ICT-56-2020 — Next Generation Internet of Things

D 3.3
Evaluation of the DL accelerator designs

Document information	
Contract number	957197
Project website	www.vedliot.eu
Dissemination Level	PU
Nature	R
Contractual Deadline	31.10.2022
Author	Rene Griessl (UNIBI)
Contributors	Karol Gugala (ANT), Grzegorz Latosinski (ANT), Daniel Ödman (EmbeDL), Hans Salomonsson (EmbeDL), Mario Porrmann (UOS), Marco Tassemeier (UOS), Pedro Trancoso (CHALMERS), Fareed Mohammad Qararyah (CHALMERS), Stavroula Zouzoula (CHALMERS), Kevin Mika (UNIBI), Florian Porrmann (UNIBI), Jens Hagemeyer (UNIBI)
Reviewers	Marcelo Parsin (UNINE), Pascal Felber (UNINE)

The VEDLIoT project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957197.

Changelog		
v0.1	2022-08-15	Initial draft derived from D3.1
v0.2	2022-09-19	Refined structure
v0.3	2022-10-06	Accuracy results added
v0.4	2022-10-10	Internal review
v1.0	2022-10-31	Finalization
v1.1	2023-10-25	Review of Hailo-8 results; added OrinNX results

Inhalt

Executive Summary

This deliverable contains joint work of WP6 and WP3, providing an overview of the current state of the edge AI systems and how VEDLIoT can leverage and improve the existing ecosystem. The content has been produced by Task 6.1 (Survey of Deep Learning Inference Tools, Interfaces and Compilers) and Task 3.1 (Evaluation of existing architectures and compilers for DL), as the two tasks are related, covering different aspects of edge AI systems. A preliminary report focusing on tools and compilers has been submitted as D6.1 (Report on existing hardware/software interfaces for DL and compilers) before.

In D3.1 the current state of the art in edge AI processing was presented, it walks the reader through the detailed description of the most popular deep learning frameworks, platforms, and compilers. It covers prominent deep-learning models and in-depth discussion of novel deep learning accelerator platforms, including their architectures and performance.

Deliverable D3.3 builds up on D3.1, it is the in-depth evaluation of available DL accelerator platforms within VEDLIoT, comparing the performance and energy-efficiency of different accelerators among each other's and against a commonly accepted baseline. In comparison to D3.1 the amount of evaluated platforms was extended and the accuracy for each platform was measured to validate the consistency of the performance evaluation.

The Kenning platform presented in this report aims to provide an interface for switching between various compilers depending on chosen hardware and models. Kenning provides support for a wide range of available technology, offering automated benchmarking, characterization, and deployment. In D3.3 Kenning was used for evaluation of open-source DL compilers, the results are compared to vendor tools. In addition it generated gradient graphs for direct comparison of different tool chains.

In summary, this report presents an in-depth overview of deep learning tools, models and technology, combined with a sophisticated methodology for evaluation and benchmarking new accelerators in a heterogenous hardware environment. Extensive benchmarking and research have been performed, and results of vendor dependent DL compilers have been validated by accuracy checks.

1 Introduction

This report is a survey of Deep Learning Inference Tools, Interfaces and Compilers, as well as deep learning platforms and benchmarking, which incorporate either generic compute devices like CPUs or GPUs, or specialized accelerators for deep learning in different AIoT domains. Differences in vendor dependent toolchains have been analysed by calculation of the accuracy for each platform. As shown in Figure 1, this deliverable focused on the work which was performed in Task 6.1 (Survey of Deep Learning Inference Tools, Interfaces and Compilers) as well as Task 3.1 (Evaluation of existing architectures and compilers for DL), as the two tasks are closely related, covering different aspects of edge AI systems. This deliverable was previously released in an intermediate version as D3.1, which does not include the validation of the vendor specific toolchain by accuracy analysis.

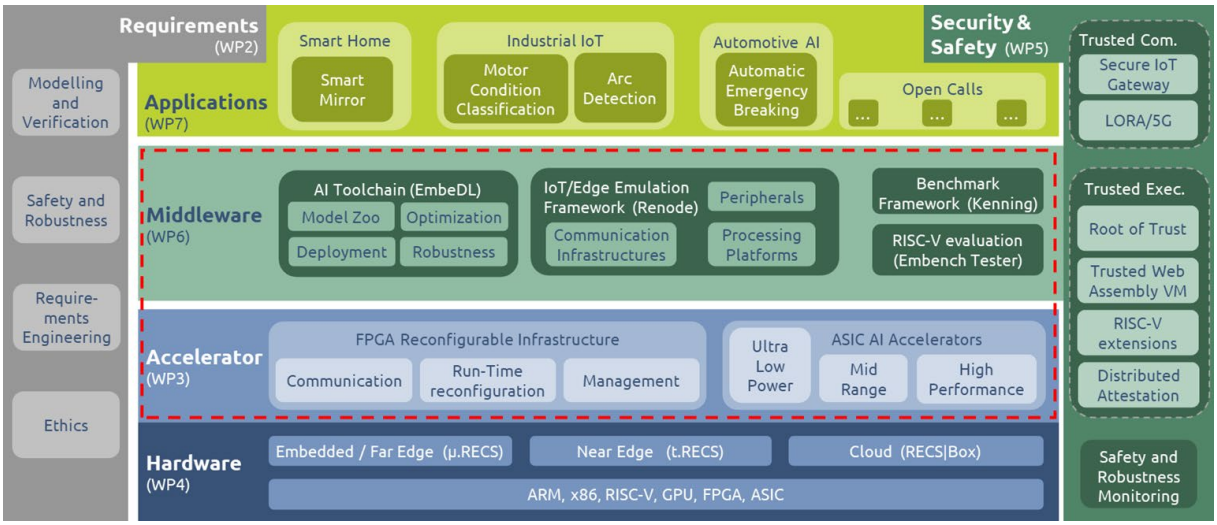

Figure 1: VEDLIoT abstracted overview. Parts covered by this report appear in the middle part (WP6 and WP3)

Currently, there are many frameworks and libraries for model training. With the increasing interest in deploying machine learning, and specifically deep learning models on IoT devices, the deep learning compilers are developed.

Deep learning compilers optimize deep neural networks and create an efficient runtime for a given target device. There are frameworks such as Apache TVM, Glow, TensorFlow Lite and many others that generate optimized runtimes for various deep learning accelerators.

In addition, various vendors are releasing more and more hardware platforms that can be used to run deep learning models – they differ in energy consumption, computational power and inference speed.

The aim of this report is to:

- Elaborate on existing Deep Learning frameworks for training and inference, as well as the involved compilers and models
- Provide an overview of the large spectrum of novel deep-learning hardware platforms and accelerators
- Present comprehensive benchmarking results obtained on various hardware platforms

- Present a tool for automated benchmarking and deployment, as well as the integration into the VEDLIoT platform
- Comparison of vendor specific DL compiler vs. open-source available DL compiler

The report contains the following Chapters:

- Chapter 2 (Deep Learning Tool Sets)
  - (Deep Learning deployment stack) describes the typical deep learning application development flow from training to deployment on IoT devices.
  - (Deep Learning Frameworks) lists the most popular and up-to-date deep learning frameworks.
  - (Deep Learning Compilers) lists the compilers for deep learning models from one representation to another.
  - (Deep Learning Models) introduces different commonly used models collected in a VEDLIoT Model Zoo.
- Chapter 3 (Evaluation of DL Accelerators)
  - (Methodology) defines a common methodology that is used during measurements and benchmarking in the following chapters.
  - (Deep Learning Platforms) discusses the architecture, performance and energy efficiency of novel deep learning accelerator ASICs and IP Cores.
  - (Accuracy Results) presents accuracy for DL compilers used in the Evaluation
  - (Evaluation Results) focuses on the extensive benchmarking activities on the different platforms.
  - (Comparison of AI Accelerators) presents and discusses the result of the benchmarking.

2 Deep Learning Tool Sets

2.1 Deep Learning deployment stack

This Chapter lists and describes typical actions performed on deep learning models before deployment on target devices.

2.1.1 From training to deployment

A deep learning application deployed on IoT devices usually goes through the following process:

- a dataset is prepared for a deep learning process,
- evaluation metrics are specified based on a given dataset and outputs,
- data in the dataset goes through analysis, data loaders that perform the pre-processing are implemented,
- deep learning model is either designed from scratch or the baseline is selected from a wide selection of existing pre-trained models for the given deep learning application (classification, detection, semantic segmentation, instance segmentation, etc.) and adjusted to a particular use case,
- a loss function and learning algorithm is specified along with a deep learning model,
- the model is trained, evaluated and improved,
- the model is compiled to a representation that is applicable to a given target,
- the model is executed on a target device.

2.1.2 Dataset preparation

If a model is not available or is trained for a different use case, the model must be trained or re-trained.

Each model requires a dataset — a set of sample inputs (audio signals, images, video sequences, OCT images, other sensors) and usually also outputs (association to class or classes, object location, object mask, input description). The dataset is usually subdivided into:

- training dataset — the largest subset that is used to train the model,
- validation dataset — the relatively small set that is used to verify the model performance after each training epoch (the metrics and loss function values demonstrate if there is any overfitting during the training process),
- test dataset — the subset that acts as the final evaluation of a trained model.

It is required that the test dataset is mutually exclusive with the training dataset so that the evaluation results are not biased in any way.

Datasets can be either designed from scratch or found for instance in:

- Kaggle datasets,

- Google Dataset Search,
- Dataset list,
- Universities' pages,
- Open Images Dataset,
- Common Voice Dataset.

2.1.3 Model preparation and training

Currently, the most popular approach is to find an existing model that fits a given problem and performs transfer learning to adapt the model to the requirements. In transfer learning the existing model is slightly modified in its final layers to adapt to a new problem, and the last layers of the model are trained using a training dataset. Finally, some additional layers are unfreezed and the training is performed on a larger number of parameters at a very small learning rate — this process is called fine-tuning.

Transfer learning gives a better starting point for the training process, allows us to train a correctly performing model with smaller datasets and reduces the time required to train the model. The intuition behind this is that there are lots of common features between various objects in real-life environments, and the features learned in one deep learning scenario can be reused in another scenario.

Once the model is selected, adequate data inputs pre-processing needs to be provided in order to perform valid training. The input data should be normalized and resized to fit input tensor requirements. In case of the training dataset, especially if it is quite small, applying reasonable data augmentations, like random brightness, contrast, cropping, jitters, rotations can significantly improve the training process and prevent the network from overfitting.

In the end, a proper training procedure needs to be specified. This step includes:

- specifying loss function for the model. Some weights regularizations can be specified, along with the loss function, to reduce the chance of overfitting
- specifying optimizers (like Adam, Adagrad). This includes setting hyperparameters properly or adding schedules and automated routines to set those hyperparameters (i.e., scheduling the learning rate value, or using LR-Finder to set the proper learning rate for the scenario)
- specifying or scheduling the number of epochs, i.e., early stopping can be introduced
- providing some routines for quality metrics measurements
- providing some routines for saving an intermediate model during training (periodically, or the best model according to some quality measure)

2.1.4 Model optimization

A successfully trained model may require some optimizations in order to run on a given IoT hardware. The optimizations may regard the precision of weights, or the computational representation, or the model structure.

Models are usually trained in FP32 precision or mixed precision (FP32 + FP16, depending on the operator). Some targets, on the other hand, may significantly benefit from changing the FP32 precision to FP16, INT8 or INT4 precision. The optimizations here are straightforward for the FP16 precision, but the integer-based quantization require calibration datasets to reduce the precision without a significant loss of the models' quality.

Other optimizations change the computational representation of the model by, e.g., layer fusion, specialized operators for convolutions of a particular shape, and other.

In the end, there are algorithmic optimizations that change the whole model structure, like weight's pruning, conditional computation, model distillation (the current model acts as a teacher that is supposed to improve the quality of a much smaller model).

If the model optimizations are applied, the optimized models should be evaluated using the same metrics as the original model. This is required in order to find any quality drops.

2.1.5 Model compilation and deployment

Deep learning compilers can transform model representation to:

- a source code for a different programming language, e.g., Halide, C, C++, Java that can be later used on a given target,
- a machine code utilizing available hardware accelerators with supporting libraries, e.g., OpenGL, OpenCL, CUDA, TensorRT, ROCm,
- FPGA bitstream,
- other targets.

Those compiled models are optimized to perform as efficiently as possible on a given target hardware. In the final step, the models are deployed on a hardware device.

2.2 Deep Learning Frameworks

This Chapter describes the most popular deep learning frameworks for model training and development.

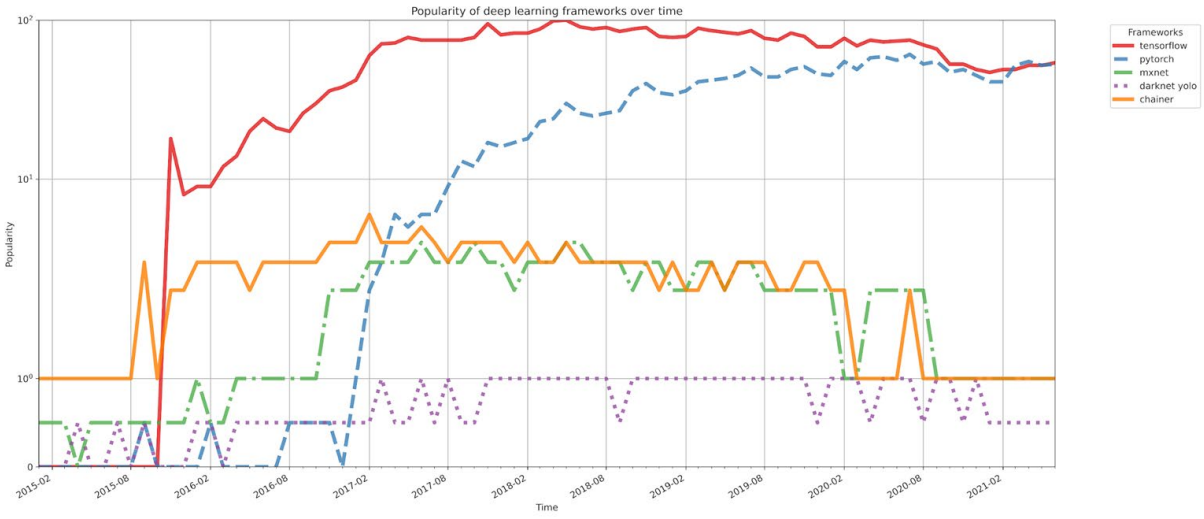

Figure 2: Popularity of frameworks over time (based on Google Trends), between 2016-06 till 2021-06. The popularity is normalized to range between 0 and 100, where 100 means the highest popularity in a given period of time.

2.2.1 TensorFlow

2.2.1.1 General information

- Homepage: https://www.tensorflow.org/
- License: Apache-2.0
- Repository: https://github.com/tensorflow/tensorflow
- Documentation: TensorFlow documentation
- Analyzed release: 2.5.0
- Supported languages:
  - C++
  - Python
  - C (TFLite)
  - JavaScript (TensorFlow.JS)
- Partially supported languages:
  - C#
  - Go
  - Haskell
  - Java
  - JavaSript
  - Julia
  - R
  - Ruby

- Scala
- Swift

2.2.1.2 Description

TensorFlow is a library for machine learning algorithms development. The API is available in Python, but it is possible to use models in C++, C (using TFLite), JavaScript (using TensorFlow.JS) and other languages above.

It allows training models on CPU, GPUs, TPUs and in distributed environments consisting of GPU/TPU clusters. It provides the Datasets (tf.data) API for easy data loading, pre-processing and distribution across training hardware. The models can be either designed using low-level TensorFlow API, or using Keras blocks to keep the code simpler.

TensorFlow comes with a Tensorboard application that allows a user to track the performance metrics, quality metrics, training progress and custom data from the browser. Tensorboard allows us to analyse the model structure, plot metrics' changes during training, plot weights' values distribution to make sure that the training process is progressing correctly.

TensorFlow 1.x operated in a session mode — this means the model graph was first constructed, along with training and data processing functions, and then all functions were executed in a session. During inference, the session also needed to be constructed. This disallowed the user to customize the inference or training process using native Python.

Since TensorFlow 2.x, the eager execution of inference and training became available, so it is possible to run TensorFlow functions outside of a session and get immediate results. Still, the seemingly most efficient way to train the model is to use TensorFlow APIs for data pre-processing and training loop creation. Those require the user to use the TensorFlow functions only, in order to form optimized training flow (utilizing multi-threading pre-processing, parallel training and data fetching, and other optimizations).

2.2.1.3 TensorFlow children frameworks and enhancements

2.2.1.3.1 Keras

Keras wraps up low-level TensorFlow calls and provides a simplified API for designing deep learning models.

Keras provides high-level APIs for layers, models, data pre-processing, optimizers, metrics and losses that can be used to create new models with relatively small amount of code. It comes with a long list of ready-to-use models for image classification, which can be used in transfer learning, or they also can be used directly.

It is also integrated with the TensorFlow 2.x releases.

2.2.1.3.2 Sonnet

As Keras, Sonnet provides a high-level API for layers and modules wrapping up the low-level TensorFlow calls. It is designed, built and by researchers at Deep Mind.

2.2.1.3.3 TensorFlow Lite

TensorFlow Lite is a deep learning model compiler for running models on mobile and IoT devices. It can be described more in-depth in Deep Learning Compilers.

2.2.1.3.4 TensorFlow.JS

TensorFlow.JS contains an API for converting and deploying deep learning models in JavaScript. It supports both Keras and TensorFlow saved models.

2.2.1.3.5 TensorFlow Addons

TensorFlow Addons is an optional library providing additional layers, callbacks, activations, metrics, data pre-processing routines and more that haven't been added yet to the TensorFlow library.

2.2.1.3.6 TensorFlow Model Garden

TensorFlow Model Garden is a repository with a large base of various state-of-the-art deep learning models, including official, community and research models.

2.2.1.3.7 TensorFlow Model Optimization Toolkit

TensorFlow Model Optimization Toolkit is a set of tools for model compression — quantization, weights pruning and clustering. It is used in TensorFlow Lite for various hardware-specific model optimizations.

2.2.1.3.8 ONNX conversions

TensorFlow does not have built-in routines for ONNX conversions. The tensorflow-onnx and onnx-tensorflow projects can be used for converting the models to and from the ONNX. These projects allow exporting models from TensorFlow and importing models to TensorFlow. Both repositories are maintained by the ONNX community. Check the tf2onnx and onnx_tf support status pages for supported ONNX operators.

2.2.2 PyTorch

2.2.2.1 General information

- Homepage: https://pytorch.org/
- License: BSD-3
- Repository: https://github.com/pytorch/pytorch
- Documentation: PyTorch documentation
- Analyzed release: 1.7.1
- Supported languages:
  - C++
  - Java
  - Python

2.2.2.2 Description

PyTorch is a Python package for Tensor computation with strong GPU acceleration and training deep neural networks. It is deeply integrated into Python, is imperative and can cooperate seamlessly with the Python code. This allows to highly customize the training, inference and processing flow.

The library consists of both low-level and high-level API for model designing. It provides an extensive API for data pre-processing and efficient data fetching.

PyTorch allows the user to run and train the models on CPUs, GPUs and in distributed systems. Thanks to PyTorch XLA it is also possible to train the model on Cloud TPUs.

The library provides JIT compilation to create very efficient deep learning flows.

While it does not provide a dedicated tool for visualizing learning data (like loss or metrics), it can work with TensorBoard.

2.2.2.3 PyTorch children frameworks and enhancements

2.2.2.3.1 Distiller

IntelLabs/distiller project is a Python package for neural network compression. It provides:

- Pruning — weights pruning, structured pruning,
- Regularizations — they ease the further process of pruning and quantization,
- Quantization algorithms — conversion to INT8 networks,
- Knowledge distillation — training smaller networks by utilizing knowledge from larger networks trained for the same problem,
- Conditional computation,
- Other optimization techniques.

It supports converting the compressed models to ONNX.

2.2.2.3.2 PyTorch Mobile

PyTorch Mobile provides API for running PyTorch models on iOS, Android and Linux. It supports 8-bit kernels, per-channel quantization, dynamic quantizations and more. PyTorch Mobile supports GPU, DSP and NPUs accelerators.

2.2.2.3.3 PyTorch Vision and Audio

torchvision is a package that consists of popular vision datasets, vision model architectures and image transformations. torchaudio is a package for PyTorch in the audio domain — it provides methods for loading and processing audio data.

2.2.2.3.4 Detectron2

Detectron2 is a library built on top of PyTorch for computer vision algorithms, especially object detection, instance and semantic segmentation and pose estimation. It provides methods for training models for above-mentioned tasks, and an impressive list of pre-trained models.

2.2.2.4 ONNX conversions

PyTorch has native support for converting its models to ONNX. The documentation contains the list of supported operators.

There is no official support for importing the ONNX models to PyTorch — this topic is discussed in the ONNX import feature request in PyTorch Github issue tracker. The ToriML/onnx2pytorch and fumihwh/onnx-pytorch project projects work on adding ONNX import support to PyTorch. The former creates a nn.Module object from the ONNX file, while the latter generates Python code with model definition from the ONNX file.

2.2.3 MXNet

- Homepage: https://mxnet.apache.org
- License: Apache-2.0
- Repository: https://github.com/apache/incubator-mxnet
- Documentation: MXNet documentation
- Analyzed release: 1.8.0
- Supported languages:
  - C++
  - Clojure
  - Java
  - Julia
  - Perl
  - Python
  - R
  - Scala

2.2.3.1 Description

Apache MXNet is a deep learning framework designed for both efficiency and flexibility. It allows mixing symbolic and imperative programming — MXNet contains a dynamic dependency scheduler that automatically parallelizes both symbolic and imperative operations on the fly.

Similarly to PyTorch, MXNet provides a NumPy-like programming interface. As TensorFlow and PyTorch, MXNet is supported in various deep learning projects for model deployment and optimization, like TVM, TensorRT or OpenVINO.

2.2.3.2 MXNet children frameworks and enhancements

2.2.3.2.1 GluonCV

Gluon CV Toolkit provides an impressive and up-to-date collection of the state-of-the-art deep learning models for computer vision tasks, like classification, object detection, segmentation, pose estimation, action recognition and depth prediction. It also provides ready to use models for PyTorch.

2.2.3.2.2 GluonNLP

Gluon NLP Toolkit provides methods for loading and processing text data and training NLP models. It also provides models' galleries for NLP problems, like text generation, machine translation, and question answering.

2.2.3.2.3 MXBoard

MXBoard provides an API for visualizing MXNet data in TensorBoard.

2.2.3.3 Other deep learning frameworks

Other deep learning frameworks that are worth mentioning are listed below:

Darknet framework is a C/C++-based deep learning framework, in which next releases of state-of-the-art object detection algorithms, called YOLO were created. It provides heavy CUDA/CUDNN optimizations, allowing it to run those architectures in FP32, FP16 and even INT8 precision.

Chainer is a Python-based solution using CuPy to communicate between Python and CUDA/CUDNN.

CNTK is a Microsoft Cognitive Toolkit framework, currently discontinued.

DL4J is a deep learning framework for Java.

2.3 Deep Learning Compilers

This Chapter lists selected Deep Learning Compilers that convert the Deep Learning applications to optimized runtimes for Deep Learning Targets.

2.3.1 Apache TVM

2.3.1.1 General information

- Homepage: https://tvm.apache.org/
- License: Apache-2.0
- Repository: https://github.com/apache/tvm
- Documentation: TVM documentation
- Analyzed release: 0.7.0

2.3.1.2 Description

Tensor Virtual Machine (TVM) is a hierarchical multi-tier compiler stack and runtime system for deep learning models [1]. The aim of the TVM project is to provide a minimum deployable module for a diverse list of targets. It supports exporting models from:

- TensorFlow,
- PyTorch,
- MXNet,
- ONNX,
- Keras,
- CoreML,
- Caffe2,
- Darknet.

For each of the above frameworks there is an import frontend that converts the model to the Intermediate Representation (IR) in Relay.

2.3.1.2.1 Relay Intermediate Representation language

As described in [1] machine learning, and especially deep learning in various deep learning frameworks are represented in a form of graphs. The model representations present in the Deep Learning Frameworks are usually graph-based, very domain-specific, and lacking higher-level language features, like functions, recursions, or control flow.

Relay is a purely functional, statistically-typed programming language for differentiable computations expressing machine learning models [1]. It is the second generation of the NNVM compiler framework, which was used in MXNet. The model is represented as a set of functions in an imperative manner rather than a graph. Apart from functions representing typical machine learning operations, Relay also provides other features, like flow control, let bindings, recursion or scopes.

Models from various frameworks, where the representation of architecture vary heavily, are converted to this intermediate representation. Relay representation of the model is later

used during model optimization and compilation to the optimized hardware implementation.

2.3.1.2.2 Compilation flow

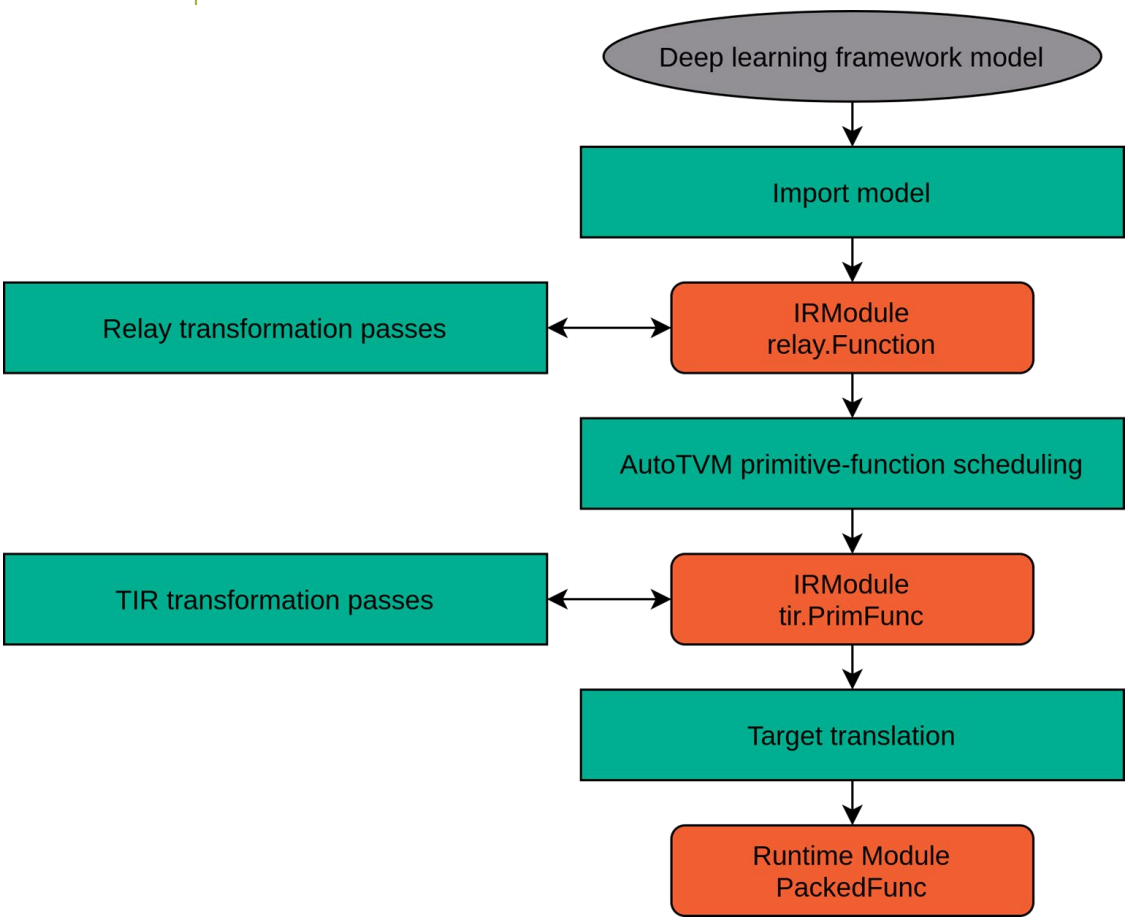

Figure 3: The TVM compilation flow

Figure 3 shows the compilation flow of the TVM framework, based on [2]

First, the machine learning model is converted to the IRModule using one of the import frontends. The created IRModule consists of Relay functions relay.Function that describe the computations in the model at a high level.

Secondly, the IRModule is subjected to transformation passes. There are three types of transformations:

- Relay transformation passes — they replace one or many operations in the IRModule with the optimized, functionally equivalent relay.Function. Those transformations usually involve constant folding, dead-code elimination, tensor layout transformation, fusions (i.e., creating conv2d-relu optimized blocks).

  Note: The transformed IRModule can be approximately equivalent to the original IRModule, for example if weights quantizations are applied [30].

- Lowering, also called TIR (Tensor Intermediate Representation) transformation passes — they replace the relay.Function high-level functions with the target-specific conversions and implementations. The lowering optimizations perform such operations, as flattening, target-specific function replacements or wrapping. Also,

some general transformations, similar to Relay transformation passes, like dead-code elimination at a lower level, are applied.

- The lowering does not perform low-level optimizations that can be handled by the compilers, like LLVM or NVCC. The lowering transformations convert the relay.Function objects to tir.PrimFunc objects.
- Search-space and learning-based transformations — such kind of transformations are used during model fine-tuning on a target device. The target-specific optimizations may consist of several hyperparameters that can be tuned empirically on device to achieve the best performance.
- For example, in the case of CUDA there are parameters for tiles dimensionality, cooperative fetching that can be adjusted to maximize data locality and optimize memory accesses. In CPU targets tile factor, vectorization, unrolling optimizations and other techniques are applied with different parameters to increase cache hit rate of memory access and to encourage the compiler (LLVM) to utilize SIMD instructions to greatly improve performance.
- After defining the search space consisting of compilation and optimization parameters, the cost function is minimized using such algorithms as XGBoost or Genetic Algorithms to find the good solution efficiently.

The Relay and lowering transformation passes are applied based on rules — they are deterministic. The transformations can be either applied globally (Module passes), or they can affect particular functions, based on defined rules (Function passes). The search-space and learning-based transformations are based on benchmarks on target hardware.

After Relay transformation passes and lowering the IRModule consists of tir.PrimFunc objects that contain elements including loop-nest choices, multi-dimensional load and store operations, threading, and tensor instructions [2].

The final step in the compilation process is the translation of the IRModule contents to the target-specific executable format that is possibly lightweight and minimal. The generated Module consists of PackedFunc functions that define the work of the model. Those functions interface with the accelerators via Device API. The Device API provides functions for activating the device (SetDevice), allocating and freeing data space (AllocDataSpace, FreeDataSpace), copying data (CopyDataFromTo) from target host to target device, and other. In this scenario, the target host can be the CPU, and the target device can be CUDA device, or TPU.

2.3.1.2.3 Running the model

The model is compiled to library files (.so, .o object files) that can be loaded and run using the TVM Runtime. The TVM Runtime can be accessed using Java, JavaScript, Python or C++.

2.3.1.3 Features

- High-level optimizations, like operator fusion and layout change,
- Memory reuse at the graph and operators level,
- Tensorized computations,
- Latency hiding,
- On-target fine-tuning using RPC server,
- Customizable transformation passes,
- Customizable translators,

- Weights' quantization,
- Large support of input model types — TensorFlow, Keras, PyTorch, MXNet, ONNX, darknet and other,
- Can cooperate with TensorFlow Lite to compile models for Edge TPUs.

2.3.1.4 Supported targets and acceleration libraries

- ARM Mali GPU (also Bifrost architecture)
- ARM Ethos-N
- CPUs supported by LLVM
- CUDA, CUDNN, CUBLAS, TensorRT-GPUs, Jetson platforms,
- FPGA using VTA accelerator
- Google Coral
- Hexagon
- Intel Graphics
- Microcontrollers (using microTVM)
- OpenCL
- ROCM
- SDAccel
- Intel FPGA SDK for OpenCL (AOCL)
- Metal runtime library
- Vulkan
- OpenGL
- NNPack
- TensorFlow Lite-TPU
- VITIS-AI

2.3.2 TensorFlow Lite

2.3.2.1 General information

- Homepage: https://www.tensorflow.org/lite
- License: Apache-2.0
- Repository: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite
- Documentation: TensorFlow Lite documentation
- Analyzed release: 2.5.0

2.3.2.2 Description

TensorFlow Lite is a set of tools that enables on-device inference of machine learning models on mobile, embedded and IoT devices. It provides:

- TFLite Model Converter for TensorFlow models (from Keras HDF5 format, concrete functions and saved model) that converts them to TFLite FlatBuffers,
- TFLite Interpreter and inference, which runs the TensorFlow Lite model on-device.

8It allows only converting TensorFlow models, so in order to use models from other frameworks, the conversion to ONNX, and later to TensorFlow using onnx-tensorflow described in TensorFlow Model Optimization Toolkit would be necessary.

2.3.2.2.1 TFLite FlatBuffers

FlatBuffers is an efficient cross platform serialization library that can be used in C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust and Swift [3]. The reasons FlatBuffers are used in TensorFlow Lite are:

- Access to serialized data without any parsing or unpacking — unlike ProtoBuffers present in TensorFlow, data in FlatBuffers can be accessed directly without unpacking data or parsing it to some intermediate representation.
- High memory efficiency — after loading a TFLite file, all the data is available without any additional allocations.
- High speed — the FlatBuffers access and manipulation performance is close to raw C/C++ structures.
- Flexibility — FlatBuffers allow adding optional fields, which makes them easy to adapt and open to new additions. It also eases backward compatibility for the TFLite models.
- Strong typing — strong typing allows detecting any type inconsistencies at compile time, removing the need for type and consistency checks
- Tiny code footprint, high portability — the generated code is minimal, there are no additional dependencies required for the FlatBuffers library, and it easily supports cross platform building.

The above features make FlatBuffers an excellent choice for low-power devices with memory and computational power limitations.

2.3.2.2.2 TFLite Model Converter

Apart from changing model representation to TFLite FlatBuffers, TFLite model converter provides algorithms for optimizing the model and adapting it to target accelerators.

The TFLite model converter can perform following optimizations [4]:

- Quantization — it provides methods for post-training FP16, dynamic range, 8-bit and 16-bit integer quantizations. TFLite performs quantization using calibration dataset. To reduce the prediction qualities even further, TensorFlow and TensorFlow Lite allow performing quantization-aware training. In quantization-aware training, the model is quantized in forward passes, the loss of the quantized model is computed and added to overall loss. This leads to creating a model that is more robust to the final quantization.
- Pruning — TFLite and TensorFlow provide an algorithm for magnitude-based weights pruning. This creates sparse deep learning models — in such models some of the connections are removed (meaning the weights of those connections are zeroed).
- Clustering [HMD16] — clustering reduces the number of unique weight values in a model. For each layer, the K-Means algorithm is used to subdivide weight values into K clusters. For each cluster, the centroid is computed and treated as a weight value for all weights present in a cluster. The actual weight values are replaced with centroid ID, so each layer in the model consists of K floating-point centroids, and kernel matrices with i.e., 2-bit centroid ID values.

Depending on the accelerator type, one or more of the above-mentioned optimizations can be applied to the model to significantly reduce model size, and processing time.

2.3.2.2.3 TFLite Interpreter and inference

The TFLite library first needs to load the model from the .tflite file and pass it through the Interpreter. The Interpreter allocates tensors and prepares the model for inference.

By default, TFLite uses CPU kernels that are optimized for the ARM Neon instruction set [32]. If the target device has an accelerator for some of the deep learning operations, like matrix multiplications, those computations can be delegated to the accelerator using Delegates.

Delegates enable hardware acceleration of TensorFlow Lite models by leveraging on-device accelerators [32], such as:

- GPU,
- TPU,
- Qualcomm Hexagon DSP,
- NPU.

During the compilation process, the delegates need to be registered so the Interpreter can use them to form a final TFLite computational graph. If there is a part of the graph (single operation, or multiple operations) that can be handled by the accelerator, and the format of input data and output data of such a graph block is compliant with the rest of the graph, then the TFLite replaces this part of the graph with the appropriate delegate implementation.

Depending on the operator's support in the delegate, the whole graph's execution can be moved to the accelerator, or it can run some unsupported parts on the CPU.

TFLite, apart from graph construction and delegate support, provides an API for pre-processing inputs and post-processing outputs. Some of the accelerators, like TPUs present in Google Coral, support only INT8 inference. TFLite provides functions for converting the inputs and outputs.

2.3.2.3 Features

- Very small binary size — ~1MB with all supported operators (32-bit ARM builds), and less than 300kB when using only operators required by the common image classifiers [32],
- Small model size, thanks to TFLite FlatBuffers,
- Weights quantization,
- Connections pruning,
- Clustering,
- Customizable delegates.

2.3.2.4 Supported targets

- ARM CPUs
- DSPs
- GPUs (via OpenCL and OpenGL ES)
- Hexagon
- Microcontrollers
- NPUs
- TPUs

2.3.3 OpenVINO

2.3.3.1 General information

- Homepage: https://docs.openvinotoolkit.org/latest/index.html
- License: Apache-2.0
- Repository: https://github.com/openvinotoolkit/openvino

- Documentation: OpenVINO documentation
- Analyzed release: 2021.3

2.3.3.2 Description

OpenVINO provides a necessary toolset and optimized runtime dedicated to the efficient execution of neural networks on Intel CPUs, FPGAs and specialized accelerators [5]

It supports the following formats:

- Caffe,
- TensorFlow,
- MXNet,
- ONNX.

OpenVINO performs the following model optimizations:

- 8-bit quantizations,
- convolutions conversion to Winograd-compliant format for faster inference on CPUs with Vector Extensions 512 (AVX-512) instruction set.

2.3.3.3 Supported targets

- Intel CPUs
- Intel Integrated Graphics,
- Intel Arria 10 FPGA GX
- Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- VPUs — Intel Movidius Myriad X VPUs and Intel Neural Compute Stick 2

2.3.4 ONNX Runtime

2.3.4.1 General information

- Homepage: https://www.onnxruntime.ai/
- License: MIT
- Repository: https://github.com/microsoft/onnxruntime
- Documentation: ONNX Runtime documentation
- Analyzed release: v1.8.0

2.3.4.2 Description

ONNX Runtime is a cross-platform inference and training machine-learning framework. It can be used to accelerate both inference and learning on target hardware.

The aim of the project is to allow running any ONNX model on a given platform.

On its own, ONNX Runtime does not introduce many new optimizations in comparison to other compilers — after loading the model to the memory it performs:

- constant folding,
- optional cast transformations between float32 and float16,
- redundant node eliminations,
- node fusions,
- data layout optimizations — conversion from NCHW to NCHWc layout for better vectorization and cache reuse.

Beside above-mentioned optimizations, ONNX Runtime performs graph partitioning. Given the available deep learning accelerators, ONNX Runtime partitions the graph into the

largest possible subgraphs that can be executed on available accelerators. This means that ONNX Runtime will try to accelerate as many parts of the model as possible.

In order to support completeness of the Runtime, the ONNX Runtime has default CPU and CUDA execution providers. In case the available accelerators do not support some parts of the model graph, the Runtime will fall back to those default providers.

2.3.4.3 Features

- Full support for any ONNX model,
- Basic model optimizations,
- Multiple threads can invoke inference on the same inference session object — all kernels are constant and stateless [dev21].
- Support for various execution providers — libraries and accelerators.

2.3.4.4 Supported targets and acceleration libraries

- CUDA, CUDNN, TensorRT-GPUs, Jetson platforms
- OpenVINO
- Apache TVM — used as NUPHAR execution provider
- NNAPI
- Vitis AI
- Direct ML
- Intel oneDNN

2.3.5 ONNC

2.3.5.1 General information

- Homepage: https://onnc.ai/
- License: BSD-3
- Repository: https://github.com/ONNC/onnc
- Documentation: ONNC documentation
- Analyzed release: 1.3.0

2.3.5.2 Description

ONNC (Open Neural Network Compiler) is a collection of open source, modular, reusable compiler algorithms and toolchains targeted on deep learning accelerators — it translates ONNX models to the proprietary DLA code [6]. ONNC has its own intermediate representation design that supports both fine-grained operators such as multiplier-accumulator and coarse-grained operators such as convolution, which simplifies the efforts in writing conversions for new DLAs. ONNC has an interactive compiler that is able to retry from the intermediate pass automatically upon compilation failures [6]. It also has a flexible pass manager that allows adding new optimization passes easily.

ONNC is integrated with the LLVM bit code runtime and backend. Any accelerator that already has the LLVM support can be integrated seamlessly with the ONNC [6].

ONNC aims towards providing easy to adapt deep learning compilers that can be used for DLAs with fine- and coarse-grained operators.

ONNC supports most of the ONNX operators.

ONNC is also the first open-source deep learning compiler supporting NVDLA [7]It creates an NVDLA loadable that was formerly generated only by the proprietary TensorRT library.

The loadable generation was documented based on the TensorRT library — the ONNC is said to support more deep learning models than the proprietary alternative [7].

2.3.5.3 Supported targets

- NVDLA
- Sophon BM168X

2.3.6 Glow

2.3.6.1 General information

- Homepage: https://ai.facebook.com/tools/glow/
- License: Apache-2.0
- Repository: https://github.com/pytorch/glow
- Documentation: Glow documentation
- Analyzed release: not versioned

2.3.6.2 Description

Glow stands for Graph-Lowering and is a deep learning compiler created within the PyTorch environment.

2.3.6.2.1 Compilation flow

As described in [8] Glow introduces two levels of Intermediate Representations (IR) — High-level IR and Low-level IR.

High-level IR is a strongly typed graph that uses high-level operations to describe the computations in the model, i.e., convolutions, pooling and activation layers. It consists of:

- constants — values that are constant during model inference, i.e., weights and biases,
- placeholders — symbolic nodes that are not backed by a concrete tensor at the compilation time, i.e., input and output tensors with an unknown shape,
- functions — nodes representing high-level deep learning operations, such as convolutions, batch normalization, pooling or activation functions,
- predicates — nodes representing functions that can be disabled according to the boolean flag (it can be used to avoid some unnecessary computations, especially in Recurrent Neural Networks, where the inputs vary in length),

Glow applies basic transformations like operator node replacements or unnecessary node removal. In addition, Glow can perform optimizations on constant nodes, like quantization, transposition and other adaptations.

After performing some high-level transformations, Glow performs node lowering. Node lowering breaks the high-level operators into low-level linear algebra operator nodes [8], such as matrix multiplication and broadcasted add instead of fully connected layer. All computations in the graph that are not necessary for the inference case (i.e., gradient computations) are removed from the graph. At the end of the node lowering process, the graph consists of very simple operations that are subjected to additional optimizations, both target independent and target specific.

Once node lowering is finished, Glow performs the IRGen (IR generation) step converting the graph to low-level IR. Low-level IR is an instruction-based representation that operates on tensors referenced by address [8]. This allows the compiler to perform low-level memory optimizations (both hardware-independent and hardware-specific) that are not possible at the higher level of model abstraction, like:

- in-place element-wise arithmetic,
- asynchronous DMA operations.

Low-level IR optimizations hide the latency of memory operations to help utilize the execution units of the hardware effectively. They also minimize the overall memory usage.

In the end, low-level operations are compiled using appropriate target backend. Thanks to converting the computation graph to the very low-level representation consisting of basic calculations, it is possible to easily compile the model for a given target without implementing many compute kernels representing high-level deep learning functions.

2.3.6.2.2 Model optimizations

The model optimizations in Glow can be divided into [8]:

- Graph-level optimizations:
  - "dead" code elimination
  - node transposing and sinking of transpose — this transposes both tensors and function calculations to minimize the amount of matrix transpositions throughout the graph,
  - regression nodes simplification,
  - pooling and post-pooling operations optimizations,
  - converting multiple concatenation nodes into single concatenation node.
- Profile-guided quantization to INT8 format.
- IR-level optimizations:
  - peephole optimizations — small, local optimizations that replace sequences of instructions with more efficient instruction of sequence of instructions,
  - dead store elimination — removes stores into weights or allocations if these stores are not going to be used,
  - allocation sinking — moves buffer allocations right before the first use of the buffer to reduce the lifetime of the buffer and improve memory consumption,
  - data-parallel operations stacking — replaces the sequential implementation of kernel operations to parallel implementation wherever possible,
  - other minor optimizations.

2.3.6.2.3 Running the model

Glow provides a runtime that is capable of partitioning models, queuing requests and executing models across multiple devices [8]. Glow runtime consists of:

- partitioner — responsible for splitting the network into sub-networks that can be run on multiple devices. The network is divided into multiple devices based on memory constraints, estimated time cost and communication cost between devices. This allows Glow to minimize the inference time.
- provisioner — responsible for assigning sub-graphs to specific devices and calling backend and Device Manager to compile and load each subgraph to the appropriate device.
- device manager — handles network loading, memory transfers, execution on devices and tracks hardware state.
- executor — handles the execution of a network, tracks execution state and propagates inputs and outputs of sub-graphs. It asynchronously handles inference requests and returns the collated results.

2.3.6.3 Features

- Model representation lowering to very simple operations, easy for adaptation to new hardware (no need to support lots of deep learning operations, only basic linear algebra),
- Memory-wise optimizations,
- Latency hiding,
- Weights' quantization,
- Multi-accelerator inference.

2.3.6.4 Supported targets

- OpenCL
- ARM
- Intel CPUs

2.4 ONNX Compatibility

Open Neural Network Exchange is an open standard for representing machine learning models. It is actively developed, and more and more deep learning operators are supported with next operator releases. The ONNX format is backward compatible, meaning the models exported to earlier releases of Operator sets (opsets) should be supported by new ONNX releases.

Most of the current deep learning frameworks support exporting its models to the ONNX representation and importing models from the ONNX representation.

This format is widely used for transferring the model from one framework to another. It is also used by deep learning compilers to read models from deep learning frameworks that are not natively supported.

2.4.1 General information

- Homepage: https://onnx.ai/
- License: Apache-2.0
- Repository: https://github.com/onnx/onnx
- Documentation: ONNX documentation
- Analyzed release: 1.9.0

2.4.2 ONNX conversion support grid

Model Name	mxnet (ver. 1.8.0)	pytorch (ver. 1.8.1+cu111)	tensorflow (ver. 2.4.1)
DenseNet201	supported / supported	supported / unsupported	supported / supported
MobileNetV2	supported / supported	supported / unsupported	supported / supported
ResNet50	supported / supported	supported / unsupported	supported / supported
VGG16	supported / supported	supported / unsupported	supported / supported
DeepLabV3 ResNet50	ERROR / unverified	supported / unsupported	Not provided / Not provided
Faster R-CNN ResNet50 FPN	ERROR / unverified	supported / unsupported	Not provided / Not provided

Mask R-CNN	ERROR / unverified	supported / unsupported	Not provided / Not provided

While ONNX is actively developed so the newest state-of-the-art deep learning models can be supported by this format, the importing/exporting routines of deep learning frameworks may have a different support for various operations.

