Deep Software Stack Optimization for AI-Enabled Embedded Systems

Aug. 20, 2025

Prof. Seongsoo Hong, Namcheol Lee, Geonha Park, and Taehyun Kim
{sshong, nclee, ghpark, thkim}@redwood.snu.ac.kr
Dept. of Electrical and Computer Engineering,
Seoul National University

Special Thanks to

- Sapphire Stream Technology
  - For sponsoring this tutorial and providing RUBIK Pi 3 boards to attendees

https://www.sapphirestreamtech.com/

Tutorial Outline

Lecture 1: *From Inference Driver to Inference Runtime* (50 min)	
Lecturer	Seongsoo Hong
Topics	Step-by-Step Inference Driver Walkthrough
	Internals of LiteRT
Brief Break (10 min)	
Lecture 2: *Model Slicer* (50 min)	
Lecturer	Seongsoo Hong
Topic	Model Slicer: Slicing and Conversion Tool for LiteRT
Brief Break (10 min)	
Lecture 3: *Throughput Enhancement on Heterogeneous Accelerators* (1h 30 min)	
Lecturer	Namcheol Lee
Topic	Implementing a Pipelined Inference Driver for Heterogeneous Processors

Contents

On-Device AI

- Refers to executing AI models directly on edge devices
- On-device AI enables
  - Real-time, private, and offline inference
- Challenge
  - Operates under resource constraints

Optimization Techniques for On-Device AI

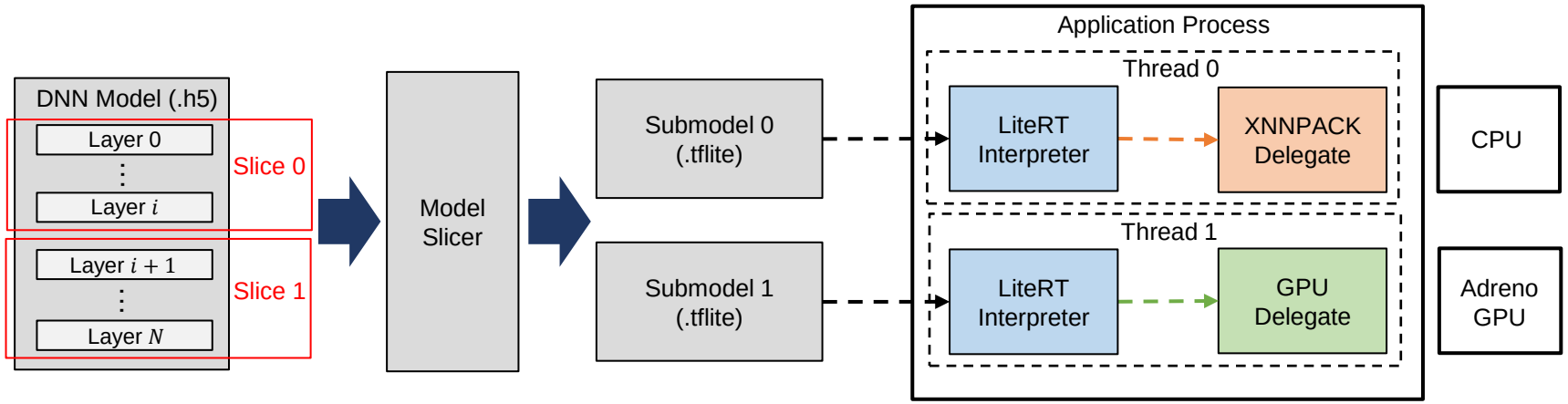
- Common techniques
  - Model quantization
  - Pruning
  - Knowledge distillation
  - Low-rank adaption
  - Weight streaming
  - Sparse inference (e.g., sparse GEMM)

Our Exercise Problem

- Lossless stream processing in on-device AI
  - Real-world relevance
    - Surveillance cameras
    - Automotive perception applications
  - A transient surge in input data stream may result in data loss
    - When input stream's data rate exceeds device inference throughput
  - Need to scale up inference throughput dynamically

DNN Pipelining (1)

- Scale up inference throughput via pipelining on multiple accelerators
  - Partition DNN model into submodels
  - Create worker threads that perform inference for submodels
  - Allocate the worker threads to available, heterogeneous accelerators

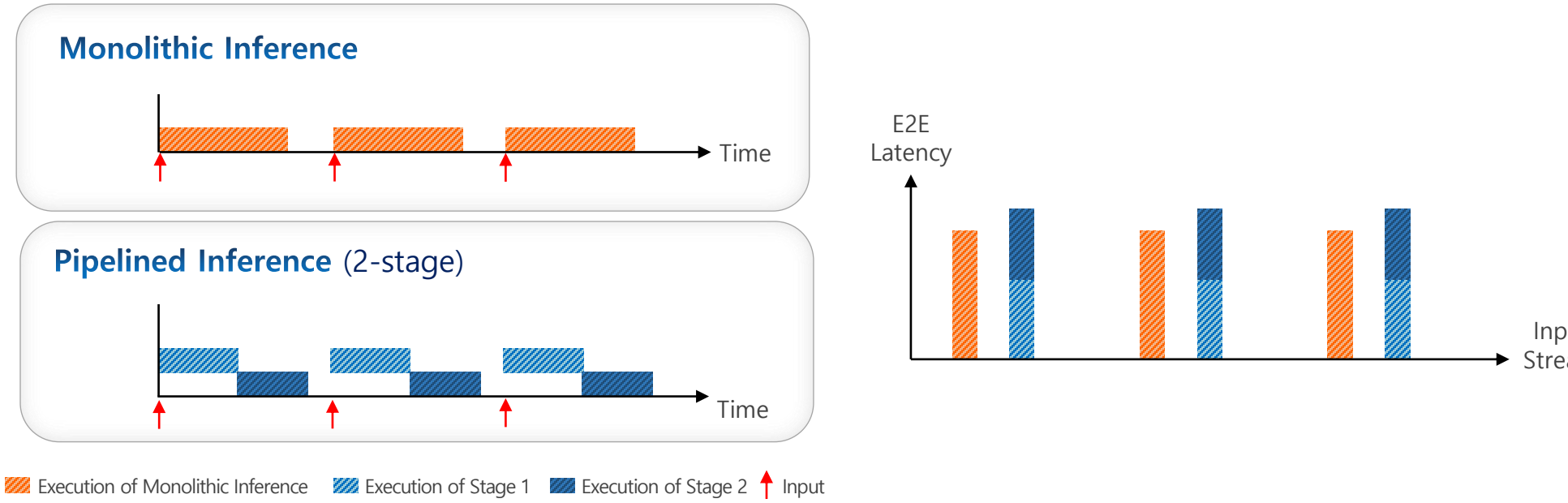


DNN Pipelining (2)

- Case analysis
  1. If input stream's data rate ≤ device's inference throughput:
     - Pipelining can be detrimental
       - May increase end-to-end (E2E) latency for DNN inference
         - Some stages may run on relatively slower processor (e.g., CPU)
         - Communication delays could be added
  2. If input stream's data rate > device's inference throughput:
     - Pipelining is beneficial
       - The legacy system may incur unbounded E2E latency due to back logs
       - Pipelining can increase throughput, enabling bounded E2E latency

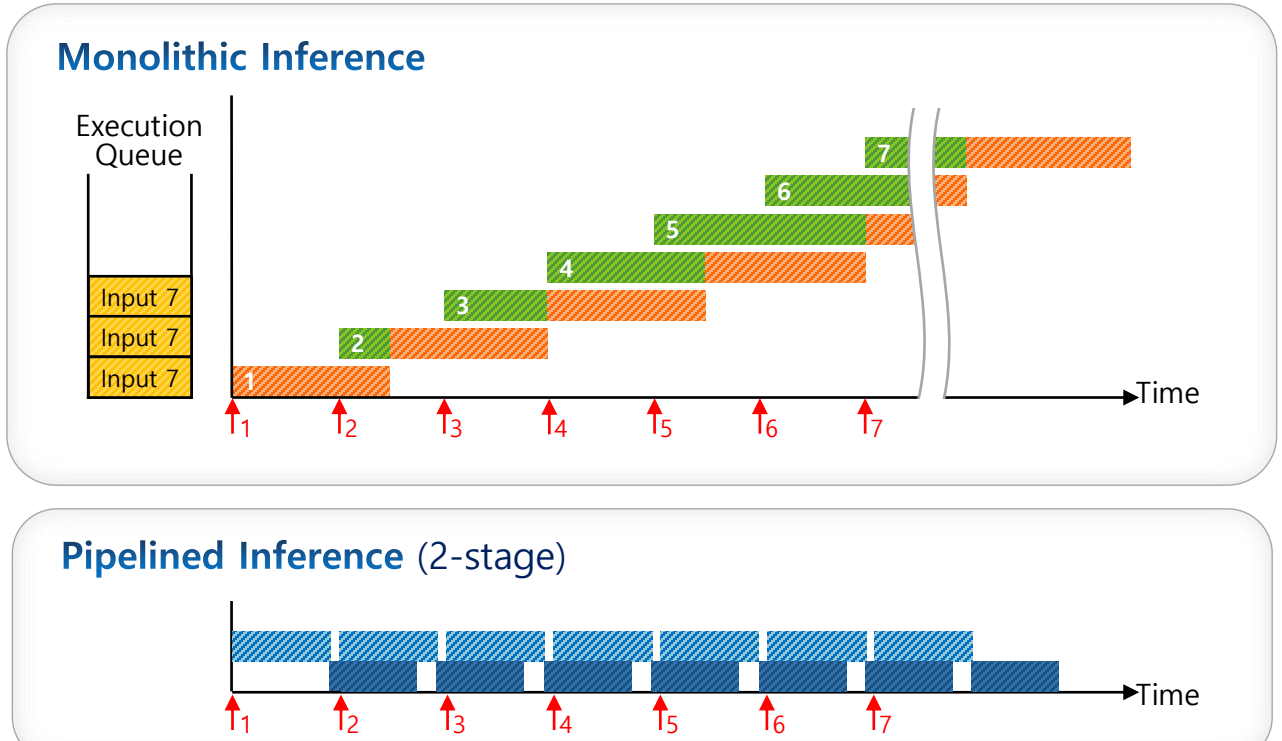
DNN Pipelining (3)

