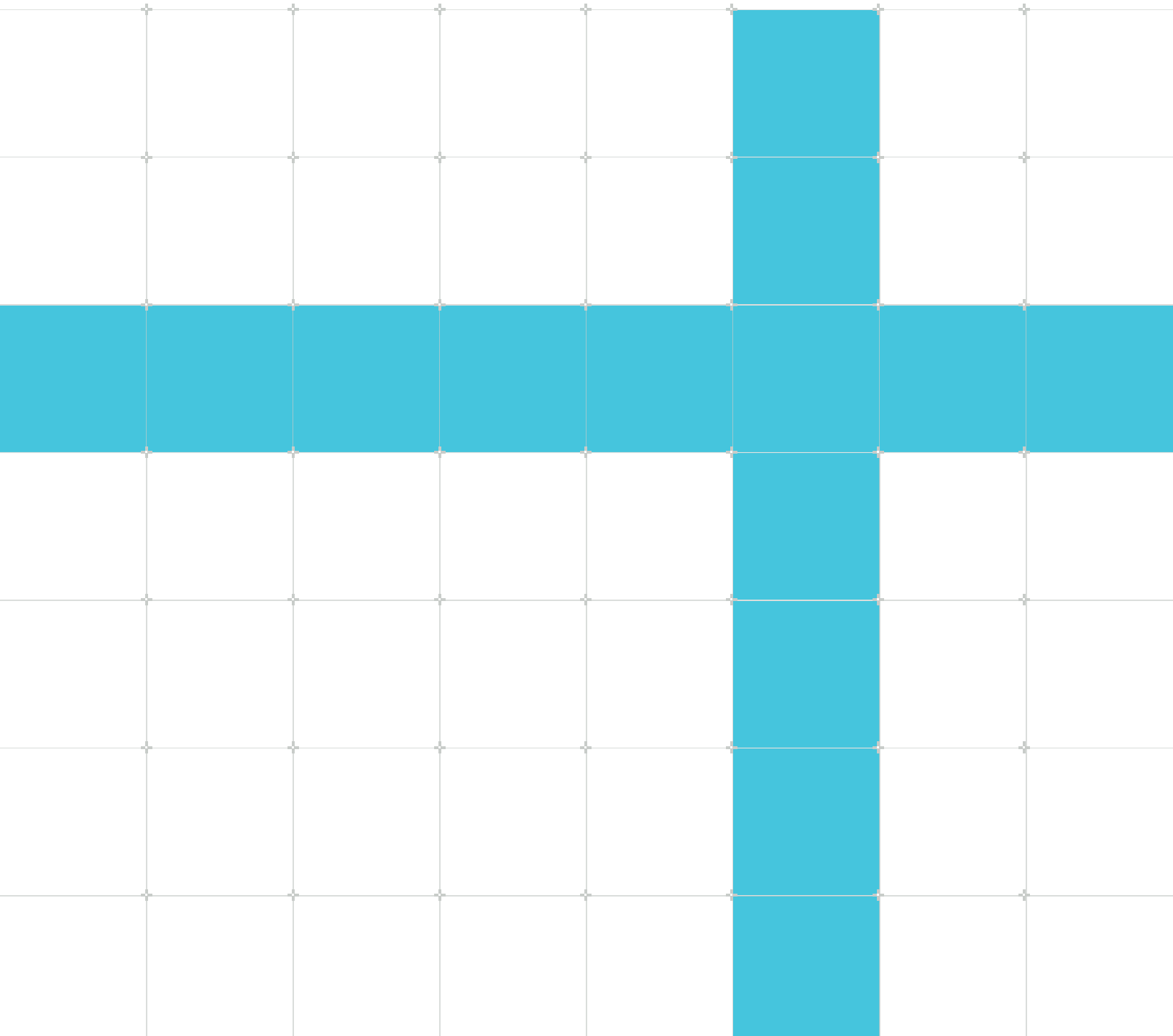
ML Developers Guide for Cortex-M Processors and Ethos-U NPU

Version 1.0

Issue 03
109267_0100_03_en



ML Developers Guide for Cortex-M Processors and Ethos-U NPU

Release information

Document history

Issue	Date	Confidentiality	Change
0100-03	13 October 2023	Non-Confidential	Initial release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

Thank you for using the Arm Cortex-M processors optionally with an Ethos-U Network Processing Unit (NPU) in your Machine Learning (ML) edge device application. To provide you with the best experience for developing ML applications with Arm processors we design offer solutions that cover hardware IP, tools, and software to make product development easy and productive. In addition Arm provides supportive material and collaborates with many AI partners to complement our solution, for example with optimized ML models, MLOps integrations, and evaluation boards. The Arm tool and software support is therefore expandable in various ways.

This Overview chapter contains:

1.1 Chapter overview

This User's Guide starts with this overview of the ML development process. It introduces you to the Arm technology and products that support ML development workflows from starting ML model training up to debugging on hardware. In addition it contains the following chapters:

- ML software development on Arm Cortex-M Processors describes the concept of ML software development for resource constraint systems, exemplified with TensorFlow Lite for Microcontollers (TFLM).
- Arm Ethos-U NPU provides information about the various Ethos-U processors and access to detailed technical documentation including hardware and software considerations.
- Tool Support for the Arm Ethos-U NPU explains the Vela compiler that transforms a ML model for execution on Ethos-U NPU and covers the ML Interference Advisor, Arm Virtual Hardware, and SDS-Framework for analysis, verification, and training.
- The Arm ML Zoo which gives you access to pre-trained models for various types of applications. It explains how to use the information that is provided.
- ML Embedded Evaluation Kit provides ML examples for a range of use cases that help to create your own applications for Cortex-M CPU and Ethos-U NPU.
- CMSIS-Pack Based Machine Learning Examples showing ML software integration using software components in form of CMSIS-Packs, and using of CMSIS-Toolbox to create compilation setup.

- [Project Integration of TensorFlow Lite Micro] provides an overview of the integration of the TensorFlow Lite Micro runtime and models into new or existing projects.
- Profiling and optimization of ML models describes how to analyze and optimize the ML model execution on Arm Ethos-U processor based systems.
- MLOps Systems describes the integration of the Arm foundation tools into MLOps systems that automate training and help selecting optimal ML models for your applications.
- Resources for Ethos-U gives an overview of the available resources and eco-system partners that support Ethos-U NPUs.

1.2 Target Audience

This Getting Started user's guide assumes some top-level knowledge about Cortex-M software development. It is written for the following audiences:

- Embedded Developers that want to utilize microcontroller devices that incorporate Ethos-U processors and need easy access to development tools, software examples, and additional usage information.
- MLOps system architects that want to extend support to the Ethos-U NPU processor series and need to integrate the various development tools into the development flows.
- Data scientists that analyze data to develop new ML models and require statistics of the model performance by using the various software tools.

1.3 Benefits of Machine Learning

Applying machine learning in edge devices enables a new range of applications. For example:

- Predictive maintenance: where sensors in a system are used to identify likely failures, allowing for proactive maintenance to prevent downtime.
- Speech recognition, making it possible to interact with devices using natural language.
- Image detection in factory automation to improve efficiency, reduce errors, and increase safety in manufacturing environments.
- Medical diagnosis to assist in medical treatment, helping to design personalized treatment plans.
- Enable robots to perceive and understand their environment, such as recognizing objects, detecting obstacles, and tracking people. These are just a few examples. The overall possibilities of machine learning are vast, and the technology is constantly evolving making it possible to innovate in many new areas.

Typically, machine learning models require a high quantity of reliable data for the models to perform accurate predictions. When training a machine learning model, engineers need to target and collect a large and representative sample of data. Data from the training set can be a collection of images, sensor data, and data collected from individual users of a device.

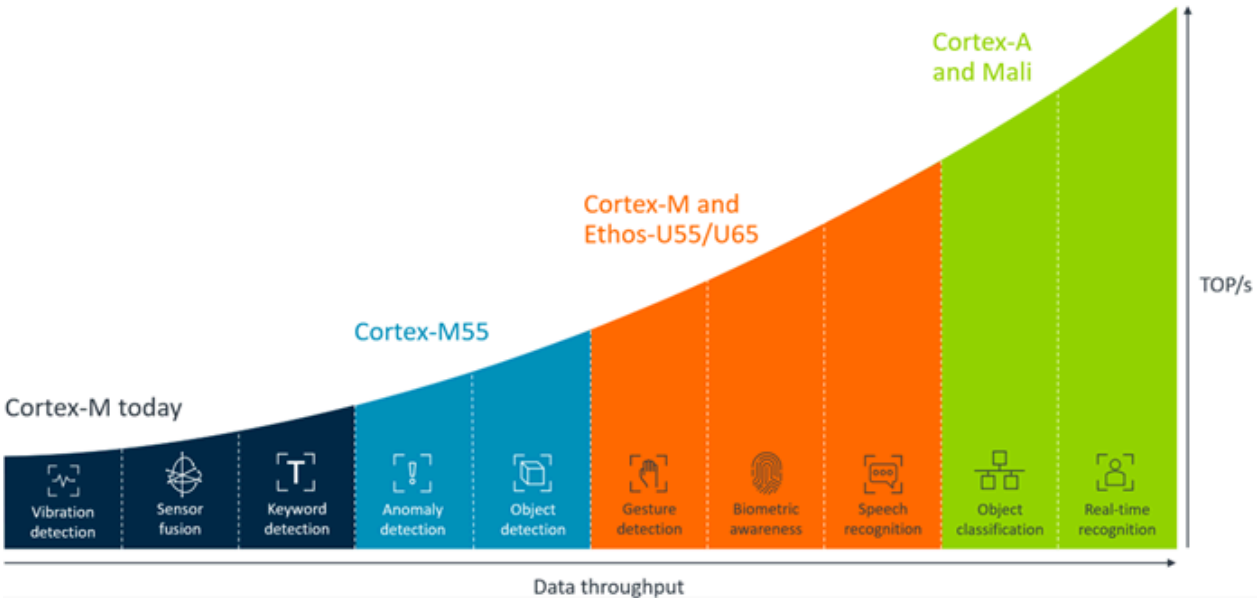
Today, AI and ML algorithms that operate on data from IoT endpoint devices frequently execute on cloud servers. But to meet real-time requirements of embedded systems the execution of the actual AI algorithm should be transferred to the edge device.

1.4 Compute Requirements

The compute requirements for machine learning algorithms can vary widely depending on the type of algorithm, the size of the dataset that is required during model training, the overall complexity of the problem, and the timing requirements of the application.

Therefore, Arm offers a broad range of optimized processors targeting machine learning applications on edge devices (see figure).

Figure 1-1: Arm ML Processor Portfolio



Arm ML Processor Portfolio:

- Even the smallest Arm processor, the Cortex-M0/M0+ processor, executes simple ML algorithms. For example, a predictive maintenance system that uses sensor data is possible with this type of processor.
- Starting with Cortex-M4, the processors offer hardware floating point arithmetic and SIMD instructions that accelerate DSP and simple machine learning algorithms. With such processors applications that utilize sensor fusion, a combination of various sensor data sources can be implemented.
- The Cortex-M55 and Cortex-M85 processors extend the architecture with Helium vector instructions that enable more demanding ML algorithms as they are required for speech keyword spotting or object and anomaly detection.

- The Ethos-U is a micro NPU which enable extremely low-power machine learning inference at the endpoint. It works in combination with an Arm Cortex processor and provides an enormous increase in ML performance.

For more demanding applications the Arm ML Processor Portfolio includes also Mali graphics processors, or the Cortex-A processors. However, this manual focuses on the development flow of tiny edge devices that use a single-core Cortex-M processor optionally paired with an Ethos-U NPU.

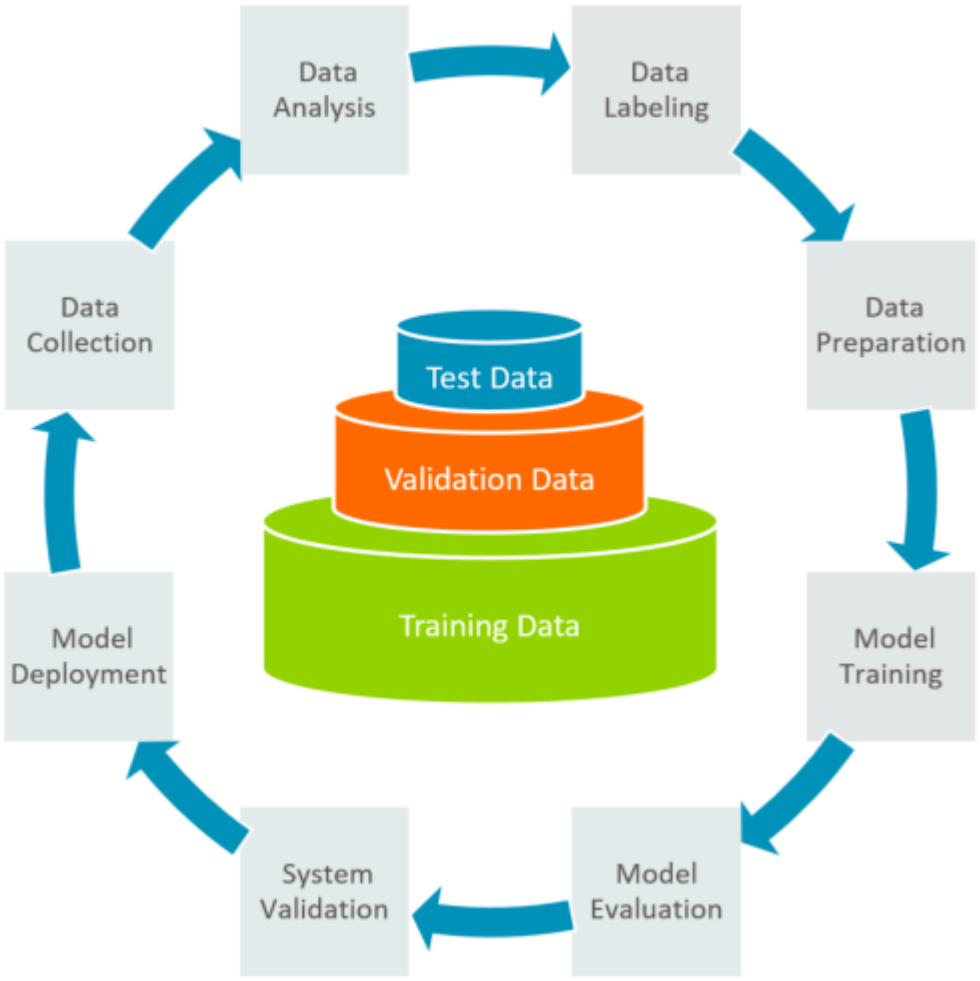
1.5 Overall Development Process

The software and system design of an embedded IoT and ML device can be separated into two parts:

- The classic embedded IoT software that requires efficient device drivers that interface with peripherals, a communication stack with security, and firmware update services.
- The system part that implements the machine learning algorithm. The ML part is frequently designed by utilizing Software-as-a-Service (SaaS) cloud environments that are specialized for ML algorithm development.

The machine learning algorithm is developed using a MLOps workflow. MLOps is a set of practices for developing, deploying, and maintaining machine learning models in production devices. The picture below shows the process steps of a MLOps development flow.

Figure 1-2: MLOps Process Steps



MLOps Process Steps:

- Data Collection: is the foundation of a machine learning project. It is important that the machine learning model has enough data to learn from, the data covers as much scenarios as possible, and the data collected are accurate as this is critical for the performance of the model.
- Data Analysis: requires understanding of the scenarios that are recorded in the data collection. It may require that the data is cropped or cleaned before progressing to the next step as the collected data may have a mixture of scenarios.
- Data Labelling: is the annotation of the collected and cleaned data. For example, the data for a fitness tracker might be labelled with “walk”, “run”, and “rest” to describe the activity.
- Data Preparation: is making collected data available for the model training. Data might be separated also into test data (for smoke testing), validation data, and training data, which is typically the largest data set.
- Model Training: performs the training of the machine learning model. It is a process in which one or more machine learning algorithms are fed with training dataset from which it can learn.

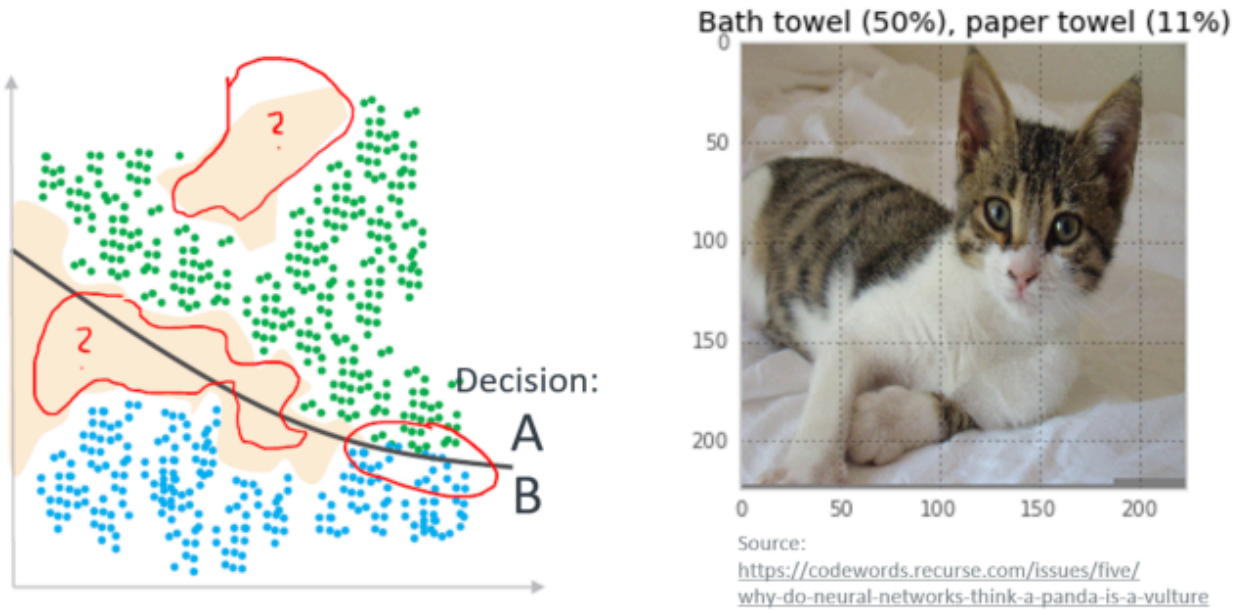
- Model Evaluation: there are several algorithms to choose from, and it is an iterative process to determine the best ML algorithm for any given problem. In this step the best suited algorithm for the problem will be chosen.
- System Validation: is testing of the ML algorithm along with the model data when running on the final target system. The verification of the ML algorithm may be performed using a reduced set of validation data.
- Model Deployment: is the integration of the ML algorithm along with the model data into the final target system, which is in our case an embedded IoT application.

Machine learning models are typically tested and developed in isolated systems. The training of the ML model will mostly take place in the cloud, because both an extensive data set, and high compute power is a prerequisite. The algorithm execution based on the ML model can then take place directly on the IoT endpoint device.

Like humans learn and improve upon past experiences, the machine learning algorithms adaptively improve their performance as the number of samples available for learning increases. Correct decisions can only be made in areas where training data exist. “Learning” means therefore that ML algorithms are retrained based on new data that delivers new “experience”.

For example, if a picture recognition application never “saw” the picture of a cat, it cannot be correctly qualified (see below). It is therefore expected that IoT endpoint systems that incorporate AI and ML technology require periodic updates.

Figure 1-3: Missing Training Data



1.6 Tools and Software Components

The following tools and software components are provided by Arm and support several MLOps process steps.

Step: Data Collection	Description
Keil MDK	For classic embedded IoT software development targeting Cortex-M processors.
SDS Framework	For data capturing, optionally combined with MDK Middleware to interface with Networks, USB, or File System.

Step: Model Evaluation	Description
Arm Compiler for Embedded	Commercial C/C++ Compiler for all Cortex-M processors.
Arm GNU Toolchain	GCC C/C++ Compiler (community supported); not recommended for Cortex-M processors with Helium extension.
Arm LLVM Embedded Toolchain	C/C++ CLang Compiler for all Cortex-M processors (community supported).
CMSIS-NN	Software library of neural network kernels optimized for various Arm Cortex-M processors.
Ethos-U Vela	Compiler for mapping ML models the Ethos-U processors.
Arm Virtual Hardware	Simulation model for estimating inference time on different Cortex-M/Ethos-U target systems.

Step: System Validation	Description
Arm Virtual Hardware	Simulation model for streaming validation data to different Cortex-M/Ethos-U target systems.
SDS Framework	For playback of captured data to Arm Virtual Hardware.

Model Deployment	Description
Open-CMSIS-Pack	Packaging technology and delivery mechanism for software components including ML models.
Keil MDK	For integration of the final ML model into the Cortex-M/Ethos-U target system.
Trusted Firmware	Open source software projects for IoT systems; includes MCU boot for firmware update.

Refer to MLOps Systems for information on how to integrate these tools and software components into an MLOps system.

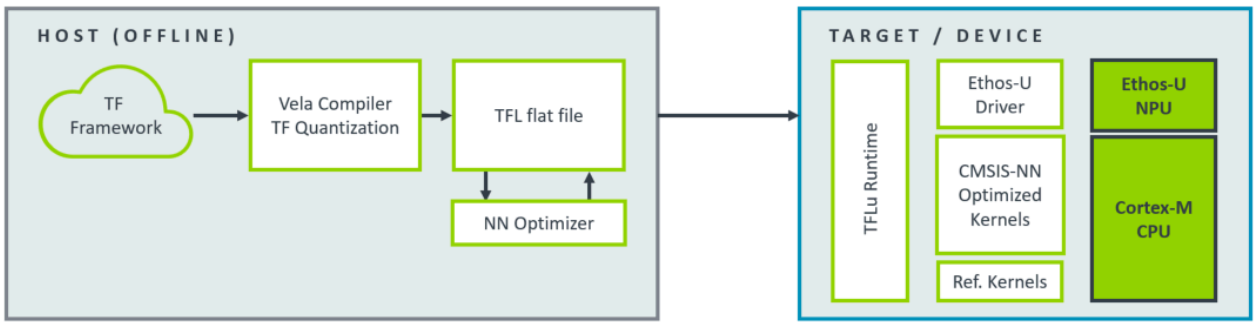
The MLOps development flow delivers the algorithms that are integrated into an Cortex-M based ML application. Typically a library with ML model data is required, that is optimized towards the final target processor.

1.7 Targeting Ethos-U NPU Processors

The Ethos-U Vela compiler uses an ML model as input and generates an optimized binary for the Ethos-U NPU. The picture below outlines the software development flow for ML models utilizing

an Ethos-U NPU. Compared to a single-core Cortex-M processor system the overall changes to the development flow are minimal.

Figure 1-4: Software Development Flow



The Host (Offline) process steps start with a trained ML model using the TensorFlow machine learning framework. Use for ML model conditioning techniques such as collaborative clustering, pruning, and quantization aware training (QAT) to improve model performance on Ethos-U while preserving its accuracy. Use TensorFlow Lite post-training quantization to `int8` data types to speed-up the ML model. The resulting TFL flatbuffer file (`*.tflite` file) is then transformed for execution on Ethos-U NPU using the Vela Compiler. The NN Optimizer identifies graphs to run on Ethos-U and optimizes, schedules, and allocates these graphs. Finally lossless compression reduces size of `*.tflite` file.

