Image Classification based on ExecuTorch

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example program uses the NuMaker-X-M55M1D evaluation board to implement an image classification and inference application based on the ExecuTorch.
BSP Version	M55M1 Series BSP CMSIS V3.01.003 ExecuTorch v1.0.1
Hardware	NuMaker-X-M55M1D

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

ExecuTorch is a lightweight PyTorch inference execution framework developed by Meta®. It is specifically designed for edge devices (Edge/Embedded/Mobile). Its goal is to enable PyTorch trained models to be deployed efficiently, securely, and controllably to resource constrained devices. Its design objectives are:

- Deplorability
  - Not dependent on Python
  - C++ Runtime
  - Able to be executed on Linux/Android/RTOS/MCU
- Cuttable
  - Compiles the actual operators (operators/kernels) used
  - Suitable for SRAM/Flash-constrained systems
- Controllability
  - Static graph
  - Removes the dynamic behavior (control flow / guards) from the original model diagram
- Scalability
  - Backend / Delegate Mechanism
  - Supports NPU / DSP

The NuMicro® M55M1 series microcontroller (MCU) features Arm® Cortex®-M55 processors running up to 220 MHz. In addition to the Cortex-M55 core, the MCU integrates an Arm Ethos™-U55 Neural Processing Unit (NPU) as an auxiliary processor to accelerate neural network operations. Additionally, it features a HyperBus interface, facilitating the expansion of RAM requirements during model computation.

ExecuTorch supports the Arm Ethos-U delegation mechanism, making it suitable for deploying neural network models on the M55M1 using the ExecuTorch framework, and accelerating neural network operation through NPU computation during inference.

1.1 Principle

The core of this example lies in using ExecuTorch to delegate the model to a PTE (PyTorch ExecuTorch format) inference file that supports Arm Ethos-U (NPU), enabling the model to accelerate inference using Ethos-U on the M55M1. The entire workflow is shown in Figure 1-1 and consists of the following three steps:

1. Model export: Converts a dynamic PyTorch program into a static analyzable graph. In this static analyzable graph, there are no control flow operators (if/loop), and the model shape/dtype must be static.
2. Compiling an ExecuTorch program: This process converts the static analysis graph into an ExecuTorch program that supports the target hardware. The ExecuTorch program is saved in the PTE static inference file format. For Ethos-U, this involves several parts:
   - Quantization: Since Ethos-U is an integer processor, quantization is necessary. Therefore, the functionality of the quantizer “executorch.backends.arm.quantizer” has been utilized for quantization.
   - Partitioning: This determines which parts of the model are accelerated by Ethos-U and which parts may fall back to the CPU. ARM's partitioning strategy aims to partition the entire graphics for execution on Ethos-U.
   - Ethos-U backend compilation: For Ethos-U partitions, after converting the IR (Intermediate Representation) to TOSA IR (ARM's Machine Learning Application Standard), it is fed into the Vela compiler to generate an Ethos-U command stream.
   - Lowering the format: Converting the intermediate representation of the graphics into a format used at runtime. This includes creating delegates (loaded at runtime). Ethos-U 's workload, memory planning, and serialization of the entire program into a single PTE (ExecuTorch Program) file.
3. Executing the ExecuTorch program on the device: The ExecuTorch runtime loads the ExecuTorch program on the device and performs model inference according to the instructions of the ExecuTorch program. When the runtime executes the Ethos-U backend operator in the model graph, it calls the Ethos-U driver to perform Ethos-U accelerator computation.

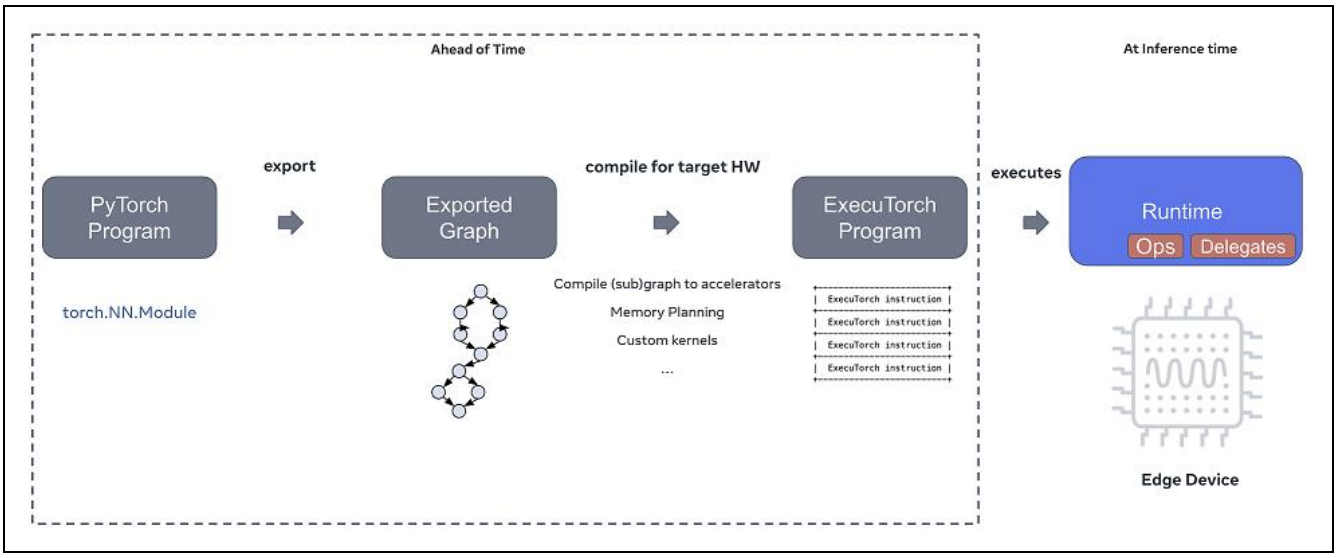

Figure 1-1 1Workflow (Source: How ExecuTorch Works)

The Step1 and Step2 described above are called Ahead-of-Time (AOT) compilation, which is an offline process. Therefore, an ExecuTorch environment needs to be set up on a PC to

execute the compilation script and export the model PTE file. For related environment setup instructions, please refer to the README.md of ExecuTorch on GitHub. This document will focus on the compilation script and runtime environment.

1.2 MobileNetV2 Introduction

MobileNetV2 is an efficient and lightweight convolutional neural network (CNN) architecture proposed by Google® in 2018. It represents a significant improvement over MobileNetV1, aiming to provide low-latency and high-accuracy visual models for mobile devices such as smartphones and embedded devices. Its core architecture consists of "inverted residuals with linear bottlenecks".

1.2.1 Core Architecture

MobileNetV2 employs inverted residual modules, which is the biggest difference from traditional residual networks (such as ResNet).

- Traditional residual networks (RestNet): It adopts a structure of "compression → convolution → dilation" (hourglass shape). First, the input dimension is reduced to decrease computational load, and then the high dimension is restored after processing.
- Inverted residual module: It adopts an "expansion → convolution → compression" structure (olive-shaped).
  - Expansion: First, use 1x1 convolution to expand the number of low-dimensional channels (usually by 6 times) to enter the high-dimensional feature space.
  - Depthwise: Lightweight deep convolutions in high dimensions allow the model to capture richer features.
  - Projection/Bottleneck: Projects features back into a lower-dimensional channel.

1.2.2 Key Technologies

- Linear Bottlenecks

  The research team discovered that using non-linear activation functions (such as ReLU) in low-dimensional space (bottleneck layers) leads to significant information loss. Therefore, after the final 1x1 convolution in the inverted residual module, ReLU is not used, and linear features are directly output. This ensures that the model retains crucial information when compressing data.

- ReLU6 activation function

  MobileNetV2 extensively uses ReLU6, which limits the output to [0, 6]. This prevents numerical overflow and improves the robustness of the model when using low-precision computations (such as Float16 or Quantized INT8) on mobile devices.

- Depthwise Separable Convolution

Inherited from V1, it splits standard convolution into Depthwise (channel-wise processing) and Pointwise (pixel-wise fusion). This reduces computation by about 8 to 9 times, while only slightly decreasing accuracy.

