VEDLIoT

Very Efficient Deep Learning in IoT

ICT-56-2020 - Next Generation Internet of Things

D 6.1
Report on existing hardware and software interfaces for DL and compilers

Document information	
Contract number	957197
Project website	www.vedliot.eu
Dissemination Level	PU (public)
Nature	R
Contractual Deadline	31.07.2021
Author	Karol Gugala (ANT)
Contributors	Karol Gugala (ANT), Grzegorz Latosinski (ANT), Elaheh Malekzadeh (EMBEDL), Pedro Trancoso (CHALMERS), Fareed Mohammad Qararyah (CHALMERS), Stavroula Zouzoula (CHALMERS), René Griessl (UNIBI), Kevin Mika (UNIBI), Mario Porrmann (UOS), Marco Tassemeier (UOS)
Reviewers	Eric Knauss (UGOT), Jens Hagemeyer (UNIBI), Oliver Brunnegård (VEONEER)
The VEDLIoT project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957197.	

Changelog		
V 0.1	2021-06-15	Initial draft
V 0.2	2021-06-20	First input for model chapter
V 0.3	2021-06-22	Add input for kenning chapter
V 0.4	2021-06-25	First input on accelerators
V 0.5	2021-06-26	Added IP core input
V 0.6	2021-06-29	Added comparison
V 0.7	2021-07-02	Kenning and Model chapters complete
V 0.8	2021-07-07	Accelerator sections complete
V 0.9	2021-07-09	Version for internal review
V 0.95	2021-07-28	Review comments resolved
V 0.98	2021-07-29	Format fixes
V 1.0	2021-07-30	Finalization

Table of Contents

Executive Summary

The VEDLIot (Very Efficient Deep Learning in IoT) project aims to deliver a comprehensive solution for edge AI processing. The project focuses on all aspects of the system:

- Hardware platforms
- AI accelerators used in edge AI computing
- Software frameworks
- Applications

It is crucial to understand the current state of the edge AI systems and how VEDLIoT can leverage and improve the existing ecosystem. This report discusses the current state of the art in edge AI processing and walks the reader through the detailed description of most popular deep learning frameworks, platforms and compilers. While focused on the work which was performed in Task 6.1 (Survey of Deep Learning Inference Tools, Interfaces and Compilers), it also includes preliminary results of Task 3.1 (Evaluation of existing architectures and compilers for DL), as the two tasks are related, covering different aspects of edge AI systems.

Each framework is described in terms of supported programming languages, also licensing is considered. The report also presents a general description of how the framework works and which tasks are best suited for it. Derivative frameworks are listed for each discussed framework. Deep learning platforms are divided into categories based on their integration level:

- Computing platforms
- AI dedicated SoCs and standalone accelerators
- AI accelerator IP cores

Each platform provided is described in detail including performance metrics. The deep learning compilers section of the document lists the most popular open source compilers. Each compiler is described in terms of:

- Build process
- General usage
- Features
- Supported platforms

One of the most important aspects of this document is a summary of the interoperability between the frameworks and compilers. The report discusses the possibility of converting deep learning model formats using ONNX. Also, the document presents the results of model conversion tests between different frameworks.

The conclusion of the report is that there is no outstanding platform, compiler and framework. Each of them provides slightly different features and seems to focus on different aspects. However, as long as the deep learning models are convertible, the frameworks can be switched. It seems that it would be more reasonable to provide an interface switching between various compilers depending on chosen hardware and model than implementing a new one-for-all compiler supporting as many target devices as possible. The Kenning platform presented in the report aims to provide such functionality.

1. Introduction

This report is a survey of Deep Learning Inference Tools, Interfaces and Compilers. It also covers various deep learning platforms, which incorporate either generic compute devices like CPUs or GPUs, or specialized accelerators for deep learning in different AIoT domains. As shown in Figure 1.1, this deliverable focused on the work which was performed in Task 6.1 (Survey of Deep Learning Inference Tools, Interfaces and Compilers), however, it also includes preliminary results of Task 3.1 (Evaluation of existing architectures and compilers for DL), as the two tasks are related, covering different aspects of edge AI systems. In the related deliverable D3.1 (Evaluation of existing architectures and compilers for DL), the Topics of Task 3.1 are covered in more detail, while still including parts from Task 6.1.

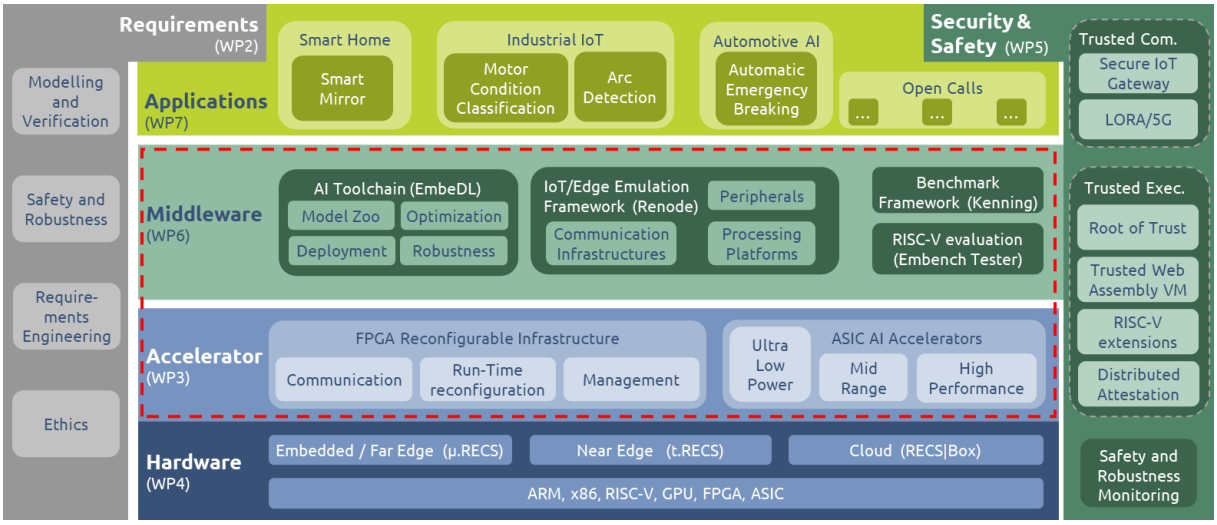

Figure 1.1: VEDLIoT abstracted overview. Parts covered by this report appear in the middle part (WP6 and WP3)

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 being developed.

Deep learning compilers optimize deep neural networks and create an efficient runtime for a given target device. There are such frameworks 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 the report is to:

- provide a list of existing Deep Learning frameworks for training and inference,
- provide a list of hardware vendor interfaces for deep learning acceleration,
- provide a list of compilers from deep learning models (from various frameworks) to another target (e.g. programming language, binary for a given accelerator),
- comprehensively map out the relationships between Deep Learning frameworks, compilers and target hardware,

- provide an overview of the large spectrum of available deep-learning hardware platforms and accelerators.

The report contains the following chapters:

- Chapter 2 (Deep Learning deployment stack) describes the typical flow of the deep learning application development from training to deployment on IoT device,
- Chapter 3 (Deep Learning Frameworks) lists the most popular and up-to-date deep learning frameworks,
- Chapter 4 (Deep Learning Platforms) lists various target devices designed for deep learning model acceleration analyzed in Task 3.1,
- Chapter 5 (Deep Learning Compilers) lists the compilers for deep learning models from one representation to another,
- Chapter 6 (Kenning) describes the Kenning framework for testing and running deep learning frameworks on IoT devices,
- Chapter 7 (ONNX Compatibility) shows the support of various frameworks for importing and exporting the ONNX models,
- In the appendix, Kenning report examples shows sample benchmarks on edge devices generated with Kenning tool.

2. Deep Learning deployment stack

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

2.1 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 preprocessing 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.

2.2 Dataset preparation

If a model is not available or is trained for a different use case, the model has to 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 the evaluation results are not biased in any way.

Datasets can be either designed from scratch or found in e.g.:

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

2.3 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 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)

2.4 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.

2.5 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 i.e. OpenGL, OpenCL, CUDA, TensorRT, ROCm libraries,
- 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.

3. Deep Learning Frameworks

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

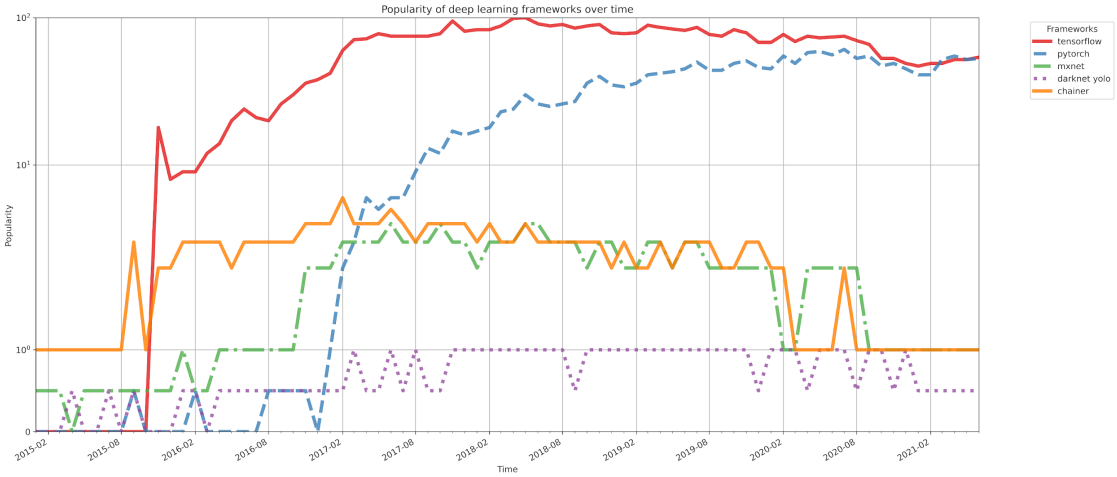

Figure 3.1: Popularity of frameworks over time (based on Google Trends), between 2016-06 till 2021-06. The popularity is normalized to range between 0 and 100, where 100 means the highest popularity in a given period of time.

3.1 TensorFlow

3.1.1 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
- Javascript
- Julia
- R
- Ruby
- Scala
- Swift

3.1.2 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, preprocessing 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 analyze 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).

3.1.3 TensorFlow children frameworks and enhancements

3.1.3.1 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 preprocessing, 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.

3.1.3.2 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.

3.1.3.3 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.

3.1.3.4 TensorFlow.JS

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

3.1.3.5 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.

3.1.3.6 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.

3.1.3.7 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.

3.1.3.8 ONNX conversions

TensorFlow does not have built-in routines for ONNX conversions. The tensorflow-onnx and onnx-tensorflow projects for converting the models to and from the ONNX.

The 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.

3.2 PyTorch

3.2.1 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

3.2.2 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.

3.2.3 PyTorch children frameworks and enhancements

3.2.3.1 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.

3.2.3.2 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.

3.2.3.3 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.

3.2.3.4 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.

3.2.4 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.

3.3 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

3.3.1 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.

3.3.2 MXNet children frameworks and enhancements

3.3.2.1 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.

3.3.2.2 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.

3.3.2.3 MXBoard

MXBoard provides an API for visualizing MXNet data in TensorBoard.

3.3.3 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.

4. Deep Learning Platforms

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

4.1 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.

4.2 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.

4.3 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.

4.4 Dedicated AI Accelerators for Ultra-Low-Power

4.4.1 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.

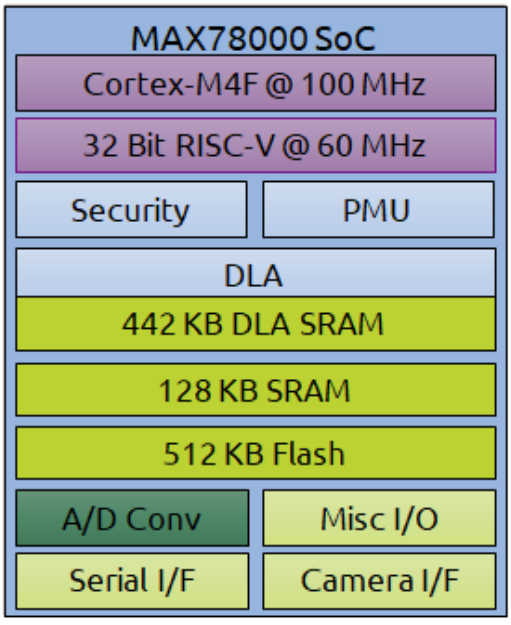

Figure 4.1: Block diagram of the MAX78000 [Jan21a]

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.

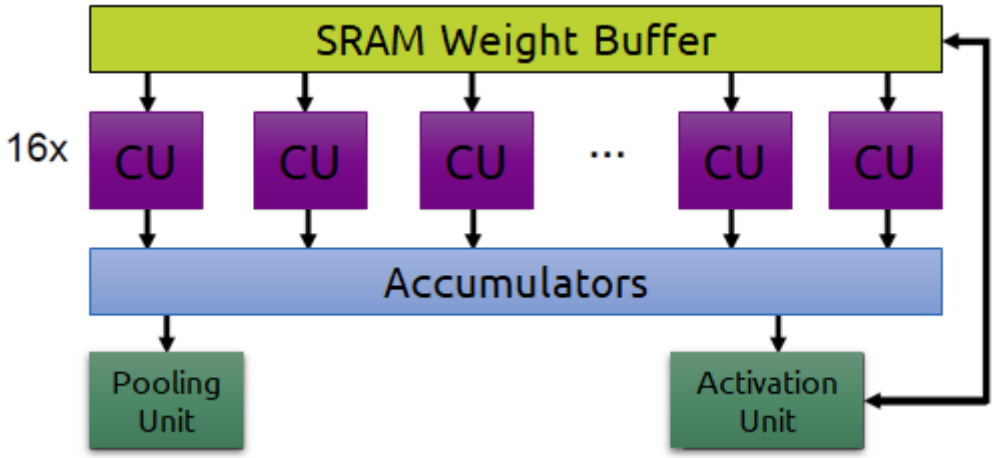

Figure 4.2: Block diagram of one AI core [Jan21a]

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

4.4.2 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 analog 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 analog and digital components to perform the MAC operations. Custom SRAM cells are used to store operands and weights. After digital to analog conversion, the multiply-accumulate step is performed in analog processing. Finally, the sums are converted back to digital. The analog compute engine can process operand with a precision of up to 16 bits. [Jan20]

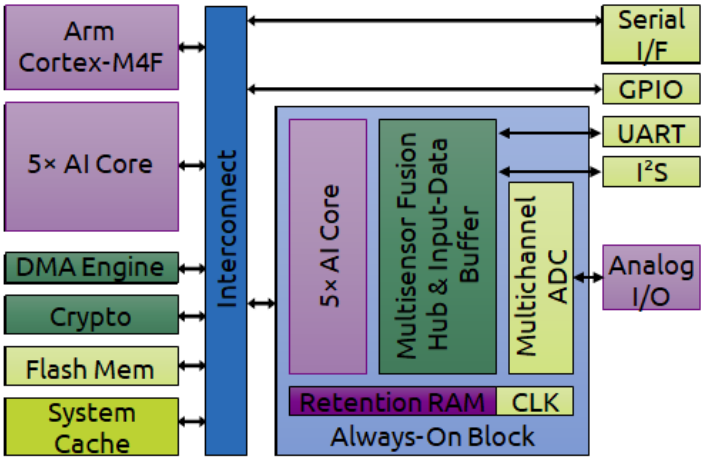

Figure 4.3: Architecture of the GPX-10 SoC [Jan20]

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.

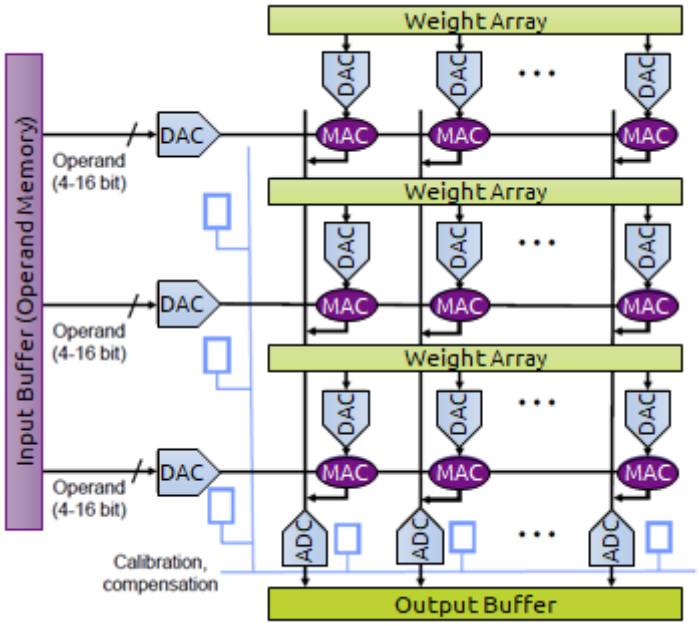

Figure 4.4: Compute engine of the GPX-10, combining digital storage and analog processing [Jan20]

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

4.4.3 Kneron KL520/KL720

Kneron KL520 [Gwe20d] 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 [Gwe20e].

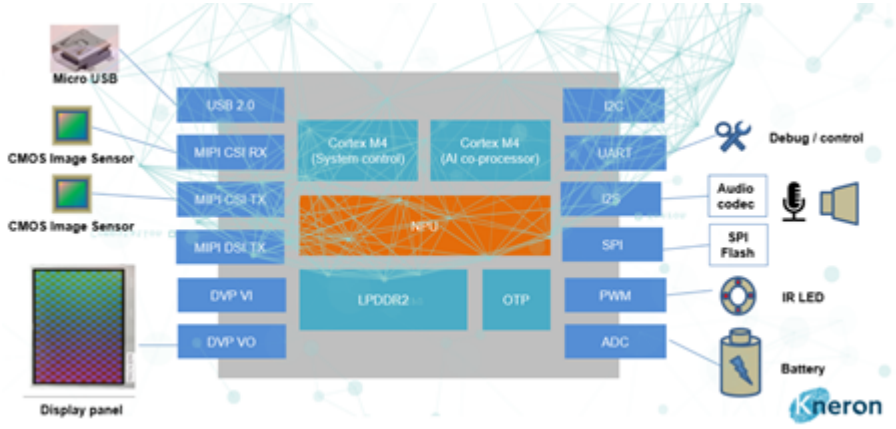

Figure 4.5: Kneron KL520 block diagram

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)

4.4.4 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. [Gwe20a]

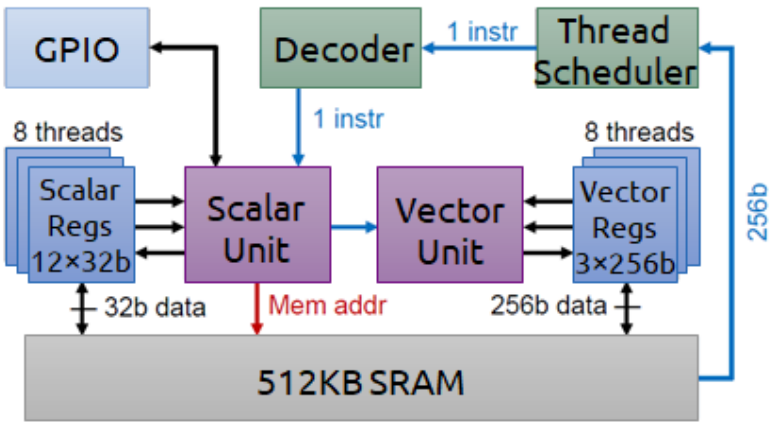

Figure 4.6: Architecture overview of the Xcore CPU [Gwe20a]

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.

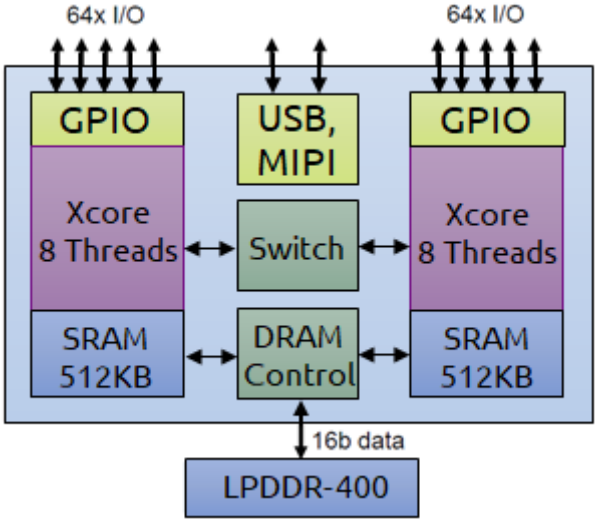

Figure 4.7: Block diagram of the Xcore.ai [Gwe20a]

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. [Gwe20a]

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

4.4.5 GreenWaves Technologies GAP8 and GAP9

GAP8 [Whe18a] and GAP9 [Gwe20c] 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.

