



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

Version 1.2

Non-Confidential

Copyright © 2023–2025 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

109267_0102_01_en

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

Copyright © 2023–2025 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0102-01	12 May 2025	Non-Confidential	Minor updates.
0101-01	9 April 2024	Non-Confidential	Added information about Cortex-M52, Ethos-U85, and Corstone-315.
0100-04	7 November 2023	Non-Confidential	Initial release

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly 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 this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm's trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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.....	8
1.1 Target audience.....	9
1.2 Machine Learning on edge devices.....	9
1.2.1 ML compute requirements.....	10
1.3 Overview of the ML development process.....	11
1.4 Tools and software for ML development.....	14
1.5 Targeting Ethos-U NPUs.....	15
2. ML software development for Arm Cortex-M processors.....	17
2.1 ML software framework options.....	18
2.2 Example software development flow using TFLM.....	19
2.2.1 Create a TensorFlow Lite model file.....	20
2.2.2 Convert the model file to a C/C++ header file.....	20
2.2.3 Identify the inputs and outputs of the NN model.....	21
2.2.4 Integrate the TFLM runtime library.....	22
2.2.5 Integrate the inference functions.....	24
2.2.6 Run the inference and process the results.....	25
2.3 Re-training an ML model.....	26
2.4 Further information.....	26
3. Arm Ethos-U NPU.....	27
3.1 Ethos-U hardware architecture.....	28
3.1.1 Ethos-U performance configuration.....	29
3.1.2 Ethos-U bus manager interfaces.....	30
3.1.3 Differences between Ethos-U55, Ethos-U65 and Ethos-U85.....	31
3.1.4 Power, security, and performance analysis.....	31
3.2 Ethos-U system integration.....	32
3.2.1 Ethos-U integration in a Cortex-M system.....	32
3.2.2 Ethos-U integration with an Ethos-U subsystem.....	33
3.2.3 Ethos-U85 system integration.....	35
3.3 Corstone reference designs.....	35
3.4 ML software support for Ethos-U.....	38
3.4.1 Ethos-U custom operators.....	39

3.4.2 ML software for microcontrollers with Cortex-M and Ethos-U NPU.....	40
3.4.3 ML software for ML subsystems in a larger SoC.....	41
3.5 Software architecture scenarios and use cases.....	42
3.6 Additional software and tools for Ethos-U.....	42
3.7 Porting Ethos-U software to a new hardware platform.....	43
3.7.1 Security configuration for Ethos-U in a TrustZone system.....	43
3.7.2 An example of Ethos-U initialization.....	44
3.7.3 Software integration for the Ethos-U micro NPU in custom designs.....	45
3.7.4 Linker script design.....	46
3.8 Customizing the Ethos-U driver and RTOS integration.....	48
3.8.1 Putting the processor to sleep while the Ethos-U NPU is running.....	52
3.8.2 Adding RTOS support.....	53
3.8.3 Ethos-U driver configuration.....	54
4. Tool support for the Arm Ethos-U NPU.....	55
4.1 Ethos-U Vela compiler.....	56
4.1.1 Requirements.....	57
4.1.2 Installation.....	57
4.1.3 Usage.....	58
4.1.4 Command examples.....	60
4.1.5 Optimization considerations for the Vela compiler.....	61
4.2 Machine Learning Inference Advisor.....	64
4.2.1 Requirements.....	65
4.2.2 Installation.....	65
4.2.3 Usage.....	65
4.2.4 Command examples.....	67
4.3 Arm Virtual Hardware.....	67
4.4 SDS Framework.....	68
4.4.1 SDS Recorder Interface.....	68
4.4.2 SDS Metadata.....	69
4.4.3 SDS Utilities.....	72
4.4.4 SDS Playback.....	72
5. The Arm ML Zoo.....	74
5.1 Integrating an Arm ML-Zoo model.....	75
6. ML Embedded Evaluation Kit.....	76

6.1 Getting started with the ML Embedded Evaluation Kit.....	76
6.1.1 Supported platforms.....	76
6.1.2 System and software requirements.....	77
6.1.3 Check out the repository.....	78
6.1.4 Compile the default projects.....	78
6.1.5 Additional resources.....	79
6.2 Beyond the basics.....	80
6.2.1 The build process.....	80
6.2.2 Build options for build_default.py.....	83
6.2.3 Software components.....	84
6.2.4 Creating custom applications with the ML Embedded Evaluation Kit.....	85
7. CMSIS-Pack based ML examples.....	86
7.1 Prerequisites.....	86
7.2 Compiling the CMSIS-Pack based ML examples.....	87
7.3 Using TFLM CMSIS-Packs in your own project.....	89
7.3.1 Add the TFLM software components.....	89
7.3.2 Add the ML model to your project.....	90
7.3.3 Use the TFLM API.....	92
8. Profiling and optimizing ML models.....	94
8.1 Ethos-U Vela optimizations.....	94
8.2 Operator mapping and usage.....	95
8.3 MLIA guided optimizations (Experimental).....	95
8.4 Ethos-U performance profiling.....	96
9. MLOps systems.....	97
9.1 License activation.....	98
9.2 Example projects.....	98
9.3 vcpkg.....	99
10. Resources for Ethos-U.....	101
10.1 Product pages.....	101
10.2 Product document.....	101
10.3 Software and examples.....	102
10.4 Other resources.....	102
10.5 Partner solutions.....	103

1. Overview

Thank you for using [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, Arm offers hardware IP, tools, and software that make product development easy and productive. In addition, Arm provides supporting material and collaborates with many AI partners to complement our solution, for example with optimized ML models, MLOps integrations, and evaluation boards.

This guide contains the following information:

- [Overview](#) provides an overview of the ML development process, introducing the Arm technology and products that support the entire ML development workflow from ML model training through to debugging on hardware.
- [ML software development for Arm Cortex-M processors](#) describes the concepts involved in developing ML software for resource-constrained systems, with an example using TensorFlow Lite for Microcontrollers (TFLM).
- [Arm Ethos-U NPU](#) provides information about the different Ethos-U processors, including hardware and software design considerations.
- [Tool Support for the Arm Ethos-U NPU](#) describes the Vela compiler that transforms ML models for execution on Ethos-U NPUs, as well as the ML Interference Advisor, Arm Virtual Hardware, and SDS-Framework for analysis, verification, and training.
- [The Arm ML Zoo](#) introduces a repository containing pre-trained models for various types of applications. This section also explains how to use the models in the repository.
- [ML Embedded Evaluation Kit](#) provides ML examples for a range of use cases that help you to create your own applications for systems based on the Cortex-M CPU and Ethos-U NPU.
- [CMSIS-Pack based ML examples](#) demonstrates ML software integration using software components in the form of CMSIS-Packs, and using CMSIS-Toolbox to manage the build environment.
- [Profiling and optimizing ML models](#) describes how to analyze and optimize the execution of ML models 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 select 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.

1.1 Target audience

This guide assumes some basic knowledge about Cortex-M software development. It is written for the following audiences:

- Embedded Developers that want to use 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 support the Ethos-U NPU processor series and need to integrate the various development tools into their development flows.
- Data scientists that analyze data to develop new ML models and need software tools to gather statistics about model performance.

1.2 Machine Learning on edge devices

Implementing ML on edge devices enables a new range of applications. Examples of these applications include the following:

Predictive maintenance

Sensors in a system identify likely failures, allowing for proactive maintenance to prevent downtime.

Speech recognition

Use natural language to interact with devices.

Image detection in factory automation

Improve efficiency, reduce errors, and increase safety in manufacturing environments.

Medical diagnosis

Assist in medical treatment, helping to design personalized treatment plans.

Computer vision

Enable robots to perceive and understand their environment by recognizing objects, detecting obstacles, and tracking people.

These are just a few examples. The possibilities of ML are vast, and the technology is constantly evolving making it possible to innovate in many new areas.

Typically, ML models require a large quantity of reliable data for the models to perform accurate predictions. When training an ML model, engineers need to collect a large and representative sample of data. This training data could be a collection of images, sensor data, or data collected from individual users of a device.

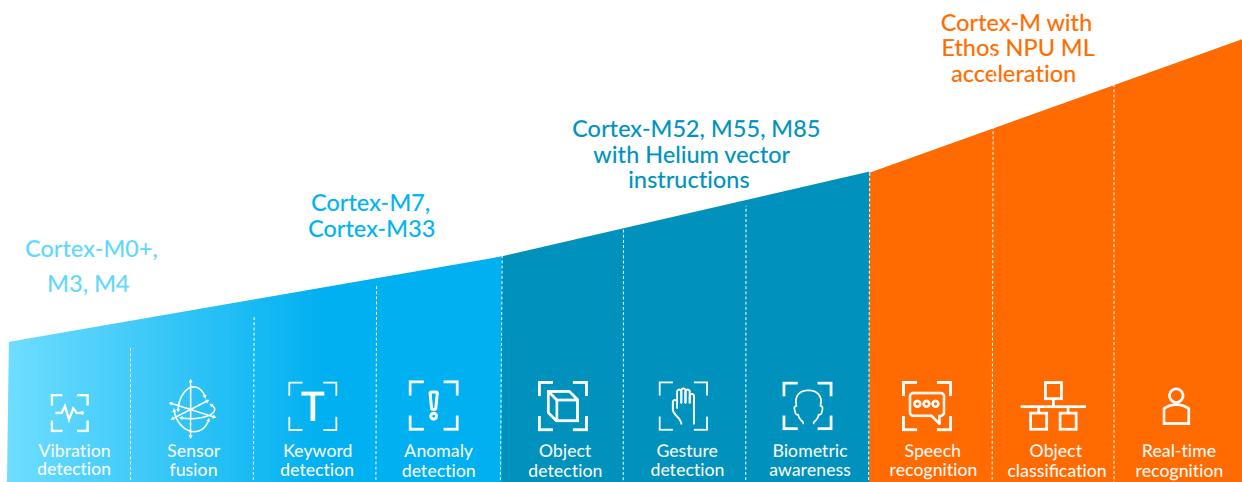
Today, AI and ML algorithms that operate on data from IoT endpoint devices frequently execute on cloud servers. However, to meet the real-time requirements of embedded systems, the actual AI algorithm must execute on the edge device.

1.2.1 ML compute requirements

The compute requirements for different 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.

Arm therefore offers a broad range of optimized processors targeting machine learning applications on edge devices, as shown in the following diagram:

Figure 1-1: Arm ML processor portfolio



- Even the smallest Arm processor, the Cortex-M0/M0+ processor, can execute simple ML algorithms. For example, you could use the Cortex-M0/M0+ processor to implement a predictive maintenance system that monitors data from a single sensor.
- Starting with the Cortex-M4, Arm processors provide hardware floating-point arithmetic and SIMD instructions that can accelerate DSP and simple ML algorithms. For example, these processors can run applications that use sensor fusion to merge data from multiple sensors.
- The Cortex-M55 and Cortex-M85 processors extend the architecture with Helium vector instructions that enable more complex ML algorithms. For example, these processors can run applications that use speech keyword spotting or object and anomaly detection.
- Ethos-U is a family of microNPUs that enables extremely low-power ML inference at the endpoint. Ethos-U operates in combination with an Arm Cortex processor and provides an enormous increase in ML performance.

For more demanding applications, the Arm ML processor portfolio also includes Mali graphics processors and the Cortex-A processors. However, these processors are outside the scope of this guide. This guide focuses on the development flow for tiny edge devices that use a single-core Cortex-M processor optionally paired with an Ethos-U NPU.

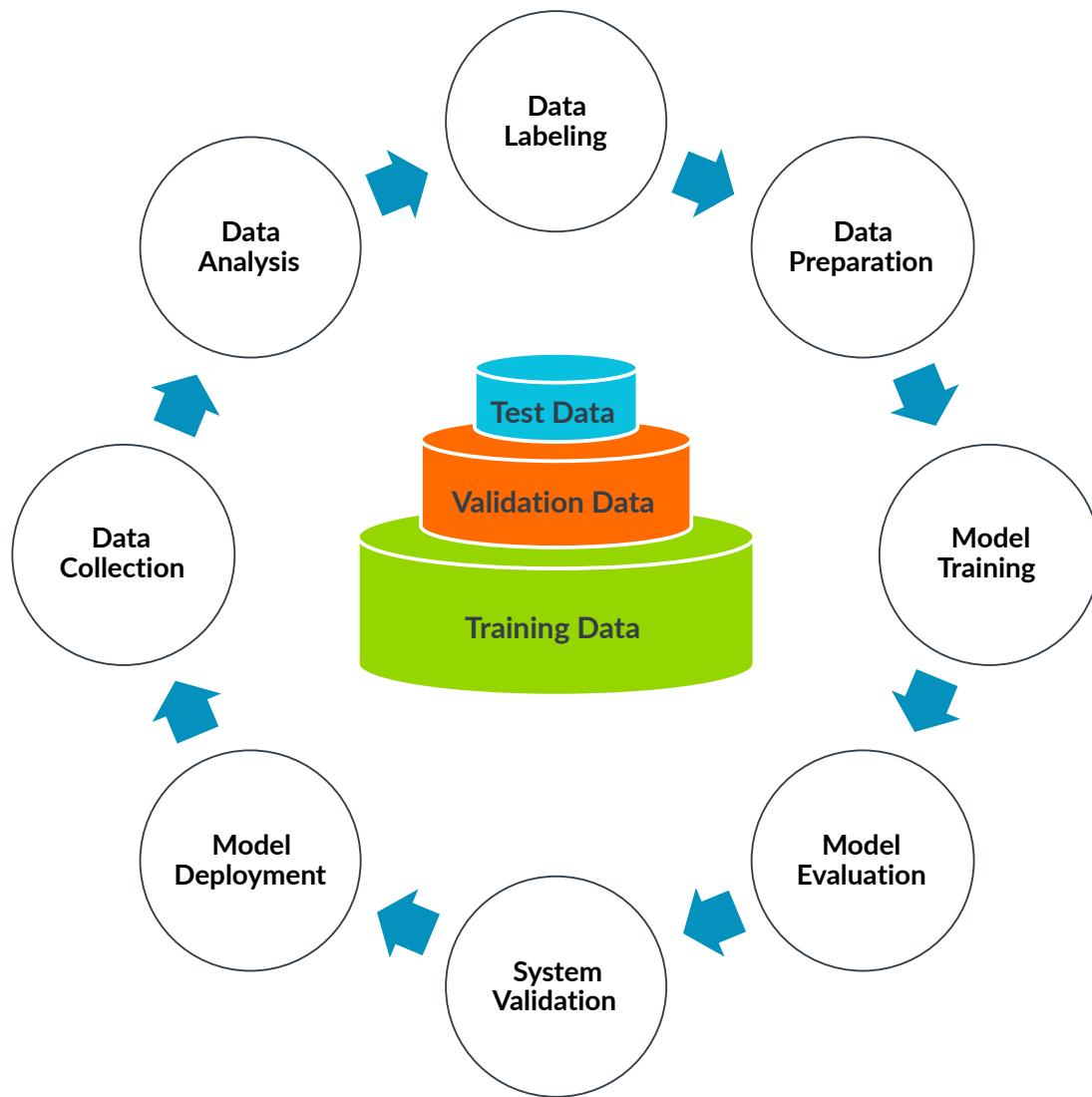
1.3 Overview of the ML development process

The software and system design of an embedded IoT and ML device consists of two parts:

- The classic embedded IoT software part. This part 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 using Software-as-a-Service (SaaS) cloud environments that are specialized for ML algorithm development.

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

Figure 1-2: MLOps process steps



The MLOps process contains the following steps:

Data collection

Data collection is the foundation of an ML project. The ML model must have enough data to learn from, the data must cover as many scenarios as possible, and the data must be accurate. The quantity and quality of the data is critical for the performance of the model.

Data analysis

Data analysis requires an understanding of the different scenarios that are represented in the data collection. Data may need to be cropped or cleaned if the collected data contains a mixture of scenarios.

Data labelling

Data labelling is the annotation of the collected and cleaned data. For example, data for a fitness tracker might be labelled with “walk”, “run”, and “rest” to describe the different activities that are represented.

Data preparation

Data preparation makes the collected data available for training the model. For example, you might separate data into training data, validation data, and test data for smoke testing. Training data is typically the largest data set.

Model training

Model training performs the training of the ML model. Training is the process by which a machine learning algorithm is fed with a training dataset from which it can learn.

Model evaluation

For any given ML problem, there are several different algorithms to choose from. Evaluation of these different models is an iterative process to choose the best ML algorithm for the problem.

System validation

System validation tests the ML algorithm with the model data, running on the final target system. Verification of the ML algorithm might be performed using a reduced set of validation data.

Model deployment

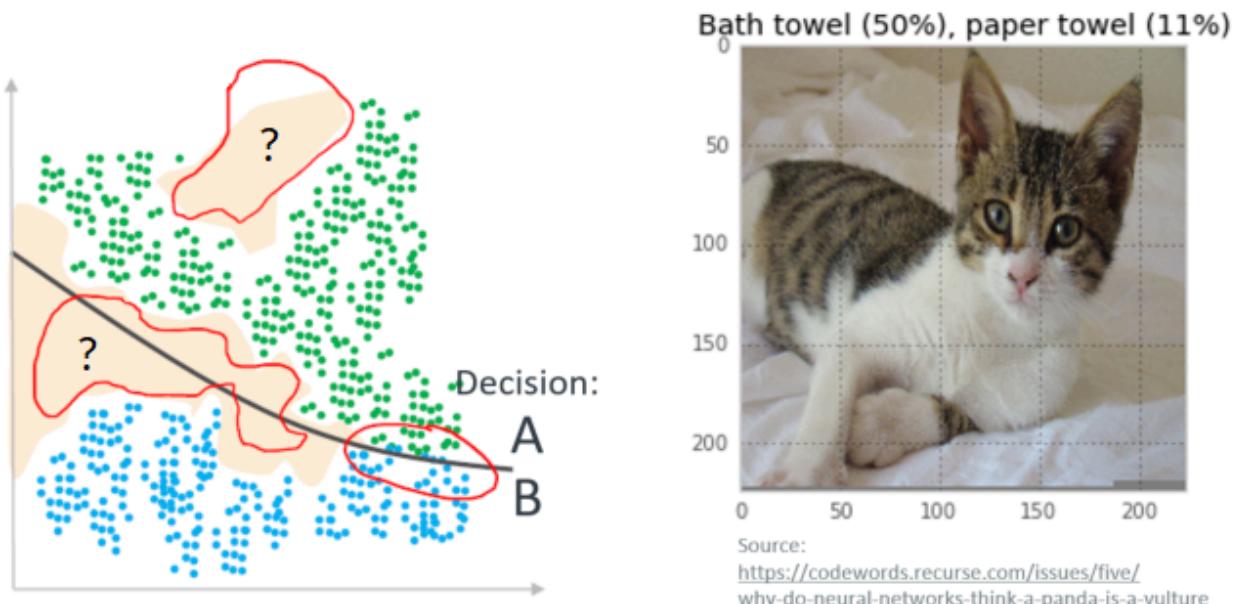
Deployment is the integration of the ML algorithm with the model data into the final target system, for example an embedded IoT application.