The Target / Device uses this `*.tflite` file as ML model for execution with the TFLu runtime system. The Ethos-U driver schedules the operators for execution on Ethos-U, whereas the CMSIS-NN library executes operators that cannot be mapped to Ethos-U using a software implementation on Cortex-M.

See Ethos-U Vela for more information about the Ethos-U Vela compiler.

2. ML software development on Arm Cortex-M Processors

This chapter covers the concept of ML software development on Cortex-M processors, with illustration based on an example in TensorFlow Lite for Microcontollers (TFLM).

2.1 Why run ML applications on Cortex-M processors? And which Cortex-M to use?

Arm Cortex-M processors are widely used in modern microcontroller products. They are used in a wide range of applications; from simple controllers inside toys and home appliances, to sophisticated smart home products, medical devices, and subsystems in automotives/industrial segments. Cortex-M based microcontrollers are widely available and because they are easy to use and are low-cost, it is natural for ML application developers to create applications using these devices.

As mentioned in the last chapter, there are different types of Cortex-M processors, all of which have different levels of ML processing capabilities. An easy way to define the ML performance of the processor is based on the number of OPs (operations) per clock cycle. Neural Network (NN) model processing is often based on MAC (Multiply-Accumulate) operations, with each MAC operation considered as two OPs (Multiply + Add). A rough estimation of the relative ML performance of Cortex-M processors can be established based on the processor's instruction set support and pipeline behaviors. This is as follows:

Processor	Instruction set	OPs at 100MHz
Cortex-M3	A multiply-accumulate instruction (MLA, 2 OPs) takes 2 cycles. Because the processor also needs to execute memory load operations for NN processing (assuming 1:1 ratio of MAC vs load), the average OPs/cycle is 0.6.	0.06 GOPs/sec
Cortex-M4, Cortex-M33	The DSP/SIMD instructions support a dual MAC operation (4 OPs). Because the processor also needs to execute memory load operations for NN processing (assuming 1:1 ratio of MAC vs load), the average OPs/cycle is 2.	0.2 GOPs/sec
Cortex-M7	This processor supports dual issue of DSP/SIMD and memory load, so the average OPs/cycle is 4.	0.4 GOPs/sec
Cortex-M55, Cortex-M85	With Helium technology, these processors can handle 8 MACs/cycle in parallel with data load. As a result, the average OPs/cycle is 16.	1.6 GOPs/sec

The performance can be scaled up with higher clock frequencies in the processors. However, there are other factors to consider:

- Memory wait states can increase with increasing clock speed.
- There are other operations involved in NN model processing.

- At the application level, there are other data processing tasks involved. For example, a Keyword Spotting (KWS) application involves audio data processing tasks. These workloads must be considered when selecting a microcontroller device for a ML application.

A simple real-time KWS application can run in a Cortex-M3 based microcontroller. However, since the DSP/SIMD instructions in Armv7-M and Armv8-M architectures provide much better performance for signal processing, a Cortex-M4 or a more advanced processor is recommended. Armv6-M processors, like Cortex-M0 and Cortex-M0+, can also handle certain levels of ML applications. However, usually these devices have small memory sizes, so it is more challenging to run complex ML applications on them. Examples of running ML applications on the Cortex-M0 / Cortex-M0+ devices are illustrated in the following third-party articles:

- How to Implement Cough Detection on the Cortex-M0
- Qeexo AutoML Shrinks Automated Machines Learning Footprint to Fit Cortex-M0(+)

Additional information on running ML applications on Cortex-M microcontrollers is available in the following whitepaper:

- Machine Learning on Arm Cortex-M Microcontrollers

2.2 ML software framework options

When creating ML applications, one of the first decisions is deciding which ML software framework should be used. Currently, there are several ML software framework options that are available for Cortex-M based microcontrollers. For example:

- TensorFlow Lite for Microcontrollers (TFLM)
- MicroTVM
- PyTorch
- PaddlePaddle

The ML software framework to be used could be dependent on:

- The performance and memory footprint of the ML framework.
- The availability of hardware accelerator support for the targeted device.
- The availability of suitable ML models for the targeted application.
- The ease of development and the integration of ML ops.

In this chapter, the TensorFlow Lite for Microcontrollers (often referred to as TensorFlow Lite Micro or TFLM) is used for illustrative purposes because:

- The NN model weights in TFLM can be quantized to 8-bit integers, which helps reduce the memory footprint. Because memory sizes are often limited in microcontroller devices, this aspect makes TFLM attractive for microcontroller applications.
- The use of 8-bit weights also allows the NN processing to be carried out efficiently in embedded processors such as the Cortex-M processor family. Recent Cortex-M processors like

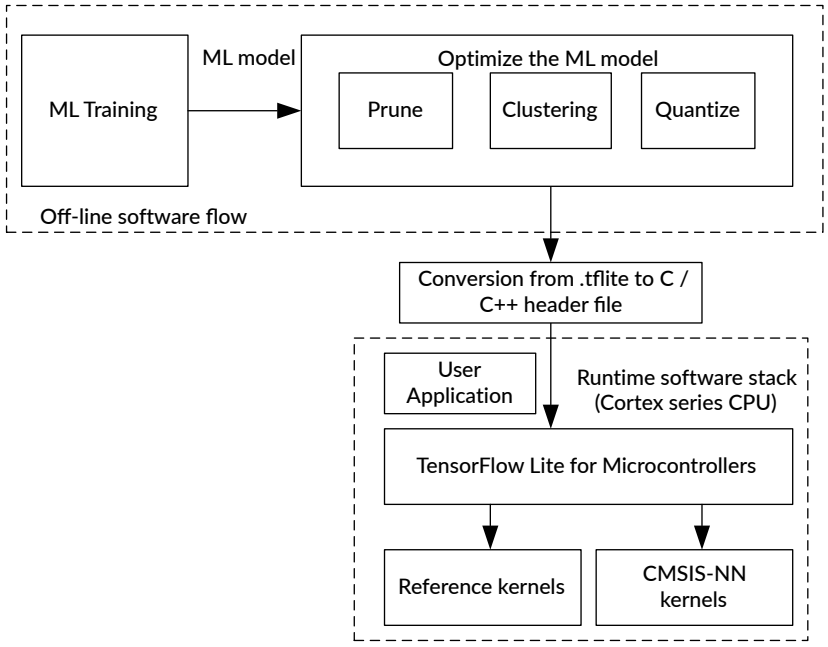
the Cortex-M55 and Cortex-M85 provide 8-bit vector-dot-product operations as part of the Helium technology, which makes these processors highly efficient in handling NN models in TFLM.

- Many ML hardware accelerators like the Ethos-U micro-NPUs are designed to accelerate quantized ML models and have software support for TFLM.
- ML models created for TFLM are widely available.

2.3 TensorFlow Lite for Microcontrollers (TFLM)

TFLM was created by Google. The TFLM runtime library running on the microcontroller is an interpreter that reads the ML model and carries out the required operations. An overview of the software flow is shown below.

Figure 2-1: TFLM software flow overview



In TFLM, when a NN (Neural Network) model is created it is represented as a TensorFlow Lite model file (.tflite, also known as FlatBuffers). The methods for creating a TensorFlow Lite model are listed below:

- Creating a TensorFlow Lite model using the TensorFlow Lite Model Maker. To use this tool to create a model, a data set is required for training the model.
- Using an existing TensorFlow Lite model. TensorFlow.org provides a range of examples. You can also find other models from various model zoos (e.g. The Arm Model Zoo is available at https://github.com/ARM-software/ML-zoo). Please note, not all TensorFlow Lite models can be used on Cortex-M based microcontrollers.

- In some instances you may need to modify an existing TensorFlow Lite model. For example, you can re-train a KWS model to allow it to detect different keywords. To do this, you need a dataset for the training. The retraining of an ML model is often referred to as “Transfer Learning”. Please note, not all ML models can be re-trained using this method.
- In some instances, it is possible to convert a model from one type to another. For example, you can convert a TensorFlow model into a TensorFlow Lite model using the TensorFlow Lite Converter. If you have a model in the ONNX (Open Neural Network Exchange) format (e.g. export from Matlab), you can first convert it into the TensorFlow model (there are many tutorials on the Internet), then convert it into the TensorFlow Lite model.

To enable the NN model to run efficiently on a Cortex-M processor, the runtime library for TFLM supports the integration of CMSIS-NN, a library of optimized NN functions for Cortex-M processors. If the TFLM interpreter encounters a ML operator that is not supported by the CMSIS-NN library, the reference kernel functions would be used instead.

2.4 An overview of the software development flow with TFLM

When creating a ML project with TFLM, there are several steps to undertake:

2.4.1 Preparing a NN model for integration into a C++ project

A developer would, in many instances, use a pre-trained ML model. If a pre-trained NN model from TFLM is being used, the file extension is .tflite. This needs to be converted to a C/C++ header file before integrating into a C++ project.

In this section, the illustrations are based on the micro_speech example application available in the TensorFlow Github repository at this location:

https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech

In this location, a pre-trained model file called micro_speech.tflite can be found. To convert this model to a C header for use in a microcontroller project, the xxd utility in Linux is used:

```
$> xxd -i micro_speech.tflite > model.cc
```

The output file model.cc contains:

```
unsigned char micro_speech_tflite[] = {
  0x20, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x00, 0x00,
  ...
  0x00, 0x00, 0x00, 0x04, 0x03, 0x00, 0x00, 0x00
};
unsigned int micro_speech_tflite_len = 18800;
```

After the conversion, the following modifications are needed:

- The variable names (created from filenames) need to be updated to match the variable name in the application code.
- The “const” keyword should be added so that the data is not copied into the SRAM when the device starts up.

To identify the right variable name, we need to examine the example application code main_functions.cc in the same repository . In this file, the following code fragment provides the right variable name:

```
void setup() {
  tflite::InitializeTarget();
  // Map the model into a usable data structure. This doesn't involve any
  // copying or parsing, it's a very lightweight operation.
  model = tflite::GetModel(g_micro_speech_model_data);
  ...
```

Using this information, we know that the character array name should be `g_micro_speech_model_data`. The modified version of model.cc is as follows:

```
const unsigned char g_micro_speech_model_data[] = {
  0x20, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x00, 0x00,
  ...
  0x00, 0x00, 0x00, 0x04, 0x03, 0x00, 0x00, 0x00
};
const unsigned int g_micro_speech_model_data_len = 18800;
```

The modified model.cc can then be used in the C++ programming environment.

2.4.2 Determining the inputs and outputs of the NN model

When using a NN model created by a 3rd party, you will need to identify information relating to the inputs and output of the NN model. TensorFlow Lite provides a tool called Model Analyzer. This tool can either be used in Google Colab or via a Jupyter Notebook (Note: A .ipynb file is available on the Model Analyzer page).

There are also a number of third-party tools for analyzing and visualizing TensorFlow Lite models.

2.4.3 Preparing the TFLM runtime library

There are two ways to obtain the TFLM runtime library:

- Compiling from the source
- Using CMSIS-PACK

2.4.3.1 Compiling from the source

To compile a TFLM runtime library for generic Cortex-M devices, follow the steps in this page:

https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/cortex_m_generic

When doing this, you can enable CMSIS-NN support by ensuring that the `OPTIMIZED_KERNEL_DIR=cmsis_nn` option is included in the command line.

Optionally, if you are creating a TFLM runtime library for a device containing a Corstone-300 subsystem, the instructions on the following page should be used instead.

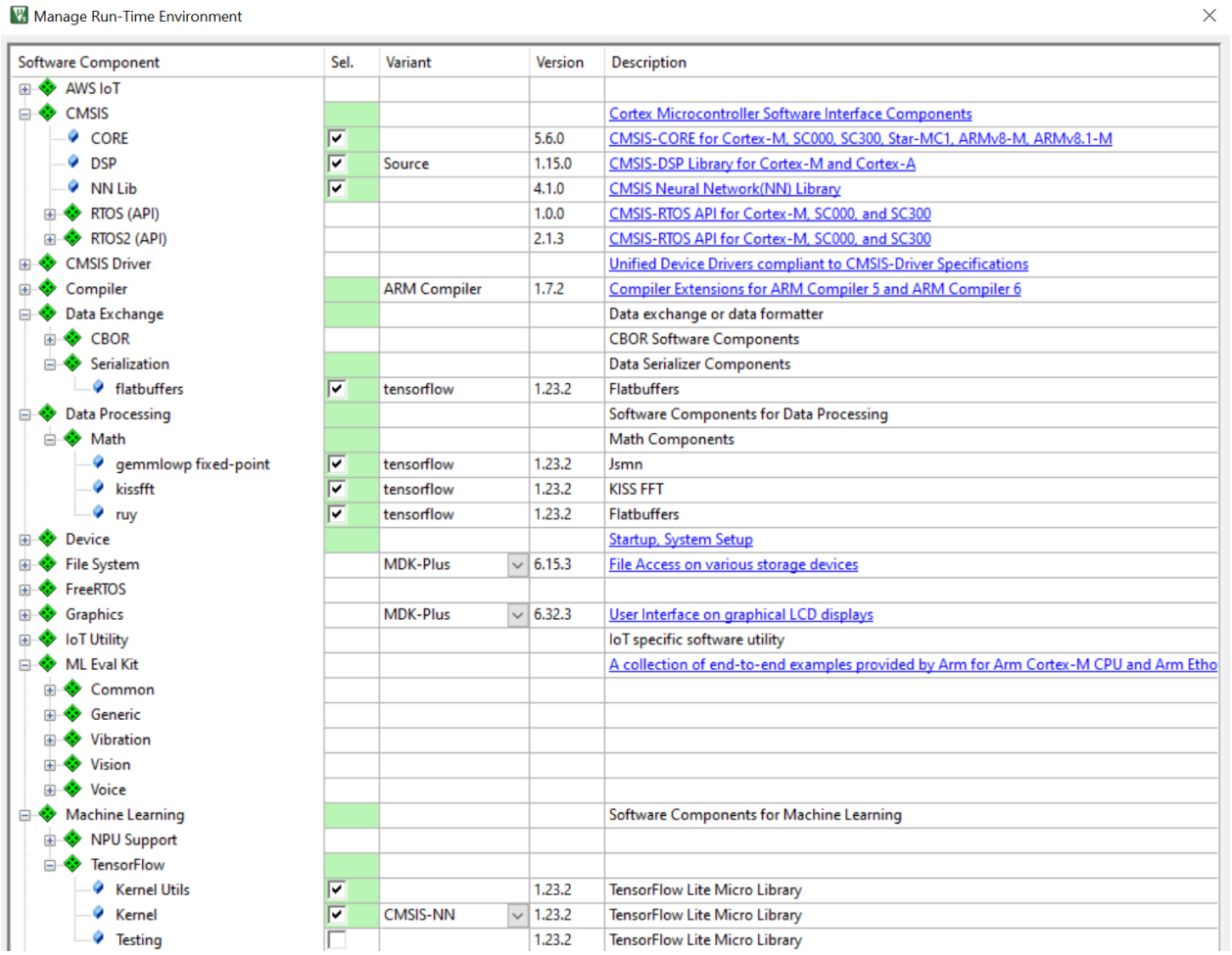
https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/cortex_m_corstone_300

The compilation setup in the Corstone-300 directory enables the use of Ethos-U micro-NPU.

2.4.3.2 Using CMSIS-Pack

When using an IDE that supports CMSIS-Packs, you can integrate the TensorFlow Lite runtime via the CMSIS-Pack mechanism. For example, if you are using Keil MDK, the TFLM software components can be selected as follows:

Figure 2-2: TFLM available in PACK installer



In the TensorFlow Lite kernel, there is an option to use CMSIS-NN as well as Ethos-U. When using CMSIS-NN, the CMSIS-NN library also needs to be selected.

The CMSIS-Pack mechanism allows you to import some of the ML application APIs in the ML Embedded Evaluation Kit. Examples of using TFLM in CMSIS-Pack is available in the following Github repository:

https://github.com/ARM-software/ML-examples/tree/main/cmsis-pack-examples

2.4.4 Integration of the inference functions

There are several key aspects to consider when creating the NN inference application. In this section, the micro-speech example code is used.

2.4.4.1 Creating a TensorArena

To use Tensorflow Lite micro, a memory region in the RAM needs to be assigned. This memory region, called TensorArena, is used to store the inputs, outputs and intermediate values during inferences. The size of this memory is dependent on the NN model. In the micro_speech example mentioned earlier, the size of this memory is 10KB. The corresponding code fragment (in main_functions.cc) for declaring the TensorArena is as follows:

```
// Create an area of memory to use for input, output, and intermediate arrays.
// The size of this will depend on the model you're using, and may need to be
// determined by experimentation.
constexpr int kTensorArenaSize = 10 * 1024;
uint8_t tensor_arena[kTensorArenaSize];
```

In the micro_speech example, the TensorArena memory is allocated to the TFLM interpreter when the following code (in main_functions.cc) is executed:

```
  // Allocate memory from the tensor_arena for the model's tensors.
  TfLiteStatus allocate_status = interpreter->AllocateTensors();
  if (allocate_status != kTfLiteOk) {
    MicroPrintf("AllocateTensors() failed");
    return;
  }
```

2.4.4.2 Input buffer

While TensorArena provides the memory needed for the input data, a pointer for the input data is needed so that the program code can transfer data to it. In addition, applications might need additional data buffer(s) when pre-processing the input data.

In the micro_speech example, the application code main_functions.cc contains the following pointers:

- The “model_input_buffer” which points to the data inside the model_input object.
- The “model_input” which points to the input data inside the TensorArena.

The corresponding code fragments of these pointers are:

```
  TfLiteTensor* model_input = nullptr;
  ...
  int8_t* model_input_buffer = nullptr;
  ...
  model_input = interpreter->input(0);
  ...
  model_input_buffer = model_input->data.int8;
```

The KWS application code uses an additional data array called “feature_buffer” to store the results of the audio processing. During the inference process, the data from the feature_buffer[] array is copied into the model_input_buffer in each iteration of the main loop. The corresponding code fragment in main_functions.cc is as follows:

```
// Copy feature buffer to input tensor
for (int i = 0; i < kFeatureElementCount; i++) {
  model_input_buffer[i] = feature_buffer[i];
}
```

The “feature_buffer” array is declared in the following line in main_functions.cc:

```
int8_t feature_buffer[kFeatureElementCount];
```

The size of the feature buffer (kFeatureElementCount) is defined in micro_model_settings.h in either the micro_features or the simple_features directories. The “kFeatureElementCount” is defined by the following code fragment:

```
// The following values are derived from values used during model training.
// If you change the way you preprocess the input, update all these constants.
constexpr int kFeatureSliceSize = 40;
constexpr int kFeatureSliceCount = 49;
constexpr int kFeatureElementCount = (kFeatureSliceSize * kFeatureSliceCount);
```

The data constants in the above code fragment define the shape of the input data. Further information regarding the input data shape of the micro_speech example is covered in the README.md file in the train directory.

2.4.4.3 Running the inference and processing the result

The inference starts using the Invoke function in the interpreter object. In the micro_speech example, the Invoke function is executed using the following code fragment in main_functions.cc :

```
...
tflite::MicroInterpreter* interpreter = nullptr;
...
// Run the model on the spectrogram input and make sure it succeeds.
TfLiteStatus invoke_status = interpreter->Invoke();
if (invoke_status != kTfLiteOk) {
  MicroPrintf("Invoke failed");
  return;
}
...
```

After the inference operation, the result is stored in the output data in the TensorArena. The values in the output data array are relative probabilities for each voice-command (e.g. Yes, No, Up, Down). Before the result can be used to carry out a response, a post-processing step is needed, and this is handled by the ProcessLatestResults() function in the recognize_commands.cc file.

After post-processing, the application code is then able to use the result to trigger a response(s). This is handled by the RespondToCommand() function in the command_responder.cc file.

2.5 Re-training a ML model

When creating ML applications with an existing trained ML model, there might be a need to partially re-train the model to modify the behavior of the NN (Neural Network). For the micro speech example, the ML model can be re-trained to recognize different keywords. The micro speech example provides information on how-to re-train the reference model. This information is available in the following page:

https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech/train

The re-training process requires a speech command dataset. A reference dataset was released by Google research and this is in the following link:

http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz (version 2)

Information regarding version 1 of this dataset is available on this blog.

Version 1 (v0.01) of the command dataset provides audio samples of the following words: Yes, No, Up, Down, Left, Right, On, Off, Stop, Go.

Version 2 (v0.02) added additional words to the dataset. The full list is available in the following paper: https://arxiv.org/abs/1804.03209 / https://arxiv.org/pdf/1804.03209.pdf (pdf version)

2.6 Further Information

A few links that provide useful information are listed below.

2.6.1 Additional TFLM examples

Other than the micro speech example, additional examples can be found in the Tensorflow Lite Micro Github repository

2.6.2 Further information about the TensorFlow Lite for microcontroller

Further information about the operation of the TensorFlow Lite for Microcontrollers is available in the following page: https://www.tensorflow.org/lite/microcontrollers/get_started_low_level

2.6.3 Using mbed CLI

The micro speech example for the Arm development boards uses mbed tools. Further information about installing mbed CLI (command line interface) is available from the link below: https://

os.mbed.com/docs/mbed-os/v6.16/mbed-os-pelion/machine-learning-with-tensorflow-and-mbed-os.html

3. Arm Ethos-U NPU

The majority of Machine Learning (ML) applications contain Neural Network (NN) inference operations. While it is possible to handle this in software running on a processor, we can increase the performance significantly if a hardware accelerator is used. The use of a hardware accelerator in NN inference usually improves the energy efficiency and allows a higher processor bandwidth to handle other tasks. There are many types of NN inference hardware accelerators: E.g. Arm® Ethos™-U series - a family of hardware accelerators designed for microcontrollers / System-on-Chips (SoC) which are known as Neural Processing Units (NPUs).

The Ethos-U NPU is a small and power-efficient processor that is used to reduce both the inference time and memory requirements needed to run Machine Learning (ML) Neural Networks (NN).

Currently the Ethos-U family contains two designs:

- Ethos-U55
- Ethos-U65

These two NPUs are already available in commercial products. For example:

- Ethos-U55 is used in the Alif Ensemble™ family from Alif Semiconductor™.
- Ethos-U65 is used in the i.MX 93 family from NXP®.