Table 1 shows the ONNX conversion support for some of the popular models across various deep learning frameworks. Each row represents a different deep learning model. Each column represents a different deep learning framework. Each cell shows export to ONNX and import from ONNX support for a given model and framework.

First of all, the model is downloaded for a given framework. Secondly, the model is converted to ONNX. In the end, the ONNX model is converted back to the framework's format.

The values in cells are in <export support> / <import support> format.

The possible values are:

- supported if export or import succeeded,
- unsupported if export or import is not implemented for a given framework,
- ERROR if export or import ended up with error for a given framework,
- unverified if import could not be tested due to lack of support for export or error during export.
- Not provided if the model was not provided for the framework.

Currently, MXNet, PyTorch and TensorFlow support the most popular deep learning models for image classification problem. While TensorFlow and MXNet support both import and export of ONNX models, the PyTorch currently allows only model exporting, which disallows using models developed in other frameworks to be used and altered in PyTorch.

The object detection and instance segmentation models pose more problems to the exporting routines. While PyTorch is able to export the models, the MXNet fails with unsupported operators.

2.5 Deep Learning Models

2.5.1 VEDLIoT Model Zoo

Throughout this project, we focus on 9 state-of-the-art (SOTA) and commonly used models (17 model configurations in total), which make up the VEDLIoT model set. VEDLIoT model set was created to have a common set of models across partners for obtaining comparable benchmarks and for development.

In the model set, we span a wide range of different domains including image classification, object detection, instance segmentation, semantic segmentation and natural language processing (NLP). Furthermore, we cover a broad spectrum of computational requirements, such as models suited for mobile devices and models that are too demanding for edge devices. Over the course of this project, effort will be put into compressing the larger models to fit on the edge.

There exist benchmark suites such as MLPerf [9], aimed to create comparable benchmarks across models and hardware platforms. We opt to use the model set rather than MLPerf mainly because we want to explore a larger set of models both with diverse workloads and applications pertaining to the use cases in WP7. Additionally, we want more control over the models that we experiment on.

For the model set, we mostly acquired the models from Tensorflow Keras applications API which contain several of the most popular deep neural networks. The remaining models were found on TFHub and publicly available github repositories. We then compiled the models in both onnx and tensorflow protobuf (.pb) versions.

2.5.1.1 Bert

Transformers have quickly risen in popularity in the field of natural language processing (NLP). The transformer introduces the self-attention layer, which has proven to be a key advance in NLP models. Bert is a special kind of transformer that consists of a stack of bidirectional self-attention encoders [10] (see Figure 5.1).

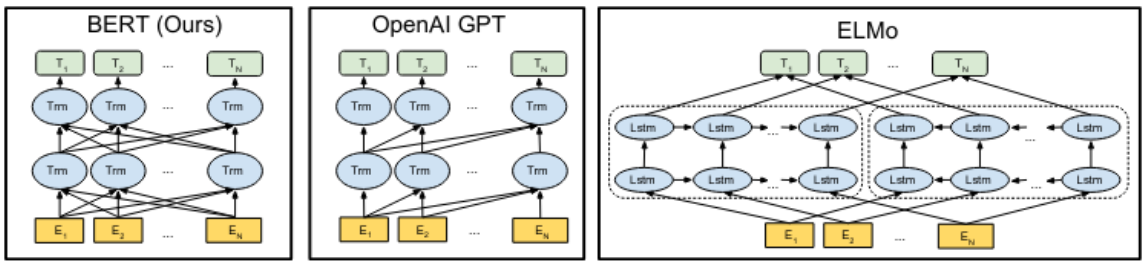

Figure 4 : Bert: (From [10]) An illustration of the difference between BERT, GPT and ELMo

2.5.1.2 ResNet

ResNet is one of the most well-known and used deep neural networks [11] for image classification. It popularized the use of skip connections, which made it possible to build much deeper models. They also note that when the model is expanded, the memory and computation can cause a problem. Thus, the bottleneck version of the residual block is introduced. The bottleneck residual block consists of a 1x1 convolution that decreases the number of filters (see Figure 5). This is then followed by the compute intensive 3x3 convolution and then another 1x1 convolution that increases the filter size back again before lastly applying the skip connection. This way the computation that is required is

decreased, because most of the computation is located in 3x3 convolution. We use three different configurations of this model, ResNet50, ResNet101 and ResNet152.

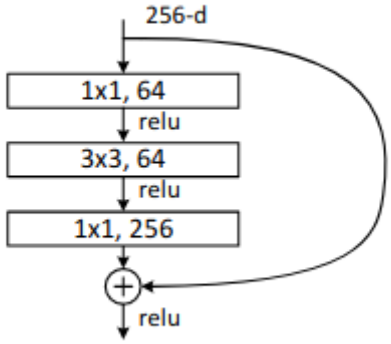

Figure 5: ResNet: An example of a residual bottleneck block introduced in [11] (Figure 5 from [11])

2.5.1.3 DeeplabV3+

DeepLabV3+ leverages atrous convolutions (see Figure 6) to increase the resolution in the context of pixelwise semantic segmentation while still keeping the same receptive field. It also incorporates a spatial pyramid pooling for increased multi scale context capture [12].

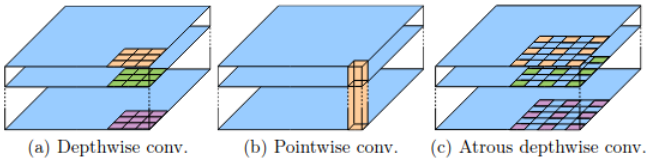

Figure 6: DeepLab: (From [12]) Subfigure (c) shows an atrous depthwise convolution

2.5.1.4 YoloV4

The Yolo model family is the most popular in the one-stage object detection category, predicting bounding boxes end-to-end without intermediate proposal steps. Improvements in later versions of Yolo compared to YoloV1 include the introduction of anchor boxes, the introduction of a *neck* that combines features from different layers of the backbone, and improved training. The YoloV4 network consists of a CSPDarkNet53 backbone with a SPP-PAN neck and YoloV3 head. The YoloV4 training introduces new augmentations such as the Mosaic augmentation, in which four different images are combined into one. [13]

2.5.1.5 Mask R-CNN

Mask R-CNN on the other hand is a two-stage detector [14]. First a backbone processes the image, then a Region of Interest pooling layer generates candidate regions for objects that the last part of the network makes the predictions on. Mask R-CNN covers not only object detection but also instance segmentation. While Mask R-CNN is considered very powerful in terms of accuracy, it is also notoriously slow due to the two-stage structure.

2.5.1.6 MobileNetV3

MobileNetV3 is a lightweight image classification model aiming to minimize the latency on mobile CPUs. The architecture uses the inverted residual bottleneck introduced in MobileNetV2, which aims to keep the dimensions in the skip connection minimized and instead expands the amount of filters before the convolution (see Figure 7).

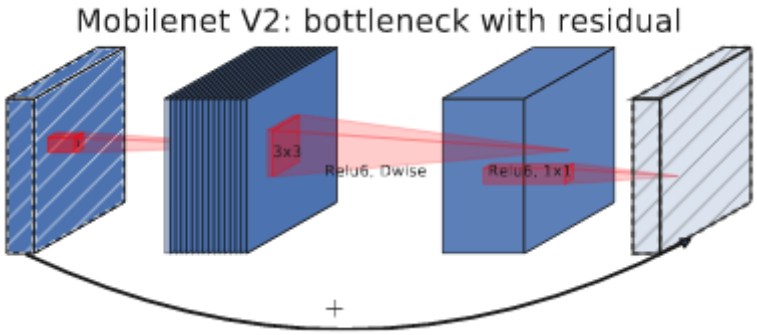

Figure 7: MobNet: Inverted residual bottleneck. (Figure 3 from [15])

It also uses operations such as depthwise separable convolutions, hard-swish activations and squeeze-and-excite modules to increase the performance per latency [15]. MobileNet architectures typically have a scaling factor that is set to trade off computation and accuracy. We keep it constant at 1.0 throughout this project.

2.5.1.7 EfficientNet

The base model (designed for image classification), EfficientNet-B0, is the product of a neural architecture search that optimizes for floating point operations and accuracy [16]. The model is very similar to a MobileNet with minor differences. It applies a compound scaling to the model width, depth and image size to increase the models' performance. The result is a family of models B0 to B7 ranging from 0.39 billion ops to 37 billion ops. We opt to use B0, B3 and B7 in order to cover the full range of computational demand.

2.5.1.8 EfficientDet

This is the object detection adaptation of EfficientNet. EfficientDet is composed of an EfficientNet with a novel bidirectional feature pyramid (see Figure 8), creating a new family that follows the same notation as EfficientNet, D0-D7. Here we use the subset D0, D3 and D7.

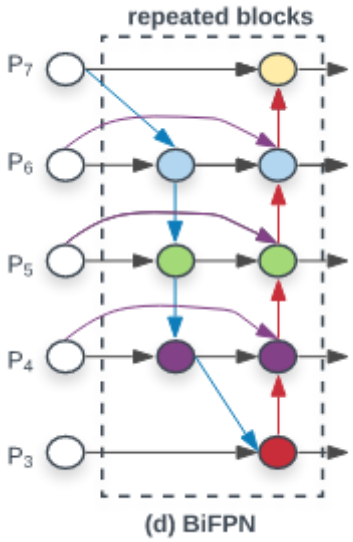

Figure 8: EDet: (from [17]) The bidirectional feature pyramid

2.5.1.9 ShuffleNetV2

The main contribution of ShuffleNetV2 is the redesign of the shufflenet block [18]. The old shufflenet block is yet another implementation on the residual block, but with some tricks to decrease computational complexity. Firstly, the 1x1 convolutions are grouped and the 3x3 convolution is replaced with a depthwise convolution. The consequence of this is that some filter channels are completely decoupled, which could be harmful to the accuracy. In order to address the decoupling, the paper describes a channel shuffle operation.

The new shufflenet block design uses channel split for more efficient data usage and later employs a channel shuffle after the split data paths are recombined. This is illustrated in Figure 5.6.

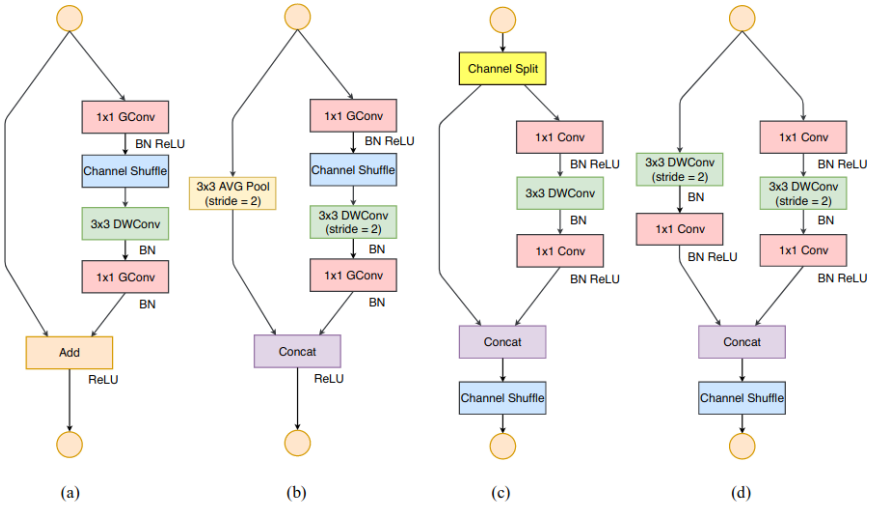

Figure 9: ShNet: (Figure 3 from [18]) (a) and (b) shows the ShuffleNetV1 block and (c) and (d) shows the ShuffleNetV2 block

3 Evaluation of DL Accelerators

3.1 Methodology

This report is the evaluation of ML accelerators in the VEDLIoT project, were all accelerators that could be acquired are going to be tested. For this evaluation, metrics are defined concerning performance, efficiency and accuracy. Furthermore, measurements are described regarding performance, power and accuracy evaluation.

3.1.1 Metrics

Within the scope of this project and considering both the hardware development and the use-cases requirements, the different partners have agreed on a set of metrics that is relevant for the evaluation of the different systems evaluated and developed within this project. The set of metrics is divided into four different categories: **system metrics** (platform and I/O); **performance metrics** (applications); **quality metrics**; and **efficiency metrics.** The different metrics, their definitions and units are presented in the tables below.

In addition to the metrics, we also include other definitions that are relevant for the evaluation such as: benchmark setup characterization, hardware platform characterization, and hardware platform versatility.

For this first evaluation of existing hardware we use a subset of these metrics for most of the hardware since certain metrics are specific to certain types of hardware (e.g., hardware resources which is mostly relevant to the FPGA-based accelerators to be developed in the future in the project). While there was an effort to have a set of metrics that could be used across most of the devices, in certain cases it is not possible to collect the values for the proposed metric. All exceptions are clearly justified in the evaluation section, when presenting the results for the different hardware devices.

System Metrics (Platform and I/O)	
Peak performance [GOPS]	Theoretical value determined by the analysis of the available hardware and operating frequency. This value is collected from the vendors as it is usually reported in the spec sheets for the different devices. This value is reported in Giga (10^9) Operations Per Second (GOPS). As before, an operation is a multiplication or an addition (a MAC instruction accounts for 2 operations).
Memory capacity [B]	Total internal (on-chip) memory capacity. Whenever possible, breakdown of the capacity for different internal memory buffers (e.g., activation or weight buffer). This value is reported in Bytes or its multiple KB (2^{10}), MB (2^{20}), GB (2^{30}).
Memory bandwith [MB/s]	Bandwidth for off-chip (and off-accelerator) memory accesses. This is the theoretical value reported by the vendors for the off-chip memory bandwidth, reported in Megabyte (2^{20}) per second (MB/s).

Memory latency [s]	Latency for off-chip (and off-accelerator) memory accesses. This is the theoretical value reported by the vendors for the off-chip memory latency, reported seconds (s) or one of its multiples ms (10^{-3}), μs (10^{-6}), ns (10^{-9}).
I/O bandwidth [bps]	Bandwidth for data transfers between the accelerator to the "outside" world for uploads and downloads (models, weights, configurations, etc.). This is the theoretical value reported by the vendors for the off-chip I/O bandwidth, reported in bit-per-second (bps).
I/O latency [s]	Latency for data transfers between the accelerator to the "outside" world for uploads and downloads (models, weights, configurations, etc.). This is the theoretical value reported by the vendors for the off-chip I/O latency, reported in second (s).
Idle power [W]	Power value measured after the system/accelerator has booted or it is on but not executing any inference at the time. The value is reported in Watt (W).
Reconfiguration time [s]	Dead time from starting the update process till system is running again. This value is reported in seconds and it is only relevant for reconfigurable accelerators (CGRA or FPGA).
Cost [$]	Mix of hardware costs and service costs (edge IoT); basic service costs (e.g., infrastructure) and AI service costs (e.g., consistency check). For the sake of comparison with other international works, this metric value is reported in US dollars ($).

Performance Metrics (Application)	
Inference time [s]	Inference time for the completion of the inference operation. This time does not include any data pre- or post-processing times. Also in same cases where multiple inferences are part of the same execution (batch > 1), this time accounts for the execution of all inferences in the request. A batch of 1 is used to report the lowest inference time while a batch>1 is used to report highest inference throughput.
Latency [s]	Total Inference Time for a request including both data pre- and post-processing times.
Number of operations [Ops]	Total number of Multiply and Accumulate operations required to determine an inference result. This number is determined by analysing the number of

	static MAC instructions in the DL models. A merged multiply-add (or multiply-accumulate) corresponds to two operations.
Number of instructions [Ins]	Total number of instructions required to determine an inference result. This metric is only valid for certain hardware devices and it is also not comparable between hardware devices that support different Instruction Set Architectures (ISAs).
Achieved performance [GOPS] [IPS]	In contrast to peak performance this is the performance that is achieved for the execution of a specific DL model. This value can be reported in GOPS (as described in the previous table) or/and in Inferences Per Second (IPS). In all cases that the input to the inference is an image, this value is the same as what is reported in other studies as Frames Per Second (FPS).
Performance ratio [%]	This value represents the ratio between the achieved performance and the peak performance as a percentage value. This metric is interesting to determine how effectively the hardware is being used for the inference operation.
Memory utilization [B]	This value represents the maximum memory utilized for the execution of the inference operation. This value is reported in Bytes (B) and its multiple KB, MB, GB, etc.
Memory requests	Total number of requests to main memory (requests that are not satisfied in the internal memory resources of the accelerator) that occur during the inference operation. Whenever possible, the number of misses that occur in the different internal memory buffers should be reported. This is not possible to collect for all hardware devices. It is important to note the size in Byte of each request.
Achieved memory bandwith [MB/s]	In contrast to the device's memory bandwidth this is the performance that is achieved for the execution of a specific DL model. This value is reported Megabyte (2^{20}) per second (MB/s).
Achieved I/O bandwidth [bps]	In contrast to the device's I/O bandwidth this is the performance that is achieved for the execution of a specific DL model. This value is reported in bits-per-second (bps).
Resources utilized [number of LUTs, registers, etc.]	This value represents the hardware resources utilized for the execution of the inference operation on a particular reconfigurable device (FPGA).
Power [W]	Average of sample measurements of the total power of the system measured doing the execution of the inference operation. This value is reported in Watt (W). Whenever possible the detailed power breakdown

	between different internal components (e.g., core and uncore). The power for the inference only can be obtained by deducting the "Idle power" from this Power value.
Inference energy [J]	Energy consumed to perform the inference operation. Note that if batch > 1 this will be the energy for all the inference requests to complete. This value is computed from the **Inference time** and the **Power** values and is reported in Joule (J).
Total energy [J]	Energy consumed to execute the complete request including the data pre- and post-processing in addition to the inference. Note that if batch > 1 this will include the energy for all the inference requests to complete. This value is computed from the **Latency** and the **Power** values and is reported in Joule (J).

Quality Metrics	
Accuracy	Definition is depending on the used algorithm. The relevant metric(s) need to be defined together with the together with the benchmark. (e.g., Classification Metrics: accuracy, precision, recall, F1-score, ...; Statistical Metrics: Correlation; ...) A minimum/target quality should be defined for each benchmark (we then need to define if/how to reward quality that is higher than the target)

Efficiency Metrics	
Power efficiency [GOPS/W]	Combined metric that measures the power efficiency of the execution of the inference operation on a particular architecture. This metric is a combination of the **Achieved performance** and **Power** and is reported in Giga Operations Per Second (GOPS) per Watt (W).
Area efficiency [GOPS/mm^2]	Combined metric that reports how efficient is the area utilization for **Achieved performance** and is reported in Giga Operations Per Second (GOPS) per square millimetre (mm^2).
Cost efficiency [GOPS/$]	Combined metric that shows the efficiency of the hardware for a certain execution in terms of its **Achieved Performance** when considering its

	Cost. This value is reported in Giga Operations Per Second (GOPS) per US dollar ($).

Benchmark Setup Characterization	
Benchmark	Benchmark model, data set, and required characteristics used for the evaluation (cf. Table VEDLIoT Benchmark Models).
Toolchain	Toolchain used to obtain the measured results.
Numerical Format	Numerical format used for the benchmark implementation on the accelerator (e.g., FP32, BF16, INT8)

Hardware Platform Characterization	
Technology	Process for silicon implementations (e.g., TSMC 7nm); FPGA architecture for reconfigurable platforms (e.g., Xilinx Zynq UltraScale+ ZU9EG-1)
Accelerator Device	Used hardware accelerator SoC/ASIC
System Architecture	Architecture of the system used for benchmarking, e.g., COM (formfactor) / Evaluation board
System Integration	Integration of the accelerator into a system environment for benchmarking, e.g., PCIe card in the RECS system; evaluation board with Ethernet connection
Availability	Availability of the platform (samples/general availability)
Reliability/Dependability	Characterization of specific hardware platform support to enhance reliability or dependability

Hardware Platform Versatility	
Supported Toolchains	List of toolchains that are supported by the hardware accelerator (e.g., Caffe, MXNet, ONNX, Pytorch, TensorFlow/TF-Lite)
Supported Benchmarks	List of DL architectures that are supported by the hardware accelerator (e.g., ResNet, YOLOv4, …)

Supported Data Formats	Data formats supported by the hardware accelerator (e.g., FP32, BF, TF, INT8)

3.1.2 Performance Measurements

The performance measurements will be done with a subset of the Model Zoo presented in Chapter 5. Three relevant models were chosen and their corresponding operations per inference were calculated to determine the achieved performance. As a reference the theoretical maximum performance was investigated from either manufactures references or calculated from clock frequency and amount of execution unit. Finally for reproducibility the tool flow used for the measurements was noted.

3.1.2.1 Models

In the interest of time, the benchmarking has for the most part been focused on a subset of models. These models are ResNet50, MobileNetV3 Small and YoloV4. ResNet50 is a mid-sized model with basic operations that are expected to be supported on most hardware. MobileNetV3 targets smaller devices in terms of computational demand but introduces some more modern operations such as HardSwish [15] that might not be supported. Lastly, YoloV4 is not only more demanding in terms of computations, but it also utilizes the Mish activation [19] that might not be supported by hardware.

The Multiply-add operations are counted using a tool developed by EmbeDL and then converted to operations denoted as Ops (1 Mul-Adds = 2 Ops). These numbers are then verified by comparing to the numbers reported in the papers. The ResNet could be directly compared to the paper without modifications, but there is a discrepancy. The flops stated in the paper, corresponds to our measured Mul-Adds. For MobileNetV3 Small only the backbone part of the model is reported. This is verified by only measuring over the backbone, but the numbers reported in this table accounts for both backbone and classification head. YoloV4 also only reports backbone numbers and for other input sizes. In order to verify these numbers, we measured only the backbone and resized the input to the size used in the paper. The determined operations per inference a noted in Table 1.

Table 1: OPs and Multiply-Adds of model subset

Model	Operations / Inference
MobileNetV3 Small	0.178 GOps (89 million Mul-Adds)
ResNet50	7.78 GOps (3.89 billion Mul-Adds)
YoloV4	60.4 GOps (30.2 billion Mul-Adds)

3.1.2.2 Performance

In ML applications the performance isn't a direct metric. The return value of the inference tools is "Inference Time" which needs to be calculated to GOPS. Therefore, the operations per inference were calculated in the last paragraph to be multiplied with Inferences per second:

$$Inferences\ per\ second = \frac{1}{Inference\ Time}$$

$$Performance = \frac{1}{Inference\ Time} * Model\ Operations\ per\ Inference$$

As reference the theoretical maximum performance of each architecture was investigated. For some accelerators these values could be found on the manufacture's web presence or corresponding datasheets. In other cases the values needed to be calculated, e.g., x86 processors where the AVX2 unit process 32 FP32 operations per second. Multiplying this with the number of cores and the frequency, the theoretical maximum could be determent.

3.1.2.3 Toolflow

Each hardware is optimized for a certain toolflow, e.g., NVIDIA hardware works best with TensorRT or Intel hardware works best on OpenVINO. From a benchmark point of view it would be nice to have a single toolflow to be used with every hardware, but that would be an advantage for the one hardware optimized for this flow.

The approach for this evaluation was using the corresponding hardware-software combination and feet the models from the VEDLIoT Model Zoo. Unfortunately, it turned out, that some vendor specific software doesn't support 3rd party models. This leaded to using the models that are recommended from the vendor. In the end the accuracy for each DL toolchain was measured to ensure that the models are comparable. For IoT workloads batch size = 1 was examined for lowest latency. For Cloud workloads the batch size was tuned to get the maximum performance from the device, e.g., Intel Myriad batch size = 4 or NVIDIA Xavier NX batch size = 8.

3.1.3 Power Measurements

There are various ways to measure power and in VEDLIoT we need to have multiple possibilities due to the variety of hardware. Here we basically differ between manual operated measurements where the data needs to be read from an external meter and integrated measurements where the management system directly feeds the data to the application.

For integrated power measurements we want to introduce the perspective of integrating the RECS REST interface into the Kenning profiler for direct generation of statistics.

3.1.3.1 Manual

There are two basic ways to physically measure power. The first is the resistive way where a resistor is placed in series with the design under test (see Figure 10). The voltage over this resistor needs to be measured by an oscilloscope or a multimeter. The current can then be calculated by ohm's law:

$$\text{Current} = \frac{\text{Voltage}}{Resistance}$$

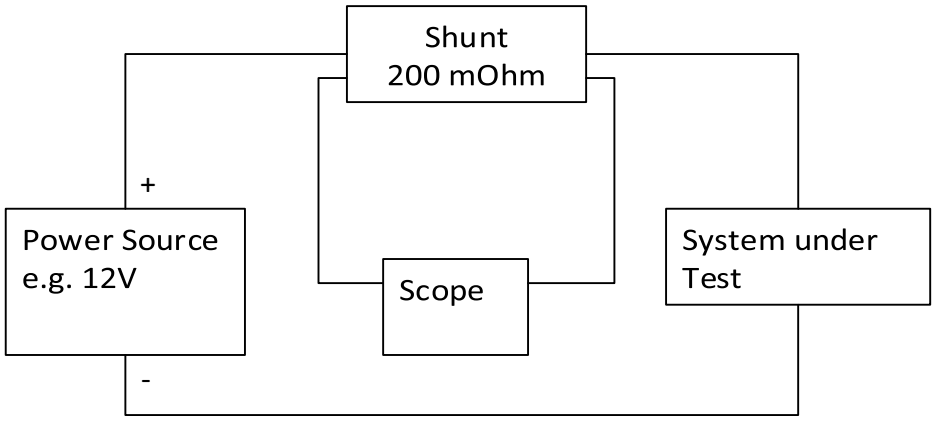

Figure 10: Resistive Measurement

For high-speed measurements an oscilloscope is required. The advantage of high-speed measurements is not only higher accuracy but also showing more details of the application running on the device. Figure 11 shows a shot during inference on one of the accelerators. On the X-axis each division is 400 ms, there is a small drop every ~550 ms which correlates with the inference time. In addition, the oscilloscope measures the mean voltages between the cursors given a very accurate power measurement.

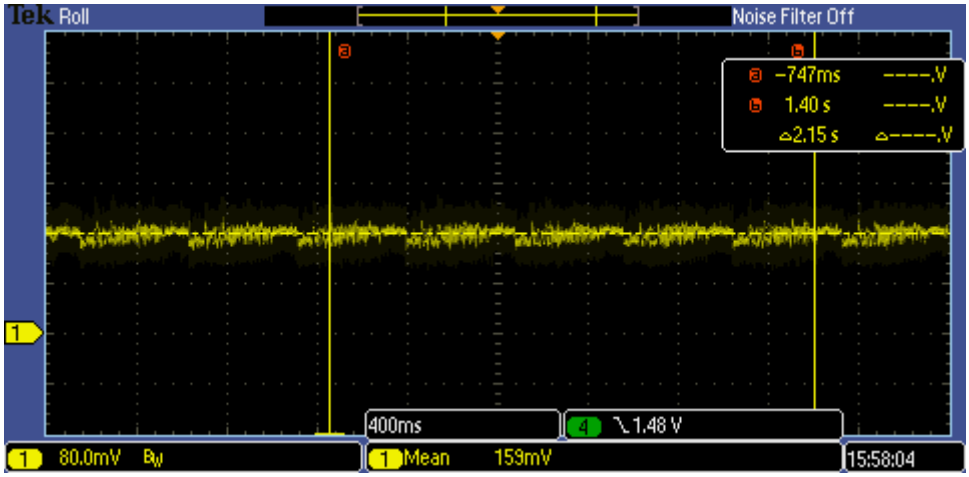

Figure 11: Oscilloscope based high-speed measurement of supply current

The second way to measure power is the inductive way. An inductive probe is clamped on the cable from the power source. This probe measures the electric field induced by the current flowing through the cable, it is also connected to an oscilloscope. The results are close to the resistive way using an oscilloscope, but the advantage is that it can be directly used with most of the evaluation boards without modifying the hardware.

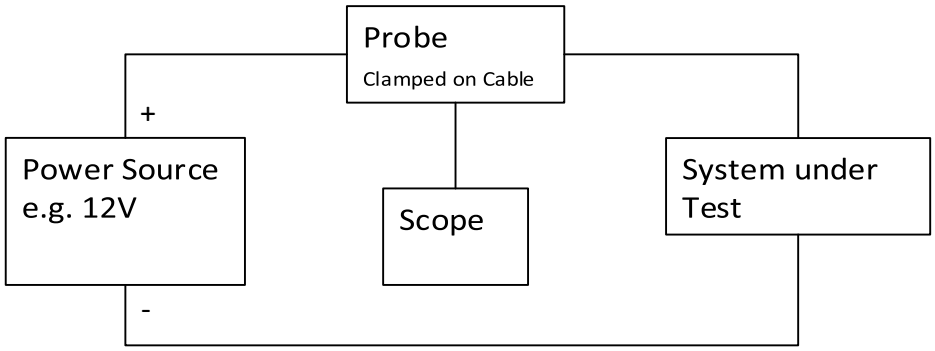

Figure 12: Inductive Measurements

3.1.3.2 Integrated

The main interface for managing the RECS server is the WebGUI that can be used in a regular web browser. It allows navigation to every Microserver in the system. The management allows remote control over on/off/reset, serial console, KVM etc. and monitoring of certain important sensor values can be accessed.

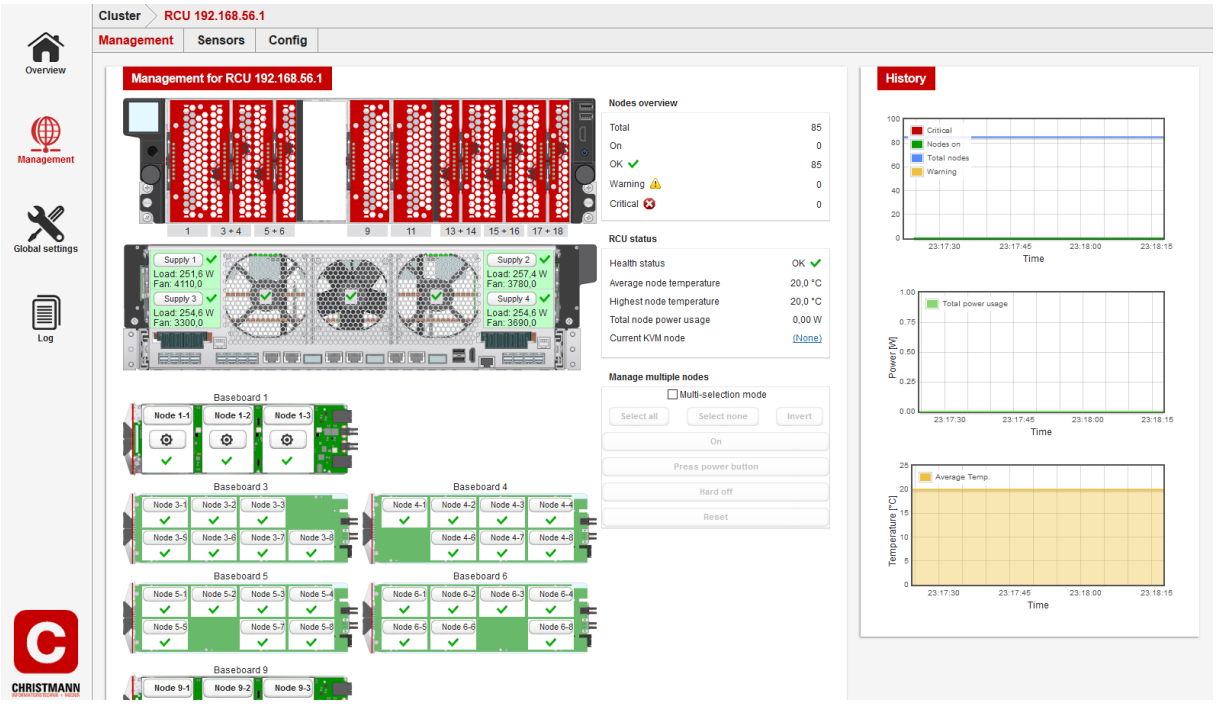

Figure 13: Node selection screen of the RECS_Master WebGUI

The primary task of the WebGUI is to monitor system health. A low-speed power measurement with an interval of one per second is available directly as graph in the GUI or remotely via REST interface.

One of the possible future improvements of the test system is enabling the Kenning framework to deliver the data in a form required by the WebGUI. To do so, a few extensions have to be implemented in the Kenning framework:

- Addition of new readouts from sensors regarding performance in the threads responsible for measurements collection
- Addition of a data-provider capable of reading the sensors' data for the model inference
- Addition of a backend presenting the data in a REST format required by the WebGUI

Kenning's modular nature makes the above changes relatively easy to integrate. Due to the fact that Kenning is an open-source framework, it can be easily adjusted to work with WebGUI.

3.1.4 Accuracy

The three models, which were used for the evaluation, are initially trained on either the ImageNet dataset (*ResNet50* and *MobileNetV3small*) or the COCO dataset (*YoloV4*). Furthermore, while *ResNet50* and *MobileNetV3small* are classification networks, *YoloV4* is an object detector. Therefore, for the accuracy evaluation, we employed two different metrics. We calculated the Top1 and Top5 accuracy for the classifiers, while for *YoloV4*, the mean average precision (mAP) was determined using the COCO-API.

3.1.4.1 Classifier Accuracy

The classifier networks that were used for the measurements were trained on the 1000-classes of the ImageNet dataset [20]. Therefore, the output of each inference is a list containing the probability of the respective class. We used the IDs of the five classes with the highest probability for our accuracy measurements. The original ImageNet 2012 validation dataset, consisting of 50.000 images, was used to evaluate the networks properly. The accuracy was then calculated using the ground truth IDs for each image, provided together with the dataset and the network outputs. For each image, it was first checked if the five IDs the network calculated contained the respective ground truth ID; if so, the *top5* counter was incremented and checked if the first ID, i.e., the one with the highest probability, matches the ID, in that case, the *top1* counter was also incremented. This was done for each of the 50.000 images, and at the end, the accuracies were calculated as follows (where img_{cnt} is 50.000):

$$top1\% = \frac{top1_{cnt}}{img_{cnt}} * 100$$

$$top5\% = \frac{top5_{cnt}}{img_{cnt}} * 100$$

3.1.4.2 Object Detection Accuracy

The *YoloV4* model derived from the original darknet model, which was trained on the 80-classes of the COCO 2017 dataset [21] was used. The output of an inference consists of numerous detections found in the input image. Each detection consists of the following elements: the class ID of the object detected, the network's confidence that this detection is correct, and the object's bounding box in the form of x-, y-coordinates, width and height. For each image in the validation dataset, this information is stored together with the name of the input image. The COCO 2017 validation dataset was used consisting of 5.000 images to determine the accuracy. After all images were processed, the output containing all detections was processed using the COCO-API [22], provided alongside the dataset, to calculate the mean average precision (mAP) based on the COCO 2017 ground truth

annotations. Although the COCO-API calculates several mAP values, we only report $\boldsymbol{mAP^{IoU(.50)}}$ and $\boldsymbol{mAP^{IoU(.50:.95)}}$ here. IoU stands for *Intersection over Union* and indicates the overlap of two bounding boxes, i.e., the overlap between the bounding box of the detection and the ground truth. In this context, $mAP^{IoU(.50)}$ means that detections with an $IoU \geq 0.5$ are considered as correct. On the other hand, $mAP^{IoU(.50:.95)}$ is calculated as the average AP over a range of minimum *IoUs*, in this case from 0.5 to 0.95, with a step size of 0.05. Meaning that 10 AP values are calculated ($IoU \geq 0.5, IoU \geq 0.55, IoU \geq 0.6, \ldots, IoU \geq 0.95$), and $mAP^{IoU(.50:.95)}$ is then defined as the average over these 10 APs.

3.1.5 Kenning

Kenning is an open-source project developed by Antmicro for implementing and testing pipelines for deploying deep learning models on edge devices.

3.1.5.1 Deployment flow

Deploying deep learning models on edge devices usually involves the following steps:

- Preparation and analysis of the dataset, preparation of data pre-processing and output post-processing routines,
- Model training (usually transfer learning), if necessary,
- Evaluation and improvement of the model until its quality is satisfactory,
- Model optimization, usually hardware-specific optimizations (e.g., operator fusion, quantization, neuron-wise or connection-wise pruning),
- Model compilation to a given target,
- Model execution on a given target.

There are different frameworks for most of the above steps (training, optimization, compilation and runtime). The cooperation between those frameworks differs and may provide different results.

This framework introduces interfaces for those above-mentioned steps that can be implemented using specific deep learning frameworks.

Based on the implemented interfaces, the framework can measure the inference duration and quality on a given target. It also verifies the compatibility between various training, compilation and optimization frameworks.

3.1.5.2 Kenning structure

The kenning module consists of the following submodules:

- core — provides interface APIs for datasets, models, compilers, runtimes and runtime protocols,
- datasets — provides implementations for datasets,
- modelwrappers — provides implementations for models for various problems implemented in various frameworks,
- compilers — provides implementations for compilers of deep learning models,
- runtimes — provides implementations of runtime on target devices,
- runtimeprotocols — provides implementations for communication protocols between host and tested target,
- onnxconverters — provides ONNX conversions for a given framework along with a list of models to test the conversion on,
- resources — contains project's resources, like RST templates, or trained models,
- scenarios — contains executable scripts for running training, inference, benchmarks and other tests on target devices.

3.1.5.3 Model preparation

3.1.5.3.1 The kenning.core.dataset.Dataset classes

Classes that implement the methods from kenning.core.dataset.Dataset are responsible for:

- preparing the dataset, including the download routines (use --download-dataset flag to download the dataset data),
- pre-processing the inputs into the format expected by most of the models for a given task,
- post-processing the outputs for the evaluation process,
- evaluating a given model based on its predictions,
- subdividing the samples into training and validation datasets.

Based on the above methods, the Dataset class provides data to the model wrappers, compilers and runtimes to train and test the models.

The datasets are included in the kenning.datasets submodule. Check out the Pet Dataset wrapper for an example of Dataset class implementation.

3.1.5.3.2 The kenning.core.model.ModelWrapper classes

The ModelWrapper class requires implementing methods for:

- model preparation,

- model saving and loading,
- model saving to the ONNX format,
- model-specific pre-processing of inputs and post-processing of outputs, if necessary,
- model inference,
- providing metadata (framework name and version),
- model training,
- input format specification,
- conversion of model inputs and outputs to bytes for the kenning.core.runtimeprotocol.RuntimeProtocol objects.

The ModelWrapper provides methods for running the inference in a loop from data from the dataset and measures both the quality and inference performance of the model.

The kenning.modelwrappers.frameworks submodule contains framework-wise specifications of ModelWrapper class — they implement all methods that are common for all the models implemented in this framework.

For the Pet Dataset wrapper object there is an example classifier implemented in TensorFlow 2.x called TensorFlowPetDatasetMobileNetV2.

3.1.5.3.3 The kenning.core.compiler.ModelCompiler classes

Objects of this class implement compilation and optional hardware-specific optimization. For the latter, the ModelCompiler may require a dataset, for example to perform quantization or pruning.

The implementations for compiler wrappers are in kenning.compilers. For example, TFLiteCompiler class wraps the TensorFlow Lite routines for compiling the model to a specified target.

3.1.5.4 Model deployment and benchmarking on target devices

Benchmarks of compiled models are performed in a client-server manner, where the target device acts as a server that accepts the compiled model and waits for the input data to infer, and the host device sends the input data and waits for the outputs to evaluate the quality of models.

3.1.5.4.1 The general communication protocol

The communication protocol is message-based. There are:

- OK messages — indicate success, and may come with additional information,

- ERROR messages — indicate failure,
- DATA messages — provide input data for inference,
- MODEL messages — provide model to load for inference,
- PROCESS messages — request processing inputs delivered in DATA message,
- OUTPUT messages — request results of processing,
- STATS messages — request statistics from the target device.

The message types and enclosed data are encoded in format implemented in the kenning.core.runtimeprotocol.RuntimeProtocol-based class.

The communication during inference benchmark session is as follows:

- The client (host) connects to the server (target),
- The client sends the MODEL request along with the compiled model,
- The server loads the model from request, prepares everything for running the model and sends the OK response,
- After receiving the OK response from the server, the client starts reading input samples from the dataset, preprocesses the inputs, and sends DATA request with the preprocessed input,
- Upon receiving the DATA request, the server stores the input for inference, and sends the OK message,
- Upon receiving confirmation, the client sends the PROCESS request,
- Just after receiving the PROCESS request, the server should send the OK message to confirm that it starts the inference, and just after finishing the inference the server should send another OK message to confirm that the inference is finished,
- After receiving the first OK message, the client starts measuring inference time until the second OK response is received,
- The client sends the OUTPUT request in order to receive the outputs from the server,
- The server sends the OK message along with the output data,
- The client parses the output and evaluates model performance,
- The client sends STATS request to obtain additional statistics (inference time, CPU/GPU/Memory utilization) from the server,
- If the server provides any statistics, it sends the OK message with the data,
- The same process applies to the rest of input samples.

The way of determining the message type and sending data between the server and the client depends on the implementation of the kenning.core.runtimeprotocol.RuntimeProtocol class. The implementation of running inference on the given target is implemented in the kenning.core.runtime.Runtime class.

3.1.5.4.2 The kenning.core.runtimeprotocol.RuntimeProtocol classes

The RuntimeProtocol class conducts the communication between the client (host) and the server (target).

The RuntimeProtocol class requires implementing methods for:

- initializing the server and the client (communication-wise),
- waiting for the incoming data,
- sending the data,
- receiving the data,
- uploading the model inputs to the server,
- uploading the model to the server,
- requesting the inference on target,
- downloading the outputs from the server,
- (optionally) downloading the statistics from the server (i.e., performance speed, CPU/GPU utilization, power consumption),
- notifying of success or failure by the server,
- parsing messages.

Based on the above-mentioned methods, the kenning.core.runtime.Runtime connects the host with the target.

Look at the TCP runtime protocol for an example.

3.1.5.4.3 The kenning.core.runtime.Runtime classes

The Runtime objects provide an API for the host and (optionally) the target device. If the target device does not support Python, the runtime needs to be implemented in a different language, and the host API needs to support it.

The client (host) side of the Runtime class utilizes the methods from Dataset, ModelWrapper and RuntimeProtocol classes to run inference on the target device. The server (target) side of the Runtime class requires implementing methods for:

- loading model delivered by the client,
- preparing inputs delivered by the client,
- running inference,
- preparing outputs to be delivered to the client,
- (optionally) sending inference statistics.

Look at the TVM runtime for an example.

3.1.5.5 ONNX conversion

Most of the frameworks for training, compiling and optimizing deep learning algorithms support ONNX format. It allows conversion of models from one representation to another.

The ONNX API and format is constantly evolving, and there are more and more operators in new state-of-the-art models that need to be supported.

The kenning.core.onnxconversion.ONNXConversion class provides an API for writing compatibility tests between ONNX and deep learning frameworks.

It requires implementing:

- method for importing ONNX model for a given framework,
- method for exporting ONNX model from a given framework,
- list of models implemented in a given framework, where each model will be exported to ONNX, and then imported back to the framework.

The ONNXConversion class implements a method for converting the models. It catches exceptions and any issues in the import/export methods, and provides the report on conversion status per model.

Look at the TensorFlowONNXConversion class for an example of API usage.

3.1.5.6 Running the benchmarks

All executable Python scripts are available in the kenning.scenarios submodule.

3.1.5.6.1 Running model training on host

The kenning.scenarios.model_training script is run as follows:

```
python -m kenning.scenarios.model_training \

kenning.modelwrappers.classification.tensorflow_pet_dataset.TensorFlowPetDatasetMob
ileNetV2 \
  kenning.datasets.pet_dataset.PetDataset \
  --logdir build/logs \
  --dataset-root build/pet-dataset \
  --model-path build/trained-model.h5 \
  --batch-size 32 \
  --learning-rate 0.0001 \
  --num-epochs 50
```

By default, kenning.scenarios.model_training script requires two classes:

- ModelWrapper-based class that describes model architecture and provides training routines,
- Dataset-based class that provides training data for the model.

The remaining arguments are provided by the form_argparse class methods in each class, and may be different based on the selected dataset and model. In order to get full help for the training scenario for the above case, run:

```
python -m kenning.scenarios.model_training \

kenning.modelwrappers.classification.tensorflow_pet_dataset.TensorFlowPetDatasetMob
ileNetV2 \
  kenning.datasets.pet_dataset.PetDataset \
  -h
```

This will load all the available arguments for a given model and dataset.

The arguments in the above command are:

- --logdir — path to the directory where logs will be stored (this directory may be an argument for the TensorBoard software),
- --dataset-root — path to the dataset directory, required by the Dataset-based class,
- --model-path — path where the trained model will be saved,
- --batch-size — training batch size,
- --learning-rate — training learning rate,
- --num-epochs — number of epochs.

If the dataset files are not present, use --download-dataset flag in order to let the Dataset API download the data.

3.1.5.6.2 Benchmarking trained model on host

The kenning.scenarios.inference_performance script runs the model using the deep learning framework used for training on a host device. It runs the inference on a given dataset, computes model quality metrics and performance metrics. The results from the script can be used as a reference point for benchmarking of the compiled models on target devices.

The example usage of the script is as follows:

```
python -m kenning.scenarios.inference_performance \

kenning.modelwrappers.classification.tensorflow_pet_dataset.TensorFlowPetDatasetMob
ileNetV2 \
  kenning.datasets.pet_dataset.PetDataset \
  build/result.json \
  --model-path
kenning/resources/models/classification/tensorflow_pet_dataset_mobilenetv2.h5 \
  --dataset-root build/pet-dataset
```

The obligatory arguments for the script are:

- ModelWrapper-based class that implements the model loading, I/O processing and inference method,
- Dataset-based class that implements fetching of data samples and evaluation of the model,
- build/result.json, which is the path to the output JSON file with benchmark results.

The remaining parameters are specific to the ModelWrapper-based class and Dataset-based class.