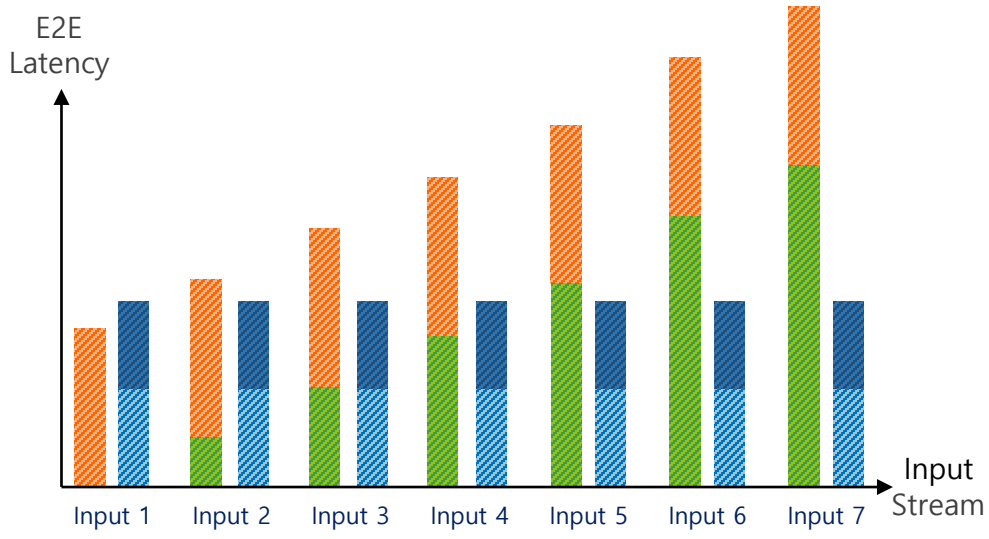
1. If *input stream's data rate ≤ device's inference throughput*



DNN Pipelining (4)

2. If *input stream's data rate > device's inference throughput*





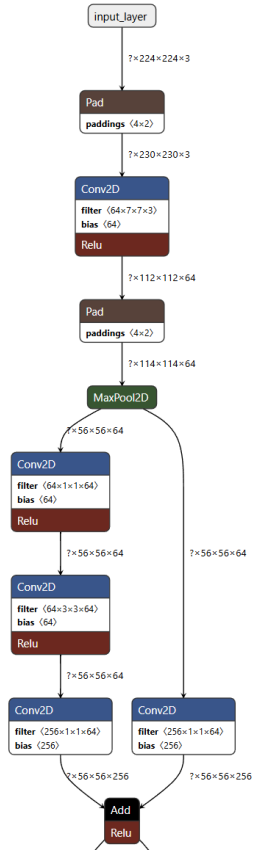
Target DNN Model: ResNet50 (1)

- We will use ResNet50 FP32 throughout the tutorial which is
  - A 50 layer deep convolutional neural network for image classification
  - Introduced in 2015 by He et al.[1] with the concept of skip connections
- Why ResNet50?
  - Commonly used for frame-by-frame classification
  - Moderate computational demand
  - Widely used as a benchmark model in inference optimization

[1]He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

Target DNN Model: ResNet50 (2)

- Key details
  - Input
    - 224x224 RGB image (224x224x3)
  - Output
    - 1x1000 tensor
- Observe model structure using Netron[2]
  - **`netron ./models/resnet50.tflite`**



[2]https://netron.app/

Contents

Hardware: RUBIK Pi 3 (1)

- Overview
  - A Singe Board Computer (SBC)
    built on Qualcomm AI platforms for developers
    - Based on QCS6490 SoC
  - Supports multiple operating systems such as
    - Android 13, Qualcomm Linux, Debian 13
  - Supports SDKs with Qualcomm AI stack
  - Designed and manufactured by Thundercomm

Hardware: RUBIK Pi 3 (2)

- Specification
  - RUBIK Pi 3 with QCS6490 SoC
    - CPU: 8 x Kryo 670 cores
    - GPU: Adreno 643L
    - NPU: Hexagon 770
    - RAM: 8 GB LPDDR4X
    - Storage: 128 GB UFS 2.2

Hardware: RUBIK Pi 3 (3)

- Comparison with Raspberry Pi 5
  - RUBIK Pi offers superior AI performance and faster storage, making it ideal for edge AI education and prototyping

	RUBIK Pi 3	Raspberry Pi 5
CPU	**8 x Kryo 670** (1 x Cortex-A78 @ 2.7 GHz, 3 x Cortex-A78 @ 2.4 GHz, 4 x Cortex-A55 @ 1.9 GHz)	4 x Cortex-A76 @ 2.4GHz
GPU	**Adreno 643L @ Up to 812 MHz**	VideoCore VII @ Up to 800MHz
NPU	**Hexagon 770 (12 TOPS)**	N/A
RAM	8GB	Up to 16GB
ROM	**128GB UFS 2.2**	N/A (Need to purchase external SD card separately)
Price	$179	$120 (16 GB Model)

Hardware: RUBIK Pi 3 (4)

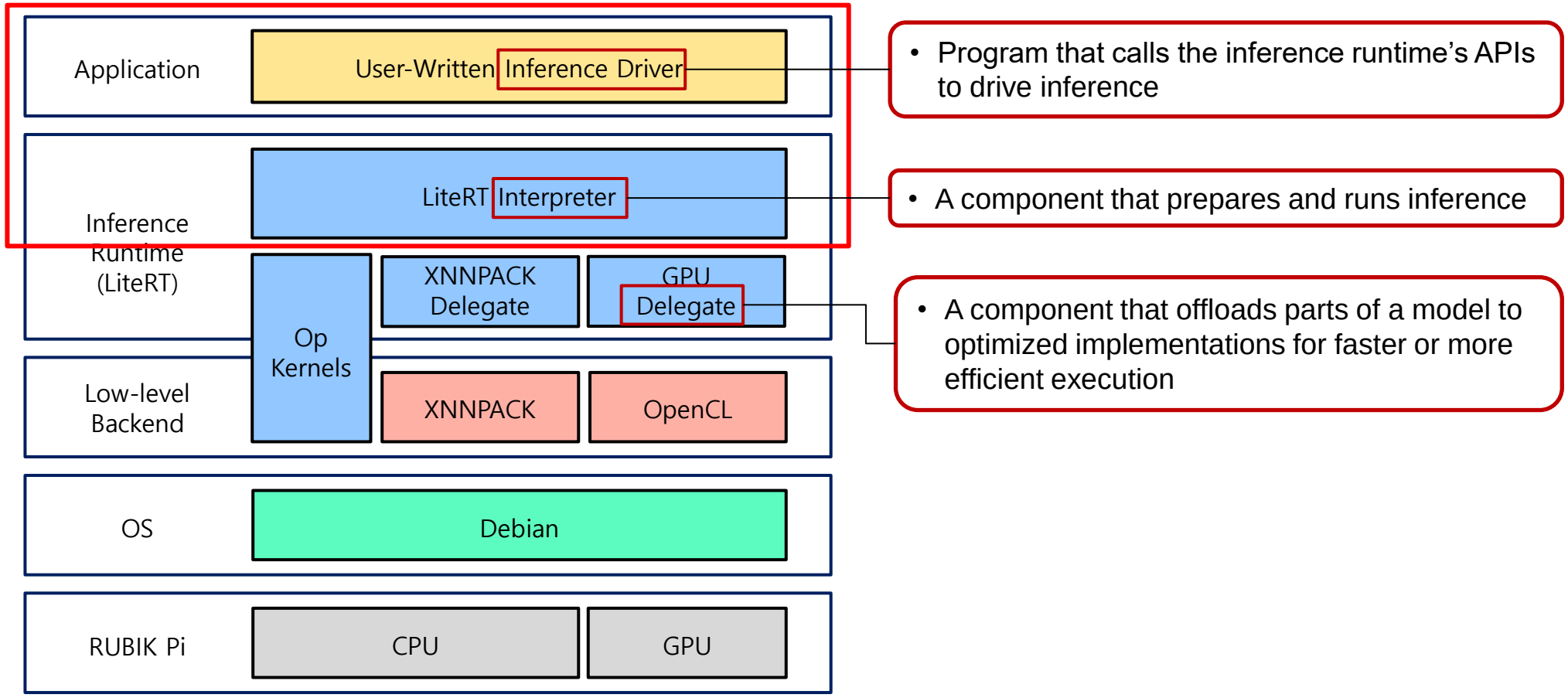
- Ideal for hands-on learning of embedded AI software stack
  - RUBIK Pi 3 with QCS6490 SoC enables realistic, practice-oriented exercises aligned with real-world industrial applications

Source: Thundercomm (https://www.thundercomm.com/rubik-pi-3-embedded-world-2025/)

Taxonomy

- Inference driver
  - Program that calls the inference runtime's APIs to drive inference
- Inference runtime
  - A software environment that executes DNN inference
    - Such as LiteRT (Google), QNN (Qualcomm), TensorRT (Nvidia)
- LiteRT interpreter
  - A component that prepares and runs inference
- Delegate
  - A component that offloads parts of a model to optimized implementations for faster or more efficient execution