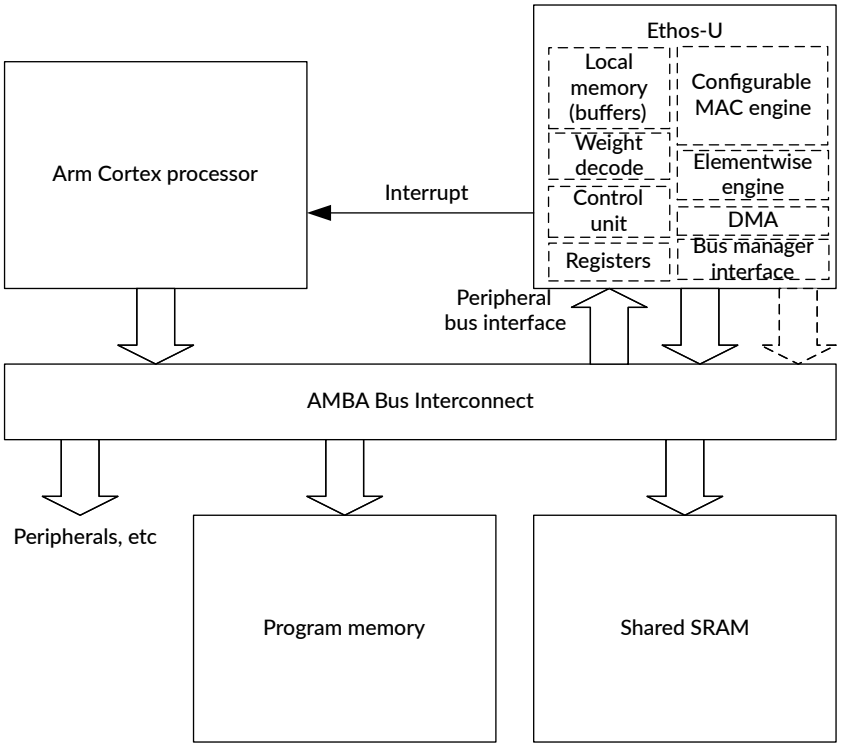
You can try out Ethos-U55/U65 without using real hardware. For example, you can use a simulated environment such as Arm Virtual Hardware (AVH) or Fixed Virtual Platform (FVP). These tools are described in other parts of this Getting Started Guide.

3.1 Ethos-U Hardware Concept

A Ethos-U NPU requires a host processor to control its operations. Typically, an Arm Cortex-M processor is used. The Ethos-U NPU has a range of programmable registers and several hardware interfaces:

- A peripheral bus interface to allow a processor to access the internal registers.
- One or two AMBA® bus manager interfaces to access the system memories.
- An interrupt output for sending an interrupt event to the host processor.

Figure 3-1: Concept of a microcontroller system with a Ethos-U



The majority of the processing in an NN inference is based on Multiply-Accumulate (MAC) computations. Inside the Ethos-U NPU there is a configurable MAC engine to handle MAC operations. The number of MACs that can be carried out per clock cycle is configurable by chip designers, ranging from 32 to 256 for the Ethos-U55 and either 256 or 512 for the Ethos-U65. There is also an element-wise data processing engine for other computations. To enhance the efficiency of NN inferences, the Ethos-U NPUs have a local memory inside for buffering the data they process. The Ethos-U NPUs also include a Direct Memory Access (DMA) engine so that data can be copied, before it is needed, from the shared memory to the local memory.

Because Ethos-U NPUs are designed for embedded systems that, in many instances, have a limited amount of memory, Ethos-U NPUs support NN weight data compression. The weight data is decoded on-the-fly during inference operations.

The operations of the Ethos-U are controlled by a number of registers which are memory mapped on the processor system. When the system needs to carry out a NN inference, the operations are broken down into a number of smaller jobs that can be carried out by the Ethos-U NPU. Under the control of software libraries, the jobs are issued to the Ethos-U NPU via the peripheral bus interface. Each time a job is completed the Ethos-U NPU issues an interrupt request to the processor so that the software library can then issue the next job.

In addition to the aforementioned key interfaces, the Ethos-U NPUs support additional interfaces, e.g. an interface for power management. More detailed information for the Ethos-U NPUs can be found in the:

- Ethos-U55 Technical Reference Manual
- Ethos-U65 Technical Reference Manual

There are a large number of Neural Network models available and in a number of instances some of those network models contain operators that are not supported by the Ethos-U hardware. In such cases, the unsupported operators will need to be handled by the software running on the processor. The ML operators supported in the Ethos-U55 and Ethos-U65 are documented in the following links:

- Ethos-U55: Supported data types and operators
- Ethos-U65: Supported data types and operators

3.2 ML Software Support for Ethos-U

A ML software project is usually based on a specific ML software framework. The Ethos-U NPUs support the TensorFlow Lite Micro (TFLM), a popular ML software framework which is optimized for microcontrollers, as well as MicroTVM. Because the weights in the NN models in TFLM are quantized to 8-bit integers, the memory footprint of the NN models is reduced. Another advantage of using quantized NN models is that generally these models perform very well on hardware with support for vector-dot-product operations (e.g. Ethos-U, as well as a range of modern Arm Cortex processors).

The development of a ML application can be divided into two parts:

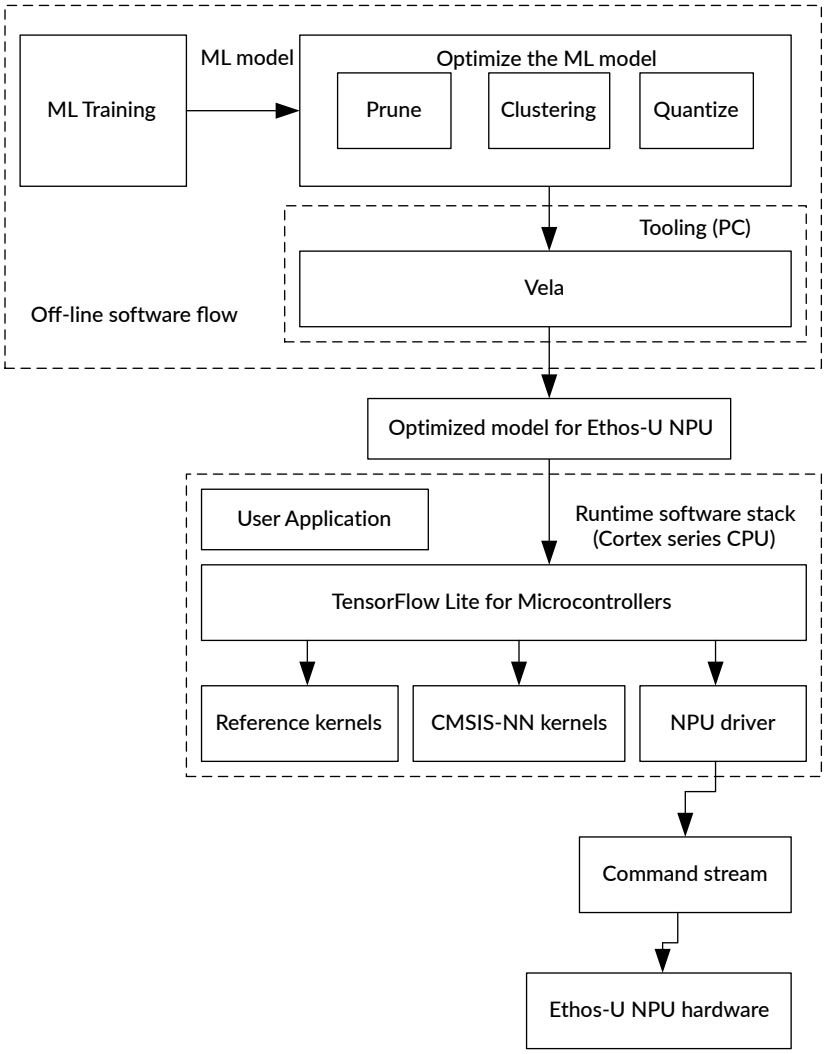
- Off-line development flow. This consists of:
  - The preparation of the ML model
  - Quantizing the ML model to use 8-bit weight data (TF Quantization tooling - TOCO)
  - Optimizing the ML model using a tool call Vela. This tool identifies operators inside the model that can be handled by the Ethos-U and replaces them with Ethos-U functions. If an ML operator is not supported by Ethos-U but is supported by an optimized function in the CMSIS-NN library, CMSIS-NN would be used instead. The Vela compiler also handles memory layout optimizations. See Ethos-U Vela for more information. The Vela compiler runs on a desktop PC or similar device.
- The NPU runtime software stack. This consists of a range of software components running on the target hardware and interact with each other in specific ways. These include:
  - User Application
    - The User Application runs the required functions and makes calls to the TensorFlow Lite for Microcontrollers (TFLμ) library when it performs an inference of the model.
  - The TensorFlow Lite for Microcontroller (TFLμ)
    - The TFLμ framework is compiled into a C++ library that contains a copy of the optimized model from Vela (i.e. the modified .tflite file) along with versions of Reference and CMSIS-NN kernels. This library is then used by the User Application to perform inferences.

During an inference, the model is parsed one operator at a time and the corresponding kernels are executed. The exception to this is when it encounters a TensorFlow Lite Custom operator. In this case, the library sends the operator and associated tensor data to the Ethos-U NPU driver instead.

- The Ethos-U NPU driver which controls the Ethos-U NPU
  - The Ethos-U NPU driver handles the communication between the TFLμ framework and the Ethos-U NPU to process a custom operator. When the Ethos-U NPU has completed its processing, it signals back to the driver, which in turn informs the TFLμ library.
- The CMSIS-NN library
  - This contains highly optimized and performant kernels that accelerate a subset of operators in the TFLμ framework. It is needed to handle the ML operators that are not supported by Ethos-U.
- Reference Kernels
  - This contains a set of kernels for all operators in the TFLμ framework. They are used when the TFLμ framework encountered ML operators that are not supported by the Ethos-U or the CMSIS-NN library.

An overview of the software development flow is illustrated in the diagram below.

Figure 3-2: Software development concept



3.3 Ethos-U Custom Operators

TensorFlow Lite for microcontrollers (TFLμ) is an interpreter. It reads a NN model (in form of .tflite in the memory) and carries out the functions of the NN operators. In order to allow NN model processing to be accelerated by the Ethos-U NPU, TFLμ supports a feature called custom operators.

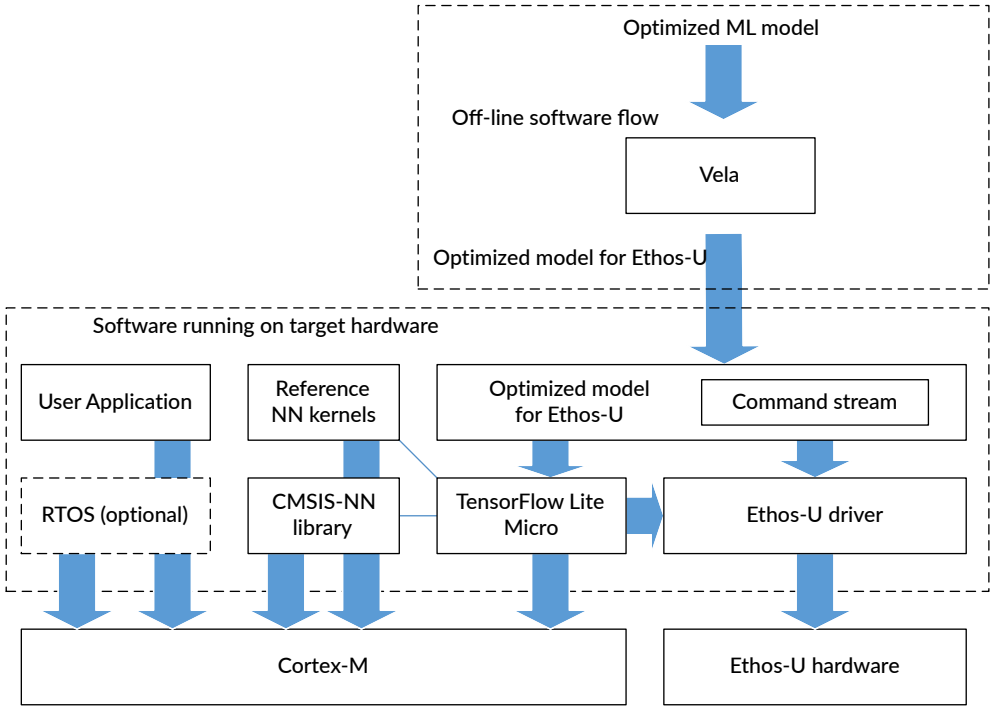
During the compilation of a NN model, the Vela compiler groups the sequence of operators that can be accelerated by the uNPU into a custom Ethos-U operator. The Ethos-U custom operator has 5 input tensors – tensor for command stream, a flash tensor for constant Read-Only data such as weights and biases, scratch tensor for the tensor arena, scratch fast tensor for the spilling feature of the Ethos-U65(scratch fast tensor is not used for the Ethos-U55) and a tensor for the inputs of the model. From the TF Lite Micro application code, you need to define an

interpreter specifying the model, the operator resolver, the tensor arena, and its size and pass these parameters to TF Lite Micro. During the inference TensorFlow Lite Micro reads the Ethos-U custom operator and executes it on the Ethos-U microNPU. You can read more about the Ethos-U custom operator here.

3.4 ML software for microcontrollers with Cortex-M and Ethos-U NPU

In a Cortex-M based microcontrollers with Ethos-U NPU, the aforementioned runtime software stack provides the software required to support the Ethos-U NPU. This includes the User Application, which uses the TFLμ library to execute parts of the optimized model (command stream) on the Ethos-U NPU. Based on the application requirements, addition software components like an RTOS might be needed.

Figure 3-3: MCU software integration

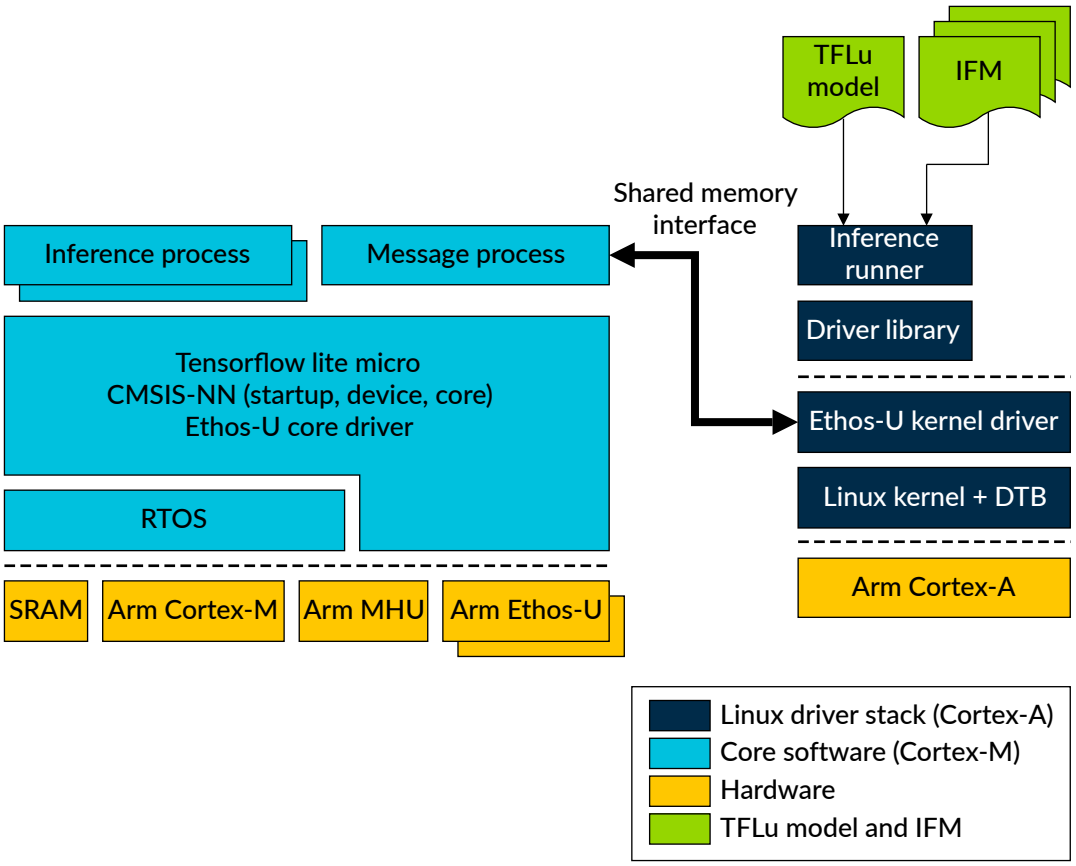


3.5 ML software for ML subsystems in a larger SoC

In the scenario where the Ethos-U is used in a ML subsystem within an SoC with Cortex-A processors, the applications running in the Linux environment communicates with the ML subsystem using a Ethos-U kernel driver. This driver then communicates with the Cortex-M processor via a Message Handling Unit (MHU). After receiving the information, the Cortex-M processor controls the Ethos-U NPU that carries out the inference.

The Linux driver stack is provided as an example of how a rich operating system like Linux can dispatch inferences to an Ethos-U subsystem. The source code is provided. In accordance with the license, you can modify and further develop the source code.

Figure 3-4: ML subsystem software stack concept



The software components contains:

- Inference runner
  - The inference runner is a test application that runs inferences on the Ethos-U driver stack. The inference runner inputs a TFLμ model that was optimized by Vela and an input file containing input feature map (IFM) data. The output from the inference runner is an output feature map (OFM) file.
- Driver library
  - The driver library is a thin C++ interface around the kernel-user API (UAPI) header file that the kernel driver exports to user space. The driver library enables user space applications to detect NPU capabilities, create buffers, register networks, and run inferences.
- Kernel driver
  - The kernel driver is the bridge between user space and the Ethos-U subsystem. It presents a UAPI that allows a user space application to run inferences. The inference request from user space is forwarded to the Ethos-U subsystem that runs the inference.

- Linux kernel and DTB
  - Any vanilla Linux kernel can be used. The Debug and Trace Bus (DTB) entry for the Ethos-U subsystem is documented in the kernel driver.

3.5.1 Software Architecture Scenarios and use cases

The software architecture for Ethos-U subsystem can be categorized in two major scenarios:

Linux dispatches inferences

1. Linux allocates DRAM memory for the network, input feature map (IFM), and the output feature map (OFM).
2. An inference request is sent from Linux to the Ethos-U subsystem.
3. The Ethos-U subsystem executes inference and returns an inference response. This use case is implemented and has been verified.

Ethos™-U running without Linux

1. The Ethos-U subsystem is capturing IFMs and running inferences without the help of Linux. Linux is busy, in sleep mode, or even powered down.
2. The Ethos-U subsystem captures an IFM (audio, video, or sensor data) and runs inference.
3. When the Ethos-U subsystem detects something of interest, Linux is notified.

A possible use for this use case would be an AI speaker scanning audio for a particular word or a camera scanning faces to trigger an unlock event.

This use case is not implemented in the Linux driver stack. You would have to implement this use case.

3.6 Additional software/tools for Ethos-U

An additional experimental tool called ML Inference Advisor (MLIA) is available to help developers analyze and optimize NN models on a range of Arm based hardware targets such as Ethos-U and Cortex-A processors. For more information about MLIA, see MLIA Usage.

For ML developers that are using PyTorch, there are 3rd party tools available that can convert PyTorch models to TensorFlow Lite. For example, TinyNeuralNetwork from Alibaba. Additional information about converting PyTorch to TensorFlow Lite via ONNX (Open Neural Network Exchange) is available in this blog and this GitHub page.

3.7 Ethos-U Hardware Description

This section covers more details on the Ethos-U NPU hardware.

3.7.1 Ethos-U configurations

The Ethos-U55 NPU and Ethos-U65 NPU are configurable to meet various performance points as outlined in the following tables:

Ethos-U55 NPU configuration options:

Configuration	Number of MACs per cycle	Internal memory	Performance@500MHz
256	256	48KB	256 GOP
128	128	24KB	128 GOP
64	64	16KB	64 GOP
32	32	16KB	32 GOP

Ethos-U65 NPU configuration options:

Configuration	Number of MACs per cycle	Internal memory	Performance@1GHz
512	512	96KB	1 TOP
256	256	48KB	512 GOP

3.7.2 Bus manager interfaces

The Ethos-U55 and Ethos-U65 each have two AMBA 5 AXI interfaces for connecting to the memory system, named M0 and M1:

- To optimize performance of the Ethos-U NPU, the AXI interface M0 should be connected to a high-speed, low-latency memory, such as SRAM. The memory is used for dynamic storage of runtime data during the inference of the neural network.
- The AXI interface M1 is used for memory transactions that tolerate lower bandwidth and higher latency. The AXI M1 interface can therefore be connected to memory that is slower or less burst efficient, for example flash or DRAM. The memory is used for the non-volatile storage of the runtime software stack (including the User Application) and the neural network definition (including weights).
- For the Ethos-U55 NPU, the AXI interface M1 is read-only. For the Ethos-U65 NPU, the AXI interface M1 is read/write.

The M0 and M1 ports typically connect to an interconnect, which allows the M0 and M1 AXI interfaces to access any memory. The Vela compiler schedules high bandwidth, low-latency memory transactions on the AXI interface M0, and all other transactions on the AXI interface M1.

3.7.3 Ethos-U system integration

There are two common types of system arrangements when an Ethos-U NPU is integrated into a microcontroller / SoC. These are:

- An Ethos-U NPU controlled by a Cortex-M processor, which also runs the application codes. You also have access to the source code requires for driving the Ethos-U meaning you can use the Cortex-M series CPU for tasks other than machine learning.
- An Ethos-U NPU integrated into an ML subsystem together with a Cortex-M processor. In this scenario, the ML subsystem is part of a larger SoC with one or more Cortex-A processors. The Cortex-A processor would dispatch the NN inference workloads to the Cortex-M processor in the ML subsystem, with the Cortex-M processor managing the low level control.

In both cases, the system designer(s) must ensure that the Ethos-U NPU is able to:

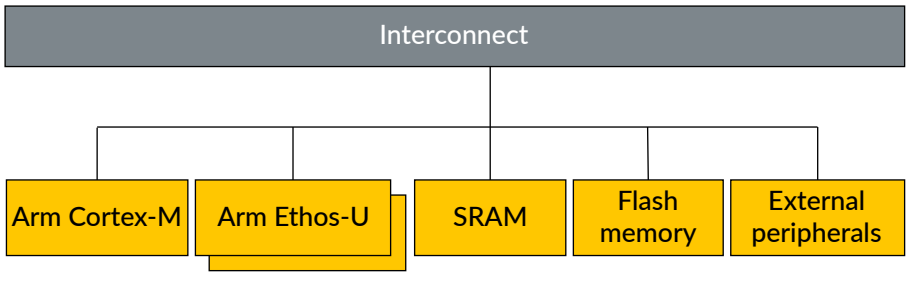
- access to the memory system via its bus manager interfaces.
- accept bus transactions from the host processor via its peripheral bus interface. The Ethos-U NPU has an AMBA 4 APB interface that provides access to its registers. The base address of the registers block is system specific, and is normally located a peripheral address range.
- generate interrupt request to the processor via its interrupt output. This is connected to the interrupt controller for the host processor, for example, the NVIC interrupts on a Cortex-M processor

In addition, system designers must also connect the power management interface and security management interface appropriately. For example, if the Ethos-U microNPU is integrated in a system with TrustZone security extension, the hardware configuration can allow the Ethos-U NPU to be restricted to be used by the secure firmware only, or to be available to the applications running in the Non-secure world.

3.7.3.1 Ethos-U integration in a Cortex-M system

The Ethos-U system is paired with a Cortex-M CPU. The system is highly configurable and can be built in many different ways. The following figure shows a typical Ethos-U system.