1.2.3 Network Architecture and Performance

MobileNetV2 consists of an initial fully convolutional layer and 19 residual bottleneck layers, and its features are shown in Table 1-1 1

Features	Description
Input dimensions	224 x 224 x 3
Output size	1000 (Number of ImageNet categories)
Number of parameters	Approximately 3.4 M (less than V1)
ImageNet accuracy	Top-1 has a success rate of approximately 72.0% (better than V1's 70.6%).
Computational complexity (MACs)	Approximately 300 M

Table 1-1 1Features

1.3 Execution Result

1.3.1 Generating PTE File

This stage involves executing the model compilation script on a PC to convert the MobileNetV2 model into a PTE file executable by the ExecuTorch runtime. The environment used is Ubuntu 24.04, with ExecuTorch and its dependent Python packages installed on Ubuntu.

When executing the MobileNetV2 compilation script, the pre-trained MobileNetV2 model will be automatically downloaded, so please maintain a network connection. After execution, a file named *"mv2_arm_delegate_ethos-u55-256.pte"* will be generated, as shown in Figure Figure 1-2 2 Using the Netron visualization tool to view this file will show the converted model graph, as shown in Figure Figure 1-3 3

The transformed model graph will have one EthosUBackend operator and two operators that need to be processed by the CPU (quantize_per_tensor and dequantize_per_tensor). The CPU-processing operators need to be specifically registered in the kernel table in the code.

```
chchen59@chchen59: /media/sf_ubuntu_24.04
Input   Off-chip Flash bandwidth                0.00 MB/batch
Weight  Off-chip Flash bandwidth                3.19 MB/batch
Output  Off-chip Flash bandwidth                0.00 MB/batch
Total   Off-chip Flash bandwidth                3.20 MB/batch
Total   Off-chip Flash bandwidth  per input     3.20 MB/inference (batch size 1)

Neural network macs                        300838272 MACs/batch

Info: The numbers below are internal compiler estimates.
For performance numbers the compiled network should be run on an FVP Model or FPGA.

Network Tops/s                                0.04 Tops/s

NPU cycles                                  6852245 cycles/batch
SRAM Access cycles                          3886216 cycles/batch
DRAM Access cycles                                0 cycles/batch
On-chip Flash Access cycles                       0 cycles/batch
Off-chip Flash Access cycles                3518227 cycles/batch
Total cycles                                7269834 cycles/batch

Batch Inference time               14.54 ms,   68.78 inferences/s (batch size 1)

class GraphModule(torch.nn.Module):
    def forward(self, x: "f32[1, 3, 224, 224]"):
        # No stacktrace found for following nodes
        alloc: "i8[1, 3, 224, 224]" = executorch_exir_memory_alloc(((1, 3, 224, 224), torch.int8))
        cortex_m_quantize_per_tensor_default: "i8[1, 3, 224, 224]" = torch.ops.cortex_m.quantize_per_tensor.out(x, 0.018455125391483307, -15, -128, 127, torch.int8, out = alloc);  x =
alloc = None
        lowered_module_0 = self.lowered_module_0
        executorch_call_delegate = torch.ops.higher_order.executorch_call_delegate(lowered_module_0, cortex_m_quantize_per_tensor_default);  lowered_module_0 = cortex_m_quantize_per_t
nsor_default = None
        getitem: "i8[1, 1000]" = executorch_call_delegate[0];  executorch_call_delegate = None
        alloc_1: "f32[1, 1000]" = executorch_exir_memory_alloc(((1, 1000), torch.float32))
        cortex_m_dequantize_per_tensor_default: "f32[1, 1000]" = torch.ops.cortex_m.dequantize_per_tensor.out(getitem, 0.03407548740506172, -76, -128, 127, torch.int8, out = alloc_1);
 getitem = alloc_1 = None
        return (cortex_m_dequantize_per_tensor_default,)

chchen59@chchen59:~/mv2_pte$
chchen59@chchen59:~/mv2_pte$
```

Figure 1-2 2Compilation Results

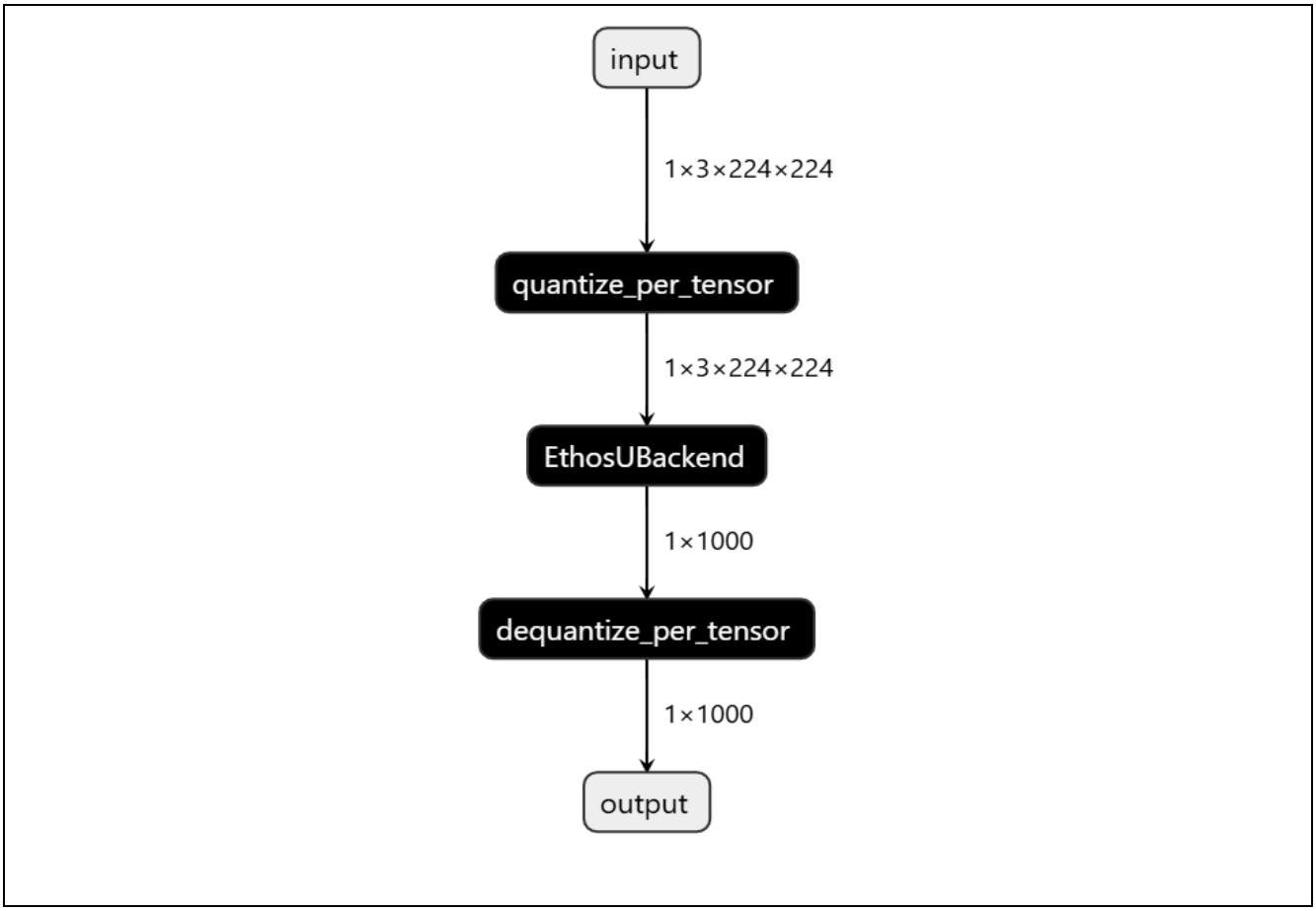

Figure 1-3 3Diagram

1.3.2 Execution

Since the MobileNetV2 PTE file exceeds the built-in Flash capacity, the file needs to be placed on an SD card and renamed *"mv2.pte"*. The runtime program will then load it from the SD card into HyperRAM for execution.