Software Stack on RUBIK Pi 3



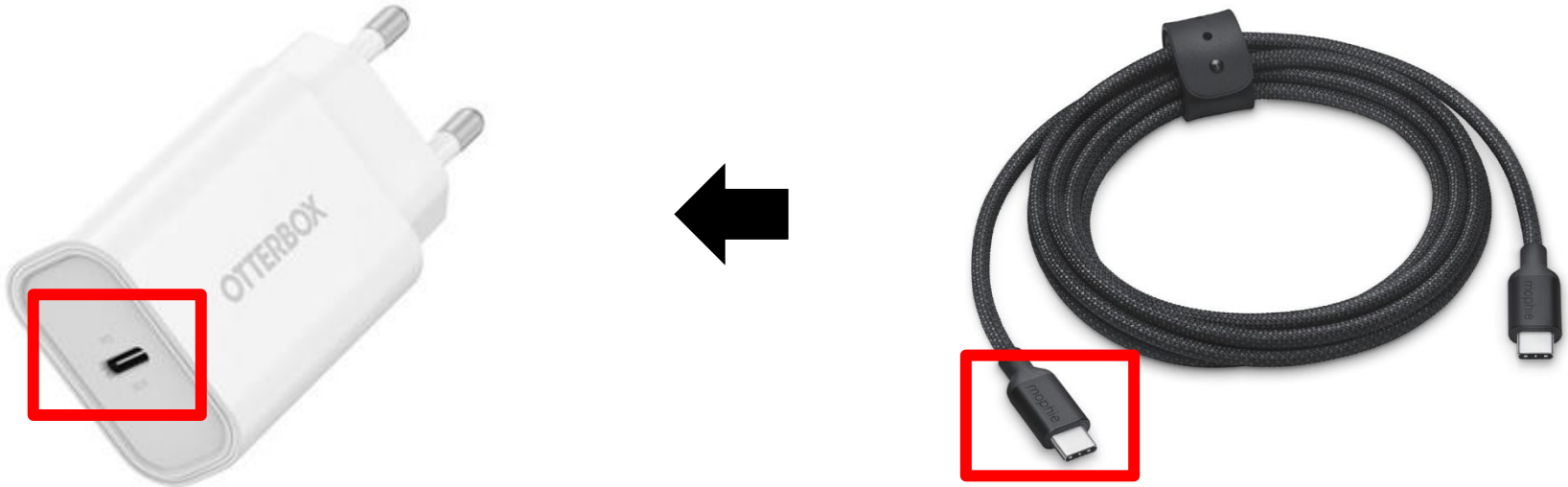
Contents

Hands-On: Connect to RUBIK Pi

- Objective
  - Establish an SSH connection from your laptop to RUBIK Pi
- Do
  - Follow the instructions in this section
- Verify
  - Your VS Code should show a SSH session connected to RUBIK Pi
- Time
  - 10 minutes

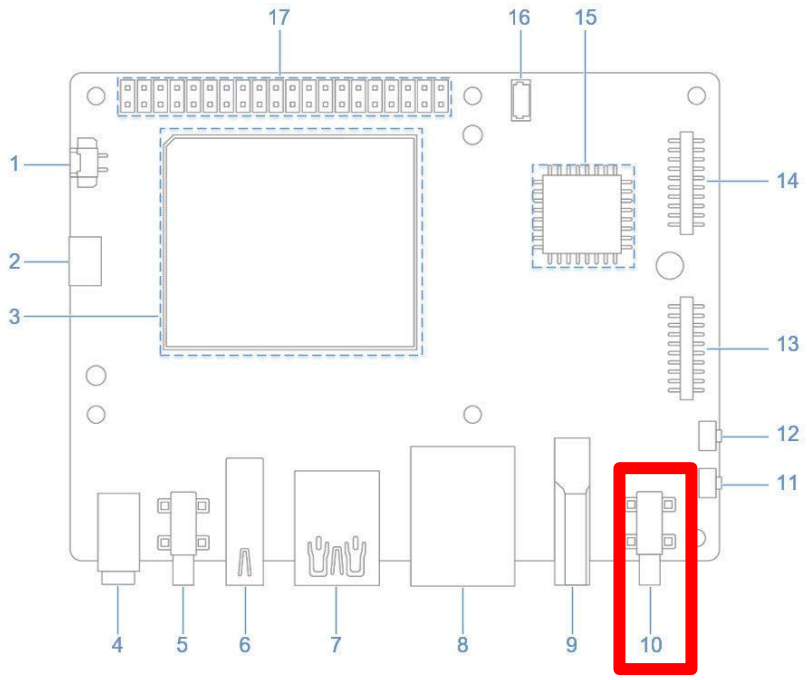
Supply Power (1)

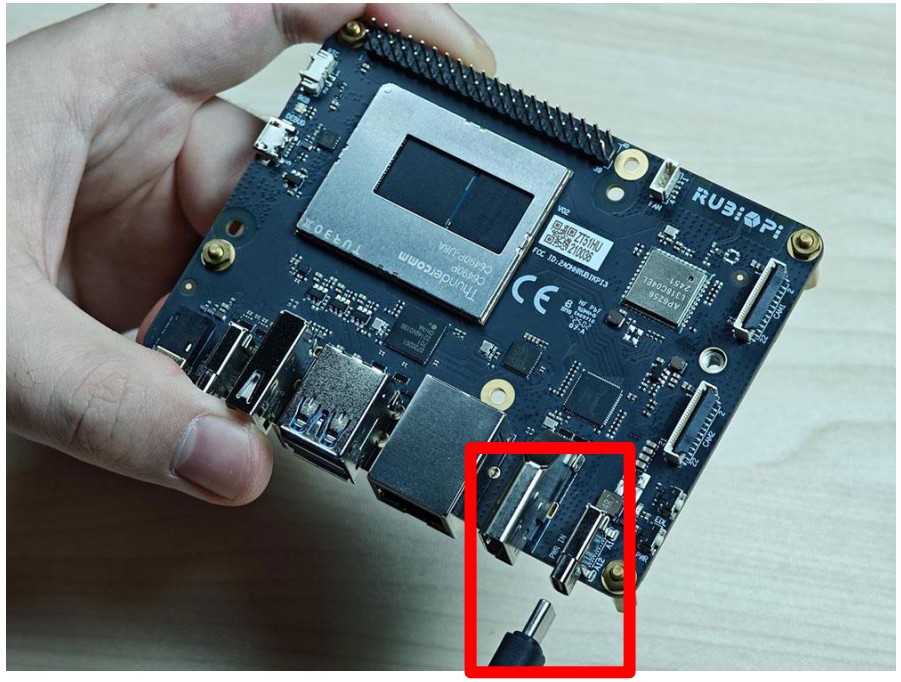
- Connect power cable to the external power supply, and make sure it is switched on



Supply Power (2)

- Connect Type-C power cable to port 10 on RUBIK Pi



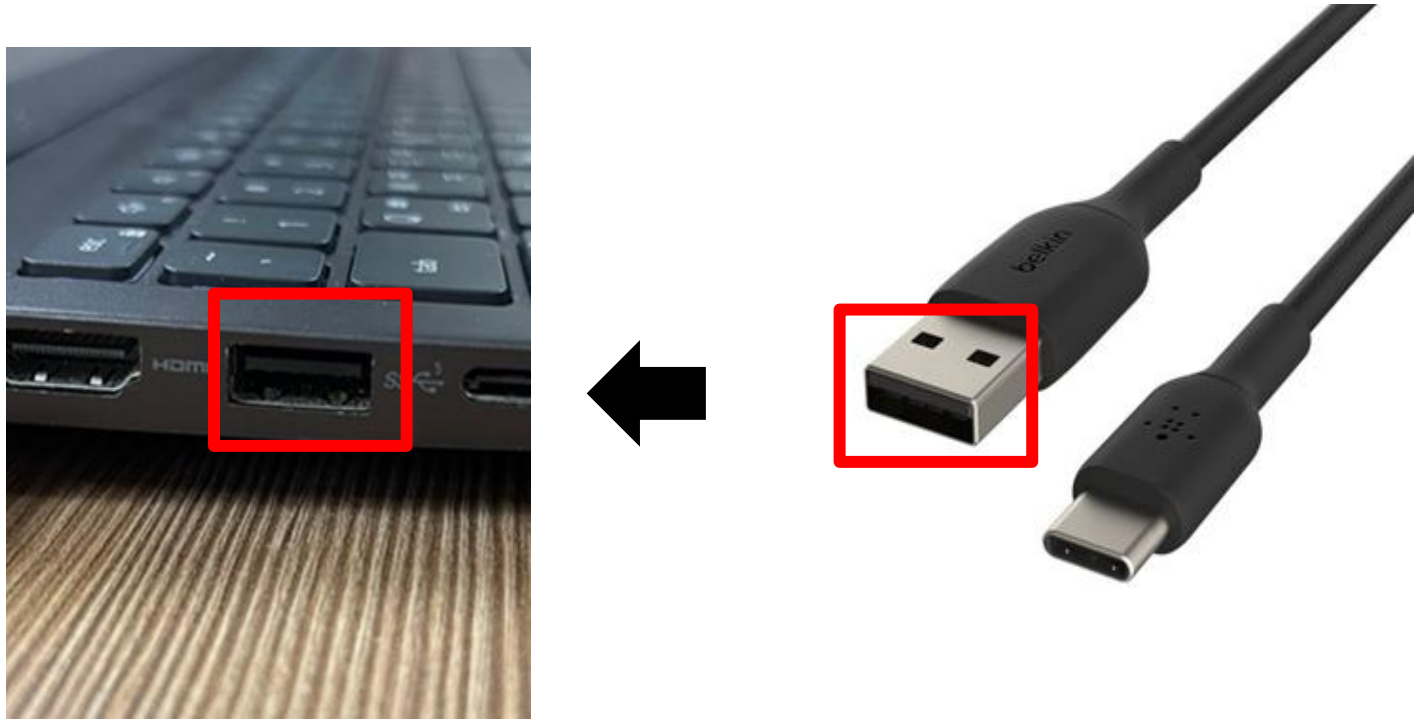


Connect RUBIK Pi to Host PC

- We will access the RUBIK Pi using SSH with Visual Studio Code for development ease
  - To do so
    1. Connect the device to the host PC with a USB-A to USB-C cable
    2. Establish the device's Wi-Fi connection using ADB (Android Debug Bridge)
    3. Make an SSH connection to the device