Figure 3-5: Ethos-U system



The system contains:

- Cortex-M

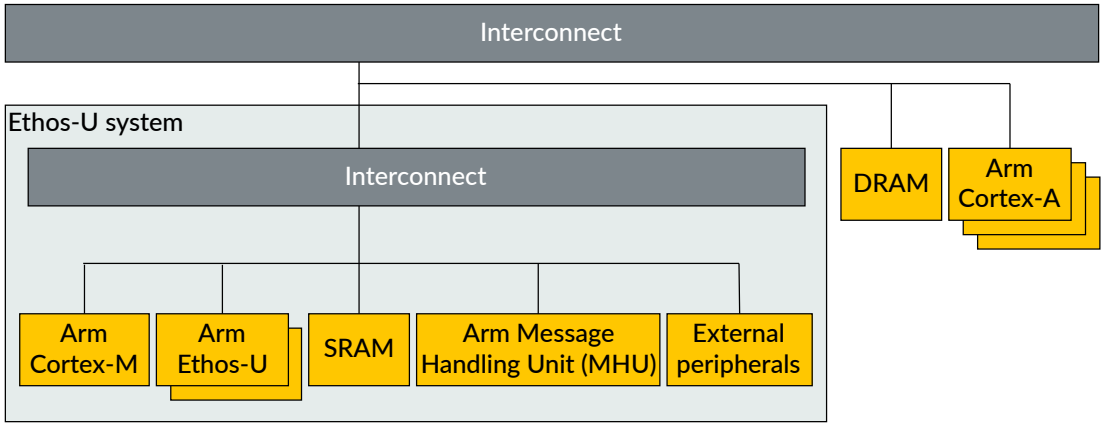
    - The Cortex-M series CPU is the application processor that controls one, or multiple Ethos-U NPUs. You can specify your preferred Cortex-M series CPU, but the recommended CPUs are:
        - Cortex-M4
        - Cortex-M7
        - Cortex-M33
        - Cortex-M55
        - Cortex-M85
- Ethos-N NPU
    - Either an Ethos-U55 NPU or an Ethos-U65 NPU can be paired with the Cortex-M CPU, but the Ethos-U65 NPU has been designed to optimize data transfer between the slower memory and the fast memory cache.
- SRAM
    - The input feature map (IFM) data and the output feature map (OFM) data is stored in SRAM. You can specify your preferred amount of SRAM, but optimal performance is obtained when the network is placed fully in SRAM. If the network cannot be placed fully in SRAM, only the temporary data is stored in SRAM.
- Flash memory
    - The weights and biases are stored in flash memory, DRAM, or SRAM.
- External peripherals
    - Controllers for external peripherals, such as a microphone or camera, can be added.

3.7.3.2 Ethos-U subsystem

The Ethos-U subsystem can connect to a Linux host and various other operating systems.

The following figure shows a typical Ethos-U subsystem.

Figure 3-6: Ethos-U subsystem



The system contains:

- Ethos-U NPU
  - Either an Ethos-U55 NPU or an Ethos-U65 NPU can be paired with the Cortex -M CPU. The Ethos-U65 NPU has been designed to optimize data transfer between the slower memory and the fast memory cache and is therefore the recommended NPU.
- Message Handling Unit
  - Any type of mailbox, similar to the Arm Message Handling Unit (MHU), can be used.
  - NOTE: For an example of using an MHU, see the Arm CoreLink SSE-200 Subsystem for Embedded Technical Reference Manual.
- DRAM
  - Weights, biases, and the input feature map (IFM) and the output feature map (OFM) data are stored in slower, high latency memory like DRAM.
- Cortex-A processor(s)
  - The Cortex-A series CPU only communicates with the Cortex-M series CPU. The Cortex-A series CPU has no direct contact with the Ethos-U NPU. Communication between the CPUs is based on a memory interface in DRAM and the MHU doorbell.

3.7.4 Differences between Ethos-U55 and Ethos-U65

There are several key differences between the Ethos-U55 and the Ethos-U65:

Feature	Ethos-U55	Ethos-U65
Number of MACs/ cycle	32/64/128/256	256 or 512
Manager Bus interface	Two 64-bit AXI supporting on-chip SRAM and embedded flash	Two 128-bit AXI supporting on-chip SRAM, DRAM and flash

Feature	Ethos-U55	Ethos-U65
Host CPU support	Cortex-M85, Cortex-M55, Cortex-M7, Cortex-M4, Cortex-M33	Cortex-M85, Cortex-M55, Cortex-M7

Because Ethos-U65 has a wider bus interface(s) and additional hardware resources, in average it provides around 50% higher performance than the Ethos-U55. (Information source here).

For systems with DRAM/DDR, such as Cortex-A systems running Linux, the Ethos-U65 is more suitable because the bus interface is designed to support memories with longer latency.

3.7.5 Additional features

To enable Ethos-U NPUs to be used in a wide range of systems, a range of additional features are provided:

- Power management interface: Ethos-U55 and Ethos-U65 provide Q-channels (see AMBA 4 Low Power Interface Specification) for the management of clock and power gating. This interface connects to system level power management hardware (e.g. power control build on Arm CoreLink PCK-600 Power Control Kit).
- Security management: If an Ethos-U NPU is used in a TrustZone enabled system, software running in the Secure state can restrict the access permission from the Non-secure world. Privileged software can also control whether the Ethos-U NPU is privileged access only or can be accessed from both privileged and unprivileged software. Two hardware signals are available to define the access permission when the NPU comes out of reset. Contents of registers inside the NPU are also cleared at reset to prevent data leakage.
- Performance Monitoring Unit (PMU): The PMU supports a 48-bit cycle counter and four 32-bit event counters which can be used to measure activities inside the NPU. This allows software developers to analyze the characteristics of the NN workload and identify potential performance issues.

3.7.6 Corstone-300 and Corstone-310 reference designs

Instead of building new systems from scratch, Arm provides reference system designs in Arm Corstone subsystem products to help system designers to create Cortex-M based systems. The Arm Corstone-300 and Arm Corstone-310 reference designs provide examples of building a secure System-on-Chip featuring a Cortex-M and Ethos-U NPU.

- Corstone-300 provides a reference system design for the Cortex-M55 processor
- Corstone-310 provides a reference system design for the Cortex-M85 processor

To help software developers to test their software, simulation models of Corstone-300 and Corstone-310 are available as Fixed Virtual Platforms (FVPs), and these models are available on the Arm website.

These FVPs included simulation models of the Cortex-M55/M85 processor, Ethos-U55 and Ethos-U65 microNPUs, and allows software developers to test their ML software easily. These simulation

models are not cycle accurate, especially for the timing model of the processors in these models. However, they are able to provide a good guidance of the processing time required on the Ethos-U. For example, the Corstone-300 FVP provides roughly 90% accuracy for the Ethos-U55 cycle timing. Please note that the accuracy of the Ethos-U65 in the FVP model is lower than Ethos-U55.

When simulating a NN inference workload with FVP, and if there are operators that are not supported by the Ethos-U NPU, then the operator processing falls back to the processor. In these cases, you cannot rely on the FVP models to provide you with an accuracy estimation of processing time. If cycle timing accuracy is required, you can chose to use other methods for evaluation such as:

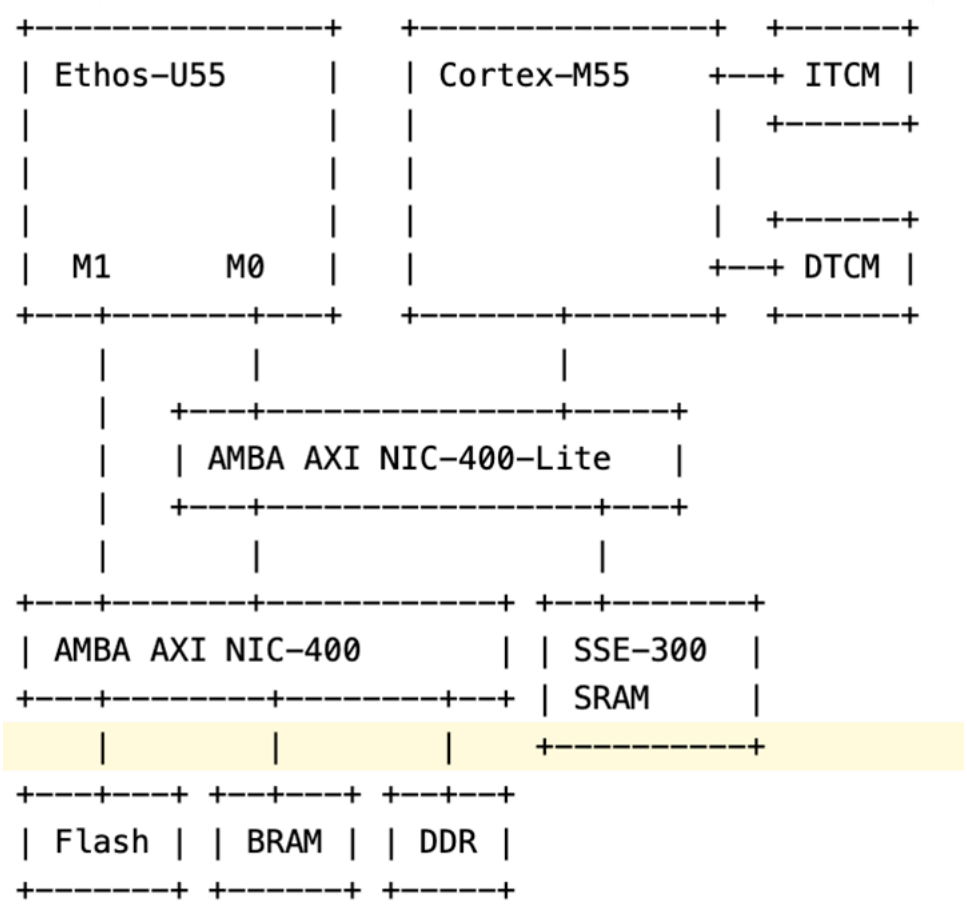
- FPGA platform such as Arm MPS3 FPGA board,
- Arm IP explorer, or
- Other hardware.

Arm provides the following FPGA images for the Arm MPS3 FPGA board:

- AN552 – for Corstone-300 with Cortex-M55 processor and Ethos-U55 microNPU.
- AN555 – for Corstone-310 with Cortex-M85 processor and Ethos-U55 microNPU.

A simplified illustration of the system integration is as follows:

Figure 3-7: Simplified illustration of the AN552 system integration



Above, a simplified picture of the Corstone-300 memory system. The Ethos-U has access to the SRAM, Flash, BRAM (FPGA block RAM) and DDR memory. Therefore, on the design above you cannot store the tensor arena in the D-TCM or I-TCM and access it from the Ethos-U.

The advantage of running applications on real hardware (e.g. the MPS3 board) is that you get cycle accurate performance figures for both the CPU and the NPU. The Corstone-300 and Corstone-310 designs serve as valuable examples for integrating an Ethos-U to a SoC – we recommend you have a read of the technical reference manual of the reference design before starting your own design.

3.8 Porting Ethos-U software to a new hardware platform

This section describes the key points for software porting.

3.8.1 Software porting key points

To use the Ethos-U hardware, the software needs to have correct information on:

- The base address of the Ethos-U microNPU register block.
- The interrupt assignment of the Ethos-U microNPU.
- The address of the shared memory used by the Ethos-U microNPU.

In addition, for systems with TrustZone security extension:

- Hardware designers need to connect the security configuration signals (PORPL and PORSL) to determine the security level of the microNPU after a hard reset.
- Secure firmware needs to configure the system to enable the Ethos-U microNPU to be used in the correct security domain.

Finally, the software must also

- invoke an initialization function for the Ethos-U microNPU before using it. The initialization function (ethos_init()) is a part of the Ethos-U driver.
- Provide an interrupt handler to call the Ethos-U interrupt handler function in the Ethos-U driver.

Similar to any other embedded software projects, the software developers must also prepare linker script / scatter file to allow the toolchain to generate program image that match the system memory map of the hardware platform.

3.8.2 Security configuration for Ethos-U in a TrustZone system

If an Ethos-U microNPU is being used in a TrustZone enabled system, the secure firmware needs to:

- Configure the memory map and/or the TrustZone peripheral protection controller so that the Ethos-U register block is accessible in the correct security address range.
- Configure the memory map and/or the TrustZone memory protection controller so that the shared memory used by the Ethos-U microNPU is accessible to the microNPU.
- Configure the PROT register in the Ethos-U microNPU to configure the current security and privileged level.
- Configure the security domain of the Ethos-U interrupt in the interrupt controller. In Cortex-M processors, this setting is handled by the Interrupt Target Non-secure (ITNS) register.

It is possible to change the security state of the Ethos-U microNPU when the system is ON. This requires a soft reset. Please see the following pages in the Technical Reference Manual:

- Ethos-U55 Boot flow information page
- Ethos-U65 Boot flow information page

3.8.3 An example of Ethos-U hardware initialization

An example of Ethos-U microNPU initialization can be found in the ML Embedded Evaluation Kit.

The example initialization function is in ethosu_npu_init.c

Inside this file, the Ethos-U initialization function is called by `arm_ethosu_npu_init()` in the following code:

```
int arm_ethosu_npu_init(void)
{
int err = 0;
/* Initialise the IRQ */
arm_ethosu_npu_irq_init();
/* Initialise Ethos-U device */
void* const ethosu_base_address = (void *)(ETHOS_U_BASE_ADDR);
if (0 != (err = ethosu_init(
  &ethosu_drv, /* Ethos-U driver device pointer */
  ethosu_base_address, /* Ethos-U NPU's base address. */
  get_cache_arena(), /* Pointer to fast mem area - NULL for U55. */
  get_cache_arena_size(), /* Fast mem region size. */
  ETHOS_U_SEC_ENABLED, /* Security enable. */
  ETHOS_U_PRIV_ENABLED))) /* Privilege enable. */
{
  printf_err("failed to initialise Ethos-U device\n");
  return err;
}
...
  return 0;
}
```

In the code above, the base address of the register block is specified by ETHOS_U_BASE_ADDR. When using a device support package that is compatible with the CMSIS-CORE standard, this is usually defined in the device's header file.

This initialization function calls an interrupt initialization function `arm_ethos_npu_irq_init()`.

```
static void arm_ethosu_npu_irq_init(void)
{
    const IRQn_Type ethosu_irqnum = (IRQn_Type)ETHOS_U_IRQN;
    /* Register the EthosU IRQ handler in our vector table.
     * Note, this handler comes from the EthosU driver */
    NVIC_SetVector(ethosu_irqnum, (uint32_t)arm_ethosu_npu_irq_handler);
    /* Enable the IRQ */
    NVIC_EnableIRQ(ethosu_irqnum);
    debug("EthosU IRQ#: %u, Handler: 0x%p\n",
          ethosu_irqnum, arm_ethosu_npu_irq_handler);
}
```

In the code above, the IRQ number of the Ethos-U is specified by ETHOS_U_IRQN. When using a device support package that is compatible with the CMSIS-CORE standard, this is usually defined in the device's header file.

The example code setup the exception vector as part of the initialization – this is not required if the exception vector is stored in a non-volatile memory and the exception vector is already defined there.

Optionally, software developer can setup the priority level of the interrupt in the NVIC. When using a device support package that is compatible with the CMSIS-CORE standard, the `NVIC_SetPriority` function can be used. See this page for the list of NVIC management functions in the CMSIS-CORE.

The example code provides the following wrapper function for the interrupt handler:

```
void arm_ethosu_npu_irq_handler(void)
{
    /* Call the default interrupt handler from the NPU driver */
    ethosu_irq_handler(&ethosu_drv);
}
```

This enables the interrupt handler function in the Ethos-U driver to be called.

3.8.4 Software integrating for Ethos-U micro NPU in custom designs

The ethos-u-core-platform project is one of the recommendation starting point for software developers to create software packages for custom hardware targets with Ethos-U NPUs. This project provides a way to produce a firmware binary for a defined target. Inside the `targets` directory, you can see examples of how the support for Corstone-300 and Corstone-310 reference design targets are added.

To define your custom target, go to `targets/demo` and edit the `CMakeLists.txt` and `target.cpp` base on your design (search for "ToDo" to find the code you need to edit). In `target.cpp`, you have to specify the base address for your Ethos-U as well as the interrupt number relative to the Ethos-U interrupt from the Vector Interrupt Table of the Cortex-M.

You also need to provide either

- linker script (if compiling with GCC), or
- scatter file (if compiling with Arm Compiler)

for the memory map of your target. FPGA venders usually provide automated tools for generating linker scripts for a given target – example from Xilinx. The `targets/corstone-300/platform.ld` linker script is targeted for the MPS3 FPGA board loaded with the Corstone-300 reference design. The linker script defines two load regions – rom_exec and rom_dram corresponding to loading the application in the I-TCM and in the DDR. When you deploy an application, the boot loader copies the two binaries to their respective physical address in memory. The CPU reset is lifted and the entry point for the CMSIS runtime is called. GCC will scatter load data listed in the copy table, in our case data from DDR to BRAM. The copy of data happens in the `ethos-u/core_software/cmsis/CMSIS/Core/Include/cmsis_gcc.h`, function `__cmsis_start`. Then, constructors and `main()` functions are called. You need to create a linker script specific to your custom target.

3.8.5 Linker script design - placement of the Tensor Arena and model weights in the memory

Using the baremetal examples in the core-platform project, let us examine how you place the tensor arena and the model in memory from the embedded code. The tensor arena is defined in `ethos-u-core-platform/applications/baremetal/main.cpp` with the following definition:

```
__attribute__((section(".bss.tensor_arena"), aligned(16))) uint8_t
 TFLuTensorArena[tensorArenaSize];
```

This places the tensor arena array in the .bss.tensor_arena section of your memory map. Inside the scatter file and linker scripts, you can see that the .bss.tensor_arena symbol is placed in the SRAM or in the DRAM depending on whether we compile the application for Ethos-U55 or Ethos-U65.

Figure 3-8: Example placement of Ethos-U arena in a linker script for GCC

```
LOAD_REGION_SRAM SRAM_START SRAM_SIZE
{
    ; 2MB SSE-300 SRAM (3 cycles read latency) from M55/U55
    SRAM SRAM_START SRAM_SIZE
    {
    #if (ETHOSU_MODEL == 0)
        ; Place network model in SRAM
        * (network_model_sec)
    #endif

    #if (ETHOSU_ARENA == 0)
        ; Place tensor arena in SRAM
        * (.bss.tensor_arena)
    #endif

    ; Place scratch buffer in SRAM
    * (.bss.ethosu_scratch)
    }
}
```

Above: A snippet from the scatter file for the Corstone-300 for placing different symbols in the SRAM.

In the above snippet, if the ETHOSU_ARENA cmake parameter is equal to 0, the .bss.tensor_arena is placed in the SRAM. ETHOSU_ARENA equalling 0 corresponds to the case when building the application for Ethos-U55 with Shared_Sram memory mode and hence the tensor arena should be placed in the Sram.

Placing the model in memory follows the same logic. The model is an array of read-only data, and it can be placed in different in various parts of your memory with a section attribute. For

example, in the `ethos-u-core-platform/applications/baremetal/models/ethos-u55-128/keyword_spotting_cnn_small_int8/model.h` file, you have the following definition for placement of the model.

```
unsigned char networkModelData[] __attribute__((aligned(16),
 section("network_model_sec")))
```

The networkModelData array is generated after compiling the model with Vela for an Ethos-U55-128 for a specified memory mode and the array is placed in the network_model_sec section of the memory map. The network_model_sec section is then placed in the appropriate location in memory in the linker script and scatter file.

3.9 Customization of the Ethos-U Driver and RTOS integration

There are several scenarios that you would like to customize the Ethos-U driver. For example:

- In a bare metal application, after the Ethos-U NPU starts an inference operation, you might want to put the processor and some other parts of the microcontroller or SoC into sleep mode.
- In an application with an RTOS running, there could be multiple application threads that contain ML work loads. As a result, you might want to add semaphore operations in the Ethos-U driver to ensure that only one application thread can access to the Ethos-U NPU at a time.
- In an application with an RTOS running, after the Ethos-U NPU starts an inference operation, you might want to allow the RTOS to context switch into other RTOS application threads, and resume the application thread when an interrupt is received from the Ethos-U NPU.

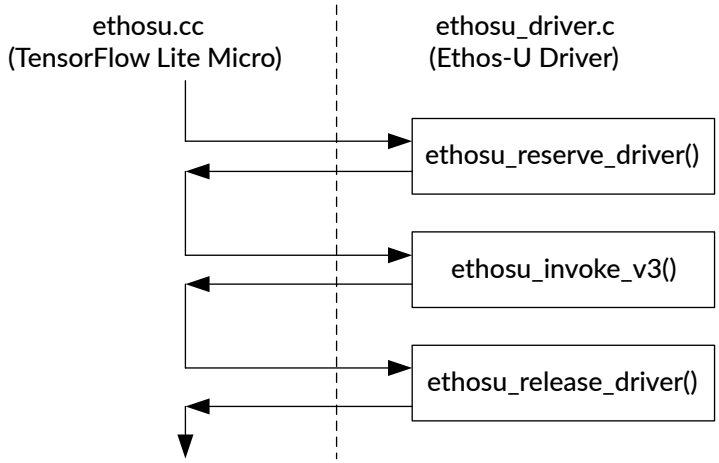
In order to allow these customization, some of the functions in the Ethos-U driver are defined with a weak attribute. It means that the implementations of those functions can be overridden by a different implementation without the need to modify the source code of the Ethos-U driver. Example of the weak functions included:

- Inference begin and end (ethosu_inference_begin, ethosu_inference_end)
- D-cache maintenance functions (e.g. D-cache flush (clean), D-cache invalidate)
- RTOS functions (mutex, semaphore)
- Ethos-U Interrupt handler

Some of these weak functions are dummy functions and need to be ported. The source code of the Ethos-U driver is available in: https://review.mlplatform.org/plugins/gitiles/ml/ethos-u/ethos-u-core-driver

The Ethos-U driver source code is in the file `ethosu_driver.c`. During an inference operation, the driver code (including the `ethosu_invoke_v3()` function) in the `ethosu_driver.c` is called by `ethosu.cc` in the TensorFlow Lite Micro kernel when the kernel encounters a custom operator. By default, this function returns when the inference is completed.