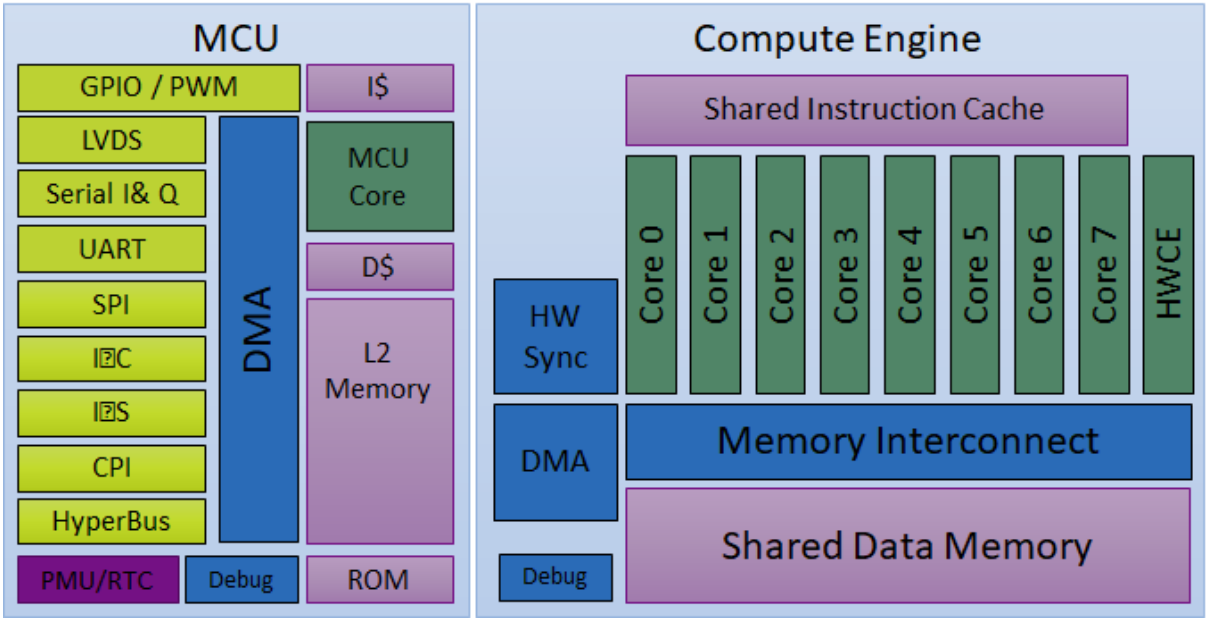

Figure 4.8: Block diagram of the GAP8 processor [Whe18a]

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

4.4.6 Kendryte K210

The K210 [Gwe19a] 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.

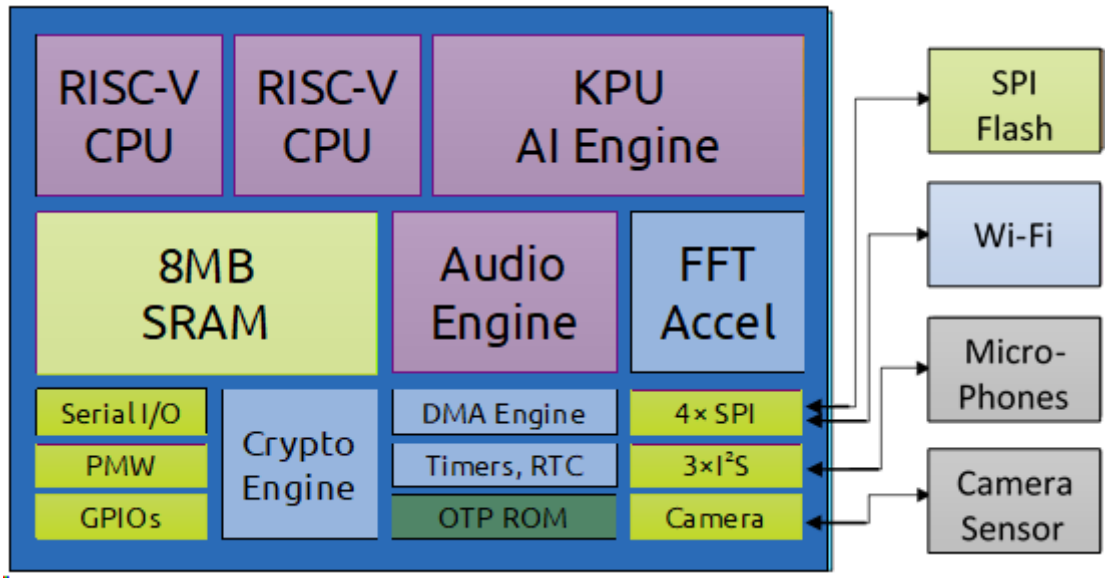

Figure 4.9: Block diagram of the Kendryte K210 SoC [Gwe19a]

The whole chip has 8MB SRAM, where two are dedicated for the KPU. The other 6BM 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 5.1MB 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 10fps.

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 analyzing 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 analog 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)
  - 800MHz
  - 6MB SRAM
  - 245MB flash

Accelerator:

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

4.4.7 NDP120

The NDP120 [Dem20d], 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 1,4MB 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

100MHz at 1,1V. In this mode more sophisticated AI and DSP algorithms can analyze 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 48kHz sampling. The chip also has an 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.

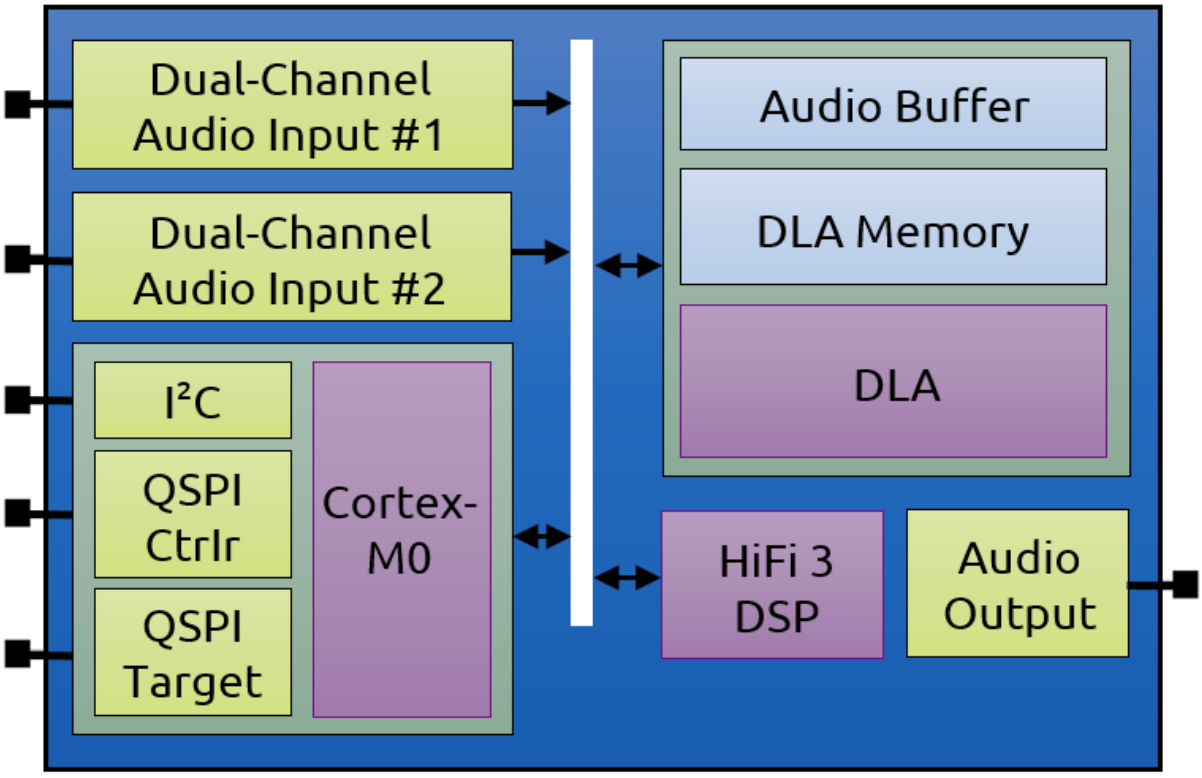

Figure 4.10: Block diagram of the NDP120 SoC [Dem20d]

Taking a look at the neural networks the chip supports convolutional neural networks, gated recurrent units and long/short-term-memory. Up to 896KB 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 20MHz
  - Boost 100MHz
  - 1.4MB SRAM

Accelerator:

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

Communication:

- 3x I^2S Input, QSPI, I^2C, GPIO

Software:

- Tensorflow, Python

Performance:

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

Availability:

Volume production starts in Q3 2021

4.5 Dedicated AI Accelerators

4.5.1 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.

4.5.2 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 a efficiency of 2 TOPS/W, 10 times the efficiency of the TPUv3. Unlink the big TPUs the Edge TPU is programmed in TensorFlow Lite.

4.5.3 Intel Myriad X

Intel Movidius Myriad X [Gwe18a] 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 700MHz and act as managing cores, as well as 16 programmable vector processors with 256KB 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 2.5MB 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 10GbE are also included.

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

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

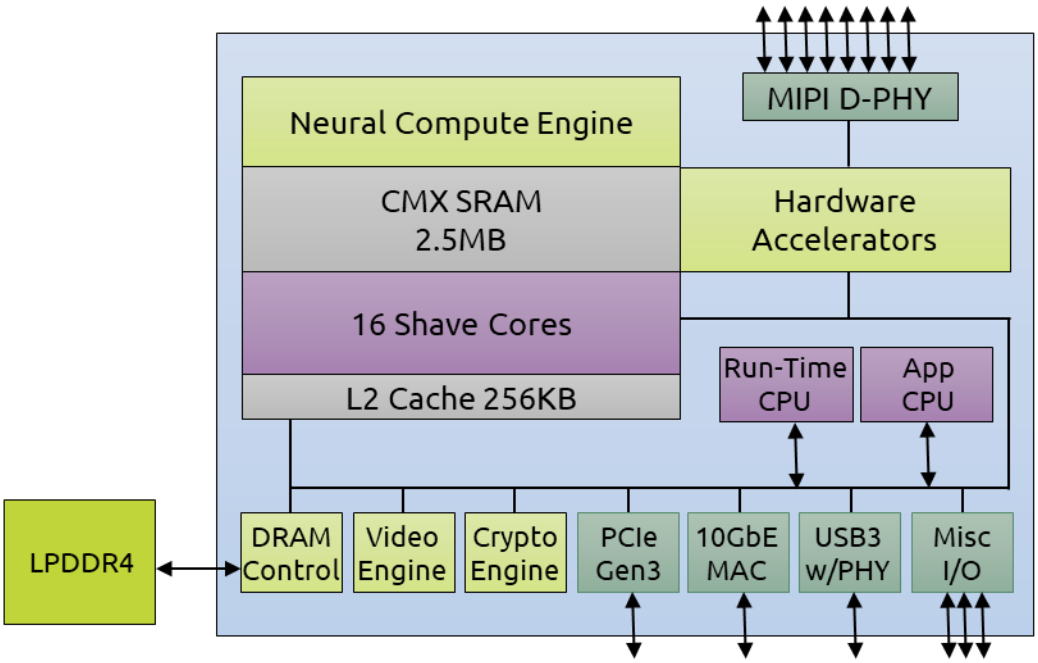

Figure 4.11: Block diagram of the Myriad X [Gwe18a]

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 256KB 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 @ 2.5W
- 0.5 TOPS/W

4.5.4 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 [Dem20e], 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 1x. However, in contrast to an FPGA, this processor does not require RTL development, place and route, or timing optimizations, because the compiler handles everything.

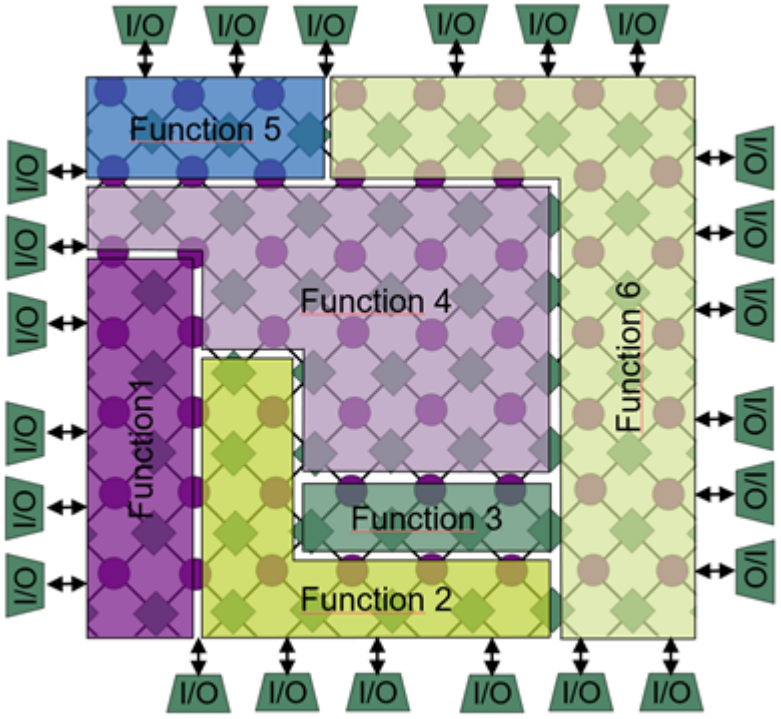

Figure 4.12: Function blocks inside the PE and DMR Array [Dem20e]

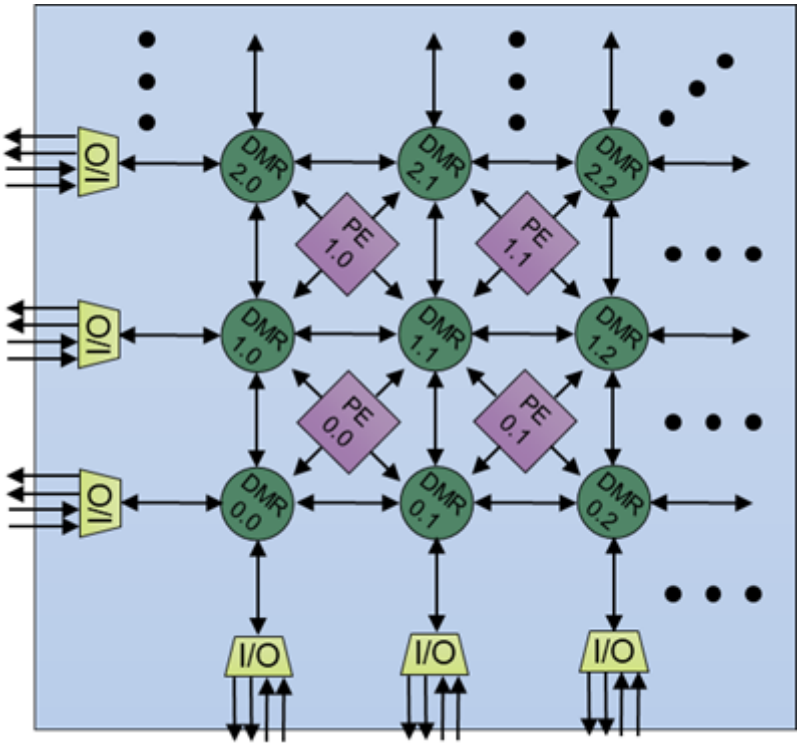

Figure 4.13: Detailed view of the Accelerator structure [Dem20e]

Figure 2x 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 14nm 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 @ 8W (Hier passt was mit den Performance daten nicht, im MPrep ist 6600 TOPS angegeben, aber 6600/8 sind nicht 1,6)
- 1.6 TOPS/W (INT8)

Availability:

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

4.5.5 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 [Dem19a], 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.

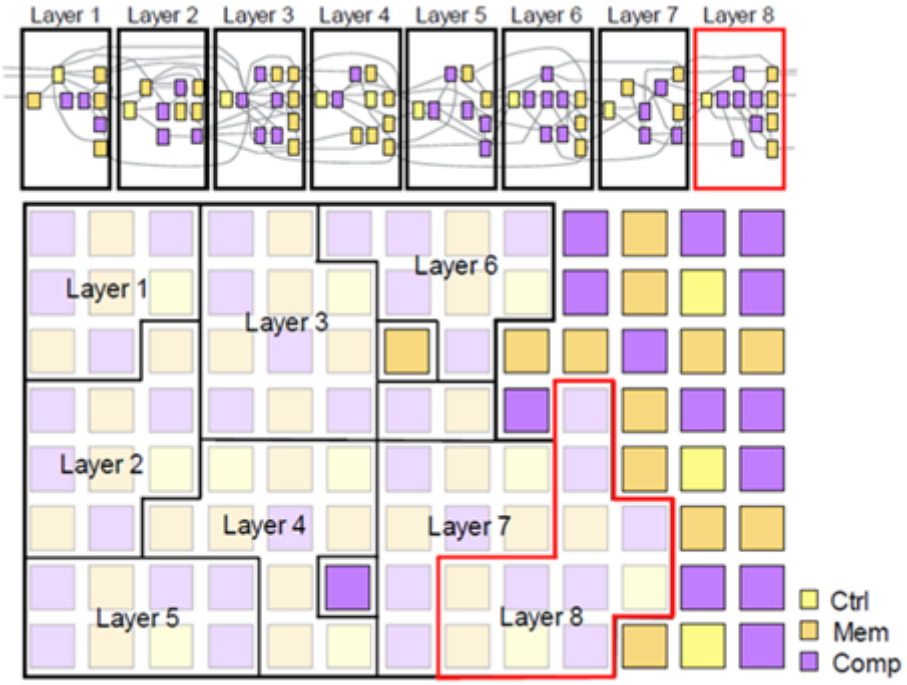

Figure 4.14: Visualization of the DNN layers on the chip [Dem19a]

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 3 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.

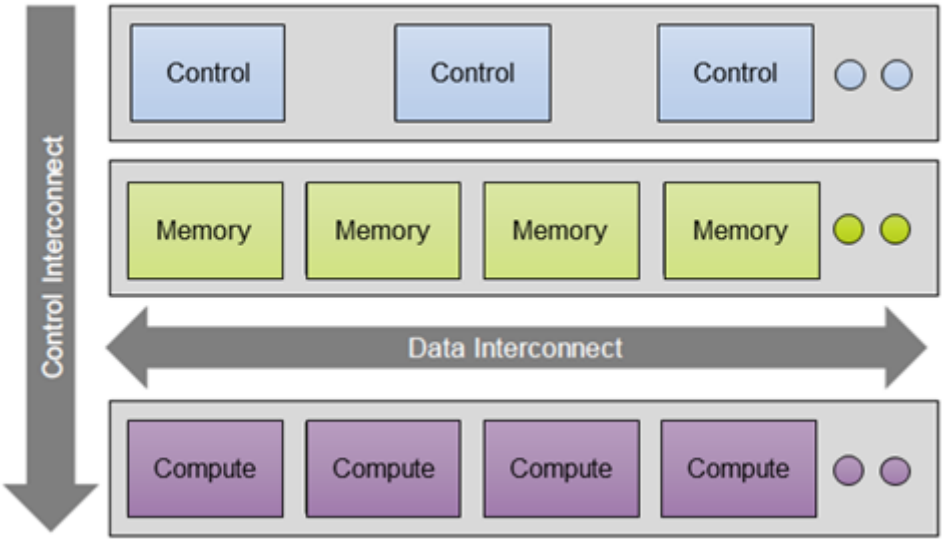

Figure 4.15: Hailo-8 Bus Structure [Dem19a]

Figure 4 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 @ 9W
- 2.8 TOPS/W

Availability:

- M.2 PCIe, mPC

4.5.6 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 [Whe19a] manufactured in 7nm. Figure 5 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 color-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.