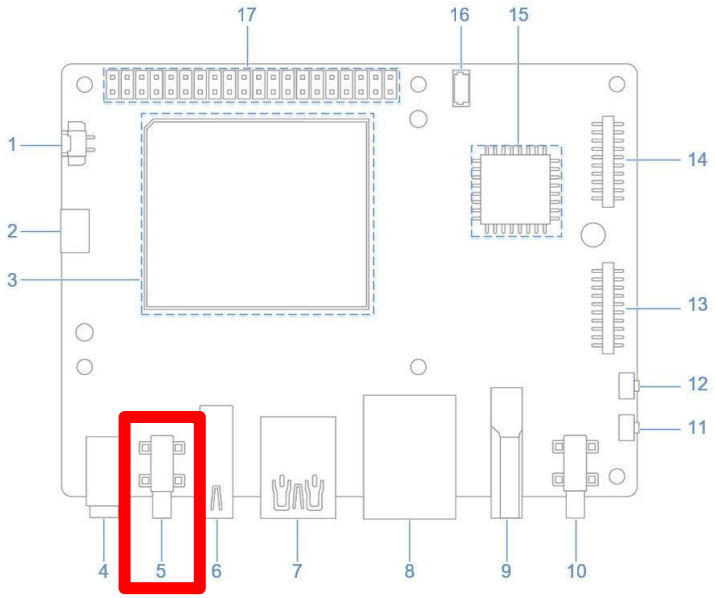
1. Connect Device to Host PC with USB-A to USB-C Cable (1)

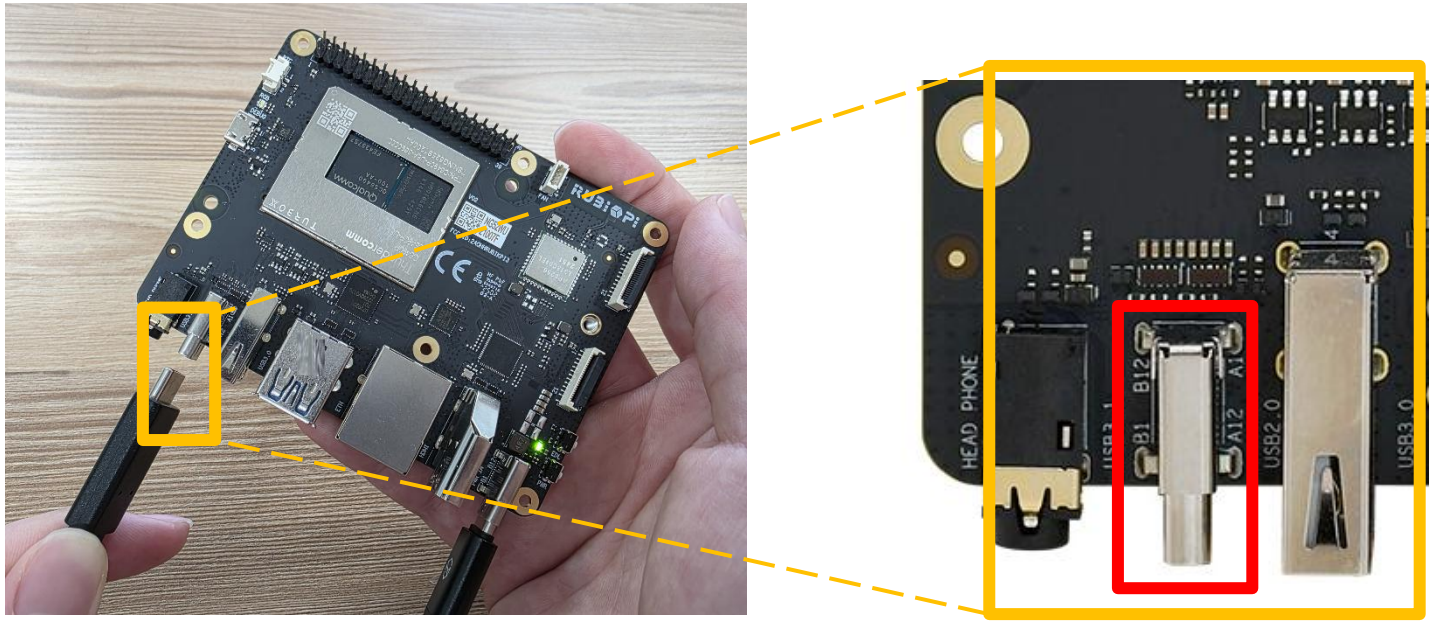
- Connect the USB-A end of the cable to host PC



1. Connect Device to Host PC with USB-A to USB-C Cable (2)

- Connect the USB-C end of the cable to port 5





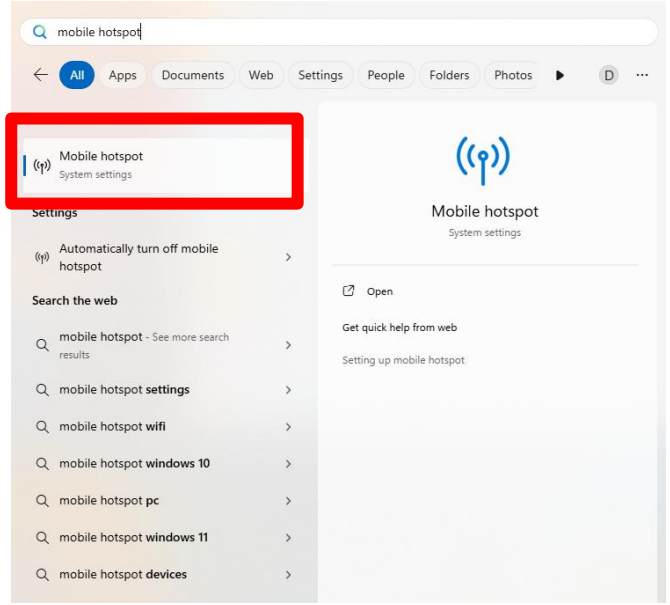
1. Connect Device to Host PC with USB-A to USB-C Cable (3)

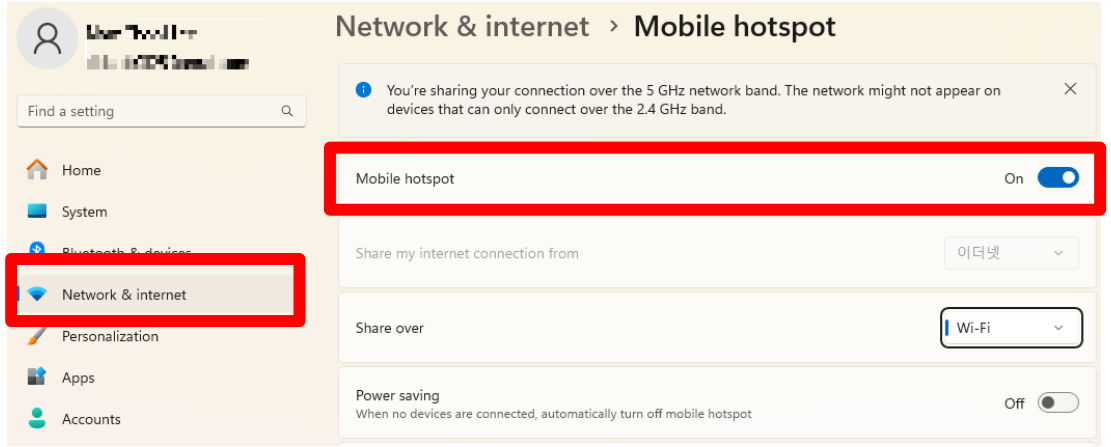
- Check the connection using ADB
  - It should show one device as below
    - The identifier of the device varies across devices

```
C:\Users\Lee>adb devices
List of devices attached
f10e86f9        device
```

2. Establish Wi-Fi Connection (1)

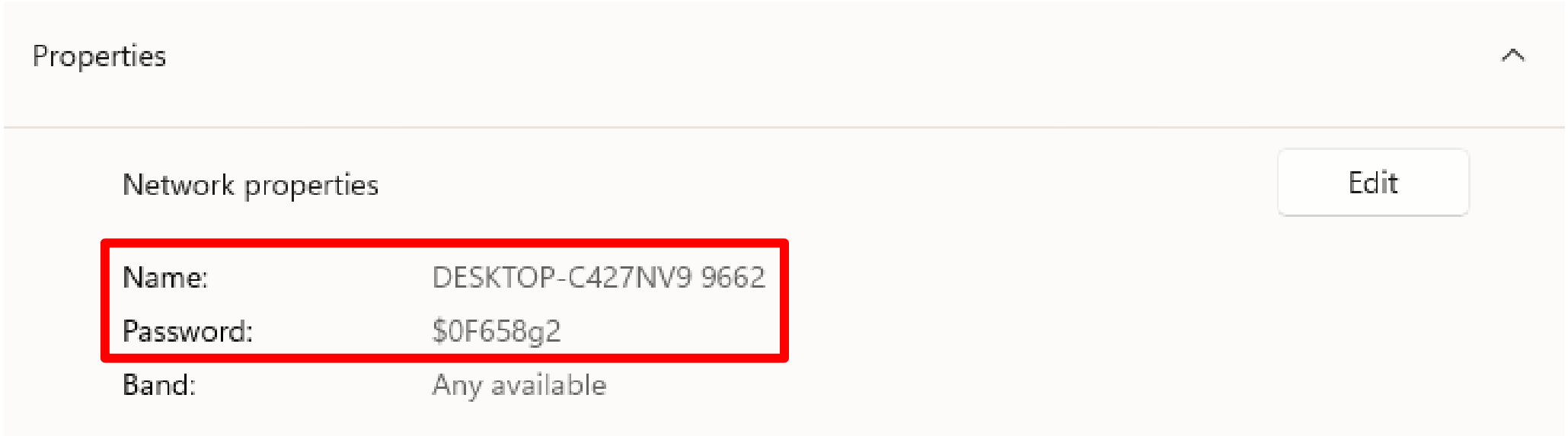
- Enable hotspot on host PC
  - Go to **Settings > Network & Internet**