After the example program is compiled and programmed, the execution result is shown in Figure Figure 1-4 4

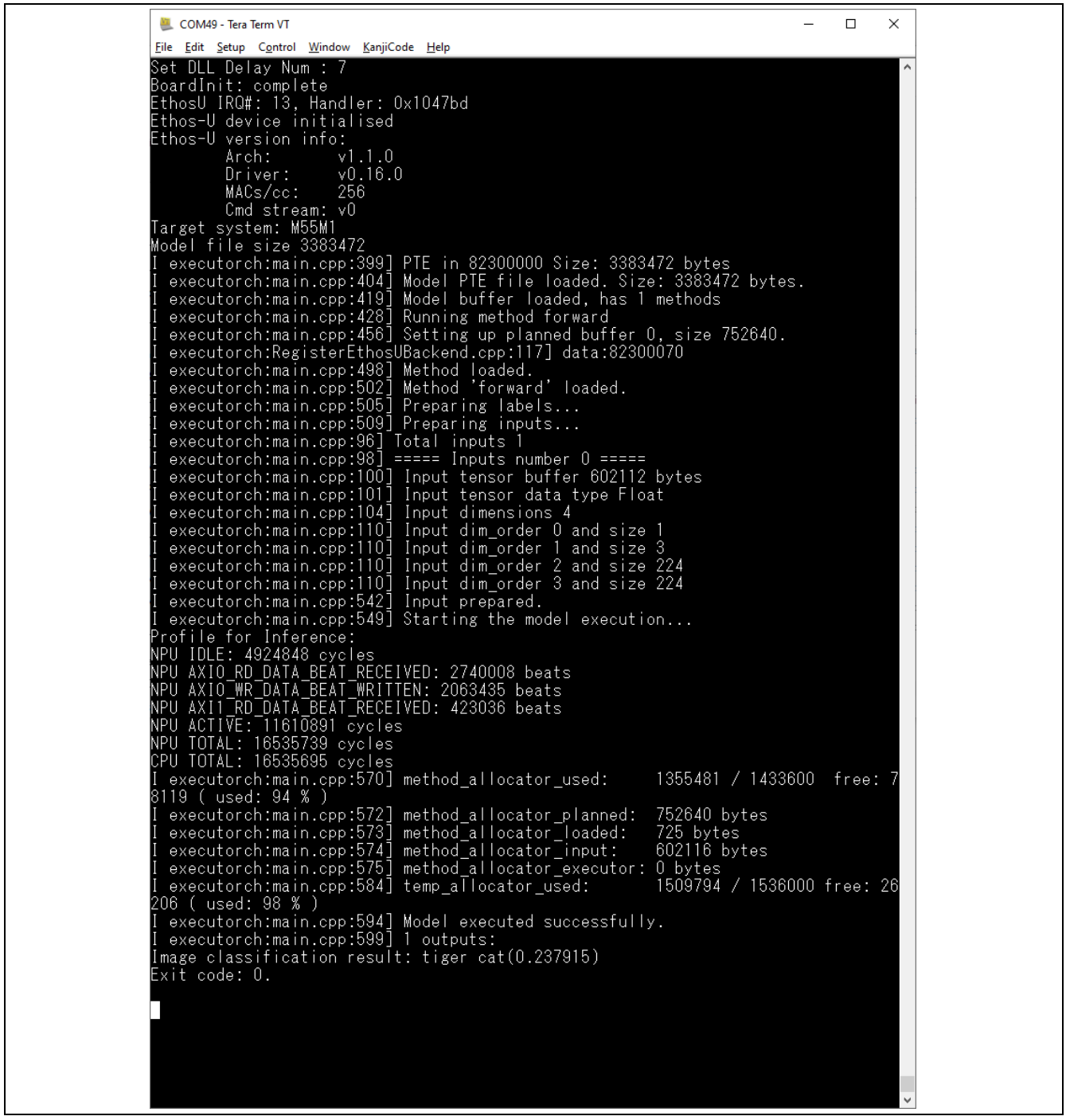

Figure 1-4 4Program Execution Results

2. Code Description

2.1 MobileNetV2 Compilation Script (mv2_pte.py)

The MobileNetV2 compilation script code is located in *Script/mv2_pte.py*. The mv2_pte.py script is mainly used to export and quantize the PyTorch MobileNetV2 model, optimize and convert it for Arm Ethos-U55 and Cortex-M backends, ultimately generating an ExecuTorch format model that can run on embedded devices. The complete instructions are as follows:

1. Main functions:

- Download and load the MobileNetV2 pre-trained model.
- Prepare sample input (random tensors or real images can be used).
- Export the model as a PyTorch Exported Program.
- Perform post-training quantization (PTQ) and export the quantize model again.
- The Arm Ethos-U55 has been split, compiled, and optimized.
- Perform Cortex-M backend-related conversions.
- Convert to ExecuTorch format and save as a *".pte"* file.

2. Main steps explained:

1) Loading the model and preparing input:

```
# Get example input
def get_example_inputs(use_real_input):
    tensor_size = (1, 3, 224, 224)
    input_batch = (torch.randn(tensor_size),)
    if use_real_input == True:
        print("Loaded real input image dog.jpg")
        import urllib

        url, filename = (
            "https://github.com/pytorch/hub/raw/master/images/dog.jpg",
            "dog.jpg",
        )
        try:
            urllib.URLopener().retrieve(url, filename)
        except:
            urllib.request.urlretrieve(url, filename)
        from PIL import Image
        from torchvision import transforms

        input_image = Image.open(filename)
        preprocess = transforms.Compose(
            [
                transforms.Resize(256),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
```

```
                transforms.Normalize(
                    mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
                ),
            ]
        )
        input_tensor = preprocess(input_image)
        input_batch = input_tensor.unsqueeze(0)
        input_batch = (input_batch,)
    return input_batch
```

- Use torchvision to download MobileNetV2 pre-trained weights.
- get_example_inputs function can optionally load a random tensor or download a real image (dog.jpg) and preprocess it.

2) Export Model:

```
# Get mobilenetv2 form torchvision
original_model = mobilenet_v2(weights=MobileNet_V2_Weights.DEFAULT)
print(type(original_model))

example_inputs = get_example_inputs(True)

model = original_model.eval()

exported_program = torch.export.export(model, example_inputs, strict=False)
graph_module = exported_program.module(check_guards=False)
#_ = graph_module.print_readable()

# We only support Python3.10 and above, so use a later pickle protocol
torch.export.save(
    exported_program,
    f"mv2_exported_program.pt2",
    pickle_protocol=5,
)
```

- Set the model to eval mode.
- Use torch.export.export to export the model and example inputs as an Exported Program.
- Use torch.export.save to store the exported model, specifying pickle protocol 5 (requiring Python 3.10+).

3) Quantification process

```
from executorch.backends.arm.ethosu import EthosUCompileSpec
from executorch.backends.arm.util._factory import create_partitioner, create_quantizer

from executorch.backends.arm.quantizer import (
    EthosUQuantizer,
    get_symmetric_quantization_config,
    get_symmetric_a16w8_quantization_config,
)
```

```
from torchao.quantization.pt2e.quantize_pt2e import convert_pt2e, prepare_pt2e

# Create a compilation spec describing the target for configuring the quantizer
# Some args are used by the Arm Vela graph compiler later in the example. Refer to Arm
Vela documentation for an
# explanation of its flags: https://gitlab.arm.com/artificial-intelligence/ethos-u/ethos-
u-vela/-/blob/main/OPTIONS.md
compile_spec = EthosUCompileSpec(
            target="ethos-u55-256",
            system_config="Ethos_U55_High_End_Embedded",
            memory_mode="Shared_Sram",
            extra_flags=["--verbose-operators", "--verbose-cycle-estimate"],
            config_ini="Arm/vela.ini"
        )

# Create and configure quantizer to use a symmetric quantization config globally on all
nodes
print(compile_spec)
quantizer = create_quantizer(compile_spec)

operator_config = get_symmetric_quantization_config(is_per_channel=True)
quantizer.set_global(operator_config)

# Post training quantization
quantized_graph_module = prepare_pt2e(graph_module, quantizer)
quantized_graph_module(*example_inputs) # Calibrate the graph module with the example
input
quantized_graph_module = convert_pt2e(quantized_graph_module)
#print(quantized_graph_module)

# Create a new exported program using the quantized_graph_module
quantized_exported_program = torch.export.export(quantized_graph_module, example_inputs)

torch.export.save(
    quantized_exported_program,
    f"mv2_quantize_exported_program.pt2",
    pickle_protocol=5,
)
```

- Import the relevant Ethos-U quantizer and config.
- Create an EthosUCompileSpec to describe the target hardware and compilation options.
- Create a quantizer and set the global symmetric quantization parameters.
- Use prepare_pt2e for pre-quantization preparation and calibrate using example input.
- Use convert_pt2e to complete the quantization and export the quantize Exported Program again.