Figure 3-9: Interface between TFLM and Ethos-U driver

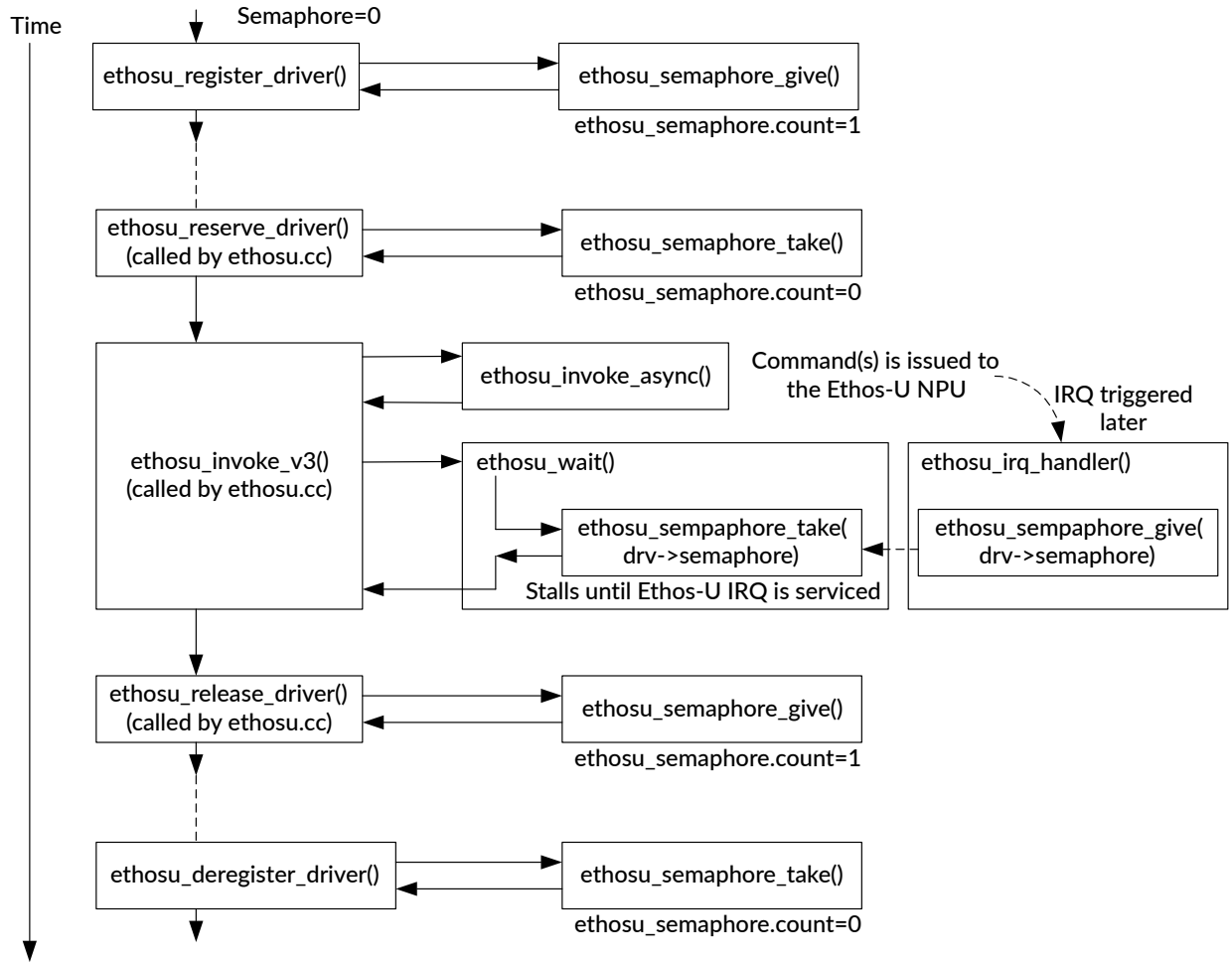


The Ethos-U driver code includes a bare metal semaphore implementation as default. Two semaphores are used:

- one semaphore (ethosu_semaphore) for reserving the driver, ensuring that if an ROTS is used, only one ML application thread can use the Ethos-U NPU at a time.
- one semaphore (drv->semaphore) is used to wait for the NPU to complete the task.

An illustration of the semaphore operations during a NN inference is shown below:

Figure 3-10: Ethos-U driver semaphore sequence during an inference



The semaphore sequence is described as follows:

- The semaphore is setup when the ethosu_register_driver function is called.
  - This happens during the execution of `ethos_init()` (the Ethos-U initialization function). For each Ethos-U NPU in the SoC, the application has to call the `ethos_init()` function to initializes the NPU, and setup the associated security & privilege level. If there are multiple Ethos-U NPU in the system, the ethos_init function is called multiple times, each time with a different driver handle (`struct ethosu_driver *drv`). So you have one Ethos-u driver handle per Ethos-U NPU.
- The `ethosu_register_driver()` function registers the driver and gives a semaphore (ethosu_semaphore). In other words, this driver now has an associated Ethos-U NPU.
- Then, in the TensorFlow Lite Micro code (`ethosu.cc`), the `ethosu_reserve_driver()` is called. This function executes the `ethosu_semaphore_take()` function, and in applications that have only one ML application thread, the `ethosu_reserve_driver()` function takes the semaphore (ethosu_semaphore) immediately because the semaphore that was created during `ethosu_register_driver()` is still available. In applications that have multiple ML application

threads, the application could stall at this stage if the Ethos-U NPU is being used by another application thread.

- After acquiring the semaphore successfully, the TensorFlow Lite Micro code(ethosu.cc) then executes `ethosu_invoke_v3()` which in turns call the `ethosu_invoke_async()`. The `ethosu_invoke_async()` function starts running the command stream on the Ethos-U NPU hardware and returns (Note: The completion of the function is asynchronous to the NPU's operations.).
- Inside `ethosu_invoke_v3()`, the TensorFlow Lite Micro code(ethosu.cc) then executes `ethosu_wait()`. This function contains a state machine, and can be operate in blocking or non-blocking mode. In this instance, blocking-mode is used:
  - The state machine switch to ETHOSU_JOB_RUNNING state. In block mode, nothing happen in this state. Because the "break" statement is not execute, the execution flow fall into the next state - ETHOSU_JOB_DONE.
  - In ETHOSU_JOB_DONE state, the code executes `ethosu_semaphore_take()` function again, but this time using a semaphore inside the driver (drv->semaphore). Because the counter in this semaphore is 0, the application thread is stalled. At this stage, the default code put the processor to sleep because the loop that is polling the semaphore contains a WFE (Wait for Event) instruction. If the semaphore function is replaced with an OS specific function, the OS can context switch into other threads.
  - Sometime later, when the Ethos-U NPU completed the inference operation, the Ethos-U interrupt handler is executed. The handler calls the `ethosu_semaphore_give()` function and updates drv->semaphore. Please note that the `ethosu_semaphore_give()` function contains a SEV instruction. For single core Cortex-M systems, the SEV is necessary. However, in multiple core systems where the processor servicing the interrupt could be different from the one executing `ethosu_semaphore_take()`, the SEV instruction is needed to wait up the other processor.
  - Now the `ethosu_semaphore_take()` function inside `ethosu_wait()` can take the semaphore (drv->semaphore) and complete the rest of the operations.
  - On success(status register of the NPU=true), the `ethosu_wait()` function returns 0 and the `ethosu_invoke_v3()` function completes.
- The TensorFlow Lite Micro code(ethosu.cc) then executes `ethosu_release_driver()`. This release the semaphore (ethosu_semaphore) and if there is another application threads that is waiting to use the Ethos-U, it can take the semaphore and resume.
- Some stage later, software can optionally execute `ethosu_deregister_driver()` if the Ethos-U is no longer needed.

In addition to the semaphore code, there are also mutex APIs that could be used to semaphore operations are atomic.

If an RTOS is used, the semaphore and mutex API must be ported to RTOS specific implementation.

Please note that:

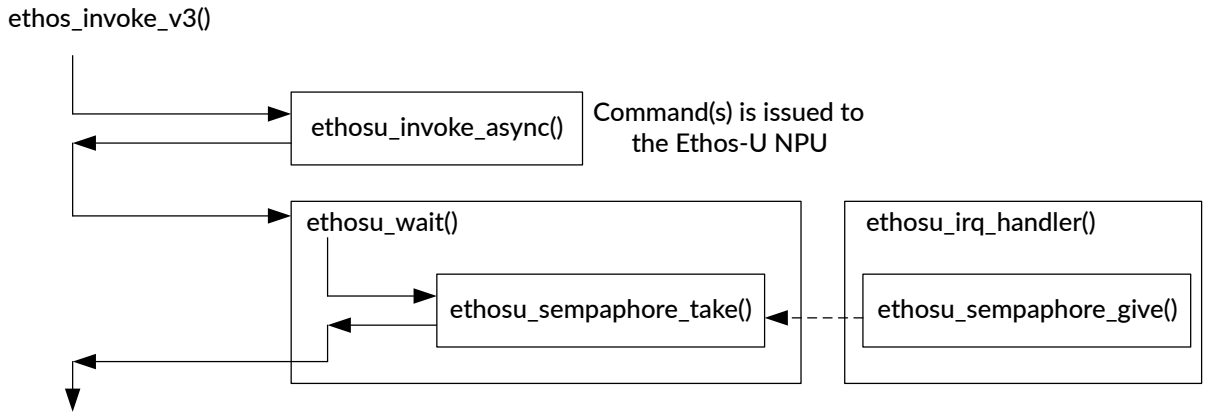
- The ethosu_semaphore_give() API is executed during ethosu_irq_handler(). Some RTOS might have different semaphore APIs for threads and interrupt handlers, so care must be taken to ensure that the correct API is used when porting the code to an RTOS.

- The Ethos-U Driver can support multiple Ethos-U NPU by having different driver handles for each NPU instance.

3.9.1 Putting the processor into sleep while the Ethos-U NPU is running in baremetal applications

By default, after an inference operation is started in the processor, the Ethos-U driver put the processor into sleep using a WFE (Wait for Event) instruction. This happens within the `ethosu_semaphore_take()` function, and the sequence of events is illustrated in the following diagram:

Figure 3-11: Ethos-U invoke function



The `ethosu_semaphore_take()` function is declared with a weak attribute, and therefore can be customized by application developers. By default this function contains an WFE operation to allow the processor to enter sleep mode, but addition power control code can be added to utilize low power capability of the microcontroller or SoC.

```
// Bare metal simulation of waiting/sleeping for and then taking a semaphore using
 intrinsics
int __attribute__((weak)) ethosu_semaphore_take(void *sem)
{
    struct ethosu_semaphore_t *s = sem;
    while (s->count == 0)
    {
        __WFE(); // Additional codes could be added for device specific power saving
 features.
    }
    s->count--;
    return 0;
}
```

3.9.2 Adding RTOS support

If an RTOS is used and there are multiple application threads, then putting the processor into sleep mode is not the best option because there could be other active threads waiting to be executed. In such case, we can suspend the current executing thread so that the RTOS can context switch into another active thread that is waiting to execute. To do this, we should replace the semaphore functions (ethosu_semaphore_take() and ethosu_semaphore_give()) in Ethos-U driver with RTOS specific semaphore functions.

With this arrangement, the current thread is put into an inactive state:

- when waiting for the NPU resource (i.e. waiting in `ethosu_reserve_driver()`), and
- after the Ethos-U starts running (i.e. `waiting in ethosu_wait()`).

With semaphore support in RTOS, the processor can context switch into other threads if there is other active thread waiting to be executed. If there is no other active thread waiting, the RTOS will execute its own idle thread, which should put the processor into sleep model providing that the idle thread contains a WFE instruction in the idle loop.

In addition, the semaphore operations in `ethosu_reserve_driver()` and `ethosu_release_driver()` ensures that, if there are multiple ML application threads sharing the same Ethos-U NPU, only one of the threads can have access to the Ethos-U NPU at a time.

Example code for FreeRTOS is available here: https://review.mlplatform.org/plugins/gitiles/ml/ethos-u/ethos-u-core-platform/+/refs/heads/master/applications/freertos/main.cpp

This contains FreeRTOS specific implementation of “weak” functions in the Ethos-U driver.

3.9.3 Ethos-U Driver Configuration

A few registers in the Ethos-U need to be configured by the Ethos-U driver in order to achieve optimal performance of the Ethos-U. All of the registers that need to be configure are located in:

- `core_driver/src/ethosu_config_u55.h`, and
- `core_driver/src/ethosu_config_u65.h`

The hardware has 4 AXI_LIMIT registers to set limits for the two ports of the AXI0 and AXI1 interfaces. The optimal setting for the AXI_LIMIT is hardware platform specific.

4. Tool Support for the Arm Ethos-U NPU

Several tools provide support for the Arm Ethos-U NPU:

- Vela

  This tool is used to compile a TensorFlow Lite flatbuffer file into an optimised version that can run on an embedded system containing an Arm Ethos-U NPU.

  In order to be accelerated by the Ethos-U NPU the network operators must be quantised to either 8-bit (unsigned or signed) or 16-bit (signed).

  The optimised model will contain TensorFlow Lite Custom operators for those parts of the model that can be accelerated by the Ethos-U NPU. Parts of the model that cannot be accelerated are left unchanged and will instead run on the Cortex-M series CPU using an appropriate kernel (such as the Arm optimised CMSIS-NN kernels).

  For more information, see Ethos-U Vela.
- MLIA - Machine Learning Inference Advisor

  The ML Inference Advisor (MLIA) is used to help AI developers design and optimize neural network models for efficient inference on Arm® targets (see supported targets) by enabling performance analysis and providing actionable advice early in the model development cycle. The final advice can cover supported operators, performance analysis and suggestions for model optimization (e.g. pruning, clustering, etc.).

  For more information, see MLIA Usage.
- Arm Virtual Hardware, VSI Interfaces for Sensors/Audio/Video

  Arm Virtual Hardware (AVH) provides simulation models, software tooling, and infrastructure that can be integrated into CI/CD and MLOps development flows.

  The Virtual Streaming Interface (VSI) is a flexible memory-mapped peripheral that is part of Arm Fixed Virtual Platforms (FVPs). VSI is used to simulate data streaming interfaces like audio, video, and sensors, commonly used in IoT and Machine-Learning applications. The system provides eight independent VSI instances that can function in parallel, allowing for multi-channel input/output interfaces.

  For more information, see Arm Virtual Hardware Usage.
- SDS Framework

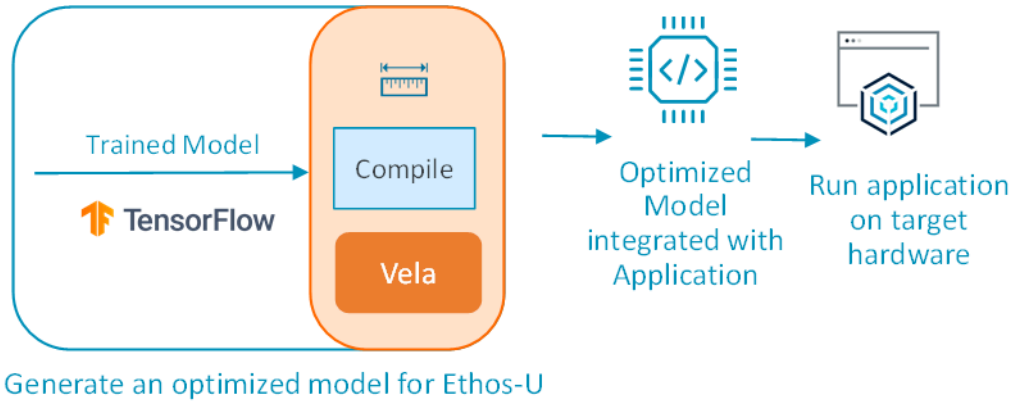
  The Synchronous Data Stream (SDS) Framework implements a data stream management, provides methods and helper tools for developing and optimizing embedded applications that integrate DSP and ML algorithms. This framework relates to the Compute Graph streaming that is part of the CMSIS-DSP compute library.

  For more information, see SDS Framework Usage.

4.1 Ethos-U Vela

Ethos-U Vela is a software tool developed by Arm that compile a TFLμ model into an optimized version that can run on the Ethos-U NPU. It accepts TensorFlow Lite models as input, applies optimizations such as memory optimization and layer fusion techniques, and generates a compiled binary that's specifically optimized for the Ethos-U architecture, maximizing the hardware's features for efficient execution of machine learning workloads.

Figure 4-1: Ethos-U Vela Development Flow



The optimized model contains TensorFlow Lite custom operators (supported operators) for those parts of the model that can be accelerated by the Ethos-U NPU. Parts of the model that cannot be accelerated are left unchanged and will instead run on the Cortex-M series CPU using an appropriate kernel.

Vela trials a number of different compilation strategies and applies a cost function to each one. It then chooses the optimal execution schedule for each supported operator or group of operators. Vela is bit-accurate with TF Lite reference kernels, and hence there is no loss of accuracy when you pass the NN through the Vela compiler.

The Vela have the ability to report estimated performance. However, the results from Vela are rough estimation and software developers should consider using MLIA together with a performance model to obtain more accurate performance data.

4.1.1 Memory optimization

The Vela compiler also performs various memory optimizations to reduce both the permanent (for example flash) and runtime (for example SRAM) memory requirements.

One such technique for permanent storage is the compression of all the weights in the model.

Another technique is cascading, which addresses the runtime memory usage. Cascading reduces the maximum memory requirement by splitting the feature maps (FM) of a group of consecutively supported operators into stripes. A stripe can be either the full or partial width of the FM. And it

can be the full or partial height of the FM. Each stripe in turn is then run through all the operators in the group.

The parts of the model that can be optimized and accelerated are grouped and converted into TensorFlow Lite custom operators. The operators are then compiled into a command stream that can be executed by the Ethos-U NPU.

Finally, the optimized model is written out as a TFLμ model and a Performance Estimation report is generated that provides statistics, such as memory usage and inference time.

The compiler includes numerous configuration options that allow you to specify various aspects of the embedded system configuration (for example the Ethos-U NPU configuration, memory types, and memory sizes). There are also options to control the types of optimization that are performed during the compilation process.

4.1.2 Requirements

The following should be installed prior to the installation of Vela:

- Windows 10 or Linux (amd64)
- Python 3.9 or higher
- Development version containing the Python/C API header files e.g. apt install python3.9-dev or yum install python39-devel
- A C99 capable compiler and associated toolchain
- For Linux operating systems, a GNU toolchain is recommended.
- For Microsoft Windows 10, Microsoft Visual C++ 14.2 Build Tools is recommended. See https://wiki.python.org/moin/WindowsCompilers

4.1.3 Installation

To install ethos-u-vela, use the following command:

```
pip3 install ethos-u-vela
```

In order for ethos-u-vela to work with an older version of NumPy that uses different C APIs, you will need to install the desired NumPy version first, and then build ethos-u-vela with that specific NumPy version. The workaround build instruction will be:

```
pip uninstall ethos-u-vela
pip install numpy==1.21.3 --force
pip install "setuptools_scm[toml]<6" wheel
pip install ethos-u-vela --no-build-isolation --no-cache-dir
```

4.1.4 Invocation

Vela runs with an input `.tflite` file passed on the command line. This file contains the neural network to be compiled. The tool then outputs an optimised `.tflite` file with a `_vela` suffix in the file name, along with performance estimate (EXPERIMENTAL) CSV files, all to the output directory. It also prints a performance estimation summary back to the console, see Vela Performance Estimation Summary.

```
Neural network model compiler for Arm Ethos-U NPUs
positional arguments:
  NETWORK               Filename of the input TensorFlow Lite for Microcontrollers
 network
options:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  --api-version         Displays the version of the external API.
  --supported-ops-report
                        Generate the SUPPORTED_OPS.md file in the current working
 directory and exit
  --list-config-files   Display all available configurations in the `config_files`
 folder and exit. To select config
                        file, use the --config argument with one of the listed
 config files (For example: --config
                        Arm/vela.ini )
  --output-dir OUTPUT_DIR
                        Output directory to write files to (default: output)
  --enable-debug-db     Enables the calculation and writing of a network debug
 database to output directory
  --config CONFIG       Vela configuration file(s) in Python ConfigParser .ini file
 format
  --verbose-all         Enable all verbose options
  --verbose-config      Verbose system configuration and memory mode
  --verbose-graph       Verbose graph rewriter
  --verbose-quantization
                        Verbose quantization
  --verbose-packing     Verbose pass packing
  --verbose-tensor-purpose
                        Verbose tensor purpose
  --verbose-tensor-format
                        Verbose tensor format
  --verbose-schedule    Verbose schedule
  --verbose-allocation  Verbose tensor allocation
  --verbose-high-level-command-stream
                        Verbose high level command stream
  --verbose-register-command-stream
                        Verbose register command stream
  --verbose-operators   Verbose operator list
  --verbose-weights     Verbose weights information
  --verbose-performance
                        Verbose performance information
  --verbose-progress    Verbose progress information
  --show-cpu-operations
                        Show the operations that fall back to the CPU
  --timing              Time the compiler doing operations
  --force-symmetric-int-weights
                        Forces all zero points to 0 for signed integer weights
  --accelerator-config {ethos-u55-32,ethos-u55-64,ethos-u55-128,ethos-u55-256,ethos-
u65-256,ethos-u65-512}
                        Accelerator configuration to use (default: ethos-u55-256)
  --system-config SYSTEM_CONFIG
                        System configuration to select from the Vela configuration
 file (default: internal-default)
  --memory-mode MEMORY_MODE
                        Memory mode to select from the Vela configuration file
 (default: internal-default)
  --tensor-allocator {LinearAlloc,Greedy,HillClimb}
```

```
                        Tensor Allocator algorithm (default: HillClimb)
 --show-subgraph-io-summary
                        Shows a summary of all the subgraphs and their inputs and
outputs
 --max-block-dependency {0,1,2,3}
                        Set the maximum value that can be used for the block
dependency between npu kernel operations
                        (default: 3)
 --optimise {Size,Performance}
                        Set the optimisation strategy. The Size strategy results in
minimal SRAM usage (does not use
                        arena-cache-size). The Performance strategy results in
maximal performance (uses the arena-
                        cache-size if specified) (default: Performance)
 --arena-cache-size ARENA_CACHE_SIZE
                        Set the size of the arena cache memory area, in bytes. If
specified, this option overrides the
                        memory mode attribute with the same name in a Vela
configuration file
 --cpu-tensor-alignment CPU_TENSOR_ALIGNMENT
                        Controls the allocation byte alignment of cpu tensors
including Ethos-U Custom operator inputs
                        and outputs (default: 16)
 --recursion-limit RECURSION_LIMIT
                        Set the recursion depth limit, may result in RecursionError
if too low (default: 1000)
 --hillclimb-max-iterations HILLCLIMB_MAX_ITERATIONS
                        Set the maximum number of iterations the Hill Climb tensor
allocator will run (default: 99999)
```

Detailed explanation of the parameters can be found in the Configuration Line Interface Reference.

4.1.5 Command Examples

In most cases all parameters required will be configured via the `vela.ini` config file. In this case the call will look like this:

```
vela --config vela.ini my_model.tflite --accelerator-config ethos-u55-128 --memory-
mode Shared_Sram --optimise Performance
```

4.1.6 vela.ini File Example

Detailed explanation of the options can be found in the Configuration File Reference

```
; Ethos-U55 High-End Embedded: SRAM (4 GB/s) and Flash (0.5 GB/s)
[System_Config.Ethos_U55_High_End_Embedded]
core_clock=500e6
axi0_port=Sram
axi1_port=OffChipFlash
Sram_clock_scale=1.0
Sram_burst_length=32
Sram_read_latency=32
Sram_write_latency=32
OffChipFlash_clock_scale=0.125
OffChipFlash_burst_length=128
OffChipFlash_read_latency=64
OffChipFlash_write_latency=64
```

4.1.7 Vela compiler - Optimization considerations

The detail of Vela Compiler usage is covered in this page. In this section, additional information related to optimization is provided.

4.1.7.1 Vela schedulers and implications on memory footprint

The Vela compiler supports two scheduling algorithms for the most common use cases:

- Optimise for maximum performance in terms of inferences per second via the performance scheduler. This is the default scheduler when using the Vela compiler.
- Optimise a neural network for minimum peak SRAM usage during an inference. In this way, you can benefit from hardware acceleration in a low SRAM budget. Vela reduces the peak SRAM usage by reusing tensors with cascading. This results in slightly lower performance and requires to re-read some weights from the memory. You can use the Size scheduler by adding the `--optimise Size` CLI option to your Vela command.