3.1.5.6.3 Testing ONNX conversions

The kenning.scenarios.onnx_conversion runs as follows:

```
python -m kenning.scenarios.onnx_conversion \
  build/models-directory \
  build/onnx-support.rst \
  --converters-list \
    kenning.onnxconverters.pytorch.PyTorchONNXConversion \
    kenning.onnxconverters.tensorflow.TensorFlowONNXConversion \
    kenning.onnxconverters.mxnet.MXNetONNXConversion
```

The first argument is the directory, where the generated ONNX models will be stored. The second argument is the RST file with import/export support table for each model for each framework. The third argument is the list of ONNXConversion classes implementing list of models, import method and export method.

3.1.5.6.4 Running compilation and deployment of models on target hardware

There are two scripts — kenning.scenarios.inference_tester and kenning.scenarios.inference_server.

The example call for the first script is following:

```
python -m kenning.scenarios.inference_tester \

kenning.modelwrappers.classification.tensorflow_pet_dataset.TensorFlowPetDatasetMob
ileNetV2 \
  kenning.compilers.tflite.TFLiteCompiler \
  kenning.runtimes.tflite.TFLiteRuntime \
  kenning.datasets.pet_dataset.PetDataset \
  ./build/google-coral-devboard-tflite-tensorflow.json \
  --protocol-cls kenning.runtimeprotocols.network.NetworkProtocol \
  --model-path
./kenning/resources/models/classification/tensorflow_pet_dataset_mobilenetv2.h5 \
  --model-framework keras \
  --target "edgetpu" \
  --compiled-model-path build/compiled-model.tflite \
  --inference-input-type int8 \
```

```
--inference-output-type int8 \
--host 192.168.188.35 \
--port 12345 \
--packet-size 32768 \
--save-model-path /home/mendel/compiled-model.tflite \
--dataset-root build/pet-dataset \
--inference-batch-size 1 \
--verbosity INFO
```

The script requires:

- ModelWrapper-based class that implements model loading, I/O processing and optionally model conversion to ONNX format,
- ModelCompiler-based class for compiling the model for a given target,
- Runtime-based class that implements data processing and the inference method for the compiled model on the target hardware,
- Dataset-based class that implements fetching of data samples and evaluation of the model,
- ./build/google-coral-devboard-tflite-tensorflow.json, which is the path to the output JSON file with performance and quality metrics.

In case of running inference on a remote edge device, the --protocol-cls RuntimeProtocol also needs to be provided in order to provide communication protocol between the host and the target. If --protocol-cls is not provided, the inference_tester will run inference on the host machine (which is useful for testing and comparison).

The remaining arguments come from the above-mentioned classes. Their meaning is following:

- --model-path (TensorFlowPetDatasetMobileNetV2 argument) is the path to the trained model that will be compiled and executed on the target hardware,
- --model-framework (TFLiteCompiler argument) tells the compiler what is the format of the file with the saved model (it tells which backend to use for parsing the model by the compiler),
- --target (TFLiteCompiler argument) is the name of the target hardware for which the compiler generates optimized binaries,
- --compiled-model-path (TFLiteCompiler argument) is the path where the compiled model will be stored on host,
- --inference-input-type (TFLiteCompiler argument) tells TFLite compiler what will be the type of the input tensors,
- --inference-output-type (TFLiteCompiler argument) tells TFLite compiler what will be the type of the output tensors,
- --host tells the NetworkProtocol what is the IP address of the target device,
- --port tells the NetworkProtocol on what port the server application is listening,

- --packet-size tells the NetworkProtocol what the packet size during communication should be,
- --save-model-path (TFLiteRuntime argument) is the path where the compiled model will be stored on the target device,
- --dataset-root (PetDataset argument) is the path to the dataset files,
- --inference-batch-size is the batch size for the inference on the target hardware,
- --verbosity is the verbosity of logs.

The example call for the second script is as follows:

```
python -m kenning.scenarios.inference_server \
  kenning.runtimeprotocols.network.NetworkProtocol \
  kenning.runtimes.tflite.TFLiteRuntime \
  --host 0.0.0.0 \
  --port 12345 \
  --packet-size 32768 \
  --save-model-path /home/mendel/compiled-model.tflite \
  --delegates-list libedgetpu.so.1 \
  --verbosity INFO
```

This script only requires Runtime-based class and RuntimeProtocol-based class. It waits for a client using a given protocol, and later runs inference based on the implementation from the Runtime class.

The additional arguments are as follows:

- --host (NetworkProtocol argument) is the address where the server will listen,
- --port (NetworkProtocol argument) is the port on which the server will listen,
- --packet-size (NetworkProtocol argument) is the size of the packet,
- --save-model-path is the path where the received model will be saved,
- --delegates-list (TFLiteRuntime argument) is a TFLite-specific list of libraries for delegating the inference to deep learning accelerators (libedgetpu.so.1 is the delegate for Google Coral TPUs).

First, the client compiles the model and sends it to the server using the runtime protocol. Then, it sends the next batches of data to process to the server. In the end, it collects the benchmark metrics and saves them to a JSON file. In addition, it generates plots with performance changes over time.

3.1.5.6.4.1 Render report from benchmarks

The kenning.scenarios.inference_performance and kenning.scenarios.inference_tester create JSON files that contain:

- command string that was used to generate the JSON file,

- frameworks along with their versions used to train the model and compile the model,
- performance metrics, including:
  - CPU usage over time,
  - RAM usage over time,
  - GPU usage over time,
  - GPU memory usage over time,
- predictions and ground truth to compute quality metrics, i.e., in form of confusion matrix and top-5 accuracy for classification tasks.

The kenning.scenarios.render_report renders the report RST file along with plots for metrics for a given JSON file based on selected templates.

For example, for the file ./build/google-coral-devboard-tflite-tensorflow.json created in Running compilation and deployment of models on target hardware the report can be rendered as follows:

```
python -m kenning.scenarios.render_report \
  build/google-coral-devboard-tflite-tensorflow.json \
  "Pet Dataset classification using TFLite-compiled TensorFlow model" \
  docs/source/generated/google-coral-devboard-tpu-tflite-tensorflow-classification.rst \
  --img-dir docs/source/generated/img/ \
  --root-dir docs/source/ \
  --report-types \
    performance \
    classification
```

Where:

- build/google-coral-devboard-tflite-tensorflow.json is the input JSON file with benchmark results
- "Pet Dataset classification using TFLite-compiled TensorFlow model" is the report name that will be used as title in generated plots,
- docs/source/generated/google-coral-devboard-tpu-tflite-tensorflow-classification.rst is the path to the output RST file,
- --img-dir docs/source/generated/img/ is the path to the directory where generated plots will be stored,
- --root-dir docs/source is the root directory for documentation sources (it will be used to compute relative paths in the RST file),
- --report-types performance classification is the list of report types that will form the final RST file.

The performance type provides report sections for performance metrics, i.e.:

- Inference time changes over time,
- Mean CPU usage over time,

- RAM usage over time,
- GPU usage over time,
- GPU memory usage over time.

It also computes mean, standard deviation and median values for the above time series.

The classification type provides report section regarding quality metrics for classification task:

- Confusion matrix,
- Per-class precision,
- Per-class sensitivity,
- Accuracy,
- Top-5 accuracy,
- Mean precision,
- Mean sensitivity,
- G-Mean.

The above metrics can be used to determine any quality losses resulting from optimizations (i.e., pruning or quantization).

3.1.5.7 Adding new implementations

Dataset, ModelWrapper, ModelCompiler, RuntimeProtocol, Runtime and other classes from the kenning.core module have dedicated directories for their implementations. Each method in base classes that requires implementation raises NotImplementedError exceptions. Implemented methods can be also overridden, if necessary.

Most of the base classes implement form_argparse and from_argparse methods. The first one creates an argument parser and a group of arguments specific to the base class. The second one creates an object of the class based on the arguments from the argument parser.

Inheriting classes can modify form_argparse and from_argparse methods to provide better control over their processing, but they should always be based on the results of their base implementations.

3.2 Deep Learning Platforms

This Chapter lists various IoT Deep Learning Platforms on which the deep learning applications can be deployed.

3.2.1 Graphics Processing Unit

GPUs are known to be the most popular choice when it comes to running and training deep neural networks — they have a large number of cores, and they have a huge memory bandwidth. Deep learning models do not usually have any conditional code, they consist of massive numbers of simple operations on tensors making GPUs a very good inference accelerator.

There are lots of libraries that can be used to accelerate deep learning models on GPUs of various vendors:

- CUDA, CUDNN, TensorRT — NVIDIA libraries that are used to run deep learning, computer vision and other computationally demanding algorithms that can make use of GPU,
- OpenGL — originally a Graphics library, due to its support on a large variety of GPU platforms it can be also used to accelerate deep learning applications,
- OpenGL ES — similarly to OpenGL, it supports various GPUs, but for Embedded Systems, i.e., ARM Mali GPU,
- OpenCL — a cross-platform GPU computation library,
- ROCm — a development platform that can be used for deep learning applications on AMD platforms.

3.2.2 NVIDIA Jetson

NVIDIA Jetson devices are edge-AI SoCs based on ARM CPUs and with NVIDIA GPUs in a single chip. Such a solution allows to run deep learning models with real-time performance in small, energy efficient devices. One of the advantages of Jetson platforms is shared (unified) memory for CPU and GPU, meaning that there is no need to transfer data from RAM to VRAM, which significantly reduces time spent on data transfers.

Available Jetson releases:

- Jetson Xavier AGX — 8-core ARM v8.2, 512-core Volta GPU with 64 Tensor Cores, 2x NVDLA engines, 32GB unified CPU-GPU memory,
- Jetson Xavier NX — 6-core NVIDIA Carmel ARM v8.2, 384-core Volta GPU with 48 Tensor Cores, 2x NVDLA engines, 8GB unified CPU-GPU memory,
- Jetson Nano — 4-core ARM A57, 128-core Maxwell GPU, 4GB unified CPU-GPU memory,
- Older releases — Jetson TK1, Jetson TX1, Jetson TX2.

3.2.3 Deep learning inference on CPUs

It is possible to run deep learning models inference on CPUs, however it requires heavy optimizations to run efficiently. Newer CPUs provide support for vector SIMD operations, i.e., RISC-V Vector extensions, or Intel and AMD Advanced Vector Extensions. The extensions can accelerate the inference process significantly if the in-model operations are implemented in a way AVX or RVV can be used. Other ways to accelerate deep learning models on CPUs is to use Winograd convolutions, sparse models and quantization.

3.2.4 Dedicated AI Accelerators for Ultra-Low-Power

3.2.4.1 MAXIM MAX78000

The MAX78000 [Jan21a] is the first DLA augmented microcontroller by Maxim, it targets object-detection and audio processing at very high efficiency enabling battery powered designs. Its main processing core is a 32-bit ARM Cortex-M4F microprocessor that has a 32-bit RISC-V co-processor and an additional accelerator for AI workloads. The RISC-V is used for low-power sensor reading while the ARM is calculating the application tasks and controls the DLA. The two processors share 128 KB of ECC protected SRAM and 512 KB of flash memory. A security unit provides hardware accelerated 256 bit AES encryption. A power management unit can decide between 7 power states to put the whole device into the most efficient mode. [23]

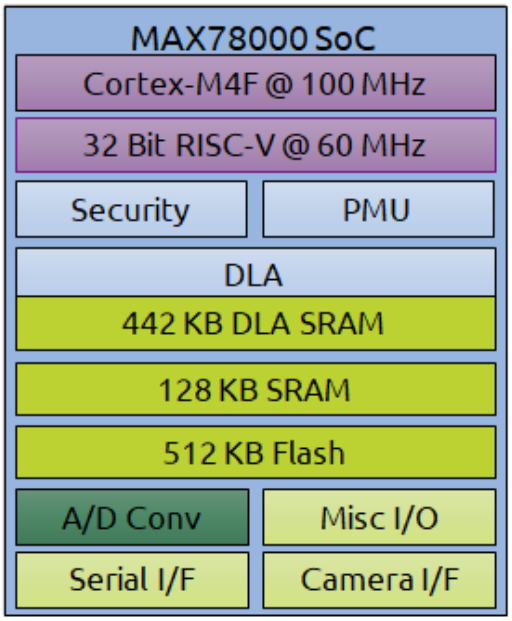

Figure 14: Block diagram of the MAX78000 [23]

The DLA has a total of 442KB local SRAM in the 64 AI cores for holding up to 3.45 million binary weights. Normally this is too small to hold models for object-detection, but the Maxim software reduces the weights for the middle layers. This reduces the accuracy but enables ultra-low-power object-detection. Each AI core has 16 convolution units that can perform

one 8bit MAC operation per clock cycle. They can be grouped to a 4x4 matrix to optimize 2D convolutions. In addition, the AI cores have a Pooling and an Activation Unit to further increase energy efficiency. [23]

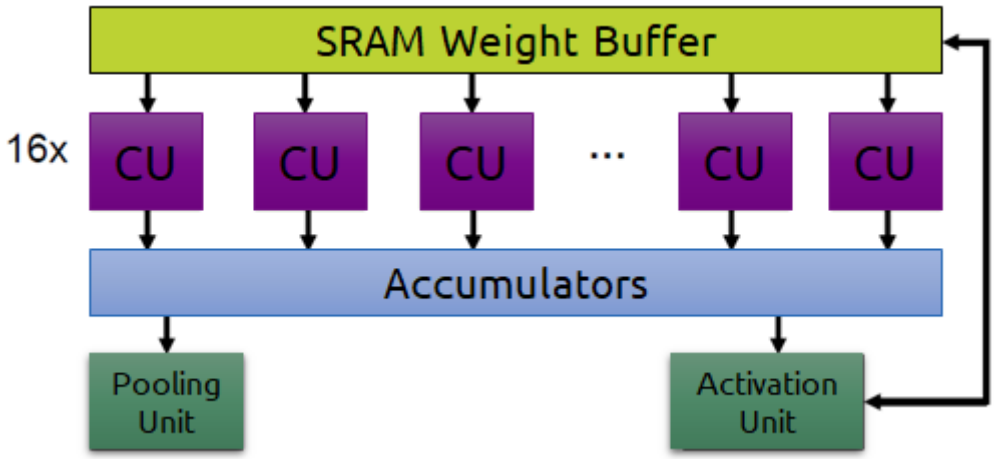

Figure 15 : Block diagram of one AI core [23]

The I/O capabilities of the MAX78000 are similar to a small microcontroller. There is a 12bit wide parallel camera interface, a I2S digital audio interface and 52 general-purpose I/O pins.

The 28mW low-power consumption at 56 GOPS leads to an efficiency of 2 TOPS/W. This makes the MAX78000 a highly efficient DLA that is already available.

Regarding the software side, the MAX78000 supports the convolutional toolsets PyTorch and TensorFlow. A dedicated Maxim software converts the trained networks to executable code for the microcontroller.

Main Processing Unit:

- Arm Cortex-M4F (32 bit) @ 100 MHz
- RISC-V (32 bit) @ 60 MHz
- 128KB SRAM
- 512KB Flash

Accelerator:

- 64 AI Cores @ 50 MHz
- 16x Convolution Units (1x1) grouped (4x4) Pooling Unit, Activation Unit, Absolute-Value and ReLU
- 442K 8-bit Weight Capacity (SRAM)

Interfaces:

- 52 GPIOs (UART, I2C, I2S, SPI), 12-bit Parallel Camera IF

Software:

- PyTorch
- TensorFlow

Performance:

- 56 GOPS @ 28 mW;
- 2 TOPS/W

Availability:

- In production, available at various distributors

3.2.4.2 Ambient GPX-10

The GPX-10 [Jan20a] from the US company Ambient Scientific is an SoC supporting on-device inference and retraining, targeting speech recognition, image processing, sensor fusion and industrial applications. The SoC combines an Arm Cortex-M4F CPU with ten AI cores and a wide variety of analogue and digital interfaces. As for most AI accelerators, MAC units are the key part of the architecture. For the GPX-10, Ambient Scientific has developed a new approach, combining analogue and digital components to perform the MAC operations. Custom SRAM cells are used to store operands and weights. After digital to analogue conversion, the multiply-accumulate step is performed in analogue processing. Finally, the sums are converted back to digital. The analogue compute engine can process operand with a precision of up to 16 bits. [Jan20]

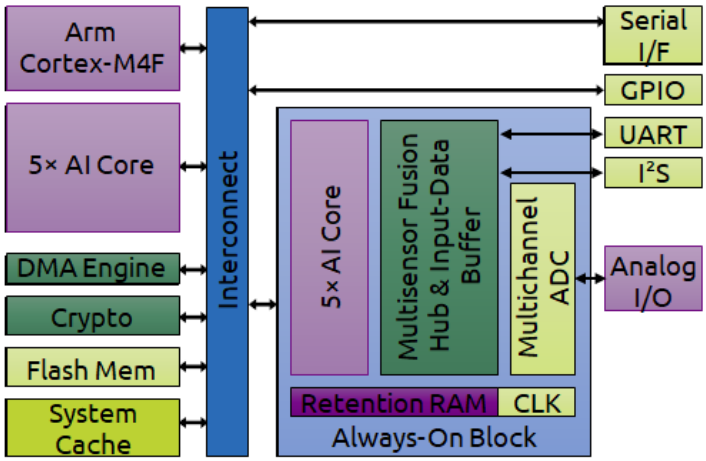

Figure 16: Architecture of the GPX-10 SoC [Jan20]

The ten AI cores are evenly distributed between an always-on block and the processor subsystem, which is powered down whenever possible to minimize power consumption. In addition to the embedded CPU and AI engines, a dedicated hardware security module is integrated for offloading AES encryption.

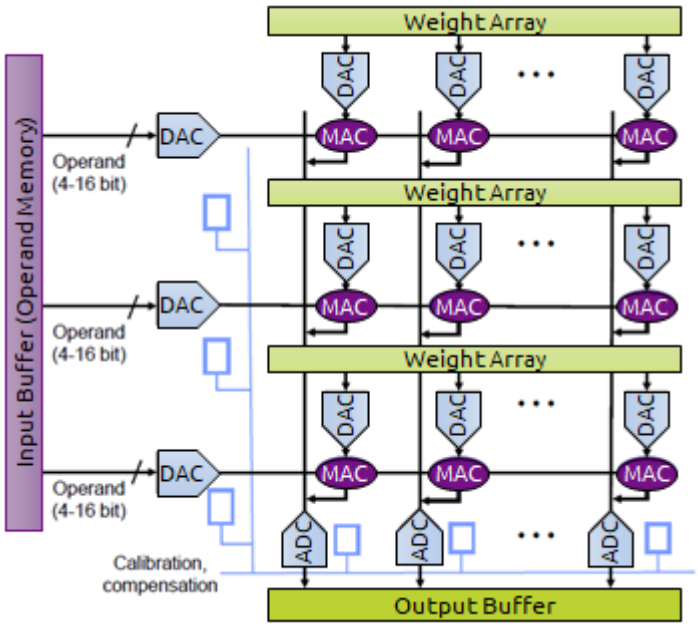

Figure 17: Compute engine of the GPX-10, combining digital storage and analog processing [Jan20]

Ambient Scientific's software development kit supports AI algorithm and model development using popular AI frameworks such as Tensor Flow, Pytorch, Café, etc.

Main Processing Unit:

- Arm Cortex-M4F (100 MHz)
- 320 kB SRAM
- 512 kB Flash

Communication:

- 1x QSPI, 1x I2S Master, 2x SPI, 3x I2C, 2x UART, 16x GPIOx16
- 8 channel 16-bit ADC

Mixed-Signal Accelerator:

- SRAM based analog MACs
- DLA Speed 150 MHz
- Always-On Block can wake-up the processor

Performance:

- 512 GOPS @ 120mW
- 4.25 TOPS/W

3.2.4.3 Kneron KL520/KL720

Kneron KL520 [24] is an AI SoC for smart-home and IoT segment applications mainly in the image processing domain (e.g., smart locks, security cameras, drones, smart home appliances, and robotics). The SoC contains an AI co-processor (Kneron NPU core) to

accelerate neural network processing, which is designed to accelerate the computing layers of a convolutional neural network (CNN).

One of the KL520 key features is being ready for the integration with an image sensor making it easy for the development of solutions for the applications mentioned above.

A newer version (KL720) is available with double efficiency [25].

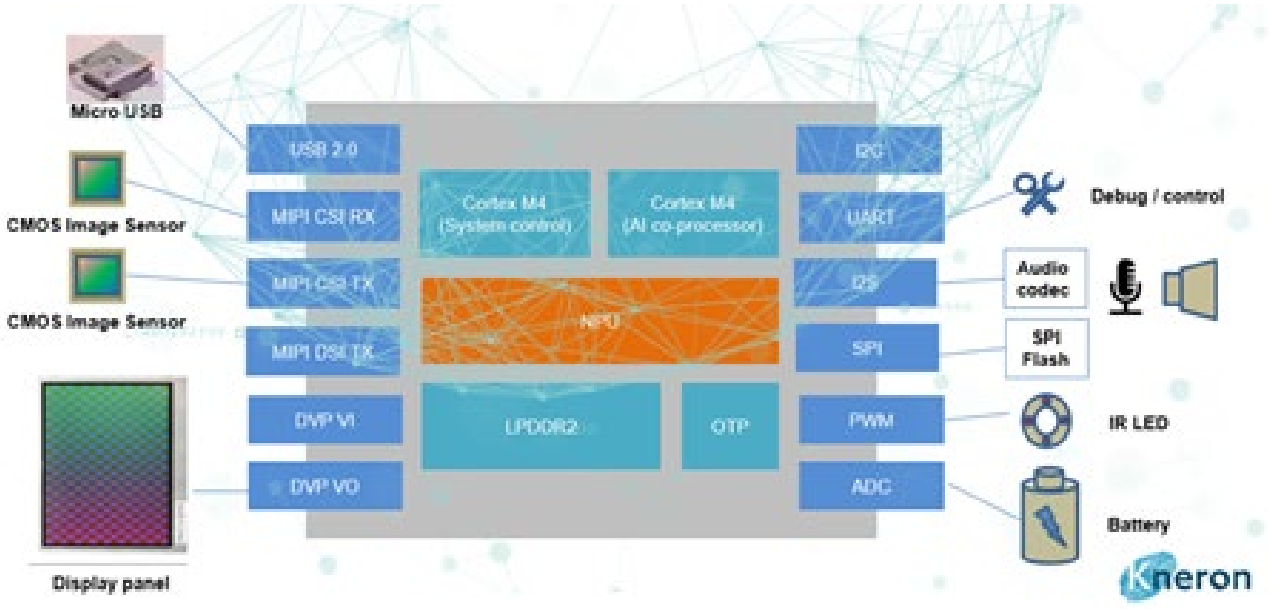

Figure 18: Kneron KL520 block diagram

The key specification points for the KL520 and KL720 are presented below.

Processor:

- 2x ARM Cortex-M4@200MHz (KL520)
- ARM Cortex-M4@400MHz (KL720)

Memory:

- Up to 64MB RAM (96 KB internal) (128MB for the KL720)
- Up to 64MB SPI NOR Flash (96 KB internal) (128MB for the KL720)

NPU Accelerator:

- 64x Convolution Units (3x3) @300 Mhz achieving 346 GOPS for 8-bit int (1st gen in KL520)
- Maximum Frequency @ 696 MHz; Peak Throughput of 8-bit mode: 1425 GOPS; 1024MAC/cycle (2nd gen in KL720)

Communication:

- CSI / DSI
- DVP
- I2C
- SPI

- UART
- USB
- For KL720 also: USB 3.0 device interface; USB 2.0 host interface; PWM; GPIO; SDIO; SDCARD

Power:

- KL520: 692 GOPS/Watt (500 mW)
- KL720: 1.725W@Yolov3_608

Software frameworks:

- ONNX, TensorFlow, Keras, Caffe

Availability:

- KL520: now (M.2 / mPCI Module)

3.2.4.4 XMOS Xcore.ai

The XMOS Xcore.ai is a microcontroller offering flexible I/Os, DSP and neural network acceleration capabilities, mainly targeting intelligent IoT devices for smart home and industrial deployment. It includes two cores based on the XMOS Xcore architecture, each running at 800MHz and handling up to eight threads. In contrast to previous XMOS chips, each core includes a 256-bit vector processing unit. The VPU supports different data types including 32-, 16- and 8-bit integers, complex integers and single-bit values. Additionally, it offers instructions for multiply and accumulate operations with a performance of 51.2GOPS for 8-bit values. For external connection the chip offers a USB PHY and a two-lane MIPI interface. Additional interfaces can be emulated thanks to the real-time capabilities of the cores. The cores are capable of running real-time operating systems such as FreeRTOS. [26]

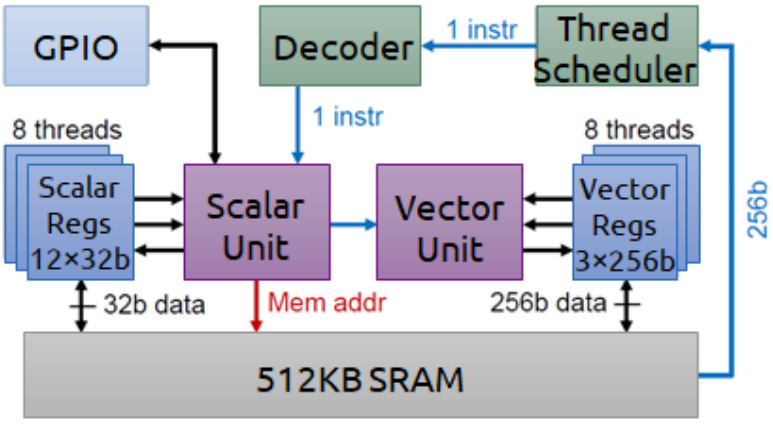

Figure 19: Architecture overview of the Xcore CPU [26]

For software development, XMOS provides libraries for DSP and neural-network functions as well as an LLVM compiler and additional programming tools. To deploy a neural networks model on the chip the models have to be converted to TensorFlow Lite. The model can then be converted to a run-time model using the XCOM converter. [26]

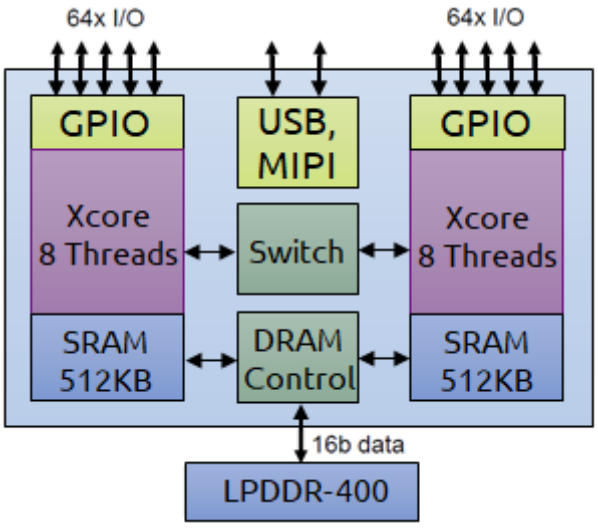

Figure 20: Block diagram of the Xcore.ai [26]

The Xcore.ai can either be used as a standalone SoC running an operating system or as an accelerator for external host CPUs. In the standalone case, a whole core will be running the operating system. When running as an external accelerator, a part of one core has to be used for the communication with the host CPU. [26]

Features

- 2x Xcore at 800MHz with eight threads each
  - 512KB SRAM per core
  - 64 I/Os per core
  - connected over a 25Gbps switch
- 16-bit 200MHz LPDDR1 interface
- USB and 2-lane-MIPI interface (connecting to the internal switch)

Package

- 14mm BGA-265
- 7mm QFN-60 (with less I/Os)

Software

- TensorFlow Lite

Performance

- One Core at 500mW (typical)
  - AI (INT8) 51GOPS
  - Binary Networks 408 GOPS
  - DSP (FSP32) 0.8Gflop/s

3.2.4.5 GreenWaves Technologies GAP8 and GAP9

GAP8 [27] and GAP9 [28] are two generations of ultra-low power GAP processors based on RISC-V architecture and highly optimized for audio and image processing applications. GAP processors bring together the low power benefits of a microprocessor with the flexibility of programmable RISC-V cores as a compute cluster. The compute engine benefits from an extended RISC-V instruction set architecture that can support single-instruction-multiple-data operations and bit manipulations targeted for convolutional neural networks.

GAP8 consists of 9 32-bit RI5CY cores in total, one high-performance micro-controller core and 8 cores for compute-intensive instructions combined with a convolutional neural network accelerator (HWCE). GAP9 has an extra core in its compute engine, for a total of 10 RISC-V cores. GAP9 processing cores can run on a higher clock speed of 400MHz whereas the controller core of GAP8 runs up to 250MHz and other cores run with 175MHz clock speed.

Regarding the communication capabilities of GAP9 and GAP8, both include HyperBus, SPI, Camera parallel interface and general purpose input/outputs interfaces.

GAP processors can be used with GAP software development kit which supports tflite and onnx frameworks for deep neural networks. [27]

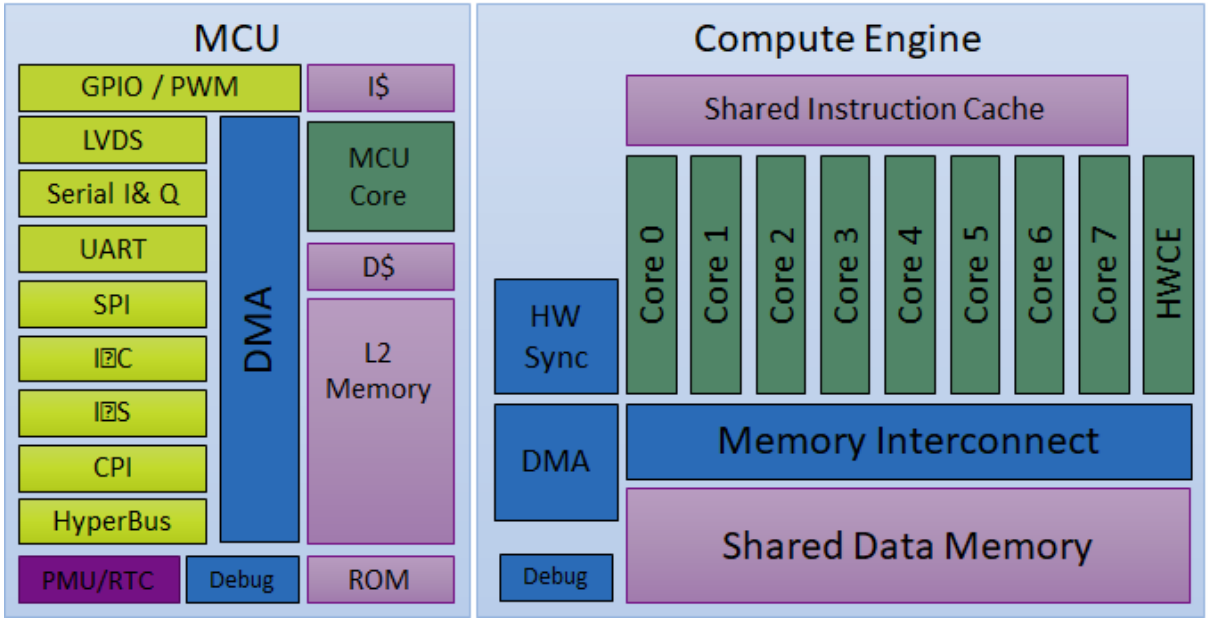

Figure 21: Block diagram of the GAP8 processor [27]

Regarding the performance, GAP9 can reach up to 160.8 GOPS consuming 50mW which results in a 3.2 TOPS/W efficiency while GAP8 reaches 22.65 GOPS at 96 mW which yields a 0.236 TOPS/W performance.

3.2.4.6 Kendryte K210

The K210 [29] is optimized for real-time vision, to match this task it comes with two RISC-V CPUs based on the open-source Rocket design and implementing a RV64 G instruction set. The nominal speed of 400MHz at 0.9V saves power and provides enough performance for

simple applications like QVGA video processing. For applications like VGA processing it can be overclocked to 800MHz at 1.0V. Out of the reason that the core does not support a Memory Management Unit, typically one CPU runs an operating system and the other a neural network performing for example image recognition. The heavy convolution layers are offloaded to the KPU, which is Kendrytes AI engine that is processing INT16 weights instead of INT32 and therefore needs less power. An additional power saving measure is a rather slow operating rate reached by the MAC units producing only one result every four cycles. The KPU includes two prefetch engines, one is fetching weights and the other one activations. Both designs are optimized for convolution layers but there is no specific logic for normalization, activation functions or pooling. These functions must be executed on the CPU which becomes the bottleneck. [30]

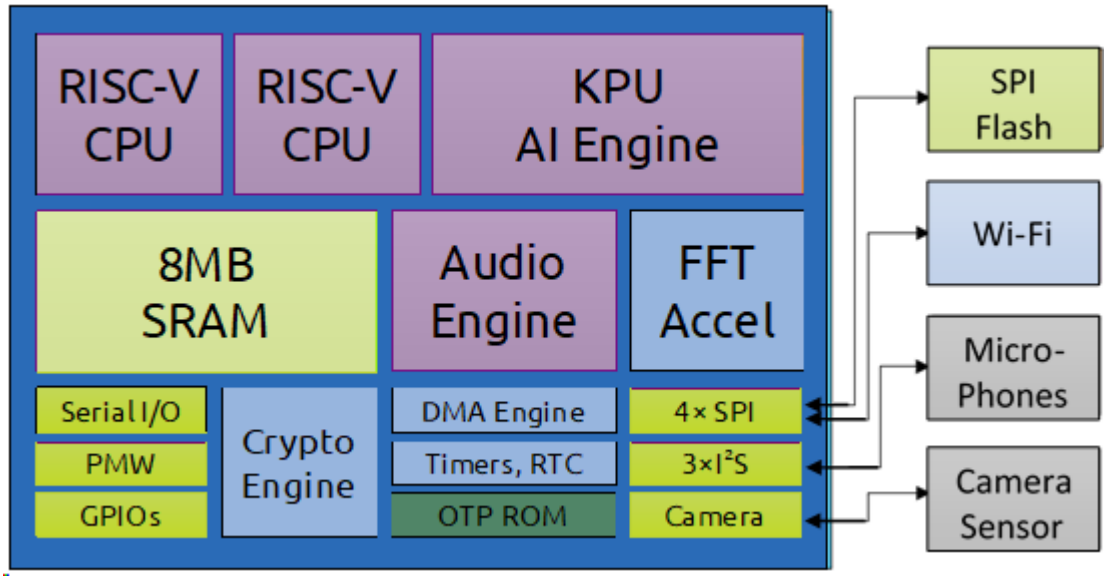

Figure 22: Block diagram of the Kendryte K210 SoC [30]

The whole chip has 8 MB SRAM, where two are dedicated for the KPU [31]. The other 6 MB are intended for a neural network, the RTOS and application code. The neural network could be the Tiny YOLOv2 real-time object detection neural network containing 5.1 MB of parameters. To perform larger neural networks the chip has 256MB external flash memory, which could be reached through a serial port. Using the external flash memory will reduce performance to less than 10 fps.

Additionally, the chip offers a DSP core having up to eight audio channels. It functions as an audio processor and can determine the location of particular sounds by analysing signals from a microphone array.

The chip connects to a single camera module using standard parallel interface and can also link to a small display (VGA) to provide a local video output. Although it lacks analogue I/O a motor can be controlled by PWM outputs.

Regarding the software side, the chip supports FreeRTOS and Kendryte provides a tool for converting a pre-trained neural network to run on the K210. Therefore, developers can use TensorFlow, Keras and Darknet to build and train their neural network.

Main Processing Unit:

- 2x RISC-V RV64G (64 bit)

- 800 MHz
- 6 MB SRAM
- 245 MB flash

Accelerator:

- 64x MAC (576 bit)
  - 18 INT16 @ 400MHz
  - 2 MB SRAM

Interfaces:

- Parallel Camera IF
- GPIOs / PWM / I2S

Software:

- FreeRTOS
- Tensorflow/Keras/Darknet

Performance:

- 460 GOPS @ 1 W
- 0.460 TOPS/W

Availability:

- In production, available at various distributors

3.2.4.7 NDP120

The NDP120 [32], which is optimized for speech recognition, is the successor of the NDP10x. The ultra-low-power edge AI processor owns an Arm Cortex-M0 having 1.4 MB SRAM. To save power while supporting auditory wake-up events like the typical magic words it has an always-on mode with a CPU frequency of 20MHz. The normal frequency is 100 MHz at 1.1 V. In this mode more sophisticated AI and DSP algorithms can analyse the stored audio. To store the audio the chip provides a separate SRAM buffer which can hold up to 10 seconds of audio. The audio is received by four microphones connected to a Pulse-density-modulation (PDM) with up to 48 kHz sampling. The chip also has a Hifi 3 DSP providing simple audio-filter tasks like beam forming, near- and far field processing. The PDM is also reconfigurable as an I2S/TDM interface. Over the I2C several low frequency sensor data from the accelerometer and infrared detectors can be received and fused by the NDP120, which is additionally described as a multipurpose graph processor. [31]

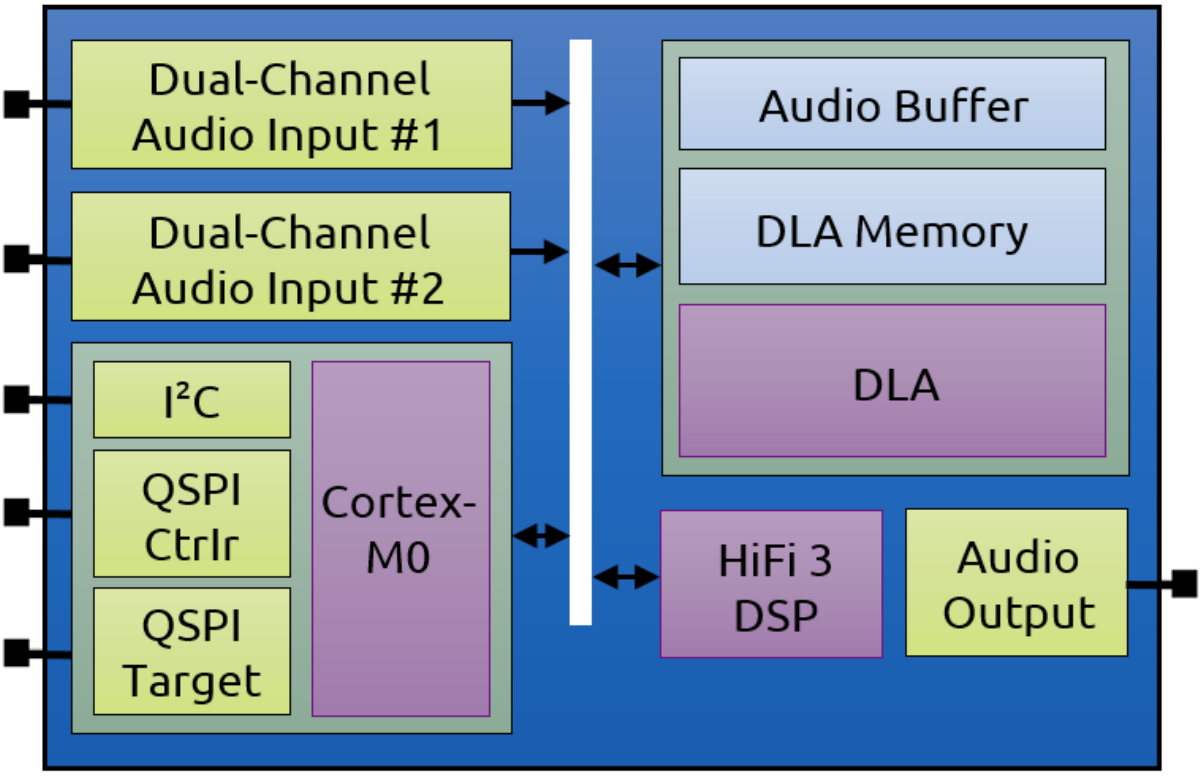

Figure 23: Block diagram of the NDP120 SoC [31]

Taking a look at the neural networks the chip supports convolutional neural networks, gated recurrent units and long/short-term-memory. Up to 896 KB of neural-network parameters could be stored. When the network size is bigger, mixed-precision models with INT8, INT4, INT2 and binary parameters could be used. For example, in binary mode more than seven million parameters may be saved. For high-precision tasks also INT16 activations could be used.

Regarding the software side developers can implement their network with a Tensorflow or Python based interface provided by a software-development kit. Both directly compile pre-trained models for the NDP120. Additionally, the company offers a training development kit that let customers target the chips hardware.

Main Processing Unit:

- Arm Cortex-M0
  - Always-on 20 MHz
  - Boost 100 MHz
  - 1.4 MB SRAM

Accelerator:

- Streaming 16-bit word Input
- 4-bits weights, 8-bits activation
- Fully connected Layers
- 896 KB of neural network parameters

Communication:

- 3x I²S Input, QSPI, I²C, GPIO

Software:

- Tensorflow, Python

Performance:

- 1900 GOPS @ < 500 mW
- 3.8 TOPS/W

Availability:

Volume production starts in Q3 2021

3.2.5 Dedicated AI Accelerators

3.2.5.1 Tensor Cores

Tensor Cores are specialized processing units that perform 4x4 FP16 matrix multiplications, and can add a third FP16 or FP32 matrix to the currently processed matrix to compute a final FP32 result. Such operations are one of the most common operations in deep neural networks inference. Tensor Cores significantly improve the performance of convolutions and other operations involving matrix multiplication.

Tensor Cores are present in NVIDIA GPUs starting from Volta architecture.

Tensor Cores are also present in NVIDIA Jetson Xavier edge-AI devices.

3.2.5.2 Google Coral

Google launched the first Tensor Processing Unit (TPU) in 2015. It is an accelerator that is specially optimized for neural network machine learning. Google has developed its own software for the accelerator TensorFlow. Google itself uses the TPU for Google Street View processing, but also in the projects AlphaGo and AlphaZero.

To support especially small neural networks in the area of edge processing, Google has brought out a smaller version of the TPU, the Edge TPU or Google Coral. The chip comes from the manufacturer embedded in a variety of form factors with different communication options, including M.2, mini-PCIe, and USB.

The exact structure of the Google Edge TPU is not known, but like its bigger brother works exclusively with INT8. However, networks that use other data types can be transformed. A microprocessor report wrote about the original TPU made by Google in 2015. The basic structure of the Pes could be similar, but since the performance is so different, it cannot be used as a reference. The TPUv3 has an output of 90 TOPS and consumes 450 W, which means 0.2 TOPS/W, whereas the Edge TPU has a performance of 4 TOPS at a power consumption of 2 W. Instead of TensorFlow, which results in an efficiency of 2 TOPS/W, 10 times the

efficiency of the TPUv3. Unlink the big TPUs the Edge TPU is programmed in TensorFlow Lite.

3.2.5.3 Intel Myriad X

Intel Movidius Myriad X [33] is a vector processing unit that encapsulates the Neural Compute Engine (NCE), as a deep learning accelerator. Thanks to its 16 Streaming Hybrid Architecture Vector Engine (SHAVE) cores, the NCE, high-throughput memory and filtering, sharpening and gamma correction hardware accelerators, Myriad is highly optimized for computer vision, video processing applications and deep learning inference.

Myriad includes 2 SPARC-compatible LEON CPUs that run at 700 MHz and act as managing cores, as well as 16 programmable vector processors with 256 KB of L2 cache storage which are based on VLIW SHAVE microarchitecture. The SHAVE cores are capable of executing deep neural network activation functions such as ReLU and Tanh, while the NCE is responsible for doing the heavy-lifting for convolutional and pooling layers. Myriad uses a 2.5 MB on-chip SRAM as well as a 32-bit LPDDR4.

As for Myriad's I/O capabilities, there are 16 MIPI lanes that can be used to connect up to 8 cameras directly to the device. USB 3.1, PCIe Gen3 interface and 10 GbE are also included.

Myriad is capable of achieving 0.5 trillion operations per second (TOPS) by performing 1242 GOPS while consuming only 2.5 W of power.

Regarding the software support, Myriad includes a software development kit with a custom deep learning compiler that supports Caffe and Tensorflow models. [33]

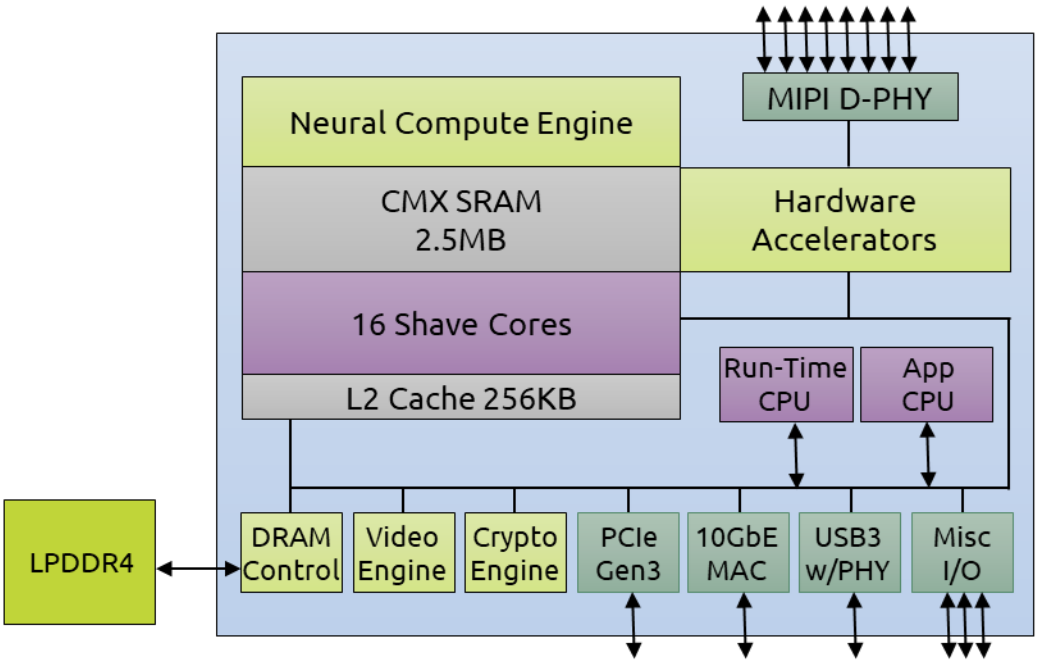

Figure 24: Block diagram of the Myriad X [30]

Main Processing Unit:

- 2x SPARC-compatible Leon CPU (700 MHz)

- 16x Shave Cores VLIW (700 MHz)
  - Scalar unit 64 bit
  - Vector unit 256 bit
- L2 256 KB SRAM
- 2.5 MB SRAM in Connection Matrix (CMX)
- LPDDR4 32 bit

Accelerator:

- Neural Compute Engine
- H.265, H.264, AES
- Hardware Accelerators: debayering, gamma correction, filtering, and sharpening

Interfaces:

- PCIe Gen3 x1, 10GbE, USB3, 8x MIPI

Software:

- Tensorflow
- Caffe

Performance:

- 1242 GOPS @ 2.5 W
- 0.5 TOPS/W

3.2.5.4 Coherent Logix HX40416

Coherent Logix has served the military and aerospace sectors for 15 years with software-defined radios and other niche applications. With this expertise, they are now entering the commercial market and aim to establish themselves in the field of Edge-AI with their configurable processor HyperX. The first available variant of this processor will be the HX40416 [34], but a whole range of devices will follow. The HX40416 is suitable for a wide range of applications, including 3G/4G receivers, GPS receivers, and sensor and video processing. To make all this possible, the processor is equipped with various function blocks, a general-purpose processor, a DSP and a neural accelerator.