4) Ethos-U Backend Segmentation and Optimization:

```
from executorch.backends.arm.ethosu import EthosUPartitioner
from executorch.exir import (
```

```
    EdgeCompileConfig,
    ExecutorchBackendConfig,
    to_edge_transform_and_lower,
)
from executorch.extension.export_util.utils import save_pte_program

# Create partitioner from compile spec
partitioner = EthosUPartitioner(compile_spec)

# Lower the exported program to the Ethos-U backend
edge_program_manager = to_edge_transform_and_lower(
            quantized_exported_program,
            partitioner=[partitioner],
            compile_config=EdgeCompileConfig(
                _check_ir_validity=False,
            ),
        )
```

- Create an EthosUPartitioner and perform partitioning based on the compile spec.
- Use to_edge_transform_and_lower for segmentation, optimization, and IR reduction.

5) Cortex-M back-end conversion:

```
# Transform so we can use ops from the Cortex M backend
edge_program_manager = transform_for_cortex_m_backend ( edge_program_manager )
```

- Executing transform_for_cortex_m_backend converts some operations into operations supported by Cortex-M.

6) Convert to ExecuTorch format and save:

```
# Convert edge program to executorch
executorch_program_manager = edge_program_manager.to_executorch(
            config=ExecutorchBackendConfig(extract_delegate_segments=False)
        )

_ = executorch_program_manager.exported_program().graph_module.print_readable()

# Save pte file
save_pte_program(executorch_program_manager, "mv2_arm_delegate_ethos-u55-256.pte")
```

- Use to_executorch to convert to ExecuTorch format.
- Use save_pte_program to save as a *.pte* file for deployment on embedded devices.

2.2 Runtime Main Program (main.cpp)

2.2.1 Main steps explained:

1. Define section:

```
#define MODEL_AT_HYPERRAM_ADDR (0x82300000)
```

```
#define __PROFILE__

#define MV2_INPUT_TENSOR_INDEX      0
#define MV2_OUTPUT_TENSOR_INDEX     0

//select image pattern: 0: cat, 1: dog, 2: tiger
#define IMAGE_PATTERN_INDEX         0
```

- MODEL_AT_HYPERRAM_ADDR: The address where the model is stored in HyperRAM.
- __PROFILE__: Enables performance analysis.
- MV2_*_TENSOR_INDEX: Input/output tensor indices of the MobileNetV2 model.
- IMAGE_PATTERN_INDEX: Select the test image (0: cat, 1: dog, 2: tiger).

2. Memory pool configuration:

```
// To speed up inference time. Using temp_allocation_pool for EthosU's scratch buffer
__attribute__((section(".bss.sram.data"), aligned(32)))
uint8_t s_au8TempAllocationPool[1500 * 1024U];

__attribute__((section(".bss.sram.data"), aligned(32)))
uint8_t s_au8MethodAllocationPool[1400 * 1024U];
```

- Two large memory pools are pre-allocated and placed in SRAM/HyperRAM.
- s_au8TempAllocationPool: Temporary buffer (1.5 MB) for Ethos-U NPU.
- s_au8MethodAllocationPool: Memory pool (1.4 MB) used for method execution.

3. Log output function:

```
void et_pal_emit_log_ message (
    __ET_UNUSED et_timestamp_t timestamp,
    et_pal_log_level_t level,
    const char* filename,
    __ET_UNUSED const char* function,
    size_t line,
    const char* message,
    __ET_UNUSED size_t length) {
    fprintf(stderr, "%c executorch:%s:%zu] %s\n", level, filename, line, message);
}
```

- ExecuTorch platform-level log output implementation.
- Direct ExecuTorch's internal logs to standard error output (stderr).
- The log level, file name, line number, and message content are displayed in a formatted manner.

4. Auxiliary function group:

1) Model input tensor information display:

```
// Print model input tensor information
static void PrintModelInputTensorInfo(Method& objMethod)
{
    MethodMeta objMethodMeta = objMethod.method_meta();
    size_t nNumInputs = objMethodMeta.num_inputs();
    ET_LOG(Info, "Total inputs %d ", nNumInputs);
    for (size_t i = 0; i < nNumInputs; i++) {
        ET_LOG(Info, "===== Inputs number %d =====", i);
        Result<TensorInfo> objTensorMeta = objMethodMeta.input_tensor_meta(i);
        ET_LOG(Info, "Input tensor buffer %d bytes", objTensorMeta->nbytes());
        ET_LOG(Info, "Input tensor data type %s ",
torch::executor::toString(objTensorMeta->scalar_type()));

        size_t nNumDims = objTensorMeta.get().sizes().size();
        ET_LOG(Info, "Input dimensions %d ", nNumDims);
        Tensor::DimOrderType *pDimOrder = const_cast<TensorImpl::DimOrderType
*>(objTensorMeta->dim_order().data());
        Tensor::SizesType *pDimSize = const_cast<TensorImpl::SizesType*>(objTensorMeta-
>sizes().data());

        for(int32_t j = 0; j < nNumDims; j ++)
        {
            ET_LOG(Info, "Input dim_order %d and size %d", pDimOrder[j], pDimSize[j]);
        }
    }
}
```

- Analyze and display detailed information about all input tensors of the model. This includes the number of inputs, the memory size of each tensor, the data type, the order of dimensions, and their size.

2) Model loaded into HyperRAM:

```
// Load model from SD card to HyperRAM
static int32_t PrepareModelToHyperRAM(void)
{
#define MODEL_FILE "0:\\mv2.pte"
#define EACH_READ_SIZE 512

    TCHAR achSDPath[] = { '0', ':', 0 };    /* SD drive started from 0 */
    f_chdrive(achSDPath);          /* set default path */

    int32_t i32FileSize;
    int32_t i32FileReadIndex = 0;
    int32_t i32Read;

    if(!ModelFileReader_Initialize(MODEL_FILE))
    {
        printf("Unable open model %s\n", MODEL_FILE);
        return -1;
    }
```

```
    i32FileSize = ModelFileReader_FileSize();
    printf("Model file size %i \n", i32FileSize);

    while(i32FileReadIndex < i32FileSize)
    {
        i32Read = ModelFileReader_ReadData((BYTE *)(MODEL_AT_HYPERRAM_ADDR +
i32FileReadIndex), EACH_READ_SIZE);
        if(i32Read < 0)
            break;
        i32FileReadIndex += i32Read;
    }

    if(i32FileReadIndex < i32FileSize)
    {
        printf("Read Model file size is not enough\n");
        return -2;
    }

#if 0
    /* verify */
    i32FileReadIndex = 0;
    ModelFileReader_Rewind();
    BYTE au8TempBuf[EACH_READ_SIZE];

    while(i32FileReadIndex < i32FileSize)
    {
        i32Read = ModelFileReader_ReadData((BYTE *)au8TempBuf, EACH_READ_SIZE);
        if(i32Read < 0)
            break;

        if(std::memcmp(au8TempBuf, (void *)(MODEL_AT_HYPERRAM_ADDR + i32FileReadIndex),
i32Read)!= 0)
        {
            printf("verify the model file content is incorrect at %i \n",
i32FileReadIndex);
            return -3;
        }
        i32FileReadIndex += i32Read;
    }

#endif
    ModelFileReader_Finish();

    return i32FileSize;
}
```

- Read the "mv2.pte" model file from the SD card.
- Load data into the specified address in HyperRAM in increments of 512 bytes.
- Includes a file integrity check mechanism (verification code annotated with #if 0).
- Returns the actual file size read, or a negative value if it fails.