On the Ethos-U55, you also have the option to optimise a network for performance within a given memory limit. Imagine that the key model for your design will be TF Lite Micro's person detection model, and you would like to know the amount of SRAM memory required by the Ethos-U to accelerate the inference. When you compile the model for the Ethos-U with the following command:

```
vela person_detect.tflite --accelerator-config=ethos-u55-128 --config <path to your
 vela.ini> --memory-mode=Shared_Sram --system-config=Ethos_U55_High_End_Embedded
```

You obtain a memory footprint summary as follows:

```
Total SRAM used                                 72.72 KiB
Total Off-chip Flash used                      263.55 KiB
```

The *Total SRAM used* value shows the peak SRAM usage of the Ethos-U microNPU for the inference. Hence, from top-level architecture standpoint you need to have more than 72.72 KiB of SRAM on your SoC to perform the inference. The 263.55 KiB is the size of the Read-Only data stored in the Flash in the case of the Ethos-U55. Note that the Vela compiler only has visibility over the input, output, and intermediate tensors of your model running on the Ethos-U microNPU. The compiler does not have visibility over the kernels you are using when compiling TF Lite Micro and it does not know anything about your wider embedded software stack (if you are using a RTOS or how much memory you allocate in your embedded application code). Therefore, from top-level architecture standpoint you would need to top up the 72.72 KiB of SRAM for the inference by the amount of memory needed for the rest of the software stack.

Can you accelerate the inference while using smaller amount of SRAM? You can when using the –arena-cache-size CLI option to the Vela command. For example, imagine you do not want to sacrifice 72.72 KiB of your memory budget solely for the inference and you want the inference to be performed in no more than 60 KiB. You can schedule the execution of the network in less

than 60 KiB by passing –arena-cache-size 61440 CLI option to the Vela command when optimising the model. An important assumption in this optimisation is that to achieve lower SRAM usage, the microuNPU will need to re-read more weights from the memory wired to AXI1. Hence, it is good to make sure that your memory is capable to deliver the required bandwidth on AXI1. To help designers to get better insight of the memory access throughput, the Ethos-U NPU provides a Performance Monitoring Unit (PMU). The PMU contains counters for measuring hardware events such as memory accesses on specific memory interface. If you run the ML model on the Corstone-300 FVP or FPGA and reading the following PMU counters after the processing is completed:

- The axi0_rd_data_beat_received,
- axi0_wr_data_beat_written, and
- axi1_rd_data_beat_received

You can then obtain the number of beats that were transferred on the two AXI interfaces and deduce the expected bandwidth. When the Ethos-U carries out a memory transaction, for example a read request, it reads data in beats. The Ethos-U can be configured to issue requests in 64-byte beats, 128 byte. For the Ethos-U65, the NPU can be configured to work with 64, 128 or 256-byte beats. The number of beats is configured via the max_beats field of the AXI_LIMIT registers. If for example you have an Ethos-U55 configured to access the memory in 64-byte beats and the microNPU needs to read 50 bytes of data, the system will still do a 64-byte transaction. Having said that, the Ethos-U is designed to issue memory transactions as close as possible to the maximum beats that are configured.

To enable the bus traffic to be analyzed, the Ethos-U provides counters in its Performance Monitor Unit (PMU). Example of using PMU counters for this calculation is in the following section.

4.1.7.2 Examples of using PMU counters in the Ethos-U NPU

Consider compiling a network that maps fully to the Ethos-U with Vela's Performance scheduler and the following performance numbers are obtained from the Ethos-U's Performance Monitor Unit:

PMU Counter	Value
ETHOSU_PMU_NPU_ACTIVE + ETHOSU_PMU_NPU_IDLE	649597
ETHOSU_PMU_AXI0_RD_DATA_BEAT_RECEIVED	222602
ETHOSU_PMU_AXI1_RD_DATA_BEAT_RECEIVED	56511
ETHOSU_PMU_AXI0_WR_DATA_BEAT_WRITTEN	88584

If you assume a frequency of 500MHz, the total NPU cycles would translate to 500MHz / 649597 = 769 inferences per second. The PMU counter results on AXI0 translate to an average bandwidth of 769 * (222602 + 88584) * 8 / (1024 * 1024) = 1825 MB/s(we multiply by 8 to convert to bytes and divide by 1024 * 1024 to obtain MB). On AXI1, you would have an average bandwidth of 769 * 56511 * 8 / (1024 * 1024) = 331 KB/s.

Compiling the same network with the Size scheduler results in the following performance results:

PMU Counter	Value
ETHOSU_PMU_NPU_ACTIVE + ETHOSU_PMU_NPU_IDLE	1111167
ETHOSU_PMU_AXI0_RD_DATA_BEAT_RECEIVED	97227
ETHOSU_PMU_AXI1_RD_DATA_BEAT_RECEIVED	146649
ETHOSU_PMU_AXI0_WR_DATA_BEAT_WRITTEN	32504

This time if you assume a frequency of 500MHz, you obtain approximately 449 inferences/second and would get average bandwidth of 444MB/s on AXI0 and 502KB/s on AXI1. From SoC architecture standpoint you need to ensure that your system is capable of delivering the above bandwidths to achieve 449 inferences per second.

4.1.7.3 Memory modes

From an application standpoint, TensorFlow Lite Micro provides two memory regions that you can control – the tensor arena (read/write data) and the model (constant read-only data such as weights or biases of the neural network). The Ethos-U55 supports two different placement schemes of the tensor arena and the model.

1. The tensor arena is in the memory connected to the AXI0 interface (usually SRAM), and the model is in the memory connected to AXI1(usually Flash). In Vela's vocabulary, this memory configuration is called Shared_Sram.
2. The tensor arena and the model are both placed in the same memory. This configuration is called Sram_Only. Note that from hardware standpoint, the AXI0 and AXI1 interfaces are also both connected to the same memory.

The Ethos-U65 supports the following placements of the tensor arena and the model.

1. The tensor arena and the constant data can reside in the memory connected to AXI 1(usually DRAM). The memory connected to the AXI0 interface (usually SRAM) is only used as a cache to store the most accessed tensors to perform the inference. Note that this memory mode is only available for the Ethos-U65. This memory mode is called Dedicated_Sram when compiling a network with Vela. In this memory mode, the –arena-cache-size parameter controls the amount of SRAM available on your system.
2. The tensor arena and the model are both connected to the memory utilising the AXI0 interface. This configuration is called Sram_Only.

Note that for an Ethos-U55, the AXI1 interface is read-only and hence the tensor arena must be placed in the memory wired to AXI0. On an Ethos-U65, the AXI1 interface is read/write and the tensor arena can be placed in the memory connected to AXI1. The benefit is that you can store larger models in the DRAM and still obtain acceleration.

4.2 MLIA Usage

MLIA (Machine Learning Inference Advisor) is currently an experimental software tool and is provided as-is, without any guarantees or warranties of its functionality, reliability, or suitability for any specific purpose

4.2.1 Requirements

It is recommended to use a virtual environment for MLIA installation, and a typical setup for MLIA requires:

- Ubuntu® 20.04.03 LTS
- Python® 3.8.1 or higher
- Ethos™-U Vela

4.2.2 Installation

To install MLIA, run the following command:

```
pip install mlia
```

4.2.3 Invocation

MLIA usage is as follows:

```
usage: mlia [-h] [-v] {check,optimize} ...
ML Inference Advisor 0.6.0
Help the design and optimization of neural network models for efficient inference on
 a target CPU or NPU.
Supported Targets/Backends:
```

Target	Backend(s)	Status	Advice: comp/perf/opt
Cortex-A <cortex-a>	Arm NN TensorFlow Lite delegate <armnn-tflite-delegat…	BUILTIN	YES/NO/NO
Ethos-U55 <ethos-u55>	Vela <vela> Corstone-310 <corstone-310> Corstone-300 <corstone-300>	BUILTIN NOT INSTALLED NOT INSTALLED	YES/YES/YES
Ethos-U65 <ethos-u65>	Vela <vela> Corstone-310 <corstone-310> Corstone-300 <corstone-300>	BUILTIN NOT INSTALLED NOT INSTALLED	YES/YES/YES
TOSA	TOSA Checker	NOT INSTALLED	YES/NO/NO

```
| <tosa>       | <tosa-checker>                |                 |                          |
└──────────────┴───────────────────────────────┴─────────────────┴──────────────────────────┘
Comp/Perf/Opt: Advice categories compatibility/performance/optimization
Use command 'mlia-backend' to install backends.
options:
  -h, --help            Show this help message and exit
  -v, --version         Show program's version number and exit
Commands:
  {check,optimize}
    check               Generate a full report on the input model
    optimize            Show the performance improvements (if any) after applying
                        the optimizations
```

4.2.4 Command Examples

A very typical tool invocation for Ethos-U on a Corstone-3xx based device is shown here. You can run `mlia -h` first, validating that you have the necessary backend(s) installed.

```
# Explicitly specify the target profile and backend(s) to use
# with --backend option
mlia check ~/models/ds_cnn_large_fully_quantized_int8.tflite \
    --target-profile ethos-u55-256 \
    --performance \
    --backend "vela" \
    --backend "corstone-300"
```

4.3 Arm Virtual Hardware Usage

For detailed information on AVH's capabilities and how to utilize them effectively, refer to the Getting Started Guide. This guide provides a comprehensive walkthrough of the setup process, explains the navigational structure of the AVH platform, and offers essential technical details to maximize the utility of the system.

The Getting Started Guide provides a step-by-step guide through the Continuous Integration (CI) workflow operation and its setup using Arm Virtual Hardware (AVH). Here's an overview of its contents:

Overview: Overview of the common steps in the CI workflow, including local development using a toolchain such as Keil MDK and Arm Fixed Virtual Platforms, setup of a CI pipeline using GitHub Actions, automated program build and testing in the cloud with AVH, and failure analysis and local debugging.

Prerequisites: The guide outlines the prerequisites required to reproduce the operation of the example project.

Develop tests: It introduces the concept of developing unit tests using the Unity Framework. Includes an example project.

- Forking the Getting Started GitHub repository: The guide explains the process of creating a GitHub repository by either creating a new one or forking from the AVH GetStarted repository.

- Setup local project on your PC: It provides instructions on setting up the local project on your PC. This includes cloning the repository onto your local PC and setting up the project in Keil MDK.
- Implement tests: The guide outlines how to implement tests using the Unity Framework and how to redirect standard output to be visible during the debug session1.
- Build and Run the example in Keil MDK: It provides steps on how to build and execute the program in Keil MDK. It also explains how to export the project for use in command-line CI environments.

Setup CI pipeline: The guide provides information on setting up the CI pipeline which is triggered on every code change via push and pull requests. The CI implementation in the example is implemented with GitHub Actions.

AWS setup: The guide explains the process of setting up AWS to enable the execution of the example CI pipeline on cloud-hosted Arm Virtual Hardware instance.

GitHub Actions setup: It introduces the concept of running Arm Virtual Hardware with GitHub Actions.

4.4 SDS Framework Usage

For effective ML algorithm selection, training, and validation a large set of representative and qualified data is a pre-requisite. As explained under section “III. Development Process” correct decisions can only be done when sufficient training data exist. ML algorithms can be introduced in a step-by-step approach where first a classic algorithm is deployed, and scenarios of interest are captured. Having a deployed fleet of IoT endpoint devices that captures such data is therefore invaluable.

To capture real-world data from sensors and audio sources, Arm developed the Synchronous Data Stream (SDS) Framework. It consists of several components:

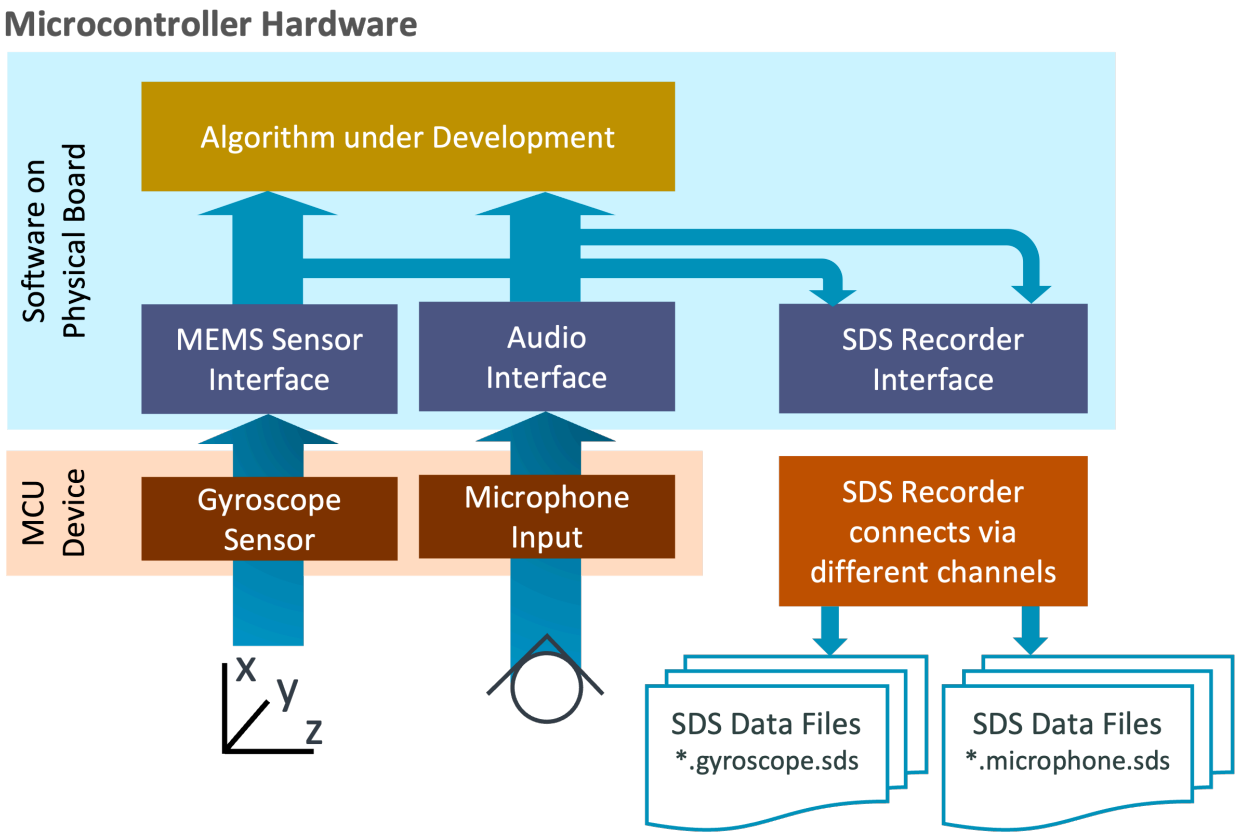
- SDS Recorder Interface: provides methods to record real-world data in SDS data files for analysis and development of DSP and ML algorithms.
- SDS Metadata: describe the content of SDS data files along with scaling and formatting information.
- SDS Utilities: tools to record, convert, and display SDS data files.
- SDS Playback: Using Arm Virtual Hardware with the Virtual Streaming Interfaces (VSI) allows to stimulate an algorithm under development with real-world data.

4.4.1 SDS Recorder Interface

The SDS Recorder Interface is designed for recording real-world data in SDS data files. The SDS Recorder is an integral part of the target application and enables data streaming via various interfaces such as TCP/IP via Ethernet, UART, or USB. It can also be used to capture data in a

deployed IoT endpoint device to report situations where the current ML model has gaps in the training data.

Figure 4-2: SDS Recorder Interface



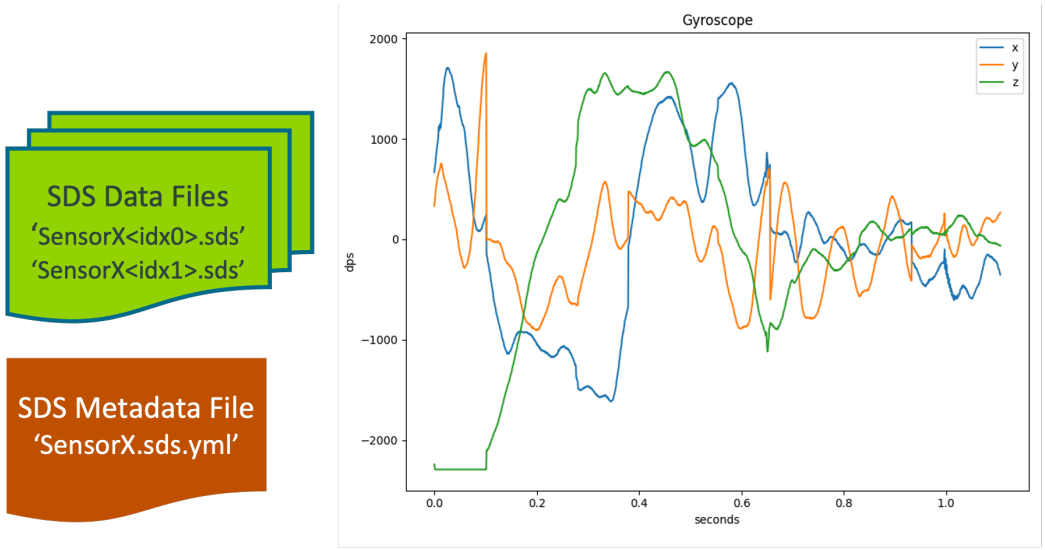
The SDS Recorder Interface is used to capture real-world data with the sensors of the final hardware target.

SDS connects via different channels and can be also embedded in deployed IoT endpoint devices.

4.4.2 SDS Metadata

The SDS Metadata file provides information about the content of SDS data files. This metadata information is used to display meaningful information to the user. It also identifies the data streams for inputs to DSP design utilities, MLOps development workflows, or AVH data playback.

Figure 4-3: SDS Metadata File

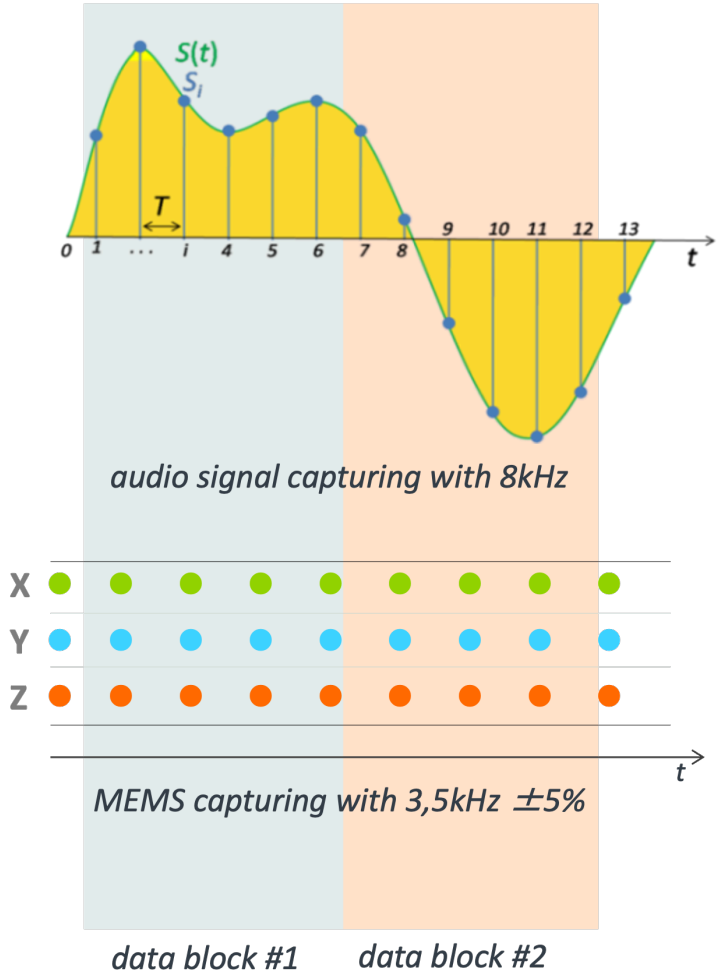


The SDS metadata file provides information about the content of SDS data files; used to display meaningful information and provide context for MLOps systems.

The SDS data files have multiple use cases such as:

- Input to Digital Signal Processing (DSP) development tools such as filter designers
- Input to Machine Learning (ML) model classification, training, and performance optimization.
- Verify execution of DSP algorithm on Cortex-M targets with off-line tools.
- Playback real-world SDS data files for algorithm validation using Arm Virtual Hardware.

Figure 4-4: SDS Sample Frequencies



Sensors may have independent clock sources with tolerances which results in different size records for block procession algorithms.

The SDS Recorder is a flexible software component that can be connected to various output channels. The table below shows the different communication speeds that can be achieved depending on the output channel.

Development Board	Output Channel	Measured speed	Comment
NXP IMXRT1050-EVKB	TCP/IP via Ethernet	2 MB/s	
NXP IMXRT1050-EVKB	File System	2.85 MB/s	MicroSD card
NXP IMXRT1050-EVKB	VCOM (High-Speed)	11.8 MB/s	
ST B-U585I-IOT02A	VCOM (Full-Speed)	600 kB/s	
ST B-U585I-IOT02A	UART	80 kB/s	Baud Rate: 921600

Refer to SDS Recorder in the GitHub repository for access to examples.

4.4.3 SDS Utilities

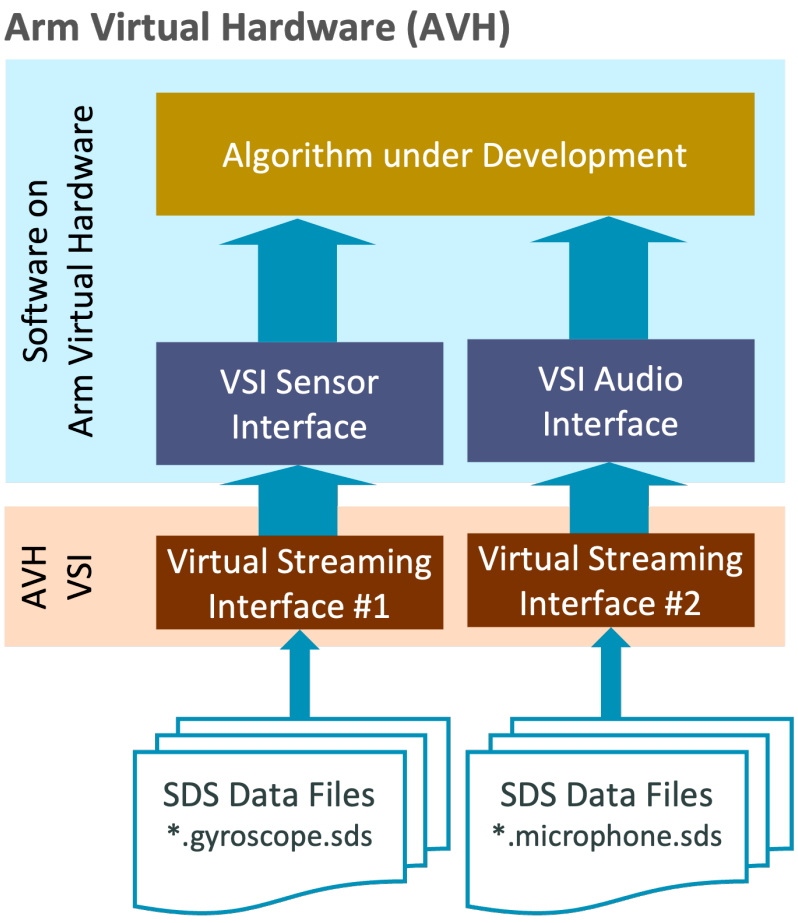
These utilities allow to analyze and convert the SDS data files.

Refer to SDS Utilities in the GitHub repository for more information and access to these utilities.