Coherent Logix describes the structure of the processor to be similar to an FPGA; there is an array of processing elements (Pes) and a programmable interconnection structure through data memory routers (DMRs). An example of an arrangement of functions in this array of Pes and DMRs is depicted in Figure 25. However, in contrast to an FPGA, this processor does not require RTL development, place and route, or timing optimizations, because the compiler handles everything.

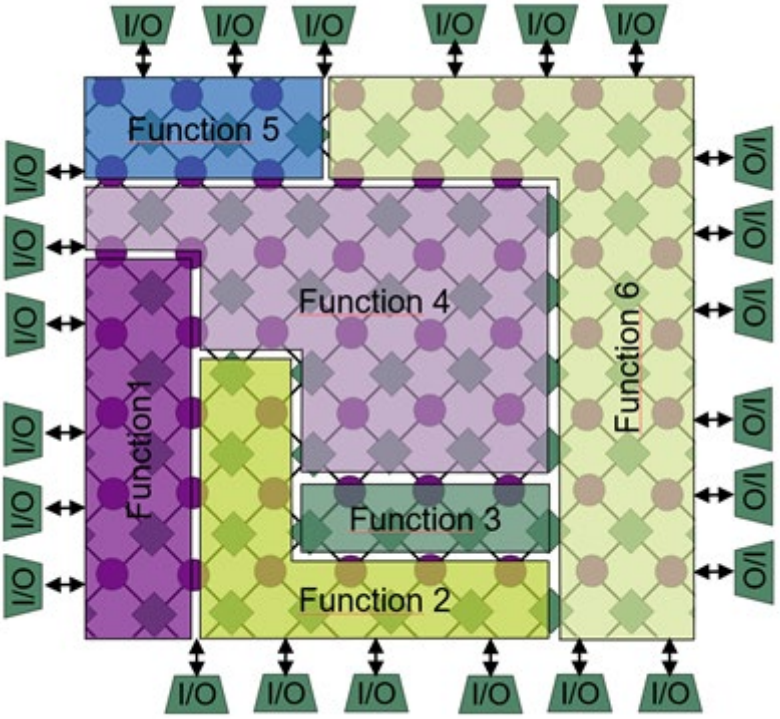

Figure 25: Function blocks inside the PE and DMR Array [34]

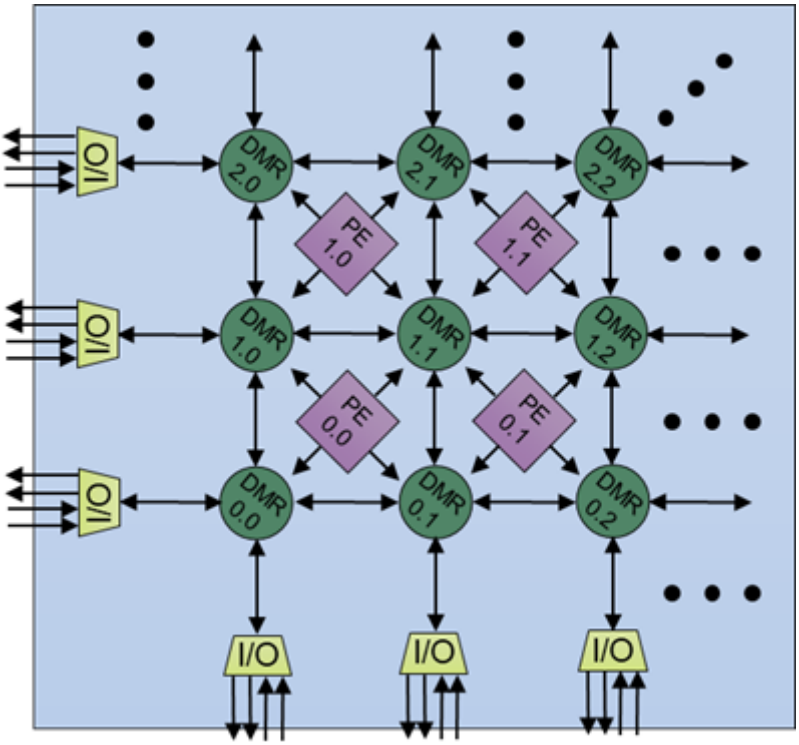

Figure 26: Detailed view of the Accelerator structure [34]

Figure 26 shows a detailed view of the accelerator. It is shown that each PE is connected to the DMR matrix via four lines. A DMR thus has eight configurable communication ports; each

port has 4x16-bit channels. DMRs have instruction memory (8Kx16-bit) and data memory (16Kx16-bit). There is a total of 20 MB of memory on the chip. Another interconnect not shown in the graphic, is a message fabric, which allows dynamic processor configuration by loading new instructions from external memory. The DMRs also implement a pass-through mode to transfer data through the fabric quickly.

The HX40416 is manufactured using the 14 nm FinFET process, and samples should be available from 3Q21. The accelerator has 416 Pes interconnected in a 16x26 matrix and 459 DMRs. A high-level operating system like Linux can run on the integrated SiFive U54 CPU. The performance is estimated to be 1.664 INT16 MAC operations per cycle, resulting in a performance of 6.6 trillion operations per second (TOPS) at a clock speed of 2 GHz.

The chip will Integrate DDR DRAM and Coherent plans to integrate the rest of the IOs via a chiplet design similar to the current Ryzen CPUs from AMD. A wide variety of interfaces can then be provided, including CAN, Ethernet, HDMI, PCIe, and USB.

The HX40416 can be programmed with the included software library, which provides ready-to-use blocks, e.g., for computer vision, deep learning, image processing, video encoding, or wireless communications. For classical neural networks, the Caffe and Tensorflow frameworks are supported. Additionally, the Coherent Logix SDK contains tools that allow programmers to convert Python code, containing OpenCV graph API calls, into a computational graph based on HyperX library functions.

Main Processing Unit:

- 4x RISC-V SiFive U54 (2 GHz)
- 4x 32-bit DDR4-1600 (25.6 GB/s)
- 20 MB SRAM

Accelerators:

- 16x26 Processing Elements (416 PEs)
- 64-bit SIMD DSP (2 GHz)
- 17x27 Data Memory Routes (459 DMRs)
- 8Kx16-bit Instruction storage
- 16Kx16-bit Data storage

Communication:

- PCIe, GbE, USB, MIPI, HDMI, Interlaken

Performance:

- 1600 GOPS @ 8 W
- 1.6 TOPS/W (INT8)

Availability:

- Q3 2021, the Chip could be integrated into a SMARC Module

3.2.5.5 Hailo-8

Hailo is a small company (50 employees) from Israel. It has been on the market for two years and Claims that its new chip, Hailo-8 [35], can run the entire ResNet-50 in internal SRAM while consuming only 1.7 W. The network, working on images with a resolution of 224x224 px, achieves a performance of 672 FPS. This corresponds to 395 FPS/W, making the chip over 20x more energy efficient than an NVIDIA Jetson AGX Xavier.

The chip is manufactured in 16 nm and has an integrated ARM Cortex-M4 CPU, which controls the inference operations in standalone mode, in this mode no Host System is needed to provide Data or Control to the Chip. Furthermore, the chip can work as a co-processor; for this, it contains different communication interfaces, including Ethernet, MIPI, and PCIe. A unique feature of the accelerator is that it does not use external DDR memory and works solely on the internal SRAM. If the neuronal Network exceeds the on Chip SRAM capacity, multiple of these chips can be connected via Ethernet or PCIe to work in parallel. [35]

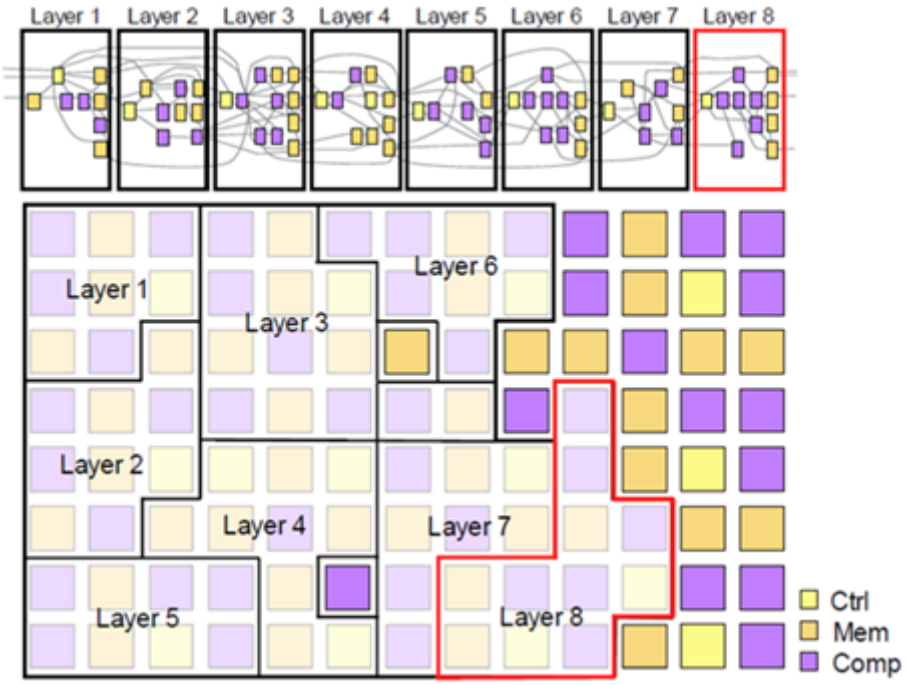

Figure 27: Visualization of the DNN layers on the chip [35]

In contrast to many other architectures, e.g., the Google Coral TPU, the accelerator does not work with a single set of function blocks such as pooling activation or convolution. Instead, it relies on an approach, where several small blocks of these units exist, and can be combined to form larger units. However, it still follows a data-flow architecture. Figure 27 shows the configurable architecture of the Hailo-8. It can be seen that each layer of the neural network is mapped onto the chip in such a way that it has access to the exact number of units it requires. There will always be a control unit with a different number of memory and compute blocks, depending on the requirements of the neural networks. The manufacturer states that this type of architecture can achieve better hardware utilization. [35]

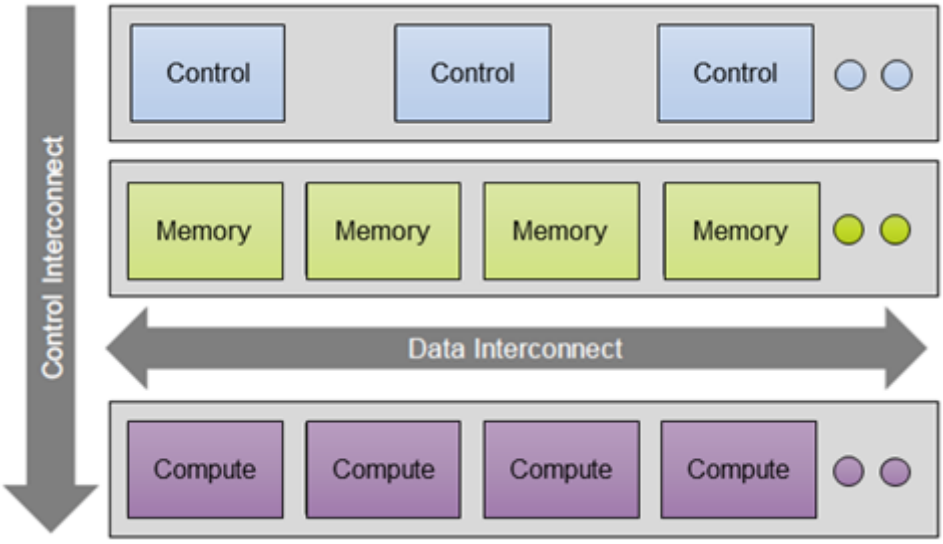

Figure 28: Hailo-8 Bus Structure [35]

Figure 28 depicts the bus structure, in terms of the control and data bus, of the Hailo-8. Furthermore, it can be seen that one control unit can serve multiple compute and memory units. The primary processing elements are INT8-based, but four INT8 multipliers can be combined to handle INT16 operations. It is not known how many MAC units a PE combines.

The performance of the Hailo-8 is reported to be 26 TOPS, similar to an NVIDIA Jetson AGX Xavier autonomous driving processor. It should be noted that it does not support many of the features the NVIDIA chip possesses. However, raw computing power is not the most significant advantage of the Hailo-8 that is its power efficiency, i.e., 335 FPS/W with a ResNet-50. As mentioned above, the manufacturer specifies Ethernet, MIPI, or PCIe as IO. At the moment, only Tensor Flow is mentioned as a supported framework for machine-learning.

Main Processing Unit:

- Cortex M4
- 32 MB SRAM (estimated by MPR)

Accelerators:

- Coarse-grained Array of PE, MEM and Control
- Configurable during runtime

Communication:

- PCIe Gen.3 x4, GbE, MIPI

Performance:

- 26 TOPS @ 9 W
- 2.8 TOPS/W

Availability:

- M.2 PCIe, mPCIe

3.2.5.6 Sophon BM1880

The Sophon family developed by Bitmain consists of low-power neural processors. In 2015 the company Bitmain, which had until then been mainly active in the crypto mining sector, started developing a neural processor. So far, they have released two neural accelerator chips for the edge computing market. The current and third variant is the BM1880 [30] manufactured in 7nm. Figure 29 depicts the block diagram of the BM1880; it shows the two CPUs next to the accelerator. On the one hand, a dual-core ARM A53 CPU and, on the other hand, a single-core RISC-V CPU, responsible for all real-time operations, is implemented. Additionally, there is a video subsystem present, which implements a variety of functions such as en- or decoding, cropping, scaling or colour-space conversions. The accelerator itself has 2 MB of SRAM and achieves a performance of 1 TOPS when using INT8 data. Furthermore, the figure also shows the very extensive IO blocks, which offer a multiplicity of interfaces. Especially noteworthy are the DDR4 interface, USB, and Ethernet. PCIe is unfortunately not available. [30]

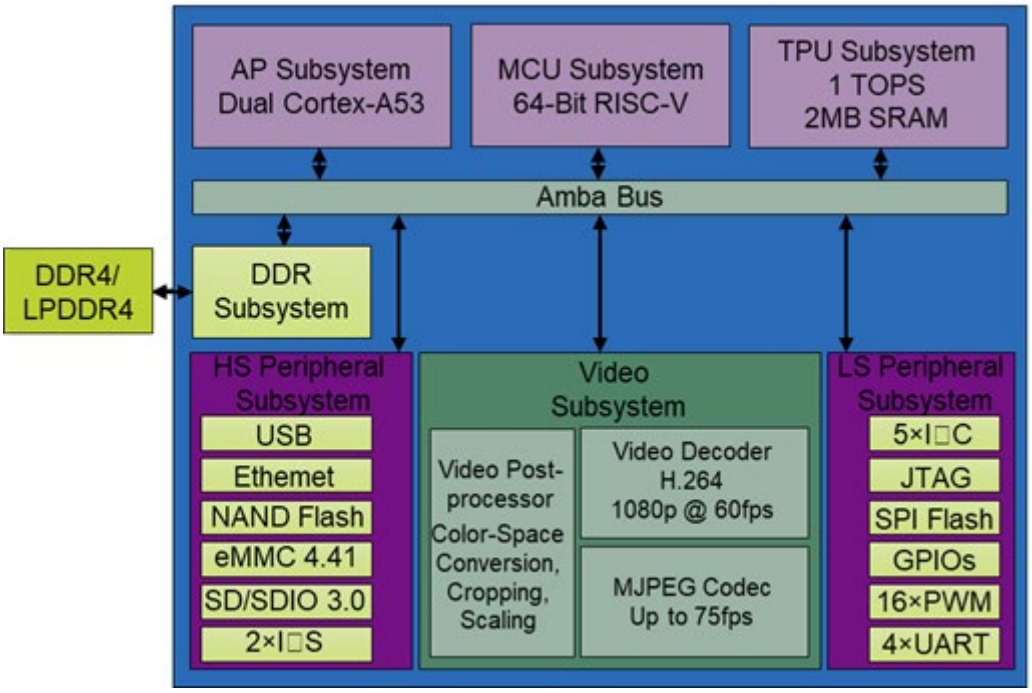

Figure 29: Block diagram of the Sophon BM1880 SoC [30]

While the first two Sophon chips were designed as co-processors, the BM1880 breaks with this tradition and is designed for standalone operation due to its Linux-capable ARM CPU. Like many other accelerators, the TPU is optimized for INT8 rather than FP32 like the old Sophon chips. The TPU contains 512 multiply-accumulate (MAC) units operating at a speed of 1.0 GHz. Each of these units can perform one INT8 operation resulting in a total performance of 1 TOPS. The external memory interface is a 32-bit DDR4 interface, and each of the blocks like CPU, TPU, and video subsystem can access it directly.

The BMNet compiler can be used to program the BM1880. It supports almost all common machine-learning frameworks such as Caffe, Tensorflow, or ONNX.

Since the chip is mainly offered as a compute stick, similar to the Intel Myriad, communication is done via USB. An evaluation board is available for standalone usage, where the Ethernet interface and the other IOs are accessible.

Main Processing Unit:

- 2x ARM Cortex A53 (1500 MHz)
- RISC-V MCU (1000 MHz)
- LPDDR4 32-bit

Accelerators:

- TPU
- 2MB SRAM
- H.264

Communication:

- GbE, USB3, 8x MIPI

Performance:

- 1000 GOPS @ 2.5W
- 0.4 TOPS/W

3.2.5.7 Blaize El Cano

El Cano [36] is a processor by Blaize that targets commercial and enterprise computer vision, as well as automotive and industrial applications. It manages multiple workloads dynamically based on their bandwidth and compute requirements and can execute multiple nodes in parallel due to graph-streaming technology. [36]

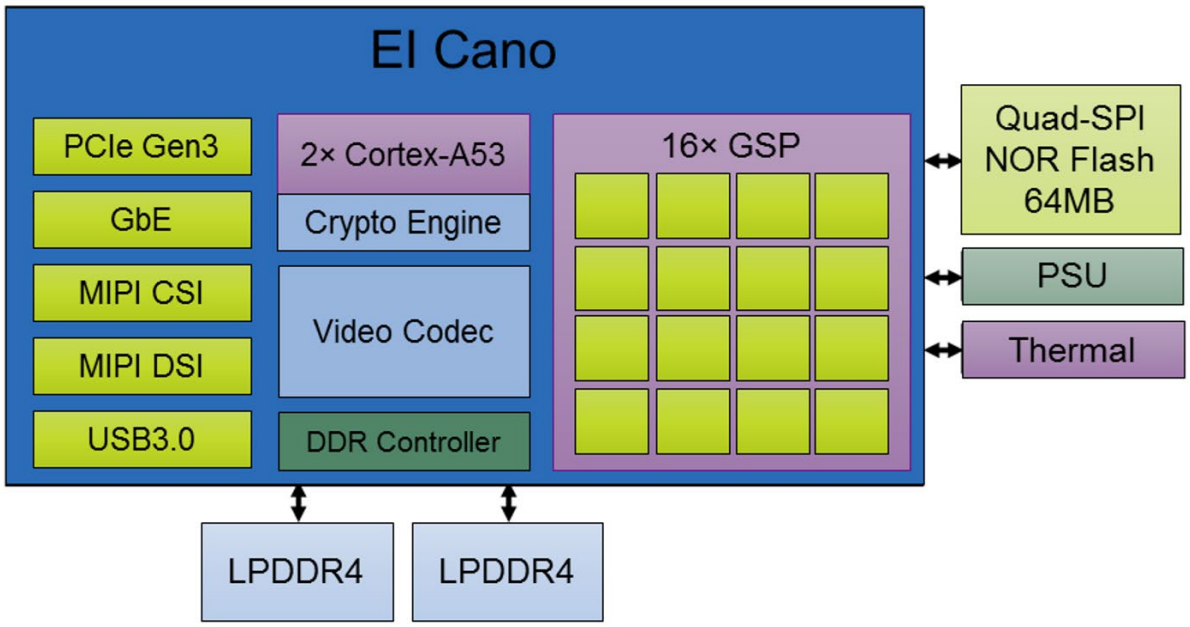

Figure 30: Sequential (TensorFlow-based) and streaming (Blaize) neural-network execution [36]

El Cano integrates two Cortex-A53s with 32KB L1 and 512KB L2 cache each that work with an ARM crypto accelerator. The Cortex-A53s are used to run an operating system and application software, while the ARM crypto accelerator to guarantee secure operation. On the chip there are two 32-bit LPDDR4 interfaces that support up to 16GB/s total bandwidth, and a video codec that encodes/decodes a single 4K video stream to 30fps.

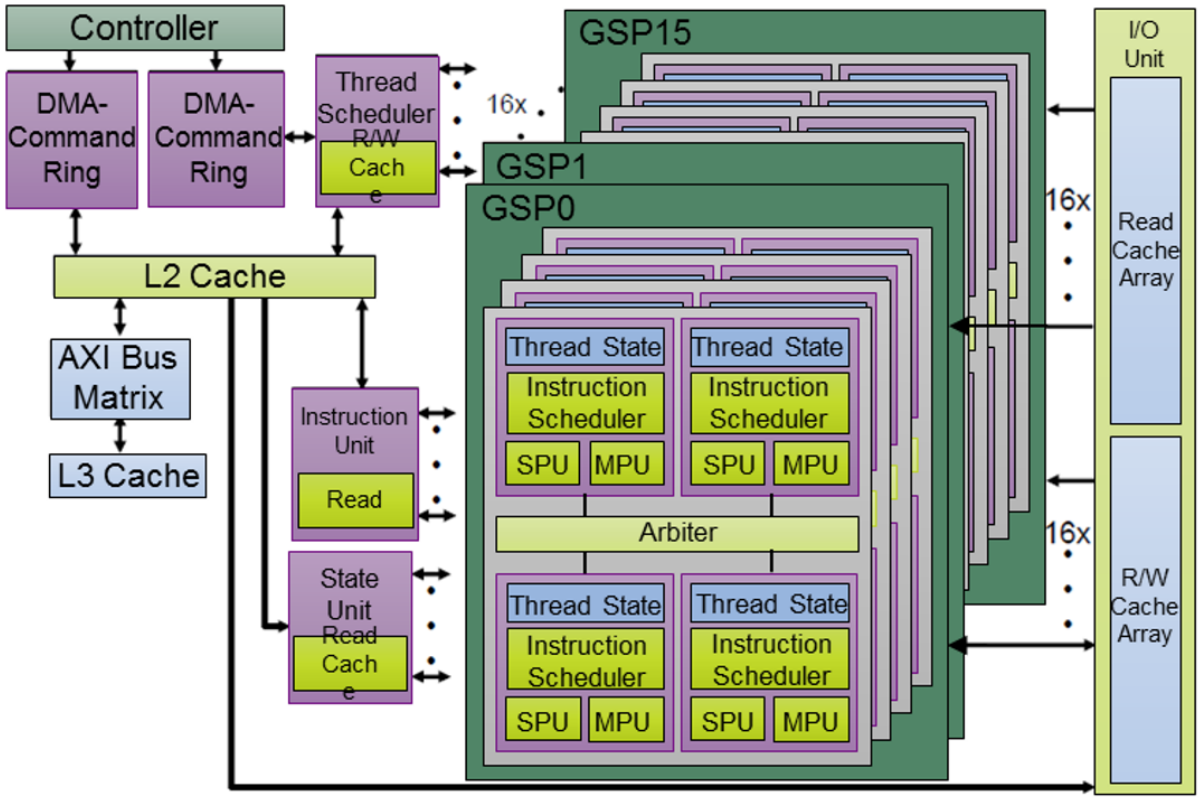

Figure 31: Blaize Pathfinder system-on-module [36]

The chip integrates 16 programmable graph-streaming processors (GSPs) that handle neural network operations, image signal processing and other tasks. Each GSP has four sets of quad processors, where every quad integrates four cores connected to an arbiter. Every core consists of thread-state memory, an instruction scheduler, a scalar processing unit (SPU) and a multiprocessing SIMD unit (MPU). The GSPs support operations up to 64-bit integers along with 8-bit floating point.

The El Cano chip has 4 camera interfaces, 1 CAN, 1 GbE, 1 PCIe Gen3 and 1 USB3.0 I/O interfaces.

The 6W power consumption at 16 TOPS leads to an efficiency of 2.7 TOPS/W.

Regarding the software side, El Cano supports the frameworks Caffe, Pytorch and TensorFlow.

Main Processing Unit:

- 2x Cortex-A53 (1000 MHz)
  - 32KB L1 cache per core
  - 512KB L2 cache per core
  - ARM crypto accelerator
  - 4K video processing @ 30fps

- 2x LPDDR4-2133 32 bit

Accelerator

- 16x Graph-streaming Processors (GSP) @ 800 MHz
  - Up to 64b operations (INT, FP)
  - processor units per GSP
- Multiple graphs at once

Interfaces:

- PCIe Gen3 (x4), GbE, CAN, 4x MIPI

Performance:

- 16 TOPS @ ~6W
- 2.7 TOPS/W

3.2.5.8 Infer X1

The InferX X1 [37] is an AI co-processor by Flex Logic that targets high-performance edge devices. The chip has a four-lane PCIe Gen3/4 interface, 32 GPIO ports and a single 32-bit LPDDR4 interface. It also has a 4MB L3 SRAM and a reconfigurable tensor processor.

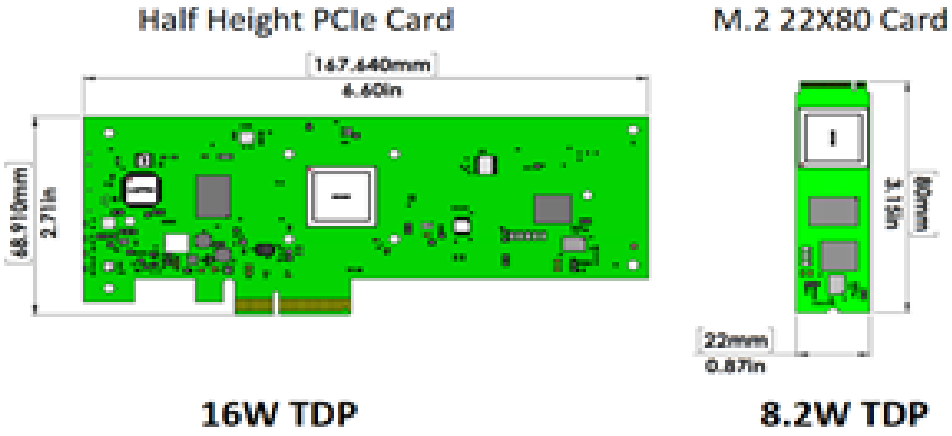

Figure 32: Size and power comparison of Half Height PCIe Card and M.2 22x80 Card

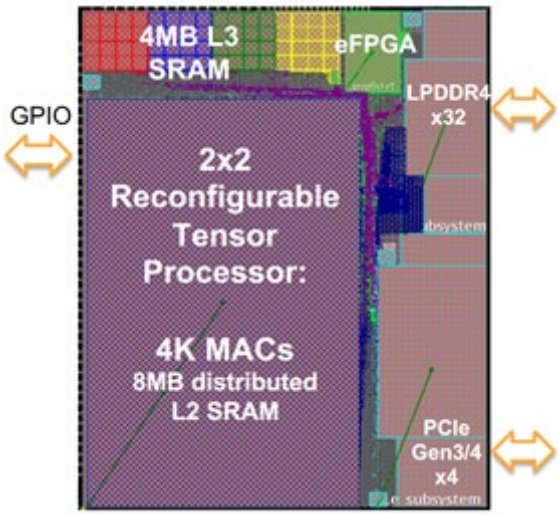

Figure 33: InferX X1 chip

For the tensor processor, there are 2x2 NNMax tiles integrated on the chip that include EFLX programmable logic and 16 NNMax clusters per tile. Each cluster is a 64 MAC systolic array and integrates 256 KB SRAMs for local weight storage. On the chip there is also an 8 MB distributed L2 SRAM. The multipliers of the MACs are 16x8 bit and the accumulator 32 bits.

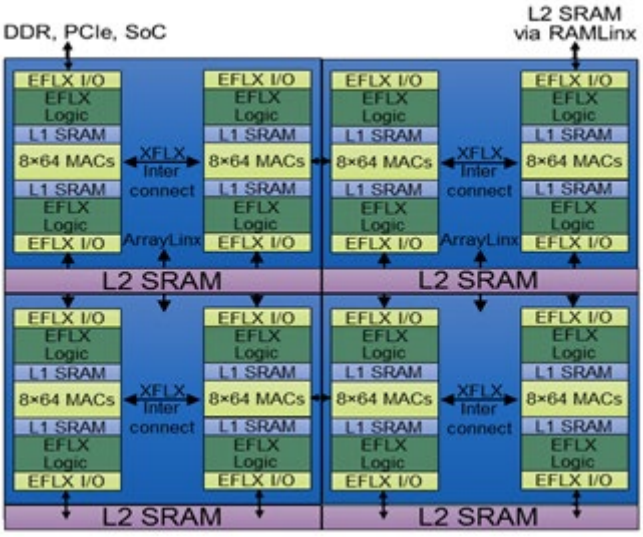

Figure 34: InferX X1 coprocessor chip [37]

The co-processor uses a data-flow architecture that reduces DRAM bandwidth by pipelining layer operations.

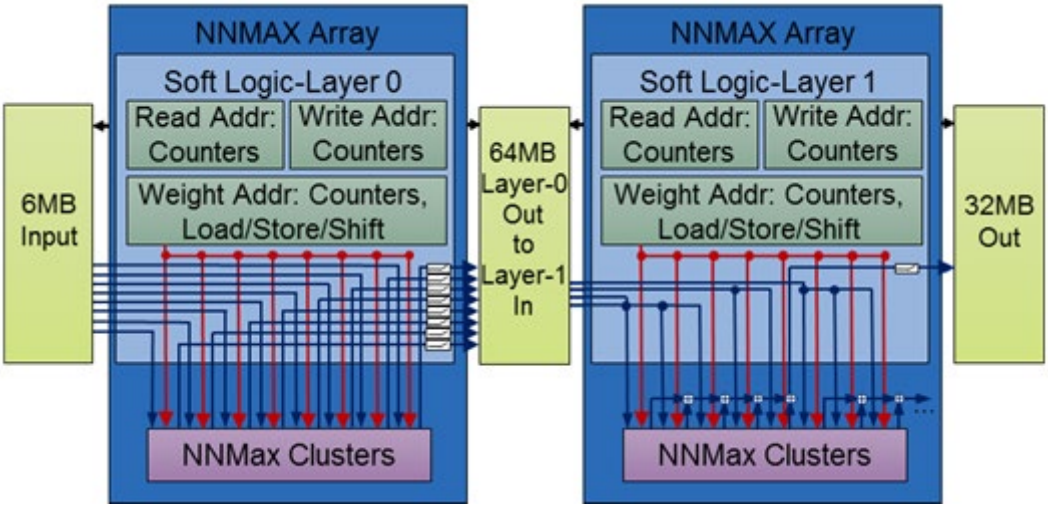

Figure 35: Example of InferX pipeline operations [37]

The coprocessor has an estimated power consumption of 4W and delivers approximately 8.5 TOPS that leads to an efficiency of 2.125 TOPS/W.

Regarding the software side, InferX X1 supports the TensorFlow Lite and ONNX models.

Co Processing Unit:

- 4K MACs in 2x2 NNMAX tiles (1000 MHz)
- 8 MB distributed SRAM
- 4 MB L3 SRAM
- LPDDR4 32-bit

Interfaces:

- PCIe Gen3 x4, 32x GPIO

Software:

- TensorFlow Lite, ONNX

Performance:

- 8.5 TOPS @ 4 W (est.)
- 2.125 TOPS/W (est.)

3.2.6 IP-Cores for ML-Acceleration

In the following subsections, the architecture and features of IP-cores targeting the acceleration of machine learning are presented. The selection comprises commercial designs as well as open-source architectures that can be used free of charge. Target technologies vary from FPGAs to ASICs; hence, also the achievable performance differs significantly.

3.2.6.1 NVIDIA Deep Learning Accelerator (NVDLA)

NVDLA is a free, open source and standardized architecture for accelerating deep learning inference.

NVDLA source code including documentation, HDL and software implementation can be found in the NVDLA Github group.

The NVDLA hardware design is implemented in Verilog. It is a modular architecture built from the following components [38]:

- Convolution Core — optimized high-performance convolution engine,
- Single Data Processor — single-point lookup engine for activation functions,
- Planar Data Processor — planar averaging engine for pooling,
- Channel Data Processor — multi-channel averaging engine for advanced normalization functions,
- Dedicated Memory and Data Reshape Engines — memory-to-memory transformation acceleration for tensor reshape and copy operations.

The accelerator design is parametrized — the above blocks can be independently configured, scaled or removed.

NVDLA can operate in two modes [38]:

- Headless — unit-by-unit management of the NVDLA hardware happens on the main system processor.
- Headed — delegates the high-interrupt frequency tasks to a companion microcontroller that is tightly coupled to the NVDLA sub-system.

The NVDLA connects with the rest of the system via:

- Configuration Space Bus (CSB) interface — synchronous, low-bandwidth, low-power, 32-bit control bus to be used by a CPU to access the NVDLA configuration registers (NVDLA functions as a slave).
- Interrupt interface — NVDLA includes a 1-bit level-driven interrupt that is asserted when a task has been completed or when an error occurs.
- Data Backbone (DBB) interface — connects NVDLA and the main system memory subsystems. It is a synchronous, high-speed, configurable data bus.

The NVDLA hardware is natively supported and handled by the TensorRT library.

3.2.6.1.1 Features

- Supported data types — binary, int4, int8, int16, int32, fp16, fp32 and fp64
- Supported image memory format — planar images, semi-planar images or other packed memory formats
- It can accept sparse convolution weights
- It can support Winograd convolution
- It can support batched convolution
- Configurable convolution buffer size, MAC array size

3.2.6.2 Versatile Tensor Accelerator (VTA)

VTA is a parameterizable accelerator that accelerates the bulk of the deep learning compute graphs [39]. It incorporates a simple processor that can perform dense linear algebra operations. It also adopts decoupled access-execute to hide memory access latency.

The code is available under the tvm-vta repository.

It consists of four modules:

- LOAD module — takes care of loading input and weights tensors from DRAM into data-specialized on-chip memories.
- STORE module — stores results produced by the compute core back to DRAM
- COMPUTE module — performs dense linear algebra computations with its GEMM core, and general computations with tensor ALU. It also loads data from DRAM into the register file, and loads micro-op kernels into the micro-op cache.
- INSTRUCTION FETCH module — loads instructions from DRAM and decodes them to route them into one of three command queues (for above-mentioned modules).

It also has 4 CISC instructions:

- LOAD — loads a 2D tensor from DRAM into buffers,
- GEMM — performs matrix-matrix multiplications,
- ALU — performs such operations as min, max, add, multiply shift,
- STORE — stores a 2D tensor from the output buffer to DRAM.

VTA is part of the Apache TVM (see Section 5.1), but can act as an independent deep learning accelerator.

3.2.6.3 AccDNN

AccDNN (Accelerator Core Compiler for Deep Neural Network) automatically converts deep neural networks, trained using Caffe, to RTL code that can be synthesized for FPGAs without additional FPGA design effort. The solution has been developed in cooperation between the University of Illinois, Urbana-Champaign and IBM. In their academic research, it is named DNNBuilder. The design generation is performed in three stages. First, the net structure is obtained from the Caffee net file, which is used as the basis for the implementation. In the second step, the RTL-design is generated, combining parameterizable Verilog models with the required on-chip memory for the weights. The library of Verilog models enables arbitrary quantization for weights and activations. In the final step, the RTL implementation is synthesized using the vendor-specific FPGA design tools. [40]

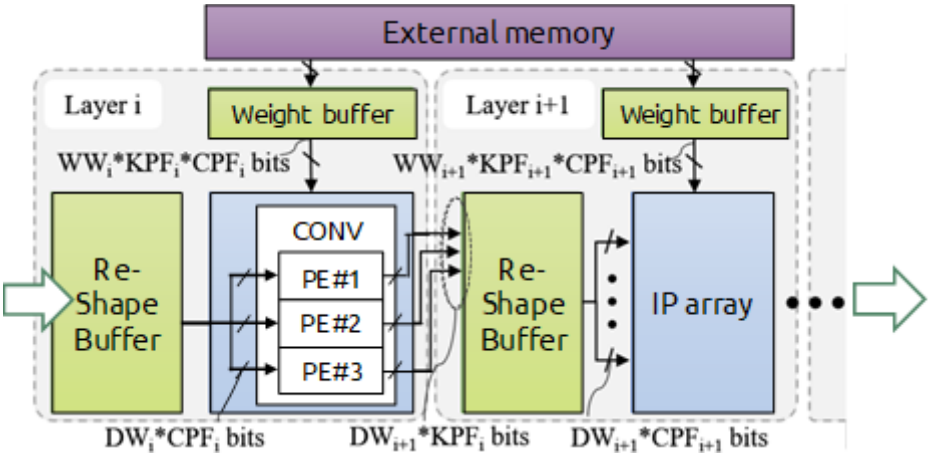

Figure 36: Example of an accelerator architecture generated by AccDNN [40]

In [40] a tool for automatic design space exploration is presented, which can be used for optimizing the large number of parameters that needs to be selected during architecture generation. The resulting structure is a pipeline, implementing a pipeline stage for each major layer (i.e., convolutional or fully connected layers). Other layers, like activation layers or batch normalization are aggregated to the major layers in order to reduce the number of required pipeline stages and thus minimize latency. An example, comprising two pipeline stages generated by AccDNN is depicted in Figure 36. [40]

Limitations
In its current version, AccDNN only supports models trained with the Caffee framework. Additionally, only convolutional layers, max pooling layers, fully connected layers, and batch normalization layers can be used. The total number of convolutional and fully connected layers in the network should be less than 15. [41]

Performance
In [40] various benchmarks have been implemented on different FPGAs. An implementation of YOLO9000, running on a Xilinx Zynq XC7Z45 FPGA with a clock frequency of 200MHz achieved a performance of 468 GOPS (Fix8) and 234 GOPS (Fix16), respectively. On a larger Xilinx Kintex UltraScale KU115 the performance increased to 4218 GOPS (Fix8) and 2109 GOPS (Fix16) for an automatically generated implementation running at a clock frequency of 220MHz.

Availability
https://github.com/IBM/AccDNN, Apache-2.0 License

3.2.6.4 VSORA AD1028

Combining a DSP and a deep learning accelerator, the AD1028 [42] from the french IP provider VSORA mainly targets autonomous driving. The combination of DSP and AI architecture eliminates the need for additional accelerators. Figure 37 gives a high-level overview of the architecture. Details about the internal structure are not disclosed. [42]

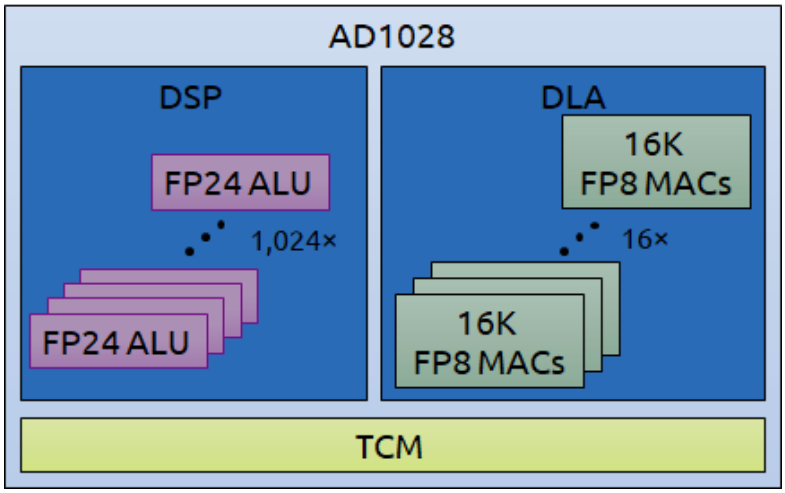

Figure 37: High-level architecture of the VSORA AD1028 [42]

The DSP integrates 1,024 24-bit floating-point ALUs (7 exponent bits, 16 mantissa bits, and 1 sign bit). The ALUs can be used in SIMD or MIMD mode and can execute several complex instructions per cycle. The DLA consists of a series of cores, each built up of 16k floating-point multiply-accumulate units (4 exponent bits, 3 mantissa bits, and 1 sign bit). In the AD1028, 16 cores are integrated, providing a total of 262,144 MAC units. Keeping the high number of ALUs and MACs utilized requires fast on-chip memory. Therefore, DSP and DLA are connected to a tightly coupled memory (TCM), which can be sized based on the application requirements. For performance and resource analyses, 105 MByte internal SRAM have been used. [42]

Development environment
For programming the architecture, C++ and Matlab are typically used for the DSP, while TensorFlow is supported for the DLA.

Architectural features

- DSP: 1024 ALUs in a single core
- DLA: 256k MACs divided into 16 cores
- TCM: User configurable memory size
- IEEE754 floating point with user selectable accuracy (exponent and mantissa)

Resource requirements and performance

- Size: 35 mm² for a 7 nm technology (logic only, excluding memory)
- DSP: 4 Tflops @ 2 GHz
- DLA: 1024 Tflops @ 2 GHz
- Power: 35 W
- 29 TOPS/W

3.2.6.5 Andes NX27V

The Andes NX27V [43] is a RISC-V single-issue five-stage microcontroller processor. It implements the RISC-V RV64 ISA with memory atomics, 16-bit compressed instructions, single-precision FPU, hardware multiplier/divider, user-level interrupts and the vector extension (A, C, F, M, N and V extension). Furthermore, it includes proprietary Andes

extensions. Use cases for the NX27V are digital baseband applications, image/vision processing and neural-network acceleration. [43]

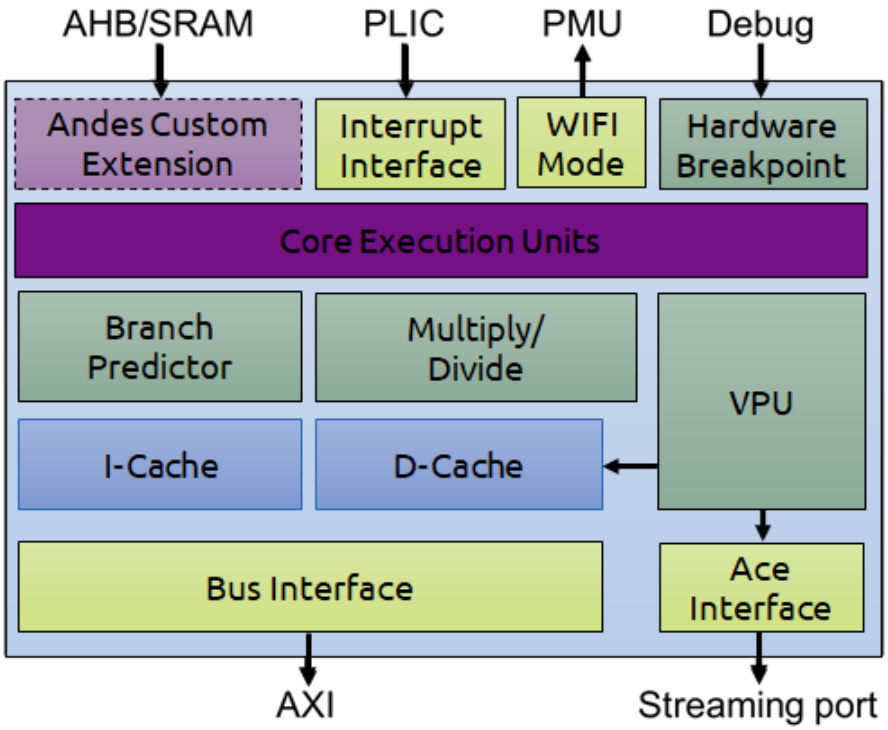

Figure 38: Architecture of the Andes NX27V CPU [43]

The vector processing unit can be configured to different bit widths to fit the given requirements. Possible bit widths are 128, 256 and 512 bits. In order to satisfy the high memory bandwidth demand of the VPU, different data paths are integrated. This includes direct access of the VPU to the shared D-Cache and to the external memory via AXI. In addition, Andes added the streaming interface Ace. In addition to single core implementations, the NX27V can also be configured as a multi core. [43]

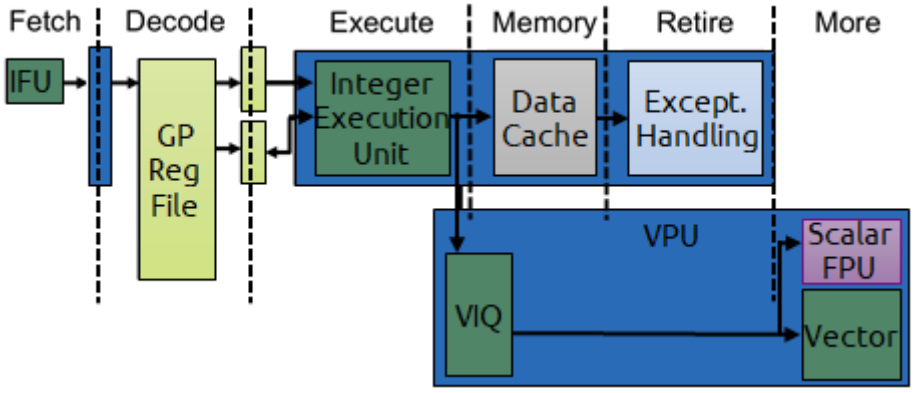

Figure 39: Integration of the vector pipeline into the NX27V scalar unit via the vector instruction queue (VIQ) [43]

Architectural Features

- Up to 512 bits RVV Vector Processing Unit
  - Out-of-order execution for the VPU
  - Proprietary extension with Bfloat16 and INT4 data types

    - Proprietary extension with the Ace streaming interface
- Configurable 8-64 KB I-Cache and D-Cache
- AXI-Interface
- Support for MXNet, Pytorch and Tensorflow

Performance

- Estimated with 7 nm TSCM: 1.5 GHz and 48 GFLOPS (FP32) [43]

The Andes NX27V is available as a proprietary IP core. The IP core includes the pre-integrated core with CPU subsystem including PLIC, Timer and Debug Module.