3) Custom memory allocator:

```
// Setup our own allocator that can show some extra stuff like used and free
// memory info
class ArmMemoryAllocator : public executorch::runtime::MemoryAllocator {
public:
    ArmMemoryAllocator(uint32_t u32Size, uint8_t* pu8BaseAddress)
        : MemoryAllocator(u32Size, pu8BaseAddress), used_(0) {}

    void* allocate(size_t nSize, size_t nAlignment = kDefaultAlignment) override {
        void* pvRet = executorch::runtime::MemoryAllocator::allocate(nSize, nAlignment);
        if (pvRet != nullptr) {
            // Align with the same code as in MemoryAllocator::allocate() to keep
            // used_ "in sync" As alignment is expected to be power of 2 (checked by
            // MemoryAllocator::allocate()) we can check it the lower bits
            // (same as alignment - 1) is zero or not.
            if ((nSize & (nAlignment - 1)) == 0) {
                // Already aligned.
                used_ += nSize;
            } else {
                used_ = (used_ | (nAlignment - 1)) + 1 + nSize;
            }
        }
        return pvRet;
    }

    // Returns the used size of the allocator's memory buffer.
    size_t used_size() const {
        return used_;
    }

    // Returns the free size of the allocator's memory buffer.
    size_t free_size() const {
        return executorch::runtime::MemoryAllocator::size() - used_;
    }

private:
    size_t used_;
};
```

- The base memory allocator that inherits from ExecuTorch.
- Added memory usage tracking feature.
- Provides the used_size () and free_size () methods to monitor memory usage.
- Calculate the actual usage of memory alignment.

4) Image normalization processing:

```
// ImageNet mean/std
static const float s_afImagenetMean[3] = {0.485f, 0.456f, 0.406f};
static const float s_afImagenetStd[3]  = {0.229f, 0.224f, 0.225f};

// Normalize RGB888 image data to input tensor
static void NormalizeImageToInputTensor(
```

```
    Tensor *pInputTensor,
    uint8_t *pu8ImageData
)
{
    Tensor::SizesType *pnDimSize = const_cast<TensorImpl::SizesType*>(pInputTensor-
>sizes().data());

    int32_t C = pnDimSize[1];          //channel
    int32_t H = pnDimSize[2];          //height
    int32_t W = pnDimSize[3];          //width

    auto *pfTensorData = pInputTensor->mutable_data_ptr<float>();
    auto *pi8TensorData = pInputTensor->mutable_data_ptr<int8_t>();
    auto *pu8TensorData = pInputTensor->mutable_data_ptr<uint8_t>();

    for(int32_t c = 0; c < C; c ++)
    {
        for(int32_t h = 0; h < H; h++ )
        {
            for(int32_t w = 0; w < W; w++ )
            {
                uint8_t u8Color = pu8ImageData[(((h * W) + w) * 3) + c];

                if(pInputTensor->scalar_type() ==  ScalarType::Float)
                {
                    // Normalize to [0,1], then standardize
                    float fColor = u8Color / 255.0f;
                    // Standardize
                    *pfTensorData = (fColor - s_afImagenetMean[c]) / s_afImagenetStd[c];
                    pfTensorData ++;
                }
                else if(pInputTensor->scalar_type() ==  ScalarType::Char)
                {
                    int16_t ui16color = u8Color;
                    *pi8TensorData = (int8_t)(ui16color - 128);
                    pi8TensorData++;
                }
                else if(pInputTensor->scalar_type() ==  ScalarType::Byte)
                {
                    *pu8TensorData = u8Color;
                    pu8TensorData++;
                }
            }
        }
    }
}
```

- Convert RGB888 format image data into model input tensors.
- It supports three data types: Float (normalized), Char (minus 128), and Byte (direct copy).
- The Float type is standardized using the mean and standard deviation of ImageNet.
- The data is arranged in a channel-first format (C, H, W).

5) Prepare input tensors:

```
// Prepare input tensors from image data
static Result<executorch::extension::BufferCleanup> PrepareInputTensorsFromImage(
    Method& objMethod,
    executorch::runtime::MemoryAllocator& objAllocator,
    std::vector<std::pair<char*, size_t>>& vecInputBuffers)
{
    // Get method meta to setup input tensors
    MethodMeta method_meta = objMethod.method_meta();
    size_t nNumInputs = method_meta.num_inputs();
    size_t nNumAllocated = 0;
    ET_CHECK_OR_RETURN_ERROR(
        nNumInputs == 1,
        InvalidArgument,
        "MobileNetV2 input should be 1");

    void** vppInputs = static_cast<void**>(objAllocator.allocate(nNumInputs *
sizeof(void*)));

    Result<TensorInfo> objTensorMeta = method_meta.input_tensor_meta(0);
    ET_CHECK_OK_OR_RETURN_ERROR(objTensorMeta.error());

    // Input is a tensor. Allocate a buffer for it.
    void* vpData = objAllocator.allocate(objTensorMeta->nbytes());
    ET_CHECK_OR_RETURN_ERROR(
        vpData != nullptr,
        MemoryAllocationFailed,
        "Could not allocate memory for input buffers.");
    vppInputs[nNumAllocated++] = vpData;

    // Create the input tensor
    TensorImpl objTensorImpl = TensorImpl(
                                    objTensorMeta.get().scalar_type(),
                                    objTensorMeta.get().sizes().size(),

const_cast<TensorImpl::SizesType*>(objTensorMeta.get().sizes().data()),
                                    vpData,
                                    const_cast<TensorImpl::DimOrderType*>(
                                        objTensorMeta.get().dim_order().data()));
    Tensor objTensor(&objTensorImpl);

    Error err = Error::Ok;
    if (vecInputBuffers.size() > 0) {
        auto [image_data_ptr, image_data_size] = vecInputBuffers.at(0);
        //Normalize RGB888 image data to float
        NormalizeImageToInputTensor(&objTensor, (uint8_t *)image_data_ptr);
    }
    else
    {
        // If input_buffers.size <= 0, we don't have any input, fill t with 1's.
        for (size_t j = 0; j < objTensor.numel(); j++) {
            switch (objTensor.scalar_type()) {
            case ScalarType::Float:
                objTensor.mutable_data_ptr<float>()[j] = 1.;
                break;
```

```
            case ScalarType::Char:
                objTensor.mutable_data_ptr<int8_t>()[j] = 1;
                break;
            case ScalarType::Byte:
                objTensor.mutable_data_ptr<uint8_t>()[j] = 1;
                break;
            default:
                ET_LOG(
                    Error, "Failed to handle input data type");
            }
            break;
        }
    }

    // Set the input tensor to the method
    err = objMethod.set_input(objTensor, 0);

    if (err != Error::Ok) {
        ET_LOG(
            Error, "Failed to prepare input %zu: 0x%" PRIx32, 0, (uint32_t)err);
        // The BufferCleanup will free the inputs when it goes out of scope.
        executorch::extension::BufferCleanup cleanup({vppInputs, nNumAllocated});
        return err;
    }

    return  executorch::extension::BufferCleanup({vppInputs, nNumAllocated});
}
```

- Prepare input tensors for MobileNetV2 model.
- Get the input tensor specifications from the method's meta data.
- Allocate appropriate memory and create tensor objects.
- Call NormalizeImageToInputTensor () to fill in the image data.

6) Softmax implementation:

```
// Softmax for float vector
static void SoftmaxF32(std::vector<float>& vecData)
{
    /* Fix for numerical stability and apply exp. */
    auto nStart = vecData.begin();
    auto nEnd = vecData.end();

    float maxValue = *std::max_element(nStart, nEnd);
    for (auto it = nStart; it != nEnd; ++it) {
        *it = std::exp((*it) - maxValue);
    }

    float fSumExp = std::accumulate(nStart, nEnd, 0.0f);

    for (auto it = nStart; it != nEnd; ++it) {
        *it = (*it)/fSumExp;
    }
```

```
}
```

- The Softmax operation is on a floating-point vector.
- First, subtract the maximum value to prevent numerical overflow (numerical stability).
- Calculate the sum of the exponents and normalize it to obtain the probability distribution.
- Ensure that the sum of all output values is 1.

5. Main program main() description:

The flowchart of the main function is shown in Figure 2- 1.