4.4.4 SDS Playback

Arm Virtual Hardware (AVH) is available in multiple deployments such as GitHub, Qeexo AutoML, Keil Studio Cloud, and AWS AMI for flexible cloud access. In the desktop version of Keil MDK it supports test case development and verification of algorithms. With DevOps systems such as GitHub actions it supports continuous integration workflows for build and test automation. SDS data files can be used as input during validation tests. AVH also supports A/B comparisons of ML algorithms and helps therefore to select the best matching algorithm for an application. As it is part of some MLOps systems, it helps to validate ML models before deploying it to physical hardware devices.

Figure 4-5: SDS on Arm Virtual Hardware



AVH provides virtual streaming interfaces that can be used to playback SDS data to an algorithm under development; useful for repeatable validation tests or ML model verification. The SDS framework is designed to support also sensor fusion applications where the data of multiple different sources is combined. It might be useful to combine an audio signal with the information of a MEMS sensor to provide better prediction for machine failures. However, when combining data from multiple sources, tolerances of independent clock sources should be considered. The SDS framework has provisions to cope with such situations and provides independent clock information for multiple data streams.

Refer to SDS Playback in the GitHub repository for more information.

5. The Arm ML Zoo

Creating a new Neural Networks(NN) model for an application requires in depth understanding in Machine Learning (ML), and it can take very long time to optimize the network for specific use case. Fortunately, there are many different types of Neural Networks(NN) already developed for different applications and many of them are available in online databases called Model Zoo. The models in those model zoos can be used as a starting point for further development, so when an application developer creates an ML application, there is no need to create a new NN model from scratch.

The ARM ML Zoo repository hosts a variety of machine learning models optimized for ARM IP. The models cover different applications such as Anomaly Detection, Image Classification, Keyword Spotting, Noise Suppression, Object Detection, Speech Recognition, Superresolution, and Visual Wake Words.

Here's a brief overview:

1. Anomaly Detection: This includes three MicroNet models (Large, Medium, Small) that are optimized for INT8 and are compatible with TensorFlow Lite. They are particularly suitable for Cortex-M, Mali GPU, and Ethos U.
2. Image Classification: This category offers MobileNet v2 models optimized for INT8 and UINT8. They are compatible with TensorFlow Lite and are suitable for Cortex-A, Cortex-M, Mali GPU, and Ethos U.
3. Keyword Spotting: This includes CNN models (Large, Medium, Small), DNN models (Large, Medium, Small), and DS-CNN models (Large Clustered in FP32 and INT8, Large in INT8). They are optimized for INT8 or FP32 and are compatible with TensorFlow Lite. They are suitable for Cortex-A, Cortex-M, Mali GPU, and Ethos U, with some variations depending on the model.
4. Noise Suppression
5. Object Detection
6. Speech Recognition: This includes Wav2letter models (standard and pruned) and Tiny Wav2letter models (standard and pruned). They are optimized for INT8 and are compatible with TensorFlow Lite. They are suitable for Cortex-A, Cortex-M, Mali GPU, and Ethos U.
7. Superresolution: This includes the SESR model optimized for INT8 and compatible with TensorFlow Lite. It is suitable for Cortex-A and Mali GPU.
8. Visual Wake Words: This includes three MicroNet models (VWW-2, VWW-3, VWW-4) optimized for INT8 and compatible with TensorFlow Lite. They are particularly suitable for Cortex-M, Mali GPU, and Ethos U.

In most cases, software developers still need to retrain the ML model to fit the specific needs in their ML applications.

5.1 Integrate a ML-Zoo model

The content for this section is currently under development and will be added in the near future.

6. ML Embedded Evaluation Kit

The Arm ML Evaluation Kit is a tool designed to help developers build and deploy machine learning applications for Arm Cortex-M55 and Arm Ethos-U55 NPU. It provides ready-to-use software applications for Ethos-U55 systems, including image classification, keyword spotting, automated speech recognition, anomaly detection, and person detection.

The kit allows developers to evaluate the performance metrics of networks running on the Cortex-M CPU and Ethos-U NPU. It also includes a generic inference runner that can be used to develop custom ML applications for Ethos-U. The kit is based on the Arm Corstone-300 reference package, which is designed to help SoC designers build secure systems faster. The platform is available as Ecosystem FPGA (MPS3) and Fixed Virtual Platform (FVP) to allow development ahead of hardware availability.

6.1 ML Embedded Evaluation Kit - Quick Start

There are a range of options when trying out the Ethos-U NPUs.

- ML Embedded Evaluation Kit (Ethos-U ML Evaluation Kit examples): This software environment allows you to build and try out Ethos-U projects for a selection of platforms. These include Fixed Virtual Platforms and FPGA. This software environment runs in a Linux environment (it is possible to use Windows Subsystem for Linux).
- CMSIS-Pack based Machine Learning Examples - This contains examples that use CMSIS-Pack to handle software integration. This setup can be used in both Linux and Windows.

6.1.1 Using the ML Embedded Evaluation Kit

The ML Embedded Evaluation Kit provides examples based on the applications listed on this page.

A Quick Start Guide is available for the ML Embedded Evaluation Kit. A summary of the guide is as follows.

6.1.1.1 Supported platforms

In this section, we will discuss the various platforms that are compatible with the Machine Learning Evaluation Kit. These platforms range from physical hardware like the Arm MPS3 FPGA board to virtual environments such as the Fixed Virtual Platform (FVP) and Arm Virtual Hardware (AVH). Each of these platforms offers unique capabilities and advantages, making the ML Evaluation Kit a versatile tool for a variety of machine learning explorations.

- Arm MPS3 FPGA board with one of the following FPGA images:
  - AN552 FPGA image - This is based on the Corstone-300 subsystem containing Arm Cortex-M55 processor and the Ethos-U55 NPU.

    - AN555 FPGA image - This is based on the Corstone-310 subsystem containing Arm Cortex-M85 processor and the Ethos-U55 NPU.

  The FPGA images can be downloaded from here.
- Fixed Virtual Platform (FVP) for Corstone-300
- Arm Virtual Hardware (AVH) for Corstone-300
- Arm Virtual Hardware (AVH) for Corstone-310

6.1.1.2 System and software requirements

The prerequisites required is available here. A summary of the steps required is as follows:

To use the ML embedded evaluation kit, you need to use a x86 Linux system (or Windows Subsystem for Linux) with a recent release of Python3 (currently Python version 3.9 or newer is required). If the Python version on your system is 3.8 or earlier, you can update the Python version by following the instructions here. You can check the version of Python3 in your system using:

```
python3 --version
```

You should then install a number of software tools using the following commands:

```
sudo apt install -y cmake make python3 git curl unzip xxd
sudo apt install -y python3-pip
python3 -m pip install pillow
```

Next, install Python virtual environment. If using python 3.9:

```
sudo apt install -y python3.9-venv
```

If using python 3.10, install Python virtual environment support using:

```
sudo apt install -y python3.10-venv
```

You also need to have a compilation tool chain, either:

- Arm Compiler for Embedded 6.16 or later, or
- GNU Arm Embedded toolchain 10.2.1 or later

Information about the Arm Compiler for Embedded can be found here. To use Arm Compiler for Embedded, you need a valid license.

If you need to install the GNU Arm Embedded toolchain, please download a suitable version from the following links:

- https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads (for versions 11.2, 11.3 12.2 or later)

- https://developer.arm.com/downloads/-/gnu-rm (for version 10.3 or earlier)

Do not install the GNU Arm Embedded toolchain 10.3.1 20210621 using "sudo apt install gcc-arm-none-eabi". There is a known issue for that release that results in a compilation error in the ML embedded evaluation kit.

After the toolchain is installed, you need to add the toolchain binary path to the search path.

For users running Windows Subsystem for Linux (WSL): Windows' paths in the $PATH variable might contain unescaped space characters (e.g. "/mnt/c/Program Files/Microsoft VS Code/bin"). This causes problems to the build script (see details here). To solve the problem, you can use one of the following workarounds:

- create a bash script that removes the offending Windows paths in the path variable, or
- add escape character \ in the path variable, or
- enclose the paths with quotes.

6.1.1.3 Check out the repository

The following commands should be used to check out the repository:

```
git clone "https://review.mlplatform.org/ml/ethos-u/ml-embedded-evaluation-kit"
cd ml-embedded-evaluation-kit
git submodule update --init
```

6.1.1.4 Compiling the default projects

Once the toolchain and utilities are installed and the search paths are in place, the compilation script can then be run:

If using GNU Arm Embedded toolchain (gcc):

```
python3 ./build_default.py
```

If using Arm Compiler 6:

```
python3 ./build_default.py --toolchain arm
```

After the compilation, the executables are located in cmake-build-XXXXXX/bin.

To run the example projects, follow the instructions in this page: Deployment.

6.1.2 Command line options for the default build script

In the previous step we executed a script (build_default.py) to build the tests for the default configuration. The script supports additional command line options. For example, to specify the configuration of the Ethos-U55 hardware to be 32 MAC, use one of the following:

```
./build_default.py --npu-config-name ethos-u55-32 # For gcc
```

or

```
./build_default.py --npu-config-name ethos-u55-32 --toolchain arm # for Arm Compiler
```

The list of valid options for the Ethos-U configurations are:

- ethos-u55-32
- ethos-u55-64
- ethos-u55-128
- ethos-u55-256
- ethos-u65-256
- ethos-u65-512

Additional command line options are described here.

6.1.3 Documentation

There are a number of documents available in the ML Embedded Evaluation Kit git repository:

- Home page
- Quick Start Guide
- Documentation
- Building the ML embedded code sample applications from sources
- Deployment
- Customizing (Implementing custom ML application)
- Arm Virtual Hardware
- FAQ
- Troubleshooting
- Memory considerations
- Testing and benchmark
- Timing adapter
- CMAKE presets

- Coding standards and guidelines
- Appendix

6.1.4 Additional material

Arm blogs

- Optimize a ML model for fast inference on Ethos-U microNPU
- Vela Compiler: The first step to deploy your NN model on the Arm Ethos-U microNPU
- Blog: Arm ML Embedded Evaluation Kit

Learning pages

- Navigate Machine Learning development with Ethos-U processors
- Build and run the Arm Machine Learning Evaluation Kit examples

6.2 ML Evaluation kit : Under the hood

In the preceding chapters, we have introduced the ML Evaluation Kit, its purpose, and its potential applications. We have also discussed the fundamental concepts of machine learning that are crucial for understanding the workings of the kit.

Further, we provide an overview of the software components of the kit, including the TensorFlow Lite Micro runtime and the Ethos-U driver.

We explore the structure of the repository, the build process, and the key steps involved in the compilation script. We also discuss the various build options available and how they can be customized according to the user's needs.

6.2.1 More about the build process

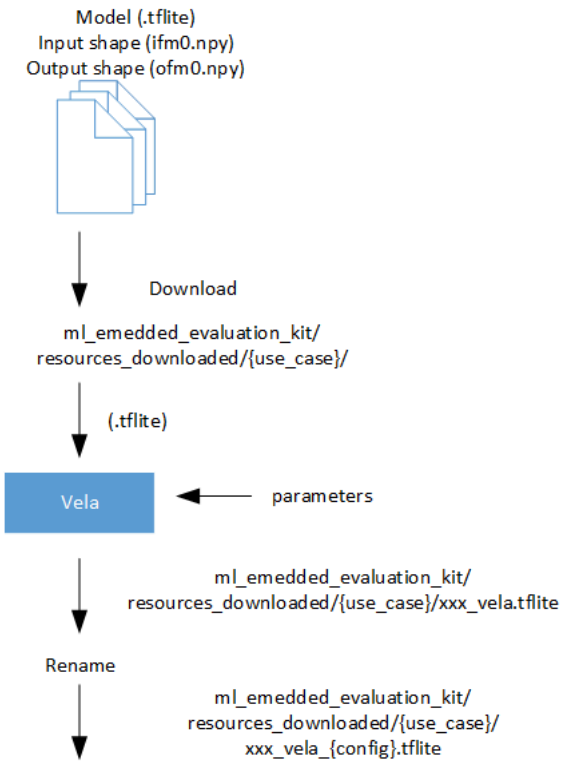
A range of materials are available to describe:

- The structure of the repository
- Build process
  - Preparing Build environment
  - Create a build directory
  - Configuring the build for the platform chosen
    - Configuring the build for MPS3 SSE-300
    - Configuring the build for MPS3 SSE-310
    - Configuring native unit-test build

The compilation script (“build_default.py”) contains the following key steps:

1. Download models for TensorFlow Lite Micro for each “use-case” and use Vela to optimize the models

Figure 6-1: Model download and optimize



2. Download additional software components, unpack them, and if needed, patch them

   The software components include:

   - FlatBuffers library from TensorFlow Lite Micro
   - kissfft (a Fast Fourier Transform library)
   - pigweed (a collection of embedded-targeted libraries)
   - gemmlowp (General Matrix Multiplication)
   - ruy (matrix multiplication library)
3. Setup build environment for each “use-case”

   This process includes setting up directories, creating the MAKEFILEs and carrying out conversion processes to enable the models and data to be included in the C++ compilation:

Input file	Conversion script
Models for TensorFlow Lite Micro (.tflite)	ml-embedded-evaluation-kit/scripts/py/gen_model_cpp.py
ML data labels (e.g. .txt)	ml-embedded-evaluation-kit/scripts/py/gen_labels_cpp.py
Audio input data (e.g. .wav)	ml-embedded-evaluation-kit/scripts/py/gen_audio_cpp.py
Image input data (e.g. .bmp)	ml-embedded-evaluation-kit/scripts/py/gen_rgb_cpp.py

The output of the converted C++ source and header files can be found in ml-embedded-evaluation-kit/cmake-build-{target}-{configs}/generated/{use-case}/

Please note that it is also possible to use the xxd utility to convert .tflite model files to byte array in C/C++.

4. Build the executables

   The source codes are compiled at this stage. If needed, additional software components (e.g. CMSIS-DSP library source) are downloaded.

6.2.2 Build options for build_default.py

The list of build options is available on this page. For example, you can use the following ETHOS_U_NPU_MEMORY_MODEL options to define the memory types used by Ethos-U NPU:

- Shared_Sram (default for Ethos-U55 NPU, available to both Ethos-U55 & Ethos-U65)
- Dedicated_Sram (default for Ethos-U65 NPU, available to Ethos-U65 only)
- Sram_Only (available for Ethos-U55 only)

If it is necessary to use different build options, instead of executing the build steps manually, it is easier to use the default build script as a starting point and modify the “cmake_command” in the script to use customized build options.

6.2.3 Software components

This section explores the software components of the Machine Learning Evaluation Kit. Key components include TensorFlow Lite Micro, which provides the runtime for the executable program, the Ethos-U Driver, and platform codes that offer flexibility to support multiple hardware targets.

6.2.3.1 TensorFlow Lite Micro

The executable program image contains the TensorFlow Lite Micro runtime, which is downloaded as part of the “build_default.py” script execution. After being downloaded, the files are located in this location: ml-embedded-evaluation-kit/dependencies/tensorflow.

The TensorFlow Lite Micro kernel provides support for Ethos-U. Some useful information about software integration and initialization is documented in ml-embedded-evaluation-kit/dependencies/tensorflow/tensorflow/lite/micro/kernels/ethos_u/README.md

In order to invoke the TensorFlow Lite Micro interpreter, the application code must include several header files and must also setup the model before invoking the micro-interpreter. To understand the bare minimal code required to setup and execute TensorFlow Lite Micro, please visit this page. It contains a good overview of TensorFlow Lite Micro low level operations.

6.2.3.2 Ethos-U Driver

The Ethos-U driver can be found in the following location: ml-embedded-evaluation-kit/dependencies/core-driver

6.2.3.3 Platform codes

The platform support codes can be a bit confusing because the evaluation kit contains files from multiple repositories, and some of them have their own platform driver codes. The ML Embedded Evaluation Kit did not use the driver code from the 3rd party repository because it needed extra flexibility to support multiple hardware targets (e.g. Corstone-300, Corstone-310). As a result, platform support codes can be found in the following locations:

Location	Note
ml-embedded-evaluation-kit/source/hal/source/platform/mps3/	This is the one that is actually used in the example projects
ml-embedded-evaluation-kit/dependencies/tensorflow/tensorflow/lite/micro/cortex_m_corstone_300	From Google TensorFlow - Not used
ml-embedded-evaluation-kit/dependencies/core-platform/targets/corstone-300	From MLplatform.org - Not used

Details of Ethos-U integration (e.g. Base address, IRQ assignment) can be found in this file: ml-embedded-evaluation-kit/source/hal/source/platform/{platform}/CMakeLists.txt. The linker scripts can be found in this location: ml-embedded-evaluation-kit/scripts/cmake/platforms/{platform}.

6.2.4 Creating custom applications in the ML Embedded Evaluation Kit

The page Implementing custom ML application contains information about creating custom applications running in the ML Embedded Evaluation Kit. It is also possible to add custom platform support.

7. CMSIS-Pack Based Machine Learning Examples

In addition to the ML Embedded Evaluation Kit, which provides a quick path for users when trying out the Ethos-U55/U65 NPUs, Arm is also working on a new set of examples based on the CMSIS-Pack, a software component packaging solution. The CMSIS-Pack based ML examples are available at this github location. The examples in this GitHub repository:

- Enables the use of the CMSIS-Pack as a software integration mechanism - which is more suitable for IDE development environments.
- Provides a greater choices of hardware support.
- Can be used in Windows environments.

Please note that the CMSIS-Pack based ML examples repository is work-in-progress and currently there are some limitation(s). These are:

- The ML “use-cases” are limited to Key Word Spotting (KWS) and Object Detection.
- Currently only Arm Compiler 6 is supported.
- The configuration of the Ethos-U NPU is not configurable.

In addition to Arm Compiler 6 and the CMSIS-Toolbox, the following tools are required:

- git, cmake, make, ninja

In a Linux system, the tools can be installed using the following command:

```
sudo apt install -y cmake make git ninja-build
```

In a Windows system, you will need to install the utilities using installers from the following websites:

Utility	website
git	https://git-scm.com/download/win
cmake	https://cmake.org/download/
GNU make	http://gnuwin32.sourceforge.net/packages/make.htm
Ninja-build	https://github.com/ninja-build/ninja/releases

It is also recommended that Visual Studio Code IDE with Keil Studio Pack Extension should be used. Alternatively, Keil Studio Cloud can also be used.

7.1 CMSIS-Toolbox

If you have already setup the CMSIS-Toolbox for your development environment, you can skip this section.

To install the CMSIS-ToolBox, please download release 2.0 or higher provided by Arm from this page. Installation details can be found on this page.

After installation, you need to ensure that:

- The CMSIS-Toolbox binaries are in the search path.
- The environment variables for the toolchain installation path is set. For example: set AC6_TOOLCHAIN_6_19_0=C:/Keil_v5/ARM/ARMCLANG/bin
- The environment variable CMSIS_PACK_ROOT is pointing to the CMSIS-Pack Root directory that stores the software packs.
- The CMSIS-Pack Root directory has been initialized. See the cpackget page for details. (e.g. cpackget init –pack-root path/to/new/pack-root https://www.keil.com/pack/index.pidx)
- The environment variable CMSIS_COMPILER_ROOT is pointing to the etc directory in the CMSIS-Toolbox (e.g. {install_path}/etc)

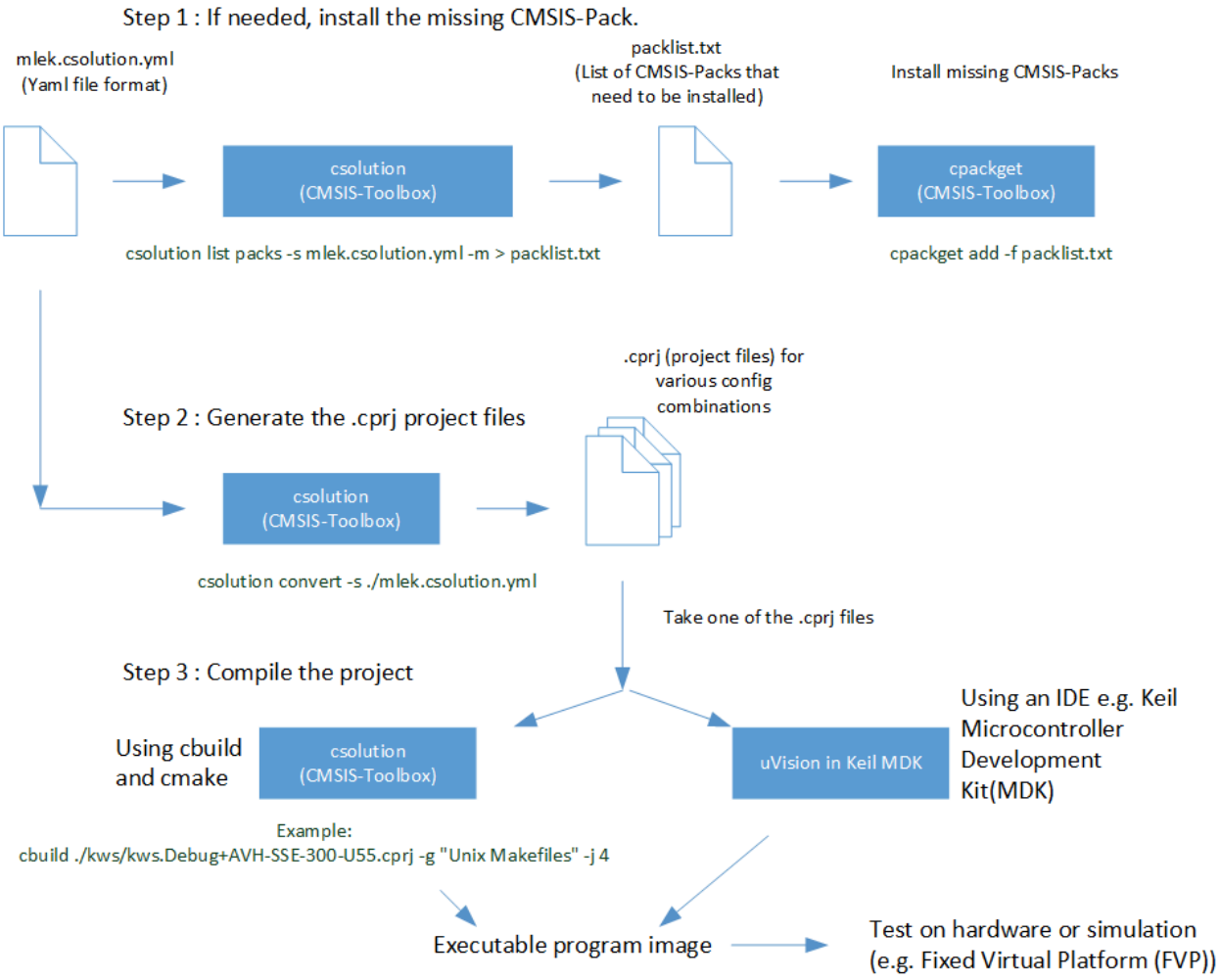
7.2 Getting Started

The workflow of the “CMSIS-Pack based ML Examples” utilizes the CMSIS-Toolbox to generate compilation setups from the Yaml files. The flow involves 3 steps:

- If some of the required software packs are not installed, install the missing CMSIS-Pack(s) .
- Generate the .cprj files from the main Yaml (.yml) file for each of the “use-cases”.
- Compile the project.