Machine learning models are typically tested and developed in isolated systems. Training of the ML model mostly takes place in the cloud, because training needs both an extensive data set and high compute power. Once training is complete, the algorithm using the ML model can then execute directly on the IoT endpoint device.

Just as humans learn and improve upon past experiences, ML 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 exists. Learning means therefore that ML algorithms are re-trained based on new data that delivers additional information.

For example, if a picture recognition application has never seen a picture of a cat, it cannot be correctly qualified. The figure below shows the result of this missing training data. It is therefore expected that IoT endpoint systems that incorporate AI and ML technology require periodic updates.

Figure 1-3: Missing training data



1.4 Tools and software for ML development

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

Step	Tool	Description
Data collection	Keil MDK	For classic embedded IoT software development targeting Cortex-M processors.
Data collection	SDS Framework	For data capturing, optionally combined with MDK Middleware to interface with Networks, USB, or File System.
Model evaluation	Arm Compiler for Embedded	Commercial C/C++ Compiler for all Cortex-M processors.
Model evaluation	Arm GNU Toolchain	GCC C/C++ Compiler (community supported); not recommended for Cortex-M processors with Helium extension.
Model evaluation	Arm LLVM Embedded Toolchain	C/C++ CLang Compiler for all Cortex-M processors (community supported).
Model evaluation	CMSIS-NN	Software library of neural network kernels optimized for various Arm Cortex-M processors.
Model evaluation	Ethos-U Vela compiler	Compiler for mapping ML models to the Ethos-U processors.
Model evaluation	Arm Virtual Hardware	Simulation model for estimating inference time on different Cortex-M/Ethos-U target systems.
System validation	Arm Virtual Hardware	Simulation model for streaming validation data to different Cortex-M/Ethos-U target systems.
System validation	SDS Framework	For playback of captured data to Arm Virtual Hardware.

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

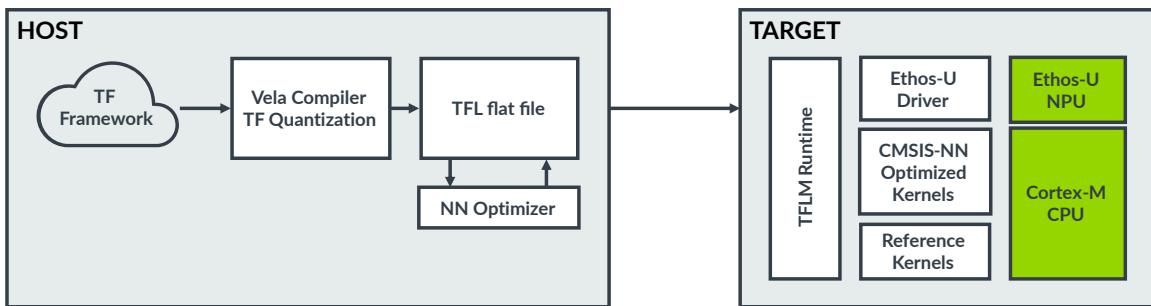
See [MLOps systems](#) for information about how to integrate these tools and software components into an MLOps system.

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 for the target processor.

1.5 Targeting Ethos-U NPUs

The Ethos-U Vela compiler takes an ML model as input and generates an optimized binary for the Ethos-U NPU. The following diagram shows the software development flow for ML models using 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 steps in the process are as follows:

1. Host (Offline) process:
 - a. Start with a trained ML model using the [TensorFlow](#) machine learning framework.
 - b. Use ML model conditioning techniques such as collaborative clustering, pruning, and quantization aware training (QAT) to improve model performance on Ethos-U while preserving accuracy.
 - c. Use TensorFlow Lite post-training quantization to `int8` data types to speed-up the ML model.

- d. The resulting TFL flatbuffer file (*.tflite file) is then transformed for execution on Ethos-U NPU using the Vela Compiler.
 - e. The NN Optimizer identifies graphs to run on Ethos-U and optimizes, schedules, and allocates these graphs.
 - f. The *.tflite file is losslessly compressed to reduce the size.
2. Target / Device process:
- a. Takes the *.tflite file for execution with the TFLU runtime system.
 - b. The Ethos-U driver schedules operators for execution on Ethos-U.
 - c. The CMSIS-NN library executes operators that cannot be mapped to Ethos-U, using a software implementation on Cortex-M.

See [Ethos-U Vela compiler](#) for more information about the Ethos-U Vela compiler.

2. ML software development for Arm Cortex-M processors

This section of the guide introduces the concepts behind ML software development for Cortex-M processors, and provides an example using [TensorFlow Lite for Microcontrollers \(TFLM\)](#).

Arm Cortex-M processors are used in a wide range of 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 automotive subsystems. Because Cortex-M based microcontrollers are widely available, low-cost, and easy-to-use, it is natural for ML application developers to create applications using these devices.

As discussed in [ML compute requirements](#), there are different types of Cortex-M processors, each with different levels of ML processing capabilities. An easy way to define the ML performance of the processor is to measure the number of operations (OPs) per clock cycle. Neural Network (NN) model processing is often based on multiply-accumulate (MAC) operations, with each MAC operation considered as two OPs, a multiply and an 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. The following table provides performance estimates for several Cortex-M processors:

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 a 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 a 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-M52	With Helium technology, these processors can handle 4 MACs/cycle in parallel with data load. As a result, the average OPs/cycle is 8.	0.8 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

Performance can be increased by running the processor at higher clock frequencies. However, there are several 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 must also perform audio data processing tasks. These workloads must be considered when selecting a microcontroller device for an ML application.

A simple real-time KWS application can run on 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. The following third-party articles provide examples of running ML applications on the Cortex-M0 and Cortex-M0+ devices:

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

The following whitepaper provides additional information about running ML applications on Cortex-M microcontrollers:

- [Machine Learning on Arm Cortex-M Microcontrollers](#)

2.1 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, including the following:

- [TensorFlow Lite for Microcontrollers \(TFLM\)](#)
- [MicroTVM](#)
- [PyTorch](#)
- [PaddlePaddle](#)

Factors that might influence your choice of ML software framework could include the following:

- 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 integration of MLOps.

This guide includes an example using TensorFlow Lite for Microcontrollers, often referred to as TensorFlow Micro or TFLM. We chose TFLM for the following reasons:

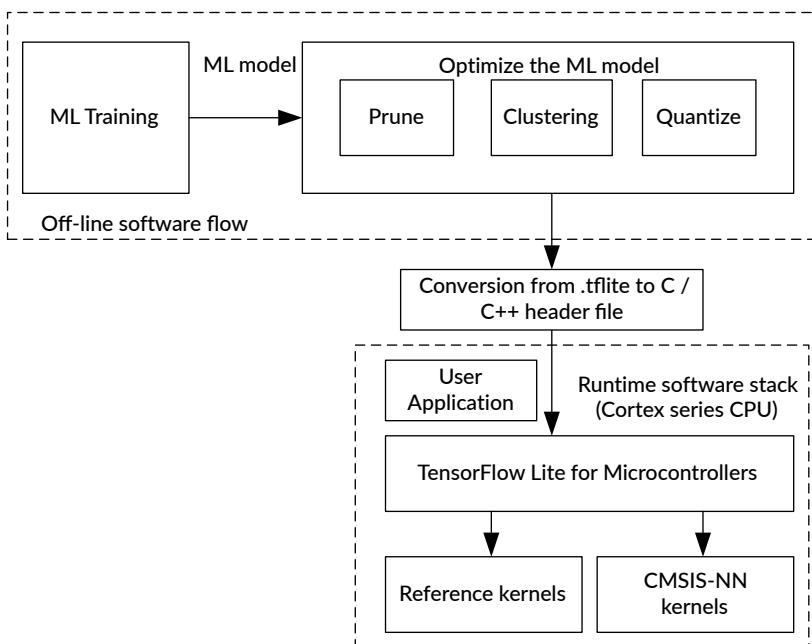
- The NN model weights in TFLM can be quantized to 8-bit integers, which helps reduce the memory footprint. Because memory sizes are often limited in microcontroller devices, this makes TFLM attractive for microcontroller applications.
- Using quantized 8-bit weights allows for efficient NN on embedded processors such as the Cortex-M processor family. Recent Cortex-M processors such as 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 when handling NN models in TFLM.
- Many ML hardware accelerators such as the Ethos-U microNPUs are designed to accelerate quantized ML models and provide software support for TFLM.

- ML models created for TFLM are widely available.

2.2 Example software development flow using TFLM

TensorFlow Lite for Microcontrollers (TFLM) was created by Google. The TFLM runtime library that runs on the microcontroller is an interpreter that reads an ML model and carries out the required operations. The following diagram shows an overview of the software flow:

Figure 2-1: TFLM software flow overview



To create an ML project with TFLM, perform the following steps:

- Create a TensorFlow Lite model file
- Convert the model file to a C/C++ header file
- Identify the inputs and outputs of the NN model
- Integrate the TFLM runtime library
- Integrate the inference functions
- Run the inference and process the results

The following sections describe each of these steps needed to create an ML project with TFLM.

2.2.1 Create a TensorFlow Lite model file

TFLM stores NN models as TensorFlow Lite model files with the file extension `.tflite`, also known as FlatBuffers.

You can use any of the following methods to obtain a TensorFlow Lite model:

- Create a new TensorFlow Lite model using the [TensorFlow Lite Model Maker](#). A data set is required to train the model.
- Use an existing TensorFlow Lite model. [TensorFlow.org](#) provides a range of [examples](#). You can also find other models from various model zoos, such as the [Arm Model Zoo at https://github.com/ARM-software/ML-zoo](#). Note that not all TensorFlow Lite models can be used on Cortex-M based microcontrollers.
- Modify an existing TensorFlow Lite model. For example, you might want to re-train an existing KWS model to allow it to detect different keywords. To do this, you need a new dataset for training. Re-training an ML model in this way is often referred to as “Transfer Learning”. Note that not all ML models can be re-trained using this method.
- Convert another type of model to a TensorFlow Lite model. 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, for example if you exported the model from Matlab, then you can first convert it to a TensorFlow model, then convert the TensorFlow model to a TensorFlow Lite model. For more information about converting ONNX to TensorFlow, see one of the many tutorials available on the Internet. Note that not all ML models can be converted to TensorFlow Lite models.

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.

This example uses the micro-speech example application available in the [TensorFlow Github repository](#), https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech.

2.2.2 Convert the model file to a C/C++ header file

If you want to use a pre-trained ML model from TFLM in your C/C++ project, you must first convert the `.tflite` file to a C/C++ header file.

For the micro-speech example application, the pre-trained model file `micro_speech.tflite` is available from the following location:

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

To convert the `micro_speech.tflite` file to a C++ header file, do the following:

1. Run the Linux `xxd` utility to convert `micro_speech.tflite`:

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

The output file `model.cc` contains the following code, using default variable names generated from the input filename:

```
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;
```

2. Examine the example application code, `main_functions.cc` in the [micro-speech repository](#) to discover the required variable names, as seen in the following code fragment:

```
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);
    ...
}
```

This code shows that the character array name should be `g_micro_speech_model_data`.

3. Edit `model.cc`, change the variable names so that they match the application code, and add the `const` keyword to ensure that the model data is not copied into SRAM when the device starts up:

```
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;
```

You can now use the modified `model.cc` in your C++ programming environment.

2.2.3 Identify the inputs and outputs of the NN model

When using a NN model created by a third party, you must identify information about the inputs and output of the NN model to ensure that when the application is feeding the data to the model, or taking the result from the model, the correct data formats are used. TensorFlow Lite provides a

tool called [Model Analyzer](#) to help with this. You can run Model Analyzer in either Google Colab or Jupyter Notebook.



The `.ipynb` file for is Jupyter Notebook is available on the [Model Analyzer page](#).

There are also a number of other third-party tools that let you analyze and visualize TensorFlow Lite models.

2.2.4 Integrate the TFLM runtime library

To integrate the TFLM runtime library, you can either compile it from the source or use an IDE that supports CMSIS-Pack. Do one of the following:

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

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

To enable CMSIS-NN support, ensure that the command line includes the `OPTIMIZED_KERNEL_DIR=cmsis_nn` option.

- To compile a TFLM runtime library for a device containing a Corstone-300 subsystem, follow the steps in this tutorial:

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 microNPU.

- When using an IDE that supports CMSIS-Pack, you can use the CMSIS-Pack mechanism to integrate the TFLM runtime library. For example, if you are using Keil MDK, select the TFLM software components as follows:

Figure 2-2: TFLM components in CMSIS-Pack installer

Software Component	Sel.	Variant	Version	Description
AWS IoT				
CMSIS	<input checked="" type="checkbox"/>		5.6.0	Cortex Microcontroller Software Interface Components
CORE				CMSIS-CORE for Cortex-M, SC000, SC300, Star-MC1, ARMv8-M, ARMv8.1-M
DSP				CMSIS-DSP Library for Cortex-M and Cortex-A
NN Lib			4.1.0	CMSIS Neural Network(NN) Library
RTOS (API)			1.0.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
RTOS2 (API)			2.1.3	CMSIS-RTOS API for Cortex-M, SC000, and SC300
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
Compiler		ARM Compiler	1.7.2	Compiler Extensions for ARM Compiler 5 and ARM Compiler 6
Data Exchange				Data exchange or data formatter
CBOR				CBOR Software Components
Serialization				Data Serializer Components
flatbuffers				
Data Processing				
Math		tensorflow	1.23.2	Flatbuffers
gemmlowp fixed-point				Software Components for Data Processing
kissfft				Math Components
ruy				
Device		tensorflow	1.23.2	Jsmn
File System		tensorflow	1.23.2	KISS FFT
FreeRTOS		tensorflow	1.23.2	Flatbuffers
Graphics				Startup, System Setup
IoT Utility		MDK-Plus	6.15.3	File Access on various storage devices
ML Eval Kit		MDK-Plus	6.32.3	User Interface on graphical LCD displays
Common				IoT specific software utility
Generic				A collection of end-to-end examples provided by Arm for Arm Cortex-M CPU and Arm Etho
Vibration				
Vision				
Voice				
Machine Learning				Software Components for Machine Learning
NPU Support				
TensorFlow				
Kernel Utils	<input checked="" type="checkbox"/>		1.23.2	TensorFlow Lite Micro Library
Kernel	<input checked="" type="checkbox"/>		1.23.2	TensorFlow Lite Micro Library
Testing	<input type="checkbox"/>		1.23.2	TensorFlow Lite Micro Library

For the **Machine Learning > TensorFlow > Kernel** setting, choose one of the following options:

- CMSIS-NN. With this option, you must also select the **CMSIS > NN Lib** option.
- Ethos-U. With this option, you must also select the **Machine Learning > NPU Support > Ethos-U Driver** option.

The CMSIS-Pack mechanism also lets you import some of the ML application APIs in the ML Embedded Evaluation Kit.

For more examples of using TFLM with CMSIS-Pack, see <https://github.com/ARM-software/ML-examples/tree/main/cmsis-pack-examples>

2.2.5 Integrate the inference functions

There are several important aspects to consider when creating the NN inference application. Examining the [micro-speech example code](#) we can see that this example application integrates the inference functions as follows:

- Create a TensorArena

TFLM requires a memory region in the RAM needs to be assigned. This memory region, called `TensorArena`, stores the inputs, outputs, and intermediate values during inferences. The size of this memory depends on the NN model. For the micro-speech example, the size of this memory is 10KB. The code fragment in `main_functions.cc` that declares 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 following code in `main_functions.cc` allocates the `TensorArena` memory to the TFLM interpreter:

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

- Create a pointer to the input buffer

While `TensorArena` provides the memory needed for the input data, the application also needs a pointer to the input data so that the program code can transfer data to it. In addition, applications might need additional data buffers when pre-processing the input data.

In the micro-speech example, the application code `main_functions.cc` contains the following pointers:

- `model_input_buffer` points to the data inside the `model_input` object.
- `model_input` points to the input data inside the `TensorArena`.

The corresponding code fragments that declare and initialize these pointers are as follows:

```
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, each iteration of the main loop copies data from the `feature_buffer[]` array to `model_input_buffer`. 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 by 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`. The following code fragment defines `kFeatureElementCount`:

```
// 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. For further information about the shape of the input data shape in the micro-speech example, see the `README.md` file in the `train` directory.

2.2.6 Run the inference and process the results

The `Invoke` function in the `interpreter` object starts the inference. In the `micro-speech` example, the following code fragment in `main_functions.cc` executes the `Invoke` function::

```
...
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;
}
...
```

The inference operation stores the result in the output data in the `TensorArena`. The values in the output data array are relative probabilities for each voice-command, for example “Yes”, “No”, “Up”, or “Down”. Before the result can be used to carry out a response, post-processing is needed. Post-processing is performed by the `ProcessLatestResults()` function in the `recognize_commands.cc` file.

After post-processing, the application code calls the `RespondToCommand()` function in the `command_responder.cc` to use the result to trigger a response.

2.3 Re-training an ML model

When creating ML applications with an existing trained ML model, you might want to partially re-train the model to modify the behavior of the NN. For example, you might want to re-train the micro-speech example to recognize different keywords. The micro-speech example provides [information on how to re-train the reference model](#).

The re-training process requires a speech command dataset. A reference dataset is available from Google research at the following location: http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz (version 2)

Information regarding version 1 of this dataset is available in the [Launching the Speech Commands Dataset 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)

2.4 Further information

Other than the micro-speech example, additional examples can be found in the [Tensorflow Lite Micro Github repository](#).

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

The micro-speech example for the Arm development boards uses Mbed tools. Further information about installing these tools is available at <https://os.mbed.com/docs/mbed-os/v6.16/mbed-os-pelion/machine-learning-with-tensorflow-and-mbed-os.html>.

The CMSIS-NN software library is already optimized for Arm Helium technology. Additional information about Helium software optimization is available in the following locations:

- [Getting started with Armv8.1-M based processor: software development hints and tips](#)
- [Helium optimization topics](#)

3. Arm Ethos-U NPU

The majority of Machine Learning (ML) applications perform Neural Network (NN) inference operations. While software running on a processor can execute NN inference operations, using a hardware accelerator can increase performance significantly. Using a hardware accelerator to perform NN inference operations usually improves energy efficiency and allows higher processor bandwidth to handle other tasks.