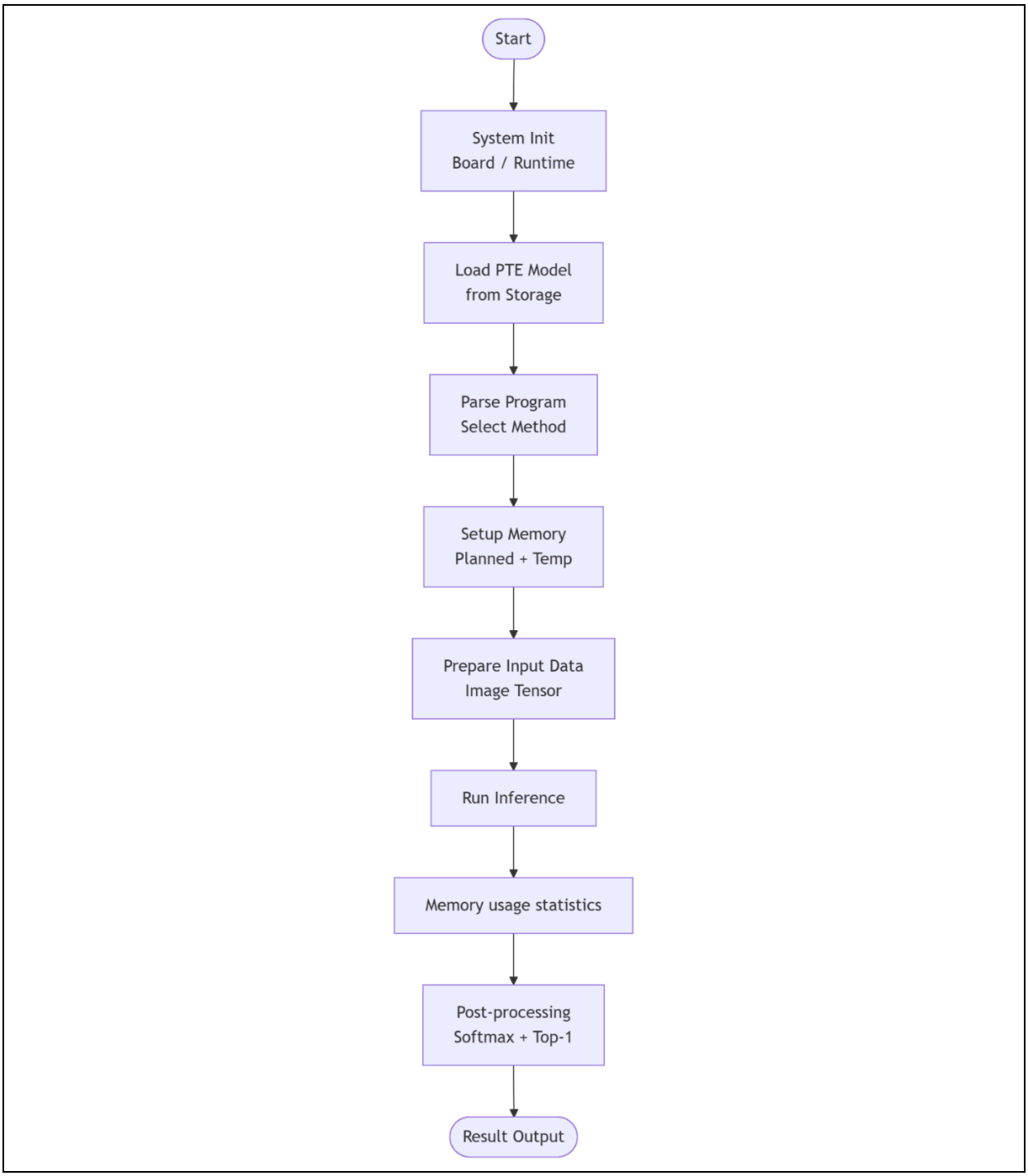

Figure 2- 1 Main Function

1) Initialization phase:

```
    /* Initialize the hardware resource (UART, NPU, Hyper RAM ) */
    BoardInit ();

#if defined(__PROFILE__)
    arm::app::Profiler objProfiler;
#endif
```

```
torch::executor::register_all_kernels();
torch::executor::runtime_init();
```

- BoardInit(): Initializes hardware resources such as UART, NPU, and HyperRAM.
- Register all the computational kernels of ExecuTorch (convolution, fully connected layers, etc.).
- Initialize the ExecuTorch execution environment.

2) Model loading and parsing:

```
// Load model form SD card to HyperRAM
uint8_t *pu8ModelPTE = (uint8_t *)MODEL_AT_HYPERRAM_ADDR;
int i32PTESize;

i32PTESize = PrepareModelToHyperRAM();

if(i32PTESize <= 0)
{
    ET_LOG(
        Info,
        "Model loading failed @ 0x%p" PRIx32,
        pu8ModelPTE);
    return -1;
}

ET_LOG(
    Info, "PTE in %p Size: %d bytes", pu8ModelPTE, i32PTESize);

// Create a data loader for the model file in memory
auto objLoader =
    torch::executor::util::BufferDataLoader(pu8ModelPTE, i32PTESize);
ET_LOG(Info, "Model PTE file loaded. Size: %u bytes.", i32PTESize);

// Parse the program file. This is immutable, and can also be reused
// between multiple execution invocations across multiple threads.
Result<torch::executor::Program> pProgram =
    torch::executor::Program::load(&objLoader);
if (!pProgram.ok()) {
    ET_LOG(
        Info,
        "Program loading failed @ 0x%p: 0x%" PRIx32,
        pu8ModelPTE,
        pProgram.error());
    return -2;
}
```

- Call PrepareModelToHyperRAM() to load the PTE model file from the SD card.
- Create a BufferDataLoader to wrap the model data in memory.
- Analyze the model file to create a “pProgram” object, containing all the methods and meta information of the model.

3) Method selection and memory planning:

```
    // For this example we just run the first method found in the program.
    const char* pchMethodName = nullptr;
    {
        const auto pchMethodNameResult = pProgram->get_method_name(0);
        ET_CHECK_MSG(pchMethodNameResult.ok(), "Program has no methods");
        pchMethodName = *pchMethodNameResult;
    }
    ET_LOG(Info, "Running method %s", pchMethodName);

    // Get method meta to setup memory planned buffers
    Result<torch::executor::MethodMeta> pMethodMeta =
        pProgram->method_meta(pchMethodName);
    if (!pMethodMeta.ok()) {
        ET_LOG(
            Info,
            "Failed to get method_meta for %s: 0x%x",
            pchMethodName,
            (unsigned int)pMethodMeta.error());
    }

    // Setup memory allocators
    ArmMemoryAllocator objMethodAllocator(
        sizeof(s_au8MethodAllocationPool), s_au8MethodAllocationPool);

    std::vector<uint8_t*> vecPlannedBuffers; // Owns the memory
    std::vector<torch::executor::Span<uint8_t>> vecPlannedSpans; // Passed to the
allocator

    size_t nNumMemoryPlannedBuffers = pMethodMeta->num_memory_planned_buffers();

    size_t nPlannedBufferMembase = objMethodAllocator.used_size();

    // Allocate memory planned buffers
    for (size_t id = 0; id < nNumMemoryPlannedBuffers; ++id) {
        size_t nBufferSize =
            static_cast<size_t>(pMethodMeta->memory_planned_buffer_size(id).get());
        ET_LOG(Info, "Setting up planned buffer %zu, size %zu.", id, nBufferSize);

        /* Move to it's own allocator when MemoryPlanner is in place. */
        uint8_t* pu8Buffer =
            reinterpret_cast<uint8_t*>(objMethodAllocator.allocate(nBufferSize));
        ET_CHECK_MSG(
            pu8Buffer != nullptr,
            "Could not allocate memory for memory planned buffer size %zu",
            nBufferSize);
        vecPlannedBuffers.push_back(pu8Buffer);
        vecPlannedSpans.push_back({vecPlannedBuffers.back(), nBufferSize});

    }
```

- Select the first method of this model to execute.
- Meta data on the acquisition method, including memory requirement information.

- Create a custom memory allocator.
- Allocate the required plan buffer for model execution based on meta data.