2. Establish Wi-Fi Connection (2)

- Enable hotspot on host PC (cont'd)
  - Check the name and password of the activated network



2. Establish Wi-Fi Connection (3)

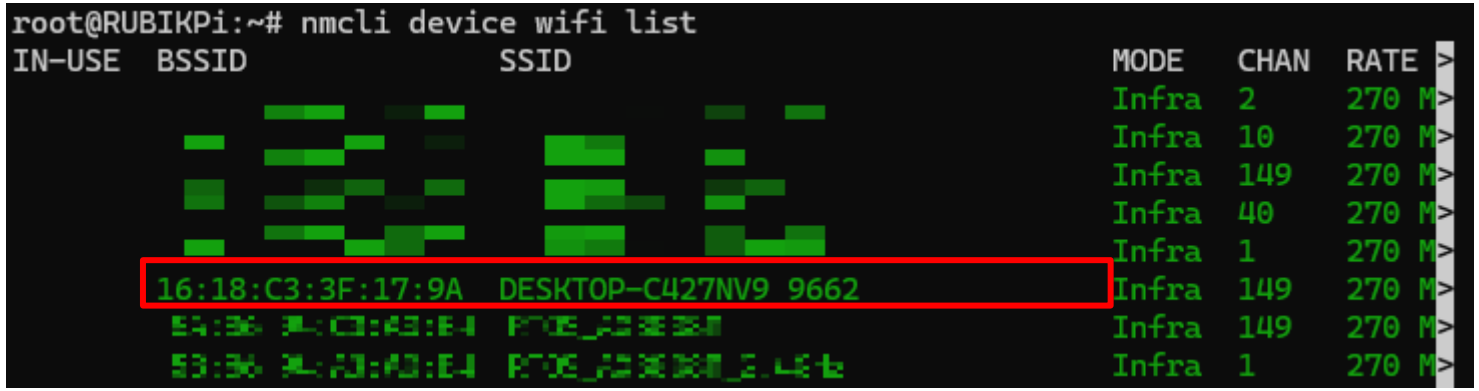
- Access the device in a command line tool with the following commands
  1. Get root access
     - **`adb root`**
  2. Enter adb shell
     - **`adb shell`**
  3. Switch to root user
     - **`su -`**

```
C:\Users\Lee>adb root
adbd is already running as root

C:\Users\Lee>adb shell
# su
root@RUBIKPi:/#
```

2. Establish Wi-Fi Connection (4)

- Connect RUBIK Pi to the mobile hotspot
  1. Check the available Wi-Fis using network manager CLI (**`nmcli`**) of Linux
     - **`nmcli device wifi list`**



2. Establish Wi-Fi Connection (5)

- Connect RUBIK Pi to the mobile hotspot (cont'd)
  2. Connect to the mobile hotspot using nmcli
     - **`nmcli device wifi connect "SSID" –ask`**

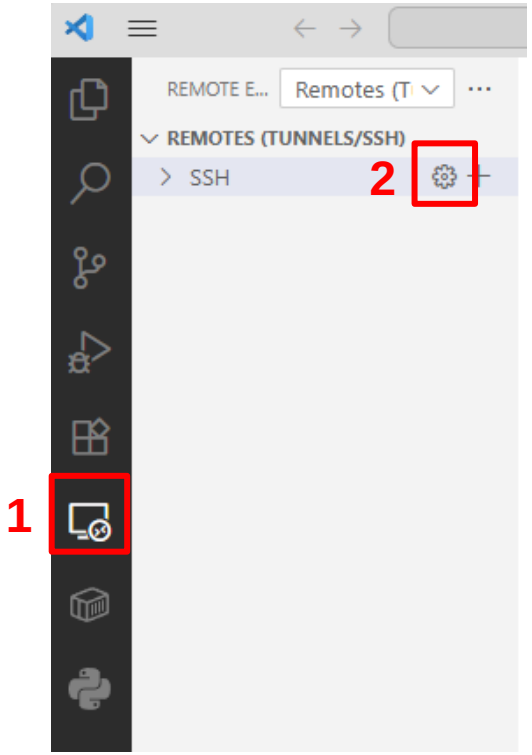
```
root@RUBIKPi:~# nmcli device wifi connect "DESKTOP-C427NV9 9662" --ask
Push of the WPS button on the router or a password is required to access the wireless
 network 'DESKTOP-C427NV9 9662'.
Password (802-11-wireless-security.psk): ********
Device 'wlan0' successfully activated with '0ea65cff-dd17-4aa1-b934-2dedbe33d58f'.
```

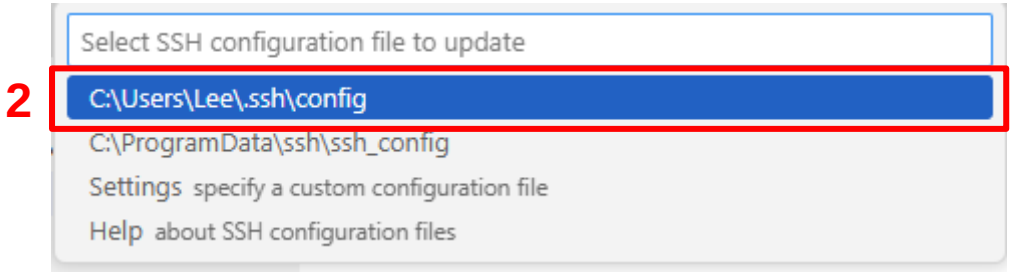
  3. Check the connection
     - **`ifconfig wlan0`**

```
root@RUBIKPi:~# ifconfig wlan0
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.137.35  netmask 255.255.255.0  broadcast 192.168.137.255
        inet6 fe80::72c0:c63d:66e2:c284  prefixlen 64  scopeid 0x20<link>
        ether 9c:b8:b4:9f:4f:5c  txqueuelen 1000  (Ethernet)
        RX packets 428199  bytes 627300827 (598.2 MiB)
        RX errors 0  dropped 17738  overruns 0  frame 0
        TX packets 89787  bytes 11357654 (10.8 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3. Make SSH Connection (1)

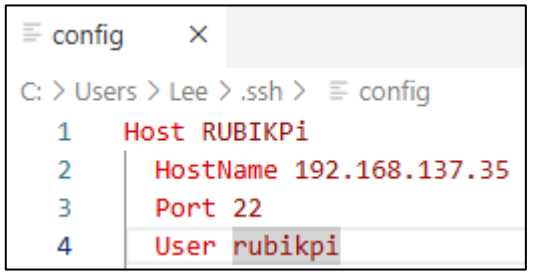
- Create SSH configuration in VS Code
  1. Open remote explorer in VS Code in the left side bar
  2. Open SSH config file



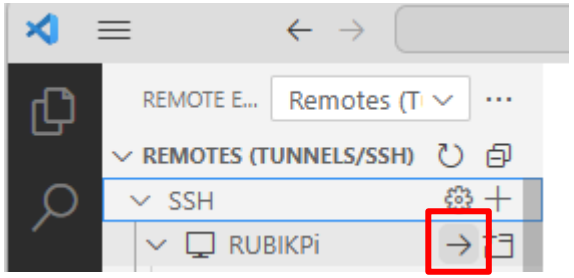


3. Make SSH Connection (2)

- Create SSH configuration in VS Code (cont'd)
  3. Add the configuration for RUBIK Pi

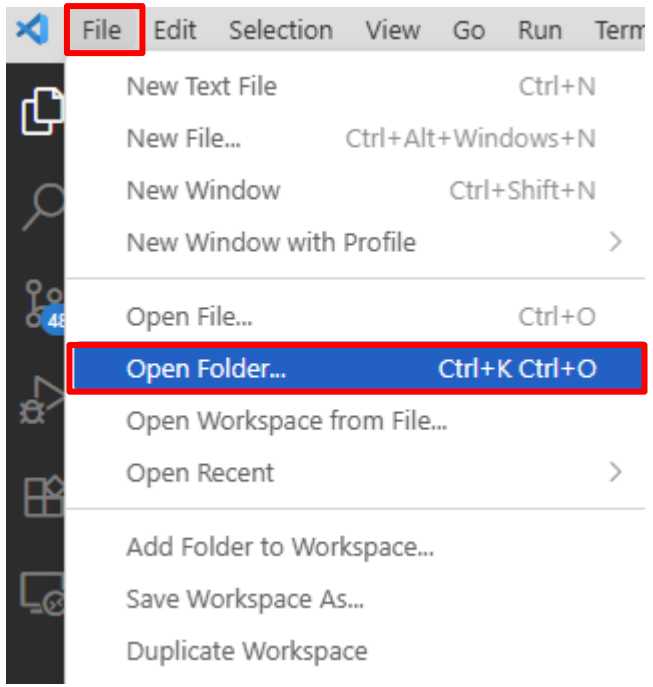


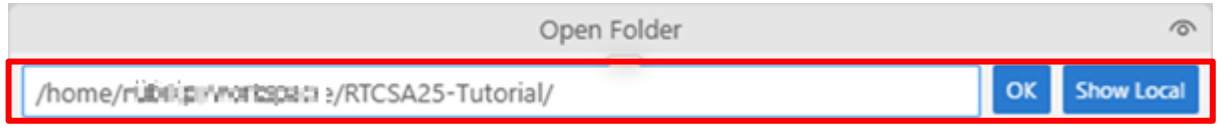
  4. Connect to the RUBIK Pi



3. Make SSH Connection (3)

- Open **`~/RTCSA25-Tutorial`** directory





Contents

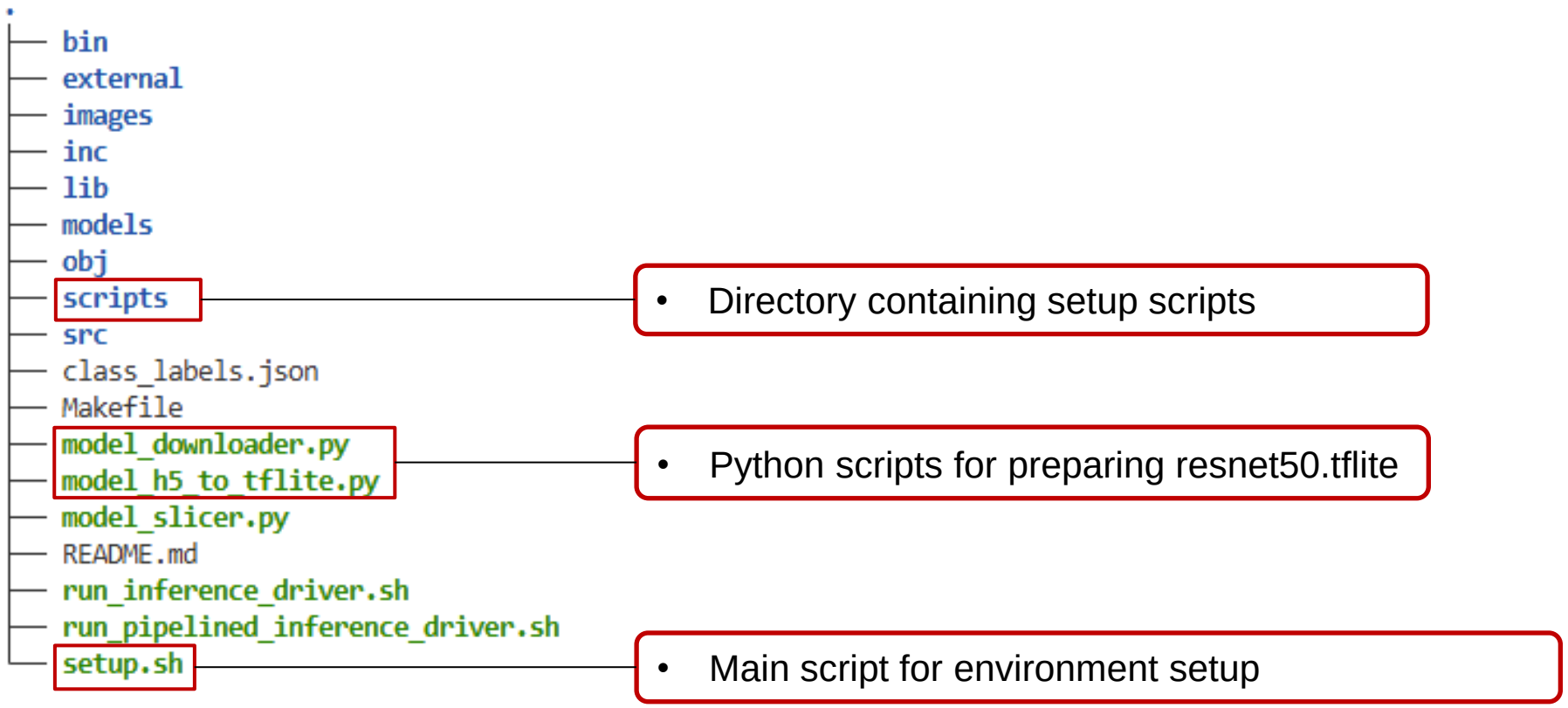
Directory Structure

- Under **`~/RTCSA2025-Tutorial`**
  - Available at https://github.com/SNU-RTOS/RTCSA25-Tutorial