The 3 steps are illustrated as below:

Figure 7-1: Concept of software workflow in the CMSIS-Pack based ML examples



By default, step 2 generates a number of .cprj files for various configuration combinations. There are many combinations because:

- The examples support a number of ML use-cases
- The examples support a number of target platforms
- The build type can be Debug or Release

You can specify to generate just one specific combination in step 2 if needed. e.g.

```
csolution convert -s ./mlek.csolution.yml -c object_detection.Release+AVH-SSE-300-U55
```

After the executable is generated, the program image can be tested on target hardware or in a simulation environment. Further information can be found here. Please note that at this stage, MPS3 FPGA image for Cortex-M55 (AN552) is not supported because the Ethos-U configuration in the FPGA image (128 MAC/cycle) is not matching the NN model (256 MAC/cycle). However,

you can still test the generated application using Arm Virtual Hardware (AVH) or Fixed Virtual Platform (FVP).


Note

Unlike the ML Embedded Evaluation Kit, the ML models and input samples have already been converted to C++ (e.g. KWS, object detection). There is therefore no need to execute the Vela Compiler. Because of this, the Ethos-U configurations supported by the ML models are fixed.

7.3 TensorFlow Lite Micro CMSIS Components

The following section will help to integrate TensorFlow Lite Micro (TFLM), Ethos-U driver and the model(s) into your project. Skip the first paragraph if you already have an existing project.

7.3.1 Add software components

The TensorFlow Lite Micro software packs are available from the CMSIS-Pack web page:

```
    - pack: tensorflow::flatbuffers
    - pack: tensorflow::gemmlowp
    - pack: tensorflow::kissfft
    - pack: tensorflow::ruy
    - pack: tensorflow::tensorflow-lite-micro
```

These packs are build from sources that are maintained and versioned by Arm on mlplatform.org.

Add components as shown below, for example by using the Manage Software Components in Keil Studio.

- The most important component is:

```
    - component: Machine Learning:TensorFlow:Kernel&Ethos-U
```

- This component with the variant `Ethos-U` requests several other software components. The dependencies can be resolved in the IDE:

```
  - component: Arm::Machine Learning:NPU Support:Ethos-U Driver&Generic U55  # see
note
  - component: ARM::CMSIS:DSP&Source
  - component: ARM::CMSIS:NN Lib
  - component: tensorflow::Data Exchange:Serialization:flatbuffers
  - component: tensorflow::Data Processing:Math:gemmlowp fixed-point
  - component: tensorflow::Data Processing:Math:kissfft
  - component: tensorflow::Data Processing:Math:ruy
  - component: tensorflow::Machine Learning:TensorFlow:Kernel Utils
```

Note that the Ethos-U Driver default variant is `Generic U55`. Depending on your target device, a different driver variant may be available from the device vendor, and should be selected.

7.3.2 Adding the ML model to your project

Generally there is two options to store your ML model on the embedded target. Either on an existing filesystem or compile it into the firmware image and flash it together with the application.

7.3.2.1 Loading from a filesystem

The TensorFlow library comes with the capability to interpret tflite files as they are stored on filesystem. This can be useful if you need to need to handle multiple models, and want to update them independently from the application.

The Keil Middleware File System provides a set of APIs to interact with a file system on a storage device like an SD card or USB flash drive. Here's a simple code snippet showing how you might load a tflite file from an SD card:

```
#include "rl_fs.h"
...
FILE *f;
char *buffer;
size_t buffer_size;
// Open the file for reading
f = fopen("/sdcard/my_model_vela.tflite", "r");
if (f == NULL) {
    // Error handling
}
// Get the size of the file and allocate a buffer
fseek(f, 0, SEEK_END);
buffer_size = ftell(f);
rewind(f)
buffer = malloc(buffer_size + 1);  // +1 for the null terminator
if (buffer == NULL) {
    // Error handling
    fclose(f);
    return;
}
// Read the file into the buffer
size_t num = fread(buffer, 1, buffer_size, f);
if (num != buffer_size) {
    printf("Failed to read file\n");
} else {
    buffer[buffer_size] = '\0';  // Null-terminate the string
    printf("File contents: %s\n", buffer);
fclose(f);
const tflite::Model* model = ::tflite::GetModel(buffer);
free(buffer);
...
```

Loading model files will require much RAM and is not suitable for all system designs. A very typical scenario for this will be a test system based on Arm Virtual Hardware, where you load many different model variants for profiling runs. Even if a file system is available on your hardware target, you may want to store the model in ROM alongside the application, as explained in the next paragraph.

7.3.2.2 Storing in firmware image

To compile the content of a tflite file into your firmware image, it needs to be represented as an array in C language syntax. A hexdump utility like xxd can help to convert the binary tflite file into a header file to include in your project.

```
xxd -i my_model_vela.tflite my_network_model.h
```

To make sure that the data is stored in ROM memory and starts at a 16 Byte alignment boundary, the definition should be changed to:

```
const unsigned char network_model __ALIGNED(16) {
  ...
}
```

In the application code, include my_network_model.h and load the model with TensorFlow:

```
#include "my_network_model.h"
...
const tflite::Model* model = ::tflite::GetModel(network_model);
...
```

7.3.3 Using the TensorFlow Lite Micro API

The following section explains the usage of the TensorFlow Lite Micro C++ API for a typical implementation. Input tensors, preprocessing, and output interpretation depends on your model and may be different.

7.3.3.1 Initialization of Ethos-U

The function `ethosu_init` initializes the Ethos-U NPU. In the ML Embedded Evaluation Kit, this function is executed during hardware initialization in the platform initialization code:

- `int platform_init()` in source/hal/source/platform/mps3/source/platform_drivers.c calls:
  - `int arm_ethosu_npu_init()` in source/hal/source/components/npu/ethosu_npu_init.c.
  - `int ethosu_init()` in ethosu_driver.c

7.3.3.2 Accessing Ethos-U via TensorFlow Lite runtime

The Ethos-U support is integrated within the TensorFlow Lite runtime and therefore the application code only calls the TensorFlow Lite interpreter to take advantage of the Ethos-U NPU.

```
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/tflite_bridge/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
```

```
// Include your model data. Replace this with the header file for your model.
#include "model.h"
// Define the number of elements in the input tensor
#define INPUT_SIZE 300*300
using MyOpResolver = tflite::MicroMutableOpResolver<10>;
// Replace this with your model's input and output tensor sizes
const int tensor_arena_size = 2 * 1024;
uint8_t tensor_arena[tensor_arena_size] __align(16);
void RunModel(int8_t* input_data) {
    tflite::MicroErrorReporter micro_error_reporter;
    tflite::ErrorReporter* error_reporter = &micro_error_reporter;
    const tflite::Model* model = ::tflite::GetModel(g_model);
    MyOpResolver op_resolver;
    // Register your model's operations here
    tflite::MicroInterpreter interpreter(model, op_resolver, tensor_arena,
                                         tensor_arena_size, error_reporter);
    interpreter.AllocateTensors();
    TfLiteTensor* input = interpreter.input(0);
    // Add checks for input tensor properties here if needed
    // Copy image data to model's input tensor
    for (int i = 0; i < INPUT_SIZE; ++i) {
        input->data.int8[i] = input_data[i];
    }
    TfLiteStatus invoke_status = interpreter.Invoke();
    // Add checks for successful inference here if needed
    TfLiteTensor* output = interpreter.output(0);
    // Add checks for output tensor properties here if needed
    // Process your output value here
    // For example, SSD models typically produce an array of bounding boxes
}
```

The template requires C++17 as the minimum language standard, which complies to the standard used by TensorFlow.

If you need to create your own model, a good starting point into the workflow is found in the TensorFlow Guide

Next step might be the Profiling and Optimization of ML Models. Arm provides many powerful tools to test and improve a models performance running on Ethos-U NPUs.

8. Profiling and Optimization of ML Models

The memory requirement and code size of a TensorFlow Lite (TFLite) Micro model can be determined by analyzing the model's parameters, layers, and the hardware platform on which it will run. Here are the key steps to estimate the memory requirement and code size:

- Model Analysis: Begin by examining the structure of the TFLite Micro model itself. This includes the number and type of layers, the input and output shapes, and the size of the model parameters. The model parameters consist of the weights and biases associated with each layer. A helpful tool can be visualization, like that provided by Netron App
- TFLite Micro Profiler: TFLite Micro provides a profiler tool that allows you to measure memory usage and code size. You can find example code and instructions in the TFLite Micro documentation.
- Ethos Performance Counters to analyze code and memory usage.
- Memory Estimation Functions: TensorFlow Lite for Microcontrollers provides memory estimation functions that allow you to estimate the memory requirement programmatically. These functions help you calculate memory usage based on model parameters and tensor shapes.
- Compilers and Linkers: Arm Compiler, LLVM, GCC Linker Code/object size reports.
- Static Analysis Tools: Static analysis tools can provide information about the code size and memory usage of your TFLite Micro model without running it. (CI)

8.1 Ethos-U Vela Optimizations

Many optimizations can be made during the compilation step with Vela. Refer to this manual page for latest information.

8.2 Operator Mapping/Usage

Running vela with the `--verbose-performance` parameter will output a table with details of the TensorFlow operator usage in your model. These data values are estimates and are not cycle accurate numbers based on running the inference on silicon.

```
############################################################################
Performance for NPU Subgraph _split_1
TFLite_operator      NNG Operator         SRAM Usage  Peak%  Op Cycles Network%
   NPU    SRAM AC    DRAM AC OnFlash AC OffFlash AC  MAC Count Network%  Util% Name
------------------- -------------------- ---------- ------ ---------- --------
 ---------- ---------- ---------- ---------- ---------- ---------- -------- ------
 --------------------
CONV_2D              Conv2DBias               629616  86.18    1889913
 46.80    1889913      21504          0          0           0   99090432
  49.99  20.48 ResNet18/activation_32/Relu;ResNet18/batch_normalization_32/
FusedBatchNormV3;ResNet18/conv2d_38/BiasAdd/ReadVariableOp/resource;ResNet18/
conv2d_38/BiasAdd;ResNet18/conv2d_39/Conv2D;ResNet18/conv2d_38/Conv2D1
CONV_2D              Conv2DBias               730624 100.00    2127584    52.69
  2127584      21504          0          0           0   99090432    49.99  18.19
```

```
 ResNet18/batch_normalization_33/FusedBatchNormV3;ResNet18/conv2d_39/BiasAdd/
ReadVariableOp/resource;ResNet18/conv2d_39/BiasAdd;ResNet18/conv2d_39/Conv2D
ADD                 Add                       43008   5.89      16128     0.40
    16128       8064          0          0          0          0     0.00   0.00
 ResNet18/activation_33/Relu;ResNet18/add_15/add
AVERAGE_POOL_2D     AvgPool                   27648   3.78       4224     0.10
     2200       4224          0          0          0      24576     0.01   2.27
 ResNet18/average_pooling2d_1/AvgPool
```

Cycles and Network% give a good indication which layer is compute and memory intensive.

MAC Count of 0 gives an indication of Operators that are not off-loaded to Ethos-U NPU. These will still be executed optimized by CMSIS-NN typically, but consume CPU time. If this is the case for a majority of your network, the model used is not suitable for Ethos-U.

8.3 MLIA guided optimizations (Experimental)

The ML Inference Advisor (MLIA) is a tool designed to assist AI developers in designing and optimizing neural network models for efficient inference on Arm® targets. It is ideal for a quick evaluation of whether a NN maps fully to Ethos-U. In addition, it enables performance analysis and provides actionable advice early in the model development cycle. The advice can cover supported operators, performance analysis, and suggestions for model optimization, such as pruning and clustering.

MLIA works with sub-commands

- check to perform compatibility or performance checks on the model, and “optimize” to apply specified optimizations.
- optimize to apply specified optimizations. Optimization is only available for Keras model format (h5), not tflite.

When MLIA is setup with a performance model such as the Corstone Fixed Virtual Platform (FVP), it provides performance information of the inference. In the case where the Corstone-300 FVP is used and when all of the inference operations are running on the Ethos-U55, the result value is within +/-10% accuracy. The error margin can be higher when another performance model, such as the Corstone-310 FVP, is used. Please note that MLIA is more like a wrapper of tools/components and does not include its own performance model (hence an FVP is needed for obtaining performance data). In contrast, Vela includes a performance estimator, but the performance estimation in Vela is not accuracy - it is fine for relative comparison with other Vela performance estimate values, but should not be used in situations where accuracy is needed.

For an in-depth evaluation, for example, to understand the full software flow and experiment with memory layout arrangements, do even to prototype a full application with pre-processing/post-processing, then ML Embedded Evaluation Kit or the CMSIS-Pack based example could be more suitable.

8.4 Ethos-U Performance Profiling

The content for this section is currently under development and will be added in the near future.

9. MLOps Systems

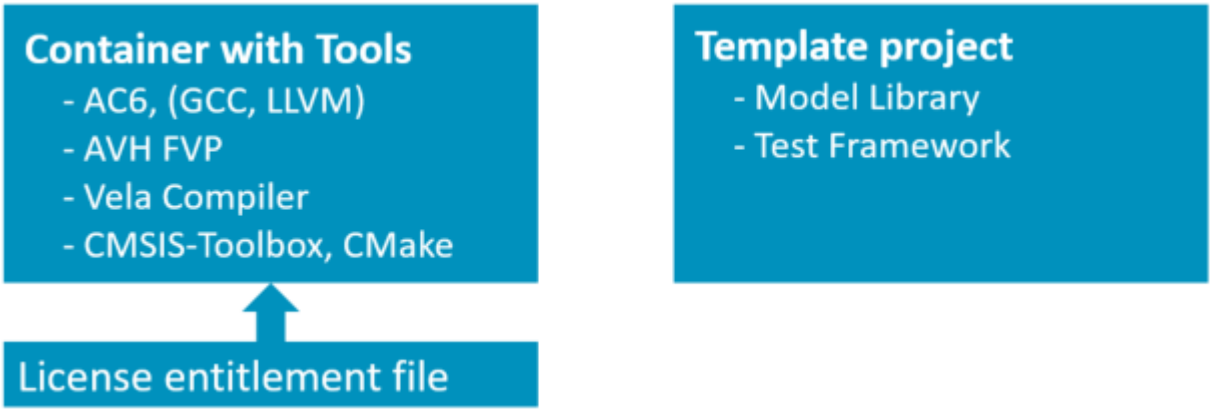
Arm provides a set of foundation tools and software components to enable MLOps systems and the overall development flow for machine learning applications. Arm is also working with several MLOps partners to integrate these components into established MLOps systems. The Arm Partner Ecosystem Catalog provides a searchable list of AI partners.

9.1 Tools and Project Templates

The section Overall Development Process describes the MLOps process steps for developing machine learning applications.

Arm provides Tools and Template Projects outlined in the picture below to optimize MLOps process steps for targeting Cortex-M and Ethos-U processors.

Figure 9-1: MLOps Components



An MLOps system typically uses a container with all required development tools.

The github repository Arm-Software/AVH-MLOps contains:

- Setup of MLOps foundation tools, exemplified by using Docker. Most of the tools are downloaded from the Arm Tools Artifactory.
- Template project that generates a ML Model library and verifies execution using Arm Virtual Hardware (AVH-FVP). It supports all relevant Cortex-M and Ethos-U targets and can be used with different toolchains.
- GitHub actions that exemplify typical MLOps operations.
- Software Pack delivery of the ML Model library using Open-CMSIS-Pack technology.

9.2 License Entitlement

To activate the Arm development tools in an MLOps system a license is required. Contact Arm for further details and on how to become an Arm MLOps partner.

These MLOps system license is activated externally of the container using the command below. It is required to run the `activate` command once every 24 hours to keep the license up-to-date.

```
armlm activate --code <license-code-here> --as-user arm_mlops --to-file
 arm_mlops_license
```

The `activate` command generates a license files that is imported into the container with the command `armlm import`. Refer to User-based Licensing User Guide for more information.

9.3 Example Projects

The example projects use the `MLOps.csolution.yml` file in CMSIS-Toolbox format. The projects show how to create a library with a trained ML model and how to evaluate the performance.

The `MLOps.csolution.yml` file uses following sub-projects:

- `ML_Model.cproject.yml`: creates a library of the trained ML model.
- `ML_Test.cproject.yml`: creates a model evaluation test using the ML model library. It reports timing using AVH in combination with CMSIS-View and the `eventlist` utility.

`MLOps.csolution.yml` supports various compilers (AC6, GCC, LLVM) and defines various `target-types` and `build-types` that represent different processors as shown in the list below.

target-type	Selects the target processor
`+CM0`	Cortex-M0
`+CM0plus`	Cortex-M0+
`+CM3`	Cortex-M3
`+CM4`	Cortex-M4
`+CM4_FP`	Cortex-M4 with FPU
`+CM7`	Cortex-M7
`+CM23`	Cortex-M23
`+CM33`	Cortex-M33
`+CM55`	Cortex-M55
`+CM85`	Cortex-M85
`+CM55_Ethos`	Cortex-M55 with Ethos-U
`+CM85_Ethos`	Cortex-M85 with Ethos-U

build-type	Selects the code optimization
`.speed`	Optimize for speed

build-type	Selects the code optimization
`.size`	Optimize for size
`.balanced`	Balanced optimization for speed and size

These `target-type` and `build-type` definitions can be used together with the `--toolchain` option to create libraries for the various processors and compiler toolchains as shown in the CMSIS-Toolbox command line example below. For example, this `cbuild` command translates for a Cortex-M7 processor with code size optimization using the GCC toolchain:

```
cbuild MLOps.csolution.yml --context +CM7.size --toolchain GCC
```

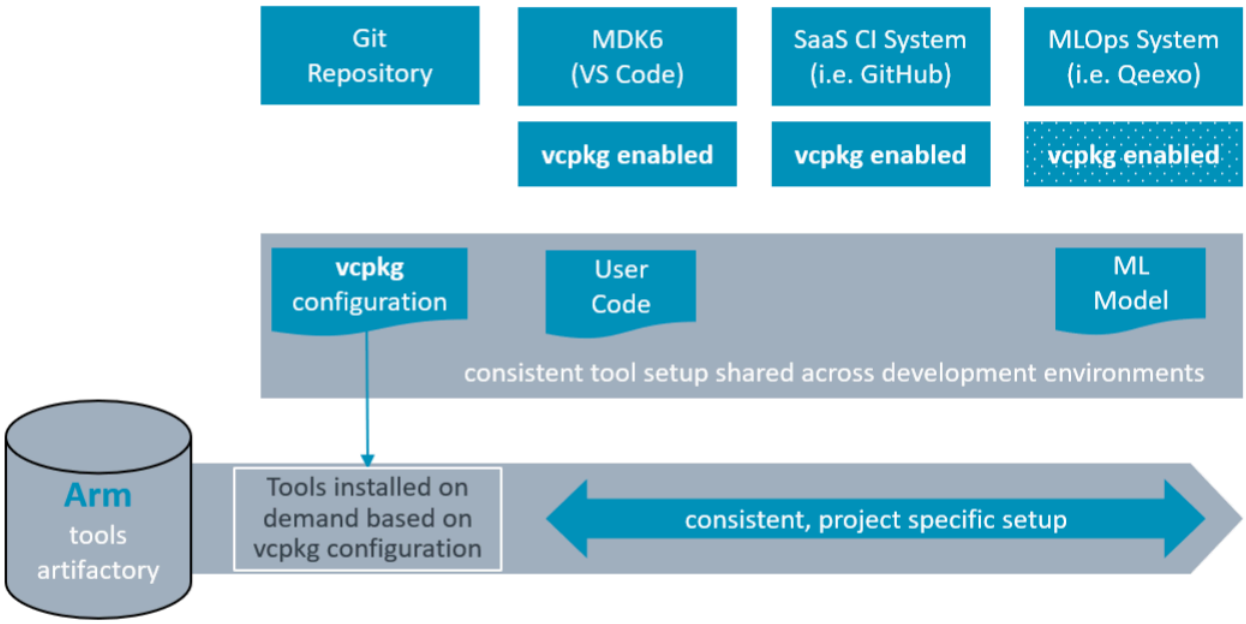
Example projects show how to run the test process using the AVH VSI interfaces for Audio, SDS, Video.

A CMSIS-Pack template shows how to deliver the Model Library to IDEs for integrating the ML model in embedded projects. This pack contain source code templates that ease the integration into application program.

9.4 vcpkg

Tools may be also managed using the Microsoft tool vcpkg. Refer to "Install tools on the command line using vcpkg" for further details.

Figure 9-2: Arm Tools Artifactory



Currently the vcpgk process is experimental and we therefore recommend to download and install the tools for MLOps systems with native OS commands.

10. Resources for Ethos-U

The following resources are available for Ethos-U:

- Product pages
- Product document
- Software and examples
- Other materials
- Partner's solution

10.1 Product pages

The following resources are available:

Product	Resource
Ethos-U55	https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u55
	https://developer.arm.com/Processors/Ethos-U55
Ethos-U65	https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u65
	https://developer.arm.com/Processors/Ethos-U65

10.2 Product document

The following resources are available:

Product	Resource
Ethos-U55	Technical Reference Manual
Ethos-U55	Product Brief
Ethos-U65	Technical Reference Manual
Ethos-U65	Product Brief
Ethos-U55/U65	Arm Ethos-U NPU Application development overview

10.3 Software and examples

Open-source software components and documentation for developing Ethos-U NPU software can be found in:

- https://review.mlplatform.org/plugins/gitiles/ml/ethos-u/ethos-u

In addition, the following resources are available:

- Vela compiler
- TensorFlow Lite for Microcontrollers
- Arm Model Zoo
- ML Inference Advisor (MLIA)
- Ethos-U ML Embedded Evaluation kit
- Using the Arm Corstone-300 with Arm Cortex-M55 and Arm Ethos-U55 NPU - Jupyter notebook
- Ethos-U Core Platform
- CMSIS-Pack based Machine Learning Examples
- CMSIS-NN
- Additional ML examples for Arm processors
- learn.arm.com: Navigate Machine Learning development with Ethos-U processors
- learn.arm.com: Build and run the Arm Machine Learning Evaluation Kit examples
- Running TVM on bare metal Arm(R) Cortex(R)-M55 CPU, Ethos(TM)-U55 NPU and CMSIS-NN
  - Documentation: Running TVM on bare metal Arm(R) Cortex(R)-M55 CPU and Ethos(TM)-U55 NPU with CMSIS-NN
- Benefit of pruning and clustering a neural network for before deploying on Arm Ethos-U NPU

10.4 Other materials

The following resources are available:

Product	Resource
Ethos-U55	Technical Overview of Ethos-U55 - video
Ethos-U55	Running Machine Learning on Arm's Ethos-U55 NPU - slide
Cortex-M55 + Ethos-U55	Arm M55 and U55 Performance Optimization for Edge-based Audio and Machine Learning Applications
TensorFlow Lite Micro	TensorFlow Lite Micro low level operations
ML model optimization	Optimize a ML model for fast inference on Ethos-U microNPU
Vela compiler	Vela Compiler: The first step to deploy your NN model on the Arm Ethos-U microNPU
Arm ML Embedded Evaluation Kit	Blog: Arm ML Embedded Evaluation Kit
Using uTVM with Ethos-U	tinyML Summit 2023: Arm Ethos-U support in TVM ML framework - video

10.5 Partner's solution

The following resources are available:

Partner	Product/solution
Sensory	Sensory Speech Technologies on Arm IP
Arcturus	Energy-Efficient ML Vision App Development with Ethos-U65/i.MX 93