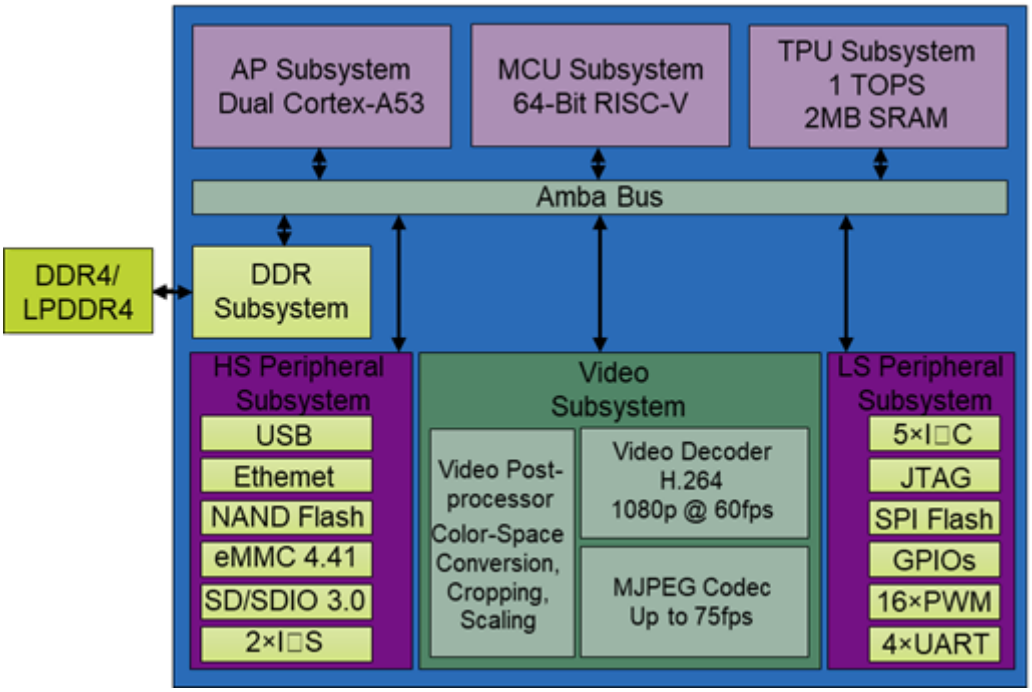

Figure 4.16: Block diagram of the Sophon BM1880 SoC [Whe19a]

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

4.5.7 Blaize El Cano

El Cano [Dem20f] 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.

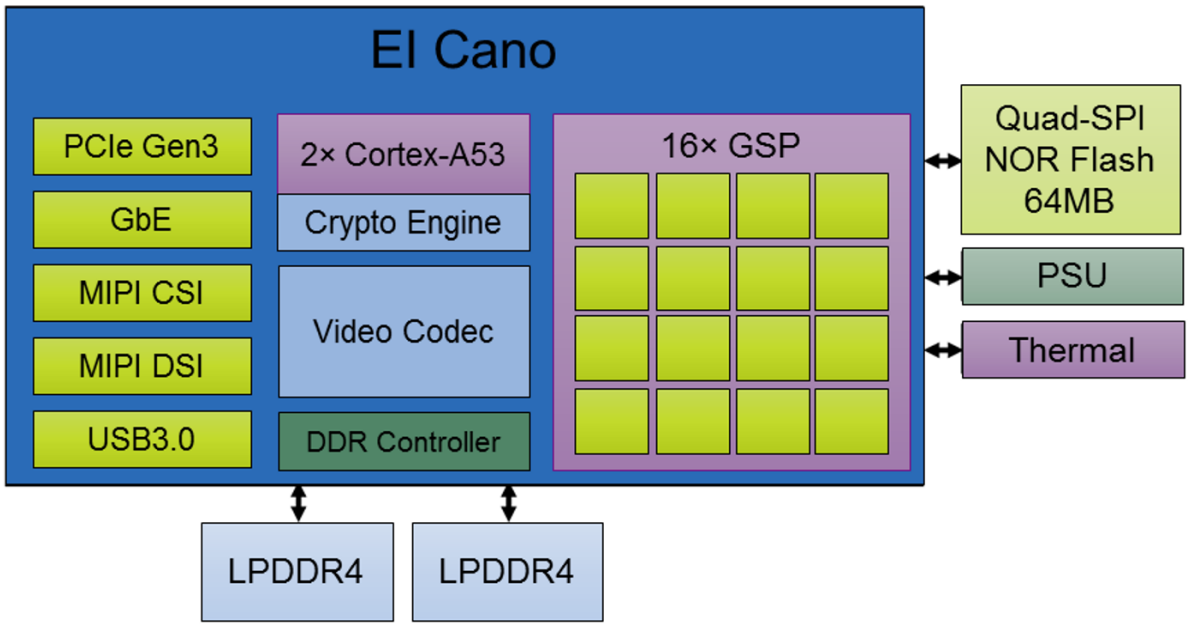

Figure 4.17: Sequential (TensorFlow-based) and streaming (Blaize) neural-network execution [Dem20f].

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.

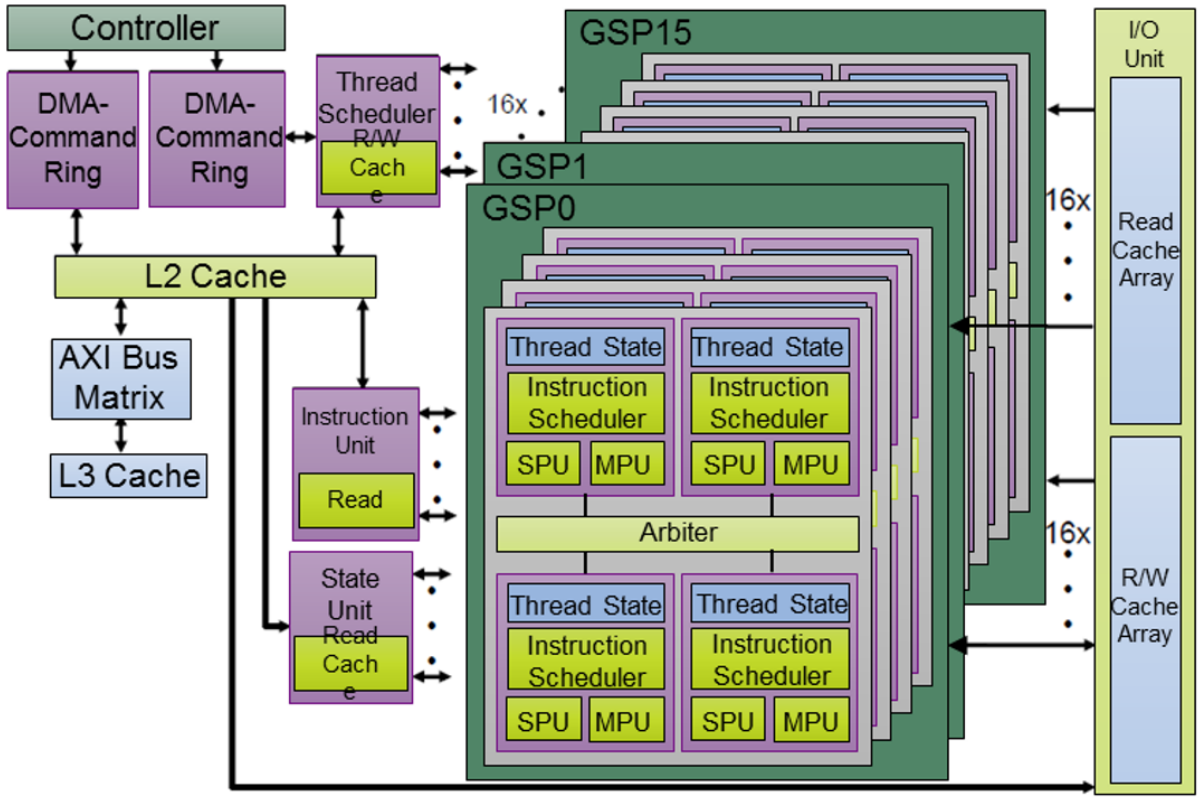

Figure 4.18: Blaize Pathfinder system-on-module [Dem20f]

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)

- 4 processor units per GSP
- Multiple graphs at once

Interfaces:

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

Performance:

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

4.5.8 Infer X1

The InferX X1 [Dem19b] 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.

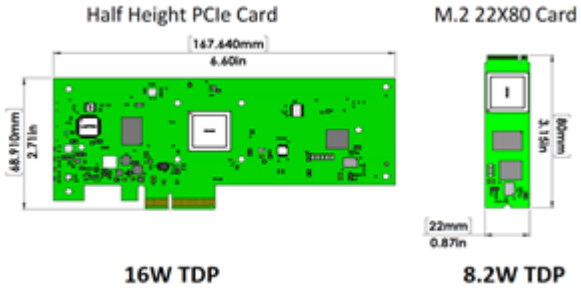

Figure 4.19 : Size and power comparison of Half Height PCIe Card and M.2 22x80 Card

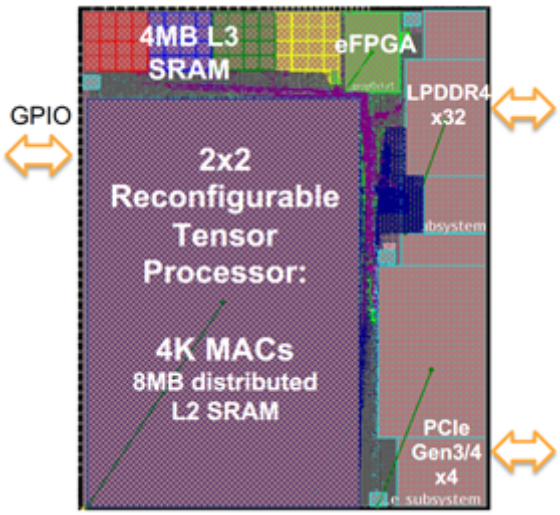

Figure 4.20: InferX X1 chip

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 256KB SRAMs for local weight storage. On the chip there is also an 8MB distributed L2 SRAM. The multipliers of the MACs are 16x8 bit and the accumulator 32 bits.

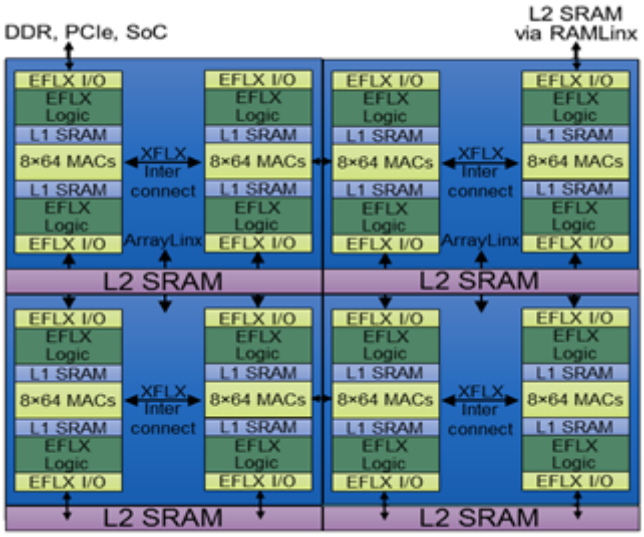

Figure 4.21: InferX X1 coprocessor chip [Dem19b]

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

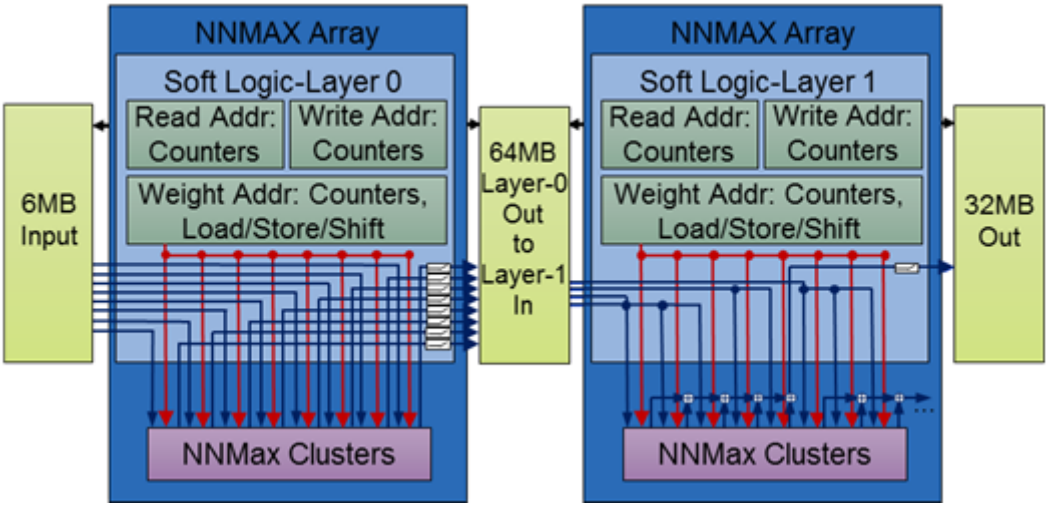

Figure 4.22: Example of InferX pipeline operations [Dem19b]

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)
- 8MB distributed SRAM
- 4MB L3 SRAM
- LPDDR4 32-bit

Interfaces:

- PCIe Gen3 x4, 32x GPIO

Software:

- TensorFlow Lite, ONNX

Performance:

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

4.6 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 ve used free of charge. Target technologies vary from FPGAs to ASICs; hence, also the achievable performance differs significantly.

4.6.1 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 [Dev21a]:

- 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 [Dev21a]:

- 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.

4.6.1.1 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

4.6.2 Versatile Tensor Accelerator (VTA)

VTA is a parameterizable accelerator that accelerates the bulk of the deep learning compute graphs [MCV+19]. 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.

4.6.3 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. [ZWZ+18]

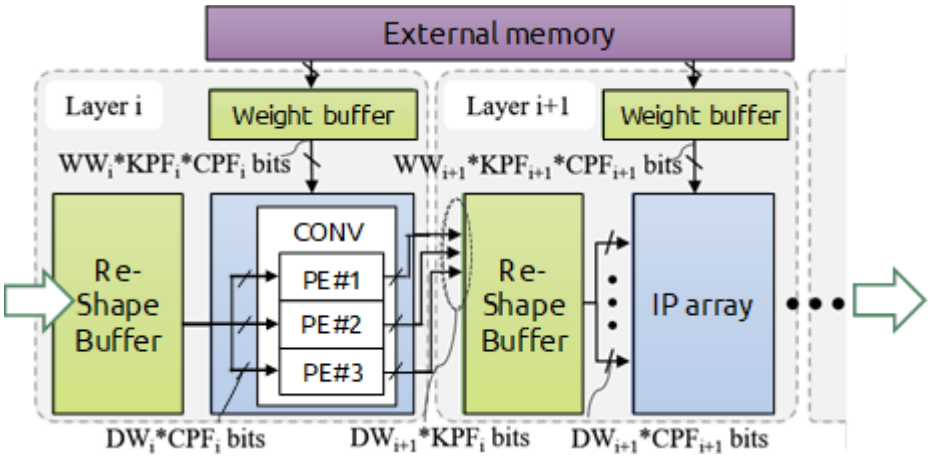

Figure 4.23: Example of an accelerator architecture generated by AccDNN [ZWZ+18]

In [ZWZ+18] 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 the figure above. [ZWZ+18]

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. [AccDNN]

Performance
In [ZWZ+18] 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

4.6.4 VSORA AD1028

Combining a DSP and a deep learning accelerator, the AD1028 [Dem20a] from the french IP provider VSORA mainly targets autonomous driving. The combination of DSP and AI architecture eliminates the need for additional accelerators. The figure below gives a high-level overview of the architecture. Details about the internal structure are not disclosed.

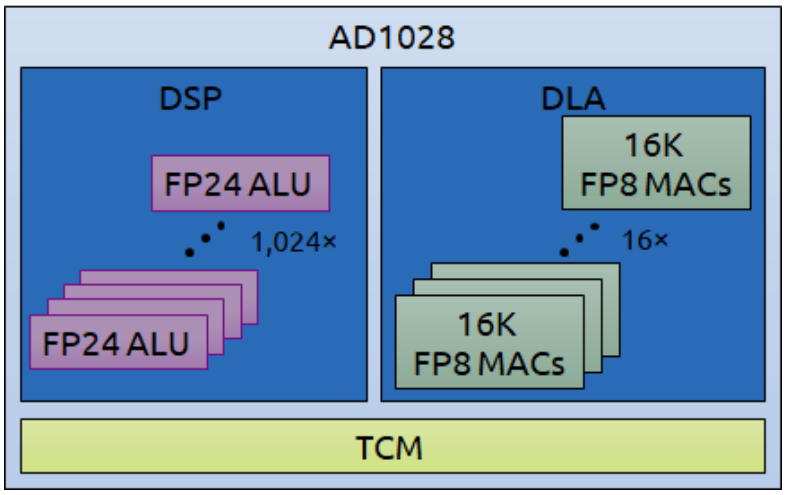

Figure 4.24: High-level architecture of the VSORA AD1028 [Dem20a]

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. [Dem20a]

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

4.6.5 Andes NX27V

The Andes NX27V [Dem20b] 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.

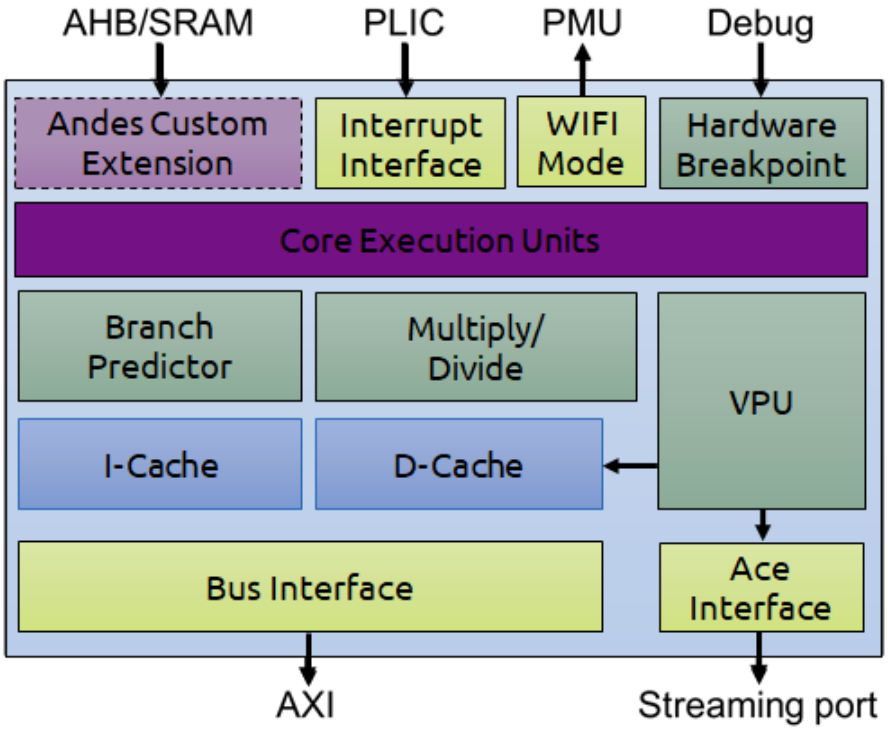

Figure 4.25: Architecture of the Andes NX27V CPU [Dem20b]

The vector processing unit can be configured to different bit widths to fit the given requirements. Possible bit widths are 128, 256 and 512 bit. 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. [Dem20b]

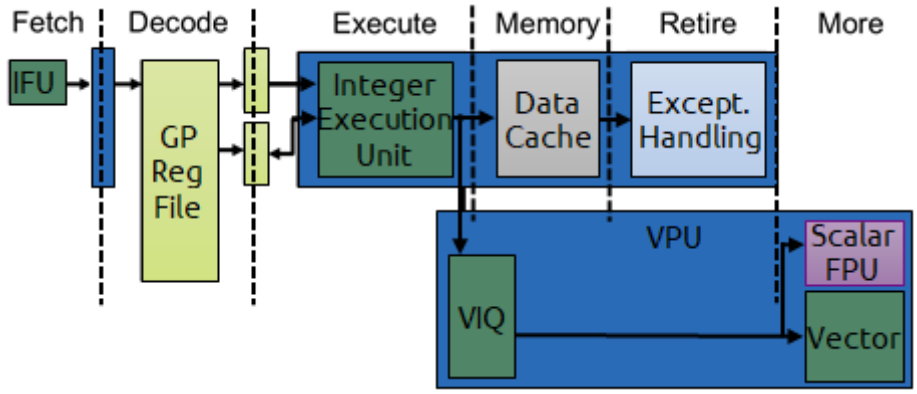

Figure 4.26: Integration of the vector pipeline into the NX27V scalar unit via the vector instruction queue (VIQ) [Dem20b]

Architectural Features

- Up to 512 bit 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 8-64KB I-Cache and D-Cache
- AXI-Interface
- Support for MXNet, Pytorch and Tensorflow

Performance

- Estimated with 7nm TSCM: 1.5GHz and 48 GFLOPS (FP32) [Dem20b]

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.

4.6.6 LeapMind Efficiera

The LeapMind Efficiera [Gwe20b] 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. [Gwe20b]

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

LeapMind Efficiera is available as a proprietary IP core.