```
.
├── bin
├── external
├── images
├── inc
├── lib
├── models
├── obj
├── scripts
├── src
├── class_labels.json
├── Makefile
├── model_downloader.py
├── model_h5_to_tflite.py
├── model_slicer.py
├── README.md
├── run_inference_driver.sh
├── run_pipelined_inference_driver.sh
└── setup.sh
```

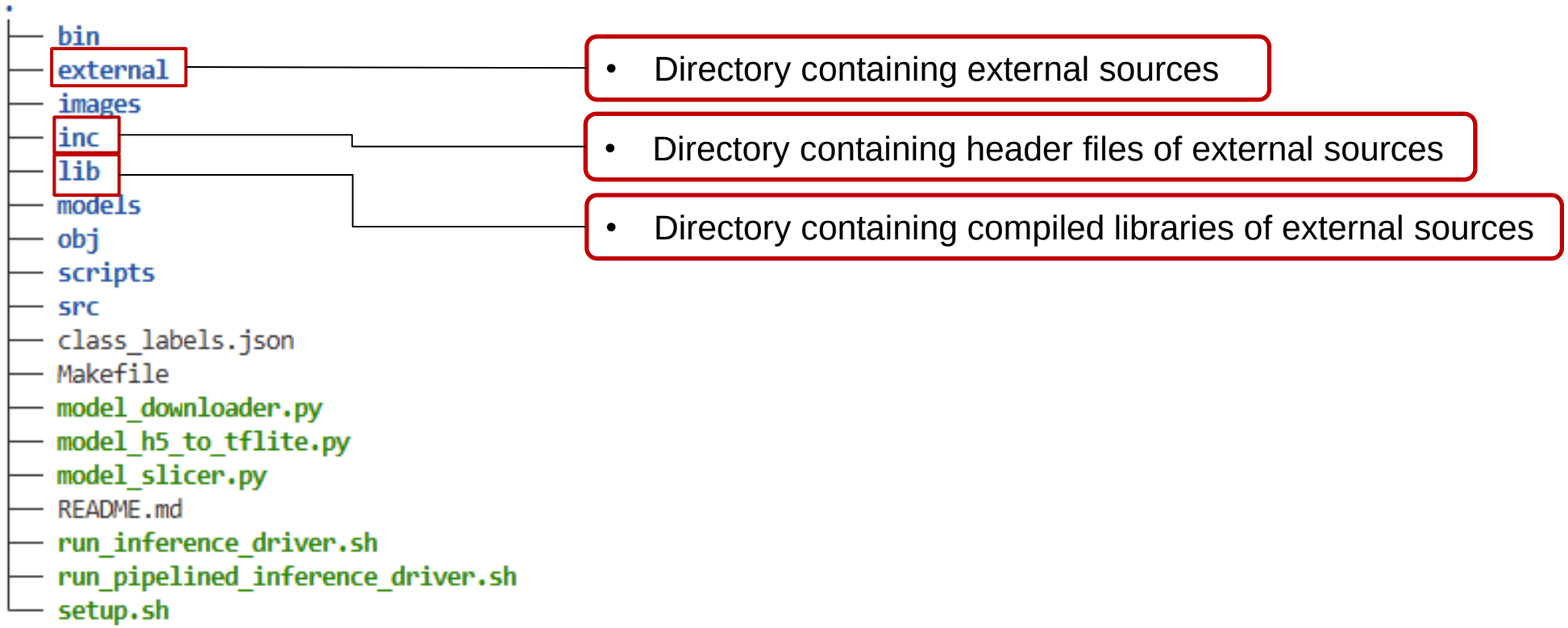
Setup Scripts

- Scripts to install software packages and prepare required files



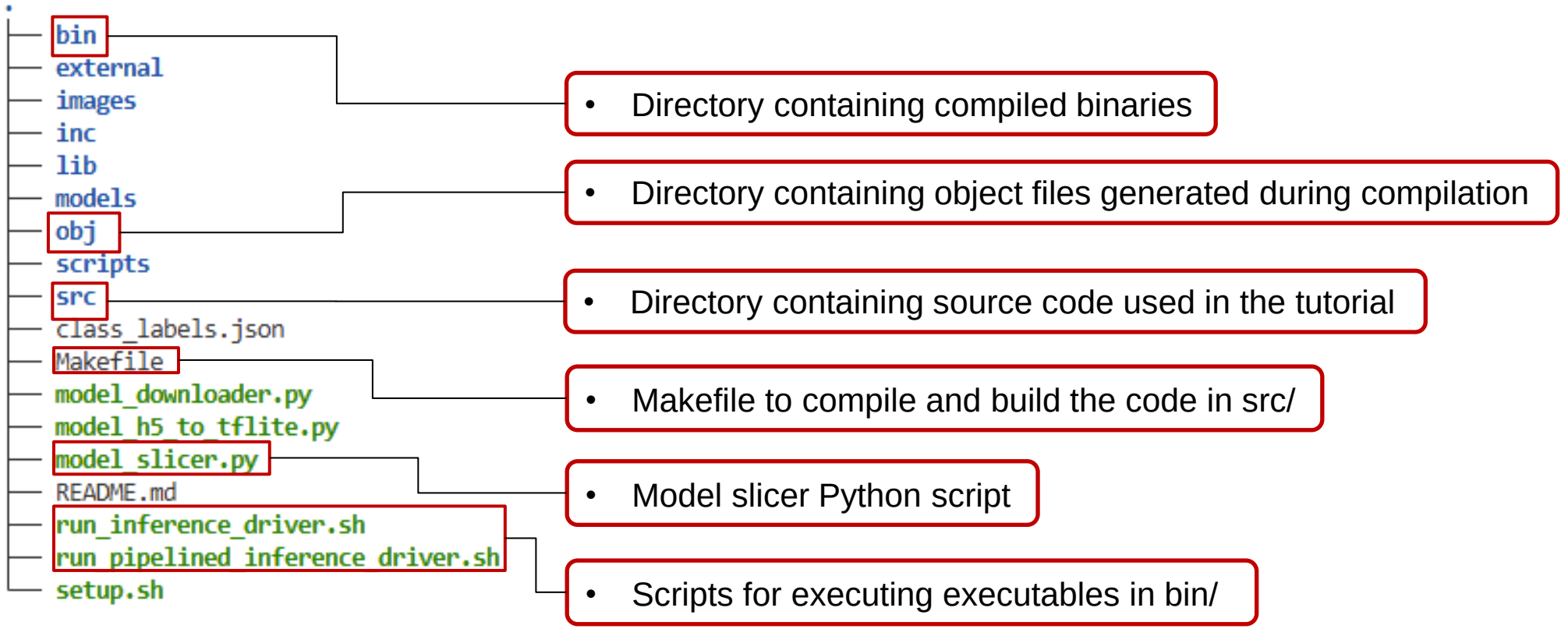
External Libraries

- Directories for externals libraries such as LiteRT



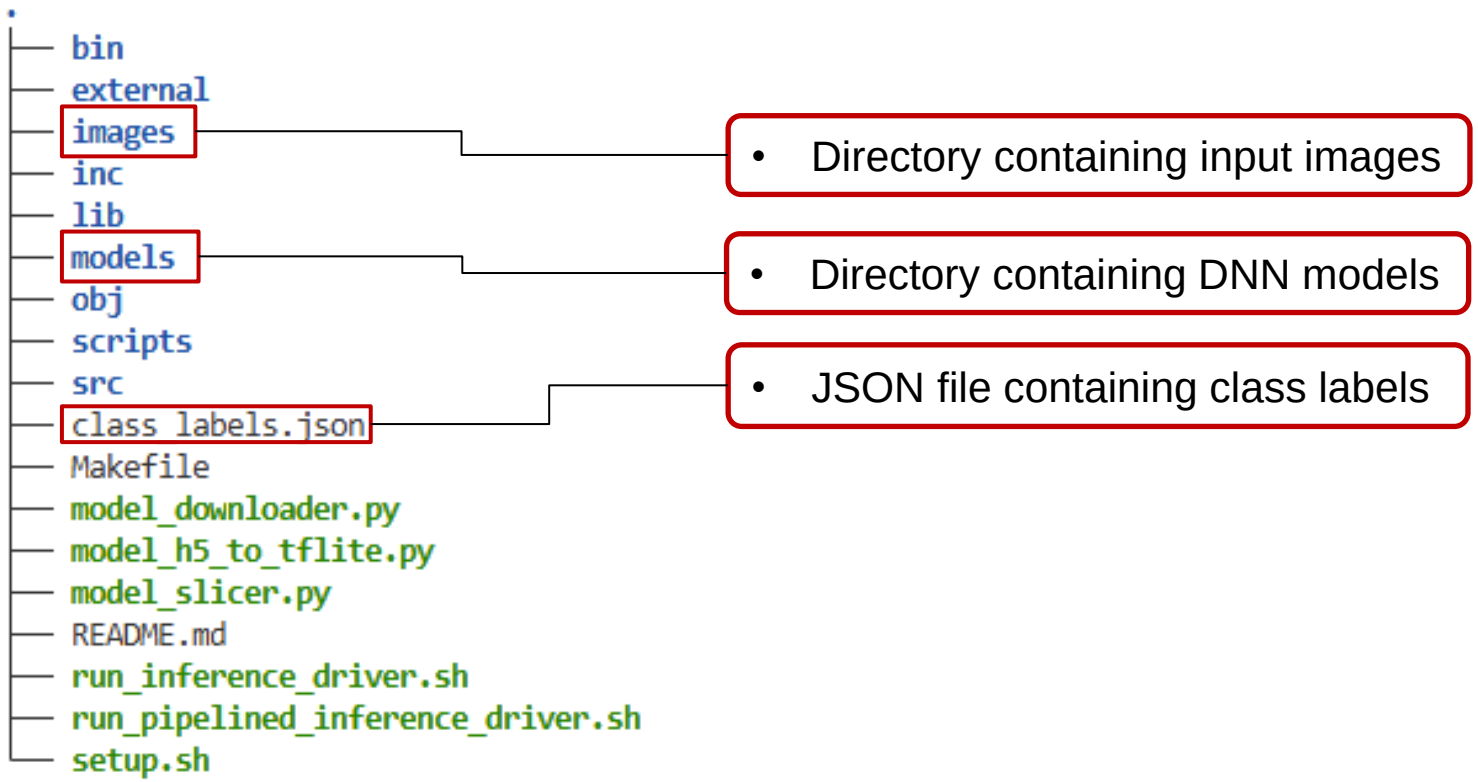
Source Code and Build File

- Source code and build file used throughout the tutorial



Input Data and Model Files

- Input files used by the compiled binaries



Contents

Header Files

- Include necessary header files

RTCSA25-Tutorial > src > inference_driver.cpp