There are many types of NN inference hardware accelerators. For example, Arm Ethos-U is a family of hardware accelerators designed for microcontrollers and System-on-Chips (SoC) which are known as Neural Processing Units (NPUs).

Ethos-U NPUs are small, power-efficient processors that reduce both the inference time and memory requirements needed to run ML neural networks. The Ethos-U family contains the following designs:

- [Ethos-U55](#)
- [Ethos-U65](#)
- [Ethos-U85](#)

These NPUs are available in commercial products. For example:

- Ethos-U55 is used in the:
 - [Alif Ensemble](#) family from Alif Semiconductor.
 - [PSoC Edge](#) from Infineon.
 - [WiseEye2 AL Processor](#) from Himax Technologies ([Low cost development board](#) from Seeed Studio).
- Ethos-U65 is used in the [i.MX 93](#) family from NXP.

You can also evaluate 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 \(FVP\)](#). For more information about these tools, see [Tool support for the Arm Ethos-U NPU](#).

The Ethos-U85 is the latest member in the Ethos-U product family. It offers up to 2048 MAC units and supports a wide range of NN models including transformer networks. At the same time the energy efficiency can be up to 20% better than previous Ethos-U designs. Ethos-U85 builds on previous generations and offers the same consistent toolchain, so developers can leverage previous Arm ML software investments. More technical details about Ethos-U85 will be released later this year. The remaining contents of this document focus on Ethos-U55 and Ethos-U65.

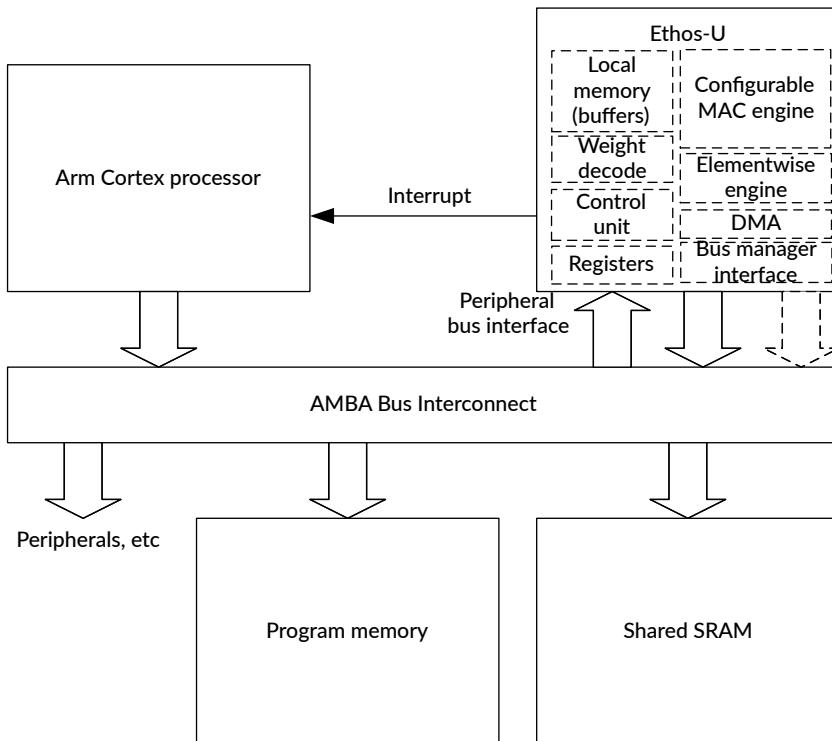
3.1 Ethos-U hardware architecture

An Ethos-U NPU requires a host processor to control its operations. Typically, the host processor is an Arm Cortex-M processor. The Ethos-U NPU provides several hardware interfaces:

- A peripheral bus interface allows the host processor to access the Ethos-U NPU programmable registers.
- Two AMBA bus manager interfaces to access the system memories.
- An interrupt output sends interrupt events to the host processor.

For more information, see [Ethos-U bus manager interfaces](#).

Figure 3-1: Overview of a microcontroller system with an Ethos-U NPU



Most 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. Chip designers can configure the number of MACs that can be carried out per clock cycle, as follows:

- Ethos-U55: 32 to 256
- Ethos-U65: either 256 or 512
- Ethos-U85: 128 to 2048

For more information about the relationship between the number of MACs and performance, see [Ethos-U performance configuration](#). The Ethos-U NPUs also provide an element-wise data processing engine for other computations.

To enhance the efficiency of NN inferences, Ethos-U NPUs contain local memory 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 shared memory to local memory.

Ethos-U NPUs are designed for embedded systems that often have limited memory. To reduce memory usage, Ethos-U NPUs support NN weight data compression. Weight data is decoded on-the-fly during inference operations.

Ethos-U operations are controlled by several registers which are memory mapped on the processor system. When the system needs to perform an NN inference, the operations are broken into several smaller jobs that the Ethos-U NPU runs. Under the control of software libraries, jobs are issued to the Ethos-U NPU using the peripheral bus interface. Each time a job finishes, the Ethos-U NPU issues an interrupt request to the host processor so that the software library can issue the next job.

Ethos-U NPUs also support other interfaces, for example an interface for power management. For more information about these Ethos-U NPU interfaces, see the following:

- [Ethos-U55 Technical Reference Manual](#)
- [Ethos-U65 Technical Reference Manual](#)
- [Ethos-U85 Technical Reference manual](#)

There are many Neural Network models available, and some of those network models contain operators that the Ethos-U hardware does not support. In these cases, software running on the processor handles the unsupported operators. For more information about the ML operators supported by the Ethos-U55 and Ethos-U65 NPUs, see the following:

- [Ethos-U55: Supported data types and operators](#)
- [Ethos-U65: Supported data types and operators](#)
- Ethos-U85 provides support for data types and operators that are available in Ethos-U55 and Ethos-U85, along with support for transformer network and [Tensor Operator Set Architecture \(TOSA\)](#). The specification for TOSA is available from [ML Platform](#).

3.1.1 Ethos-U performance configuration

The number of MACs that can be carried out per clock cycle is configurable for both the Ethos-U55 NPU and Ethos-U65 NPU, addressing a range of performance points.

The following table shows the Ethos-U55 NPU configuration options:

Number of MACs per cycle	Internal memory	Performance@500MHz
256	48KB	256 GOPs
128	24KB	128 GOPs

Number of MACs per cycle	Internal memory	Performance@500MHz
64	16KB	64 GOPs
32	16KB	32 GOPs

The following table shows the Ethos-U65 NPU configuration options:

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

The following table shows the Ethos-U85 NPU configuration options:

Number of MACs per cycle	Internal memory	Performance@1GHz
2048	160KB	4 TOPs
1024	76KB	2 TOPs
512	48KB	1 TOP
256	32KB	512 GOPs
128	16KB	256 GOPs

3.1.2 Ethos-U 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.

To support the high inference performance, Ethos-U85 supports up to 6 AMBA 5 AXI interface:

- AXI interfaces SRAM0, SRAM1, SRAM2 and SRAM3 are for on-chip SRAM.
- AXI interfaces EXT0 and EXT1 are suitable for external memories such as DDR.

3.1.3 Differences between Ethos-U55, Ethos-U65 and Ethos-U85

There are several key differences between the Ethos-U NPUs:

Feature	Ethos-U55	Ethos-U65	Ethos-U85
Number of MACs/cycle	32/64/128/256	256 or 512	128/256/512/1024/2048
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	Up to six 128-bit AXI supporting on-chip SRAM, DRAM and flash
Host CPU support	Cortex-M85, Cortex-M55, Cortex-M7, Cortex-M4, Cortex-M33	Cortex-M85, Cortex-M55, Cortex-M7	Cortex-M85, Cortex-M55, Cortex-M7, Cortex-A520, Cortex-A510, Cortex-A57, Cortex-A55, Cortex-A53, Cortex-A35

Because Ethos-U65 has wider bus interface and additional hardware resources, on average it provides around 50% higher performance than the Ethos-U55. For more information, see the blog post [Arm Ethos-U65: Powering innovation in a new world of AI devices](#).

Ethos-U85 supports even higher number of MACs/cycle (2048 MACs) when compared to Ethos-U65 (512 MACs), and support transformer network and is fully compliant to [TOSA specification](#). Highlights of the Ethos-U85 are covered in the blog post [Arm Ethos-U85: Addressing the High Performance Demands of IoT in the Age of AI](#).

For systems with DRAM/DDR, such as Cortex-A systems running Linux, the Ethos-U65 and Ethos-U85 is more suitable because the bus interface is designed to support memories with longer latency.

3.1.4 Power, security, and performance analysis

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

- Power management interface: Ethos-U55, Ethos-U65 and Ethos-U85 provide Q-channels for the management of clock and power gating. This interface connects to system level power management hardware, for example power control infrastructure built with the [Arm CoreLink PCK-600 Power Control Kit](#). For more information about Q-channels, refer to the [AMBA 4 Low Power Interface Specification](#).
- Security management: If an Ethos-U NPU is used in a TrustZone enabled system, software running in the Secure state can restrict access permissions 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 NN workloads and identify potential performance issues.

3.2 Ethos-U system integration

There are two common system arrangements when using an Ethos-U NPU:

- An Ethos-U NPU controlled by a Cortex-M processor. In this scenario, the Cortex-M processor runs the application code, dispatching NN inference workloads to the Ethos-U NPU. Because the application code runs directly on the Cortex-M processor, you can use the Cortex-M processor for other tasks in addition to 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 runs the application code and dispatches NN inference workloads to the Cortex-M processor in the ML subsystem. The Cortex-M processor in turn manages the low-level control of the Ethos-U NPU.

In both cases, the system designer must ensure that the Ethos-U NPU is able to do the following:

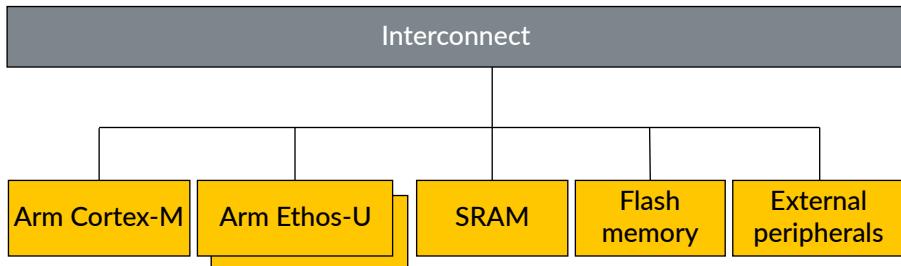
- Access the memory system using the bus manager interface.
- Accept bus transactions from the host processor using the peripheral bus interface. The Ethos-U NPU has an AMBA 4 APB interface that provides access to its registers. The base address of the registers block is system-specific, and is normally located in a peripheral address range.
- Generate interrupt requests for the host processor using the interrupt output. The interrupt output is connected to the interrupt controller for the host processor, for example the NVIC interrupt 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 NPU is integrated into a system with TrustZone security extension, the hardware can be configured to either restrict the Ethos-U NPU to be used only by the secure firmware, or to be available to the applications running in the Non-secure world.

3.2.1 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 3-2: Ethos-U system



The system contains the following components:

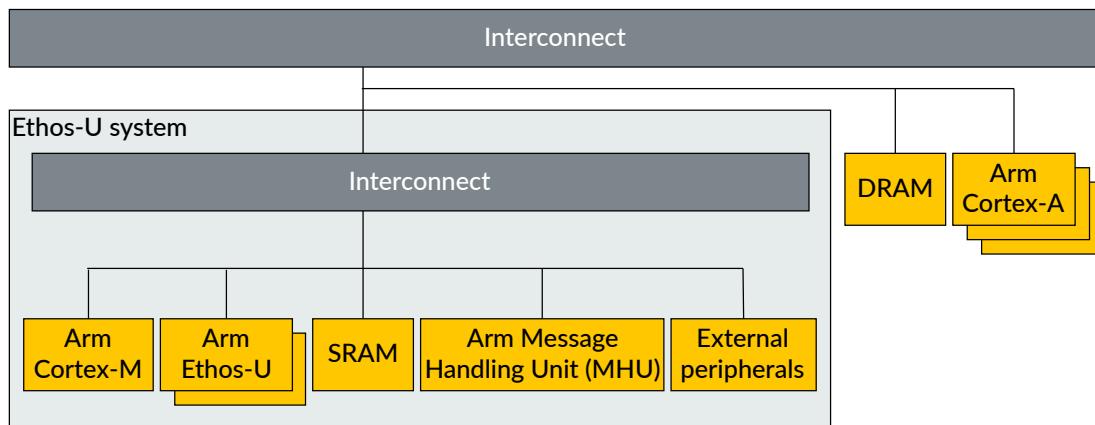
- Cortex-M
 - The Cortex-M series CPU is the application processor that controls one or more Ethos-U NPUs. You can specify your preferred Cortex-M series CPU, but Arm recommends the following CPUs:
 - Cortex-M4
 - Cortex-M7
 - Cortex-M33
 - Cortex-M52
 - Cortex-M55
 - Cortex-M85
- Ethos-N NPU
 - Either an Ethos-U55 NPU, an Ethos-U65 NPU, or an Ethos-U85 can be paired with the Cortex-M CPU, but the Ethos-U65 and Ethos-U85 NPUs have been designed to optimize data transfer between the slower memory (for example DDR) and the fast memory cache.
- SRAM
 - The input feature map (IFM) data and the output feature map (OFM) data are stored in SRAM. You can specify your preferred amount of SRAM, but optimal performance is obtained when the network is fully placed 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.

3.2.2 Ethos-U integration with an 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 3-3: A typical Ethos-U subsystem



The system contains the following components:

- Ethos-U NPU
 - Either an Ethos-U55 NPU, an Ethos-U65 NPU, or an Ethos-U85 NPU can be paired with the Cortex-M CPU. The Ethos-U65 and Ethos-U85 NPUs have been designed to optimize data transfer between the slower memory (for example DDR) 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.



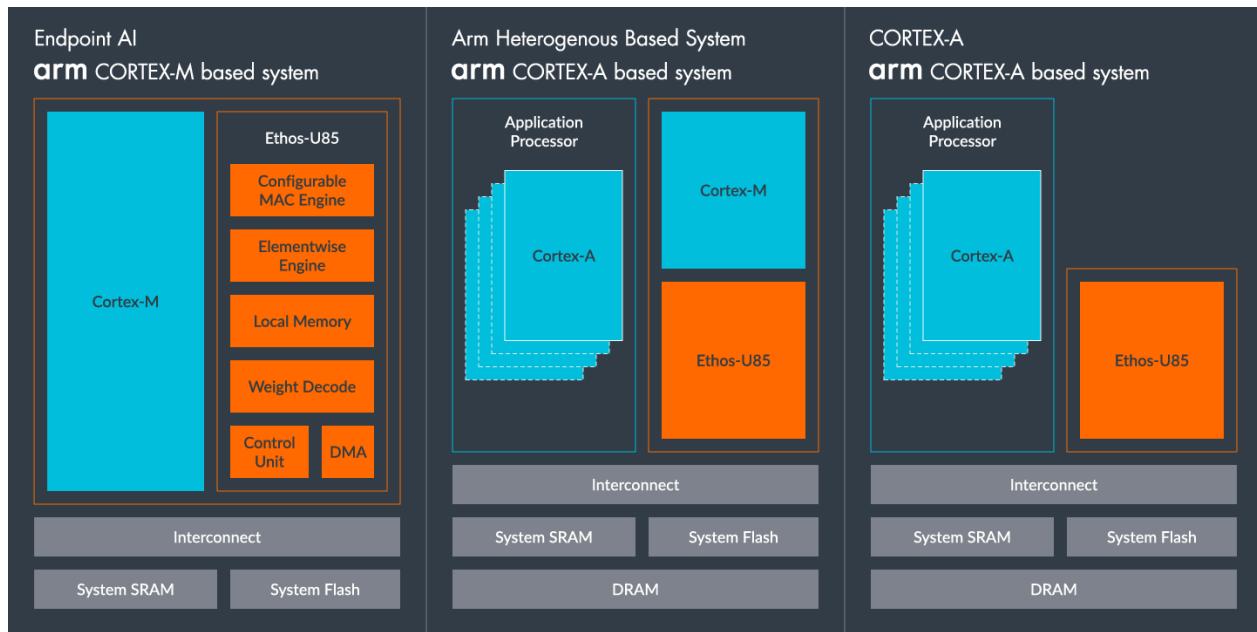
For an example of using an MHU, see the [Arm CoreLink SSE-200 Subsystem for Embedded Technical Reference Manual](#).

-
- DRAM
 - Weights, biases, the input feature map (IFM), and the output feature map (OFM) data are stored in slower, high latency memory such as 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.

3.2.3 Ethos-U85 system integration

Ethos-U85 supports the two aforementioned system integration methods, plus the possibility to be managed by Cortex-A system directly.

Figure 3-4: Ethos-U85 system integration arrangements



3.3 Corstone reference designs

Instead of building new systems from scratch, Arm provides reference system designs in Arm Corstone subsystem products to help system designers create Cortex-M based systems. The Arm Corstone-3xx series 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 with a choice of either the Ethos-U55 or Ethos-U65 NPU.
- [Corstone-310](#) provides a reference system design for the [Cortex-M85](#) processor with an Ethos-U65 NPU.
- [Corstone-315](#) provides a reference system design for the [Cortex-M85](#) processor with an Ethos-U65 NPU.
- [Corstone-320](#) provides a reference system design for the [Cortex-M85](#) processor with an Ethos-U85 NPU.

To help software developers to test their software, simulation models of the Corstone-300 and Corstone-310 reference designs are available as Fixed Virtual Platforms (FVPs). These models are available on the [Arm Developer website](#).

The FVPs include simulation models of the corresponding Cortex-M processor and Ethos-U NPU. These models allow software developers to test their ML software easily. Note that the simulation models are not cycle accurate, especially for the processors in these models. However, they can provide an indication of expected processing time on the Ethos-U NPUs. For example, the Corstone-300 FVP is roughly 90% accurate for Ethos-U55 cycle timing. Note that the timing accuracy of the Ethos-U65 in the FVP model is lower than the Ethos-U55.