4.6.7 Ceva NeuPro

The Ceva NeuPro [Dem18a] 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. [Dem20c]

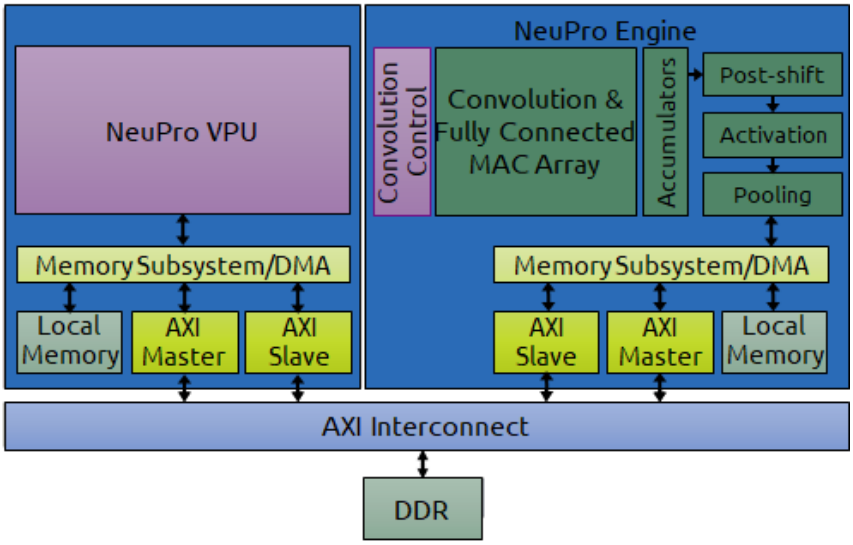

Figure 4.27: Architecture overview of the Ceva NeuPro deep-learning accelerator [Dem18a]

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

The Ceva NeuPro ist available as a proprietary IP core.

4.6.8 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-centers 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.

4.6.9 Ceva SensPro2

SensPro2 [Dem21a] 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.

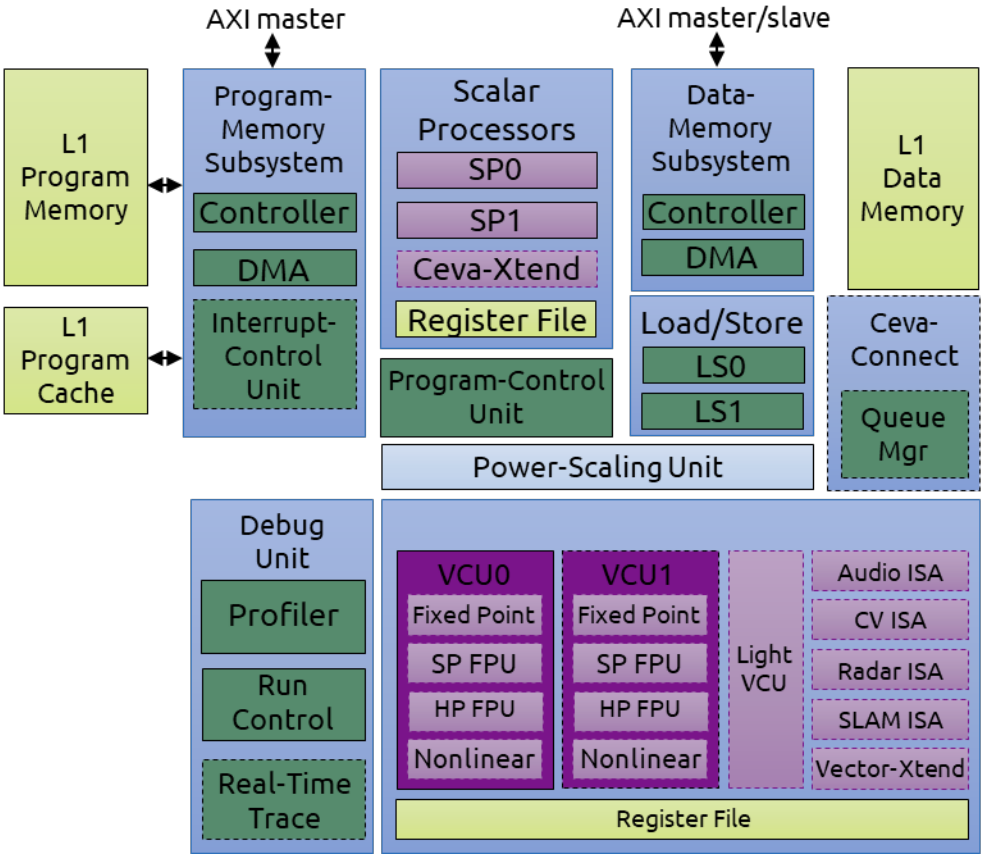

Figure 4.28: SensPro2 block diagram [Dem21a]

4.6.10 Xilinx DPU

Xilinx Deep Learning Processor (DPU) [Xil21] 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.

4.6.11 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].

4.6.12 Think Silicon Neox V

Neox V [Dem20g] 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 [Dem20e].

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

4.6.13 Imagination Technologies Series4

The Imagination Technologies Series4 deep learning accelerators [Dem20c] 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.

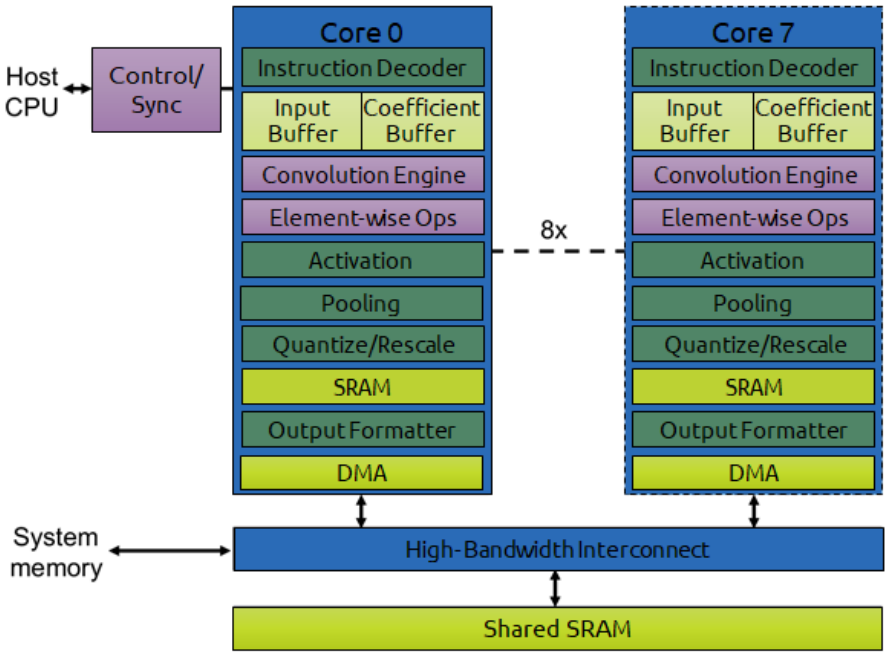

Figure 4.29: Architecture of the Imagination Technologies Series4 DLA [Dem20c]

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. [Dem20c]

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
- 256KB to 64MB SRAM
- 64KB Input Buffer / 128KB coefficient buffer per core
- Interconnect with 256 bits per cycle per core => 384GB/s at 1.5GHz
- Compiler
  - Supports layer merging and tiling
  - Distributes computation between the cores to reduce DRAM transactions
  - Enables running of different workloads in parallel

Performance

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

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

4.6.14 SiFive VIU7-256

The VIU7- 256 [Jan20b] 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.

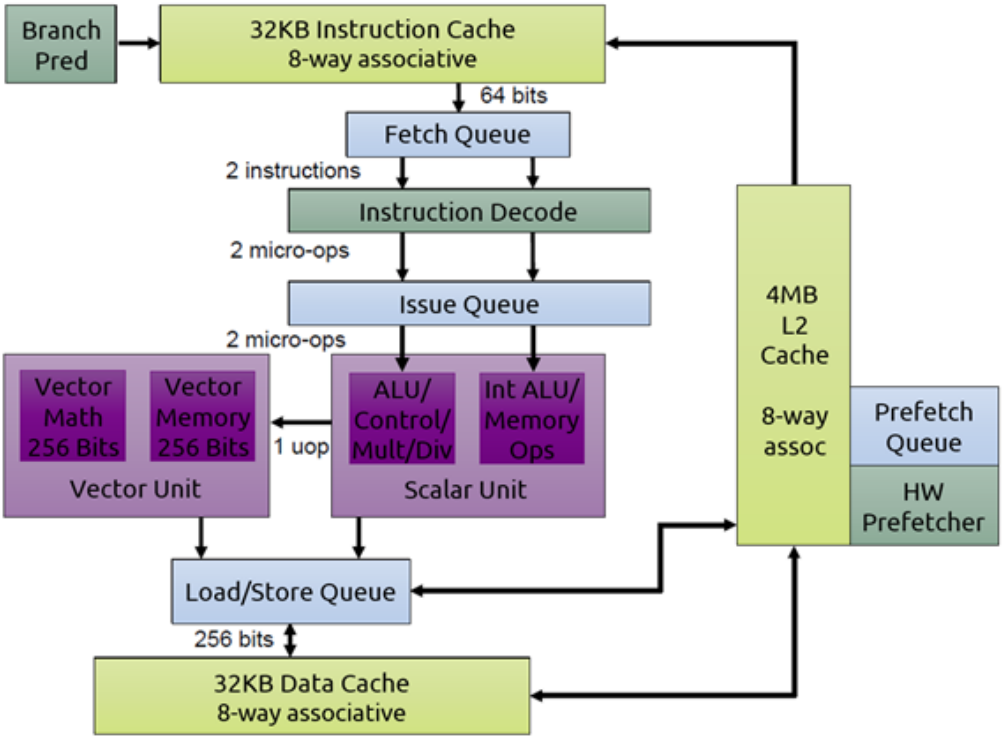

Figure 4.30: SiFive SiFive VIU7-256 microarchitecture [Jan20b]

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.

4.6.15 SiFive VIS7

The VIS7 [Jan21c] is a larger variant of the VIU7-256, supporting real 512-bit operations. 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.

4.6.16 ARM Ethos-N78

The ARM Ethos-N78 [Dem20h] 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.

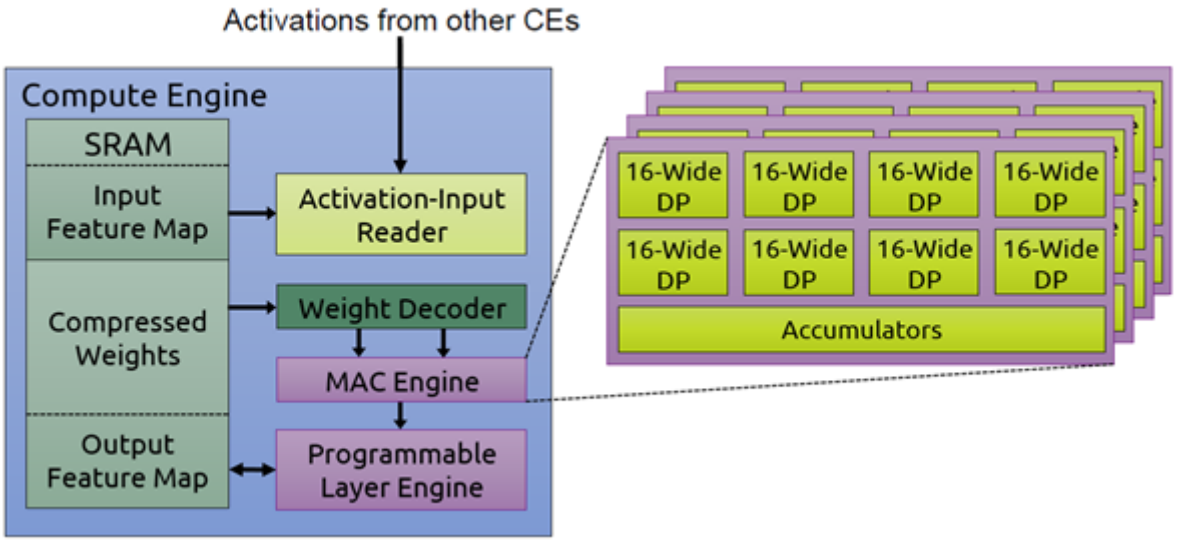

Figure 4.31: Ethos-N78 compute engine [Dem20h]

4.6.17 ARM Ethos-U55

The Ethos-U55 [Dem20i] 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.

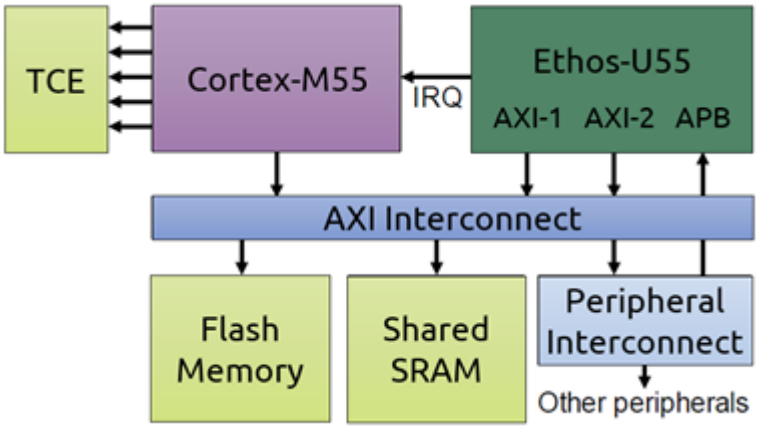

Figure 4.32: Cortex M55 pipeline with Helium [Dem20i]

4.6.18 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 [Gwe17a]. It has 8 pipeline stages and supports only in-order execution, which results in less chip area compared to out-of-order architectures.

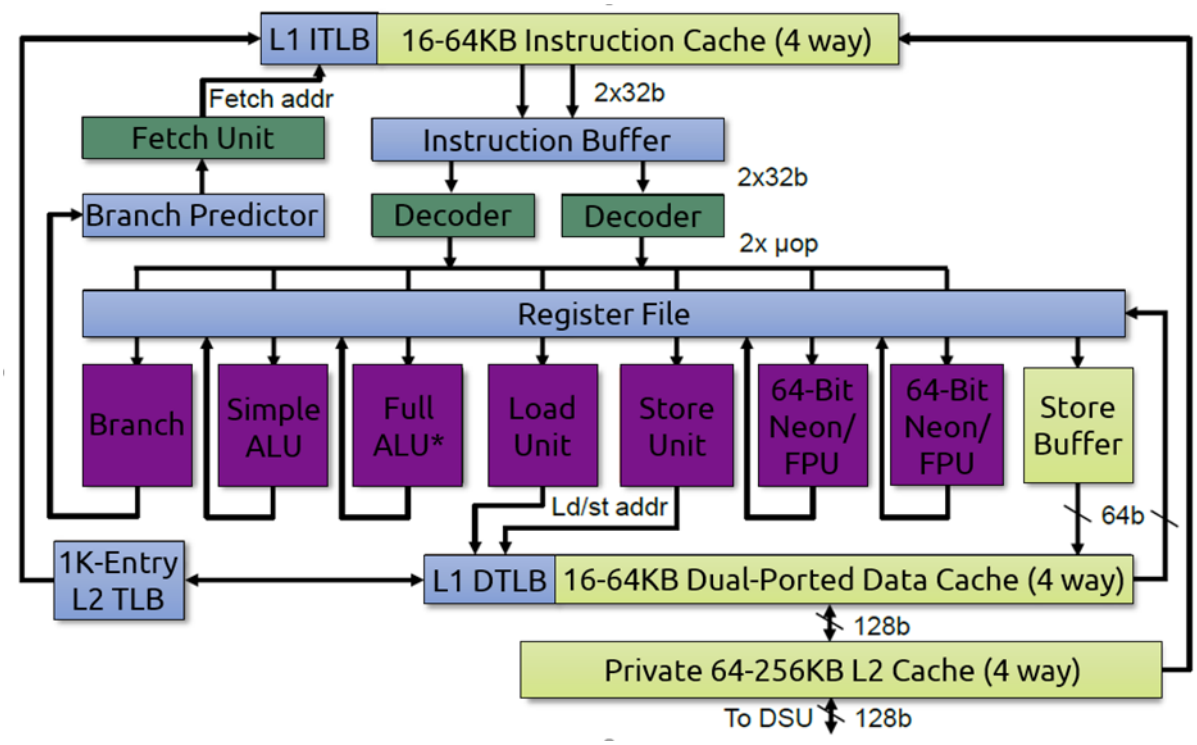

Figure 4.33: Architecture of the ARM Cortex-A55 [Gwe17a]

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 [Gwe17b]. 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 [Gwe18b].

4.7 Comparison of AI Accelerators

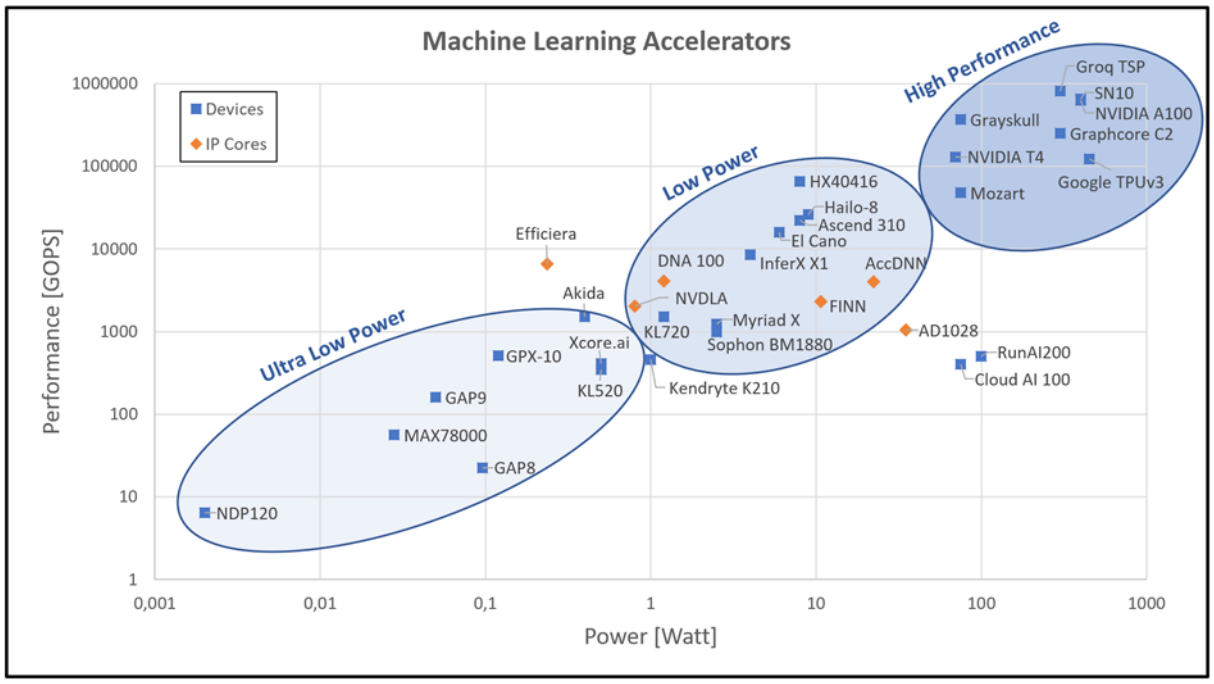

Figure 4.34: Overview of Machine Learning Accelerators

Since VEDLIoT focuses on edge computing, mainly ultra-low power and low power accelerators for machine learning have been presented in this chapter. Figure 4.33 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 will be provided in the D3.1. On average over all architectures, a power efficiency in the order of 1TOPS/Watt is achieved (represented by the diagonal of the chart).

5. Deep Learning Compilers

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

5.1 Apache TVM

5.1.1 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

5.1.2 Description

Tensor Virtual Machine (TVM) is a hierarchical multi-tier compiler stack and runtime system for deep learning models [RLW+18]. 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.

5.1.2.1 Relay Intermediate Representation language

As described in [RLW+18], 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 [RLW+18].

Relay is a purely functional, statistically-typed programming language for differentiable computations expressing machine learning models [RLW+18]. 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.