```
1   #include <iostream>
2   #include <thread>
3   #include <vector>
4   #include <opencv2/opencv.hpp>
5   #include "tflite/delegates/xnnpack/xnnpack_delegate.h"
6   #include "tflite/delegates/gpu/delegate.h"
7   #include "tflite/interpreter_builder.h"
8   #include "tflite/interpreter.h"
9   #include "tflite/kernels/register.h"
10  #include "tflite/model_builder.h"
11  #include "util.hpp"
```

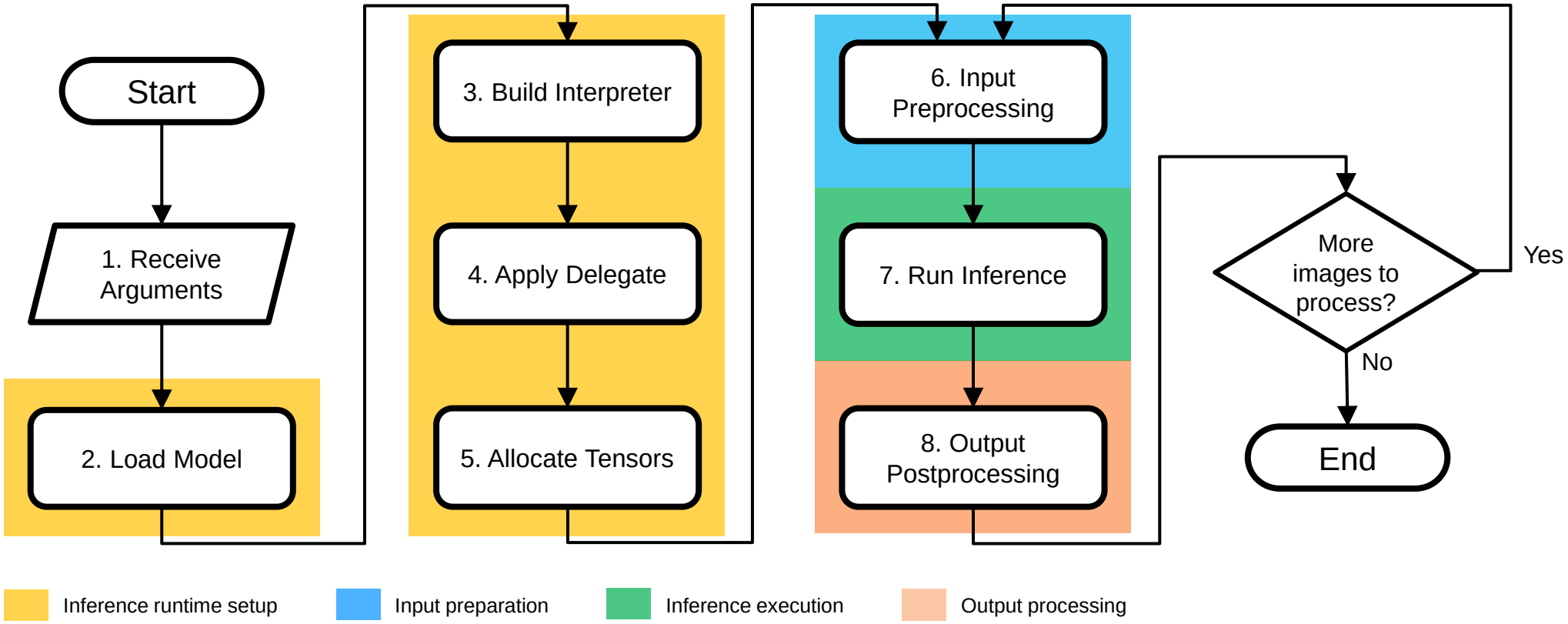
C++ standard library headers for I/O stream, thread management, and dynamic arrays

Header file for OpenCV used during input preprocessing

Header files for using LiteRT

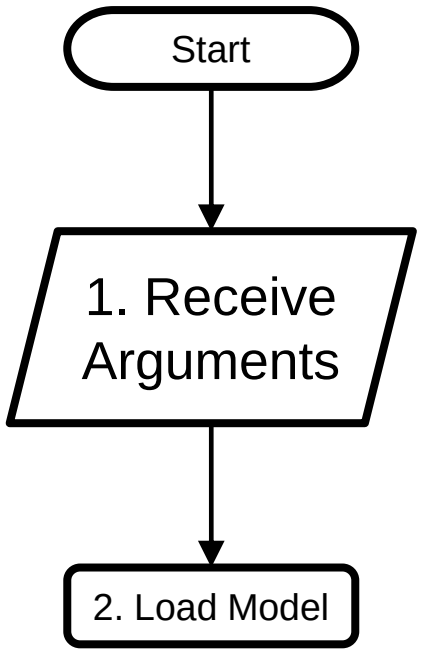
Header file for utility functions used during the tutorial

Inference Driver Code Flow



1. Receive Arguments (1)

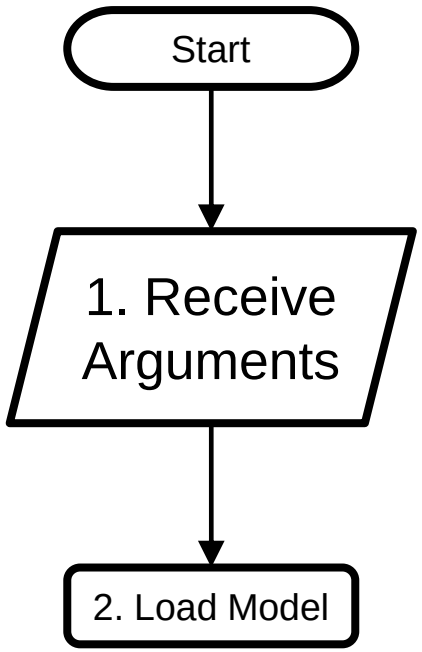
- Check the number of arguments