When simulating an NN inference workload with an FVP, if the workload includes 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 accurate estimate of processing time. If cycle accurate timing is required, you can choose other evaluation methods. For example:

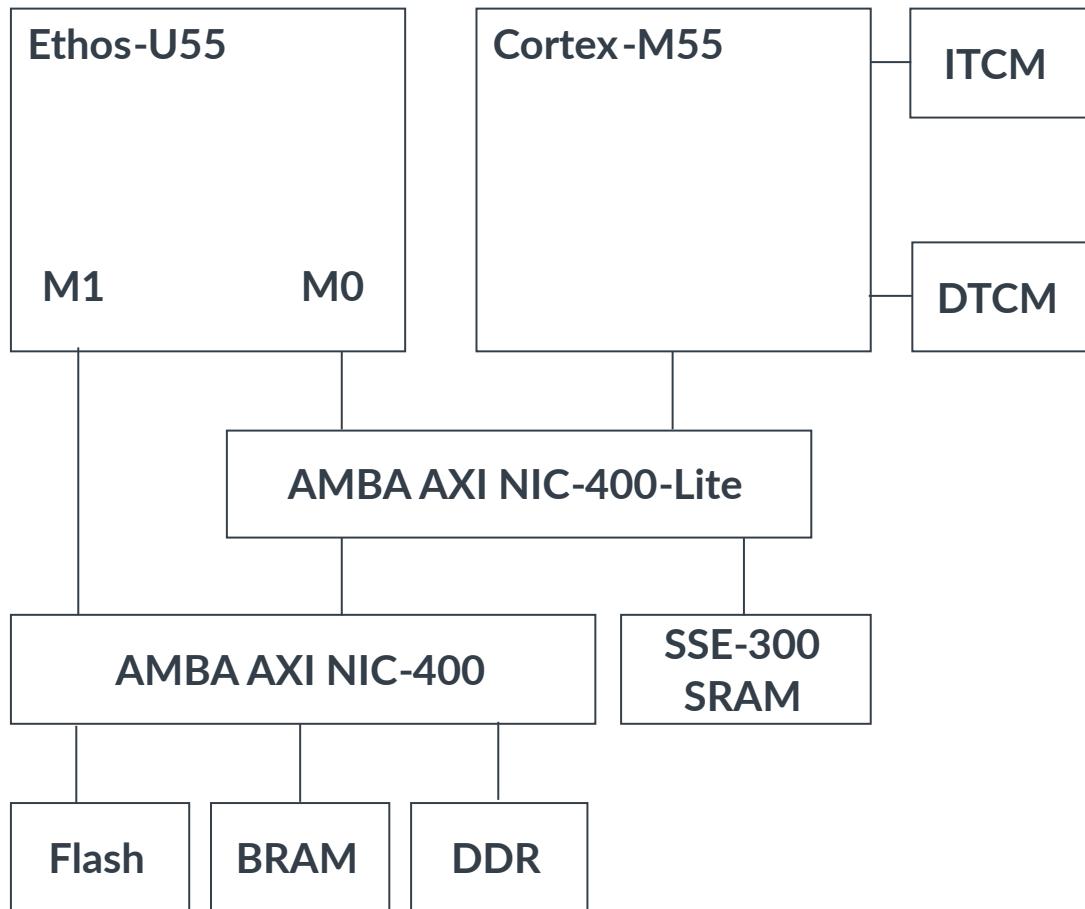
- An FPGA platform, for example the [Arm MPS3 FPGA board](#)
- [Arm IP explorer](#)
- Another hardware device using the same combination of processor and NPU as the device that you are developing.

Arm provides the following [FPGA images](#) for the Arm MPS3 FPGA board:

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

The following diagram shows a simplified overview of the AN552 system architecture:

Figure 3-5: Simplified overview of the AN552 system architecture



This diagram provides an overview of the Corstone-300 memory system. The Ethos-U NPU can access SRAM, Flash, FPGA block RAM (BRAM), and DDR memory. In this design, you cannot store the tensor arena in D-TCM or I-TCM because the Ethos-U NPU cannot access this memory.

The advantage of running applications on real hardware such as 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 into a SoC. Arm recommends that you read the technical reference manual for these reference designs before starting your own design.

3.4 ML software support for Ethos-U

An ML software project is usually based on a specific ML software framework. The Ethos-U NPUs support 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 significantly 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, for example Ethos-U, as well as a range of modern Arm Cortex processors.

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

1. Off-line development flow. This consists of:
 - Preparation of the ML model
 - Quantizing the ML model to use 8-bit weight data (TF Quantization tooling - TOCO)
 - Optimizing the ML model using the [Vela compiler](#). 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](#) is used instead. The Vela compiler also handles memory layout optimizations. See [Ethos-U Vela compiler](#) for more information. The Vela compiler runs on a desktop PC or similar device.
2. The NPU runtime software stack. This consists of a range of software components running on the target hardware that interact with each other in specific ways. These include the following:
 - User application

The user application runs the required functions and makes calls to the TFLM library when it performs an inference of the model.

- TFLM

The TFLM framework is compiled into a C++ library that contains a copy of the optimized model from Vela along with the 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 a TensorFlow Lite custom operator is encountered. 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 TFLM framework and the Ethos-U NPU to process custom operators. When the Ethos-U NPU completes its processing, it signals back to the driver, which in turn informs the TFLM library.

- The [CMSIS-NN](#) library

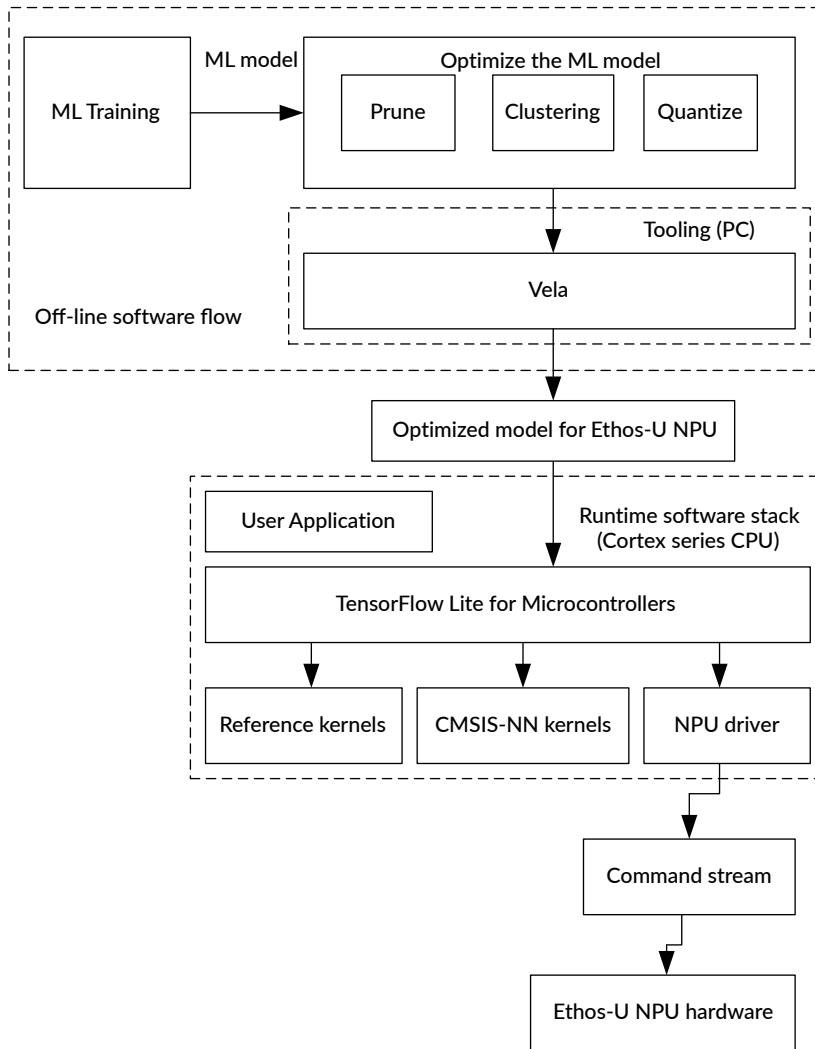
This contains highly optimized and performant kernels that accelerate a subset of operators in the TFLM framework. It is needed to handle the ML operators that are not supported by Ethos-U.

- Reference kernels

This contains a set of kernels for all the operators in the TFLM framework. They are used when the TFLM framework encounters ML operators that are not supported by the Ethos-U or the [CMSIS-NN](#) library.

The following diagram shows an overview of the software development flow:

Figure 3-6: Software development overview



3.4.1 Ethos-U custom operators

TensorFlow Lite for Microcontrollers (TFLM) is an interpreter. It reads an NN model, in the form of `.tflite` data in memory, and carries out the functions of the NN operators. In order to allow NN

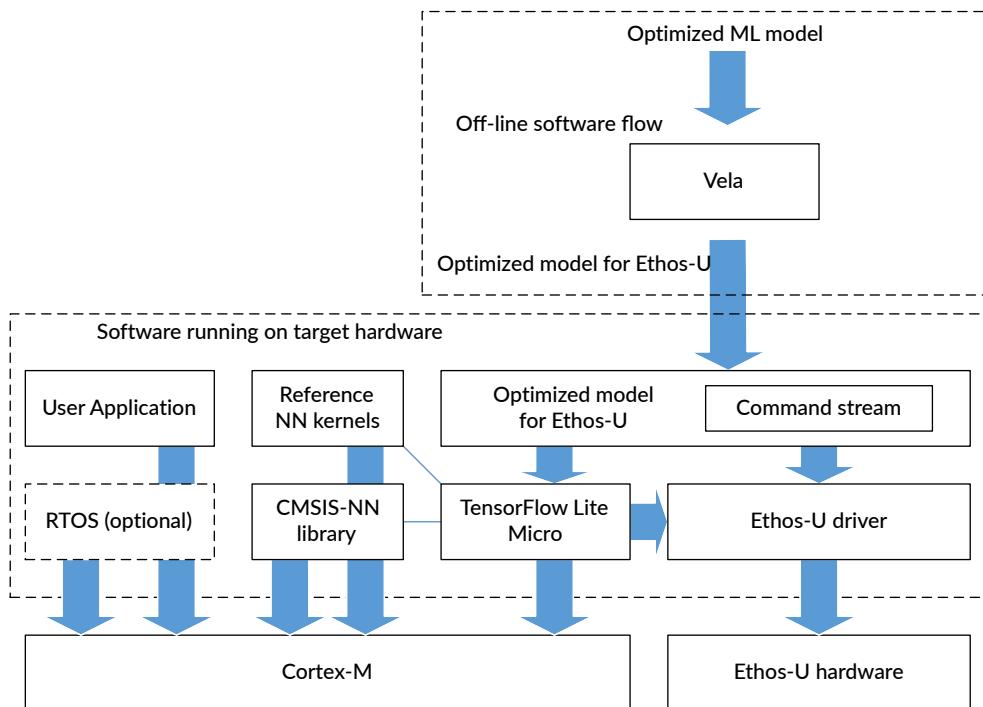
model processing to be accelerated by the Ethos-U NPU, TFLM supports a feature called custom operators.

During the compilation of a NN model, the Vela compiler groups the sequence of operators that can be accelerated by the NPU 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 NPU. You can read more about the Ethos-U custom operator [here](#).

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

In a Cortex-M based microcontroller with Ethos-U NPU, the runtime software stack provides the software required to support the Ethos-U NPU. This includes the user application, which uses the TFLM library to execute parts of the optimized model, or command stream, on the Ethos-U NPU. Based on the application requirements, additional software components such as an RTOS might be needed.

Figure 3-7: Microcontroller software overview

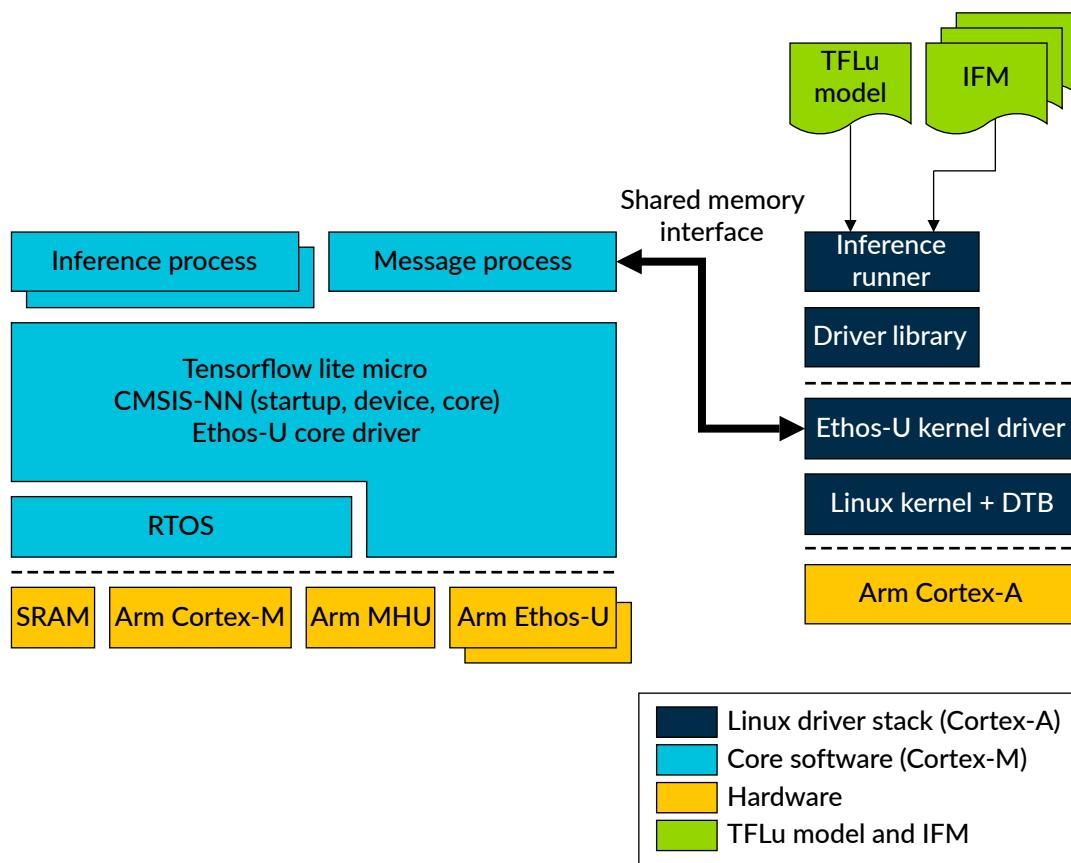


3.4.3 ML software for ML subsystems in a larger SoC

In the scenario where the Ethos-U is integrated into an ML subsystem within a SoC with Cortex-A processors, the applications running in the Linux environment use the Ethos-U kernel driver to communicate with the ML subsystem. This driver then communicates with the Cortex-M processor using 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 3-8: ML subsystem software stack overview



The software components include the following:

- Inference runner

The inference runner is a test application that runs inferences on the Ethos-U driver stack. The inference runner takes as input a TFLM model that has been optimized by the Vela compiler, 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 Software architecture scenarios and use cases

The software architecture for an Ethos-U subsystem can be categorized into two 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 situation 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 and 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 [Machine Learning Inference Advisor](#).

For ML developers using PyTorch, there are third-party tools available that can convert PyTorch models to TensorFlow Lite. For example, [TinyNeuralNetwork](#) from Alibaba. Additional information about converting PyTorch to TensorFlow Lite using [ONNX \(Open Neural Network Exchange\)](#) is available in the following resources:

- [PyTorch to TensorFlow Lite for deploying on Arm Ethos-U55 and U65](#)
- [PyTorch to TFLite](#)

3.7 Porting Ethos-U software to a new hardware platform

To use the Ethos-U hardware, software needs the following information:

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

In addition, systems with the TrustZone security extension require the following:

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

Software must also implement the following:

- Invoke the initialization function `ethos_init()` before using the Ethos-U NPU. The `ethos_init()` function is part of the Ethos-U driver.
- Provide an interrupt handler to call the Ethos-U interrupt handler function in the Ethos-U driver.

As with any other embedded software project, the software developer must also prepare a suitable linker script or scatter file to allow the toolchain to generate a program image that is compatible with the system memory map of the hardware platform.

The following sections provide more information about each of these areas.

3.7.1 Security configuration for Ethos-U in a TrustZone system

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

- 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 NPU is accessible to the NPU.

- Configure the PROT register in the Ethos-U NPU 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 is configured using the Interrupt Target Non-secure (ITNS) register.

It is possible to change the security state of the Ethos-U NPU when the system is on. This process requires a soft reset. For more information, see the following sections in the Technical Reference Manual:

- [Ethos-U55 Boot flow information](#)
- [Ethos-U65 Boot flow information](#)

3.7.2 An example of Ethos-U initialization

For an example of Ethos-U NPU initialization, see the [ML Embedded Evaluation Kit](#).

In this example, the initialization function is in `ethosu_npu_init.c`

Inside this file, the following `arm_ethosu_npu_init()` code calls the Ethos-U initialization function:

```
int arm_ethosu_npu_init(void)
{
    int err = 0;

    /* Initialize the IRQ */
    arm_ethosu_npu_irq_init();

    /* Initialize 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 this example, `ETHOS_U_BASE_ADDR` specifies the base address of the register block. 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_ethosu_npu_irq_init()` defined as follows:

```
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 this code, `ETHOS_U_IRQN` defines the IRQ number of the Ethos-U. 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 configures the exception vector as part of the initialization. This is not required if the exception vector is already defined and stored in non-volatile memory.

Optionally, the software developer can configure 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. For a list of NVIC management functions in the CMSIS-Core, see [Interrupts and Exceptions \(NVIC\) in the CMSIS-Core documentation](#).

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.

3.7.3 Software integration for the Ethos-U micro NPU in custom designs

The [ethos-u-core-platform](#) project is one of the recommendation starting points for software developers who want to create software packages for custom hardware targets with Ethos-U NPUs. This project provides a mechanism for producing firmware binaries for different defined targets. The `targets` directory contains examples that demonstrate how support for Corstone-300 and Corstone-310 reference design targets was added.

To define your custom target, do the following:

1. In `targets/demo`, edit the `CMakeLists.txt` and `target.cpp` files for your design. Search for `TODO` to find the code you need to edit.

In `target.cpp`, specify the base address for the Ethos-U and the interrupt number relative to the Ethos-U interrupt in the Cortex-M Vector Interrupt Table.

2. Create one of the following:
 - A linker script, if compiling with GCC.
 - A scatter file, if compiling with Arm Compiler.

The linker script or scatter file must match the memory map of your target. FPGA vendors usually provide automated tools for generating linker scripts for a given target.

See this [example from Xilinx](#). The `targets/corstone-300/platform.ld` linker script targets 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 the I-TCM and 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 scatter loads data listed in the copy table, in this example data from DDR to BRAM. The `_cmsis_start` function in `ethos-u/core_software/cmsis/CMSIS/Core/Include/cmsis_gcc.h` performs this copying of data. Then, constructors and `main()` functions are called.

You need to create a linker script or scatter file specific to your custom target.

3.7.4 Linker script design

Using the baremetal examples in the core-platform project, let us examine how to 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 either SRAM or DRAM depending on whether we compile the application for Ethos-U55 or Ethos-U65. The following code shows a snippet from the scatter file for the Corstone-300 that places different symbols in SRAM.