5.1.2.2 Compilation flow

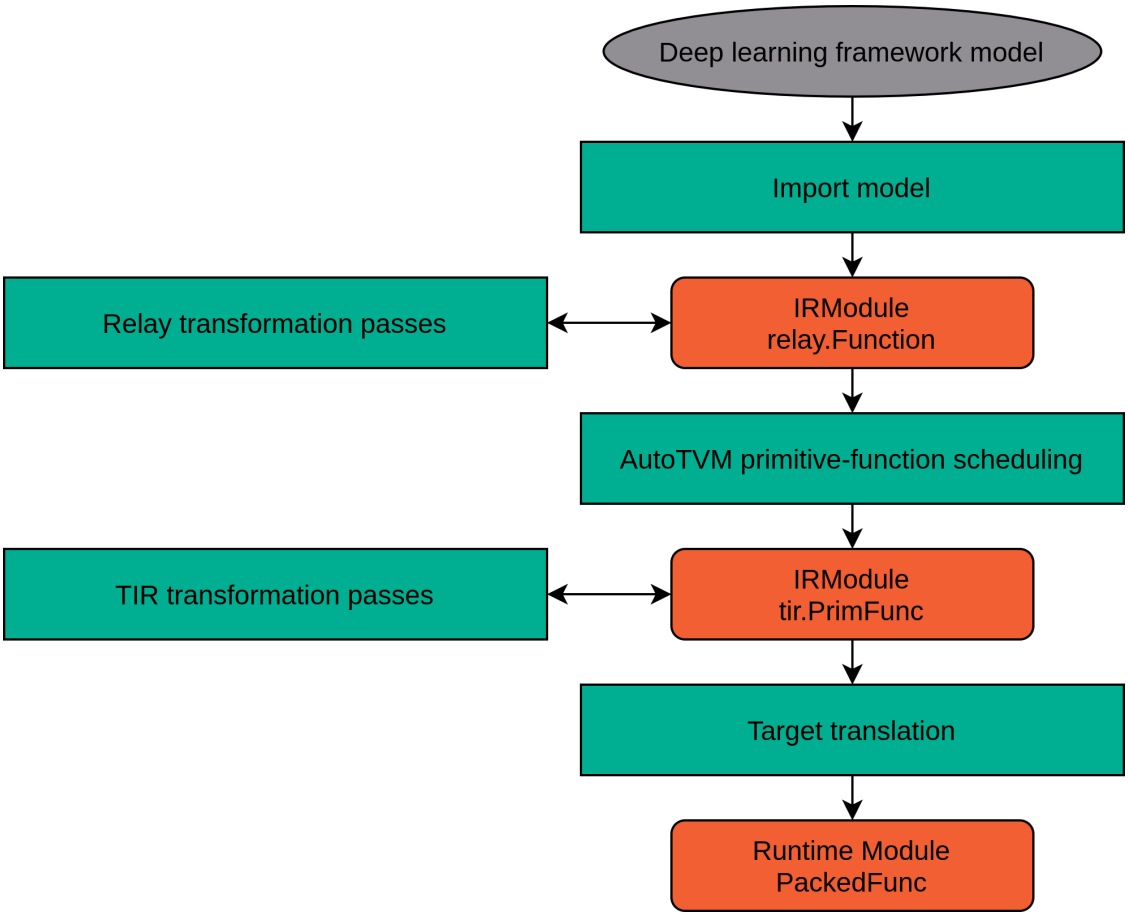

Figure 5.1: The TVM compilation flow.

Fig. 5.1 shows the compilation flow of the TVM framework, based on [Apache].

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 [Apache].

- *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 [Apache].

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.

5.1.2.3 Running the model

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

5.1.3 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.

5.1.4 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

5.2 TensorFlow Lite

5.2.1 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

5.2.2 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.

It 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.

5.2.2.1 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 [GoogleDa]. 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.

5.2.2.2 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 [GoogleDb]:

- *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.

5.2.2.3 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 [GoogleDb]. 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 [GoogleDb], 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.

5.2.3 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 [GoogleDb],
- Small model size, thanks to TFLite FlatBuffers,
- Weights quantization,
- Connections pruning,
- Clustering,
- Customizable delegates.

5.2.4 Supported targets

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

5.3 OpenVINO

5.3.1 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

5.3.2 Description

OpenVINO provides a necessary toolset and optimized runtime dedicated to the efficient execution of neural networks on Intel CPUs, FPGAs and specialized accelerators [DGF+19].

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.

5.3.3 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

5.4 ONNX Runtime

5.4.1 General information

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

5.4.2 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 fallback to those default providers.

5.4.3 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.

5.4.4 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

5.5 ONNC

5.5.1 General information

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

5.5.2 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 [LTT+19]. 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 [LTT+19]. 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 [LTT+19].

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 [LHC19]. 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 [LHC19].

5.5.3 Supported targets

- NVDLA
- Sophon BM168X

5.6 Glow

5.6.1 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

5.6.2 Description

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

5.6.2.1 Compilation flow

As described in [RFA+19], 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 [RFA+19], 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 [RFA+19]. 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.

5.6.2.2 Model optimizations

The model optimizations in Glow can be divided into [RFA+19]:

- 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.

5.6.2.3 Running the model

Glow provides a runtime that is capable of partitioning models, queuing requests and executing models across multiple devices [RFA+19]. 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.

5.6.3 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.

5.6.4 Supported targets

- OpenCL
- ARM
- Intel CPUs

6. Kenning

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

6.1 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.

6.2 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.

6.3 Model preparation

6.3.1 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.

6.3.2 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.

6.3.3 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.

6.4 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.

6.4.1 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.

6.4.2 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.

6.4.3 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.

6.5 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.

6.6 Running the benchmarks

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

6.6.1 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.TensorFlowPetD
atasetMobileNetV2 \
    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.TensorFlowPetD
atasetMobileNetV2 \
```

```
    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.

6.6.2 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.TensorFlowPetD
atasetMobileNetV2 \
    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.

6.6.3 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.

6.6.4 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.TensorFlowPetD
atasetMobileNetV2 \
    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_mobilenet
v2.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.

6.6.4.1 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).

6.7 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.

7. ONNX Compatibility

Open Neural Network Exchange is an open standard for representing machine learning models. It is actively developed, 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.

7.1 General information

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

7.2 ONNX conversion support grid

1. Table 1 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.

8. Conclusion

This document summarized the most popular deep learning accelerators, deep learning frameworks and compilers. While focused on the work which was performed in Task 6.1 (Survey of Deep Learning Inference Tools, Interfaces and Compilers), it also includes preliminary results of Task 3.1 (Evaluation of existing architectures and compilers for DL), as the two tasks are related, covering different aspects of AIoT systems. In the related deliverable D3.1 (Evaluation of existing architectures and compilers for DL), the topics of Task 3.1 are covered in more detail, while still including parts from Task 6.1. In this deliverable, the part from Task 3.1 are mainly included in Chapter 4 (Deep Learning Platforms).

The analysis of the deep learning accelerators showed that there are lots of possible acceleration techniques both for vector ALU operations, and GEMM computations. They differ in acceleration schemes, memory layout and management. The accelerators differ in terms of supported value types – some accept 32-bit and 16-bit floats, while others support only 8-bit integers, or even 1-bit weights. There are accelerators that require changes of representation of matrices, i.e. for Winograd matrix multiplication. Some accelerators also support sparse matrices, while others cannot gain any acceleration from this fact.

Deep learning frameworks mostly differ in the execution mode (eager vs graph execution), the list of supported pre-processing routines, image formats (i.e. NCHW vs NHWC), and ready-to-use models. When it comes to the list of supported operations present in deep learning models, optimizers and typical learning helpers, all of the frameworks support the similar list, which enables the developer to implement most of the models on all frameworks.

There are differences in deep learning frameworks when it comes to novel deep learning approaches, such as new learning schedulers, learning rate finders, regularization and optimization techniques. Such tools are adopted with different pace among deep learning frameworks, and some deep learning frameworks encourage users to implement such learning alterations through training callbacks. Such development aspects, as optimizations, are usually handled by third-party libraries and deep learning compilers.

Thanks to the ONNX, it is possible to migrate models between frameworks, which allows the developer to switch to another framework with ease (to some extent, i.e. it is not possible for now to import the ONNX model to PyTorch).

With the ONNX conversions and deep learning compilers supporting models from various deep learning frameworks it seems that the choice of the framework for model training and development is mostly the developer's choice based on preference.

The deep learning compilers are far more diverse than deep learning frameworks and they approach various aspects of model compilation differently. Deep learning compilers differ mostly in:

- The list of supported target platforms,
- Intermediate representation of the model and optimizations coming from this,
- Operators' granularity used for optimization and runtime – some compilers operate on simple operations, while other handle and optimize whole blocks consisting of

multiple layers,

- List of supported operators,
- List of supported input formats, which translates to list of supported deep learning frameworks,
- List of model optimization techniques.

Each of the deep learning compilers approaches the model optimization and runtime differently. This results in different memory management approaches, different operator execution schemes and overall performance on various targets.

From the initial research it seems that there is no definite leader or one-for-all solution among deep learning compilers – they specialize for a specific subset of targets on all levels of model translations, representations and runtime execution. It seems that it would be more reasonable to provide an interface switching between various compilers depending on chosen hardware and model than implementing a new one-for-all compiler supporting as many target devices as possible.

9. References

[AccDNN] https://github.com/IBM/AccDNN

[DGF+19] Alexander Demidovskij, Yury Gorbachev, Mikhail Fedorov, Iliya Slavutin, Artyom Tugarev, Marat Fatekhov, and Yaroslav Tarkan. Openvino deep learning workbench: comprehensive analysis and tuning of neural networks inference. In 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW), 783–787. 2019. doi:10.1109/ICCVW.2019.00104.

[Dem18a] Mike Demler. Ceva NeuPro accelerates neural nets. Microprocessor Report 1/2018, The Linley Group.

[Dem19a] Mike Demler. Hailo Illuminates Low-Power AI Chip. Microprocessor Report 6/2019, The Linley Group.

[Dem19b] Mike Demler. Flex Logix Moves Into Chips. Microprocessor Report 4/2019, The Linley Group.

[Dem20a] Mike Demler. Vsora Drives to Deliver Petaflops. Microprocessor Report 12/2020, The Linley Group.

[Dem20b] Mike Demler. Andes plots RISC-V vector heading. Microprocessor Report 5/2020, The Linley Group.

[Dem20c] Mike Demler. Imagination Series4 tiles tensors. Microprocessor Report 11/2020, The Linley Group.

[Dem20d] Mike Demler. Syntiant NDP120 Sharpens Its Hearing. Microprocessor Report 4/2021, The Linley Group.

[Dem20e] Mike Demler. Coherent Logix Configures Edge AI. Microprocessor Report 10/2020, The Linley Group.

[Dem20f] Mike Demler. Blaize Ignites Edge-AI Performance. Microprocessor Report 9/2020, The Linley Group.

[Dem20g] Mike Demler. Think Silicon Spins AI Accelerator. Microprocessor Report 10/2020, The Linley Group

[Dem20h] Mike Demler. Ethos-N78 Boosts AI Efficiency. Microprocessor Report 6/2020, The Linley Group

[Dem20i] Mike Demler. Cortex-M55 Supports Tiny-AI Ethos. Microprocessor Report 3/2020, The Linley Group

[Dem21a] Mike Demler. Ceva SensPro2 Doubles AI Throughput. Microprocessor Report 8/2021, The Linley Group.

[Dev21a] NVIDIA Developers. Nvdla primer. http://nvdla.org/primer.html, 2021.

[Dev21b] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2021.

[Gwe18a] Linley Gwennap. Intel Gains Myriad Customers. Microprocessor Report 12/2018, The Linley Group.

[Gwe19a] Linley Gwennap. Kendryte Embeds AI for Surveillance. Microprocessor Report 3/2019, The Linley Group.

[Gwe20a] Linley Gwennap. XMOS Xcore.ai Adds Vector Unit. Microprocessor Report 5/2020, The Linley Group.

[Gwe20b] Linley Gwennap. LeapMind jumps on binary networks. Microprocessor Report 5/2020, The Linley Group.

[Gwe20c] Linley Gwennap. GreenWaves GAP9 Goes Faster. Microprocessor Report 1/2020, The Linley Group.

[Gwe20d] Linley Gwennap. Kneron Delivers Efficient AI. Microprocessor Report 2/2020, The Linley Group.

[Gwe20e] Linley Gwennap. Kneron KL720 Boosts Efficiency. Microprocessor Report 10/2020, The Linley Group.

[HMD16] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. 2016. arXiv:1510.00149.

[Jan20a] Aakash Jani. Ambient's Analog Cuts AI Power. Microprocessor Report 11/2020, The Linley Group.

[Jan20b] Aakash Jani. SiFive VIU7-256 Takes Vector Lead. Microprocessor Report 11/2020, The Linley Group.

[Jan21a] Aakash Jani. Maxim Showcases Efficient Custom AI. Microprocessor Report [2/2021, The Linley Group.

[Jan21b] Aakash Jani. SiFive Brings Vectors to S-Series. Microprocessor Report 1/2021, The Linley Group.

[Whe18a] Bob Wheeler. RISC-V Enables IoT Edge Processor. Microprocessor Report 2/2018, The Linley Group.

[Whe19a] Bob Wheeler. Bitmain SoC Brings AI to the Edge. Microprocessor Report 2/2019, The Linley Group.

[LHC19] Wei-Fen Lin, Cheng-Tao Hsieh, and Cheng-Yi Chou. Onnc-based software development platform for configurable nvdla designs. In 2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), volume, 1–2. 2019. doi:10.1109/VLSI-DAT.2019.8741778.

[LTT+19] Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, and Luis Hsu. Onnc: a compilation framework connecting onnx to proprietary deep learning accelerators. In 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), volume, 214–218. 2019. doi:10.1109/AICAS.2019.8771510.

[MCV+19] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization. 2019. arXiv:1807.04188.

RLW+18] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new IR for machine learning frameworks. Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, Jun 2018. URL: http://dx.doi.org/10.1145/3211346.3211348, doi:10.1145/3211346.3211348.

[RFA+19] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph Lowering Compiler Techniques for Neural Networks. 2019. arXiv:1805.00907.

[ZWZ+18] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, Deming Chen, "DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs," 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018, pp. 1-8, doi: 10.1145/3240765.3240801.

[Apache] Apache Software Foundation. TVM Documentation. URL: https://tvm.apache.org/docs/ (visited on 2021-06-12).

[GoogleDa] Google Developers. FlatBuffers Documentation. URL: https://google.github.io/flatbuffers/index.html (visited on 2021-06-14).

[GoogleDb] Google Developers. TensorFlow Lite Documentation. URL: https://www.tensorflow.org/lite/guide?hl=en (visited on 2021-06-13).

[Gwe17a] Linley Gwennap: Cortex-A55 Improves Memory. Microprocessor Report 2017, The Linley Group.

[Gwe17b] Linley Gwennap: Cortex-A75 Has DynamIQ Debut. Microprocessor Report 2017, The Linley Group.

[Gwe18b] Linley Gwennap: Arm Dot Products Accelerate CNNs, Microprocessor Report 2018, The Linley Group.

[Xil21] Xilinx: DPUCZDX8G for Zynq UltraScale+ MPSoCs. https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_3/pg338-dpu.pdf, Jul 22, 2021

[Jan21c] Aakash Jani. SiFive Brings Vectors to S-Series. Microprocessor Report 1/2021, The Linley Group.

10. APPENDIX: Kenning report examples

This chapter shows the example benchmarks generated with Kenning tool on Jetson AGX Xavier and Google Coral Devboard devices.

10.1 TensorFlow MobileNetv2 performance on Jetson AGX Xavier compiled with Apache TVM

10.1.1 Inference performance metrics

This section was generated using the following Kenning invocation:

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

kenning.modelwrappers.classification.tensorflow_pet_dataset.TensorFlowPetD
atasetMobileNetV2 \
    kenning.compilers.tvm.TVMCompiler \
    kenning.runtimes.tvm.TVMRuntime \
    kenning.datasets.pet_dataset.PetDataset \
    ./build/jetson-agx-xavier-tvm-tensorflow.json \
    --protocol-cls \
        kenning.runtimeprotocols.network.NetworkProtocol \
    --model-path \

./kenning/resources/models/classification/tensorflow_pet_dataset_mobilenet
v2.h5 \
    --model-framework \
        keras \
    --target \
        nvidia/jetson-agx-xavier \
    --target-host \
        llvm -mtriple=aarch64-linux-gnu \
    --compiled-model-path \
        ./build/compiled-model.tar \
    --opt-level \
        3 \
    --host \
        192.168.188.32 \
    --port \
        12345 \
    --packet-size \
        32768 \
    --save-model-path \
        /home/nvidia/compiled-model.tar \
    --target-device-context \
        cuda \
    --dataset-root \
        ./build/pet-dataset/ \
    --inference-batch-size \
```