4) Memory manager setup:

```
// Create the hierarchical allocator for planned buffers
torch::executor::HierarchicalAllocator objPlannedMemory(
{vecPlannedSpans.data(), vecPlannedSpans.size()});

ArmMemoryAllocator objTempAllocator(
    sizeof(s_au8TempAllocationPool), s_au8TempAllocationPool);

// Create the memory manager
torch::executor::MemoryManager objMemoryManager(
    &objMethodAllocator, &objPlannedMemory, &objTempAllocator);
```

- HierarchicalAllocator: Manages pre-allocated plan buffers.
- objTempAllocator: Provides temporary memory (for Ethos-U NPU).
- objMemoryManager: Unifies the management of three types of memory allocators.

5) Loading method:

```
// Load the method from the program
Result<torch::executor::Method> pMethod =
    pProgram->load_method(pchMethodName, &objMemoryManager, pEventTracer);
if (!pMethod.ok()) {
    ET_LOG(
        Info,
        "Loading of method %s failed with status 0x%" PRIx32,
        pchMethodName,
        pMethod.error());
}
ET_LOG(Info, "Method loaded.");
```

- Load the specified method using the memory manager.
- Once the method is loaded, the inference can be executed.

6) Data preparation:

```
// label information
ET_LOG(Info, "Preparing labels...");
std::vector <std::string> vecLabels;
GetLabelsVector(vecLabels);

ET_LOG(Info, "Preparing inputs...");
size_t nInputMembase = objMethodAllocator.used_size();

// Print model input tensor information
PrintModelInputTensorInfo(*pMethod);

// Get input image data
```

```
    uint8_t u8ImgIdx = IMAGE_PATTERN_INDEX;
    const uint8_t *pu8ImgSrc = get_img_array(u8ImgIdx);

    if (nullptr == pu8ImgSrc)
    {
        ET_LOG(Error, "Failed to get image index %" PRIu32 " (max: %u)\n", u8ImgIdx,
               NUMBER_OF_FILES - 1);
    }

    // Prepare input tensors from image data
    std::vector<std::pair<char*, size_t>> vecInputBuffers = {
        {(char *)pu8ImgSrc, IMAGE_DATA_SIZE}
    };

    // Prepare the input tensors
    auto pPreparedInputs = ::PrepareInputTensorsFromImage(*pMethod, objMethodAllocator,
vecInputBuffers);

    if (!pPreparedInputs.ok()) {
        ET_LOG(
            Info,
            "Preparing inputs tensors for method %s failed with status 0x%" PRIx32,
            pchMethodName,
            pPreparedInputs.error());
    }
```

- Load the classification label vector (ImageNet 1000 classes).
- Displays detailed information about the model's input tensors.
- Obtain the pre-compiled test image based on IMAGE_PATTERN_INDEX.
- Call PrepareInputTensorsFromImage() to prepare the model input.

7) Model execution:

```
    // Execute the method
#if defined(__PROFILE__)
    objProfiler.StartProfiling("Inference");
#endif

    ET_LOG(Info, "Starting the model execution...");
    size_t nExecutorMembase = objMethodAllocator.used_size();
    Error eStatus = pMethod->execute();

#if defined(__PROFILE__)
    objProfiler.StopProfiling();
    objProfiler.PrintProfilingResult();
#endif
```

- Start performance analysis (if enabled).
- Record the amount of memory used before execution.

- Execution model inference.
- Stop the performance analysis and display the results.

8) Memory usage statistics:

```
// Calculate executor memory usage
size_t nExecutorMemsize = objMethodAllocator.used_size() - nExecutorMembase;

// Print memory usage information
if (objMethodAllocator.size() != 0) {
    size_t nMethodAllocatorUsed = objMethodAllocator.used_size();
    ET_LOG(
        Info,
        "method_allocator_used:     %zu / %zu  free: %zu ( used: %zu %% ) ",
        nMethodAllocatorUsed,
        objMethodAllocator.size(),
        objMethodAllocator.free_size(),
        100 * nMethodAllocatorUsed / objMethodAllocator.size());
    ET_LOG(
        Info, "method_allocator_planned:  %zu bytes", nPlannedBufferMemsize);
    ET_LOG(Info, "method_allocator_loaded:   %zu bytes", nMethodLoadedMemsize);
    ET_LOG(Info, "method_allocator_input:    %zu bytes", nInputMemsize);
    ET_LOG(Info, "method_allocator_executor: %zu bytes", nExecutorMemsize);
}
if (objTempAllocator.size() > 0) {
    ET_LOG(
        Info,
        "temp_allocator_used:       %zu / %zu free: %zu ( used: %zu %% ) ",
        objTempAllocator.used_size(),
        objTempAllocator.size(),
        objTempAllocator.free_size(),
        100 * objTempAllocator.used_size() / objTempAllocator.size());
}
```

- Detailed statistics on memory usage at each stage.
- Includes: total usage, plan buffer, method loading, input preparation, and execution phase.
- Calculate the percentage of usage.

9) Result processing:

```
// Get and process the outputs
std::vector<torch::executor::EValue> vsOutputs(pMethod->outputs_size());
ET_LOG(Info, "%zu outputs: ", vsOutputs.size());
eStatus = pMethod->get_outputs(vsOutputs.data(), vsOutputs.size());
ET_CHECK(eStatus == Error::Ok);

// Process the output tensor
Tensor objOutputTensor = vsOutputs[MV2_OUTPUT_TENSOR_INDEX].toTensor();
std::vector<float> vecOutputTensorData(objOutputTensor.numel());

for (int j = 0; j < objOutputTensor.numel(); j++) {
```

```
        if (objOutputTensor.scalar_type() == ScalarType::Float) {
            auto fProb = objOutputTensor.const_data_ptr<float>()[j];
            vecOutputTensorData[j] = fProb;
        } else if (objOutputTensor.scalar_type() == ScalarType::Byte) {
            auto u8Prob = objOutputTensor.const_data_ptr<uint8_t>()[j];
            vecOutputTensorData[j] = u8Prob;
        } else if (objOutputTensor.scalar_type() == ScalarType::Char) {
            auto i8Prob = objOutputTensor.const_data_ptr<int8_t>()[j];
            vecOutputTensorData[j] = i8Prob;
        }
    }
```

- Obtain the output from the completed method.
- Supports multiple output data types (Float, Byte, Char).
- They are all converted to float vectors for subsequent classification processing.

10) Classification results output:

```
    //Apply softmax to get probabilities
    SoftmaxF32(vecOutputTensorData);

    // Find the class with the highest probability
    int i32MaxCLS = 0;
    float fMaxProb = -1;

    for (int j = 0; j < objOutputTensor.numel(); j++) {
        if(vecOutputTensorData[j] > fMaxProb)
        {
            fMaxProb = vecOutputTensorData[j];
            i32MaxCLS = j;
        }
    }

    printf("Image classification result: %s(%f) \n", vecLabels[i32MaxCLS].c_str(),
fMaxProb);
```

- Softmax on the original output of the model to convert it into a probability distribution.
- Iterate through all categories to find the prediction with the highest probability.
- Output classification results: category name and corresponding probability.

2.3 Registering Kernel Operators (RegisterNativeKernels.cpp)

As mentioned in previous chapters, the compiled MobileNetV2 PTE model diagram contains two operators that require CPU computation. To reduce code ROM space, the CPU-operated operators need to register their corresponding operator call code in the kernel registry.

Using the MobileNetV2 PTE model diagram as an example, and examining it with Netron, there are two CPU operators: "cortex_m::quantize_per_tensor.out" and "cortex_m:: dequantize_per_tensor.out", both prefixed with "cortex_m". Therefore, you can find the corresponding operator call code in *ThirdParty\executorch_prebuilt\backends\cortex_m*

RegisterCodegenUnboxedKernelsEverything.cpp and copy it to the kernels_to_register table in *SampleCode\ExampleCode\ExecuTorch_ImageClassification\RegisterNativeKernels.cpp*.