Figure 3-9: 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)
    }
}
```

In the above snippet, if the `ETHOSU_ARENA` cmake parameter is equal to 0, the `.bss.tensor_arena` is placed in SRAM. `ETHOSU_ARENA` is set to 0 when building the application for Ethos-U55 with the `Shared_Sram` memory mode. In this situation, the tensor arena should be placed in 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, the `ethos-u-core-platform/applications/baremetal/models/ethos-u55-128/keyword_spotting_cnn_small_int8/model.h` file contains 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. 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 by the linker script or scatter file.

3.8 Customizing the Ethos-U driver and RTOS integration

There are several scenarios in which you might want 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 workloads. As a result, you might want to add semaphore operations to the Ethos-U driver to ensure that only one application thread can access 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. The RTOS could then resume the application thread when an interrupt is received from the Ethos-U NPU.

In order to allow these customizations, some of the functions in the Ethos-U driver are defined with a `weak` attribute. This means that the implementations of those functions can be overridden without needing to modify the source code of the Ethos-U driver. These `weak` functions include the following:

- The inference begin and end functions, `ethosu_inference_begin` and `ethosu_inference_end`.
- D-cache maintenance functions, for example D-cache flush (clean) and D-cache invalidate.
- RTOS functions including `mutex`, and `semaphore`.
- The 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 from the following location: <https://gitlab.arm.com/artificial-intelligence/ethos-u/ethos-u-core-driver>

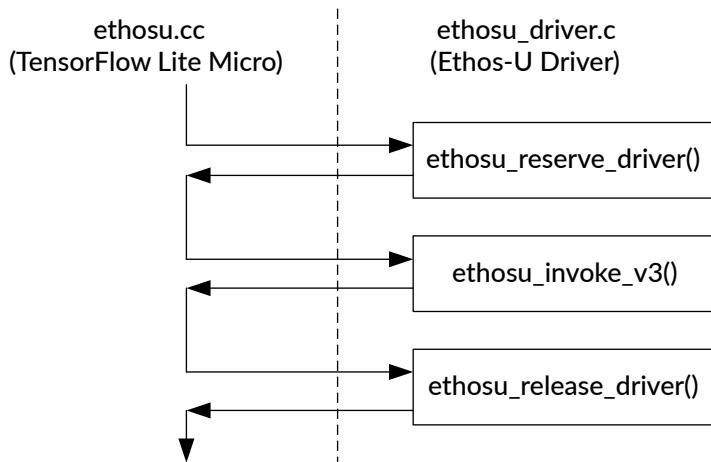
The Ethos-U driver source code is in the file `ethosu_driver.c`.

During an inference operation, code behaves as follows:

1. The TFLM kernel encounters a custom operator.
2. `ethosu.cc` in the TFLM kernel calls the driver code, including the `ethosu_invoke_v3()` function in `ethosu_driver.c`.
3. The `ethosu.cc` function returns when the inference is completed.

The following diagram shows the code sequence:

Figure 3-10: Interface between TFLM and Ethos-U driver

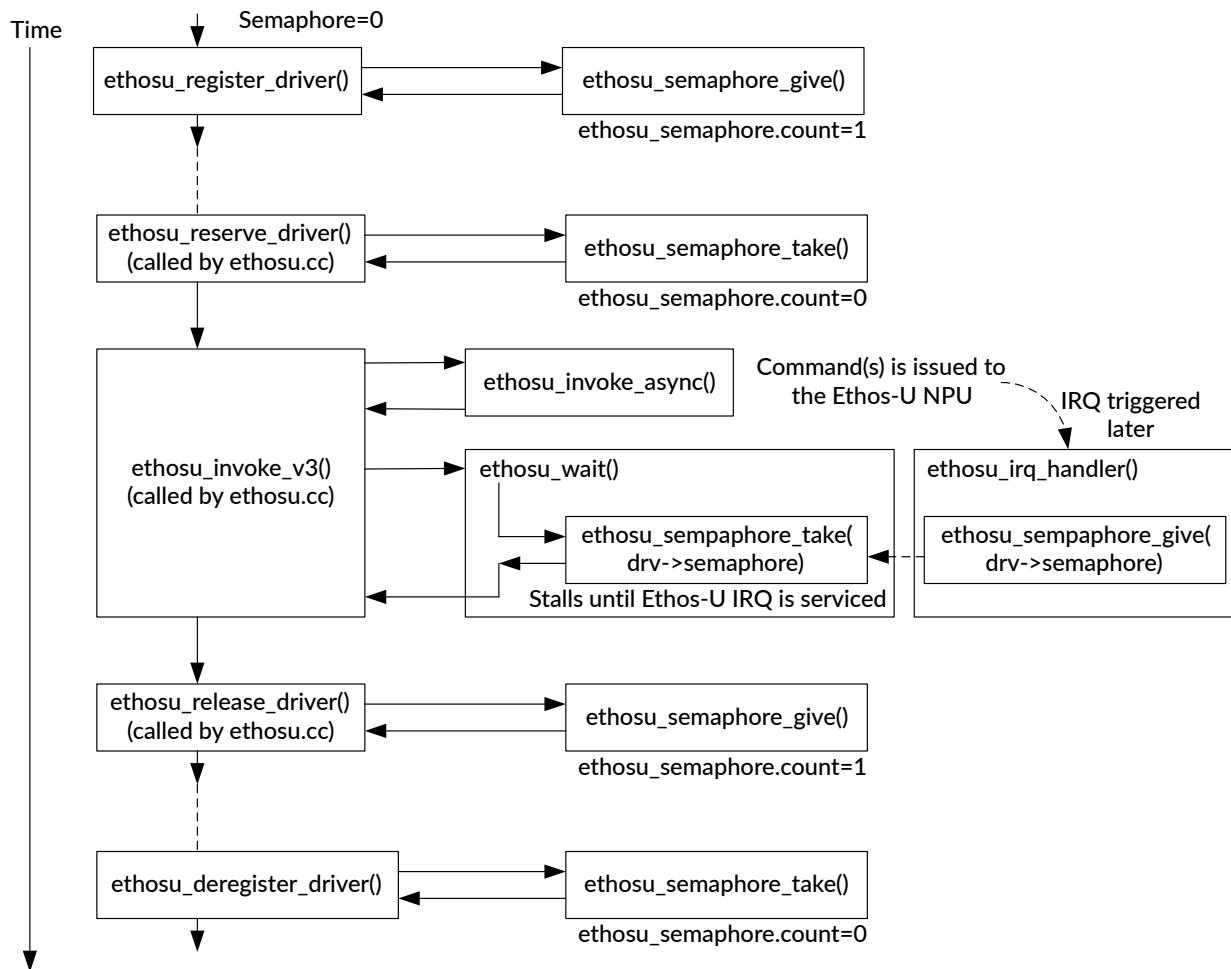


The Ethos-U driver code includes a bare metal semaphore implementation as default. Two semaphores are used:

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

The following diagram illustrates the semaphore operations during a NN inference:

Figure 3-11: Ethos-U driver semaphore sequence during an inference



The semaphore sequence behaves as follows:

- The semaphore is setup when the `ethosu_register_driver` function is called.

This happens during the execution of `ethosu_init()`, the Ethos-U initialization function.

For each Ethos-U NPU in the SoC, the application calls the `ethosu_init()` function to initialize the NPU and configure the associated security and privilege level. If there are multiple Ethos-U NPU in the system, the `ethosu_init()` function is called multiple times, each time with a different driver handle (`struct ethosu_driver *drv`). There is a different Ethos-U driver handle for each Ethos-U NPU.

- The `ethosu_register_driver()` function registers the driver and creates the `ethosu_semaphore` semaphore. This driver now has an associated Ethos-U NPU.
- In the TFLM code (`ethosu.cc`), the `ethosu_reserve_driver()` function is called.

This function executes the `ethosu_semaphore_take()` function.

In applications that have only one ML application thread, the `ethosu_reserve_driver()` function takes the `ethosu_semaphore` 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 TFLM code `ethosu.cc` then executes `ethosu_invoke_v3()`, which in turns call the `ethosu_invoke_async()` function.

The `ethosu_invoke_async()` function starts running the command stream on the Ethos-U NPU hardware and returns.



The completion of the function is asynchronous to the NPU's operations.

Note

- Inside `ethosu_invoke_v3()`, the TFLM code `ethosu.cc` then executes the `ethosu_wait()` function. 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 switches to ETHOSU_JOB_RUNNING state. In block mode, nothing happen in this state. Because the `break` statement is not executed, the execution flow falls into the next state, ETHOSU_JOB_DONE.
 - In ETHOSU_JOB_DONE state, the code executes the `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 puts the processor to sleep because the loop that polls 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 completes the inference operation, the Ethos-U interrupt handler is executed. The handler calls the `ethosu_semaphore_give()` function and updates the `drv->semaphore`.



The `ethosu_semaphore_give()` function contains a `sev` instruction. For single-core Cortex-M systems, the `sev` instruction is not 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 wake up the other processor.

- Now the `ethosu_semaphore_take()` function in `ethosu_wait()` can take the `drv->semaphore` semaphore and complete the rest of the operations.
- On success, when the status register of the NPU is true, the `ethosu_wait()` function returns 0 and the `ethosu_invoke_v3()` function completes.

- The TFLM code `ethosu.cc` then executes `ethosu_release_driver()`. This releases the `ethosu_semaphore` semaphore. If there is another application thread that is waiting to use the Ethos-U, it can take the semaphore and resume.
 - At a later point in time, 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 when those operations are atomic.

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

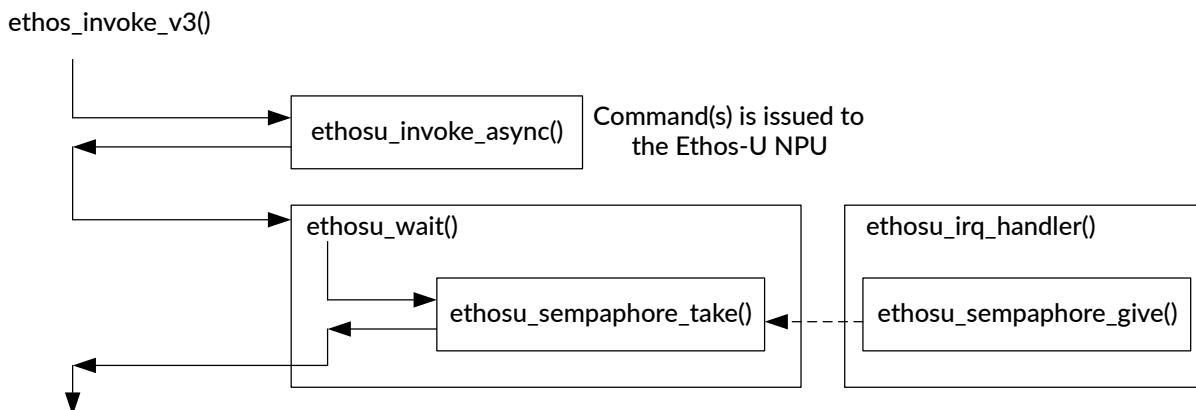
Note the following:

- The `ethosu_semaphore_give()` API is executed during `ethosu_irq_handler()`. Some RTOS might have different semaphore APIs for threads and interrupt handlers. Take care to use the correct API when porting the code to an RTOS.
 - The Ethos-U driver can support multiple Ethos-U NPUs by using different driver handles for each NPU instance.

3.8.1 Putting the processor to sleep while the Ethos-U NPU is running

By default, after an inference operation starts, the Ethos-U driver puts the processor to sleep using a `wfe` (Wait for Event) instruction. This happens in the `ethosu_semaphore_take()` function. The following diagram shows the sequence of events:

Figure 3-12: Ethos-U invoke function



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

power capabilities of the microcontroller or SoC. The following code shows where this additional power control code could be added:

```
// Bare metal simulation of waiting/sleeping 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 here
                  // for device-specific power saving features.
    }
    s->count--;
    return 0;
}
```

3.8.2 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 this 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 replace the semaphore functions `ethosu_semaphore_take()` and `ethosu_semaphore_give()` in the Ethos-U driver with RTOS-specific semaphore functions.

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

- When waiting for the NPU resource, in `ethosu_reserve_driver()`
- After the Ethos-U starts running, in `waiting` in `ethosu_wait()`

With semaphore support in the RTOS, the processor can context switch into other threads if there is another active thread waiting to be executed. If there is no other active thread waiting, the RTOS executes its own idle thread, which should put the processor into sleep mode 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()` ensure that, if there are multiple ML application threads sharing the same Ethos-U NPU, only one of the threads has access to the Ethos-U NPU at a time.

Example code for FreeRTOS is available here:

https://gitlab.arm.com/artificial-intelligence/ethos-u/ethos-u-core-platform/-/blob/25.02/applications/freertos/main.cpp?ref_type=tags

This repository contains FreeRTOS-specific implementation of the `weak` functions in the Ethos-U driver.

3.8.3 Ethos-U driver configuration

Several registers in the Ethos-U NPU must be configured by the Ethos-U driver to achieve optimal performance of the Ethos-U. The registers that need to be configured are defined in the following header files:

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

The hardware has four 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:

- Ethos-U Vela

The Ethos-U Vela tool compiles a TensorFlow Lite flatbuffer file into an optimized 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 quantized to either 8-bit unsigned, 8-bit signed, or 16-bit signed.

The optimized model contains 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 remain unchanged and instead run on the Cortex-M series CPU using an appropriate kernel, such as the Arm-optimized CMSIS-NN kernels.

For more information, see [Ethos-U Vela compiler](#).

- Machine Learning Inference Advisor (MLIA)

The Machine Learning Inference Advisor (MLIA) helps developers design and optimize neural network models for efficient inference on Arm targets by enabling performance analysis and providing actionable advice early in the model development cycle. This advice includes information about supported operators, performance analysis, and suggestions for model optimizations such as pruning, clustering, and so on.

For more information, see [Machine Learning Inference Advisor](#).

- Arm Virtual Hardware, VSI interfaces for sensors, audio, and video

Arm Virtual Hardware (AVH) provides simulation models, software tools, 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 simulates data streaming interfaces such as audio, video, and sensors, which are commonly used in IoT and ML applications. The system provides eight independent VSI instances that can function in parallel, allowing multi-channel input/output interfaces.

For more information, see [Arm Virtual Hardware](#).

- Synchronous Data Stream (SDS) Framework

The Synchronous Data Stream (SDS) Framework implements a flexible data stream management for sensor and audio data interfaces. It provides methods and tools for developing and optimizing embedded applications that integrate DSP and ML algorithms.

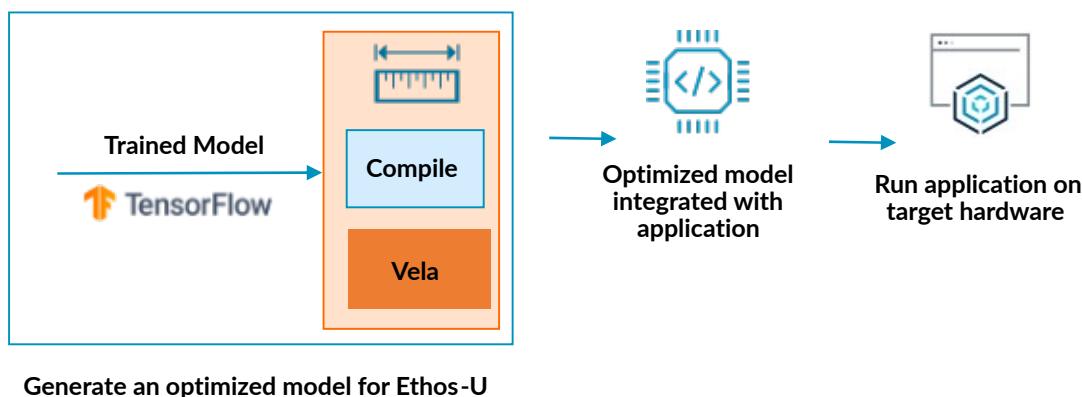
For more information, see [SDS Framework](#).

4.1 Ethos-U Vela compiler

Ethos-U Vela is a software tool developed by Arm that compiles a TFLM model into an optimized version that runs on an Ethos-U NPU. Vela takes TensorFlow Lite models as input, applies optimizations including memory optimization and layer fusion techniques, and generates a compiled binary that is specifically optimized for the Ethos-U architecture. This optimized binary maximizes use of Ethos-U NPU hardware features for efficient execution of machine learning workloads.

The following diagram shows an overview of the Ethos-U Vela development flow:

Figure 4-1: Ethos-U Vela development flow



The optimized model contains 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 instead run on the host processor (e.g. Cortex-M series CPU) using an appropriate kernel.

Ethos-U Vela attempts several different compilation strategies and applies a cost function to each one. Vela then chooses the optimal execution schedule for each supported operator or group of operators. Vela is bit-accurate with TF Lite reference kernels, and therefore there is no loss of accuracy when you optimize an NN with the Vela compiler.

The Vela compiler can report estimated performance. However, the results from Vela are only an approximate estimation. Software developers should consider using the [Machine Learning Inference Advisor](#) together with a performance model to obtain more accurate performance data.

The Vela compiler performs various memory optimizations to reduce both permanent, (for example flash) and runtime (for example SRAM) memory requirements. These optimizations include the following:

- Compressing all the weights in the model reduces permanent storage memory usage.

- Cascading reduces runtime memory usage 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 is executed by the Ethos-U NPU.

Finally, Vela outputs the optimized model as a TFLM model, and generates a performance estimation report. The report provides statistics, such as memory usage and inference time.

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

4.1.1 Requirements

The following should be installed before installing Vela:

- Windows 10 or Linux (amd64)
- Python 3.10 or higher:
 - Development version containing the Python/C API header files, for example `apt install python3.10-dev` OR `yum install python310-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> for more information.

4.1.2 Installation

To install Ethos-U Vela, run the following command:

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

There is a known issue when using Ethos-U Vela with older versions of NumPy that uses different C APIs. To work around this issue, you must install the desired NumPy version first, and then build Ethos-U Vela with that specific NumPy version, as follows:



Note

```
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
```

For more information, see [Ethos-u Vela, Known Issues, NumPy C API version change](#).

4.1.3 Usage

Ethos-U Vela takes an input `.tflite` file passed on the command line. This file contains the neural network to be compiled.

Vela outputs an optimized `.tflite` file with a `_vela` suffix in the file name, along with performance estimate CSV files, all in the output directory. Vela also prints a performance estimation summary on the console. For more information, see [Vela Performance Estimation Summary](#).

The command-line syntax for Ethos-U Vela is as follows:

```
vela [ <options> ] <network>
```

Where `<network>` is the filename of the input TFLM network, and `<options>` are any available configuration options, as follows:

-h, --help

Show the help message and exit.

--version

Show the 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 a configuration file, use the `--config` argument with one of the listed configuration files, for example `--config Arm/vela.ini`.

--output-dir OUTPUT_DIR

Specifies the output directory to write files to. The default is `output`.

--enable-debug-db

Enables the calculation and writing of a network debug database to the output directory.

--config CONFIG

Vela configuration files 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,ethos-u85-128,ethos-u85-256,ethos-u85-512,ethos-u85-1024,ethos-u85-2048}

Accelerator configuration to use. The default is ethos-u55-256.

--system-config SYSTEM_CONFIG

Specifies the system configuration to select from the Vela configuration file. The default is `internal-default`.

--memory-mode MEMORY_MODE

Memory mode to select from the Vela configuration file. The default is `internal-default`.

--tensor-allocator {LinearAlloc,Greedy,HillClimb}

Tensor Allocator algorithm. The default is `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. The default is 3.

--optimise {Size,Performance}

Set the optimization strategy. The `size` strategy results in minimal SRAM usage, ignoring `arena-cache-size` if specified. The `Performance` strategy results in maximal performance, using `arena-cache-size` if specified. The default is `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. The default is 16.

--recursion-limit RECURSION_LIMIT

Set the recursion depth limit. Setting this option too low may result in `RecursionError`. The default is 1000.

--hillclimb-max-iterations HILLCLIMB_MAX_ITERATIONS

Set the maximum number of iterations the Hill Climb tensor allocator will run. The default is 99999.

For a detailed explanation of all the available options, see [Vela Options: Command Line Interface](#).

4.1.4 Command examples

The following example shows a typical command-line usage for Vela, using a configuration file `vela.ini` to describe various properties of the Ethos-U embedded system:

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

See [Configuration File Reference](#) for a detailed explanation of all configuration file options.

The following is an example configuration file:

```
; 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
```

4.1.5 Optimization considerations for the Vela compiler

The [Ethos-U Vela compiler documentation](#) provides complete information about how to run the tool. This section provides additional information related to optimization choices.

4.1.5.1 Vela schedulers and implications on memory footprint

The Vela compiler supports two optimization strategies, each using a different scheduling algorithm:

- `--optimise Performance` optimizes a neural network for maximum performance, measured by the number of inferences per second. This is the default scheduler when using the Vela compiler.
- `--optimise size` optimizes a neural network for minimum peak SRAM usage during an inference. This option lets you benefit from hardware acceleration even with a low SRAM budget. Vela reduces the peak SRAM usage by reusing tensors with cascading. This results in slightly lower performance and requires some weights to be re-read from the memory.

On the Ethos-U55, you can also optimize a network for performance within a specified memory limit. For example, consider a scenario where the key model for your design is the [TFLM person detection model](#). You would like to know the amount of SRAM memory required by the Ethos-U to accelerate the inference. Compile the model for 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
```

Vela produces a memory footprint summary as follows:

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

Total Off-chip Flash used shows the size of the Read-Only data stored in the Flash in the case of the Ethos-U55, 263.55 KiB.

Total SRAM used shows the peak SRAM usage of the Ethos-U NPU for the inference, 72.72 KiB. However, the Vela compiler only has visibility of the input, output, and intermediate tensors of the model running on the Ethos-U NPU. The compiler does not know anything about other memory usage in your system, for example the kernels used to compile TFLM, RTOS memory usage, or memory allocated by the embedded application code. The system needs enough memory for both the inference itself and the rest of the software stack. This means that the SoC needs more than 72.72 KiB of SRAM.

However, it is possible to accelerate the inference with a smaller amount of SRAM by using the `--arena-cache-size` option. For example, imagine you do not want to sacrifice 72.72 KiB of your memory budget solely for the inference. You want to perform the inference in no more than 60 KiB. You can schedule the execution of the network in less than 60 KiB by using the `--arena-cache-size 61440` Vela command-line option when optimizing the model.

Note that with lower SRAM usage, the NPU must re-read more weights from the memory wired to AXI1. You should therefore ensure that your memory can deliver the required bandwidth on AXI1. To help designers get better insight of the memory access throughput, the Ethos-U NPU provides a Performance Monitoring Unit (PMU).

4.1.5.2 PMU counters in the Ethos-U NPU

The PMU provides counters for measuring hardware events such as memory accesses on specific memory interfaces. For example, if you run the ML model on the Corstone-300 FVP or FPGA you can read the following PMU counters after processing is complete:

- `axi0_rd_data_beat_received`
- `axi0_wr_data_beat_written`
- `axi1_rd_data_beat_received`

These performance counters report the number of beats that were transferred on the two AXI interfaces, allowing you to deduce the expected bandwidth. When the Ethos-U carries out a memory transaction, for example a read request, it reads data in beats. The size of each beat is configurable, as follows:

- The Ethos-U55 can be configured to use either 64-byte or 128-byte beats.
- The Ethos-U65 can be configured to use 64-byte, 128-byte, or 256-byte beats.
- The Ethos-U85 can be configured to use 64-byte, 128-byte, or 256-byte beats.

The number of beats is configured via the `max_beats` field of the `AXI_LIMIT` registers.

For example, if you configure an Ethos-U55 to access memory in 64-byte beats, and the NPU needs to read 50 bytes of data, the system must still perform a 64-byte transaction. However, the Ethos-U is designed to issue memory transactions as close as possible to the maximum configured beat size.

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.

4.1.5.3 Using PMU counters to determine memory bandwidth

Consider compiling a network that maps fully to the Ethos-U with Vela's `performance` scheduler. The following performance data is obtained from the Ethos-U's PMU:

PMU counter	Value
ETHOSU_PMU_NPU_ACTIVE + ETHOSU_PMU_NPU_IDLE	649597
ETHOSU_PMU_AXIO_RD_DATA_BEAT_RECEIVED	222602
ETHOSU_PMU_AXI1_RD_DATA_BEAT_RECEIVED	56511
ETHOSU_PMU_AXIO_WR_DATA_BEAT_WRITTEN	88584

Assuming a frequency of 500MHz, the total number of NPU cycles translates to $500\text{MHz} / 649597 = 769$ inferences per second.

The PMU counter results on AXIO translate to an average bandwidth of $769 * (222602 + 88584) * 8 / (1024 * 1024) = 1825 \text{ MB/s}$.



Multiply by 8 to convert to bytes and divide by $1024 * 1024$ to obtain MB

Note

On AXI1, the average bandwidth is $769 * 56511 * 8 / (1024 * 1024) = 331 \text{ KB/s}$.

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_AXIO_RD_DATA_BEAT_RECEIVED	97227
ETHOSU_PMU_AXI1_RD_DATA_BEAT_RECEIVED	146649
ETHOSU_PMU_AXIO_WR_DATA_BEAT_WRITTEN	32504

This time, again assuming a frequency of 500MHz, we see approximately 449 inferences per second with an average bandwidth of 444MB/s on AXIO and 502KB/s on AXI1. The SoC designer must ensure that the system is capable of delivering these bandwidths to achieve 449 inferences per second.

4.1.5.4 Memory modes

From the application's point of view, TFLM provides two memory regions that you can control:

- The tensor arena, containing read/write data
- The model, containing constant read-only data such as the weights or biases of the neural network

The Ethos-U55 supports the following placements of the tensor arena and the model:

1. The tensor arena is in the memory connected to the AXI0 interface, usually SRAM. The model is in the memory connected to AXI1, usually flash memory. 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 the 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 both reside in the memory connected to AXI1, usually DRAM. The memory connected to the AXI0 interface, usually SRAM, is only used as a cache to store the most frequently accessed tensors when performing 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 specifies the amount of SRAM available on your system.
2. The tensor arena and the model are both connected to the memory using the AXI0 interface. This configuration is called `sram_only`.



Note

For the Ethos-U55, the AXI1 interface is read-only and therefore the tensor arena must be placed in the memory wired to AXI0. On an Ethos-U65, the AXI1 interface is read/write, so the tensor arena can be placed in the memory connected to AXI1. The benefit of this is that you can store larger models in the DRAM and still obtain acceleration. For Ethos-U85, there can be up to 6 AXI interfaces and all of them support read/write. These interfaces are divided into two types: AXI_EXT for flash and DDR, and AXI_SRAM for on chip SRAM. For best performance, tensor arena should be placed in SRAM connected to AXI_SRAM.

4.2 Machine Learning Inference Advisor

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

The Machine Learning Inference Advisor (MLIA) helps developers design and optimize neural network models for efficient inference on Arm targets by enabling performance analysis and providing actionable advice early in the model development cycle. This advice includes information about supported operators, performance analysis, and suggestions for model optimizations such as pruning, clustering, and so on.

4.2.1 Requirements

Arm recommends using a virtual environment for MLIA installation. A typical setup for MLIA requires the following:

- Ubuntu 20.04.03 LTS
- Python 3.8.1 or higher
- [Ethos-U Vela compiler](#)

4.2.2 Installation

To install MLIA, run the following command:

```
pip install mlia
```

4.2.3 Usage

The command-line syntax for MLIA is as follows:

```
mlia [-h] [-v]

mlia check [-h] [--output-dir OUTPUT_DIR] -t TARGET_PROFILE
            [-b {armnn-tflite-delegate,vela}] [--performance]
            [--compatibility] [--json] [-d]
            model

mlia optimize [-h] [--output-dir OUTPUT_DIR] -t TARGET_PROFILE [-b {vela}]
              [--pruning] [--clustering]
              [--pruning-target PRUNING_TARGET]
              [--clustering-target CLUSTERING_TARGET] [--json] [-d]
              model
```

Where the options are as follows:

-h, --help

Show this help message and exit

-v, --version

Show the program's version number and exit

check

Generate a full report on the input model

optimize

Show the performance improvements (if any) after applying the optimizations

model

TensorFlow Lite model or Keras model.

--output-dir `OUTPUT_DIR`

Specifies the directory where MLIA creates output directory `mlia-output` for storing artifacts such as logs, target profiles, and model files. The default is the current working directory.

--performance

Perform performance checks.

--compatibility

Perform compatibility checks. This is the default for `check`.

-b {armnn-tflite-delegate,vela}, --backend {armnn-tflite-delegate,vela}

Backends to use for evaluation.

--json

Print the output in JSON format.

-d, --debug

Produce verbose output

-t `TARGET_PROFILE`, --target-profile `TARGET_PROFILE`

Built-in target profile or path to the custom target profile. Built-in target profiles are `cortex-a`, `ethos-u55-128`, `ethos-u55-256`, `ethos-u65-256`, `ethos-u65-512`, and `tosa`.

--pruning

Apply pruning optimization.

--clustering

Apply clustering optimization.

--pruning-target `PRUNING_TARGET`

Sparsity to be reached during optimization. The default is 0.5.

--clustering-target `CLUSTERING_TARGET`

Number of clusters to reach during optimization. The default is 32.

Run `mlia -h`, `mlia check -h`, or `mlia optimize -h` to see complete descriptions of these options.

The help output also shows which targets and backends are supported, as follows:

Supported Targets/Backends:

Target	Backend(s)	Status	Advice: comp/perf/opt
Cortex-A <cortex-a>	Arm NN TensorFlow Lite delegate <armnn-tflite-delegate...>	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	BUILTIN NOT INSTALLED	YES/YES/YES

	<corstone-310> Corstone-300 <corstone-300>	NOT INSTALLED	
TOSA <tosa>	TOSA Checker <tosa-checker>	NOT INSTALLED	YES/NO/NO

Use the `mlia-backend` command to install backends.

4.2.4 Command examples

Run `mlia -h` first, to validate that you have the necessary backends installed:

```
mlia -h
```