```
        1 \
    --verbosity \
        INFO

python -m kenning.kenning.scenarios.render_report \
    build/jetson-agx-xavier-tvm-tensorflow.json \
    TensorFlow MobileNetv2 on Jetson AGX Xavier compiled with Apache TVM \
    res/generated/jetson-agx-xavier-tvm-tensorflow.rst \
    --img-dir \
        res/generated/img \
    --root-dir \
        res \
    --report-types \
        performance \
        classification
```

10.1.1.1 General information

- *Model framework*: tensorflow ver. 2.4.1
- *Compiler framework*: tvm ver. 0.8.dev0

10.1.1.2 Inference time

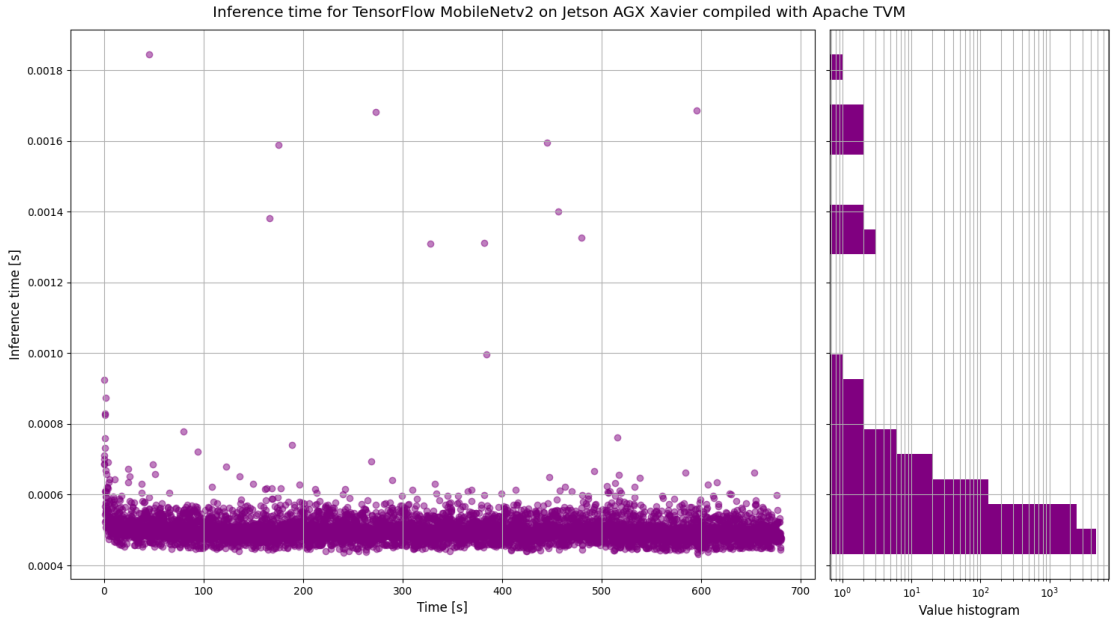

Figure 10.1 Inference time

- *First inference duration* (usually including allocation time): **0.7320548380002947**,
- *Mean*: **0.0006010748466419217 s**,
- *Standard deviation*: **0.008533153473426609 s**,
- *Median*: **0.0004953470001964888 s**.

10.1.1.3 Mean CPU usage

Figure 10.2: Mean CPU usage during benchmark

- *Mean*: **12.820110249849488 %**,
- *Standard deviation*: **1.5393588765298858 %**,
- *Median*: **12.6125 %**.

10.1.1.4 Memory usage

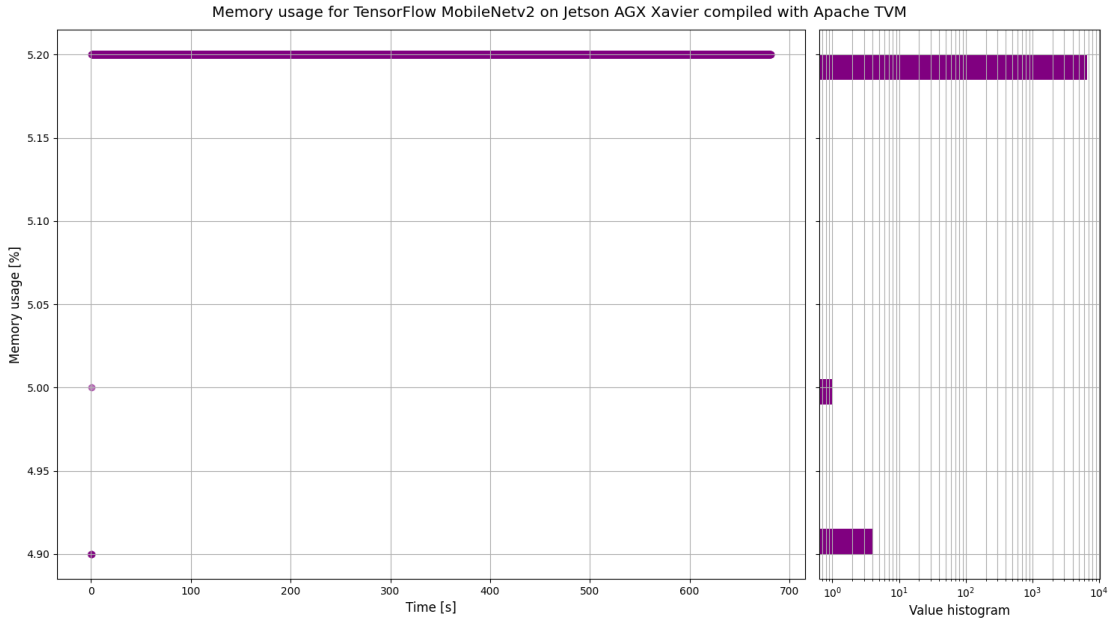

Figure 10.3: Memory usage during benchmark

- *Mean*: **5.19978928356412 %**,
- *Standard deviation*: **0.007756306759659168 %**,
- *Median*: **5.2 %**.

10.1.1.5 GPU usage

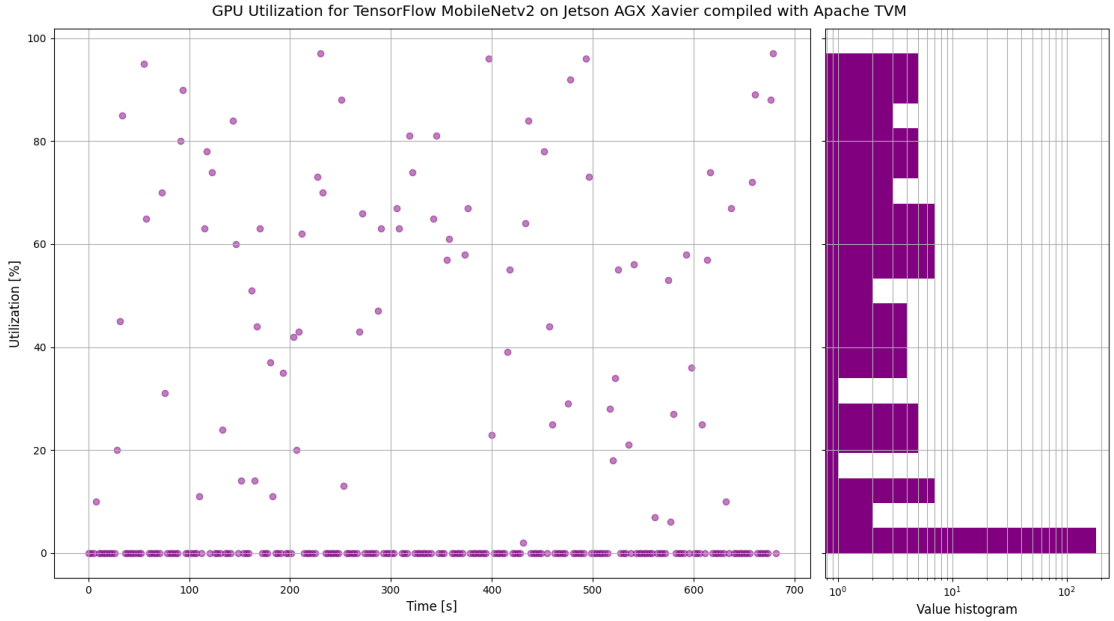

Figure 10.4: GPU utilization during benchmark

- *Mean*: **16.919847328244273 %**,
- *Standard deviation*: **28.979679429283195 %**,
- *Median*: **0.0 %**.

10.1.1.6 GPU memory usage

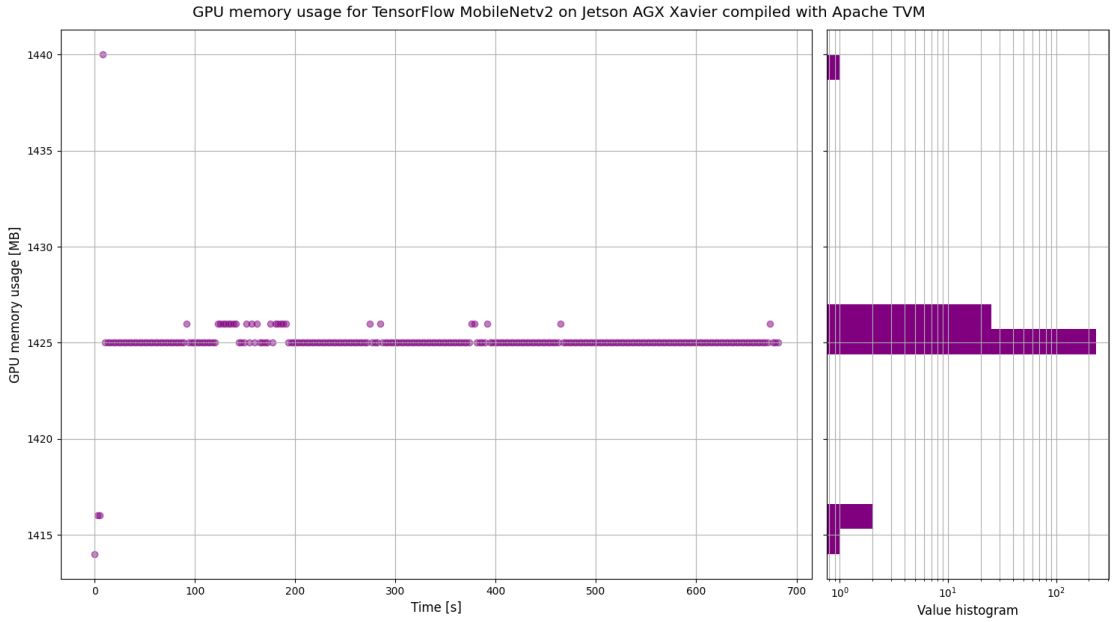

Figure 10.5: GPU memory usage during benchmark

- *Mean*: **1425.0419847328244 MB**,
- *Standard deviation*: **1.425688755390822 MB**,
- *Median*: **1425.0 MB**.

10.1.2 Inference quality metrics

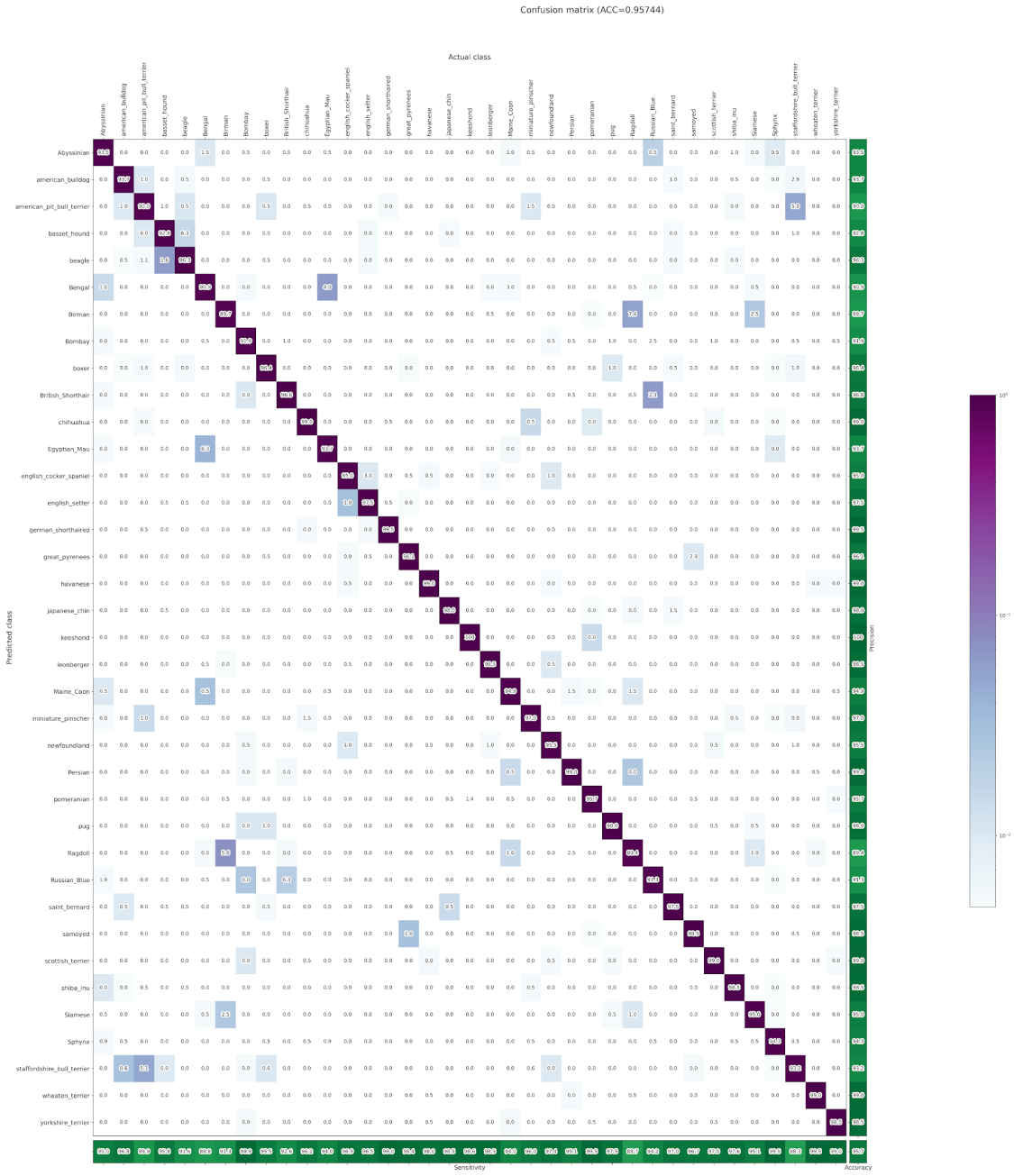

Figure 10.6: Confusion matrix

- *Accuracy*: 0.957273098380732
- *Top-5 accuracy*: 0.9961899578173902
- *Mean precision*: 0.9571990916404879
- *Mean sensitivity*: 0.9574417601857972
- *G-mean*: 0.9569641373040098

10.2 PyTorch MobileNetv2 performance on Jetson AGX Xavier compiled with Apache TVM

10.2.1 Inference performance metrics

This section was generated using the following Kenning invocation:

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

kenning.modelwrappers.classification.pytorch_pet_dataset.PyTorchPetDataset
MobileNetV2 \
    kenning.compilers.tvm.TVMCompiler \
    kenning.runtimes.tvm.TVMRuntime \
    kenning.datasets.pet_dataset.PetDataset \
    ./build/jetson-agx-xavier-tvm-pytorch.json \
    --protocol-cls \
        kenning.runtimeprotocols.network.NetworkProtocol \
    --model-path \

./kenning/resources/models/classification/pytorch_pet_dataset_mobilenetv2.
pth \
    --convert-to-onnx \
        ./build/pytorch_pet_dataset_mobilenetv2.onnx \
    --target \
        cuda -keys=cuda,gpu -libs=cudnn,cublas -arch=sm_72 -
max_num_threads=1024 -max_threads_per_block=1024 -
registers_per_block=65536 -shared_memory_per_block=49152 -
thread_warp_size=32 \
    --target-host \
        llvm -mtriple=aarch64-linux-gnu \
    --compiled-model-path \
        ./build/compiled-model.tar \
    --opt-level \
        3 \
    --host \
        192.168.188.32 \
    --port \
        12345 \
    --packet-size \
        32768 \
    --save-model-path \
        /home/nvidia/compiled-model.tar \
    --target-device-context \
        cuda \
    --dataset-root \
        ./build/pet-dataset/ \
    --inference-batch-size \
        1 \
    --verbosity \
        INFO
```

```
python -m kenning.kenning.scenarios.render_report \
    build/jetson-agx-xavier-tvm-pytorch.json \
    PyTorch MobileNetv2 on Jetson AGX Xavier compiled with Apache TVM \
    res/generated/jetson-agx-xavier-tvm-pytorch.rst \
    --img-dir \
        res/generated/img \
    --root-dir \
        res \
    --report-types \
        performance \
        classification
```

10.2.1.1 General information

- *Model framework*: torch ver. 1.8.1+cu102
- *Compiler framework*: tvm ver. 0.8.dev0

10.2.1.2 Inference time

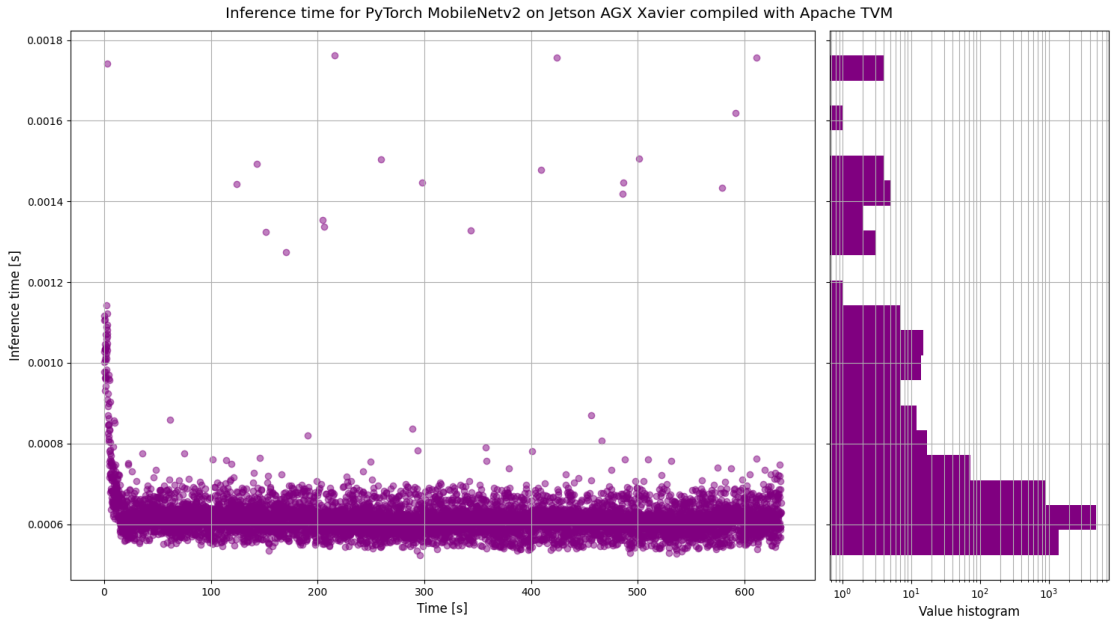

Figure 10.7: Inference time

- *First inference duration* (usually including allocation time): **2.358816786000034**,
- *Mean*: **0.0009398624625117085 s**,
- *Standard deviation*: **0.02750665627498466 s**,
- *Median*: **0.000612113999977737 s**.

10.2.1.3 Mean CPU usage

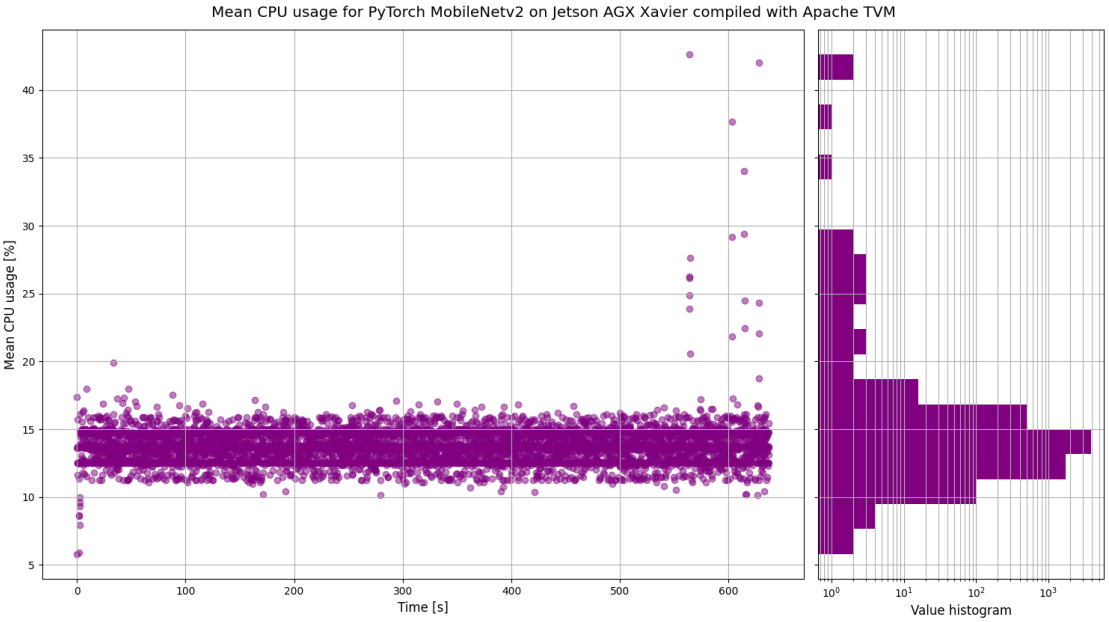

Figure 10.8: Mean CPU usage during benchmark

- *Mean*: **13.766375984568397 %**,
- *Standard deviation*: **1.3958985145804446 %**,
- *Median*: **13.75 %**.

10.2.1.4 Memory usage

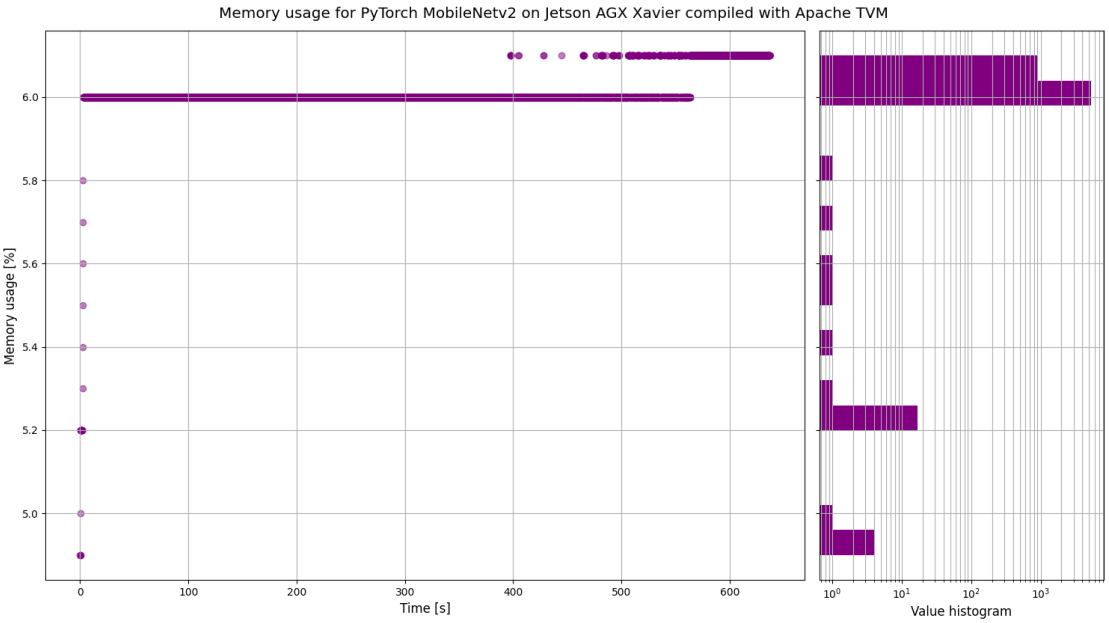

Figure 10.9: Memory usage during benchmark

- *Mean*: **6.010995016878314 %**,

- *Standard deviation*: **0.06510405017033213 %**,
- *Median*: **6.0 %**.