To summarize, the corresponding operator calls to the original code location for each prefix are as follows:

- cortex_m:

  ThirdParty\executorch_prebuilt\backends\cortex_m\RegisterCodegenUnboxedKernelsEverything.cpp

- quantized_decomposed:

  ThirdParty\executorch_prebuilt\kernels\quantized\RegisterCodegenUnboxedKernelsEverything.cpp

- aten and dim_order_ops:

  ThirdParty\executorch_prebuilt\kernels\portable\RegisterCodegenUnboxedKernelsEverything.cpp

1. Program Description:

   1) Introduction of header file:

```
#include <executorch/runtime/core/evalue.h>
#include <executorch/runtime/core/exec_aten/exec_aten.h>
#include <executorch/runtime/core/exec_aten/util/tensor_util.h>
#include <executorch/runtime/core/span.h>
#include <executorch/runtime/kernel/operator_registry.h>
#include <executorch/runtime/platform/profiler.h>
```

   - evalue.h: Provides the EValue class for encapsulating different types of data.
   - exec_aten.h: An implementation of ExecuTorch's ATen (A Tensor library).
   - tensor_util.h: Tensor manipulation tools.
   - span.h: A container view like std::span.
   - operator_registry.h: Operator registration system.
   - profiler.h: Performance analysis tool.

   2) Type alias definition:

```
using KernelSpan = ::executorch::runtime::Span<
                   const ::executorch::ET_RUNTIME_NAMESPACE::Kernel>;
```

   - A type of alias is defined to represent a Kernel Span (like an array view).

   3) Core array table definition:

```
static Kernel kernels_to_register[] = {
```

- The kernels_to_register variable is a static array containing all the computing kernels to be registered.

4) The first kernel: quantization operation.

```
    Kernel(
    "cortex_m::quantize_per_tensor.out",
    [](torch::executor::KernelRuntimeContext & context, Span<EValue*> stack) {
        ET_KERNEL_CHECK_MSG(context, stack.size() == 8, InvalidProgram, /*void*/,
"Expected %" ET_PRIsize_t "args received %" ET_PRIsize_t, (size_t)8, stack.size());
        EValue& input = *stack[0];
        EValue& scale = *stack[1];
        EValue& zero_point = *stack[2];
        EValue& quant_min = *stack[3];
        EValue& quant_max = *stack[4];
        EValue& dtype = *stack[5];
        EValue& out = *stack[6];
        const torch::executor::Tensor & input_base = input.to<torch::executor::Tensor>();
        double scale_base = scale.to<double>();
        int64_t zero_point_base = zero_point.to<int64_t>();
        int64_t quant_min_base = quant_min.to<int64_t>();
        int64_t quant_max_base = quant_max.to<int64_t>();
        torch::executor::ScalarType dtype_base = dtype.to<torch::executor::ScalarType>();
        torch::executor::Tensor & out_base = out.to<torch::executor::Tensor>();

        internal::EventTracerProfileOpScope
event_tracer_op_scope(context.internal_event_tracer(),
"native_call_quantize_per_tensor.out");
        EXECUTORCH_SCOPE_PROF("native_call_quantize_per_tensor.out");
        cortex_m::native::quantize_per_tensor_out(context, input_base, scale_base,
zero_point_base, quant_min_base, quant_max_base, dtype_base, out_base);
        internal::event_tracer_log_evalue(context.internal_event_tracer(), *stack[6]);

    }
    ),
```

5) The second kernel: dequantization operation.

```
    Kernel(
        "cortex_m::dequantize_per_tensor.out",
    [](torch::executor::KernelRuntimeContext & context, Span<EValue*> stack) {
        ET_KERNEL_CHECK_MSG(context, stack.size() == 8, InvalidProgram, /*void*/,
"Expected %" ET_PRIsize_t "args received %" ET_PRIsize_t, (size_t)8, stack.size());
        EValue& input = *stack[0];
        EValue& scale = *stack[1];
        EValue& zero_point = *stack[2];
        EValue& quant_min = *stack[3];
        EValue& quant_max = *stack[4];
        EValue& dtype = *stack[5];
        EValue& out = *stack[6];
        const torch::executor::Tensor & input_base = input.to<torch::executor::Tensor>();
        double scale_base = scale.to<double>();
        int64_t zero_point_base = zero_point.to<int64_t>();
        int64_t quant_min_base = quant_min.to<int64_t>();
```

```
        int64_t quant_max_base = quant_max.to<int64_t>();
        torch::executor::ScalarType dtype_base = dtype.to<torch::executor::ScalarType>();
        torch::executor::Tensor & out_base = out.to<torch::executor::Tensor>();

        internal::EventTracerProfileOpScope
event_tracer_op_scope(context.internal_event_tracer(),
"native_call_dequantize_per_tensor.out");
        EXECUTORCH_SCOPE_PROF("native_call_dequantize_per_tensor.out");
        cortex_m::native::dequantize_per_tensor_out(context, input_base, scale_base,
zero_point_base, quant_min_base, quant_max_base, dtype_base, out_base);
        internal::event_tracer_log_evalue(context.internal_event_tracer(), *stack[6]);

    }
    ),
```

6) Kernel Span Setup:

```
static KernelSpan kernel_span(
    kernels_to_register,
    kernels_to_register + sizeof(kernels_to_register) / sizeof(Kernel));
```

- KernelSpan object is created to wrap the kernels_to_register array so that it can be used by the ExecuTorch registration system.

7) Registration function:

```
Error register_all_kernels() {
    Error success_with_kernel_reg = register_kernels(kernel_span);

    if(success_with_kernel_reg != Error::Ok) {
        ET_LOG(Error, "Failed register all kernels");
    }
    return success_with_kernel_reg;
}
```

- This function is responsible for registering all kernels to the ExecuTorch system.

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M55M1_Series_BSP_CMSIS_V3.01.003
- ExecuTorch version
  - ExecuTorch v1.0.1
- IDE version
  - Keil uVision 5.43

3.2 Hardware Requirements

- Circuit components
  - NuMaker-X-M55M1D
- Pin Connect
  - Connect the UART0 TX (PB.13) pin to the PC UART RX to show the execution results of this example code.

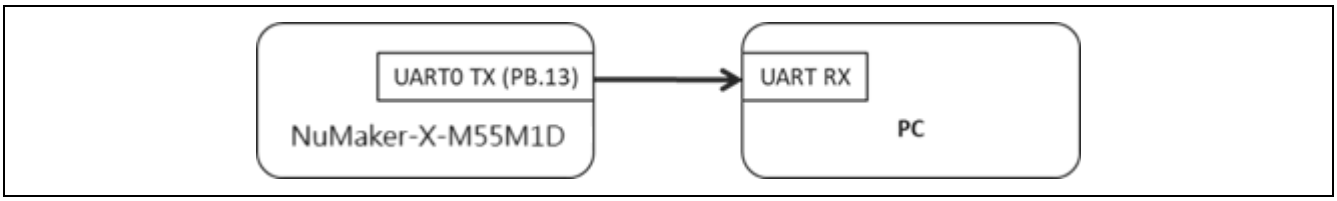

Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

EC_M55M1_ExecuTorch_ImgClass_V1.00	
Library	Sample code header and source files
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file
StdDriver	All peripheral driver header and source files
Storage	SDH disk IO source file
SampleCode	
ExampleCode	
ExecuTorch_ImageClassification	Source file of example code
Script	Script file
ThirdParty	Third party libraries
executorch	ExecuTorch library source files
FatFs	FAT file system library source files

Figure 4-1 Directory Structure

5. Example Code Execution

Navigate to the Model folder in the ExampleCode path according to the Directory Information section, copy the pre-compiled *"mv2.pte"* to the micro-SD card, and then insert the micro-SD card into the NuMaker-X-M55M1D board.

5.1 Keil MDK

1. Browse the sample code folder as described in the Directory Information section and double-click *ExecuTorch_ImageClassification.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

5.2 VS Code

1. Based on the Directory Information section, open the VSCode folder located in the ExampleCode directory within VS Code.
2. Click the CMSIS icon in the VS Code Activity Bar to open the CMSIS view.
3. Enter CMSIS mode.
   - Build
   - Load the code to memory and run.
   - Load the code to memory and enter debug mode.
4. Enter debug mode.

   Continue executing the code.

6. Revision History

Date	Revision	Description
2026.01.15	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.