The following example shows how to invoke MLIA for Ethos-U on a Corstone-300 based device, using the `--backend` option to specify the backend:

```
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

For detailed information about Arm Virtual Hardware (AVH), its capabilities, and how to utilize them effectively, refer to the [AVH Solutions Overview](#). It provides a comprehensive walkthrough of the setup process, explains the navigational structure of the AVH platform, and offers essential technical details to make best use of the system.

The AVH Solutions Overview includes the [Get Started Example](#), which provides a step-by-step guide to setting up a Continuous Integration (CI) workflow for testing and debugging embedded applications using Arm Virtual Hardware (AVH).

The [Get Started Example](#) includes the following 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](#): Outlines the prerequisites required to run the example project.
- [Develop tests](#): Introduces the concept of developing unit tests using the Unity Framework, including an example project.
 - [Create repository on GitHub](#): Explains the process of creating a GitHub repository by either creating a new one or forking one from the `AVH-GetStarted` repository.

- [Setup local project on your PC](#): 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](#): Describes how to implement tests using the Unity Framework, how to redirect standard output to be visible during the debug session, and how to build and execute the program in Keil MDK.
- [Setup CI pipeline](#): Provides information about setting up the CI pipeline which is triggered on every code change using push and pull requests. The CI implementation in the example is implemented with GitHub Actions.
 - [AWS setup](#): Explains the process of setting up AWS to enable the execution of the example CI pipeline on a cloud-hosted Arm Virtual Hardware instance.
 - [GitHub Actions setup](#): Explains how to run Arm Virtual Hardware with GitHub Actions.
- [Execute CI](#): Describes how you can manually trigger execution of the configured workflow.
- [Analyze failures](#): Explains how to view and analyze the example workflow execution to identify and fix problems.

4.4 SDS Framework

A large set of representative and qualified data is a pre-requisite for effective ML algorithm selection, training, and validation.

As explained in [Overview of the ML development process](#), ML algorithms can only make correct decisions in areas where training data exists. Capturing a large body of real-world data is therefore an important task.

Arm developed the Synchronous Data Stream (SDS) Framework to capture real-world data from sensors and audio sources. The SDS Framework contains the following 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](#) describes the content of SDS data files along with scaling and formatting information.
- [SDS Utilities](#) are tools to record, convert, and display SDS data files.
- [SDS Playback](#) uses Arm Virtual Hardware with the Virtual Streaming Interfaces (VSI) to stimulate algorithms under development with real-world data.

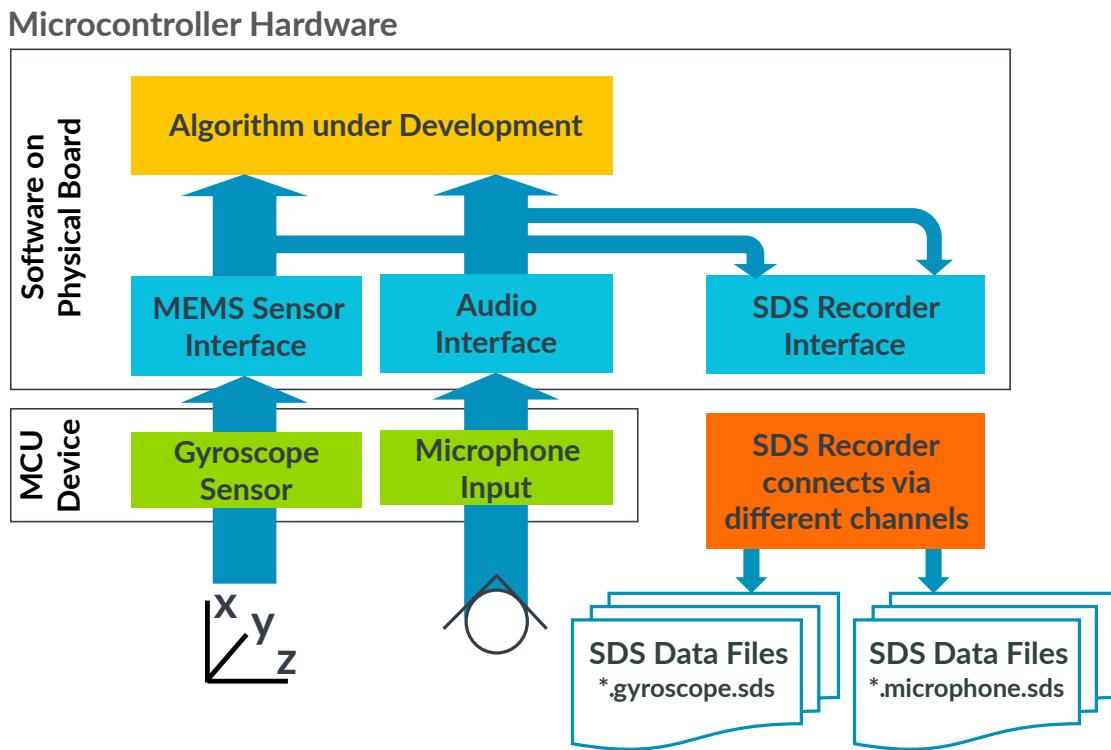
4.4.1 SDS Recorder Interface

The SDS Recorder Interface allows real-world data to be captured and recorded in SDS data files. SDS Recorder forms part of the target application and enables data streaming using various interfaces such as TCP/IP over Ethernet, UART, or USB. SDS Recorder can also capture data in

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

The following diagram shows an example system using the SDS Recorder Interface:

Figure 4-2: SDS Recorder Interface



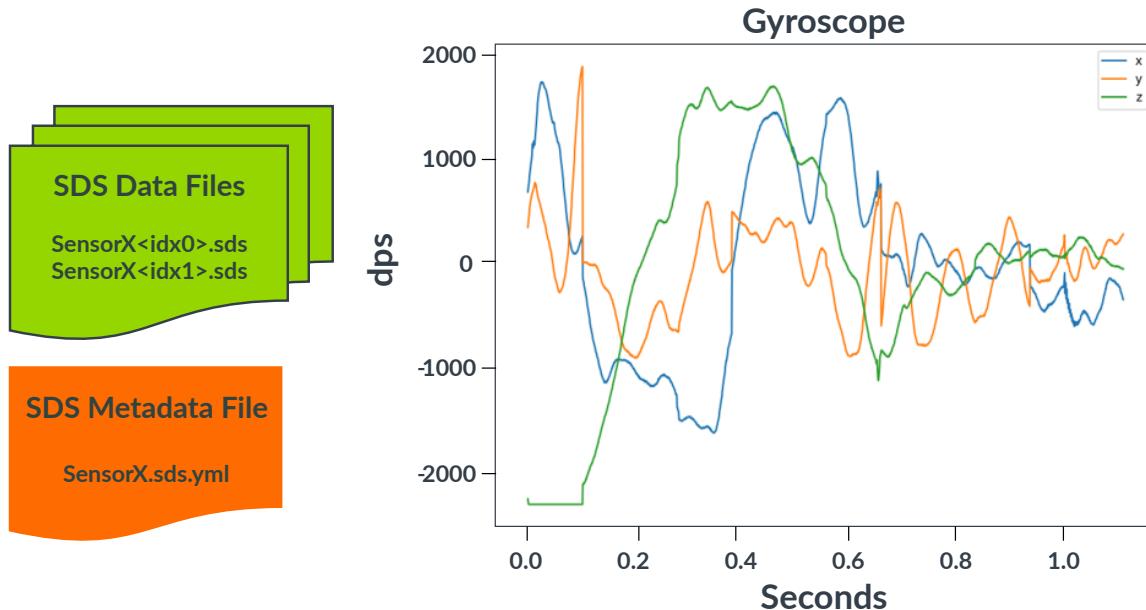
In this example, the SDS Recorder Interface captures real-world data from the gyroscope and microphone sensors on the device. SDS Recorder connects using multiple different channels and records the data in separate SDS data files.

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. SDS Metadata also identifies

the data streams for input to DSP design utilities, MLOps development workflows, and AVH data playback.

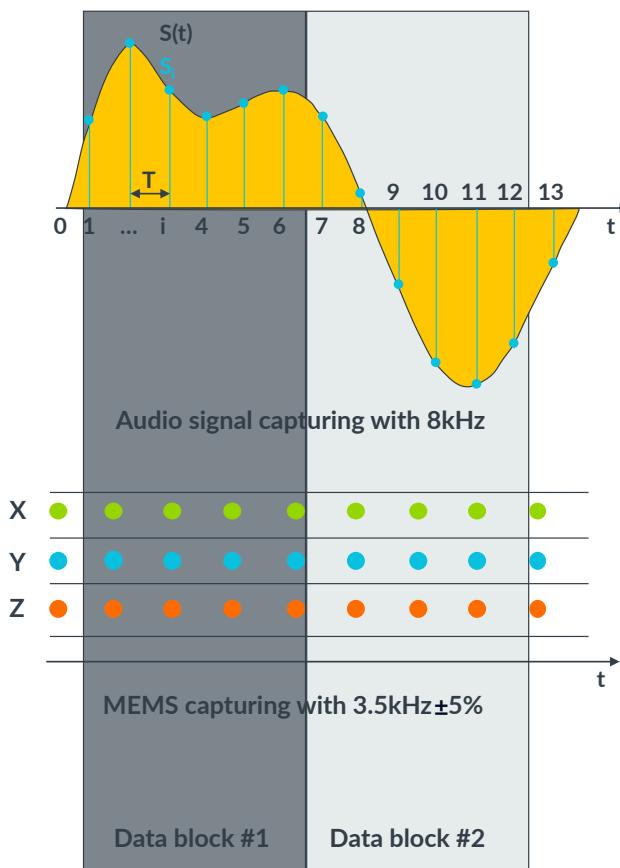
Figure 4-3: SDS Metadata file



The SDS data files have multiple uses, including the following:

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

Sensors may have independent clock sources with different tolerances which can result in different size records for block procession algorithms. The following diagram shows how this is possible, with data blocks 1 and 2 containing different numbers of samples:

Figure 4-4: SDS sample frequencies

[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

See [SDS Recorder in the GitHub repository](#) for access to examples.

4.4.3 SDS Utilities

The SDS Utilities let you analyze and convert SDS data files.

See [SDS Utilities in the GitHub repository](#) for more information about and access to these utilities.

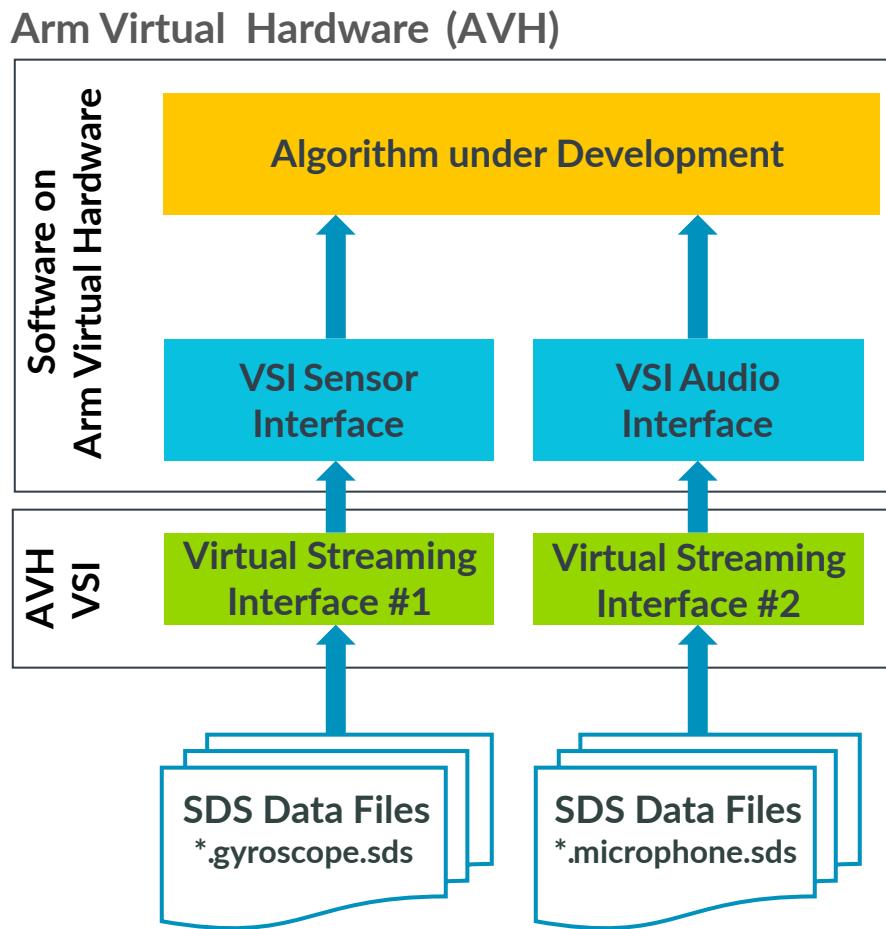
4.4.4 SDS Playback

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, AVH supports test case development and verification of algorithms. With DevOps systems such as GitHub Actions, AVH supports continuous integration workflows for build and test automation.

AVH provides virtual streaming interfaces that playback SDS data to an algorithm under development. This is useful for repeatable validation tests or ML model verification. 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.

The following diagram shows how AVH can use SDS data files to stimulate an algorithm under development:

Figure 4-5: SDS on Arm Virtual Hardware



The SDS framework can also support sensor fusion applications, which combine data from multiple different sources. For example, combining an audio signal with information from a MEMS sensor might 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.

See [SDS Playback in the GitHub repository](#) for more information.

5. The Arm ML Zoo

Creating a new Neural Network (NN) model for an application is an expensive process. It requires a deep understanding of Machine Learning (ML), and it can take a very long time to optimize the network for a specific use case.

Fortunately, many different NN models have already been developed for a variety of different applications, and many of these models are available in online databases called model zoos. These models can provide a useful starting point for further development. When an application developer needs to create an ML application, it might not be necessary to create a new NN model from scratch.

The [ARM ML Zoo repository](#) hosts a variety of ML models that are optimized for ARM IP. The models cover a range of different applications, including the following:

Anomaly Detection

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.

Image Classification

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.

Keyword Spotting

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.

Noise Suppression

RNNoise is a noise reduction network, that helps to remove noise from audio signals while maintaining any speech. This is a TFLite quantized version that takes traditional signal processing features and outputs gain values that can be used to remove noise from audio.

Object Detection

Two ML models are available: SSD MobileNet v1 (with variants for various data types: fp32, int8, uint8) and Yolo v3 Tiny (fp32). SSD MobileNet v1 is an object detection network, that localizes and identifies objects in an input image of 300x300 pixels. Yolo v3 Tiny is an object detection network (using the fp32 data type) that takes an image of 416x416 pixels and outputs detections for the image.

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.

Superresolution

This includes the SESR model optimized for INT8 and compatible with TensorFlow Lite. It is suitable for Cortex-A and Mali GPU.

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 re-train the ML model to fit the specific needs of their ML applications.

5.1 Integrating an Arm 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 ML applications for the 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 an Ecosystem FPGA (MPS3) and Fixed Virtual Platform (FVP) to allow development ahead of hardware availability.

The Arm ML Evaluation Kit runs in a Linux environment, although it is possible to use [Windows Subsystem for Linux](#).



Another option for evaluating the Ethos-U NPUs is the [CMSIS-Pack based Machine Learning Examples](#). This contains several examples which use CMSIS-Pack to handle software integration. The CMSIS-Pack based Machine Learning Examples can be used in both Linux and Windows environments.

6.1 Getting started with the ML Embedded Evaluation Kit

This guide shows you how to build the examples in the [ML Embedded Evaluation Kit](#).

For a list of the examples contained in the ML Embedded Evaluation Kit, see [Use case APIs](#) in the Git repository.

A [Quick Start Guide](#) is also available, which shows you how to build and run the keyword spotting example application.

6.1.1 Supported platforms

The Machine Learning Evaluation Kit is compatible with several different platforms. These platforms range from physical hardware such as 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 different capabilities and advantages, making the ML Evaluation Kit a versatile tool for a variety of ML explorations.

The supported platforms are as follows:

- Arm [MPS3 FPGA board](#) with one of the following FPGA images:

- [AN552](#) FPGA image. AN552 is based on the [Corstone-300](#) subsystem containing the [Arm Cortex-M55 processor](#) and the [Ethos-U55 NPU](#).
- [AN555](#) FPGA image. AN555 is based on the [Corstone-310](#) subsystem containing the [Arm Cortex-M85 processor](#) and the [Ethos-U55 NPU](#).

You can [download these FPGA images from Arm Developer](#).

- [Fixed Virtual Platform \(FVP\)](#) for Corstone-300
- [Arm Virtual Hardware \(AVH\)](#) for Corstone-300
- [Arm Virtual Hardware \(AVH\)](#) for Corstone-310

6.1.2 System and software requirements

See the [Arm ML Embedded Evaluation Kit documentation](#) to read full details of the prerequisites for running the ML Embedded Evaluation Kit.

The key requirements are as follows:

1. An x86 Linux system or Windows Subsystem for Linux.
2. Python version 3.9 or newer.

You can check the version of Python on your system using the following command:

```
python3 --version
```

If the Python version on your system is 3.8 or earlier, you can update the Python version by following the instructions in the [Arm ML Embedded Evaluation Kit documentation](#).

3. Several common software tools, which can be installed 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
```

4. Python virtual environment.

To install Python virtual environment for python 3.9, run the following command:

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

To install Python virtual environment for python 3.10, run the following command:

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

5. A compilation tool chain, one of the following:
 - Arm Compiler for Embedded 6.16 or later

For more information about Arm Compiler for Embedded, see [Arm Developer](#). Arm Compiler for Embedded requires a valid license.

- GNU Arm Embedded toolchain 10.2.1 or later

If you need to install the GNU Arm Embedded toolchain, download a suitable version from one of the following locations:

- <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, 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, for example `/mnt/c/Program Files/Microsoft VS Code/bin`. This causes [problems with the build script](#). To solve the problem, you can use one of the following workarounds:

- Create a bash script to remove Windows paths from the `$PATH` variable
- Add escape characters `\` before all spaces in the path variable
- Enclose the paths with quotes

6.1.3 Check out the repository

Run the following commands to check out the repository:

```
git clone "https://git.gitlab.arm.com/artificial-intelligence/ethos-u/ml-embedded-evaluation-kit.git"
cd ml-embedded-evaluation-kit
git submodule update --init
```

6.1.4 Compile the default projects

Run the compilation script using one of the following commands, depending on the toolchain you are using:

- GNU Arm Embedded toolchain (gcc):

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

- Arm Compiler for Embedded:

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

After the compilation finishes, the executables are located in `cmake-build-xxxxxx/bin`.

To run the example projects, follow these instructions in the [Arm ML Embedded Evaluation Kit documentation](#).

The `build_default.py` script builds the tests for the default configuration. The script supports several command line options. For example, to specify the configuration of the Ethos-U55 hardware to be 32 MAC, you can use one of the following commands:

- GNU Arm Embedded toolchain (gcc):

```
python3 ./build_default.py --npu-config-name ethos-u55-32
```

- Arm Compiler for Embedded:

```
python3 ./build_default.py --npu-config-name ethos-u55-32 --toolchain arm
```

The following are valid options for the Ethos-U configurations:

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

For a complete list of the `build_default.py` script command-line options, see the [Arm ML Embedded Evaluation Kit documentation](#).

6.1.5 Additional resources

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 \(Implementing custom ML application\)](#)

- [FAQ](#)
- [Troubleshooting](#)
- [Memory considerations](#)
- [Testing and benchmark](#)
- [Timing adapter](#)
- [CMAKE presets](#)
- [Coding standards and guidelines](#)
- [Appendix](#)

The following Arm blogs provide useful information about the Arm Machine Learning Evaluation Kit:

- [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](#)

The following Arm Learning Paths are related to the Arm Machine Learning Evaluation Kit:

- [Navigate Machine Learning development with Ethos-U processors](#)
- [Build and run the Arm Machine Learning Evaluation Kit examples](#)

6.2 Beyond the basics

This section of the guide provides the following information:

- An overview of the software components of the kit, including the TensorFlow Lite Micro runtime and the Ethos-U driver.
- The structure of the repository
- The build process and the key actions performed by the compilation script.
- Available build options and how they can be customized according to the user's needs.

[Getting started with the ML Embedded Evaluation Kit](#) introduced the ML Evaluation Kit, its purpose, and its potential applications. This guide also explains the fundamental concepts of machine learning that are crucial for understanding the workings of the ML Embedded Evaluation Kit.

6.2.1 The build process

Several existing resources describe the build process:

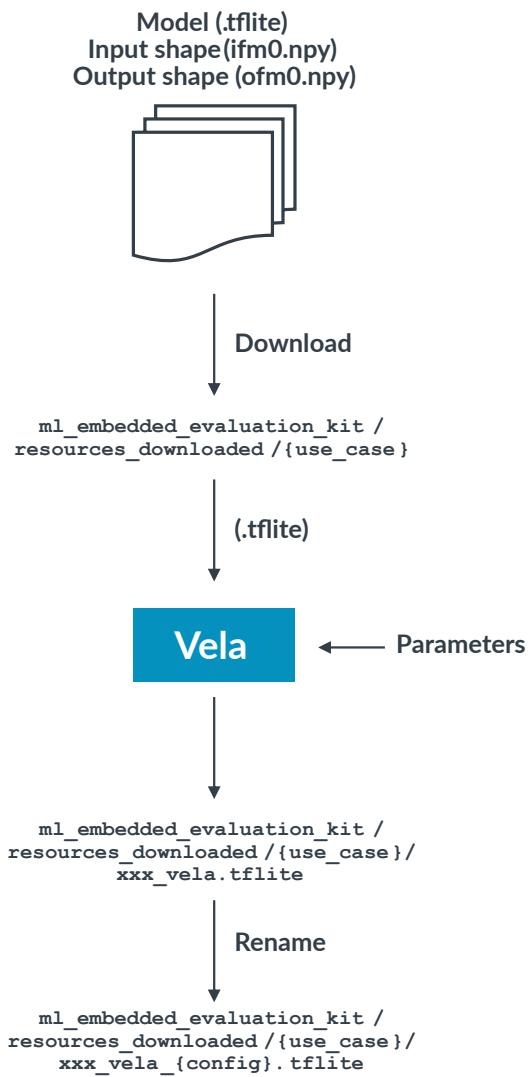
- [The structure of the repository](#)

- [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](#)
 - [Configuring the build for simple platform](#)
 - [Build the application](#)

The compilation script `build_default.py` performs the following key actions:

1. Download the TFLM models for each use case and optimize them using Vela, as shown in the following diagram:

Figure 6-1: Model download and optimize



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

These software components include the following:

- The FlatBuffers library from TensorFlow Lite Micro
- kissfft, a Fast Fourier Transform library
- pigweed, a collection of embedded-targeted libraries
- gemmlowp, a general matrix multiplication library
- ruy, a matrix multiplication library

3. Setup the build environment for each use case

This process includes setting up directories, creating the `MAKEFILE`s, 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 <code>.tflite</code>	<code>ml-embedded-evaluation-kit/scripts/py/gen_model_cpp.py</code>
ML data labels <code>.txt</code>	<code>ml-embedded-evaluation-kit/scripts/py/gen_labels_cpp.py</code>
Audio input data <code>.wav</code>)	<code>ml-embedded-evaluation-kit/scripts/py/gen_audio_cpp.py</code>
Image input data <code>.bmp</code>)	<code>ml-embedded-evaluation-kit/scripts/py/gen_rgb_cpp.py</code>

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



You can also use the `xxd` utility to convert `.tflite` model files to a byte array in C/C++.

4. Build the executables

The source codes are compiled at this stage. If needed, additional software components such as the CMSIS-DSP library source are downloaded.

6.2.2 Build options for `build_default.py`

For a complete list of build options, see the [Arm ML Embedded Evaluation Kit documentation](#).

For example, you can use the following `ETHOS_U_NPU_MEMORY_MODE` settings to define the memory type used by Ethos-U NPU:

- `Shared_Sram`. This is the default for Ethos-U55 NPU, and is available for both Ethos-U55 & Ethos-U65.
- `Dedicated_Sram`. This is the default for Ethos-U65 NPU, and is available for Ethos-U65 only.
- `Sram_Only`. This is 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 software components of the Machine Learning Evaluation Kit include the following:

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.

TensorFlow Lite Micro

The executable program image contains the TensorFlow Lite Micro runtime, which is downloaded by the `build_default.py` script. After being downloaded, the files are located in the `ml-embedded-evaluation-kit/dependencies/tensorflow` directory.

The TensorFlow Lite Micro kernel provides support for Ethos-U. For information about software integration and initialization, see `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 setup the model before invoking the micro-interpreter. For information about the minimal code required to setup and execute TensorFlow Lite Micro, see [Get started with microcontrollers, in the TensorFlow documentation](#). It contains a good overview of the TensorFlow Lite Micro low-level operations.

Ethos-U Driver

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

Platform code

The platform support code can be 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 does not use the driver code from the third-party repository because it needs extra flexibility to support multiple hardware targets such as Corstone-300 and Corstone-310. Platform support code can be found in the following locations:

Location	Note
<code>ml-embedded-evaluation-kit/source/hal/source/platform/mps3/</code>	The platform code used by the example projects.
<code>ml-embedded-evaluation-kit/dependencies/tensorflow/tensorflow/lite/micro/cortex_m_corstone_300</code>	From Google TensorFlow. Not used.
<code>ml-embedded-evaluation-kit/dependencies/core-platform/targets/corstone-300</code>	From gitlab.arm.com/artificial-intelligence/ethos-u/ethos-u-core-platform . Not used.

Details of Ethos-U integration, for example the base address and IRQ assignments, are available in this file: `ml-embedded-evaluation-kit/source/hal/source/platform/{platform}/CMakeLists.txt`.