10.2.1.5 GPU usage

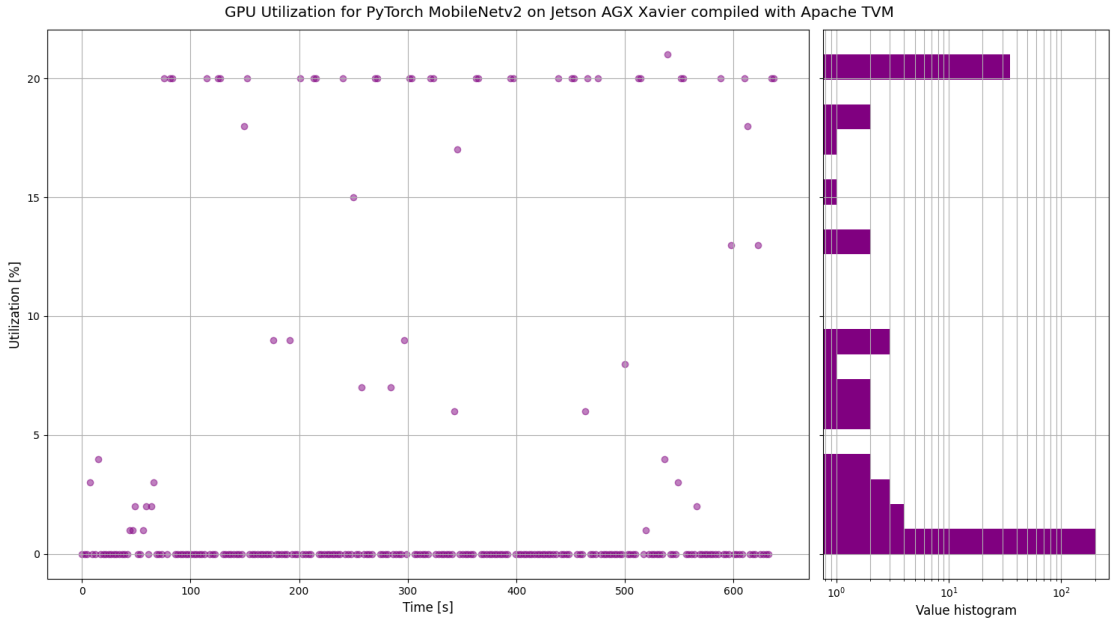

Figure 10.10: GPU utilization during benchmark

- *Mean*: **3.3908045977011496 %**,
- *Standard deviation*: **7.0835565701224175 %**,
- *Median*: **0.0 %**.

10.2.1.6 GPU memory usage

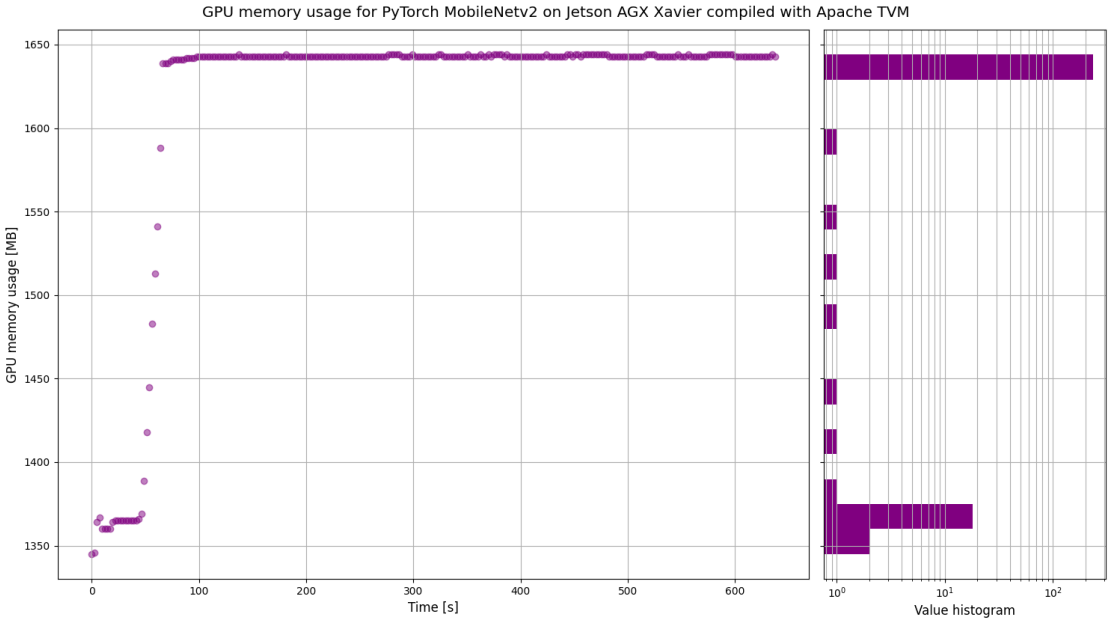

Figure 10.11: GPU memory usage during benchmark

- *Mean*: **1617.2643678160919 MB**,
- *Standard deviation*: **78.64091626621843 MB**,
- *Median*: **1643.0 MB**.

10.2.2 Inference quality metrics

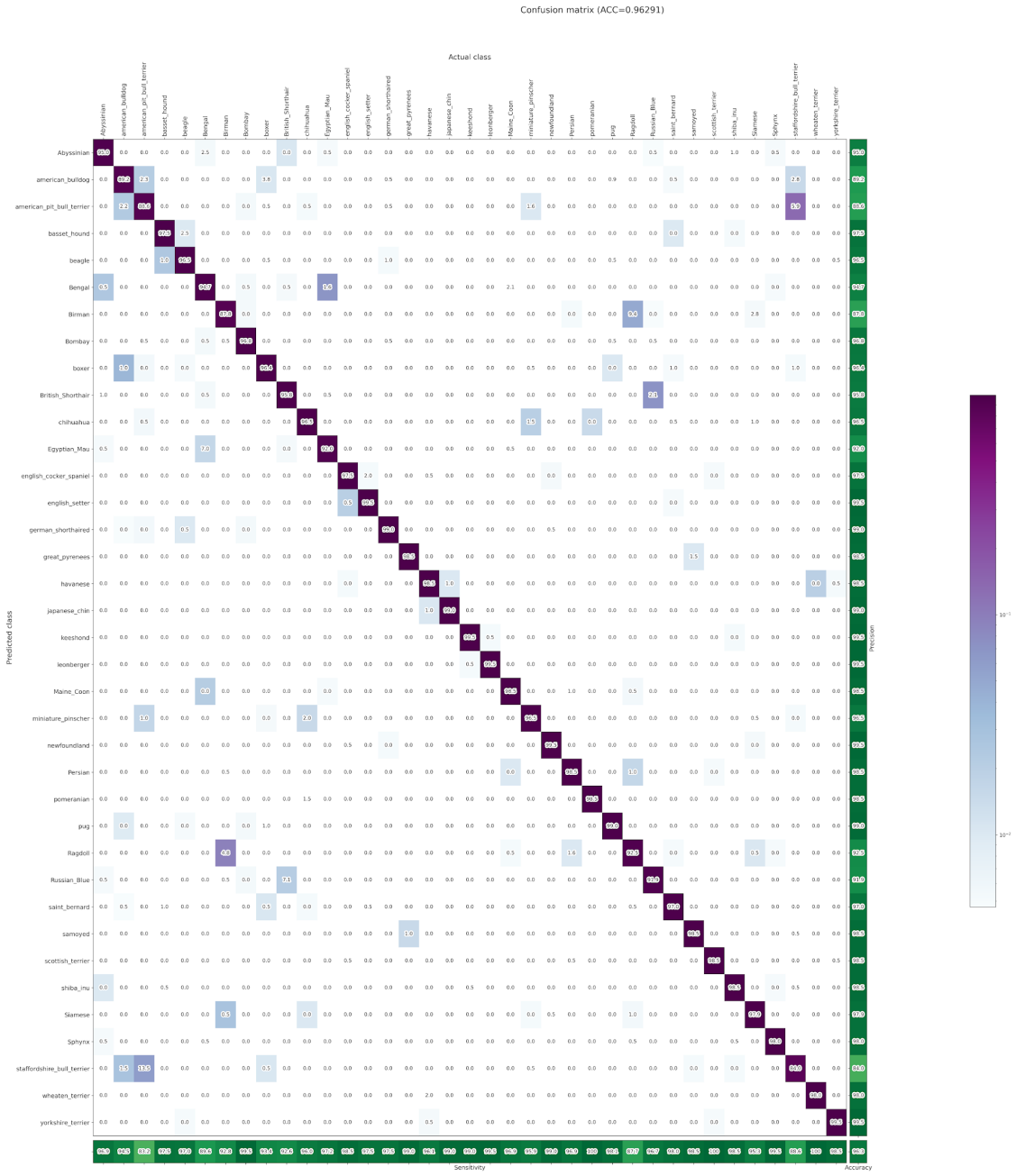

Figure 10.12: Confusion matrix

- *Accuracy*: 0.9627160157844604
- *Top-5 accuracy*: 0.9979589059736018
- *Mean precision*: 0.9627208598943292
- *Mean sensitivity*: 0.9629103528032014
- *G-mean*: 0.9621582124636907

10.3 TensorFlow MobileNetv2 performance on Google Coral Devboard compiled with TensorFlow Lite

10.3.1 Inference performance metrics

This section was generated using the following Kenning invocation:

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

kenning.modelwrappers.classification.tensorflow_pet_dataset.TensorFlowPetD
atasetMobileNetV2 \
    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_mobilenet
v2.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

python -m kenning.kenning.scenarios.render_report \
    build/google-coral-devboard-tflite-tensorflow.json \
    TensorFlow MobileNetv2 on Google Coral Devboard compiled with
TensorFlow Lite \
```

```
res/generated/google-coral-devboard-tflite-tensorflow.rst \
--img-dir \
    res/generated/img \
--root-dir \
    res \
--report-types \
    performance \
    classification
```

10.3.1.1 General information

- *Model framework*: tensorflow ver. 2.4.1
- *Compiler framework*: tensorflow ver. 2.4.1

10.3.1.2 Inference time

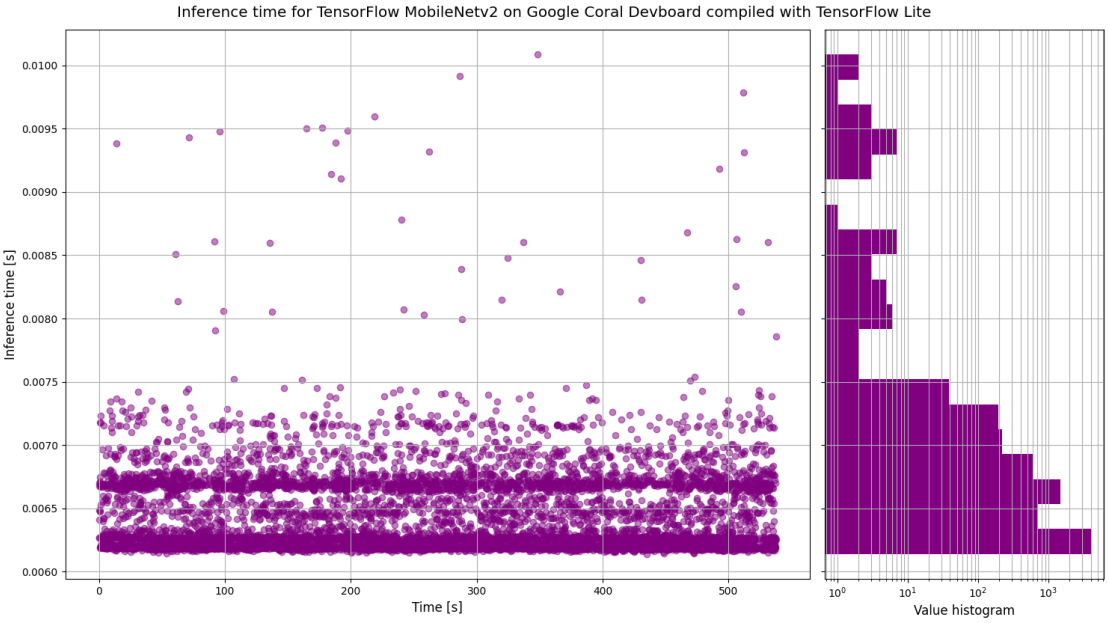

Figure 10.13: Inference time

- *First inference duration* (usually including allocation time): **0.016356768000150623**,
- *Mean*: **0.0064518570774245694 s**,
- *Standard deviation*: **0.0003517888111368472 s**,
- *Median*: **0.006280698000409757 s**.

10.3.1.3 Mean CPU usage

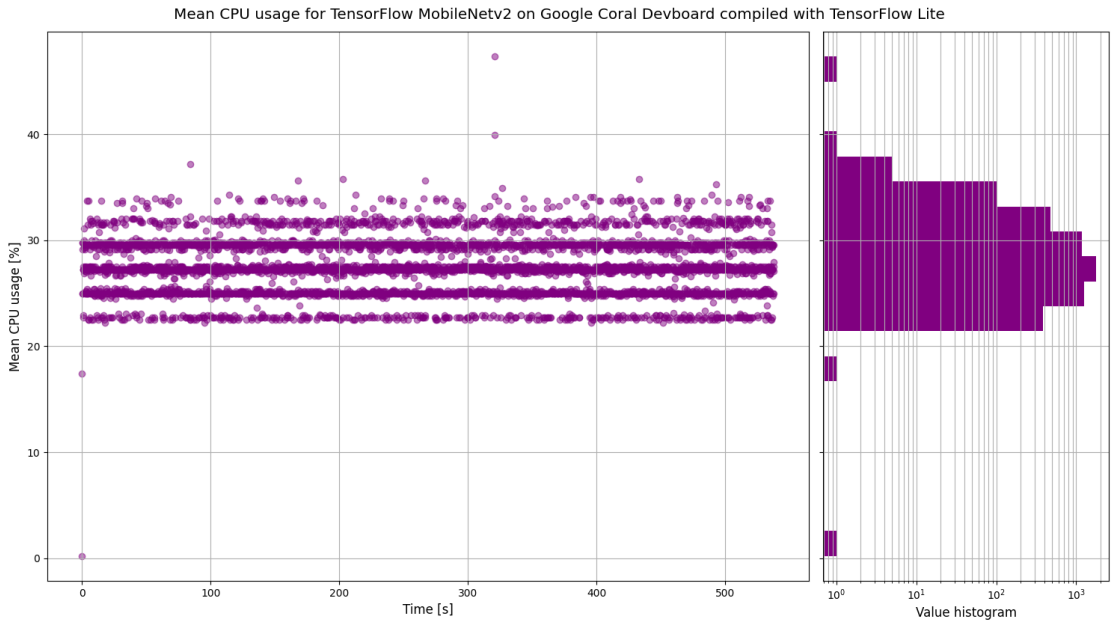

Figure 10.14: Mean CPU usage during benchmark

- *Mean*: **27.464499419055 %**,
- *Standard deviation*: **2.640756305937856 %**,
- *Median*: **27.275 %**.

10.3.1.4 Memory usage

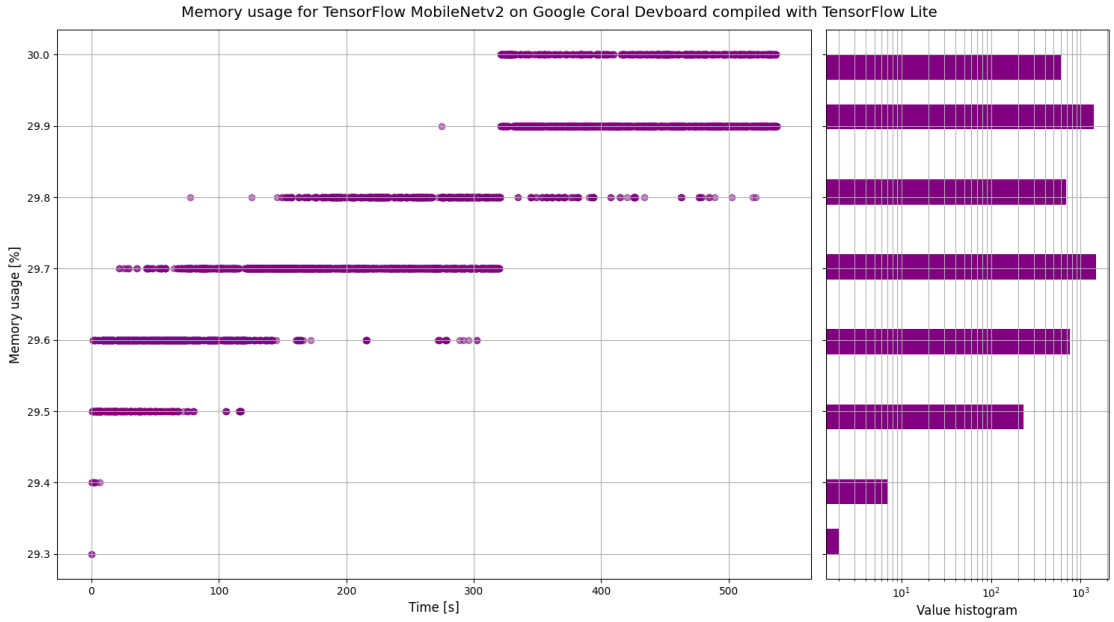

Figure 10.15: Memory usage during benchmark

- *Mean*: **29.77813710302091 %**,
- *Standard deviation*: **0.1411483821669738 %**,
- *Median*: **29.8 %**.

10.3.2 Inference quality metrics

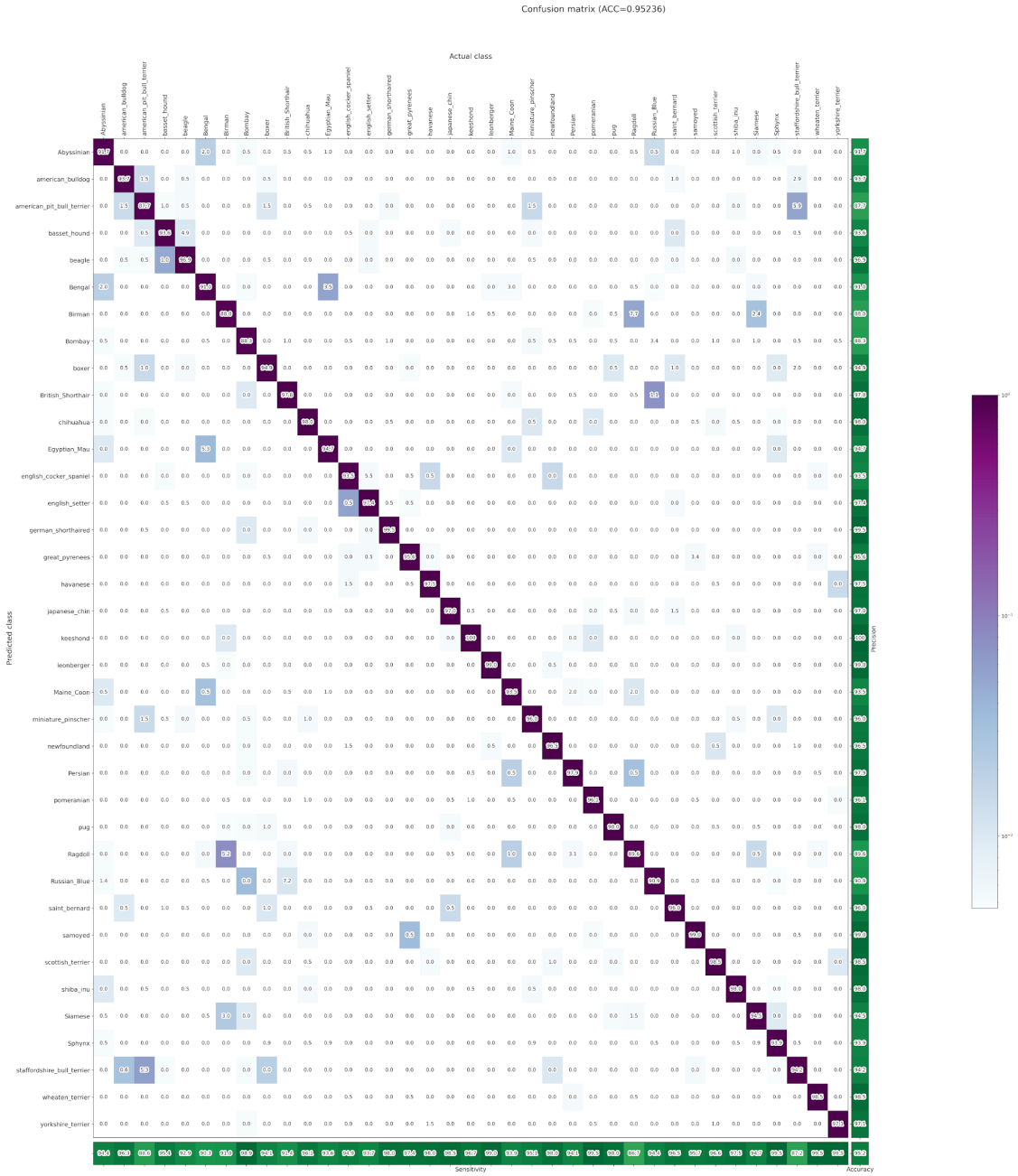

Figure 10.16: Confusion matrix

- *Accuracy*: 0.9519662539120969

- *Top-5 accuracy*: 0.9955095931419241
- *Mean precision*: 0.9518823637507444
- *Mean sensitivity*: 0.9523612306544653
- *G-mean*: 0.9517786196628326

10.4 Darknet YOLOv3 performance on Jetson AGX Xavier compiled with Apache TVM

10.4.1 Inference performance metrics

This section was generated using the following Kenning invocation:

```
python -m kenning.kenning.scenarios.inference_tester \
    kenning.modelwrappers.detectors.darknet_coco.TVMDarknetCOCOYOLOV3 \
    kenning.compilers.tvm.TVMCompiler \
    kenning.runtimes.tvm.TVMRuntime \
    kenning.datasets.open_images_dataset.OpenImagesDatasetV6 \
    ./build/jetson-agx-xavier-tvm-darknet.json \
    --protocol-cls \
        kenning.runtimeprotocols.network.NetworkProtocol \
    --model-path \
        ./kenning/resources/models/detection/yolov3.weights \
    --model-framework \
        darknet \
    --target \
        cuda -keys=cuda,gpu -libs=cudnn,cublas -arch=sm_72 -
max_num_threads=1024 -max_threads_per_block=1024 -
registers_per_block=65536 -shared_memory_per_block=49152 -
thread_warp_size=32 \
    --target-host \
        llvm -mtriple=aarch64-linux-gnu \
    --compiled-model-path \
        ./build/compiled-model.tar \
    --opt-level \
        3 \
    --host \
        192.168.188.32 \
    --port \
        12345 \
    --packet-size \
        32768 \
    --save-model-path \
        /home/nvidia/compiled-model.tar \
    --target-device-context \
        cuda \
```

```
    --dataset-root \
        ./build/open-images-dataset \
    --inference-batch-size \
        1 \
    --libdarknet-path \
        ./libdarknet-cuda10.so \
    --verbosity \
        INFO

python -m kenning.kenning.scenarios.render_report \
    build/jetson-agx-xavier-tvm-darknet.json \
    Darknet YOLOv3 on Jetson AGX Xavier compiled with Apache TVM \
    res/generated/jetson-agx-xavier-tvm-darknet.rst \
    --img-dir \
        res/generated/img \
    --root-dir \
        res \
    --report-types \
        performance \
        detection
```

10.4.1.1 General information

- *Model framework*: darknet ver. alexeyab
- *Compiler framework*: tvm ver. 0.8.dev0

10.4.1.2 Inference time

Figure 10.17: Inference time¶

- *First inference duration* (usually including allocation time): **3.6642245220000405**,

- *Mean*: **0.012031477093405607 s**,
- *Standard deviation*: **0.051218213271858506 s**,
- *Median*: **0.011220539000078134 s**.

10.4.1.3 Mean CPU usage

Figure 10.18: Mean CPU usage during benchmark

- *Mean*: **8.66158312205085 %**,
- *Standard deviation*: **5.718338128021855 %**,
- *Median*: **9.8875 %**.

10.4.1.4 Memory usage

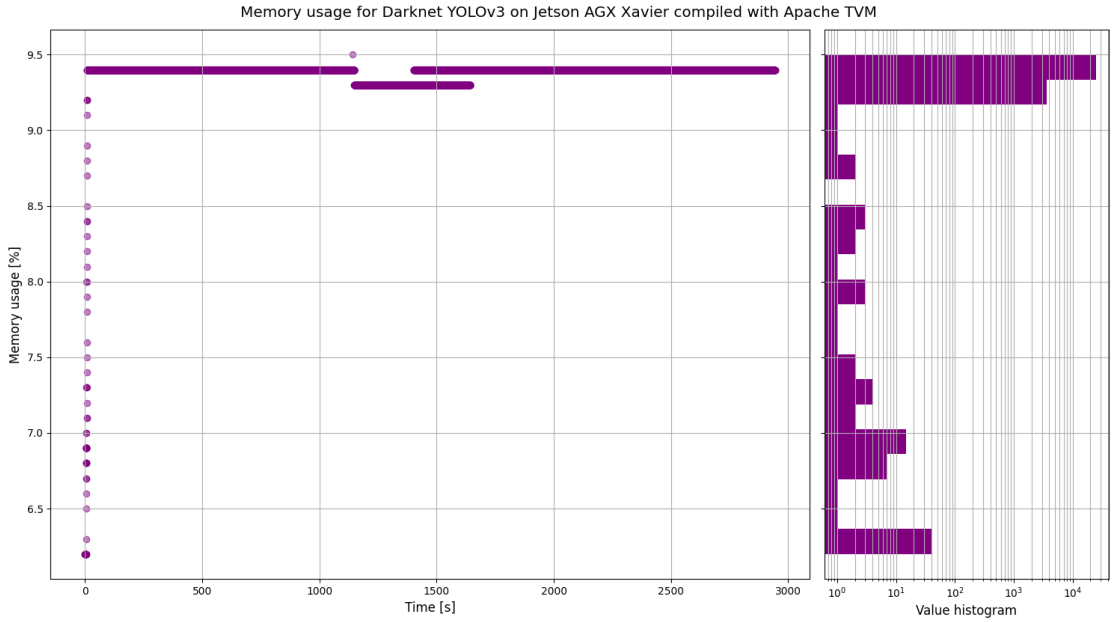

Figure 10.19: Memory usage during benchmark

- *Mean*: **9.379459821114166 %**,
- *Standard deviation*: **0.15095406538474948 %**,
- *Median*: **9.4 %**.

10.4.1.5 GPU usage

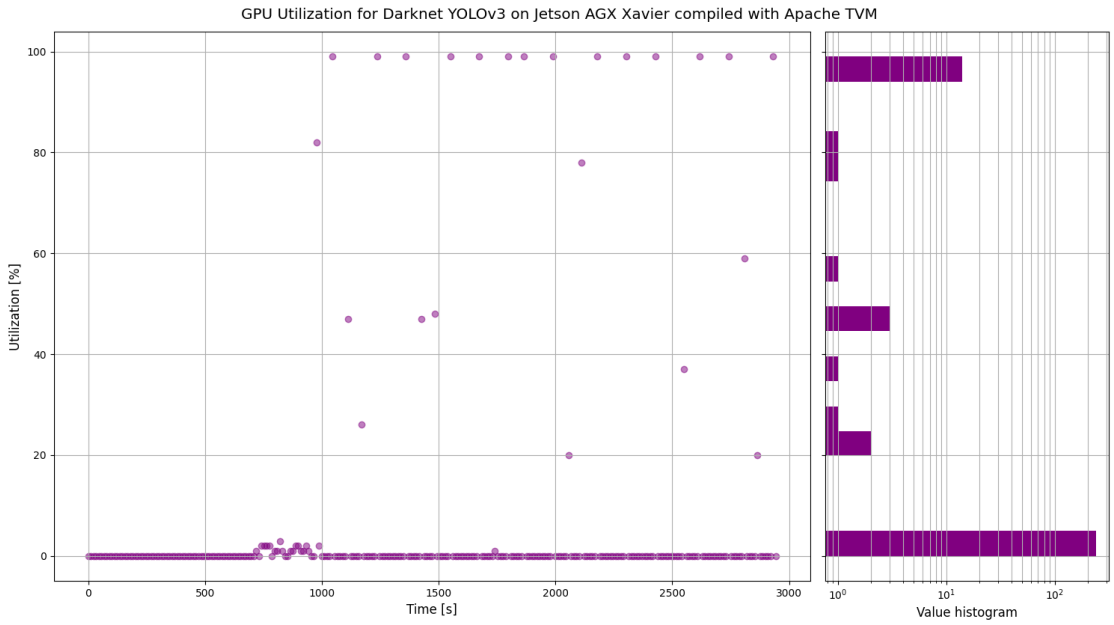

Figure 10.20: GPU utilization during benchmark

- *Mean*: **7.144486692015209 %**,
- *Standard deviation*: **23.857666463238097 %**,
- *Median*: **0.0 %**.

10.4.1.6 GPU memory usage

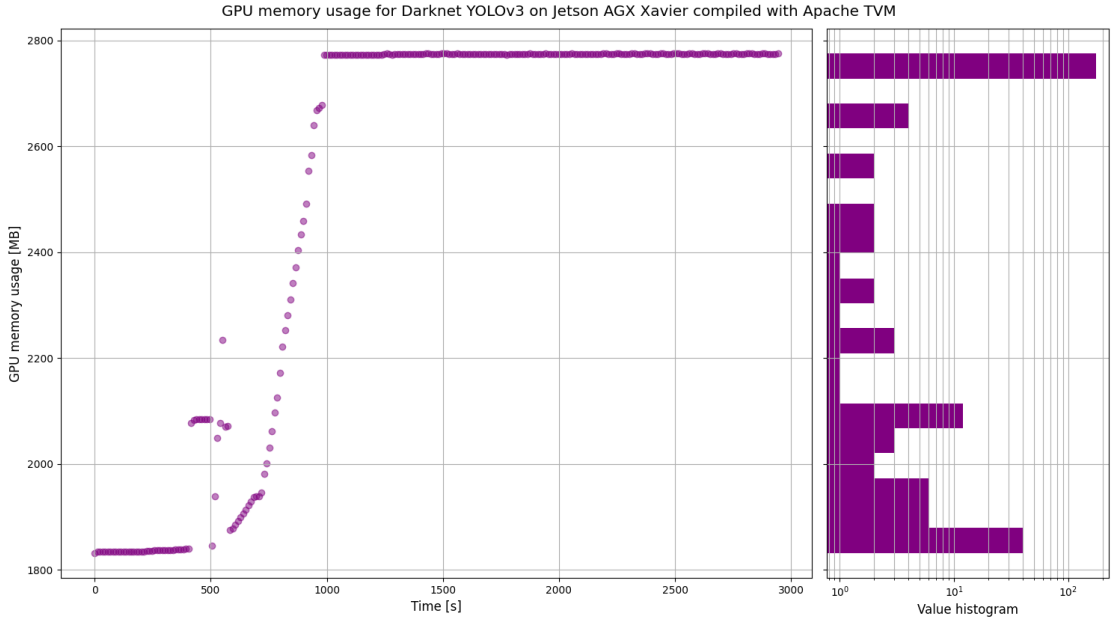

Figure 10.21: GPU memory usage during benchmark

- *Mean*: **2521.0076045627375 MB**,
- *Standard deviation*: **382.71878101474215 MB**,
- *Median*: **2774.0 MB**.

10.4.2 Object detection metrics

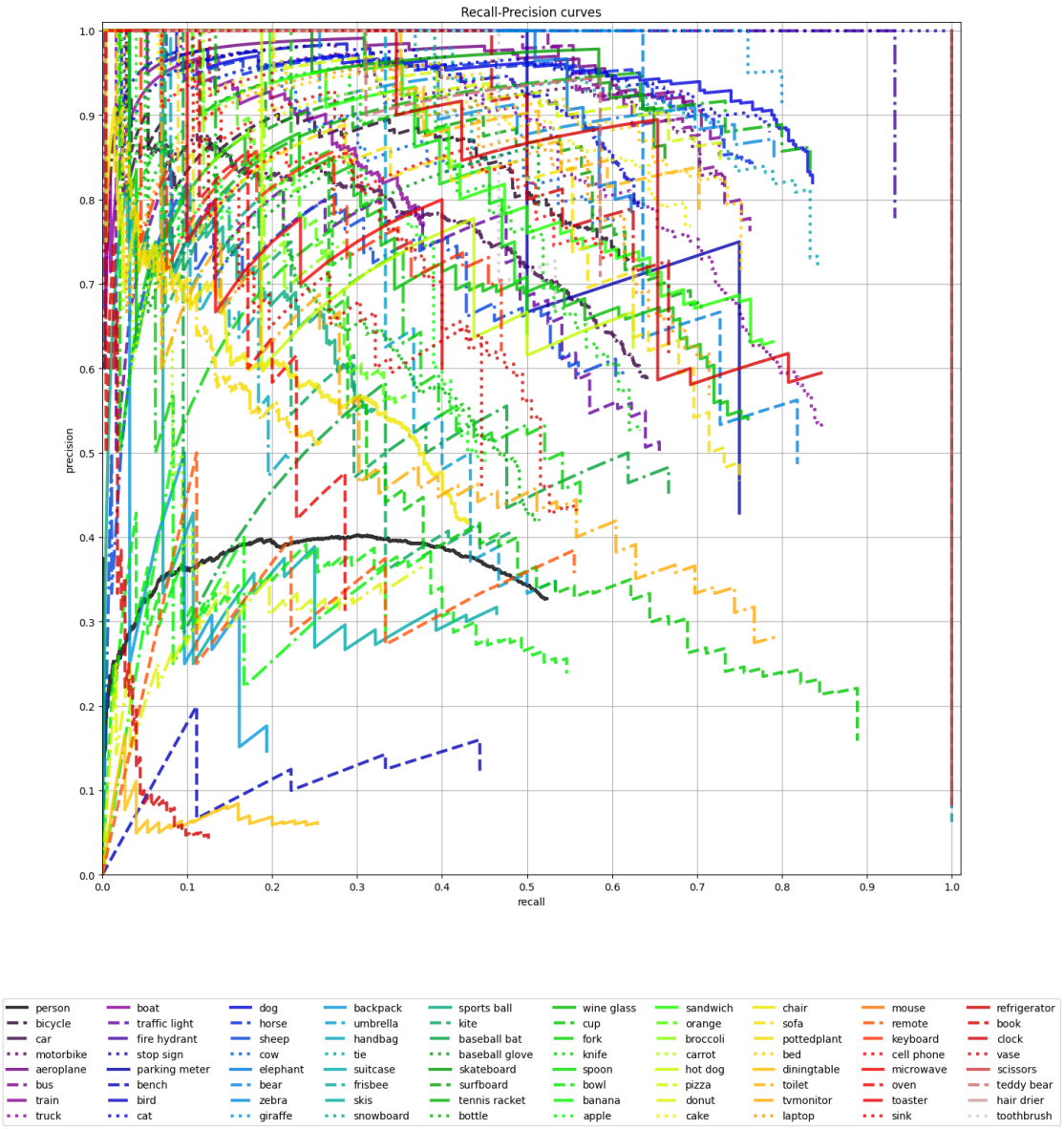

Figure 10.22: Per-Class Recall-Precision curves

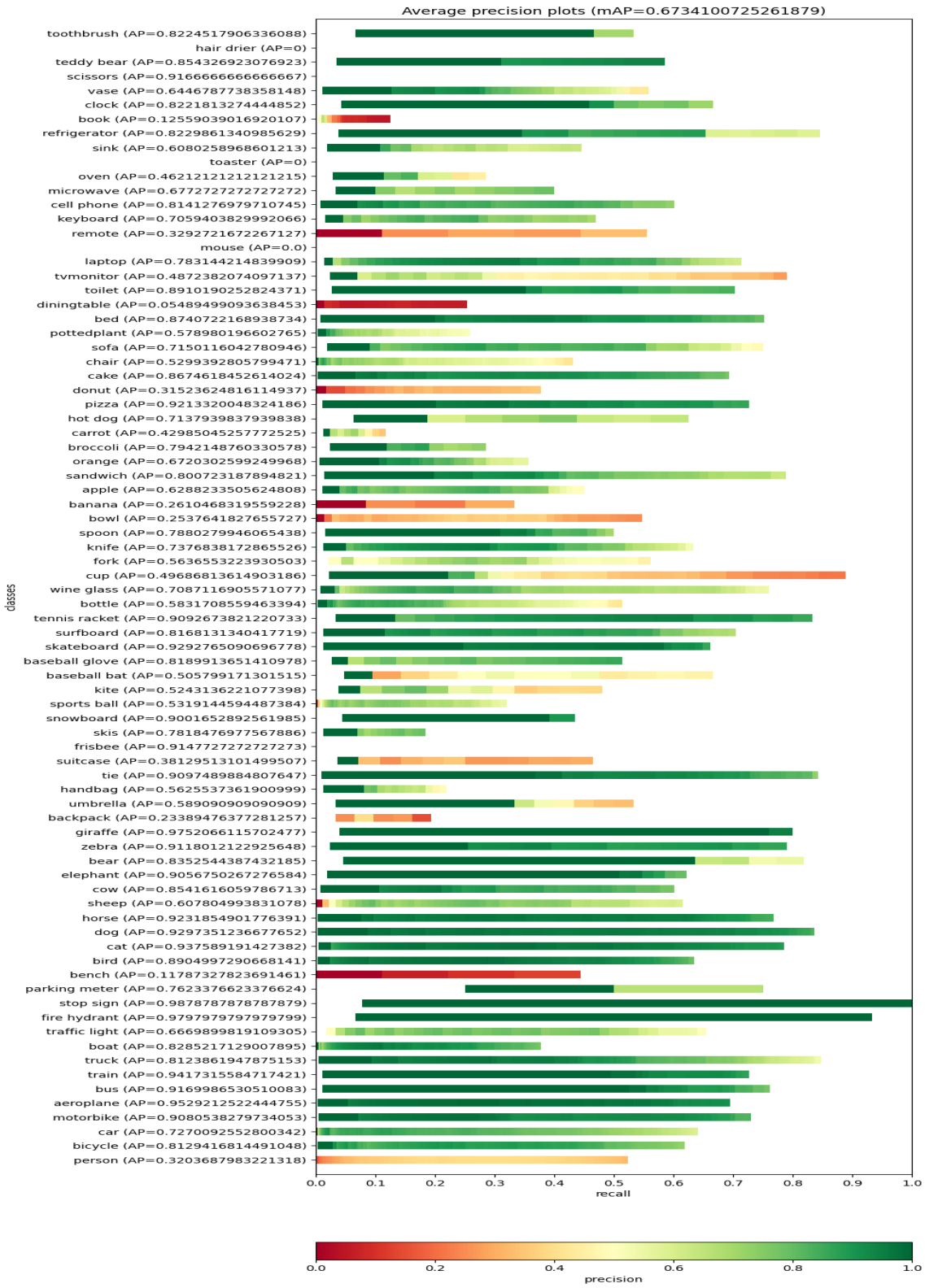

Figure 10.23: Per-Class precision gradients

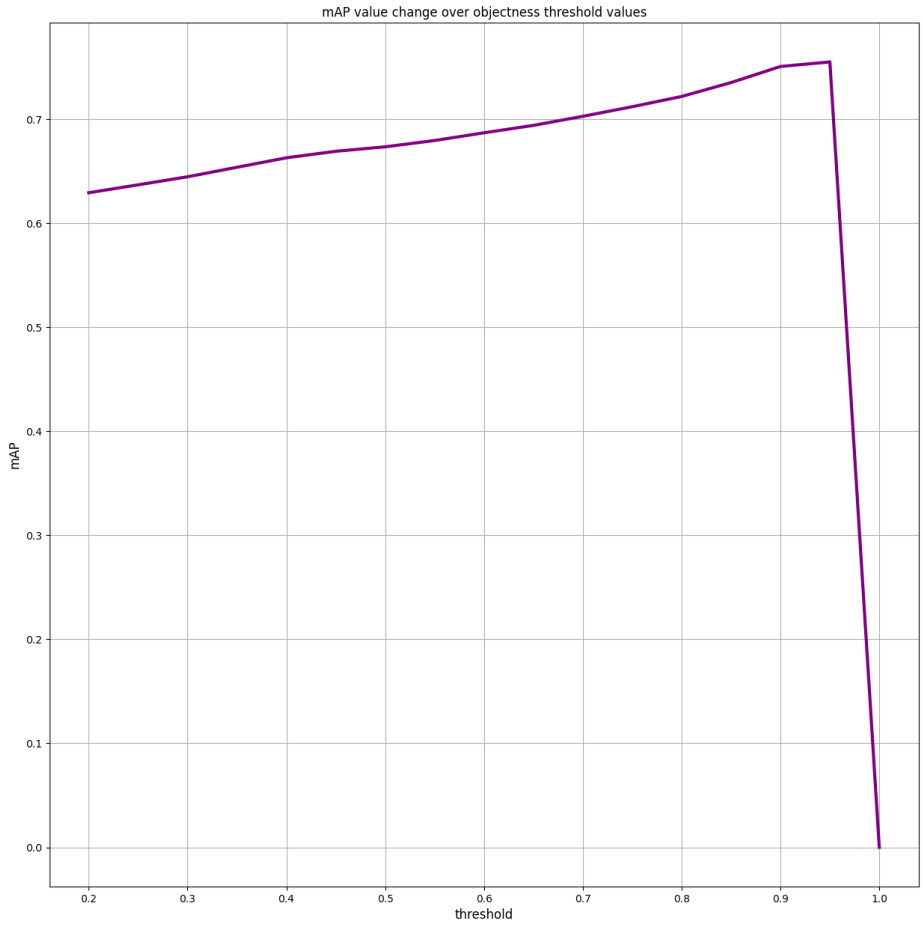

Figure 10.24 mAP values depending on threshold

- *Mean Average Precision* for threshold 0.5: 0.6734100725261879