3.2.6.6 LeapMind Efficiera

The LeapMind Efficiera [43]is a deep-learning accelerator for binary neural networks. It consists of a configurable number of MAC-units, each capable of performing one multiply and accumulate operation of 1-bit weights and 2-bit activation values (w1a2 MAC operation) per cycle. Details about the internal architecture are not disclosed. LeapMind offers a variety of software tools to prepare the model for their accelerator. The main goal of these tools is to reduce the accuracy loss, which comes with the binarization of the NN-model. Therefore, the software tools include binary-aware training which uses binarized weights for training. Another possibility suggested by LeapMind to get a higher accuracy is to double the output channels, which on the other hand increases the computing time. To compensate for the additional computing time, LeapMind offers a technique called grouped convolution. Further methods to increase the efficiency of the architecture are developed by LeapMind. Users can decide which combination of tools they want to use for their application. [44]

Features

- Configurable number of w1a2 MAC units: 1024 or 4096 (39864 in development)
- AXI4 and AXI4-Lite interfaces for connection to the rest of the SoC
- Supported Framework: TensorFlow
- Software-Stack with different approaches to reduce accuracy loss of binarization
    - Binary-aware training
    - Double channel
    - Grouped convolution

Performance

- 1024 MAC units in TSMC 22ULP technology: 0.402 mm²; 533 MHz; 1.09 TOPS; 81 mW
- 4096 MAC units in TSMC 12FFC technology: 0.442 mm²; 800 MHz; 6.55 TOPS; 236 mW
- Intel Cyclone V and 4096 Mac units: 140 MHz; 1.1 TOPS

LeapMind Efficiera is available as a proprietary IP core.

3.2.6.7 Ceva NeuPro

The Ceva NeuPro [45] is a neural network processor targeting advanced driver-assistance systems, augmented-reality headsets, drones, smartphones and surveillance cameras. It combines a scalar unit and a vector unit in a supervisor role with the NeuPro Engine. Although it mainly runs the neural network control code, the scalar unit is capable of running layer functions which cannot be run by the NeuPro Engine. The main component of the

NeuPro Engine is the MAC array. It consists of a configurable number of 8x8-bit MAC units. In combination with a convolution control unit the MAC array can efficiently perform convolution calculations. Other parts of a neural network can be accelerated with additional units for accumulation, scaling, activation and pooling. A special feature of the NeuPro is the on-device retraining, which enables adjustments to the model weights without recompiling the model. [46]

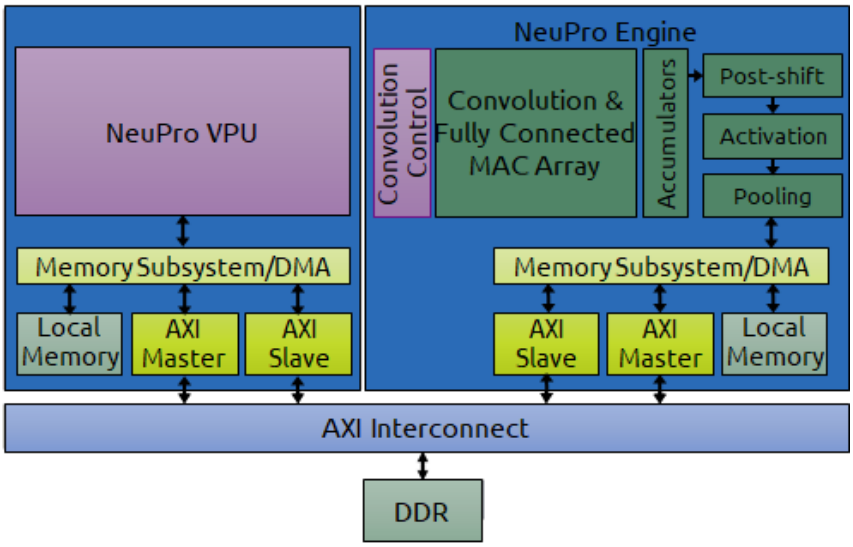

Figure 40: Architecture overview of the Ceva NeuPro deep-learning accelerator [45]

Architectural Features

- Scalar Unit including a VPU
- NeuPro Engine
  - MAC Array configurable with 512, 1024, 2048, 4096 8x8-bit MAC units
  - Additional hardware acceleration:
    - 32-bit Accumulator
    - Scaling unit
    - Activation unit for, e.g., ReLUs, parametric ReLUs, sigmoid, tangens hyperbolic
    - Pooling Unit for 2x2 and 3x3 average and maximum calculations
  - Mixed precision neural networks possible (8 and 16 bit)
  - On-device retraining

Performance

- 512 MAC units: 2.0 GHz; 2000 GOPS; 1024 GMACS/s
- 16 nm and 4096 MAC units: 1.53 GHz; 125000 GOPS; 6267 GMAC/s

The Ceva NeuPro ist available as a proprietary IP core.

3.2.6.8 Mipsology Zebra

Zebra by Mipsology is a compute engine for neural network inference. Mipsology claims that with Zebra the developers should be able to develop their solutions seamlessly without knowledge of underlying hardware technology, use of specific compilation tools or changes to the neural network, the training, the framework and the application.

Unfortunately, there is not much technical information available online.

Zebra is an IP that is to be deployed and operated on a FPGA. Mipsology claims that it can be deployed for efficient execution for the whole compute continuum from data centres to the edge. Zebra is available in the Amazon AWS marketplace.

Mipsology provides the integration for Caffe, Caffe2, Tensorflow and MXNet. A demo is available with pre-trained models for AlexNet and GoogLeNet for Caffe which is provided for the education and open-source community.

3.2.6.9 Ceva SensPro2

SensPro2 [47] combines NeuPro's deep learning acceleration capabilities with sensor fusion and computer vision features of Ceva's previous DSP and vision engines to create a high-performance architecture scalable for different sensor computations, deep learning inference and signal processing.

SensPro2 includes two vector units that can be configured in terms of MAC arrays and the capability of performing floating point operations as well as integer operations. SensPro2 can be further customized by allowing the user to add support for application-specific ISA extensions during IP configuration as well as adding custom scalar instructions using Ceva-Xtend.

Regarding performance, SensPro2 is expected to run at up to 1.6 GHz on a 7 nm technology, which can result in a 3.3 trillion INT8 operations per second for the largest configuration of SensPro2, SP1000.

Ceva SensPro2 IP core ships with a software development kit with libraries for deep learning, sensor-fusion and speech recognition. The SDK can support a variety of deep neural networks in TensorFlow Lite Micro format.

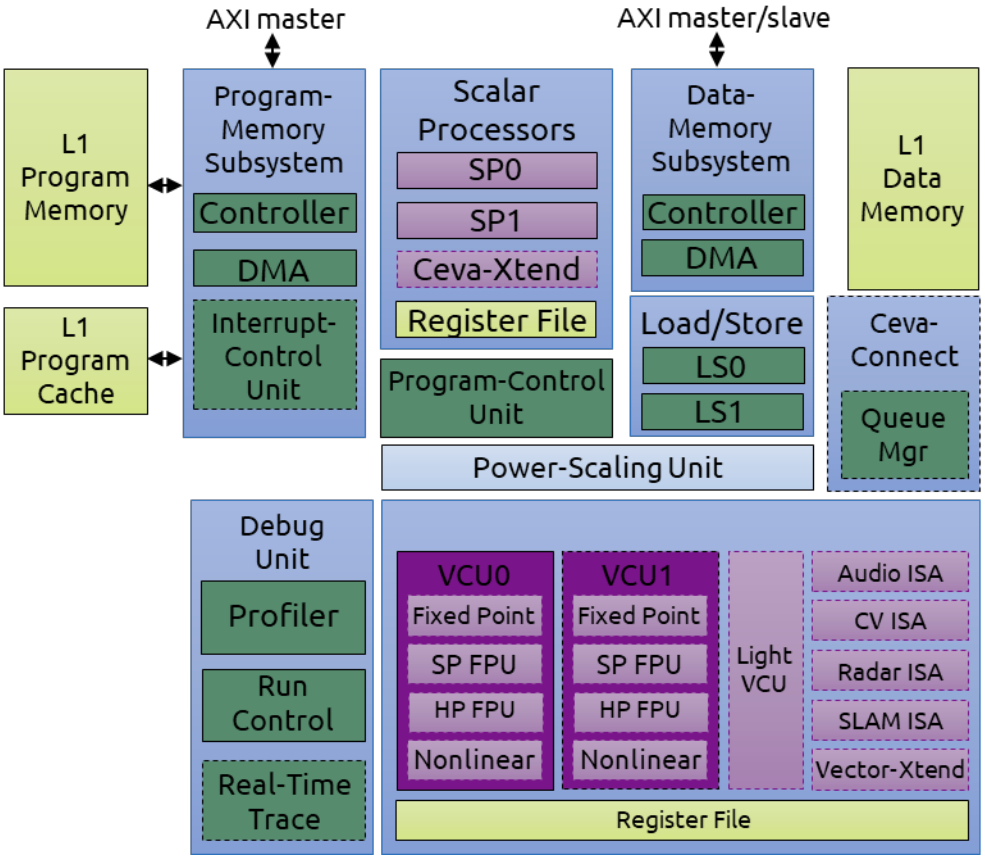

Figure 41: SensPro2 block diagram [44]

3.2.6.10 Xilinx DPU

Xilinx Deep Learning Processor (DPU) [48] is an IP core developed for accelerating convolutional neural networks inference. DPU can be implemented in programmable logic of Xilinx FPGAs and programmed to execute a deep neural network pre-compiled with custom DPU instructions. The custom instruction set of DPU supports many state-of-the-art convolutional networks.

An application processing unit is needed to invoke the DPU, facilitate data transfer and service interrupts.

DPU architecture varies based on the target Xilinx platform, but usually consists of a convolution engine of several processing elements that can be parallelized for higher performance. The architecture also includes instruction scheduler and on-chip buffer controller to control the on-chip BRAM.

The DPU is configurable in terms of the number of DPU cores in an IP package, the degree of parallelism in the convolution architecture, UltraRAM usage, DSP usage and capability to support depth-wise convolutions and activation functions.

Regarding performance, on a Zynq® UltraScale ZU7EV, with 2 DPUs of B4096 configuration, running at a rate of 333MHz, 2700GOPS peak theoretical performance can be achieved.

3.2.6.11 Xilinx FINN-R

Xilinx FINN-R is a framework for creating inference engines targeted for FPGAs. FINN-R can choose a suitable hardware implementation based on a given neural network, precision and resource constraints.

For a given neural network workload, FINN-R first chooses between a fully customised dataflow architecture and a multi-layer offload architecture. While the former architecture template lays out the workload on heterogeneous customised engines for each part of the neural network, the latter uses a more general compute array to calculate workload.

FINN-R then either tries to fold the network to fully use the engines in the dataflow architecture or generates a runtime schedule for the multilayer offload architecture. It also optimises the configuration of the accelerator for the given network.

The dataflow architecture optimised by FINN-R when benchmarked on an Ultra96 development board can perform 2318GOPS while running at 300MHz clock rate and consuming 10.7W of power. The benchmark was done using a binary 6-layer convolutional network [FINN-R18].

3.2.6.12 Think Silicon Neox V

Neox V [49] is a GPU architecture based on the RISC-V RV64C instruction set designed for machine learning, computer vision and video processing applications. Neox's multicore multithreaded GPU architecture licensable intellectual property can be configured to have 4 to 64 cores in clusters of 4, running at a rate of 800MHz, the 64-core configuration can reach 410 billion 16-bit floating point operations per second. Neox benefits from extended instruction set architecture that can support AI-specific instructions, which makes Neox a deep learning accelerator. Each core of Neox V can support eight 8-bit integer MAC operations, or sixteen 4-bit integer MAC operations, or four half-precision MAC operations in a single cycle [34].

The design comes with a custom software development kit, compiler and simulator from Think Silicon.

3.2.6.13 Imagination Technologies Series4

The Imagination Technologies Series4 deep learning accelerators [46] are targeting advanced driver-assistance systems and autonomous vehicles. The main components of the Series4 are 4096 8x8-bit MAC units. In combination with additional hardware accelerators for element-wise operations, activation functions, pooling, output formatting and rescaling, the Series4 is capable of accelerating neural networks. [46]

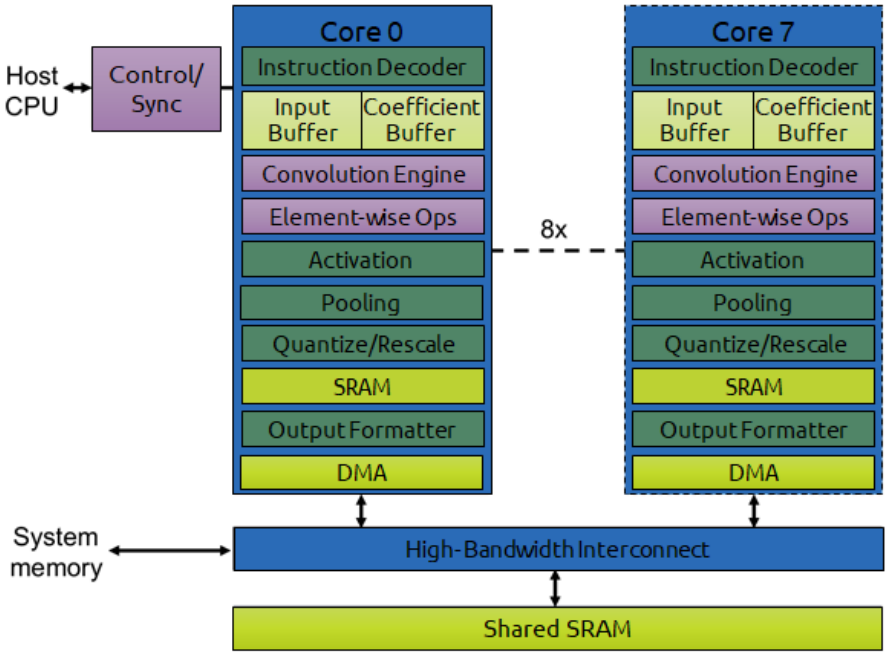

Figure 42: Architecture of the Imagination Technologies Series4 DLA [46]

The Series4 can be configured in a multicore system with up to eight similar cores. For data-transfer between the cores and to the shared SRAM, a high-bandwidth interconnect with a transfer-rate of up to 384GB/s is included. For the control tasks a third-party host CPU has to be added to the system. To efficiently distribute the computation between the cores, Imagination Technologies offers optimizations in the model compiler which group computations so that intermediate results do not have to be written to memory. This reduces DRAM transactions and therefore increases the efficiency. [46]

Architectural Features

- 4096 single-cycle 8-bit MAC units
  - Mixed precision possible (4/8/16-bit)
- Additional hardware accelerators:
  - Element-wise operation unit
  - Activation unit
  - Pooling Unit
  - Output formatting unit
  - Output Rescaling unit
- 256 KB to 64 MB SRAM
- 64 KB Input Buffer / 128 KB coefficient buffer per core
- Interconnect with 256 bits per cycle per core => 384 GB/s at 1.5 GHz
- Compiler
  - Supports layer merging and tiling
  - Distributes computation between the cores to reduce DRAM transactions
  - Enables running of different workloads in parallel

Performance

- 5 nm, single core: 1.53 GHz; 12.5 TOPS
- 5 nm, multicore (8 Cores): 1.53 GHz; 100.3 TOPS

The Imagination Technologies Series4 is available as a commercial IP core.

3.2.6.14 SiFive VIU7-256

The VIU7- 256 [50] is a RISC-V based 256-bit wide vector CPU IP-core that is sold by SiFive. It is using the older U7-series SiFive CPU and adds the 256-bit vector unit to it. It competes with CPUs like the ARM Cotex-A55 that has a 128-bit ARM NEON vector unit. Since the input of the vector unit is twice the size it outperforms the ARM processor by factor two on the same target frequency.

The vector unit supports 4x 64-, 8x32-, 16x16-, or 32x8-bit operations per cycle. It also supports 512-bit input vectors but then generates a solution every second cycle, which should increase utilization of the unit. Regarding cache sizes the IP-core is customizable to fit into the target architecture. For each core, the instruction and data caches can vary from 4KB up to 64 KB and the L2 cache can vary from 128 KB up to 1 MB.

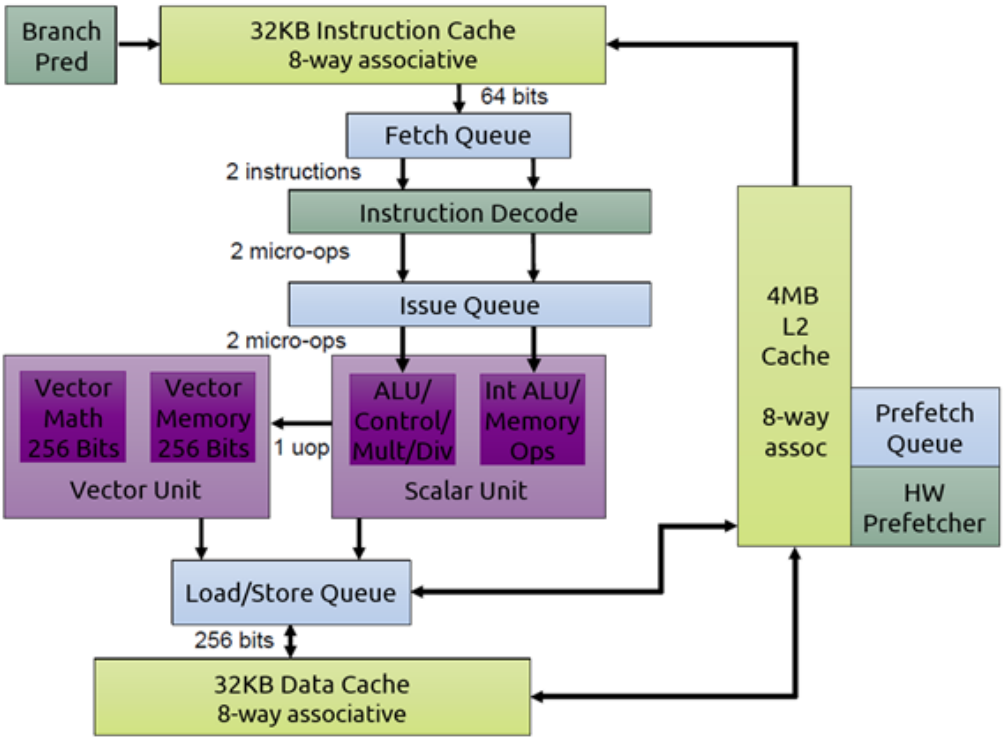

Figure 43: SiFive VIU7-256 microarchitecture [50]

SiFive reports 1800 MHz for TSMC 7 nm process, which leads to a performance of 29 Gflops for 32-bit floating point operations. In the same 7 nm process the ARM is rated at 2600 MHz having a performance of 21 Gflops. Showing that the RVV[i] enabled core is 38 % faster than the ARM core.

Utilizing the core in software is done by using RVV 1.0 instructions that are known by the RISC-V compiler.

3.2.6.15 SiFive VIS7

The VIS7 [51]is a larger variant of the VIU7-256, supporting real 512-bit operations. The basic difference is that it is designed for the SiFive S7 core which is more likely to be a microcontroller than a linux capable application processor the U7 is. Nevertheless, SiFive announces 64 Gflops for 32-bit floating point operations. It will use the same RVV 1.0 instruction set to be seamlessly integrated into the toolchain.

SiFive compares it to the ARM Cortex-R82 that also has the 128-bit NEON unit. For the S7 cores the performance is 30 % below the ARM cores but for the vector throughput the VIS7 is 4x more performant.

3.2.6.16 ARM Ethos-N78

The ARM Ethos-N78 [52] is a deep learning accelerator IP-core designed to fit the ARM Cortex-A series. The Cortex-A series is the larger application processor product from ARM. The Ethos is configurable, it can consist of 1 to 8 compute engines. Each compute engine has 4 MAC Engine arrays of dot product (DP) units organized as 8 x 16 INT8 MACs providing 128 MACs per array. As result, the total number of MACs can vary from 512 up to 4096. For a 7nm process, ARM declares the maximum frequency to be 1250 MHz, resulting in a maximum performance of 5.12 TOPS (INT8). In addition the Ethos-N78 provides Winograd convolution, which reduces the amount of operations, boosting the theoretical maximum performance to 10 TOPS.

The Programmable Layer Engine (PLE) is a Cortex-M CPU that mainly handles activation and pooling. It includes a 128-bit vector engine that matches to the output of the accumulators in the MAC arrays shifting the data into the SRAM holding the feature map.

The N78 supports multiple frameworks, Caffe, MXNet, ONNX, Pytorch and TensorFlow/-Lite. The Ethos compiler is reducing the computational load by pruning and clustering at compile time. It is also clustering sequential operations into one compute engine allowing the MAC Engine to reuse weights stored in the SRAM. [52]

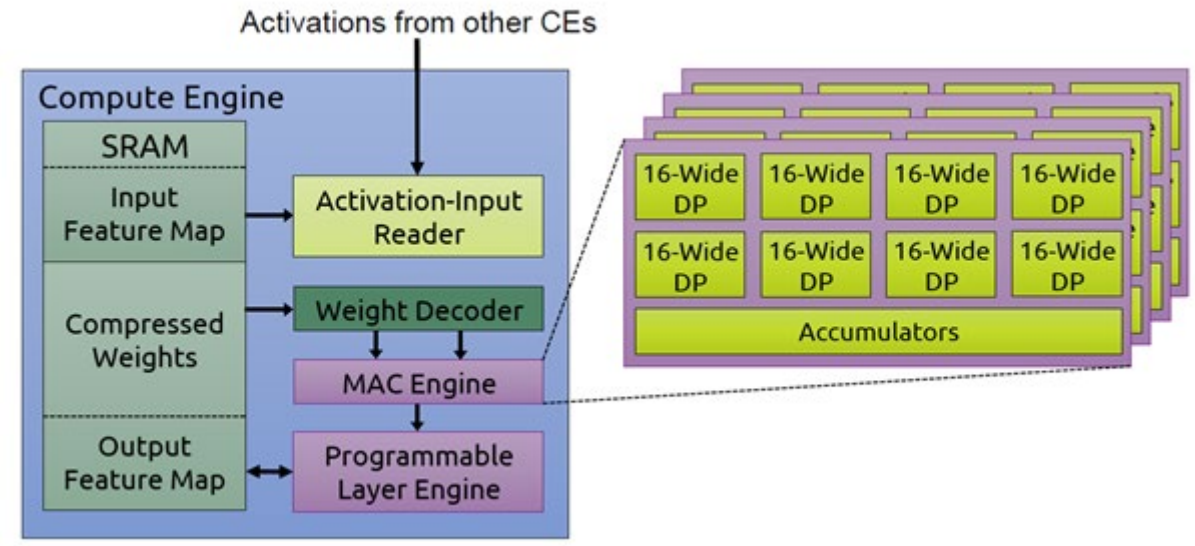

Figure 44: Ethos-N78 compute engine [52]

3.2.6.17 ARM Ethos-U55

The Ethos-U55 [53] is a deep learning accelerator IP-core that is designed to work with ARMs Cortex-M series, which is the smaller microcontroller product from ARM. The U55 is configurable regarding the number of MACs, they can be scaled from 32 units up to 256 INT8 MACs. It is optimized for the newer Cortex-M55 microcontroller that has the AXI processor bus. It is a small accelerator that targets ULP applications.

The ARM Cortex-M toolchain is supporting TensorFlow Lite. It identifies AI operations and maps it to the Ethos-U55.

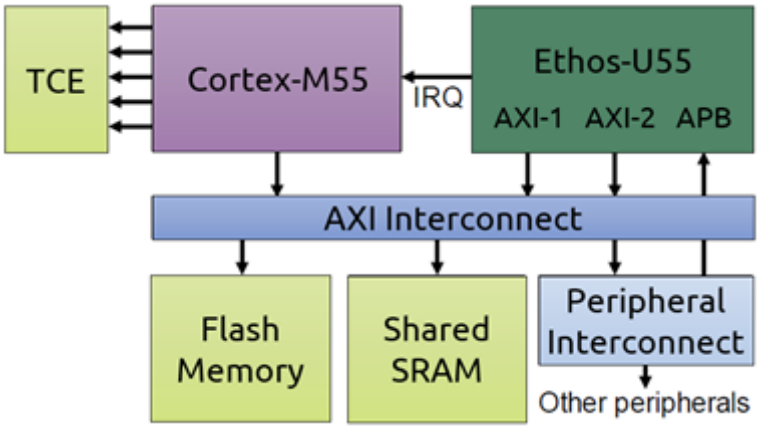

Figure 45: Cortex M55 pipeline with Helium [53]

3.2.6.18 ARM Cortex-A55

The ARM Cortex-A55 is not a dedicated AI accelerator like the other IP-cores in this Chapter. It is included because many companies refer the performance and energy efficiency of the A55 as reference for low-power inference. The A55 is the successor of the Cortex-A53 that is well known from modern mobile devices and embedded compute platforms. It increases the performance by 15 % to 20 % compared to the old version [54]. It has 8 pipeline stages and supports only in-order execution, which results in less chip area compared to out-of-order architectures. [54]

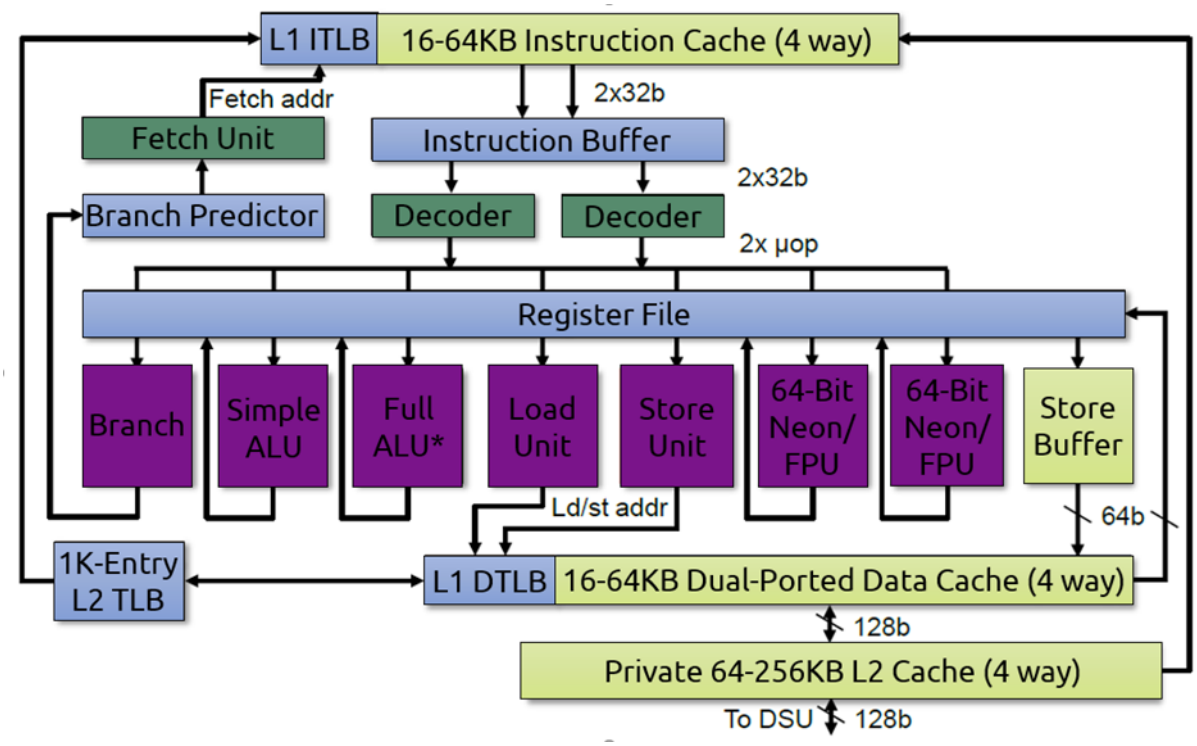

Figure 46: Architecture of the ARM Cortex-A55 [51]

Instruction and data cache can be configured from 16 KB up to 64 KB by the designer. In the A55 the data cache is dual-ported which is a major advantage compared to the A53, because it improves latency and usable bandwidth of the memory. Additionally, L2 cache can be configured from 64 KB up to 256 KB and it features the new DynamIQ architecture changing the L2 cache to be part of the core [55] Allowing it to run at CPU frequency and reduce latency from 13 cycles to 6 cycles, which gives the system a significant higher performance. The A55 has two ALUs, one is a simple one that doesn't feature dot operations, the second one is a Full ALU that supports all integer operations. They are running in parallel so that two operations per cycle can be calculated. For floating point operations, a 128-bit NEON unit is integrated that can process eight FP16 operation simultaneously. In addition, it supports the ARM dot-product instructions which provide up to 4x higher performance on CNN operations making the A55 suitable for small neural networks [56].

3.2.7 Comparison of AI Accelerators

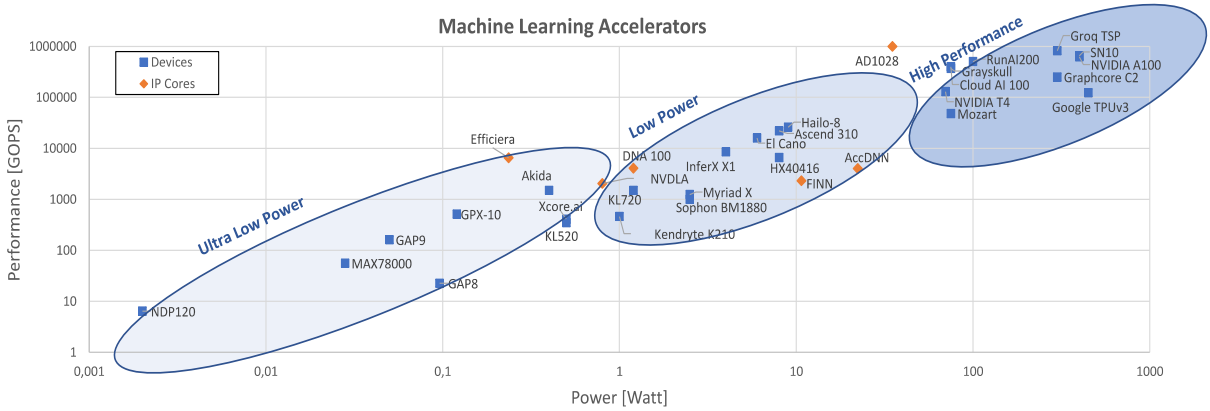

Figure 47: Overview of Machine Learning Accelerators

Since VEDLIoT focuses on edge computing, mainly ultra-low power and low power accelerators for machine learning have been presented in this Chapter. Figure 47 summarizes the different architectures, extended by various high-performance accelerators. Performance is given in INT8 GOPS, when available. If only Float is supported, the highest reported value for GOPS is used. Hence, the figure currently only gives a rough overview; a detailed comparison is provided in Chapter 3.3, where a subset of the listed hardware was benchmarked using realistic workloads. On average over all architectures, a power efficiency in the order of 1 TOPS/W is achieved (represented by the diagonal of the chart).

3.3 Accuracy Results

The accuracy measurement is the most important novelty in D3.3 compared to D3.1. During the evaluation of performance, that was reported in D3.1, it turned out that there is no possibility to have a uniform compiler for all platforms analysed. As result, vendor specific DL compilers were used for each platform, which leads to an uncertainty regarding comparability of the performance results.

In order to verify the results of the performance evaluation, this chapter reports the accuracy for each architecture. The assumption is, that if the accuracy of the DL models compiled by different DL compilers is similar, then the benchmark results are comparable.

The accuracy was measured for the three DL models that were used for the performance evaluation (ResNet50, MobileNetV3small, YoloV4). The metrics for the classification models ResNet50 and MobileNetV3small are Top1 and Top5 as described in section 3.1.4.1 and the metrics for the object detection model YoloV4 are mAP(.50) and mAP(.50:.95) as described in section 3.1.4.2.

In the case of NVIDIA devices, results could be combined to two classes. The class 'NVIDIA Volta' shows the accuracy of Volta based platforms together with all older NVIDIA architectures. The class 'NVIDIA Ampere' shows the results for Ampere based platforms only, because there were slight differences in some of the measurements. The class 'OpenVINO' combines all platforms that use OpenVINO as DL compiler (x86 and Myriad), as well as the class 'Xilinx' combines all Xilinx devices. The 'TVM' classes 'Ampere' and 'Volta' show the results for NVIDIA Jetson AGX Orin (Ampere) and NVIDIA Jetson AGX Xavier (Volta) that were analysed using the open-source compiler ApacheTVM. The differences of the results of FP16 and FP32 precision are low enough to be combined in the following charts to 'Floating Point Accuracy'. The INT8 results are named 'Integer Accuracy'.

3.3.1 ResNet50

The accuracy for ResNet50 shows that the floating-point results are very close, even the open-source compiler ApacheTVM shows similar accuracy. For the integer accuracy the results are a bit more mixed, the Volta based NVIDIA platforms have lower accuracy than the Ampere based ones as well as the open-source compiler results. Xininx, Coral, Hailo-8 and i.MX8 use vendor specific models that are optimized for INT8 operation. In some cases they show even better results than the floating-point implementation used for evaluation.

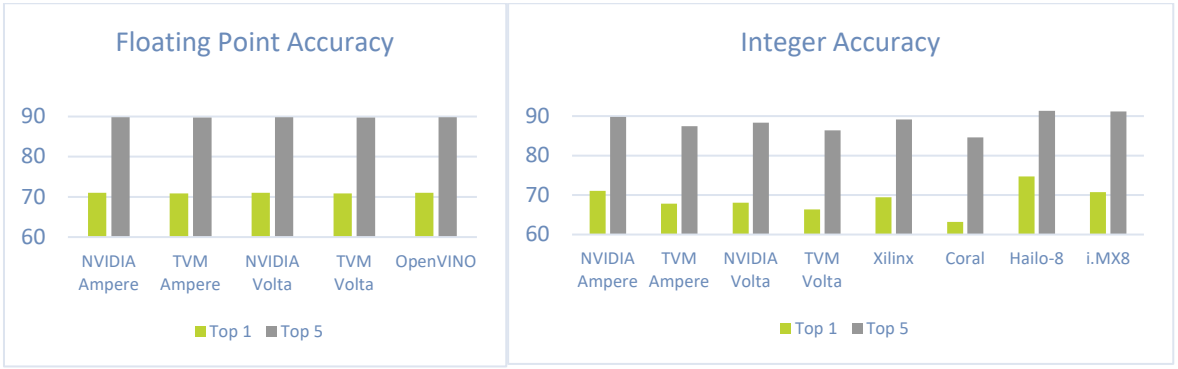

Figure 48: ResNet50 Accuracy Results

For validation of the accuracy results, Kenning was used to generate Confusion-Matrices. Since the model is trained for 1000 classes these matrices are too large to be depicted in this document.

3.3.2 MobileNetV3small

The accuracy for MobileNetV3small shows that the floating-point results are depending on the used DL compiler. The open-source compiler ApacheTVM delivered the best results, followed by OpenVINO. Looking at the integer results, the NVIDIA compiler performs better than ApacheTVM. The best result shows the Xilinx FPGA implementation. Unfortunately, Hailo-8 didn't provide the correct model, so the results are not shown here.

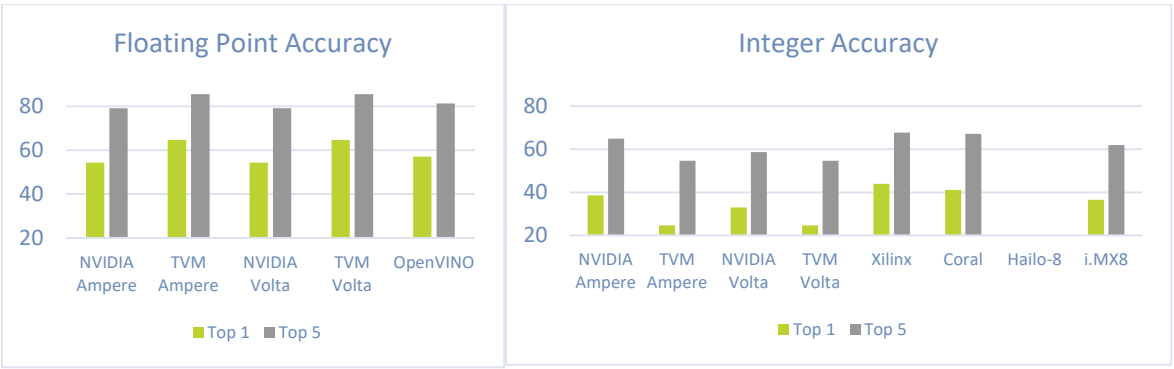

Figure 49: MobileNetV3 small Accuracy Results

For validation of the accuracy results, Kenning was used to generate Confusion-Matrices. Since the model is trained for 1000 classes these matrices are too large to be depicted in this document.

3.3.3 YoloV4

The accuracy for YoloV4 shows that the floating-point results are very close, even the open-source compiler ApacheTVM shows similar accuracy. For integer accuracy it wasn't possible to quantize the model correctly using ApacheTVM and TensorFlowLite (i.MX8). The Coral was simply too small to fit the YoloV4 model on it. NVIDIA and Hailo-8 have similar results compared with each other and the floating-point accuracy.

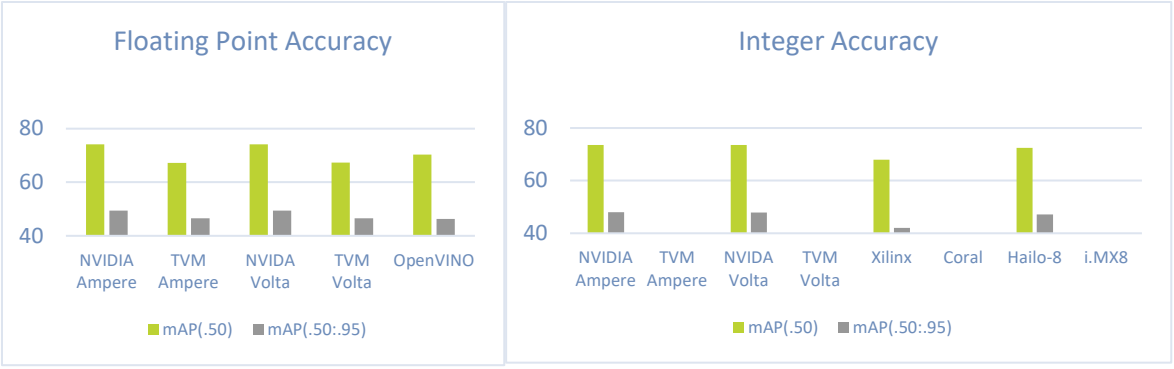

Figure 50: YoloV4 Accuracy Results

3.3.3.1 Recall-Precision Gradients

In contrast to the classification models (ResNet50 and MobileNetV3small), which can classify 1000 classes, the object detection model YoloV4 is trained for 80 classes only. That's why it

was chosen for the report of a per-class comparison, providing in-depth knowledge about similarity of results from the different architectures and toolchains.

The Kenning toolbox was feed with the detection outputs of the accuracy evaluation. Kenning rendered multiple graphs showing mAP, Histogram of True Positive IoU, Recall-Precision curves and Recall-Precision gradients. Examples of all graphs have been shown in the APPENDIX of D3.1. Comparing all graphs it was decided to present the Recall-Precision gradients here, because they allow a per-class comparison of the accuracy measurements.

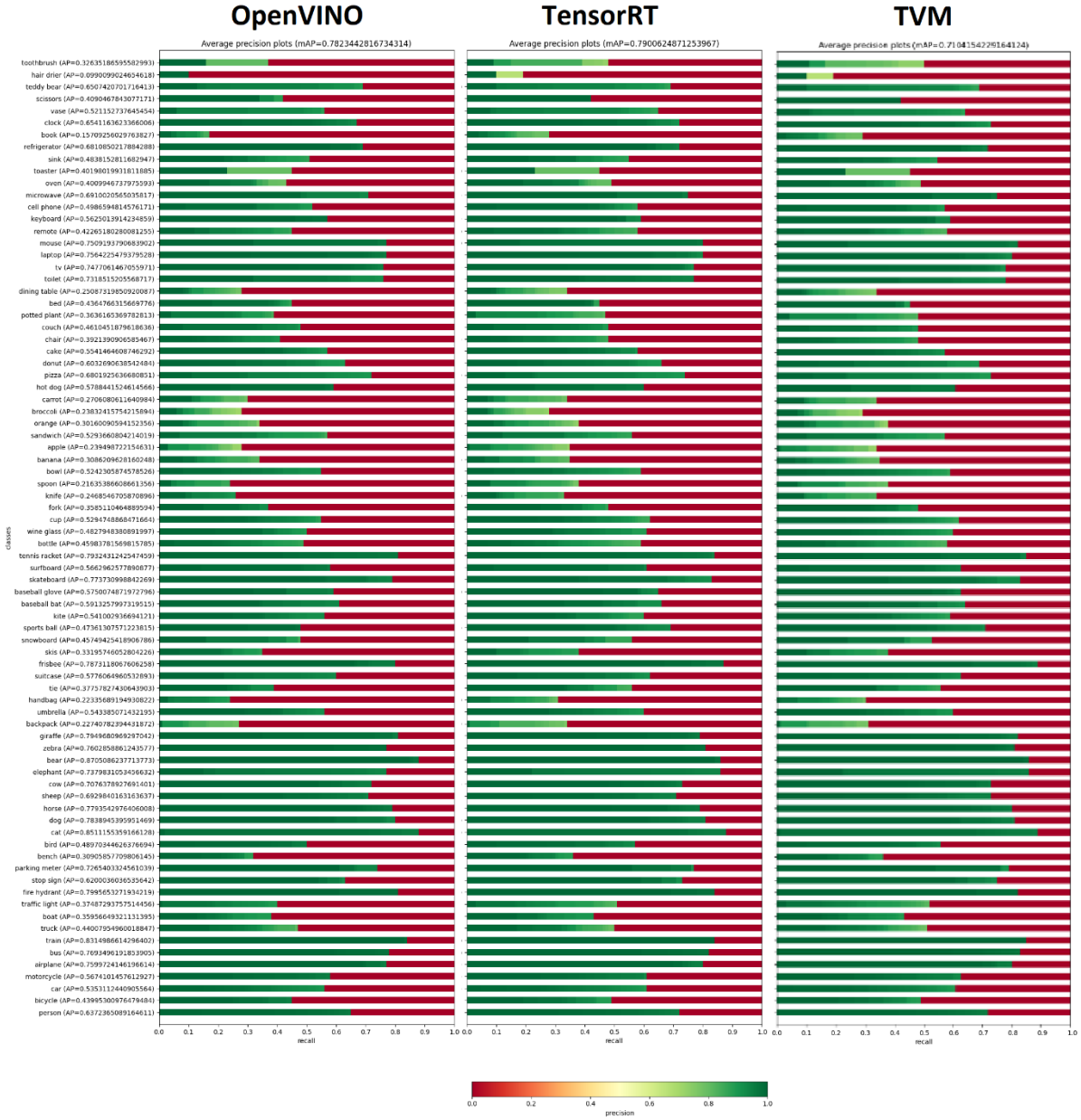

Figure 51: Recall-Precision gradients per class for YOLOv4 with FP32 precision comparing OpenVino, TensorRT and TVM results

Figure 51 shows the Recall-Precision gradients for FP32 precision. The precision per detection is indicated by color, dark green indicates a precision of 100 % and red a precision of 0 %. This is an mAP(.50) measurement which means that a positive detection needs to have a minimum intersection over union (IoU) of 50 %, everything below will be marked as

false detection with a precision of 0 %. That's why the orange color is never used in this graph. The recall indicates how many of the objects, that are present in the dataset, have been detected, e.g. class 'hair drier' in the OpenVINO plot has a recall of 0.099, showing that only ~10 % of all 'hair drier' have been detected.

The floating-point accuracy results for OpenVINO, TensorRT and ApacheTVM, which are shown in Figure 51, are very close together. Only some classes, e.g. 'toothbrush' have minor deviations. Over all it is expected that the performance results are comparable.

Figure 52: Recall-Precision gradients per class for YOLOv4 with INT8 precision comparing TensorRT, Hailo-8 and Xilinx results

Looking at the integer accuracy in Figure 52 the deviations are larger. Especially the Hailo-8 has a well-trained INT8 model that in some cases delivers better results than the floating-point accuracy depicted in Figure 51. The TensorRT and Xilinx accuracy is comparable to the floating-point accuracy and therefore INT8 results are also comparable.

This validates that the performance analysis was done on real data showing frames per second that can be achived in the use-cases. The results will be used in the VEDLIoT project as basis for hardware decisions.

3.4 Evaluation Results

3.4.1 x86 Baseline

The standard compute architecture in today's data centers are classic x86 based processors. In the VEDLIoT project there are several x86 based COM-Express microserver integrated into the RECS architecture. For the baseline evaluation two microservers have been choosen, a Xeon-D 1577 from Intel and an Epyc 3451 from AMD. Both are equipped with 16 pyhsical processor cores and due to Hyper-Threading with 32 logical processor cores. The AMD is 2 years younger. It has a newer technology and has a higher base frequency. Nevertheless, the Intel CPU performs quite well and seems to be comparable to the AMD Epyc.

