
```

Listing 1.26: Code snippet to copy the app to the target. Run on the host.

To run it, use the following command:

```
python app_mt.py --data_dir dataset --threads [NUMBER_OF_THREADS] --model_name [MODEL_NAME]
    ↪ --decision_threshold [DECISION_THRESHOLD]
```

Listing 1.27: Code snippet to deploy the model. Run on the target.

Acronyms

1D	one-dimensional
2D	two-dimensional
AdaMax	adaptive moment estimation with maximum
AI	artificial intelligence
AMD	Advanced Micro Devices
ANN	artificial neural network
APU	application processing unit
aLIGO	advanced LIGO
BBH	binary black hole
BH	black hole
BNS	binary neutron star
CBC	compact binary coalescence
CE	cosmic explorer
CNN	convolutional neural network
CPU	central processing unit
CUDA	compute unified device architecture
DP	DisplayPort
DNN	deep neural network
DL	deep learning
DPU	deep learning/data processing unit
EOS	equation of state
ET	Einstein Telescope
FAP	false alarm probability
FLOP	floating point operation
FN	false negative
FP	false positive

FPGA	field programmable gate array
GAN	generative adversarial network
GNN	graph neural network
GPU	graphical processing unit
GR	general relativity
GRB	gamma-ray burst
GW	gravitational wave
IP	intellectual property
KAGRA	Kamioka Gravitational Wave Detector
HDL	hardware description language
LIGO	Laser Interferometer Gravitational-Wave Observatory
LISA	Laser Interferometer Space Antenna
LR	learning rate
LSTM	Long Short-Term Memory
LTS	long term support
LUT	look-up table
MMA	multi-messenger astrophysics
ML	machine learning
NN	neural network
NS	neutron star
NSBH	neutron star-black hole binary
O3	observing run 3
O4	observing run 4
OS	operating system
PE	processing engine
PINN	physics-informed neural network
PISNR	partial inspiral signal-to-noise ratio
PL	programmable logic
PLD	programmable logic device
PS	processing system
PSD	power spectral density
PTQ	post-training quantization
QAT	quantization-aware training
RAM	random access memory
ReLU	rectified linear unit
RNN	residual neural network

ROC	receiver operating characteristic
sGRB	short gamma-ray burst
SNR	signal-to-noise ratio
SoC	system-on-a-chip
SOM	system on module
SSH	secure shell protocol
TAP	true alarm probability
TN	true negative
TP	true positive
UU	Utrecht University

Bibliography

[1] Advanced Micro Devices, Inc. *AMD Kria KV260 Vision Starter Kit*. Accessed 27 March 2024. Available from: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html (cit. on p. 1).

[2] Advanced Micro Devices, Inc. *AMD Kria KV260 Vision Starter Kit - Getting Started*. Accessed 28 March 2024. Available from: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit/getting-started-ubuntu/getting-started.html (cit. on pp. 1, 3).

[3] Xilinx, Inc. *Xilinx Vitis-AI Quickstart Documentation*. Accessed 27 March 2024. Available from: https://xilinx.github.io/Vitis-AI/3.0/html/docs/quickstart/mpsoc.html (cit. on p. 1).

[4] Advanced Micro Devices, Inc. *AMD Vitis AI Documentation*. Accessed 02 April 2024. Available from: https://docs.amd.com/r/3.0-English/ug1414-vitis-ai (cit. on p. 1).

[5] Xilinx, Inc. *Xilinx KV260 DPU Download Page*. Accessed 02 April 2024. Available from: https://www.xilinx.com/member/forms/download/design-license-xef.html?filename=xilinx-kv260-dpu-v2022.2-v3.0.0.img.gz (cit. on p. 1).

[6] balena. *balenaEtcher*. Accessed 27 March 2024. Available from: https://etcher.balena.io (cit. on p. 2).

[7] SSH.COM. *SSH Protocol*. Accessed 02 April 2024. Available from: https://www.ssh.com/academy/ssh/protocol (cit. on p. 2).

[8] Xilinx, Inc. *Xilinx Vitis AI*. Accessed 28 March 2024. Available from: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html (cit. on p. 4).

[9] Xilinx, Inc. *Xilinx Vitis-AI GitHub Repository*. Accessed 28 March 2024. Available from: https://github.com/Xilinx/Vitis-AI (cit. on p. 4).

[10] Docker, Inc. *Docker*. Accessed 02 April 2024. Available from: https://www.docker.com (cit. on p. 4).

[11] Docker, Inc. *Install Docker Engine on Ubuntu*. Accessed 02 April 2024. Available from: https://docs.docker.com/engine/install/ubuntu/ (cit. on p. 4).

[12] Docker, Inc. *Linux post-installation steps*. Accessed 02 April 2024. Available from: https://docs.docker.com/engine/install/linux-postinstall/ (cit. on p. 5).

[13] NVIDIA Corporation. *What is CUDA?* Accessed 02 April 2024. Available from: https://nvidia.custhelp.com/app/answers/detail/a_id/2132/~/what-is-cuda%3F (cit. on p. 5).

[14] NVIDIA Corporation. *NVIDIA CUDA GPUs*. Accessed 02 April 2024. Available from: https://developer.nvidia.com/cuda-gpus (cit. on p. 5).

[15] TechPowerUp. *GeForce MX330 GPU Specifications*. Accessed 02 April 2024. Available from: https://www.techpowerup.com/gpu-specs/geforce-mx330.c3493 (cit. on p. 5).

[16] A. Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG] (cit. on pp. 6, 7).

[17] Conda Documentation Team. *Conda Documentation*. Accessed 03 April 2024. Available from: https://docs.conda.io/en/latest/ (cit. on pp. 6, 7).

[18] Xilinx, Inc. *Xilinx Vitis AI Documentation - Model Development Workflow - Model Inspector*. Accessed 04 April 2024. Available from: https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-model-development.html#model-inspector (cit. on p. 7).

[19] Advanced Micro Devices, Inc. *AMD Vitis AI Documentation - Operators Supported by PyTorch*. Accessed 2024. Available from: https://docs.amd.com/r/3.0-English/ug1414-vitis-ai/Operators-Supported-by-PyTorch (cit. on p. 7).

[20] Advanced Micro Devices, Inc. *AMD Vitis AI Documentation - Currently Supported Operators*. Accessed 22 April 2024. Available from: https://docs.amd.com/r/3.0-English/ug1414-vitis-ai/Currently-Supported-Operators (cit. on p. 7).

[21] Xilinx, Inc. *Xilinx Vitis AI GitHub Repository - Inspector Tutorial*. GitHub repository. Accessed 04 April 2024. Available from: https://github.com/Xilinx/Vitis-AI/blob/3.0/examples/vai_quantizer/pytorch/inspector_tutorial.ipynb (cit. on p. 7).

[22] Ana Martins. *Deploying Custom Neural Networks in FPGA - Model Inspector*. GitHub repository. Accessed 04 April 2024. Available from: https://github.com/anaismartins/deploying-custom-nn-in-fpga/blob/main/src/model_inspector.ipynb (cit. on pp. 7, 8, 10).

[23] AMD. *Error Codes - Vitis AI Documentation*. https://docs.amd.com/r/3.0-English/ug1414-vitis-ai/Error-Codes. Accessed 02 June 2024. 2024 (cit. on p. 8).

[24] Xilinx. *Vitis AI Quantizer for PyTorch - README*. https://github.com/Xilinx/Vitis-AI/blob/master/src/vai_quantizer/vai_q_pytorch/README.md. Accessed 02 June 2024. 2024 (cit. on p. 8).