The linker scripts are in this location: `ml-embedded-evaluation-kit/scripts/cmake/platforms/{platform}`.

6.2.4 Creating custom applications with the ML Embedded Evaluation Kit

For information about creating custom applications running in the ML Embedded Evaluation Kit, see [Implementing custom ML application](#).

You can also [add custom platform support](#).

7. CMSIS-Pack based ML 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 also provides a set of examples based on CMSIS-Pack, a software component packaging solution. You can download the CMSIS-Pack based ML examples from the [Arm-Examples Github repository](#).

The CMSIS-Pack based ML examples provide the following benefits:

- Uses CMSIS-Pack as a software integration mechanism, which is more suitable for IDE development environments.
- Provides a greater choice of hardware support.
- Supports the Windows environment.

Note that the CMSIS-Pack based ML examples repository is a work-in-progress. Current limitations include the following:

- ML use cases are limited to Key Word Spotting (KWS) and Object Detection
- Only supports Arm Compiler 6
- Ethos-U NPU is not configurable

7.1 Prerequisites

The CMSIS-Pack based ML examples require the following tools:

- Arm Compiler for Embedded

Download [Arm Compiler for Embedded](#) and refer to the [Release Notes](#) for installation information.

- CMSIS-Toolbox

To install [CMSIS-ToolBox](#), [download release 2.0 or higher](#) and refer to the [installation documentation](#).

After installing CMSIS-Toolbox, ensure that:

- The CMSIS-Toolbox binaries are in the search path.
- The environment variables for the toolchain installation path is set, for example `AC6_TOOLCHAIN_6_19_0=C:/Keil_v5/ARM/ARMCLANG/bin.`
- The environment variable `CMSIS_PACK_ROOT` contains the path to the CMSIS-Pack root directory containing the software packs.
- The CMSIS-Pack Root directory is initialized. See the [cpackget documentation](#) for more information. For example, `cpackget init --pack-root path/to/new/pack-root https://www.keil.com/pack/index.pidx.`

- The environment variable `CMSIS_COMPILER_ROOT` contains the path to the `etc` directory in CMSIS-Toolbox, for example `{install_path}/etc`.
- `git`, `cmake`, `make`, and `ninja`

On a Linux system, install the tools with the following command:

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

On a Windows system, 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

- A suitable IDE

Arm recommends that you use [Visual Studio Code IDE](#) with [Keil Studio Pack Extension](#).

Alternatively, you can also use [Keil Studio Cloud](#).

7.2 Compiling the CMSIS-Pack based ML examples

The CMSIS-Pack based ML examples workflow uses CMSIS-Toolbox to generate compilation setups from each example's Yaml file. The flow consists of the following steps:

1. Analyze the Yaml file to produce a list of any missing CMSIS-Packs:

```
csolution list packs mlek.csolution.yml -m > packlist.txt
```

2. Install any missing CMSIS-Packs:

```
cpacket add -f packlist.txt
```

3. Generate the `.cprj` files from the main Yaml file for each of the example use-cases:

```
csolution convert ./mlek.csolution.yml
```

By default, this command generates several `.cprj` files for different 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 either Debug or Release.

If required, you can generate one specific combination using the `-c` options, as follows:

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

4. Compile the project, using either `cbuild` or an IDE.

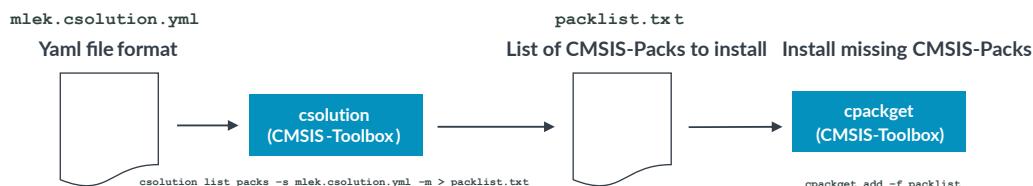
For example, to compile the KWS example using `cbuild`:

```
cbuild ./kws/kws.Debug+AVH-SSE-300-U55.cprj -g "Unix Makefiles" -j 4
```

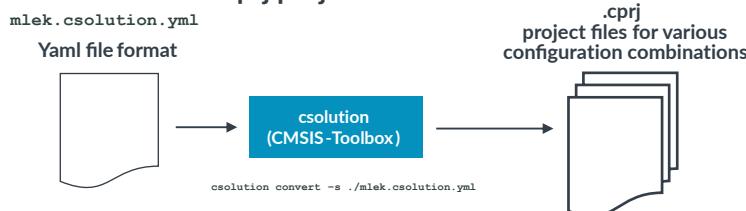
The following diagram shows this workflow:

Figure 7-1: Software workflow for the CMSIS-Pack based ML examples

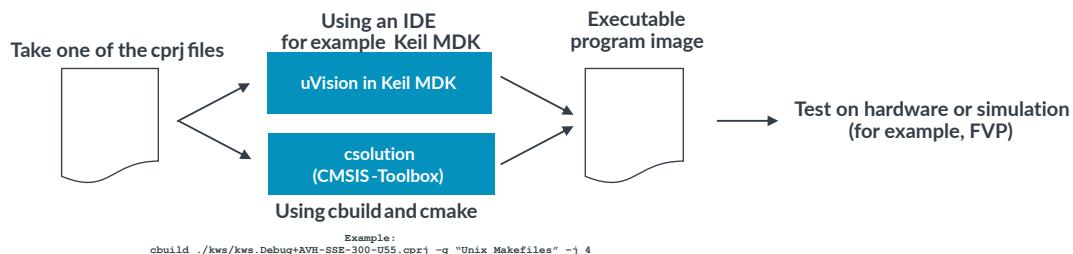
Step 1: If needed, install the missing CMSIS-Pack



Step 2: Generate the .cprj project files



Step 3: Compile the project



The generated executable can be tested on either target hardware or in a simulation environment. For more information, see the [CMSIS-Pack based ML examples documentation](#).

Note that the MPS3 FPGA image for Cortex-M55 (AN552) is not supported because the Ethos-U configuration in the FPGA image (128 MAC/cycle) does not match the NN model (256 MAC/cycle). However, you can still test the generated application using Arm Virtual Hardware (AVH) or Fixed Virtual Platform (FVP).

Unlike the ML Embedded Evaluation Kit, the ML models and input samples have already been converted to C++:



- [KWS](#)
- [Object detection](#)).

There is therefore no need to use the Vela Compiler. Because of this, the Ethos-U configurations supported by the ML models are fixed.

7.3 Using TFLM CMSIS-Packs in your own project

The information in this section of the guide helps you to integrate TensorFlow Lite Micro (TFLM), the Ethos-U driver and ML model into your own project.

The steps in the process are as follows:

- [Add the TFLM software components](#)
- [Add the ML model to your project](#)
- [Use the TFLM API](#)

These steps are discussed further in the following sections.

7.3.1 Add the TFLM software components

The TFLM 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 built from sources that are maintained and versioned by Arm on [gitlab.arm.com](#).

Add the 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 the following software components.
The dependencies can be resolved in the IDE:
 - component: Arm::Machine Learning:NPU Support:Ethos-U Driver&Generic U55
 - 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



The Ethos-U Driver default variant shown here 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 Add the ML model to your project

There are two options to store your ML model on the embedded target:

- Store the ML model on an existing filesystem.

The TensorFlow library can interpret .tflite files as they are stored on filesystem. This is useful if you 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. The following simple code snippet shows how to 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 requires a significant amount of RAM and is not suitable for all system designs. A typical scenario for this usage model is a test system based on Arm Virtual Hardware, where you load many different model variants for profiling. Even if a file system is available on your hardware target, you may want to store the model in ROM alongside the application.

- Compile the model into the firmware image and flash it together with the application.

To compile the content of a `.tflite` file into your firmware image, it needs to be represented as an array in C language syntax. Use a hexdump utility such as `xxd` to convert the binary `.tflite` file into a header file `my_network_model.h` to include in your project, as follows:

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

To ensure that the data is stored in ROM memory and starts at a 16 byte alignment boundary, define the data array as follows:

```
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 Use the TFLM API

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

First, we must initialize the Ethos-U NPU. 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 as follows:

- `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](#)

Once the Ethos-U NPU is initialized, we can access the Ethos-U using the TensorFlow Lite runtime. Ethos-U support is integrated into the TFLM runtime. The application code therefore calls the TFLM interpreter to use the Ethos-U NPU, as follows:

```
#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, refer to the [TensorFlow Guide](#) for more information about the workflow.

Once you have created your ML application, the next step you take might be to profile and optimize the ML model. Arm provides several tools to help you test and improve your model's performance running on Ethos-U NPUs. For more information, see [Profiling and optimizing ML models](#).

8. Profiling and optimizing ML models

The memory requirements and code size of a TensorFlow Lite Micro (TFLM) model depend on the model's parameters, layers, and the hardware platform on which it runs. To estimate the model's memory requirement and code size, use the following tools:

1. Model analysis.

Begin by examining the structure of the TFLM 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](#)

2. TFLM Profiler.

TFLM provides a profiler tool that lets you measure memory usage and code size. You can find example code and instructions in the TFLM documentation.

3. Performance counters.

Ethos-U provides performance counters that let you analyze code and memory usage.

4. Memory estimation functions.

TFLM provides memory estimation functions that let you estimate memory requirements programmatically. These functions help you calculate memory usage based on model parameters and tensor shapes.

5. Compilers and linkers.

Arm Compiler, LLVM, and GCC can all produce reports about code and object sizes.

6. Static analysis.

Static analysis tools can provide information about the code size and memory usage of your TFLM model without running it.

8.1 Ethos-U Vela optimizations

The Ethos-U Vela compiler optimizes neural network models. Vela provides several command-line options that influence model optimization. For more information, see [Optimize custom model with Vela compiler](#) in the Arm ML Evaluation Kit documentation.

8.2 Operator mapping and usage

Running Ethos-U Vela with the `--verbose-performance` option displays information about TensorFlow operator usage in your model.



These data values are estimates and are not cycle-accurate numbers based on running the inference on silicon.

The following is example output from Vela with the `--verbose-performance` option:

Figure 8-1: Vela verbose output example

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	86.19	1888813	46.80	1888813	21504	0	0	0	99090402	48.89	20.48	ResNet18/activation_32/Relu
CONV_2D	Conv2DBias	730624	100.00	2127584	52.69	2127584	21504	0	0	0	99090402	49.99	18.19	ResNet18/batch_normalization_32/BatchN
ADD	Add	43008	5.89	16129	0.40	16128	8064	0	0	0	0	0.00	0.00	ResNet18/activation_33/Relu
AVERAGE_POOL_2D	AvgPool	27648	3.78	4224	0.10	3200	4224	0	0	0	24576	0.01	2.27	ResNet18/average_pooling2d

In this example:

- Cycles and Network% give a good indication of which layers are compute- and memory-intensive.
- A MAC Count of 0 indicates that an operator was not off-loaded to the Ethos-U NPU. These operators are still optimized by CMSIS-NN typically, but consume CPU processing time. If this is the case for the majority of your network, the model used is not suitable for Ethos-U.

8.3 MLIA guided optimizations (Experimental)

ML Inference Advisor (MLIA) is a tool that assists AI developers in designing and optimizing neural network models for efficient inference on Arm targets. MLIA is ideal for evaluating whether a network maps fully to Ethos-U. In addition, MLIA enables performance analysis, providing 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 provides the following sub-commands:

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

When MLIA is setup with a performance model such as the Corstone Fixed Virtual Platform (FVP), it provides performance information about 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 accurate to within +/-10%. The error margin can be higher when other performance models, such as the Corstone-310 FVP, is used.

MLIA is a wrapper around other tools and components, and does not include its own performance model. This is why an FVP is needed to obtain performance data. In contrast, Vela includes a performance estimator, but the performance estimation in Vela is not cycle-accurate. Vela performance estimate values are intended to be used for relative comparisons 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, or even to prototype a full application with pre- and post-processing, then the 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.

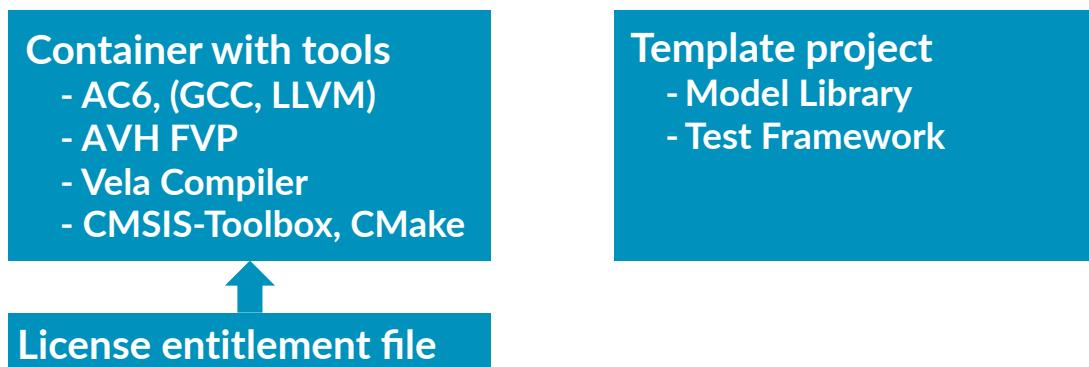
9. MLOps systems

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.

For an introduction to the MLOps process steps for developing machine learning applications, see [Overview of the ML development process](#).

The following diagram outlines the tools and template projects provided by Arm to optimize MLOps processes for 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.

9.1 License activation

To activate the Arm development tools in an MLOps system a license is required. [Contact Arm](#) for further details, and for information about how to become an Arm MLOps partner.

The following command, run outside of the container, activates the MLOps system license:

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

The `activate` command generates a license file that is imported into the container with the command `armlm import`. See [User-based Licensing User Guide](#) for more information.



The `activate` command must be run at least once every 24 hours to keep the license up-to-date.

9.2 Example projects

The repository contains several example projects that show how to create a library with a trained ML model, and how to evaluate the performance.

The example projects use the `MLOps.csolution.yml` file in [CMSIS-Toolbox](#) format.

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 several different compilers, including Arm Compiler 6, GCC, and LLVM. It defines `target-types` and `build-types` that represent different processors as follows.

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

target-type	Selects the target processor
+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
.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 different processors and compiler toolchains. For example, the following CMSIS-Toolbox `cbuild` command creates a library for a Cortex-M7 processor, optimizing for code size, using the GCC toolchain:

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

The example projects show how to run the test process using the [AVH VSI interfaces](#) for Audio, SDS, Video.

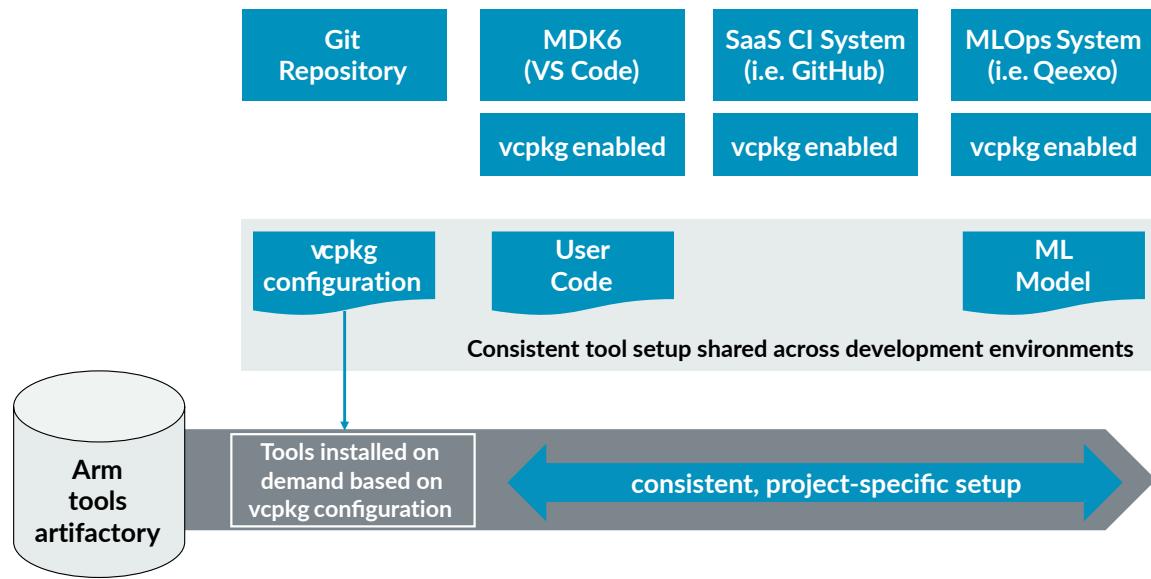
A CMSIS-Pack template shows how to deliver the Model Library to IDEs for integrating the ML model in embedded projects. This pack contains source code templates that ease integration with application programs.

9.3 vcpkg

You can also use the Microsoft tool `vcpkg` to manage tools. See “[Install tools on the command line using vcpkg](#)” for further details.

The following diagram shows how you can use `vcpkg` to manage consistent tool installation from the Arm tools artifactory:

Figure 9-2: Arm Tools Artifactory



Currently the vcpkg process is experimental and we therefore recommend that you 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 resources](#)
- [Partner solutions](#)

10.1 Product pages

The following product pages are available for Ethos-U:

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
Ethos-U85	https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u85
	https://developer.arm.com/Processors/Ethos-U85

10.2 Product document

The following product documentation is available for Ethos-U:

Product	Resource
Ethos-U	Arm Ethos-U Processor Series
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
Ethos-U85	Product Brief
Ethos-U85	Technical Reference Manual

10.3 Software and examples

Open-source software components and documentation for developing Ethos-U NPU software:

- <https://gitlab.arm.com/artificial-intelligence/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 Cortex-M55 CPU, Ethos-U55 NPU and CMSIS-NN
 - Documentation: Running TVM on bare metal Arm Cortex-M55 CPU and Ethos-U55 NPU with CMSIS-NN
- Benefit of pruning and clustering a neural network for before deploying on Arm Ethos-U NPU
- Enabling AI at the Edge with Himax WE2 AI Processor
- Case study - A Family Trip, Endangered Species Sparks Innovative Arm-based Solution

10.4 Other resources

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 (slides)
Cortex-M55 + Ethos-U55	Arm M55 and U55 Performance Optimization for Edge-based Audio and Machine Learning Applications
TFLM	TFLM low-level operations
ML model optimization	Optimize a ML model for fast inference on Ethos-U microNPU

Product	Resource
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 solutions

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
Nota.AL	Nota AI's Collaboration with Arm's AVH Corstone-300 Ethos-U65
Edge Impulse	Announcing Official Support for the Alif Ensemble E7 Development Kit