ADLINK COM-Express Xeon-D 1577

- ADLINK Express-BD7-D1577
  - Intel Xeon-D 1577
    - 16 cores
    - 32 threads
    - 1.3 GHz base frequency
    - 24 MB cache
  - 64 GB DDR4-2133 dual-channel
  - 512 GB SATA SSD HDD
- Module Cost: 2959.06 € (Mouser)

Congatec COM-Express EPYC 3451

- Congatec conga-B7E3/3451
  - AMD EPYC Embedded 3000 Model 3451
    - 16 cores
    - 32 threads
    - 2.1 GHz base frequency
    - 32 MB cache
  - 96 GB DDR4-2666 quad-channel (3 channels used)
  - 256 GB NVMe SSD HDD
- Module Cost: 1343.46 $ (Digikey)

3.4.1.1 Peak Performance

The maximum performance for x86 processors running inference can be achieved by utilizing the AVX2 SIMD units of the cores. Both processors are equipped with AVX2 [57] units that are capable of 32 32-bit operations per second or 64 16-bit operations per second. The processor has one AVX2 unit per physical core, so each processor has 16 AVX2 unit. Table 2 shows the calculated theoretical maximum performance of the x86 processors for 16-bit and 32-bit floating point operations.

Table 2: Theoretical maximum performance of x86 processors

Processor	FP16	FP32

Xeon-D 1577	1331.2 GOPS	665.6 GOPS
EPYC 3451	2191.36 GOPS	1095.68 GOPS

3.4.1.2 Toolflow

For the performance evaluation of the x86-based microservers, version 2021.4.1 of the Intel OpenVINO toolkit [58] was used. The evaluation was performed using the benchmarking app shipped with OpenVINO. Using this application, it was possible to profile the latency and throughput of the different onnx models from the VEDLIoT model chosen in Chapter 3.1.2. Furthermore, by modifying the configuration parameters, the batch size and quantization of the target model could be adjusted without the need for additional tools.

Power consumption of the microservers was measured using the WebGUI-based management interface of the RECS. Peak memory usage was determined using the *htop* application, an interactive process manager.

3.4.1.3 Results

Table 3: ResNet50 Performance (Xeon-D 1577)

Performance	FP16		FP32	
Batchsize	1	8	1	8
Inference Time [ms]	20.460	111.790	20.470	114.650
Achieved performance [Inferences/s]	48.88	71.56	48.85	69.78
Achieved performance [GOPS]	380.25	556.76	380.07	542.87
Peak performance [GOPS]	1331.2		665.6	
Performance Ratio	28.56 %	41.82 %	57.10 %	81.56 %
Cost Metrics				
Memory Utilization [GB]	1023 MB	1.18 GB	1023 MB	1.20 GB
Power [W]	56.960	60.240	58.490	60.320
Idle Power [W]	14.030	14.030	14.030	14.030
Energy / Inference [mJ]	1165.40	841.78	1197.29	864.46
GOPS / W	6.68	9.24	6.50	9.00

Table 4: ResNet50 Accuracy (Xeon-D 1577)

Classification accuracy	FP32	FP16
Top 1	66.85 %	66.86 %
Top 5	87.30 %	87.31 %

Table 5: MobileNetV3 Small Performance (Xeon-D 1577)

Performance	FP16		FP32	
Batchsize	1	8	1	8
Inference Time [ms]	1.710	4.660	1.710	4.710
Achieved performance [Inferences/s]	584.80	1716.74	584.80	1698.51
Achieved performance [GOPS]	104.09	305.58	104.09	302.34
Peak performance [GOPS]	1331.2		665.6	
Performance Ratio	7.82 %	22.96 %	15.64 %	45.42 %
Cost Metrics				
Memory Utilization [GB]	811 MB	869 MB	809 MB	869 MB
Power [W]	45.840	58.450	46.130	59.350
Idle Power [W]	14.030	14.030	14.030	14.030
Energy / Inference [mJ]	78.39	34.05	78.88	34.94
GOPS / W	2.27	5.23	2.26	5.09

Table 6: MobileNetV3 Small Accuracy (Xeon-D 1577)

Classification accuracy	FP32	FP16
Top 1	57.12 %	57.13 %
Top 5	81.40 %	81.39 %

Table 7: YoloV4 (Xeon-D 1577)

Performance	FP16		FP32	
Batchsize	1	8	1	8
Inference Time [ms]	130.77	953.56	132.15	968.65
Achieved performance [Inferences/s]	7.65	8.39	7.57	8.26
Achieved performance [GOPS]	461.88	506.73	457.06	498.84
Peak performance [GOPS]	1331.2		665.6	
Performance Ratio	34.70 %	38.07 %	68.67 %	74.95 %
Cost Metrics				
Memory Utilization [GB]	1.49 GB	2.01 GB	1.32 GB	2.01 GB
Power [W]	57.24	60.08	58.12	60.15
Idle Power [W]	14.030	14.030	14.030	14.030
Energy / Inference [mJ]	7485.27	7161.24	7680.56	7283.04
GOPS / W	8.07	8.43	7.86	8.29

Table 8: mAP accuracies for YoloV4 (Xeon-D 1577)

Classification accuracy	FP16	FP32
mAP (.50)	46.3 %	46.3 %
mAP (.50:.95)	70.3 %	70.3 %

Table 9: ResNet50 (EPYC 3451)

Performance	FP16		FP32	
Batchsize	1	4	1	4
Inference Time [ms]	21.110	59.952	21.320	59.814
Achieved performance [Inferences/s]	47.37	66.72	46.90	66.87

Achieved performance [GOPS]	368.55	519.08	364.92	520.28
Peak performance [GOPS]	2191.36		1095.68	
Performance Ratio	16.82 %	23.69 %	33.30 %	47.48 %
Cost Metrics				
Memory Utilization [GB]	1.01 GB	1.43GB	1.02 GB	1.43GB
Power [W]	91.520	96.850	92.550	98.540
Idle Power [W]	24.760	24.760	24.760	24.760
Energy / Inference [mJ]	1931.99	1451.59	1973.17	1473.52
GOPS / W	4.03	5.36	3.94	5.28

Table 10: ResNet50 Accuracy (Xeon-D 1577)

Classification accuracy	FP32	FP16
Top 1	66.85 %	66.86 %
Top 5	87.30 %	87.31 %

Table 11: MobileNetV3 Small (EPYC 3451)

Performance	FP16		FP32	
Batchsize	1	4	1	4
Inference Time [ms]	2.150	2.743	2.160	2.750
Achieved performance [Inferences/s]	465.12	1458.26	462.96	1454.55
Achieved performance [GOPS]	82.79	259.57	82.41	258.91
Peak performance [GOPS]	2191.36		1095.68	
Performance Ratio	3.78 %	11.85 %	7.52 %	23.63 %
Cost Metrics				
Memory Utilization [GB]	796 MB	936MB	796 MB	935MB

Power [W]	79.380	94.380	78.490	94.830
Idle Power [W]	24.760	24.760	24.760	24.760
Energy / Inference [mJ]	170.67	64.72	169.54	65.20
GOPS / W	1.04	2.75	1.05	2.73

Table 12: MobileNetV3 Small Accuracy (EPYC 3451)

Classification accuracy	FP32	FP16
Top 1	57.12 %	57.13 %
Top 5	81.40 %	81.39 %

Table 13: YoloV4 (EPYC 3451)

Performance	FP16		FP32	
Batchsize	1	4	1	4
Inference Time [ms]	135.94	490.384	137.19	491.962
Achieved performance [Inferences/s]	7.36	8.16	7.29	8.13
Achieved performance [GOPS]	444.31	492.68	440.27	491.09
Peak performance [GOPS]	2191.36		1095.68	
Performance Ratio	20.28 %	22.48 %	40.18 %	44.82 %
Cost Metrics				
Memory Utilization [GB]	1.47 GB	2.42GB	1.30 GB	2.3GB
Power [W]	102.3	100.2	103.4	103.3
Idle Power [W]	24.760	24.760	24.760	24.760
Energy / Inference [mJ]	13906.66	12284.12	14185.45	12704.92
GOPS / W	4.34	4.92	4.26	4.75

Table 14: mAP accuracies for YoloV4 (EPYC 3451)

Classification accuracy	FP16	FP32

mAP (.50)	46.3 %	46.3 %
mAP (.50:.95)	70.3 %	70.3 %

3.4.2 Intel Myriad X

The PCIe M.2 based accelerator was attached to a NVIDIA Jetson TX2 evaluation platform via an adapter to fit on the standard PCIe x4 connector. The specific hardware used was an AAON AI Core XM2280 module. It has a PCIe to USB chip and attaches two Intel Myriad X chips to the host system.

NVIDIA Jetson TX2 Evaluation Platform + Intel Movidius Myriad X

- NVIDIA Jetson TX2
  - Dual-Core NVIDIA Denver 2 64-Bit CPU
  - Quad-Core ARM Cortex-A57 MPCore
  - 8 GB DDR4-1866 dual-channel
  - 32 GB eMMC HDD
  - Cost: 575 €
- AAON AI Core XM2280
  - 2x AAON AI Core XM2280
  - M.2 2280 B+M key
  - Cost: 163 €

3.4.2.1 Peak Performance

The theoretical maximum performance is claimed to be 4000 GOPS. The smallest datatype supported by the Myriad is INT4. A trivial approximation of the peak performance is calculated by, e.g. dividing the INT4 performance by 4 to get INT16 / FP16 peak performance. This approximation leads to 1000 GOPS for 16-bit operations and 500 GOPS for 32-bit operations.

3.4.2.2 Toolflow

For this evaluation, the M.2-based Intel Myriad VPU was installed in an NVIDIA Jetson TX2 module. The performance analysis was performed using version 2021.4.1 of the Intel OpenVINO toolkit [58]. The benchmarking app shipped with OpenVINO was used to perform the evaluation. Using this application, it was possible to profile the latency and throughput of the different onnx models from model set v1. Furthermore, by modifying the configuration parameters, the batch size and quantization of the target model could be adjusted without the need for additional tools.

Power consumption of the Intel Myriad and its host module was measured using a Tektronix MDO4054B oscilloscope.

3.4.2.3 Results

Table 15: ResNet50 (Myriad)

Performance	FP16		FP32	
Batchsize	1	4	1	4
Inference Time [ms]	49.430	141.041	49.560	136.188

Achieved performance [Inferences/s]	20.23	28.36	20.18	29.37
Achieved performance [GOPS]	157.39	220.65	156.98	228.51
Peak performance [GOPS]	1000		500	
Performance Ratio	15.74 %	22.06 %	31.40 %	45.70 %
Cost Metrics				
Memory Utilization [GB]				
Power [W]	7.750	8.600	7.800	8.600
Idle Power [W]	5.850	5.850	5.850	5.850
Energy / Inference [mJ]	383.08	303.24	386.57	292.80
GOPS / W	20.31	25.66	20.13	26.57

Table 16: ResNet50 Accuracy (Myriad)

Classification accuracy	FP32	FP16
Top 1	66.96 %	66.96 %
Top 5	87.32 %	87.33 %

Table 17: MobileNetV3 Small (Myriad)

Performance	FP16		FP32	
Batchsize	1	4	1	4
Inference Time [ms]	19.450	30.367	19.510	30.468
Achieved performance [Inferences/s]	51.41	131.72	51.26	131.29
Achieved performance [GOPS]	9.15	23.45	9.12	23.37
Peak performance [GOPS]	1000		500	
Performance Ratio	0.91 %	2.34 %	1.82 %	4.67 %
Cost Metrics				

Memory Utilization [GB]				
Power [W]	7.200	7.850	7.200	7.850
Idle Power [W]	5.850	5.850	5.850	5.850
Energy / Inference [mJ]	140.04	59.60	140.47	59.79
GOPS / W	1.27	2.99	1.27	2.98

Table 18: MobileNetV3 Small Accuracy (Myriad)

Classification accuracy	FP32	FP16
Top 1	57.07 %	57.05 %
Top 5	81.35 %	81.35 %

Table 19: YoloV4 (Myriad)

Performance	FP16		FP32	
Batchsize	1	4	1	4
Inference Time [ms]	312.51	1019.226	332.77	1035.17
Achieved performance [Inferences/s]	3.20	3.92	3.01	3.86
Achieved performance [GOPS]	193.27	237.04	181.51	233.39
Peak performance [GOPS]	1000		500	
Performance Ratio	19.33 %	23.70 %	36.30 %	46.68 %
Cost Metrics				
Memory Utilization [GB]				
Power [W]	7.9	9.1	7.95	9.15
Idle Power [W]	5.850	5.850	5.850	5.850
Energy / Inference [mJ]	2468.83	2318.74	2645.52	2367.95
GOPS / W	24.47	26.05	22.83	25.51

Table 20: mAP accuracies for YoloV4 (Myriad)

Classification accuracy	FP16	FP32
mAP (.50)	46.3 %	46.3 %
mAP (.50:.95)	70.3 %	70.3 %

3.4.3 Google Coral TPU (M.2)

For this evaluation, the M.2-based Google Coral TPU was installed in an NVIDIA Jetson Xavier NX evaluation platform. The Coral TPU has native PCIe support for Gen.2 x1, it is directly connected to the Xavier NX.

NVIDIA Jetson Xavier NX Evaluation Platform + M.2 Google Coral TPU

- NVIDIA Jetson Xavier NX
  - NVIDIA Carmel ARM v8.2 64-Bit-CPU, 6 Cores
  - NVIDIA GPU 384 Volta Cores + 48 Tensor Cores
  - 2x NVIDIA DLA-Engines
  - 8 GB DDR4-1866 dual-channel
  - Cost: 519 €
- M.2 Google Coral TPU
  - 1x Google Coral TPU
  - Cost: 55 €

3.4.3.1 Peak Performance

The theoretical maximum performance is rated at 4 TOPS for INT8 operations.

3.4.3.2 Toolflow

The performance analysis was performed using the publicly available Coral examples [59] in combination with the .pb models from the VEDLIoT model set. For use in the example, the models first had to be converted to tflite, which was done using TensorFlow and afterward, optimized for the Coral Edge TPU, using the Edge TPU Compiler [60]. These evaluations were performed using Edge TPU Compiler version 16.0.384591198.

The power consumption of the Google Coral TPU and its host module was measured using a Tektronix MDO4054B oscilloscope.

3.4.3.3 Results

Table 21: INT8 (Coral M.2)

Performance	ResNet50	MobileNet	YoloV4
Inference Time [s]	41.07	3.88	-
Latency [s]	42.07	4.51	-

Achieved performance [Inferences/s]	24.35	257.73	-
Achieved performance [GOPS]	189.43	45.88	-
Peak performance [ops/s]	4000	4000	-
Performance Ratio	4.74 %	1.15 %	-
Cost Metrics			
Memory Utilization [GB]			
Power [W]	4.97	5.40	-
Idle Power [W]	4.38	4.38	-
Energy / Inference [mJ]	204.12	20.95	-
GOPS / W	38.12	8.50	-

Table 22: MobileNetV3 Small and ResNet50 INT8 Accuracy (Coral TPU)

Classification accuracy	MobileNetV3 Small	ResNet50
Top 1	41.15 %	63.18 %
Top 5	67.16 %	84.63 %

3.4.4 Google Coral TPU

3.4.4.1 Coral Dev Board

Coral Dev board encapsulates the Edge TPU as a system-on-module along with Quad-core Cortex-A53 CPU.

3.4.4.2 Peak Performance

Edge TPU can reach 4 TOPS in performance with a power efficiency of 2 TOPS/W [61].

3.4.4.3 Toolflow

.pb models were converted to tflite using TensorFlow 2.4 and 2.5. Then tflite models were compiled into models compatible with Edge TPU and executable on Edge TPU runtime. This was done using the Edge TPU Compiler [60], a command line tool that can take tflite models as input and process them targeting optimal inference on Edge TPU devices. Edge TPU Compiler version 16.0.384591198 was used for benchmarks.

Power was measured using a USB C power meter, which measures the whole Dev board's power consumption.

3.4.4.4 Results

Table 23: INT8 (Coral Dev)

Performance	ResNet50	MobileNet	YoloV4
Inference Time [ms]	49.7111	5.6105	
Achieved performance [Inferences/s]	20.11	178.23	
Achieved performance [GOPS]	156.505	31.72	
Peak performance [GOPS]	4000	4000	
Performance Ratio	3.9 %	0.8 %	
Cost Metrics			
Power [W]	3.9116	4.2247	
Idle Power [W]	3.366	3.366	
Energy [J]	194.4	23.7	
GOPS/W	40	7.52	

Table 24: MobileNetV3 Small and ResNet50 INT8 Accuracy (Coral TPU M.2)

Classification accuracy	MobileNetV3 Small	ResNet50
Top 1	41.15 %	63.18 %
Top 5	67.16 %	84.63 %

3.4.5 Hailo.AI Hailo-8

ADLINK COM-Express Xeon-D 1577 + Hailo.AI Hailo-8

- ADLINK Express-BD7-D1577
  - Intel Xeon-D 1577
    - 16 cores
    - 32 threads
    - 1.3 GHz base frequency
    - 24 MB cache
  - 64 GB DDR4-2133 dual-channel
  - 512 GB SATA SSD HDD
  - Cost: 2959.06 € (Mouser)
- Hailo.AI Hailo-8
  - Hailo-8 M.2 2280 B+M key AI module
  - PCIe Gen 3.0, 2 lanes (up to 16 Gbps)

- Cost: $ 199 (up-shop.org 10.2023)

3.4.5.1 Peak Performance

The Hailo-8 is an PCIe based accelerator for DL applications. The performance is rated at 26 TOPS for INT8 [62].

3.4.5.2 Toolflow

The 'Hailo Software Suite' provides compilers and a model zoo optimized for Hailo-8 accelerator. For the evaluation, version 4.11.0 of the hailo software was used.

The power measurements were done inside the RECS testbed. The Hailo-8 plugged into the PCIe-port of Intel Xeon-D1577 was measured individually without the power consumed by the CPU module.

3.4.5.3 Results

Table 25: INT8 (Hailo-8 – Batchsize 1)

Performance	ResNet50	MobileNetV3 small	YoloV4
Inference Time [ms]	2.922	0.503	27.205
Achieved performance [Inferences/s]	342.23	1988.88	34.72
Achieved performance [GOPS]	2662.57	354.02	3176.37
Peak performance [GOPS]	26000	26000	26000
Performance Ratio	10.24%	1.36%	12.22%
Cost Metrics			
Power [W]	3.89	1.81	2.52
Idle Power [W]	~0	~0	~0
Energy [J]	11.37	0.91	72.59
GOPS/W	684.47	195.60	1260.34

Table 26: INT8 (Hailo-8 – Batchsize 8)

Performance	ResNet50	MobileNetV3 small	YoloV4
Inference Time [ms]	2.922	0.503	98.767
Achieved performance [Inferences/s]	1328.50	8456.17	69.31
Achieved performance [GOPS]	10335.73	1505.20	6340.86
Peak performance [GOPS]	26000	26000	26000

Performance Ratio	39.75%	5.79%	24.39%
Cost Metrics			
Power [W]	3.89	1.81	4.00
Idle Power [W]	~0	~0	~0
Energy [J]	2.93	0.21	57.73
GOPS/W	2657.00	831.65	1584.76

Table 27: MobileNetV3 Small and ResNet50 INT8 Accuracy on Hailo-8

Classification accuracy	MobileNetV3 Small	ResNet50
Top 1	64.82 %	74.73 %
Top 5	85.33 %	91.31 %

Table 28: mAP accuracies for YoloV4 on Hailo-8

Classification accuracy	INT8
mAP (.50)	72.40 %
mAP (.50:.95)	47.19 %

3.4.6 NXP i.MX8M Plus

CONGATEC conga-SMX8-Plus

- SMARC based i.MX8M Plus Compute Module
  - 4x ARM Cortex-A53 @1.8GHz
  - Neural Processing Unit (NPU)
  - 4 GB 32 bit LPDDR4 RAM 16.0 GB/s
  - 16GB eMMC Storage
  - Cost: 207 €

3.4.6.1 Peak Performance

The NPU is the integrated accelerator for DL applications. The performance is rated at 2.3 TOPS for INT8 [63].

3.4.6.2 Toolflow

.pb models were converted to tflite using TensorFlow 2.3.1. The i.MX8M Plus supports direct execution of tflite models using TensorFlow-Lite.

The power measurements were done manually connecting a multimeter.

3.4.6.3 Results

Table 29: INT8 (i.MX8M Plus)

Performance	MobileNetV3 Small	ResNet50
Inference Time [ms]	13.7514	21.2383
Achieved performance [Inferences/s]	72.72	47.08
Achieved performance [GOPS]	12.94	366.28
Peak performance [GOPS]	2300	2300
Performance Ratio	0.5 %	15.9 %
Cost Metrics		
Power [W]	3.86	4.68
Memory utilization [MB]	57.867	87.316
Idle Power [W]	3.54	3.54
Energy [J]	53.08	99.4
GOPS/W	3.35	78.26

Table 30: MobileNetV3 Small and ResNet50 INT8 Accuracy (i.MX8M Plus)

Classification accuracy	MobileNetV3 Small	ResNet50
Top 1	63.17 %	70.76 %
Top 5	84.83 %	91.19 %

3.4.7 NVIDIA Jetson AGX Orin

NVIDIA Jetson AGX Orin Platform

- NVIDIA Jetson AGX Orin Platform
    - ARM Cortex-A78AE v8.2 64-Bit-CPU, 8 Cores
    - NVIDIA GPU 2048 Ampere Cores + 64 Tensor Cores
    - 2x NVIDIA DLA-Engines (NVDLA v2)
    - 32 GB 256-bit LPDDR5 204.8GB/s
    - Cost: 1999€

3.4.7.1 Peak Performance

Theoretical peak performance of Jetson AGX Orin is 275 TOPS for INT8 precision, including the 52.5 TOPS of the two NVDLA 2.0 cores [64].

3.4.7.2 Toolflow

For the performance evaluation of the NVIDIA Jetson AGX Orin version 8.4.1 of the TensorRT SDK [65] was used. The onnx models from model set v1 were used for this evaluation. TensorRT specific inference engines were created using the stand-alone executable trtexec. This flow allowed for direct quantization of the input model into the different data types (FP32, FP16 and INT8) examined in this project. Furthermore, changing the target batch size was also possible. After the different engines were created, trtexec was also used to profile the latency and throughput of the different models. Tegrastats, a utility from NVIDIA, was used to measure power consumption and peak memory usage, while the power mode of the module was set to maximum performance mode MAXN.

3.4.7.3 Results

Table 31: ResNet50 (Orin AGX)

Performance	INT8		FP16		FP32	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	0.843	2.950	1.168	4.840	2.154	9.351
Achieved performance [IPS]	1185.78	2711.98	856.10	1652.79	464.32	855.48
Achieved performance [GOPS]	9225.34	21099.17	6660.48	12858.74	3612.39	6655.65
Peak performance [GOPS]	137500		68750		34375	
Performance ratio	6.77 %	15.37 %	9.68 %	18.70 %	10.51 %	19.36 %
Memory utilization [GB]	5.830	5.850	5.870	5.900	5.980	6.030
Power [W]	33.915	47.759	40.101	54.139	45.877	56.031
Idle power [W]	6.900	6.900	6.900	6.900	6.900	6.900
Inference energy [mJ]	28.37	17.58	46.87	32.76	98.78	65.50
GOPS/W	274.28	442.58	166.00	237.52	78.76	118.77

Table 32: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Jetson AGX Orin

Classification accuracy	INT8	FP16	FP32

Top 1	71.04 %	71.05 %	71.04 %
Top 5	89.83 %	89.82 %	89.83 %

Table 33: MobileNetV3 Small (Orin AGX)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	1	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	0.729	1.395	0.753	1.737	0.928	2.831
Achieved performance [IPS]	1371.45	5734.32	1327.50	4604.83	1077.34	2826.32
Achieved performance [GOPS]	244.12	1020.71	236.29	819.66	191.77	503.09
Peak performance [GOPS]	137500		68750		34375	
Performance ratio	0.18 %	0.74 %	0.34 %	1.19 %	0.56 %	1.46 %
Memory utilization [GB]	5.790	5.800	5.790	5.810	7.870	7.900
Power [W]	21.755	32.724	23.454	35.711	25.743	37.301
Idle power [W]	6.900	6.900	6.900	6.900	6.900	6.900
Inference energy [mJ]	15.86	5.71	17.67	7.76	23.89	13.20
GOPS/W	11.22	31.19	10.07	22.95	7.45	13.49

Table 34: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Jetson AGX Orin

Classification accuracy	**INT8**	**FP16**	**FP32**
Top 1	36.09 %	63.18 %	63.15 %
Top 5	61.95 %	84.82 %	84.82 %

Table 35: YoloV4 (Orin AGX)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	6.498	41.205	9.503	65.764	19.532	141.031

Achieved performance [Inferences/s]	153.89	194.15	105.23	121.65	51.20	56.73
Achieved performance [GOPS]	9295.09	11726.83	6355.61	7347.48	3092.33	3426.19
Peak performance [GOPS]	137500		68750		34375	
Performance ratio	6.76 %	8.53 %	9.24 %	10.69 %	9.00 %	9.97 %
Memory utilization [GB]	5.970	6.360	6.120	6.750	7.430	8.530
Power [W]	50.34	59.804	59.893	67.067	58.720	62.510
Idle Power [W]	6.900	6.900	6.900	6.900	6.900	6.900
Inference energy [mJ]	327.07	308.03	569.19	551.32	1146.93	1101.98
GOPS/W	184.67	196.09	106.12	109.55	52.66	54.81

Table 36: mAP accuracies for YoloV4 on NVIDIA Jetson AGX Orin

Classification accuracy	INT8	FP16	FP32
mAP (.50)	48.1 %	49.4 %	49.4 %
mAP (.50:.95)	73.8 %	74.2 %	74.2 %

3.4.8 NVIDIA Jetson AGX Xavier

NVIDIA Jetson Xavier AGX Evaluation Platform

- NVIDIA Jetson Xavier AGX
  - NVIDIA Carmel ARM v8.2 64-Bit-CPU, 8 Cores
  - NVIDIA GPU 512 Volta Cores + 64 Tensor Cores
  - 2x NVIDIA DLA-Engines
  - 32 GB DDR4-2133 quad-channel

  Cost: 960 €

3.4.8.1 Peak Performance

Theoretical peak performance of Jetson AGX Xavier is 22TOPS, not considering the performance of NVDLA cores [66].

3.4.8.2 Toolflow

For the performance evaluation of the NVIDIA Jetson AGX Xavier version 8.4.1 of the TensorRT SDK [65] was used. The onnx models from model set v1 were used for this evaluation. TensorRT specific inference engines were created using the stand-alone executable trtexec. This flow allowed for direct quantization of the input model into the different data types (FP32, FP16 and INT8) examined in this project. Furthermore, changing the target batch size was also possible with ease. After the different engines were created,

trtexec was also used to profile the latency and throughput of the different models. Tegrastats, a utility from NVIDIA, was used to measure power consumption and peak memory usage. The NVIDIA Jetson Xavier AGX was tested in different power modes, the low power (LP) 15 W and the high power (HP) MAXN 30 W mode, to determine changes in energy efficiency.

3.4.8.3 Results (15 W Mode)

Table 37: ResNet50 (Xavier AGX LP)

Performance	INT8	FP16	FP32
Inference time [ms]	3.7242	5.8149	14.9318
Achieved performance [Inferences/s]	268.51	171.97	66.97
Achieved performance [GOPS]	2088	1336	521
Peak performance **[GOPS]**	22000	11000 (INT/2)	5500 (INT/4)
Performance Ratio	9.49 %	12.14 %	9.47 %
Cost Metrics			
Memory Utilization [GB]	5.935	5.953	6.502
Power [W]	8.8380	10.0140	10.2936
Idle Power [W]	4.0	4.0	4.0
Energy/Inference [mJ]	32	58	150
[GOPS/W]	236.24	133.4	50.6

Table 38: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Jetson AGX Xavier

Classification accuracy	INT8	FP16	FP32
Top 1	71.05 %	71.03 %	71.05 %
Top 5	89.82 %	89.82 %	89.82 %

Table 39: MobileNetV3 Small (Xavier AGX LP)

Performance	INT8	FP16	FP32
Inference time [ms]	2.2595	2.5541	3.0871

Achieved performance [Inferences/s]	442.57	391.52	323.92
Achieved performance [GOPS]	78.78	69.68	57.64
Peak performance [GOPS]	22000	11000 (INT/2)	5500 (INT/4)
Performance Ratio	0.35 %	0.633 %	1.048 %
Cost Metrics			
Memory utilization [GB]	5.867	5.862	6.316
Power [W]	7.5758	7.7921	7.9913
Idle Power [W]	4.0	4.0	4.0
Energy/Inference [mJ]	17	19.9	24
[GOPS/W]	10.398	8.94	7.2

Table 40: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Jetson AGX Xavier

Classification accuracy	INT8	FP16	FP32
Top 1	29.59 %	54.29 %	54.32 %
Top 5	54.22 %	79.15 %	79.14 %

Table 41: YoloV4 (Xavier AGX LP)

Performance	INT8	FP16	FP32
Inference time [ms]	20.3331	30.2533	92.4868
Achieved performance [Inferences/s]	49.18	33.05	10.812
Achieved performance [GOPS]	2970	1996	652
Peak performance [GOPS]	22000	11000 (INT/2)	5500 (INT/4)
Performance Ratio	13.5 %	18.14 %	11.85 %

Cost Metrics			
Memory utilization [GB]	6.222	6.235	7.145
Power [W]	9.6104	10.9868	10.8139
Idle Power [W]	4.0	4.0	4.0
Energy [J]	0.195	0.332	1.00
[GOPS/W]	309.04	181.66	60.292

Table 42: mAP accuracies for YoloV4 on NVIDIA Jetson AGX Xavier

Classification accuracy	**INT8**	**FP16**	**FP32**
mAP (.50)	48.3 %	49.4 %	49.4 %
mAP (.50:.95)	73.7 %	74.2 %	74.2 %

3.4.8.4 Results (30 W Mode)

Table 43: ResNet50 (Xavier AGX HP)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	1.990	7.125	3.091	13.540	7.872	47.276
Achieved performance [Inferences/s]	502.47	1122.83	323.54	590.83	127.03	169.22
Achieved performance [GOPS]	3909.19	8735.62	2517.12	4596.68	988.28	1316.53
Peak performance [GOPS]	22000		11000		5500	
Performance Ratio	17.77 %	39.71 %	22.88 %	41.79 %	17.97 %	23.94 %
Cost Metrics						
Memory Utilization [GB]	6.16 GB	6.16 GB	6.17 GB	6.16 GB	6.67 GB	6.30 GB
Power [W]	25.915	32.506	32.046	34.933	36.064	39.503
Idle Power [W]	9.244	9.244	9.244	9.244	9.244	9.244
Energy / Inference [mJ]	51.58	28.95	99.05	59.12	283.91	233.44

GOPS / W	150.85	268.74	78.55	131.59	27.40	33.33

Table 44: MobileNetV3 Small (Xavier AGX HP)

Performance	INT8		FP16		FP32	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	1.190	2.944	1.347	3.816	1.622	5.827
Achieved performance [Inferences/s]	840.31	2717.51	742.16	2096.55	616.40	1372.85
Achieved performance [GOPS]	149.57	483.72	132.10	373.19	109.72	244.37
Peak performance [GOPS]	22000		11000		5500	
Performance Ratio	0.68 %	2.20 %	1.20 %	3.39 %	1.99 %	4.44 %
Cost Metrics						
Memory Utilization [GB]	6.17 GB	6.18 GB	6.17 GB	6.61 GB	6.53 GB	6.59 GB
Power [W]	19.176	24.931	21.096	29.053	23.469	30.878
Idle Power [W]	9.244	9.244	9.244	9.244	9.244	9.244
Energy / Inference [mJ]	22.82	9.17	28.43	13.86	38.07	22.49
GOPS / W	7.80	19.40	6.26	12.84	4.68	7.91

Table 45: YoloV4 (Xavier AGX HP)

Performance	INT8		FP16		FP32	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	10.798	62.758	16.426	106.187	47.579	347.063
Achieved performance [Inferences/s]	92.61	127.47	60.88	75.34	21.02	23.05
Achieved performance [GOPS]	5593.84	7699.44	3677.21	4550.46	1269.46	1392.25
Peak performance [GOPS]	22000		11000		5500	

Performance Ratio	25.43 %	35.00 %	33.43 %	41.37 %	23.08 %	25.31 %
Cost Metrics						
Memory Utilization [GB]	6.56 GB	6.77 GB	6.56 GB	6.89 GB	7.43 GB	8.00 GB
Power [W]	31.632	36.678	36.603	39.997	40.817	43.691
Idle Power [W]	9.244	9.244	9.244	9.244	9.244	9.244
Energy / Inference [mJ]	341.55	287.73	601.22	530.90	1942.04	1895.44
GOPS / W	176.84	209.92	100.46	113.77	31.10	31.87

3.4.9 NVIDIA Jetson AGX Orin (TVM)

NVIDIA Jetson AGX Orin Platform

- NVIDIA Jetson AGX Orin Platform
    - ARM Cortex-A78AE v8.2 64-Bit-CPU, 8 Cores
    - NVIDIA GPU 2048 Ampere Cores + 64 Tensor Cores
    - 2x NVIDIA DLA-Engines (NVDLA v2)
    - 32 GB 256-bit LPDDR5 204.8GB/s
    - Cost: 1999€

3.4.9.1 Peak Performance

Theoretical peak performance of Jetson AGX Orin is 275 TOPS for INT8 precision, including the 52.5 TOPS of the two NVDLA 2.0 cores [64].

3.4.9.2 Toolflow

3.4.9.2.1 ResNet50

ResNet50 was processed with the Kenning framework using the following flow:

- For FP32:
    - Model was firstly processed with TensorFlow Lite to create a FP32 TFLite model for initial optimizations
    - Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 87), with available CUDNN, CUBLAS and TensorRT libraries. Layout was set to NHWC for data, OHWI for kernels
- For FP16:
    - Model was firstly processed with TensorFlow Lite to create a FP32 TFLite model for initial optimizations
    - Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 87), with available CUDNN, CUBLAS and TensorRT libraries. Layout was set to NHWC for data, OHWI for kernels. In addition, a transformation pass converting weights from FP32 to FP16 was applied
- For INT8:

- Model was firstly processed with TensorFlow Lite – the model was quantized here to INT8 precision using the calibration dataset. The calibration dataset was 10000 random images from the training dataset
- Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 87), with available CUDNN, CUBLAS and TensorRT libraries.. Layout was set to NHWC for data, OHWI for kernels.

3.4.9.2.2 MobileNetV3

MobileNetV3 was processed with the Kenning framework using the following flow:

- For FP32:
  - Model was loaded directly by TVM and compiled for CUDA runtime (compute capability 87), with available CUDNN, CUBLAS and TensorRT libraries.
- For FP16:
  - Model was loaded directly by TVM and compiled for CUDA runtime (compute capability 87), with available CUDNN, CUBLAS and TensorRT libraries. In addition, a transformation pass converting weights from FP32 to FP16 was applied
- For INT8:
  - Model was firstly processed with TensorFlow Lite – the model was quantized here to INT8 precision using the calibration dataset. The calibration dataset was 10000-200000 random images from the training dataset (various calibration dataset sizes were checked)
  - Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 87), with possible frameworks CUDNN, CUBLAS, TensorRT.

INT8 precision for MobileNetV3 resulted in significant quality decrease regardless of selected calibration dataset size. Looking at the confusion matrix for INT8 precision, after the quantization process some of the classes yield zero-like results – due to this some classes are always ignored in the prediction process.

Changing the calibration dataset size may result in slightly different set of classes that are no longer usable.

This may be due to the fact that scale and zero factor computed for the final layer are global for all classes – values for certain classes may be so low they fall below quantized ranges, which results in a significant clipping error.

One of possible solutions would be to split the final 1000-element tensor to separate single-value tensors for which the scale and zero-point values would be computed separately.

3.4.9.2.3 YOLOv4

YOLOv4 was processed with the Kenning framework using the following flow:

- For FP32:

- Model was loaded directly by TVM and compiled for CUDA runtime (compute capability 87), with available CUDNN, CUBLAS and TensorRT libraries. Model was kept in original NCHW format.

- For FP16:
  - Model was loaded directly by TVM and compiled for CUDA runtime (compute capability 87), with available CUDNN, CUBLAS and TensorRT libraries. Model was kept in original NCHW format. In addition, a transformation pass converting weights from FP32 to FP16 was applied
- For INT8:
  - Model was firstly processed with TensorFlow Lite – the model was quantized here to INT8 precision using the calibration dataset. The calibration dataset was 1000-4000 random images from the training dataset (various calibration dataset sizes were checked)
  - Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 87), with possible frameworks CUDNN, CUBLAS, TensorRT. Model was kept in original NCHW format.

The quantization process of YOLOv4 using the TensorFlow Lite converter was a very slow process allowing to use only small calibration datasets (around 1000-4000 random images from the training dataset) - this on the other hand resulted in models unable to predict objects correctly. Apache TVM data-aware quantization algorithm also allowed using only very small calibration datasets. Due to this, the results for INT8 precision are not included.

The possible future fix could be using different framework for quantization purposes, before loading to TVM.

3.4.9.3 Results

Table 46: ResNet50 (Orin TVM)

Performance	INT8	FP16	FP32
Inference time [ms]	0.5792	8.4106	8.5608
Achieved performance [Inferences/s]	1726.52	118.89	116.81
Achieved performance [GOPS]	13432.32	924.96	908.78
Peak performance [GOPS]	137500	68750	34375
Performance Ratio	10.23 %	1.34 %	2.64 %
Cost Metrics			
Memory Utilization [GB]	6.725	7.896	8.009

Power [W]	31.368	21.152	27.039
Idle Power [W]	6.90	6.90	6.90
Energy/Inference [mJ]	18.17	177.91	231.48
[GOPS/W]	428.22	43.73	33.61

Table 47: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Jetson AGX Orin TVM

Classification accuracy	FP32	FP16	INT8
Top 1	70,90 %	70,91 %	67,85 %
Top 5	89,74 %	89,73 %	87,45 %

Table 48: MobileNetV3 Small (Orin TVM)

Performance	INT8	FP16	FP32
Inference time [ms]	3.6054	3.3374	3.8813
Achieved performance [Inferences/s]	277.36	299.63	257.65
Achieved performance [GOPS]	49.37	53.33	45.86
Peak performance [GOPS]	137500	68750	34375
Performance Ratio	0.035 %	0.077 %	0.133 %
Cost Metrics			
Memory utilization [GB]	7.568	7.534	7.566
Power [W]	19.075	18.974	19.079
Idle Power [W]	6.90	6.90	6.90
Energy/Inference [mJ]	68.7	63.32	74.05
[GOPS/W]	2.59	2.81	2.40

Table 49: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Jetson AGX Orin TVM

Classification accuracy	INT8	FP16	FP32
Top 1	29.59 %	54.29 %	54.32 %

Top 5	54.22 %	79.15 %	79.14 %

Table 50: YoloV4 (Orin TVM)

Performance	FP16	FP32
Inference time [ms]	21.8598	25.0377
Achieved performance [Inferences/s]	45.75	39.94
Achieved performance [GOPS]	2763.3	2412.38
Peak performance [GOPS]	68750	34375
Performance Ratio	4.02 %	7.02 %
Cost Metrics		
Memory utilization [GB]	7.936	8.565
Power [W]	32.094	38.471
Idle Power [W]	6.90	6.90
Energy [J]	701.51	963.21
[GOPS/W]	86.10	62.71

Table 51: mAP accuracies for YoloV4 on NVIDIA Jetson AGX Orin TVM

mAP	FP32	FP16
mAP (.50)	67,2 %	67,2 %
mAP (.50:.95)	46,5 %	46,5 %

3.4.10 NVIDIA Jetson AGX Xavier (TVM)

NVIDIA Jetson Xavier AGX Evaluation Platform

- NVIDIA Jetson Xavier AGX
  - NVIDIA Carmel ARM v8.2 64-Bit-CPU, 8 Cores
  - NVIDIA GPU 512 Volta Cores + 64 Tensor Cores
  - 2x NVIDIA DLA-Engines
  - 32 GB DDR4-2133 quad-channel

  Cost: 960 €

3.4.10.1 Peak Performance

Theoretical peak performance of Jetson AGX Xavier is 22TOPS, not considering the performance of NVDLA cores [66].

3.4.10.2 Toolflow

3.4.10.2.1 ResNet50

ResNet50 was processed with the Kenning framework using the following flow:

- For FP32:
  - Model was firstly processed with TensorFlow Lite to create a FP32 TFLite model for initial optimizations
  - Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 72), with available CUDNN, CUBLAS and TensorRT libraries. Layout was set to NHWC for data, OHWI for kernels
- For FP16:
  - Model was firstly processed with TensorFlow Lite to create a FP32 TFLite model for initial optimizations
  - Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 72), with available CUDNN, CUBLAS and TensorRT libraries. Layout was set to NHWC for data, OHWI for kernels. In addition, a transformation pass converting weights from FP32 to FP16 was applied
- For INT8:
  - Model was firstly processed with TensorFlow Lite – the model was quantized here to INT8 precision using the calibration dataset. The calibration dataset was 10000 random images from the training dataset
  - Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 72), with available CUDNN, CUBLAS and TensorRT libraries.. Layout was set to NHWC for data, OHWI for kernels.

3.4.10.2.2 YOLOv4

YOLOv4 was processed with the Kenning framework using the following flow:

- For FP32:
  - Model was loaded directly by TVM and compiled for CUDA runtime (compute capability 72), with available CUDNN, CUBLAS and TensorRT libraries. Model was kept in original NCHW format.
- For FP16:
  - Model was loaded directly by TVM and compiled for CUDA runtime (compute capability 72), with available CUDNN, CUBLAS and TensorRT libraries. Model was kept in original NCHW format. In addition, a transformation pass converting weights from FP32 to FP16 was applied
- For INT8:
  - Model was firstly processed with TensorFlow Lite – the model was quantized here to INT8 precision using the calibration dataset. The calibration dataset was 1000-4000 random images from the training dataset (various calibration dataset sizes were checked)

- Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 72), with possible frameworks CUDNN, CUBLAS, TensorRT. Model was kept in original NCHW format.

The quantization process of YOLOv4 using the TensorFlow Lite converter was a very slow process allowing to use only small calibration datasets (around 1000-4000 random images from the training dataset) - this on the other hand resulted in models unable to predict objects correctly. Apache TVM data-aware quantization algorithm also allowed using only very small calibration datasets. Due to this, the results for INT8 precision are not included.

The possible future fix could be using different framework for quantization purposes, before loading to TVM.

3.4.10.2.3 MobileNetV3

MobileNetV3 was processed with the Kenning framework using the following flow:

- For FP32:
  - Model was converted to ONNX and then loaded by TVM and compiled for CUDA runtime (compute capability 72), with available CUDNN, CUBLAS and TensorRT libraries.
- For FP16:
  - Model was converted to ONNX and then loaded by TVM and compiled for CUDA runtime (compute capability 72), with available CUDNN, CUBLAS and TensorRT libraries. In addition, a transformation pass converting weights from FP32 to FP16 was applied
- For INT8:
  - Model was firstly processed with TensorFlow Lite – the model was quantized here to INT8 precision using the calibration dataset. The calibration dataset was 10000-200000 random images from the training dataset (various calibration dataset sizes were checked)
  - Secondly, the model was compiled using the TVM framework for CUDA runtime (compute capability 72), with possible frameworks CUDNN, CUBLAS, TensorRT.

INT8 precision for MobileNetV3 resulted in significant quality decrease regardless of selected calibration dataset size. Looking at the confusion matrix for INT8 precision, after the quantization process some of the classes yield zero-like results – due to this some classes are always ignored in the prediction process.

Changing the calibration dataset size may result in slightly different set of classes that are no longer usable.

This may be due to the fact that scale and zero factor computed for the final layer are global for all classes – values for certain classes may be so low they fall below quantized ranges, which results in a significant clipping error.

One of possible solutions would be to split the final 1000-element tensor to separate single-value tensors for which the scale and zero-point values would be computed separately.

3.4.10.3 Results

Table 52: ResNet50 (Xavier TVM)

Performance	INT8	FP16	FP32
Inference time [ms]	0.8363	7.6726	12.1483
Achieved performance [Inferences/s]	1195.74	130.33	82.31
Achieved performance [GOPS]	9302	1013.96	640.37
Peak performance [GOPS]	22000	11000 (INT/2)	5500 (INT/4)
Performance Ratio	42.228 %	9.218 %	11.643 %
Cost Metrics			
Memory Utilization [GB]	8.620	8.552	8.663
Power [W]	20.384	12.086	17.969
Idle Power [W]	4.0	4.0	4.0
Energy/Inference [mJ]	17.05	92.73	218.31
[GOPS/W]	456.33	83.89	35.64

Table 53: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Jetson AGX Xavier TVM

Classification accuracy	INT8	FP16	FP32
Top 1	71.05 %	71.03 %	71.05 %
Top 5	89.82 %	89.82 %	89.82 %

Table 54: MobileNetV3 Small (Xavier TVM)

Performance	INT8	FP16	FP32
Inference time [ms]	4.532	4.5041	9.1732
Achieved performance [Inferences/s]	220.65	222.02	109.01
Achieved performance [GOPS]	39.27	39.52	19.4
Peak performance [GOPS]	22000	11000 (INT/2)	5500 (INT/4)

Performance Ratio	0.178 %	0.359 %	0.353 %
Cost Metrics			
Memory utilization [GB]	8.365	7.699	7.766
Power [W]	9.915	9.859	10.013
Idle Power [W]	4.0	4.0	4.0
Energy/Inference [mJ]	44.94	44.41	91.85
[GOPS/W]	3.96	4.01	1.94

Table 55: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Jetson AGX Xavier TVM

Classification accuracy	INT8	FP16	FP32
Top 1	29.59 %	54.29 %	54.32 %
Top 5	54.22 %	79.15 %	79.14 %

Table 56: YoloV4 (Xavier TVM)

Performance	**FP16**	**FP32**
Inference time [ms]	18.2214	33.814
Achieved performance [Inferences/s]	54.88	29.57
Achieved performance [GOPS]	3314.75	1786.02
Peak performance [GOPS]	11000 (INT/2)	5500 (INT/4)
Performance Ratio	30.13 %	32.47 %
Cost Metrics		
Memory utilization [GB]	8.736	8.673
Power [W]	23.89	29.121
Idle Power [W]	4.0	4.0
Energy [mJ]	435.31	984.82
[GOPS/W]	138.75	61.33

Table 57: mAP accuracies for YoloV4 on NVIDIA Jetson AGX Xavier TVM

Classification accuracy	INT8	FP16	FP32
mAP (.50)	48.3 %	49.4 %	49.4 %
mAP (.50:.95)	73.7 %	74.2 %	74.2 %

3.4.11 NVIDIA Jetson Xavier NX

NVIDIA Jetson Xavier NX Evaluation Platform

- NVIDIA Jetson Xavier NX
  - NVIDIA Carmel ARM v8.2 64-Bit-CPU, 6 Cores
  - NVIDIA GPU 384 Volta Cores + 48 Tensor Cores
  - 2x NVIDIA DLA-Engines
  - 8 GB DDR4-1866 dual-channel
  - Cost: 519 €

3.4.11.1 Peak Performance

The theoretical maximum performance of 21000 GOPS that is rated by NVIDIA shows the combined performance of GPU and NVDLA cores. Unfortunately there are no separate values for GPU and NVDLA. From the NVIDIA Xavier AGX the GPU performance is known. From that value one can calculate that the Volta GPU performs 32 INT8 operations per cycle per core. This results in 13516 GOPS for the GPU of the Xavier NX.

3.4.11.2 Toolflow

For the performance evaluation of the NVIDIA Jetson Xavier version 7.1.3 of the TensorRT SDK [65] was used. The onnx models from model set v1 were used for this evaluation. TensorRT specific inference engines were created using the stand-alone executable trtexec. This flow allowed for direct quantization of the input model into the different data types (FP32, FP16, and INT8) examined in this project. Furthermore, changing the target batch size was also possible with ease. After the different engines were created, trtexec was also used to profile the latency and throughput of the different models.

Tegrastats, a utility from NVIDIA, was used to measure power consumption and peak memory usage.

3.4.11.3 Results

Table 58: ResNet50 (Xavier NX)

Performance	INT8	FP16	FP32

Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	3.106	12.236	5.07	23.658	13.282	97.185
Achieved performance [Inferences/s]	320.62	653.81	195.69	338.15	71.65	82.32
Achieved performance [GOPS]	2494.39	5086.63	1522.50	2630.82	557.47	640.43
Peak performance [GOPS]	13516		6758 (INT8 / 2)		3379 (INT8 / 4)	
Performance Ratio	18.46 %	37.63 %	22.53 %	38.93 %	16.50 %	18.95 %
Cost Metrics						
Memory Utilization [GB]	2.21	2.19	2.21	2.19	2.57	2.19
Power [W]	11.86	14.421	13.42	15.202	15.38	15.449
Idle Power [W]	4.38	4.38	4.38	4.38	4.38	4.38
Energy / Inference [mJ]	36.84	22.06	68.58	44.96	214.65	187.67
GOPS / W	210.32	352.72	113.45	173.06	36.25	41.45

Table 59: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Jetson Xavier NX

Classification accuracy	INT8	FP16	FP32
Top 1	71.05 %	71.03 %	71.05 %
Top 5	89.82 %	89.82 %	89.82 %

Table 60: MobileNetV3 (Xavier NX)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	1.571	4.438	2.059	6.051	2.264	9.887
Achieved performance [Inferences/s]	636.54	1802.61	485.67	1322.09	441.69	809.14
Achieved performance [GOPS]	113.30	320.86	86.44	235.33	78.62	144.03
Peak performance [GOPS]	13516		6758 (INT8 / 2)		3379 (INT8 / 4)	

Performance Ratio	0.8 %	2.4 %	1.3 %	3.5 %	2.3 %	4.3 %
Cost Metrics						
Memory Utilization [GB]	2.21	2.19	2.21	2.19	2.58	2.55
Power [W]	9.18	11.875	10.17	13.471	11.46	14.238
Idle Power [W]	4.38	4.38	4.38	4.38	4.38	4.38
Energy / Inference [mJ]	14.42	6.59	20.94	10.19	25.94	17.60
GOPS / W	12.34	27.02	8.50	17.47	6.86	10.11

Table 61: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Jetson Xavier NX

Classification accuracy	INT8	FP16	FP32
Top 1	29.59 %	54.29 %	54.32 %
Top 5	54.22 %	79.15 %	79.14 %

Table 62: YoloV4 (Xavier NX)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	16.493	109.062	27.814	209.990	97.144	792.084
Achieved performance [Inferences/s]	60.63	73.35	35.95	38.10	10.29	10.10
Achieved performance [GOPS]	3662.16	4430.51	2171.57	2301.06	621.76	610.04
Peak performance [GOPS]	13516		6758 (INT8 / 2)		3379 (INT8 / 4)	
Performance Ratio	27.09 %	32.78 %	32.13 %	34.05 %	18.4 %	18.05 %
Cost Metrics						
Memory Utilization [GB]	2.18	2.30	2.18	2,42	3.23	3.95
Power [W]	15.13	15.449	15.41	15.449	15.45	15.449
Idle Power [W]	4.38	4.38	4.38	4.38	4.38	4.38
Energy / Inference [mJ]	249.54	210.61	428.61	405.52	1500.87	1529.61
GOPS / W	242.05	286.78	140.92	148.95	40.24	39.49

Table 63: mAP accuracies for YoloV4 on NVIDIA Jetson Xavier NX

Classification accuracy	INT8	FP16	FP32
mAP (.50)	48.3 %	49.4 %	49.4 %
mAP (.50:.95)	73.7 %	74.2 %	74.2 %

3.4.12 NVIDIA Jetson Orin NX

NVIDIA Jetson Orin NX Evaluation Platform

- NVIDIA Jetson Orin NX
  - ARM Cortex-A78AE v8.2 64-Bit-CPU, 8 Cores
  - NVIDIA GPU 1048 Ampere Cores + 32 Tensor Cores
  - 2x NVIDIA DLAv2-Engines
  - 16 GB DDR5-3200 dual-channel
  - Cost: 643 €

3.4.12.1 Peak Performance

The theoretical maximum performance of 100000 GOPS that is rated by NVIDIA shows the combined performance of GPU and NVDLA cores. Unfortunately there are no separate values for GPU and NVDLA.

3.4.12.2 Toolflow

For the performance evaluation of the NVIDIA Jetson Xavier version 8.4.1 of the TensorRT SDK [65] was used. The onnx models from model set v1 were used for this evaluation. TensorRT specific inference engines were created using the stand-alone executable trtexec. This flow allowed for direct quantization of the input model into the different data types (FP32, FP16, and INT8) examined in this project. Furthermore, changing the target batch size was also possible with ease. After the different engines were created, trtexec was also used to profile the latency and throughput of the different models.

Tegrastats, a utility from NVIDIA, was used to measure power consumption and peak memory usage.

3.4.12.3 Results

Table 64: ResNet50 (Orin NX)

Performance	INT8		FP16		FP32	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	2.4	7.011	4.645	11.292	4.586	22.791
Achieved performance [Inferences/s]	416.73	1141.09	215.26	708.46	218.08	351.01

Achieved performance [GOPS]	3242.17	8877.67	1674.75	5511.82	1696.65	2730.85
Peak performance [GOPS]	100000		50000 (INT8 / 2)		25000 (INT8 / 4)	
Performance Ratio	3.23%	8.88%	3.35%	11.02%	6.79%	10.92%
Cost Metrics						
Memory Utilization [GB]	3.540	3.540	3.540	3.540	3.730	3.730
Power [W]	18	19	19	20	19	20
Idle Power [W]	4.700	4.700	4.700	4.700	4.700	4.700
Energy / Inference [mJ]	41.99	16.30	86.41	28.51	85.75	58.12
GOPS / W	185.27	477.29	90.04	272.86	90.73	133.87

Table 65: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Jetson Orin NX

Classification accuracy	INT8	FP16	FP32
Top 1	71.05 %	71.03 %	71.05 %
Top 5	89.82 %	89.82 %	89.82 %

Table 66: MobileNetV3 (Orin NX)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	1.261	2.543	1.245	3.173	1.619	4.977
Achieved performance [Inferences/s]	792.95	3146.16	802.96	2520.98	617.59	1607.32
Achieved performance [GOPS]	141.15	560.02	142.93	448.73	109.93	286.10
Peak performance [GOPS]	100000		50000 (INT8 / 2)		25000 (INT8 / 4)	
Performance Ratio	0.14%	0.56%	0.29%	0.90%	0.44%	1.14%
Cost Metrics						
Memory Utilization [GB]	3.740	3.940	3.740	3.940	3.850	4.140

Power [W]	10	15	10	16	10	17
Idle Power [W]	4.700	4.700	4.700	4.700	4.700	4.700
Energy / Inference [mJ]	11.98	4.70	12.70	6.39	16.68	10.58
GOPS / W	14.86	37.84	14.01	27.87	10.67	16.83

Table 67: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Jetson Orin NX

Classification accuracy	INT8	FP16	FP32
Top 1	29.59 %	54.29 %	54.32 %
Top 5	54.22 %	79.15 %	79.14 %

Table 68: YoloV4 (Orin NX)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference Time [ms]	16.645	106.194	22.813	157.256	48.841	327.626
Achieved performance [Inferences/s]	60.08	75.33	43.84	50.87	20.47	24.42
Achieved performance [GOPS]	3628.75	4550.17	2647.66	3072.69	1236.66	1474.85
Peak performance [GOPS]	100000		50000 (INT8 / 2)		25000 (INT8 / 4)	
Performance Ratio	3.63%	4.55%	5.30%	6.15%	4.95%	5.90%
Cost Metrics						
Memory Utilization [GB]	4.000	4.070	4.000	4.100	4.700	4.840
Power [W]	19	20	21	22	21	22
Idle Power [W]	4.700	4.700	4.700	4.700	4.700	4.700
Energy / Inference [mJ]	309.59	261.50	485.91	436.39	1006.13	896.88
GOPS / W	195.09	230.97	124.30	138.41	60.03	67.34

Table 69: mAP accuracies for YoloV4 on NVIDIA Jetson Orin NX

Classification accuracy	INT8	FP16	FP32

mAP (.50)	48.3 %	49.4 %	49.4 %
mAP (.50:.95)	73.7 %	74.2 %	74.2 %

3.4.13 NVIDIA Jetson TX2

NVIDIA Jetson TX2 Evaluation Platform

- NVIDIA Jetson TX2
  - Dual-Core NVIDIA Denver 2 64-Bit CPU
  - Quad-Core ARM Cortex-A57 MPCore
  - 8 GB DDR4-1866 dual-channel
  - 32 GB eMMC HDD
  - Cost: 575 €

3.4.13.1 Peak Performance

The theoretical peak performance of Nvidia Jetson TX2 is 1300 GOPS for FP16 [67]. The value used for FP32 was calculated as half of the value for FP16 (650 GOPS). Even though INT8 is not supported by the TX2 it was calculated to be 2600 GOPS. But it seems that TensorRT is falling back to FP32 operation.

3.4.13.2 Tool Flow

For the performance evaluation of the NVIDIA Jetson TX2 version 8.2.1 of the TensorRT SDK [65] was used. The onnx models from model set v1 were used for this evaluation. TensorRT specific inference engines were created using the stand-alone executable trtexec. This flow allowed for direct quantization of the input model into the different data types (FP32, FP16, and INT8) examined in this project. Furthermore, changing the target batch size was also possible with ease. After the different engines were created, trtexec was also used to profile the latency and throughput of the different models.

Tegrastats, a utility from NVIDIA, was used to measure power consumption and peak memory usage.

3.4.13.3 Results

Table 70: ResNet50 (TX2)

Performance	INT8		FP16		FP32	
Batchsize	1	8	1	8	1	8
Inference Time [ms]	18.710	126.330	10.508	67.963	18.956	125.735
Achieved performance [Inferences/s]	53.45	63.33	95.16	117.71	52.75	63.63
Achieved performance [GOPS]	415.82	492.68	740.37	915.79	410.42	495.01

Peak performance [GOPS]	CPU only		1300		650 (FP16 / 2)	
Performance Ratio			56.95 %	70.45 %	63.14 %	76.16 %
Cost Metrics						
Memory Utilization [GB]	1.91 GB	1.91 GB	1.91 GB	1.91 GB	1.91 GB	1.91 GB
Power [W]	15.808	16.599	15.016	16.074	16.112	16.750
Idle Power [W]	3.425	3.425	3.425	3.425	3.425	3.425
Energy / Inference [mJ]	295.76	262.12	157.79	136.56	305.42	263.26
GOPS / W	26.30	29.68	49.31	56.97	25.47	29.55

Table 71: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Jetson TX2

Classification accuracy	FP16	FP32
Top 1	71.03 %	71.05 %
Top 5	89.82 %	89.82 %

Table 72: MobileNetV3 (TX2)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	1	8	1	8	1	8
Inference Time [ms]	3.756	18.497	3.777	17.389	3.395	18.494
Achieved performance [Inferences/s]	266.23	432.51	264.74	460.06	294.59	432.58
Achieved performance [GOPS]	47.39	76.99	47.12	81.89	52.44	77.00
Peak performance [GOPS]	CPU only		1300		650 (FP16 / 2)	
Performance Ratio			3.62 %	6.30 %	8.07 %	11.85 %
Cost Metrics						
Memory Utilization [GB]	2.12 GB	2.12 GB	2.12 GB	1.91 GB	2.12 GB	2.12 GB
Power [W]	10.125	12.816	10.163	12.441	10.619	12.854
Idle Power [W]	3.425	3.425	3.425	3.425	3.425	3.425

Energy / Inference [mJ]	38.03	29.63	38.39	27.04	36.05	29.71
GOPS / W	4.68	6.01	4.64	6.58	4.94	5.99

Table 73: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Jetson TX2

Classification accuracy	FP16	FP32
Top 1	54.33 %	54.32 %
Top 5	79.15 %	79.14 %

Table 74: YoloV4 (TX2)

Performance	**INT8**		**FP16**		**FP32**	
Batchsize	1	8	1	8	1	8
Inference Time [ms]	125.119	952.778	74.747	549.968	124.670	950.723
Achieved performance [Inferences/s]	7.99	8.40	13.38	14.55	8.02	8.41
Achieved performance [GOPS]	482.74	507.15	808.05	878.60	484.48	508.24
Peak performance [GOPS]	CPU only		1300		650 (FP16 / 2)	
Performance Ratio			62.16 %	67.58 %	74.54 %	78.19 %
Cost Metrics						
Memory Utilization [GB]	2.40 GB	3.12 GB	1.92 GB	2.19 GB	2.32 GB	3.04 GB
Power [W]	17.617	18.034	16.333	16.788	17.582	18.142
Idle Power [W]	3.425	3.425	3.425	3.425	3.425	3.425
Energy / Inference [mJ]	2204.22	2147.80	1220.85	1154.11	2191.95	2156.00
GOPS / W	27.40	28.12	49.47	52.33	27.56	28.01

Table 75: mAP accuracies for YoloV4 on NVIDIA Jetson TX2

Classification accuracy	FP16	FP32
mAP (.50)	49.4 %	49.4 %
mAP (.50:.95)	74.2 %	74.2 %

3.4.14 NVIDIA Jetson Nano

3.4.14.1 Peak Performance

Theoretical peak performance of Nvidia Jetson Nano is 472.0 GOPS for FP16 [68]. The value used for FP32 was calculated as half of the value for FP16 (236.0 GOPS).

3.4.14.2 Toolflow

In order to create inference engines and run inference on Nvidia Jetson Nano, version 8.0.1 of NVIDIA's TensorRT library was used. TensorRT comes with a profiling tool called trtexec, which allows for latency, throughput and layer-wise benchmarking of neural networks in TensorRT representation. Onnx versions of model set v1 were used to create TensorRT inference engines and to measure timing using trtexec.

Tegrastats, a utility from NVIDIA, was used to measure the peak memory usage.

Watts up PRO, a Watt meter and power analyzer and electricity meter, was used to measure the power consumption. The mode used while measuring the time is the headless mode. However, measuring power with Watts up requires the wiring of the display mode. In this mode, the largest model we evaluated (YoloV4 FP32) was not able to execute until completion due to lack of memory, therefore the power measurement for that model is missing. Notice that this is a different method than that used for other NVIDIA platforms and thus cannot be directly compared to the other platforms.

3.4.14.3 Results

We present results for the supported precisions FP16 and FP32 and also for batch size of 1 and 8.

Table 76: ResNet50 (Nano)

Performance	FP16		FP32	
Batchsize	1	8	1	8
Inference time [ms]	39.9	269.1	73.9	490.7
Achieved performance [IPS]	25.0	29.7	13.5	16.3
Achieved performance [GOPS]	197.2	230.8	105.3	126.8
Peak performance [GOPS]	472.0		236.0	
Performance ratio	41.8 %	48.9 %	44.6 %	53.7 %
Memory utilization [GB]	2.1	2.2	2.2	2.3
Power [W]	5.7	5.8	5.9	5.9

Idle power [W]	1.7	1.7	1.7	1.7
Inference energy [mJ]	227.9	195.1	435.8	361.9
GOPS/W	34.6	39.8	17.9	21.5

Table 77: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Jetson Nano

Classification accuracy	FP16	FP32
Top 1	71.11 %	71.07 %
Top 5	89.81 %	89.83 %

Table 78: MobileNet (Nano)

Performance	**FP16**		**FP32**	
Batchsize	1	8	1	8
Inference time [ms]	9.6	42.7	11.7	58.5
Achieved performance [IPS]	103.9	187.3	85.0	136.1
Achieved performance [GOPS]	18.5	36.7	15.2	26.1
Peak performance [GOPS]	472.0		236.0	
Performance ratio	3.9 %	7.8 %	6.4 %	11.1 %
Memory utilization [GB]	1.9	2.1	1.9	2.1
Power [W]	5.1	5.2	5.2	5.2
Idle power [W]	1.7	1.7	1.7	1.7
Inference energy [mJ]	49.1	27.8	60.8	38.0
GOPS/W	3.6	7.1	2.9	5.0

Table 79: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Jetson Nano

Classification accuracy	FP16	FP32
Top 1	54.28 %	54.32 %
Top 5	79.12 %	79.12 %

Table 80: YoloV4 (Nano)

Performance	FP16		FP32	
Batchsize	1	8	1	8
Inference time [ms]	286.5	1961.0	488.4	3673.0
Achieved performance [IPS]	3.5	4.1	2.0	2.2
Achieved performance [GOPS]	210.8	246.4	123.7	131.4
Peak performance [GOPS]	472.0		236.0	
Performance ratio	44.7 %	52.2 %	52.4 %	55.7 %
Memory utilization [GB]	2.2	2.4	2.4	2.7
Power [W]	5.9	6.1	-	-
Idle power [W]	1.7	1.7	1.7	1.7
Inference energy [mJ]	1690.5	1495.2	-	-
GOPS/W	35.7	40.4	-	-

Table 81: mAP accuracies for YoloV4 on NVIDIA Jetson Nano

Classification accuracy	FP16	FP32
mAP (.50)	49.4 %	49.4 %
mAP (.50:.95)	74.2 %	74.2 %

3.4.15 NVIDIA GTX1660

The NVIDA GTX1660 PCIe-based graphics card was plugged into a x86-based desktop computer. The standard Gen.3 x16 port was used.

Desktop system

- Intel Core-I7 10700K
  - 8 cores @ 2.9 GHz
  - 16 threads
  - 16 MB Smart Cache
- 32 GB DDR4-3200 RAM
- 256 GB NVMe SSD HDD

3.4.15.1 Peak Performance

Theoretical peak performance of NVIDIA-GTX1660 Ti is reported to be 11.0 TFLOPS for FP16 and 5.5 TFLOPS for FP32 [69]. NVIDIA specs also mention 5.5 TOPS for INT32. The model used in our evaluation though is not the GTX1660Ti but instead the GTX1660r which is reported to have slightly lower theoretical performance of 5.0 TFLOPS for FP32 [70]. Based on this info we estimate the peak performance to be 10.0 TFLOPS for FP16 and 20.0 TOPS for INT8.

3.4.15.2 Toolflow

In order to create inference engines and run inference on GTX1660r, version 8.0.1 of NVIDIA's TensorRT library was used. TensorRT comes with a profiling tool called trtexec, which allows for latency, throughput and layer-wise benchmarking of neural networks in TensorRT representation. Onnx versions of model set v1 were used to create TensorRT inference engines and to measure timing using trtexec.

Nvidia-smi, a utility from NVIDIA, was used to measure power consumption and peak memory usage. Please note that in contrast to all previous measurements, nvidia-smi only measures the graphics card power and not the total system power which leads to a better efficiency.

3.4.15.3 Results

Table 82: ResNet50 (GTX1660)

Performance	INT8		FP16		FP32	
Batchsize	1	8	1	8	1	8
Inference time [ms]	1.0	5.0	1.7	8.6	3.0	15.8
Achieved performance [IPS]	990.1	1612.9	595.6	930.2	330.8	507.0
Achieved performance [GOPS]	7702.0	12548.0	4634.0	7237.0	1574.0	3944.0
Peak performance [GOPS]	20000.0		10000.0		5000.0	
Performance ratio	38.5 %	62.7 %	46.3 %	72.4 %	31.5 %	78.9 %
Memory utilization [GB]	0.7	0.7	0.8	0.9	0.9	1.0
Power [W]	100.4	101.0	101.2	101.0	101.9	103.0
Idle power [W]	10.0	10.0	10.0	10.0	10.0	10.0
Inference energy [mJ]	101.4	62.6	169.9	108.6	308.0	203.2
GOPS/W	76.7	124.2	45.8	71.7	15.5	38.3

Table 83: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA GTX-1660

Classification accuracy	INT8	FP16	FP32
Top 1	71.03 %	71.06 %	71.05 %
Top 5	89.93 %	89.82 %	89.82 %

Table 84: MobileNetV3 (GTX1660)

Performance	INT8		FP16		FP32	
Batchsize	1	8	1	8	1	8
Inference time [ms]	0.4	0.9	0.6	1.4	0.6	1.8
Achieved performance [IPS]	2405.6	9177.5	1778.1	5867.3	1722.7	4530.0
Achieved performance [GOPS]	428.2	1633.6	316.5	1044.4	306.6	806.5
Peak performance [GOPS]	20000.0		10000.0		5000.0	
Performance ratio	2.1 %	8.2 %	3.2 %	10.4 %	6.1 %	16.1 %
Memory utilization [GB]	0.7	0.7	0.7	0.7	0.7	0.7
Power [W]	63.0	90.0	73.3	100.0	75.4	103.0
Idle power [W]	10.0	10.0	10.0	10.0	10.0	10.0
Inference energy [mJ]	26.2	9.8	41.2	17.0	43.8	22.7
GOPS/W	6.8	18.2	4.3	10.4	4.1	7.8

Table 85: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA GTX-1660

Classification accuracy	INT8	FP16	FP32
Top 1	32.91 %	54.33 %	54.32 %
Top 5	58.65 %	79.17 %	79.14 %

Table 86: YoloV4 (GTX1660)

Performance	INT8	FP16	FP32

Batchsize	1	8	1	8	1	8
Inference time [ms]	6.0	33.5	9.4	63.1	17.8	123.8
Achieved performance [Inferences/s]	167.3	239.2	106.6	126.8	56.1	64.6
Achieved performance [GOPS]	10107.1	14444.8	6437.2	7659.8	3388.1	3903.8
Peak performance [GOPS]	20000.0		10000.0		5000.0	
Performance ratio	50.5 %	72.2 %	64.4 %	76.6 %	67.8 %	78.1 %
Memory utilization [GB]	0.8	1.0	1.1	1.6	1.9	2.4
Power [W]	100.8	100.0	102.4	101.0	102.8	101.0
Idle Power [W]	10.0	10.0	10.0	10.0	10.0	10.0
Inference energy [mJ]	602.4	418.1	960.8	796.4	1832.6	1562.7
GOPS/W	100.3	144.5	62.9	75.8	33.0	38.7

Table 87: mAP accuracies for YoloV4 on NVIDIA GTX-1660

Classification accuracy	INT8	FP16	FP32
mAP (.50)	47.9 %	49.4 %	49.4 %
mAP (.50:.95)	73.5 %	74.2 %	74.2 %

3.4.16 NVIDIA Tesla V100

The NVIDA Tesla V100 PCIe-based graphics card was plugged into an x86-based server system containing 8x V100 cards. The standard Gen.3 x16 PCIe port was used.

Server system

- 2x Intel Xeon Silver 4114
  - 10 cores @ 2.2 GHz
  - 13.75 MB L3 Cache
- 384 GB DDR4-2400 RAM (12 Channels)
- 2 PByte BeeGFS based NAS

3.4.16.1 Peak Performance

Theoretical peak performance of the NVIDIA Tesla V100 is reported to be 112 TFLOPS for DL applications, 29.4 TFLOPS for FP16 and 15.7 TFLOPS for FP32 [71]. Based on this info we estimate the peak performance to be 58.8 TOPS for INT8.

3.4.16.2 Toolflow

In order to create inference engines and run inference on V100, version 8.4.3.1 of NVIDIA's TensorRT library was used. TensorRT comes with a profiling tool called trtexec, which allows for latency, throughput and layer-wise benchmarking of neural networks in TensorRT representation. Onnx versions of model set v1 were used to create TensorRT inference engines and to measure timing using trtexec.

Nvidia-smi, a utility from NVIDIA, was used to measure power consumption and peak memory usage. Please note that in contrast to all previous measurements, nvidia-smi only measures the graphics card power and not the total system power which leads to a better efficiency.

3.4.16.3 Results

Table 88: ResNet50 (V100)

Performance	INT8		FP16		FP32	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	0.909	2.933	0.994	2.413	2.019	6.089
Achieved performance [IPS]	1099.85	2727.13	1006.26	3314.73	495.22	1313.80
Achieved performance [GOPS]	8556.80	21217.06	7828.72	25788.64	3852.83	10221.34
Peak performance [GOPS]	250000		125000		15700	
Performance ratio	3.42 %	8.49 %	6.26 %	20.63 %	24.54 %	65.10 %
Memory utilization [GB]	0.53 GB	0.55 GB	0.67 GB	0.70 GB	0.96 GB	1.03 GB
Power [W]	140.644	198.877	127.277	234.712	176.825	278.280
Idle power [W]						
Inference energy [mJ]	127.88	72.93	126.48	70.81	357.06	211.81
GOPS/W	60.84	106.68	61.51	109.87	21.79	36.73

Table 89: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Tesla V100

Classification accuracy	INT8	FP16	FP32
Top 1	71.05 %	71.03 %	71.05 %
Top 5	89.82 %	89.82 %	89.82 %

Table 90: MobileNetV3 (V100)

Performance	INT8		FP16		FP32	
Batchsize	1	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	1.055	1.497	0.763	1.214	0.999	1.640
Achieved performance [IPS]	947.79	5345.27	1311.19	6590.16	1001.13	4877.78
Achieved performance [GOPS]	168.71	951.46	233.39	1173.05	178.20	868.24
Peak performance [GOPS]	250000		125000		15700	
Performance ratio	0.07 %	0.38 %	0.19 %	0.94 %	1.14 %	5.53 %
Memory utilization [GB]	0.96 GB	0.68 GB	0.86 GB	0.52 GB	0.68 GB	0.89 GB
Power [W]	78.503	112.477	75.234	125.984	81.088	140.636
Idle power [W]						
Inference energy [mJ]	82.83	21.04	57.38	19.12	81.00	28.83
GOPS/W	2.15	8.46	3.10	9.31	2.20	6.17

Table 91: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Tesla V100

Classification accuracy	INT8	FP16	FP32
Top 1	29.59 %	54.29 %	54.32 %
Top 5	54.22 %	79.15 %	79.14 %

Table 92: YoloV4 (V100)

Performance	INT8		FP16		FP32	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	5.864	32.471	5.605	30.589	12.735	83.992
Achieved performance [Inferences/s]	170.54	246.37	178.41	261.53	78.52	95.25

Achieved performance [GOPS]	10300.42	14880.79	10775.99	15796.38	4742.86	5752.95
Peak performance [GOPS]	250000		125000		15700	
Performance ratio	4.12 %	5.95 %	8.62 %	12.64 %	30.21 %	36.64 %
Memory utilization [GB]	0.63 GB	0.95 GB	0.71 GB	1.27 GB	1.12 GB	2.19 GB
Power [W]	252.060	288.125	241.740	289.232	289.460	290.360
Idle Power [W]						
Inference energy [mJ]	1478.04	1169.48	1354.97	1105.93	3686.26	3048.48
GOPS/W	40.86	51.65	44.58	54.61	16.39	19.81

Table 93: mAP accuracies for YoloV4 on NVIDIA Tesla V100

Classification accuracy	INT8	FP16	FP32
mAP (.50)	48.3 %	49.4 %	49.4 %
mAP (.50:.95)	73.7 %	74.2 %	74.2 %

3.4.17 NVIDIA Tesla A100

The NVIDA Tesla A100 PCIe-based graphics card was plugged into an x86-based server system containing 8x A100 cards. The standard Gen.3 x16 PCIe port was used.

Server system

- 2x AMD EPYC 7313
  - 16 cores @ 3.0 GHz
  - 128 MB L3 Cache
- 512 GB DDR4-3200 RAM (16 Channels)
- 2 PByte BeeGFS based NAS

3.4.17.1 Peak Performance

Theoretical peak performance of the NVIDIA Tesla A100 is reported to be 312 TFLOPS for DL applications, 78 TFLOPS for FP16 and 39 TFLOPS for FP32 [72]. Based on this info we estimate the peak performance to be 156 TOPS for INT8.

3.4.17.2 Toolflow

In order to create inference engines and run inference on GTX1660r, version 8.0.1 of NVIDIA's TensorRT library was used. TensorRT comes with a profiling tool called trtexec, which allows for latency, throughput and layer-wise benchmarking of neural networks in TensorRT representation. Onnx versions of model set v1 were used to create TensorRT inference engines and to measure timing using trtexec.

Nvidia-smi, a utility from NVIDIA, was used to measure power consumption and peak memory usage. Please note that in contrast to all previous measurements, nvidia-smi only measures the graphics card power and not the total system power which leads to a better efficiency.

3.4.17.3 Results

Table 94: ResNet50 (A100)

Performance	INT8		FP16		FP32	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	0.501	0.933	0.640	1.141	0.906	1.986
Achieved performance [IPS]	1996.01	8574.49	1562.50	7011.39	1103.75	4028.20
Achieved performance [GOPS]	15528.94	66709.54	12156.25	54548.64	8587.20	31339.38
Peak performance [GOPS]	624000		312000		156000	
Performance ratio	2.49 %	10.69 %	3.90 %	17.48 %	5.50 %	20.09 %
Memory utilization [GB]	1.057	0.693	1.081	0.731	1.129	1.185
Power [W]	120	187	137	234	156	271
Idle power [W]						
Inference energy [mJ]	60.12	21.81	87.68	33.37	141.34	67.28
GOPS/W	129.41	356.74	88.73	233.11	55.05	115.64

Table 95: Top-1 and Top-5 accuracies for ResNet50 on NVIDIA Tesla A100

Classification accuracy	INT8	FP16	FP32
Top 1	71.04 %	71.05 %	71.04 %
Top 5	89.83 %	89.82 %	89.83 %

Table 96: MobileNetV3 (A100)

Performance	INT8		FP16		FP32	
Batchsize	1	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	0.524	0.702	0.541	0.805	0.580	1.044
Achieved performance [IPS]	1908.40	11396.01	1848.43	9937.89	1724.14	7662.84
Achieved performance [GOPS]	339.69	2028.49	329.02	1768.94	306.90	1363.98
Peak performance [GOPS]	624000		312000		156000	
Performance ratio	0.05 %	0.33 %	0.11 %	0.57 %	0.20 %	0.87 %
Memory utilization [GB]	0.653	0.665	0.653	0.667	0.657	1.099
Power [W]	87	112	89	118	98	137
Idle power [W]						
Inference energy [mJ]	45.59	9.83	48.15	11.87	56.84	17.88
GOPS/W	3.90	18.11	3.70	14.99	3.13	9.96

Table 97: Top-1 and Top-5 accuracies for MobileNetV3small on NVIDIA Tesla A100

Classification accuracy	INT8	FP16	FP32
Top 1	36.09 %	63.18 %	63.15 %
Top 5	61.95 %	84.82 %	84.82 %

Table 98: YoloV4 (A100)

Performance	INT8		FP16		FP32	
Batchsize	**1**	**8**	**1**	**8**	**1**	**8**
Inference time [ms]	2.142	9.774	2.491	12.656	4.238	23.173

Achieved performance [Inferences/s]	466.85	818.50	401.45	632.11	235.96	345.23
Achieved performance [GOPS]	28197.95	49437.28	24247.29	38179.52	14252.01	20851.85
Peak performance [GOPS]	624000		312000		156000	
Performance ratio	4.52 %	7.92 %	7.77 %	12.24 %	9.14 %	13.37 %
Memory utilization [GB]	0.765	1.087	0.857	1.417	1.473	2.519
Power [W]	207	351	265	396	298	398
Idle Power [W]						
Inference energy [mJ]	443.39	428.83	660.12	626.47	1262.92	1152.86
GOPS/W	136.22	140.85	91.50	96.41	47.83	52.39

Table 99: mAP accuracies for YoloV4 on NVIDIA Tesla A100

Classification accuracy	INT8	FP16	FP32
mAP (.50)	48.1 %	49.4 %	49.4 %
mAP (.50:.95)	73.8 %	74.2 %	74.2 %

3.4.18 Xilinx Ultrascale FPGAs

For a first evaluation of the performance and energy efficiency of FPGAs for ML inference, the Xilinx DPU (cf. Chapter 3.2.6.10) is used on state of the Art Xilinx UltraScale+ FPGAs. The used reconfigurable SoCs combine Quad Arm Cortex-A53 cores with an FPGA fabric in one device. Two FPGA boards have been utilized for the measurements, spanning from low-cost (Avnet Ultra96-v2 equipped with a Xilinx Zynq UltraScale+ ZU3EG) to mid-range reconfigurable SoCs (Trenz UltraSOM+ ZU15EG equipped with a Xilinx Zynq UltraScale+ XCZU15EG).

Trenz UltraSOM+ ZU15EG

- Trenz TE0808-04-BBE21-AS UltraSOM+ MPSoC Module
  - Xilinx Zynq UltraScale+ XCZU15EG-1FFVC900E FPGA
    - Quad-core Arm Cortex-A53, Dual-core Arm Cortex-R5F, Mali-400MP2
    - 341k 6-input look-up tables (LUTs)
    - 682k Flip-Flops(FFs)
    - 3,528 DSP Slices
    - 744 36kb BRAM blocks (26.2 Mb total)
    - 112 288kb URAM blocks (31.5 Mb total)
  - 4 GByte 64-Bit DDR4 SDRAM

    - 128 MByte SPI Boot Flash
- Trenz TEBF0808-04A UltraITX+ Baseboard for TE080X UltraSOM+
    - Mini-ITX form factor
    - PCIe, CAN FD transceiver, DisplayPort, SATA, 3 x USB 3.0
    - FMC HPC slot
    - Gigabit Ethernet RGMII PHY with RJ45 MagJack
    - MicroSD/MMC card socket (bootable)
    - 32 GBit (4 GByte) on-board eMMC Flash
- Cost of the complete setup: 3250$

Avnet Ultra96-v2

- Xilinx Zynq UltraScale+ MPSoC ZU3EG A484
    - Quad-core Arm Cortex-A53, Dual-core Arm Cortex-R5F, Mali-400MP2
    - 71k 6-input look-up tables (LUTs)
    - 141k Flip-Flops(FFs)
    - 360 DSP Slices
    - 216 36kb BRAM blocks (7.6 Mb total)
- 2 GB (512M x32) LPDDR4 Memory
- MicroSD card socket (bootable)
- Wi-Fi / Bluetooth
- Mini DisplayPort
- 1x USB 3.0 Type Micro-B upstream port
- 2x USB 3.0, 1x USB 2.0 Type A downstream ports
- Cost of the complete setup: 300€

The indicated cost for the systems includes the required power supplies, cooling, etc.

3.4.18.1 Peak Performance

The theoretical maximum performance that can be achieved depends on type and number of utilized DPU cores. In DPU designs with multiple DPU cores, the distribution between the cores is managed by the Xilinx Runtime library. For evaluation, we have used several configurations based on three different DPU cores, ranging from the smallest one (B512) to the largest (B4096). The number behind the "B" corresponds to the peak operations per clock cycle for the DPU core (INT8). Table 100 shows the theoretical peak performance for the used DPU cores at a clock frequency of 300 MHz and 200 MHz, respectively. While all developed FPGA designs in principle run at clock frequencies of 300 MHz and above, power limitations of the used boards limit most implementations to 200 MHz.

At design time, various parameters for the DPU configurations can be chosen, like high/low DPS usage or dedicated low power modes. For our implementations, the values have been chosen to provide maximum performance with the given power constraints.

Table 100: Peak performance of the DPU configurations used for evaluation at a clock frequency of 300 MHz and 200 MHz

DPU Name	Peak ops/clock	Peak performance (300 MHz) [GOPS]	Peak performance (200 MHz) [GOPS]
B512	512	153.6	102.4
B2304	2304	691.2	460.8
B4096	4096	1228.8	819.2

3.4.18.2 Tool Flow

For the FPGA development, Xilinx Vitis AI version 1.3 has been used together with Xilinx Vitis 2020.2. The designflow consists of two basic steps: (1) development of the target platform and (2) quantization and compilation of the AI model.

Development of the target platform

The target platform consists of a hardware design integrating the DPU and the required peripherals on the one hand and a board-specific Linux image, running on the embedded ARM on the other hand. The design flow consists of two main parts. First, the configuration and the number of cores of the DPU are specified together with the remainder of the targeted system design. Additional constrains such as the maximum clock frequency of the DPU on the used board have to be considered. After successful built of the hardware design, a Linux image needs to be created, which will run on the ARM cores of the reconfigurable SOC. The image has to be configured with the Xilinx runtime library for the DPU together with additional board-specific drivers.

Compilation of the AI model

The second flow consists of the preparation of the AI model for execution on the DPU. The model has to be present in a common AI-framework format (Tensoflow/Keras, Caffe or PyTorch). Hence, it can be trained for the intended use-case with the common tools provided by the used framework. The trained model is then quantized with the help of the Xilinx VITIS-AI library into an 8-bit Integer format. To compensate for accuracy loss due to quantization, additional training runs are performed, for which training data has to be provided. The last step, before the model can be loaded onto the reconfigurable SOC, is to compile the model for the used DPU. The resulting compiled model can be used on any FPGA, which contains a DPU with the exact same configuration, the model was compiled for.

3.4.18.3 Results

For performance evaluation, various implementations have been compared, differing in the DPU configuration, the number of integrated DPU cores, and the number of software threads. In the tables, presenting the results, single threaded refers to an implementation where pre-processing, DPU calls, and postprocessing are performed by a single thread. In the multithreaded implementations, the threads are evenly distributed between pre-processing, DPU-calls, and postprocessing.

3.4.18.3.1 ResNet50

For performance evaluation of ResNet50, a set of 1000 images has been used. Table 101 summarizes the results for the Trenz system and Table 102 the results achieved with the Ultra96-v2. For all analyzed nets, pre-processing includes reading the input data from the SD card and during postprocessing the results are stored in external DRAM.

For singlethread implementations, the latencies of the single steps (pre-processing, inference, and postprocessing) add up to the total time for one frame. When using multithreading, the latency increases but the throughput (inferences/s) also significantly grows. In the tables, B2304 x3, 200 MHz, refers to an FPGA implementation with a DPU that integrates three B2304 DPU cores, running at 200 MHz. The theoretical peak performance is based on the DPU core, the number of instances and the clock frequency.

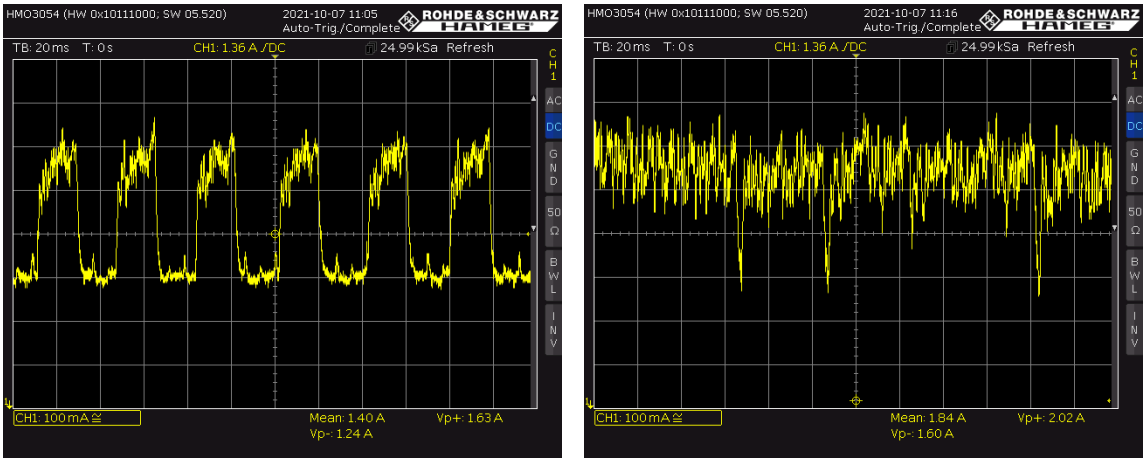

Figure 53: Example of current measurements for an FPGA configuration including two B4096 DPUs running at 200MHz on the Xilinx XCZU15EG-1 FPGA – single-threaded (left) and using 12 threads (right)

Power measurements for all experiments have been performed with high-speed current-locking, as discussed in Chapter 3.1.3. Complete system power has been measured, covering all components of the hardware platforms. Two examples are provided in Figure 53. The measurement of the single-threaded implementation on the left clearly shows when the DPU is active and reveals the low utilization of the core. The figure on the right shows a measurement with the same time resolution, where a multithreaded software implementation provides enough data to keep the multicore DPU continuously busy (the right figure has a different offset for the measured power). While in general the average power is most interesting, the graphs provide interesting insights into the running application, which will be of high interest especially for the own FPGA implementations that are planned in the project.

For the FPGA platforms, idle power has been measured in two system states and data for both is provided in the subsequent tables:

1) The system is powered on but the reconfigurable SoC is not yet configured
2) The system is powered on, the reconfigurable SoC is configured and Linux has finished booting on the embedded Arm cores but ML inference has not yet started

The results in the subsequent tables show that a high performance ratio can be achieved, especially when multithreading the implementation. It needs to be noticed that the batch size is one for all reported FPGA implementations. As can be seen in the tables, there is a trade-off between high throughput (Inferences/s) and low latency since increasing the number of threads significantly increases the throughput but at the same time increases the latency.

Table 101: Evaluation of ResNet50 on the Trenz UltraSOM+ ZU15EG FPGA system

Platform	Trenz UltraSOM+ ZU15EG					
DPU	B2304 x3, 200MHz			B4096 x2, 200MHz		
Number of threads	1	9	18	1	6	12
Inference Time [ms]	30.64	31.96	61.61	20.44	20.99	39.47

Latency [ms]	48.82	50.99	82.41	38.74	39.34	58.81
Achieved performance [Inferences/s]	20.48	93.37	96.62	25.81	94.55	100.66
Achieved performance [GOPS]	159.36	726.38	751.73	200.81	735.57	783.10
Peak performance [Gops/s]	1382.4	1382.4	1382.4	1638.4	1638.4	1638.4
Performance Ratio	11.53 %	52.55 %	54.38 %	12.26 %	44.90 %	47.80 %
Cost Metrics						
FPGA Resources	124k (36.6 %) LUTs, 216k (31.6 %) FFs, 1,302 (36.9 %) DSP, 333 (44.8 %) BRAM, 72 (64.3 %) URAM			102k (29.8 %) LUTs, 201k (29.4 %)FFs, 1,412 (40.0 %) DSP, 342 (46.0 %) BRAM, 48 (42.9 %) URAM		
Power [W]	15.26	20.94	21.38	15.26	20.54	21.26
Idle Power [W][1]	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6
Energy/Inference [J]	0.745	0.224	0.221	0.591	0.217	0.211

Table 102: Evaluation of ResNet50 on the Avnet Ultra96-v2 FPGA system

Platform	**Avnet Ultra96-v2**					
DPU	**B2304 x1, 200MHz**		**B512 x1, 200MHz**		**B512 x1, 300MHz**	
Number of threads	1	12	1	12	1	12
Inference Time [ms]	40.31	158.33	100.95	395.03	72.78	284.42
Latency [ms]	57.86	180.78	118.25	414.44	90.49	304.82
Achieved performance [Inferences/s]	17.28	25.19	8.45	10.11	11.04	14.04
Achieved performance [GOPS]	134.46	196.00	65.73	78.62	85.86	109.20
Peak performance [Gops/s]	460.8	460.8	102.4	102.4	153.6	153.6
Performance Ratio	29.18 %	42.54 %	64.19 %	76.78 %	55.90 %	71.09 %
Cost Metrics						
FPGA Resources	56k (79.7 %) LUTs, 91k (64.2 %) FFs, 326 (90.6 %) DSP, 171 (79.2 %) BRAM		41k (58.5 %) LUTs, 52k (36.8 %) FFs, 78 (21.7 %) DSP, 77.5 (35.9 %) BRAM			

[1] The first value is measured while the reconfigurable SoC is unconfigured, the second is measured after configuration with Linux completely booted

Power [W]	6.756	7.368	5.496	5.532	5.928	6.144
Idle Power [W]	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4
Energy/Inference [J]	0.391	0.292	0.651	0.547	0.537	0.438

In the following table, the achieved accuracies are provided for the implementations discussed above. The results are independent of the used DPU and hardware platform.

Table 103: Top-1 and Top-5 accuracies for ResNet50 using the Xilinx DPU on UltraScale+

Classification accuracy	INT8
Top 1	73.99 %
Top 5	91.70 %

3.4.18.3.2 YoloV4

For performance evaluation of YoloV4, a set of 404 images has been used. Table 104 summarizes the results for the Trenz system and Table 105 the results achieved with the Ultra96-v2.

Table 104: Evaluation of YoloV4 on the Trenz UltraSOM+ ZU15EG FPGA system

Platform	**Trenz UltraSOM+ ZU15EG**					
DPU	**B2304 x3, 200MHz**			**B4096 x2, 200MHz**		
Number of threads	1	9	18	1	6	12
Inference Time [ms]	198.50	202.78	369.79	118.11	119.38	205.16
Latency [ms]	218.88	225.11	397.32	141.21	139.26	225.02
Achieved performance [Inferences/s]	4.57	14.55	15.91	7.06	16.54	19.10
Achieved performance [GOPS]	275.95	878.75	960.95	426.64	999.02	1153.72
Peak performance [Gops/s]	1382.4	1382.4	1382.4	1638.4	1638.4	1638.4
Performance Ratio	19.96 %	63.57 %	69.51 %	26.04 %	60.98 %	70.42 %
Cost Metrics						
FPGA Resources	124k (36.6 %) LUTs, 216k (31.6 %) FFs, 1,302 (36.9 %) DSP, 333 (44.8 %) BRAM, 72 (64.3 %) URAM			102k (29.8 %) LUTs, 201k (29.4 %)FFs, 1,412 (40.0 %) DSP, 342 (46.0 %) BRAM, 48 (42.9 %) URAM		
Power [W]	16.22	22.94	23.18	17.30	23.30	24.50
Idle Power [W]	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6
Energy/Inference [J]	3.551	1.577	1.457	2.450	1.409	1.283

Table 105: Evaluation of YoloV4 on the Avnet Ultra96-v2 FPGA system

Platform	Avnet Ultra96-v2				
DPU	B2304 x1, 200MHz			B512 x1, 200MHz	B512 x1, 300MHz
Number of threads	1	6	12	6	6
Inference Time [ms]	223.37	424.16	850.24	1375.72	956.64
Latency [ms]	246.62	472.35	900.41	1419.17	995.49
Achieved performance [Inferences/s]	4.05	4.69	4.67	1.45	2.08
Achieved performance [GOPS]	244.91	283.16	282.00	87.56	125.86
Peak performance [Gops/s]	460.8	460.8	460.8	102.4	153.6
Performance Ratio	53.15 %	61.45 %	61.20 %	85.50 %	81.94 %
Cost Metrics					
FPGA Resources	56k (79.7 %) LUTs, 91k (64.2 %) FFs, 326 (90.6 %) DSP, 171 (79.2 %) BRAM			41k (58.5 %) LUTs, 52k (36.8 %) FFs, 78 (21.7 %) DSP, 77.5 (35.9 %) BRAM	
Power [W]	7.88	8.22	8.22	5.64	6.31
Idle Power [W]	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4
Energy/Inference [J]	1.944	1.753	1.761	3.891	3.029

In the following table, the achieved accuracies are provided for the implementations discussed above. The results are independent of the used DPU and hardware platform.

Table 106: mAP accuracies for YoloV4 using the Xilinx DPU on UltraScale+

mAP	INT8
mAP (.50)	69.4 %
mAP (.50:.95)	39.0 %

3.4.18.3.3 MobileNetV2

Since, MobileNetV3 could not be compiled for the DPU, MobileNetV2 was used as an example of a small net. A set of 1000 images has been used. Table 107 summarizes the results for the Trenz system and Table 108 the results achieved with the Ultra96-v2. In comparison to the previous examples, the utilization, i.e., the achieved performance vs. the theoretical maximum, is quite low. This is mainly due to the I/O overhead since in this case pre-processing requires more time than the inference on the DPU. Obviously, a small implementation like the B512 on the Ultra96v2 is more suitable for such nets.

Table 107: Evaluation of MobileNetV2 on the Trenz UltraSOM+ ZU15EG FPGA system

Platform	Trenz UltraSOM+ ZU15EG			
DPU	**B2304 x3, 200MHz**		**B4096 x2, 200MHz**	
Number of threads	1	18	1	12
Inference Time [ms]	7.68	8.82	6.50	7.34
Latency [ms]	28.12	43.38	26.93	30.36
Achieved performance [Inferences/s]	35.56	169.19	37.04	169.31
Achieved performance [GOPS]	21.69	103.20	22.59	103.28
Peak performance [Gops/s]	1382.4	1382.4	1638.4	1638.4
Performance Ratio	1.57 %	7.47 %	1.38 %	6.30 %
Cost Metrics				
FPGA Resources	124k (36.6 %) LUTs, 216k (31.6 %) FFs, 1,302 (36.9 %) DSP, 333 (44.8 %) BRAM, 72 (64.3 %) URAM		102k (29.8 %) LUTs, 201k (29.4 %)FFs, 1,412 (40.0 %) DSP, 342 (46.0 %) BRAM, 48 (42.9 %)URAM	
Power [W]	13.82	15.62	13.88	15.74
Idle Power [W]	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6	6.6 / 12.6
Energy/Inference [J]	0.389	0.092	0.375	0.093

Table 108: Evaluation of MobileNetv2 on the Avnet Ultra96-v2 FPGA system

Platform	Avnet Ultra96-v2					
DPU	**B2304 x1, 200MHz**			**B512 x1, 300MHz**		
Number of threads	1	6	12	1	6	12
Inference Time [ms]	10.08	19.77	40.58	15.34	29.74	60.44
Latency [ms]	28.84	41.48	67.24	34.43	52.30	83.49
Achieved performance [Inferences/s]	34.67	89.15	97.71	28.90	66.82	65.82
Achieved performance [GOPS]	21.15	54.38	59.60	17.63	40.76	40.15
Peak performance [Gops/s]	460.8	460.8	460.8	153.6	153.6	153.6
Performance Ratio	4.59 %	11.80 %	12.93 %	11.48 %	26.54 %	26.14 %

Cost Metrics						
FPGA Resources	56k (79.7 %) LUTs, 91k (64.2 %) FFs, 326 (90.6 %) DSP, 171 (79.2 %) BRAM			41k (58.5 %) LUTs, 52k (36.8 %) FFs, 78 (21.7 %) DSP, 77.5 (35.9 %) BRAM		
Power [W]	5.868	6.636	6.756	5.58	6.06	6.108
Idle Power [W]	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4	3.2 / 5.4
Energy/Inference [J]	0.169	0.074	0.069	0.193	0.091	0.093

In the following table, the achieved accuracies are provided for the implementations discussed above. The results are independent of the used DPU and hardware platform.

Table 109: Top-1 and Top-5 accuracies for MobileNetV3 using the Xilinx DPU on UltraScale+

Classification accuracy	INT8
Top 1	65.23 %
Top 5	85.25 %

3.4.19 Xilinx Versal AI Core Series

As discussed in deliverable D4.1, the new Xilinx Versal devices are promising targets for integration into the RECS platforms. The reconfigurable SoCs combine an ARM processing system with a programmable logic fabric and a variety of I/O interfaces. In addition to the classical FPGA-based SoCs, these Adaptive Compute Acceleration Platforms (ACAP) integrate new DSP engines, optional AI engines and a network on chip infrastructure for communication between the heterogeneous computing resources. A Versal AI Core Series VCK190 Evaluation Kit has been utilized for performance, power and accuracy measurements. The system is equipped with an XCVC1902-2MSEVSVA2197 ACAP with the following characteristics:

- 400 AI Engines
- 1,968 DSP Engines
- 1,968k System Logic Cells
- 899,840 LUTs
- Dual-Core Arm Cortex-A72Application Processing Unit
- Dual-Core Arm Cortex-R5F Real-Time Processing Unit

3.4.19.1 Peak Performance

Like for the Xilinx UltraScale devices, designs based on the Xilinx DPU have been used for first evaluations. All implementations are based on the DPUCVDX8G using INT8 data representation. Six different architectural variants of the DPUCVDX8G have been evaluated, summarized in Table 100. In the naming, “C32” refers to the use of 32 AI Engine cores per batch handler while “C64” utilizes 64. The number behind the “B” represents the number of batch handlers. “L2S2” indicates that there are two load/store interfaces per batch handler. Finally, “x3” means that three instances of the DPU have been instantiated. The theoretical peak performance is provided for a PL clock frequency of 333MHz and 1.25GHz AIE clock frequency.

Table 110: Resource requirements and peak performance of the DPU configurations used for evaluation

DPU Architecture	AIE Cores	DSP Slices	BRAM	URAM	LUTs	FFs	Peak perf. [TOPS]
C32B1L2S2	32	139	0	136	83k	111k	10.24
C32B1L2S2x3	96	417	0	408	249k	334k	30.72
C32B3L2S2	96	407	0	264	210k	269k	30.72
C32B6L2S2	192	809	678	343	402k	507k	61.44
C64B1L2S2	64	139	0	136	94k	133k	20.48
C64B5L2S2	320	675	0	392	356k	468k	102.4

3.4.19.2 Tool Flow

The general tool flow is identical to the design flow for the UltraScale+ FPGAs, discussed in the previous Section. For the development, Xilinx Vitis AI version 2.5 has been used together with Xilinx Vitis 2022.1.

3.4.19.3 Results

For performance evaluation, various implementations based on different DPU architectures have been compared, as will be shown in the following. Power measurements are provided for the Versal device, excluding the supporting devices on the VCK190 Evaluation Kit. In addition to the power when executing the models, idle power is reported for the system state, where the system is powered on, the Versal device is configured and Linux has finished booting on the embedded Arm cores but ML inference has not yet started.

3.4.19.3.1 ResNet50

Table 111: Evaluation of ResNet50 on the Versal AI Core Series VCK190 Evaluation Kit

Platform	Versal AI Core Series VCK190 Evaluation Kit					
DPU	**C31B1**	**C32B1x3**	**C32B3**	**C32B6**	**C64B1**	**C64B5**
Inference Time [ms]	1.908	1.908	1.882	1.965	1.391	1.588
Latency [ms]	1.908	1.908	1.882	1.965	1.391	1.588
Achieved performance [Inferences/s]	524	1056	1594	3053	719	3149
Achieved performance [TOPS]	4.077	8.216	12.401	23.752	5.594	24.499
Peak performance [TOPS]	10.24	30.72	30.72	61.44	20.48	102.4
Performance Ratio	39.81 %	26.74 %	40.37 %	38.66 %	27.31 %	23.93 %
Idle Power [W]	14.90	23.66	19.82	27.86	16.22	30.62

Power [W]	20.30	34.70	32.42	51.62	23.42	55.46
Energy/Inference [mJ]	38.74	32.86	20.34	16.91	32.57	17.61
GOPS/Watt	200.82	236.76	382.52	460.14	238.85	441.75

Table 112: Top-1 and Top-5 accuracies for ResNet50 on the Versal AI Core Series VCK190 Evaluation Kit

Classification accuracy	INT8
Top 1	73.99 %
Top 5	91.70 %

3.4.19.3.2 YoloV4

Table 113: Evaluation of YoloV4 on the Versal AI Core Series VCK190 Evaluation Kit

Platform	Versal AI Core Series VCK190 Evaluation Kit					
DPU	**C31B1**	**C32B1x3**	**C32B3**	**C32B6**	**C64B1**	**C64B5**
Inference Time [ms]	17.241	17.241	19.737	30.769	12.821	22.831
Latency [ms]	17.241	17.241	19.737	30.769	12.821	22.831
Achieved performance [Inferences/s]	58	140	152	195	78	219
Achieved performance [TOPS]	3.503	8.456	9.181	11.778	4.711	13.228
Peak performance [TOPS]	10.24	30.72	30.72	61.44	20.48	102.4
Performance Ratio	34.21 %	27.53 %	29.89 %	19.17 %	23.00 %	12.92 %
Idle Power [W]	14.90	23.66	19.82	27.86	16.22	30.62
Power [W]	19.58	35.30	31.10	43.70	22.46	48.38
Energy/Inference [mJ]	337.59	252.14	204.61	224.10	287.95	220.91
GOPS/Watt	178.92	239.55	295.20	269.52	209.76	273.41

Table 114: mAP accuracies for YoloV4 on the Versal AI Core Series VCK190 Evaluation Kit

mAP	INT8
mAP (.50)	69.4 %
mAP (.50:.95)	39.0 %

3.4.19.3.3 MobileNetV3

Table 115: Evaluation of MobileNetV3 on the Versal AI Core Series VCK190 Evaluation Kit

Platform	Versal AI Core Series VCK190 Evaluation Kit					
DPU	**C31B1**	**C32B1x3**	**C32B3**	**C32B6**	**C64B1**	**C64B5**
Inference Time [ms]	0.664	0.664	0.987	1.685	0.629	1.503
Latency [ms]	0.664	0.664	0.987	1.685	0.629	1.503
Achieved performance [Inferences/s]	1505	2216	3038	3560	1590	3326
Achieved performance [TOPS]	0.268	0.394	0.541	0.634	0.283	0.592
Peak performance [TOPS]	10.24	30.72	30.72	61.44	20.48	102.4
Performance Ratio	2.62%	1.28%	1.76%	1.03%	1.38%	0.58%
Idle Power [W]	14.90	23.66	19.82	27.86	16.22	30.62
Power [W]	17.18	26.54	22.94	31.70	18.50	34.46
Energy/Inference [mJ]	11.42	11.98	7.55	8.90	11.64	10.36
GOPS/Watt	15.59	14.86	23.57	19.99	15.30	17.18

Table 116: Top-1 and Top-5 accuracies for MobileNetV3 on the Versal AI Core Series VCK190 Evaluation Kit

Classification accuracy	INT8
Top 1	65.23%
Top 5	85.25%

3.5 Comparison of DL Accelerators

The VEDLIoT project is trying to find the best suited accelerators for the given use-cases. An evaluation of all the ML hardware, that is available at the partners, was performed. The results are summed up in the following graphics. The graphics are categorized by the three ML models that were chosen for this evaluation (ResNet50, MobileNetV3, YoloV4) (cf. Chapter 3.1.2.1). The graphs show performance in GOPS over power in Watt and the results for different quantization are marked in multiple colours. An imaginary diagonal through the coordinate system represents energy efficiency in GOPS/W. In addition, energy efficiency is visualized in bar diagrams. In comparison to the results shown in D3.1, the list of accelerators was extended by High-Performance NVIDIA Tesla GPUs, the newer generation of NVIDIA Jetson AGX Orin SoCs, NVIDIA Orin NX SoCs and the Hailo.AI Hailo-8 PCIe-based DL accelerator. In addition the performance evaluation was validated by measuring the accuracy for all vendor specific DL toolchains.

As a result of this analysis it is approved that the **NVIDIA Xavier NX**, chosen for the new VEDLIoT hardware **u.RECS**, is a good choice in terms of efficiency and that classical x86 lack far behind for ML applications. In addition the **Hailo.AI Hailo-8** is a perfect candidate for PCIe based acceleration of ML use-cases. In the following comparison the Hailo-8 results are displayed without the necessary host system, making it complicated to directly compare it with the other accelerators. Nevertheless, it is the most efficient ML-accelerator we could observe in the frame of the VEDLIoT project.

3.5.1 ResNet50

The performance results for the ResNet50 benchmarks are shown in Figure 54. Figure 55 is a more clearly representation of energy efficiency showing GOPS/W.

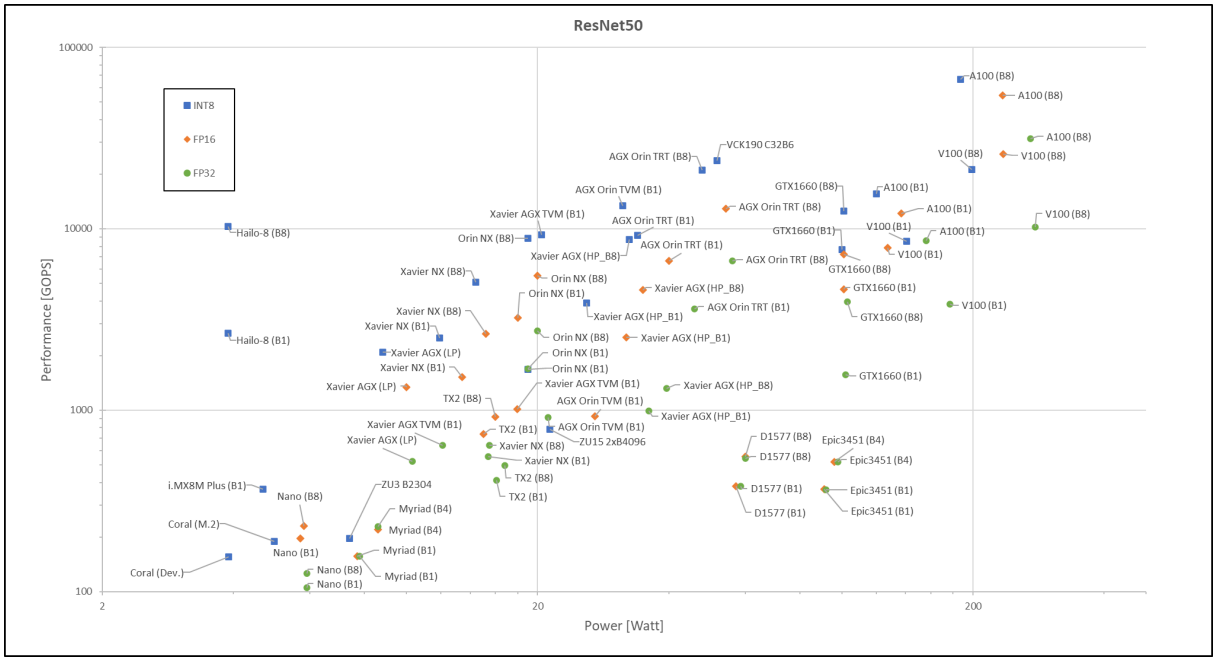

Figure 54: ResNet50 Performance Diagram

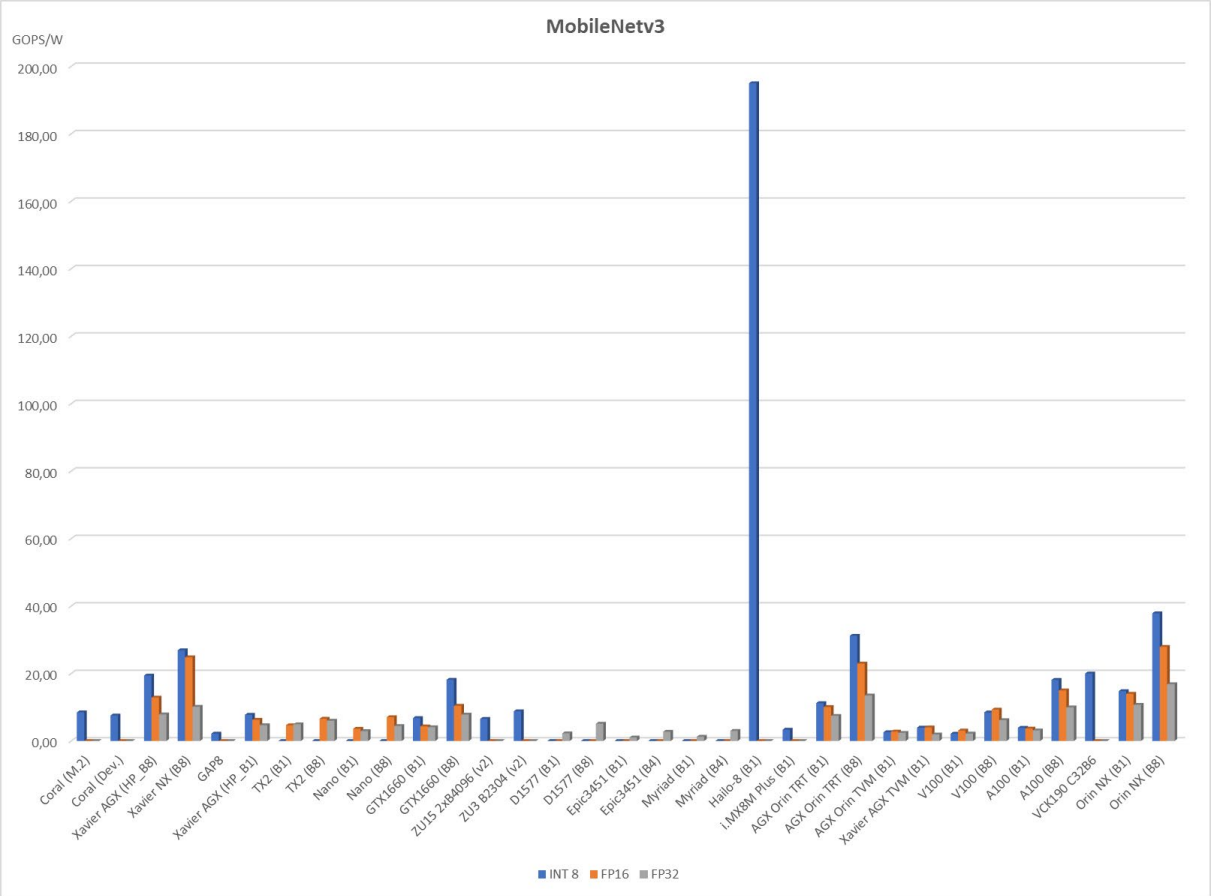

Figure 55: ResNet50 Efficiency Diagram

3.5.2 MobileNetV3 Small

The MobileNetV3 Small is the smallest model tested in the VEDLIoT project, the performance results are depicted in Figure 56. The power efficiency of all accelerators is shown in Figure 57.

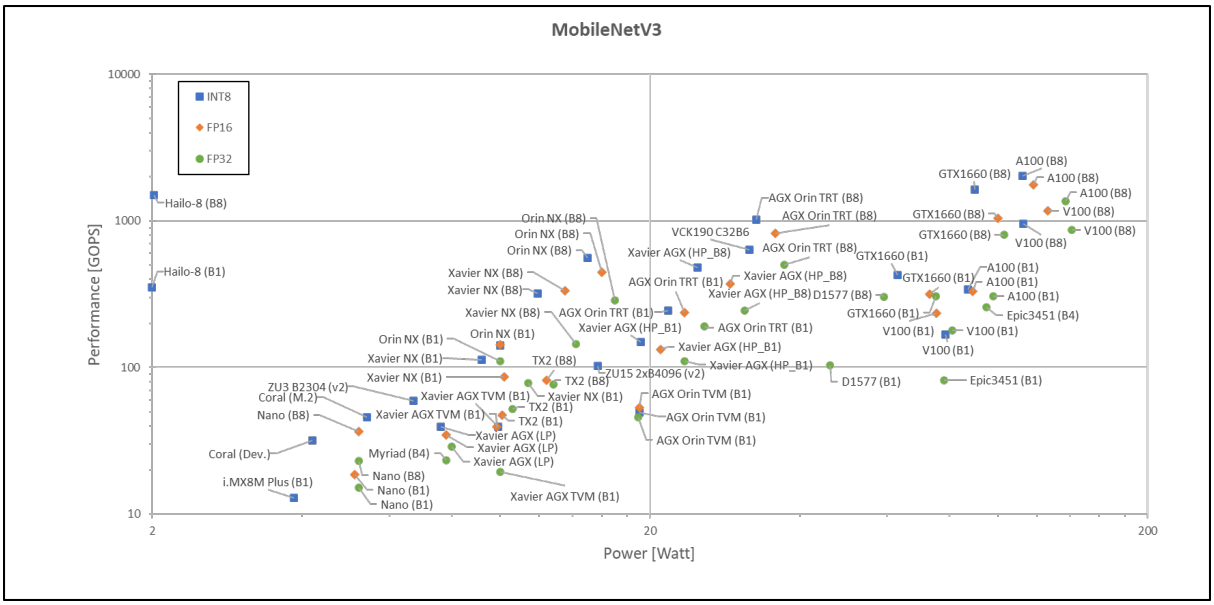

Figure 56: MobileNetV3 Performance Diagram

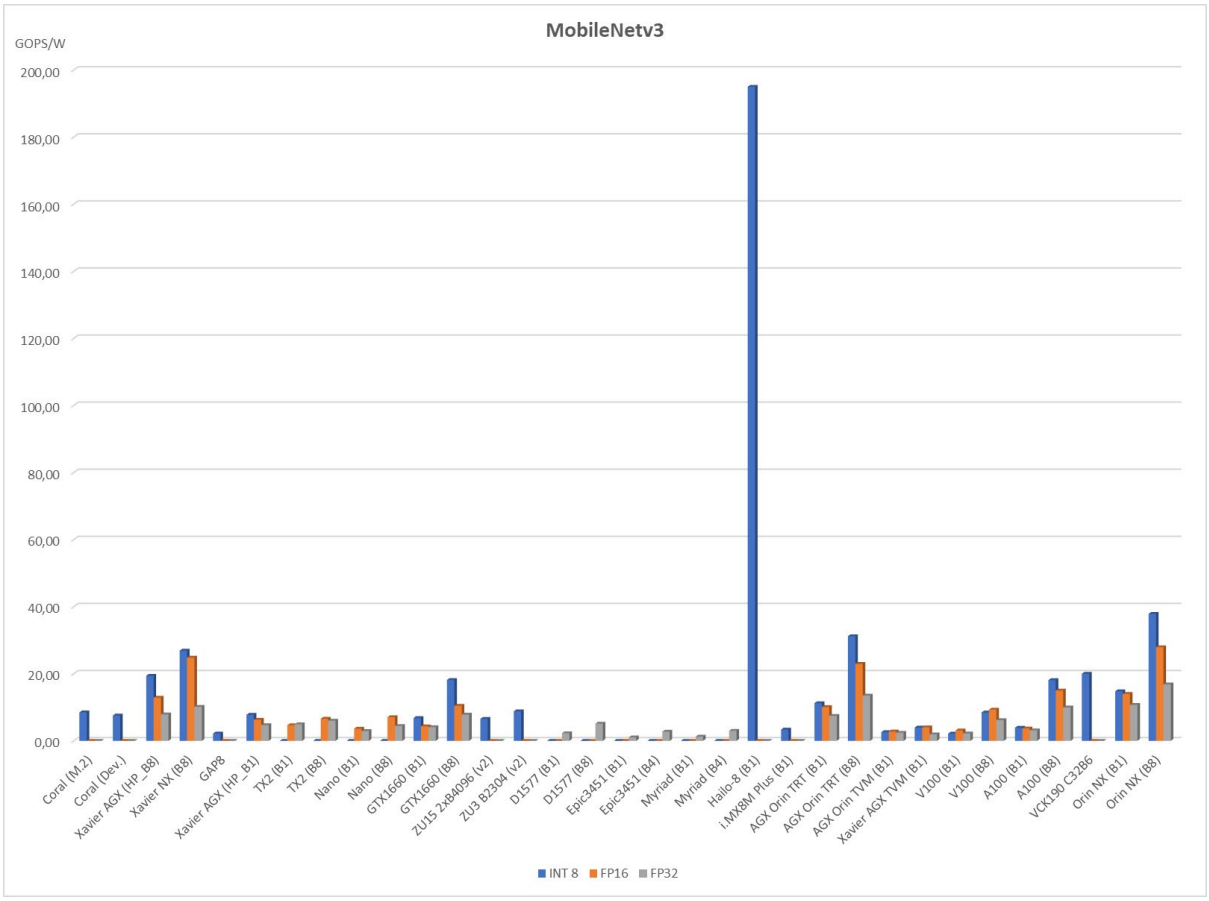

Figure 57: MobileNetV3 Efficiency Diagram

3.5.3 YoloV4

The YoloV4 model is the largest in the current set. The performance results of all tested accelerators are shown in Figure 58. The corresponding power efficiency is shown in Figure 59.

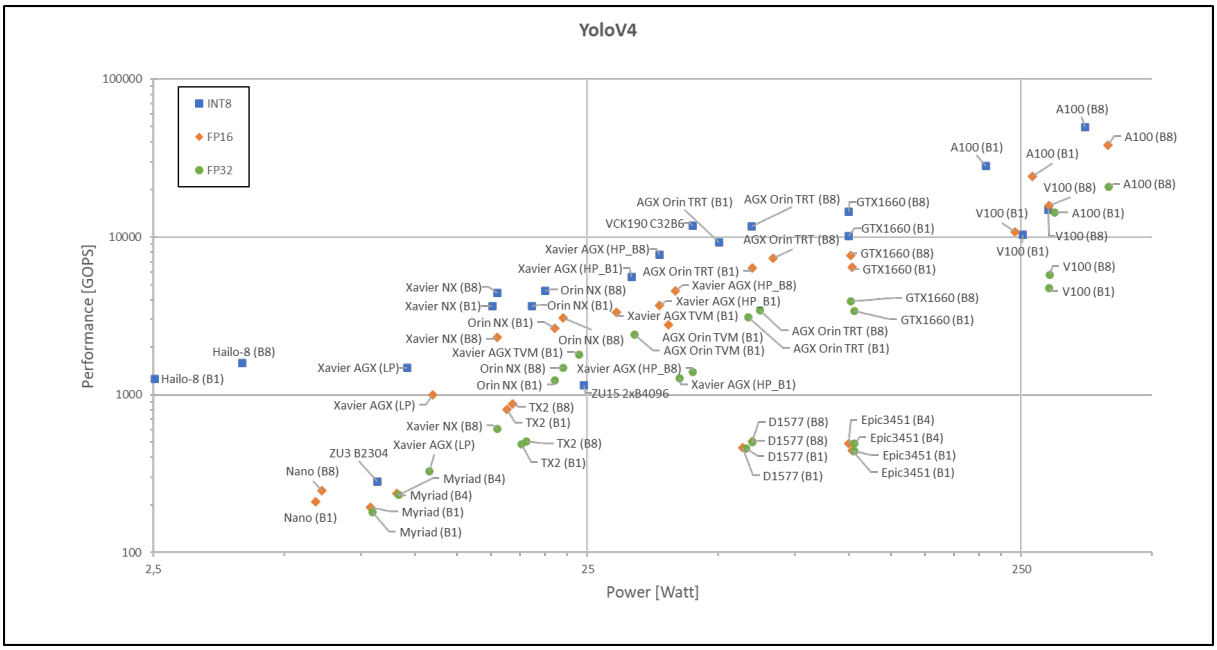

Figure 58: YoloV4 Performance Diagram

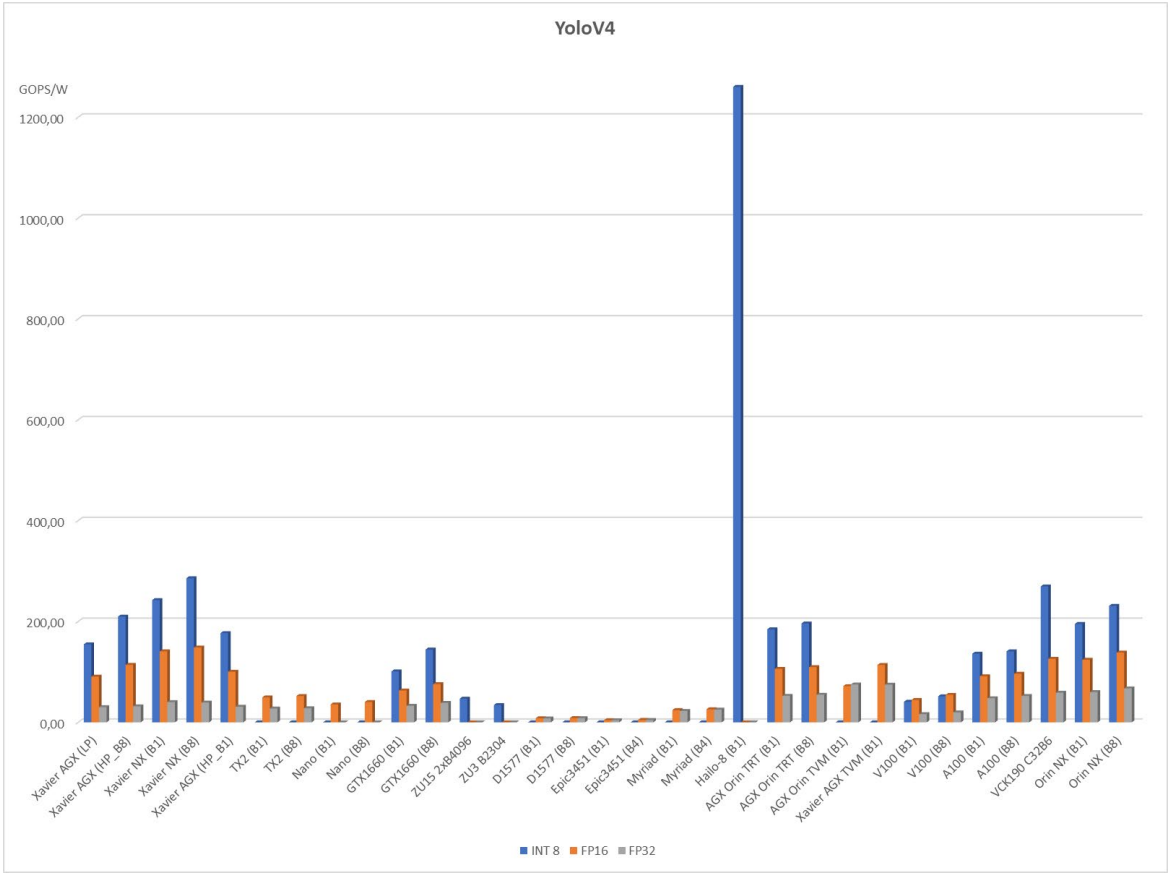

Figure 59: YoloV4 Efficiency Diagram

4 Conclusion

This report is a joint work of WP6 and WP3, providing an overview of the current state of the edge AI systems and how VEDLIoT can leverage and improve the existing ecosystem. The content has been produced by Task 6.1 (Survey of Deep Learning Inference Tools, Interfaces and Compilers) and Task 3.1 (Evaluation of existing architectures and compilers for DL), as the two tasks are related, covering different aspects of edge AI systems. This deliverable was previously released in an intermediate version as D3.1, which was extended by evaluation of newly available hardware and accuracy measurements for DL toolchains.

This report provides an overview of the current state of the art in edge AI processing, covering the typically used deployment stack and walks the reader through the detailed description of the most popular deep learning frameworks, platforms, and compilers and their position in the deep learning ecosystem. VEDLIoT has come up with a VEDLIoT model zoo, containing commonly used models for benchmarking and analysis, which is described in this report. The model zoo is continuously maintained and updated throughout the project in an attempt to include new developments in the project.

An extensive analysis of novel deep learning accelerator platforms is presented, covering both dedicated ASICs as well as IP Cores. The different architectures are briefly explained, covering key elements of processing, memory, interfaces, as well as performance and energy efficiency. A comparison at the end shows the relation of the different accelerators and their performance respective performance regions. On average, over all architectures, a power efficiency in the order of 1 TOPS/W is achieved, based on specifications extracted from information provided by the respective vendors.

In addition to the detailed analysis of novel deep learning accelerators, another huge part of the report covers an in-depth evaluation of available DL accelerator platforms, comparing the performance and energy-efficiency of different accelerators among each other's and against a commonly accepted baseline, based on benchmarking and measurement activities conducted within VEDLIoT. A methodology for measuring the different figures has been established as a common reference. Results show that the NVIDIA Jetson accelerators, particularly the Jetson NX, show high energy efficiency across different models and data formats of up to 350 GOPS/W. The benchmarks are confirmed by a comprehensive accuracy evaluation proving that all measurements are performed using representative data similar to the requirements of the use-cases.

The Kenning platform presented in this report aims to provide an interface for switching between various compilers depending on chosen hardware and models. Kenning provides support for a wide range of available technology, offering automated benchmarking, characterization, and deployment. Tight integration of Kenning with the VEDLIoT hardware platform is pursued.

In summary, this report presents an in-depth overview of deep learning tools, models and technology, combined with a sophisticated methodology for evaluation and benchmarking new accelerators in a heterogeneous hardware environment. While extensive benchmarking and research has been performed, this is still an activity to be continued beyond the frame of the VEDLIoT project, as most of the presented novel hardware platforms are not yet available for hands-on benchmarking, mainly due to supply-chain problems in the electronics industry, which are currently faced on a global scale.

5 References

[1] J. Roesch. [Online]. Available: URL: http://dx.doi.org/10.1145/3211346.3211348.

[2] "Apache Software Foundation," [Online]. Available: https://tvm.apache.org/docs/.

[3] G. Developers, "FlatBuffers Documentation," [Online]. Available: https://google.github.io/flatbuffers/index.html.

[4] G. Developers, "TensorFlow Lite Documentation," [Online]. Available: https://www.tensorflow.org/lite/guide?hl=en.

[5] A. Demidovskij, "learning workbench: comprehensive analysis and tuning of neural networks inference.," 2019.

[6] W.-F. Lin, " Onnc: a compilation framework connecting onnx to roprietary deep learning accelerators.," in *International Conference on Artificial Intelligence Circuits and Systems* , 2019.

[7] W.-F. Lin, "Onnc-based software development platform for configurable nvdla designs," in *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2019.

[8] N. Rotem, "Graph Lowering Compiler Techniques for Neural Networks," 2019.

[9] P. Mattson et al., "MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance," in *IEEE Micro, vol. 40, no. 2, pp. 8-16*, doi: 10.1109/MM.2020.2974843, 2020.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language," in *http://arxiv.org/abs/1810.04805*, CoRR, 2018.

[11] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770-778, doi: 10.1109/CVPR.2016.90, 2016.

[12] Chen, Liang-Chieh, et al., "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018.

[13] Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao, "Yolov4: Optimal speed and accuracy of object detection," in *arXiv preprint arXiv:2004.10934*, 2020.

[14] He, Kaiming, et al., "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017.

[15] A. Howard et al., "Searching for MobileNetV3," in *IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 1314-1324, doi: 10.1109/ICCV.2019.00140, 2019.

[16] Tan, Mingxing and Quoc Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning. PMLR*, 2019.

[17] Tan, Mingxing, Ruoming Pang, and Quoc V. Le., "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020.

[18] Ma, Ningning, et al., "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European conference on computer vision (ECCV)*, 2018.

[19] Diganta Misra, "Mish: {A} Self Regularized Non-Monotonic Neural Activation Function," CoRR, http://arxiv.org/abs/1908.08681, 2019.

[20] "ImageNet Dataset," [Online]. Available: https://www.image-net.org/. [Accessed 09 2022].

[21] "COCO Dataset," [Online]. Available: https://cocodataset.org/. [Accessed 09 2022].

[22] "COCO-API," [Online]. Available: https://github.com/cocodataset/cocoapi. [Accessed 09 2022].

[23] A. Jani, "Maxim Showcases Efficient Custom AI," *Microprocessor Report,* 2021.

[24] L. Gwennap, "Kneron Delivers Efficient AI," *Microprocessor Report,* 2020.

[25] L. Gwennap, "Kneron KL720 Boosts Efficiency," *Microprocessor Report,* 2020.

[26] L. Gwennap, "XMOS Xcore.ai Adds Vector Unit," *Microprocessor Report,* 2020.

[27] B. Wheeler, "RISC-V Enables IoT Edge Processor," *Microprocessor Report,* 2018.

[28] L. Gwennap, "GreenWaves GAP9 Goes Faster," *Microprocessor Report,* 2020.

[29] L. Gwennap, "Kendryte Embeds AI for Surveillance," *Microprocessor Report,* 2019.

[30] B. Wheeler, "Bitmain SoC Brings AI to the Edge," *Microprocessor Report,* 2019.

[31] M. Demler, "Syntiant NDP120 Sharpens Its Hearing," *Microprocessor Report,* 2021.

[32] M. Demler, "Syntiant NDP120 Sharpens Its Hearing," *Microprocessor Report,* 2021.

[33] L. Gwennap, "Intel Gains Myriad Customers," *Microprocessor Report,* 2018.

[34] M. Demler, "Coherent Logix Configures Edge AI," *Microprocessor Report,* 2020.

[35] M. Demler, "Hailo Illuminates Low-Power AI Chip," *Microprocessor Report,* 2019.

[36] M. Demler, "Blaize Ignites Edge-AI Performance," *Microprocessor Report,* 2020.

[37] M. Demler, "Flex Logix Moves Into Chips," *Microprocessor Report,* 2019.

[38] N. Developers, "Nvdla primer," 2021. [Online]. Available: http://nvdla.org/primer.html.

[39] T. Moreau, "A hardware-software blueprint for flexible deep learning specialization," *arXiv:1807.04188,* 2019.

[40] X. Zhang, "an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs," 2018.

[41] [Online]. Available: https://github.com/IBM/AccDNN.

[42] M. Demler, "Vsora Drives to Deliver Petaflops," *Microprocessor Report,* 2020.

[43] M. Demler, "Andes plots RISC-V vector heading," *Microprocessor Report,* 2020.

[44] L. Gwennap, "LeapMind jumps on binary networks," *Microprocessor Report,* 2020.

[45] M. Demler, "Ceva NeuPro accelerates neural nets," *Microprocessor Report,* 2018.

[46] M. Demler, "Imagination Series4 tiles tensors," *Microprocessor Report,* 2020.

[47] M. Demler, "Ceva SensPro2 Doubles AI Throughput," *Microprocessor Report,* 2021.

[48] "DPUCZDX8G for Zynq UltraScale+ MPSoCs," [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/dpu/.

[49] M. Demler, "Think Silicon Spins AI Accelerator," Microprocessor Report, 2020. [Online].

[50] A. Jani, "SiFive VIU7-256 Takes Vector Lead," *Microprocessor Report.*

[51] A. Jani, "SiFive Brings Vectors to S-Series," *Microprocessor Report,* 2021.

[52] M. Demler, "Ethos-N78 Boosts AI Efficiency," *Microprocessor Report,* 20202.

[53] M. Demler, "Cortex-M55 Supports Tiny-AI Ethos," *Microprocessor Report,* 2020.

[54] L. Gwennap, "Cortex-A55 Improves Memory," *Microprocessor Report,* 2017.

[55] L. Gwennap, "Cortex-A75 Has DynamIQ Debut," *Microprocessor Report* , 2018.

[56] L. Gwennap, "Arm Dot Products Accelerate CNNs," *Microprocessor Report* , 2018.

[57] Linley Group, "AVX2 Refreshes x86 Architecture," https://www.linleygroup.com/mpr/h/article.php?id=10794, 2011.

[58] OpenVINO, "https://docs.openvino.ai/latest/index.html," [Online].

[59] Google, "TensorFlow Lite Example," [Online]. Available: https://github.com/google-coral/tflite. [Accessed 14 10 2021].

[60] Google, "Edge TPU Compiler," [Online]. Available: https://coral.ai/docs/edgetpu/compiler/. [Accessed 14 10 2021].

[61] Google, "Edge TPU Benchmarks," [Online]. Available: https://coral.ai/docs/edgetpu/benchmarks/. [Accessed 14 10 2021].

[62] Hailo.ai, “Hailo-8 Overview,” [Online]. Available: https://hailo.ai/de/products/hailo-8/#overview. [Accessed 30 09 2022].

[63] congatec, “conga-SMX8-Plus,” [Online]. Available: https://www.congatec.com/fileadmin/user_upload/Documents/Datasheets/conga-SMX8-Plus.pdf. [Accessed 29 9 2022].

[64] NVIDIA, “Jetson AGX Orin Series Module Data Sheet,” [Online]. Available: https://developer.nvidia.com/embedded/secure/jetson/agx_orin/jetson_agx_orin_ds. [Accessed 29 07 2022].

[65] NVIDIA, “TensorRT SDK,” [Online]. Available: https://developer.nvidia.com/tensorrt.

[66] NVIDIA, “Xavier AGX Module Specs,” [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier. [Accessed 14 10 2021].

[67] NVIDIA, “Overview of Jetson Computer on Modules,” [Online]. Available: https://developer.nvidia.com/embedded/jetson-modules.

[68] NVIDIA, “Jetson Nano Module Specs,” [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/.

[69] NVIDIA, “GeForce GTX 1660 Ti Streaming Multiprocessor, In-Depth,” [Online]. Available: https://www.nvidia.com/en-us/geforce/news/geforce-gtx-1660-ti-advanced-shaders-streaming-multiprocessor/.

[70] Toms Hardware, “Nvidia GeForce GTX 1660 Review: The Turing Onslaught Continues,” [Online]. Available: https://www.tomshardware.com/reviews/nvidia-geforce-gtx-1660-turing-tu116,6027.html.

[71] NVIDIA, “NVIDIA V100,” [Online]. Available: https://www.nvidia.com/de-de/data-center/v100/. [Accessed 27 09 2022].

[72] NVIDIA, “NVIDIA A100,” [Online]. Available: https://www.nvidia.com/de-de/data-center/a100/. [Accessed 27 09 2022].

[73] KERAS, “Keras Applications,” [Online]. Available: https://keras.io/api/applications/#resnet. [Accessed 30 9 2022].

6 List of Figures

7 List of Tables

8 Abbreviations

AccDNN: Accelerator Core Compiler for Deep Neural Network

AI: Artificial Intelligence

AIoT: Artificial Intelligence of Things

ALU: arithmetic logic unit

AP: Average Precision

API: Application Programming Interface

ASIC: Application Specific Integrated Circuit

AVX: Advanced Vector Extensions

B: Bytes

CAN: Controller Area Network

CGRA: Coarse-Grained Reconfigurable Array

CPU: Central Processing Unit

DBB: Data Backbone

DL: Deep Learning

DLA: Deep Learning Accelerator

DMR: Data Memory Router

DPU: Deep Learning Processor

DSP: Digital Signal Processor

FPGA: Field Programmable Gate Array

FPS: Frames Per Second

GPU: Graphics Processing Unit

GUI: Graphical User Interface

GOPS: Giga Operations Per Second

HDMI: High Definition Media Interface

IoT: Internet of Things

IoU: Intersection over Union

IP-Core: Intellectual Property core

IPS: Inferences Per Second

IR: Intermediate Representation

ISA: Instruction Set Architecture

JIT compilation: Just-In-Time compilation

J: Joule

MAC: Multiply-Accumulate

mAP: mean Average Precision

MPU: Multiprocessing SIMD Unit

NLP: Natural Language Processing

NPU: Neural Processing Unit

OCT: Optical Coherence Tomography

ONNC: Open Neural Network Compiler

PCIe: Peripheral Component Interconnect Express

PDM: Pulse-Density-Modulation

PE: Processing Elements

PLE: Programmable Layer Engine

RAM: Random-Access Memory

RECS: Resource Efficient Cluster Server

RPC: Remote procedure call

RTL: Register-Transfer Level

SHAVE: Streaming Hybrid Architecture Vector Engine

SIMD: Single Instruction, Multiple Data

SPU: Scalar Processing Unit

SOC: System On Chip

SOTA: State-Of-The-Art

SRAM: Static random-access memory

TCM: Tightly Coupled Memory

TCP: Transmission Control Protocol

TIR: Tensor Intermediate Representation

TOPS: Trillion Operations Per Second

TPU: Tensor Processing Unit

TVM: Tensor Virtual Machine

USB: Universal Serial Bus

VPU: Vision Processing Unit

VRAM: Video Random Access Memory

VTA: Versatile Tensor Accelerator