```
27      if (argc < 5)
28      {
29          std::cerr << "Usage: " << argv[0]
30              << "<model_path> <gpu_usage> <class_labels_path> <image_path 1> "
31              << "[image_path 2 ... image_path N] [--input-period=milliseconds]"
32              << std::endl;
33          return 1;
34      }
```

1. Receive Arguments (2)

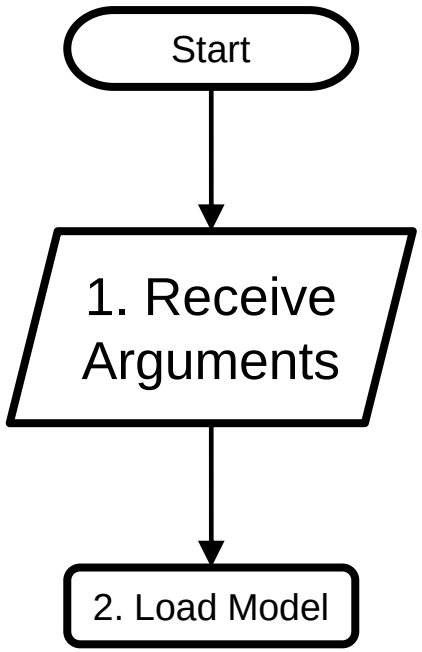
- Set variables based on the arguments




```
36      const std::string model_path = argv[1];
37
38      bool gpu_usage = false; // If true, GPU delegate is applied
39      const std::string gpu_usage_str = argv[2];
40      if(gpu_usage_str == "true"){
41          gpu_usage = true;
42      }
43
44      // Load class label mapping, used for postprocessing
45      const std::string class_labels_path = argv[3];
46      auto class_labels_map = util::load_class_labels(class_labels_path.c_str());
```

1. Receive Arguments (3)

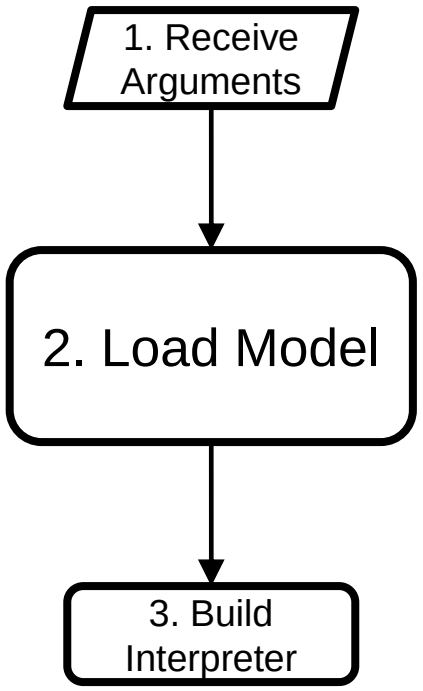
- Set variables based on the arguments (cont'd)




```
48      std::vector<std::string> images;    // List of input image paths
49      int input_period_ms = 0;            // Input period in milliseconds, default is 0 (no delay)
50      for (int i = 4; i < argc; ++i) {
51          std::string arg = argv[i];
52          if (arg.rfind("--input-period=", 0) == 0) // Check for input period argument
53              input_period_ms = std::stoi(arg.substr(15));
54          else
55              images.push_back(arg);  // Assume it's an image path
56      }
```

2. Load Model

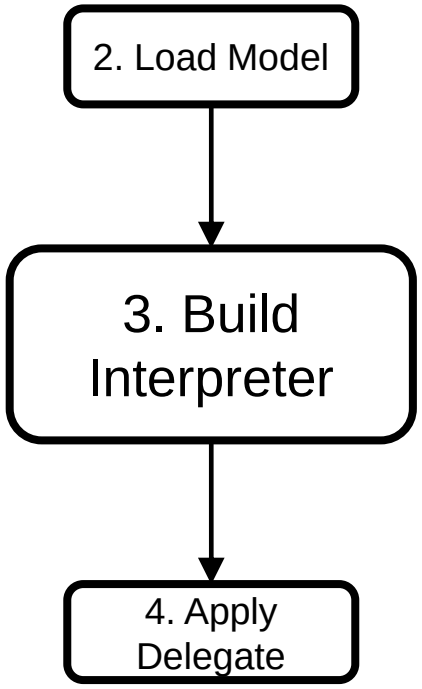
- Load a model file at **`model_path`**




```
58      /* Load model */
59      std::unique_ptr<tflite::FlatBufferModel> _litert_model =
60          tflite::FlatBufferModel::BuildFromFile(model_path.c_str());
61      if (!_litert_model)
62      {
63          std::cerr << "Failed to load model" << std::endl;
64          return 1;
65      }
```

3. Build Interpreter (1)

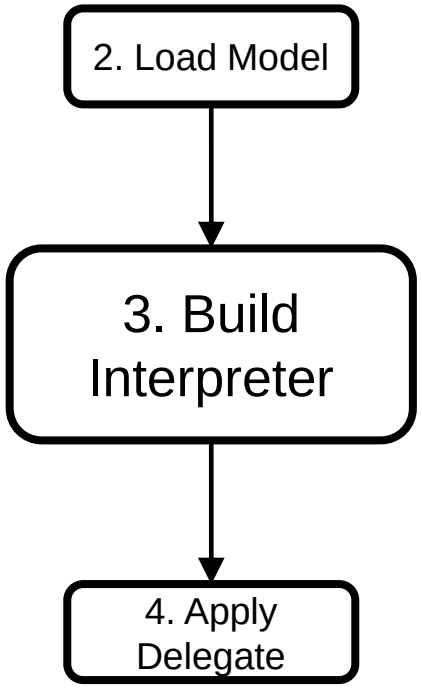
- Create an **`OpResolver`**
  - A registry that maps operator names to their corresponding implementations




```
67      /* Build interpreter */
68      tflite::ops::builtin::BuiltinOpResolver _litert_resolver;
69      tflite::InterpreterBuilder _litert_builder(*_litert_model, _litert_resolver);
70      std::unique_ptr<tflite::Interpreter> _litert_interpreter;
71      _litert_builder(&_litert_interpreter);
72      if (!_litert_interpreter)
73      {
74          std::cerr << "Failed to Initialize Interpreter" << std::endl;
75          return 1;
76      }
```

3. Build Interpreter (2)

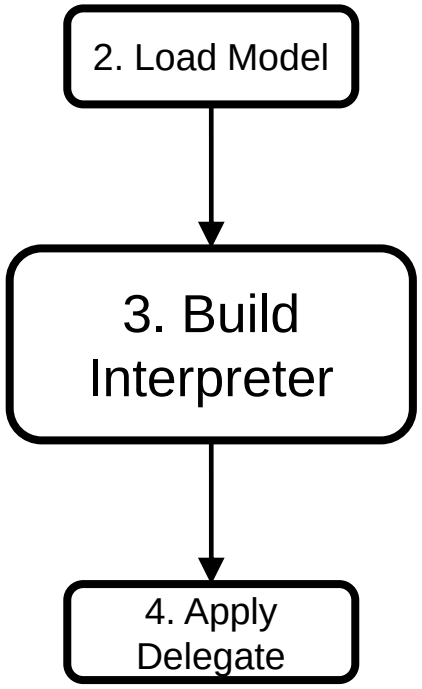
- Create an **`InterpreterBuilder`**
  - An object that constructs an interpreter instance using the builder pattern




```
67      /* Build interpreter */
68      tflite::ops::builtin::BuiltinOpResolver _litert_resolver;
69      tflite::InterpreterBuilder _litert_builder(*_litert_model, _litert_resolver);
70      std::unique_ptr<tflite::Interpreter> _litert_interpreter;
71      _litert_builder(&_litert_interpreter);
72      if (!_litert_interpreter)
73      {
74          std::cerr << "Failed to Initialize Interpreter" << std::endl;
75          return 1;
76      }
```

3. Build Interpreter (3)

- Create and build an interpreter
  - If successfully, **`_litert_interpreter`** will point to a valid interpreter object

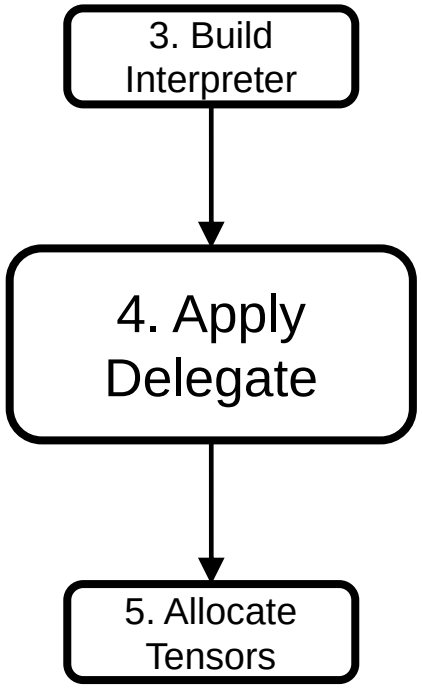



```
67      /* Build interpreter */
68      tflite::ops::builtin::BuiltinOpResolver _litert_resolver;
69      tflite::InterpreterBuilder _litert_builder(*_litert_model, _litert_resolver);
70      std::unique_ptr<tflite::Interpreter> _litert_interpreter;
71      _litert_builder(&_litert_interpreter);
72      if (!_litert_interpreter)
73      {
74          std::cerr << "Failed to Initialize Interpreter" << std::endl;
75          return 1;
76      }
```

- The interpreter builder performs **model validation**, **resolves operators** in the model, and **instantiates the internal objects** required by the interpreter

4. Apply Delegate (1)

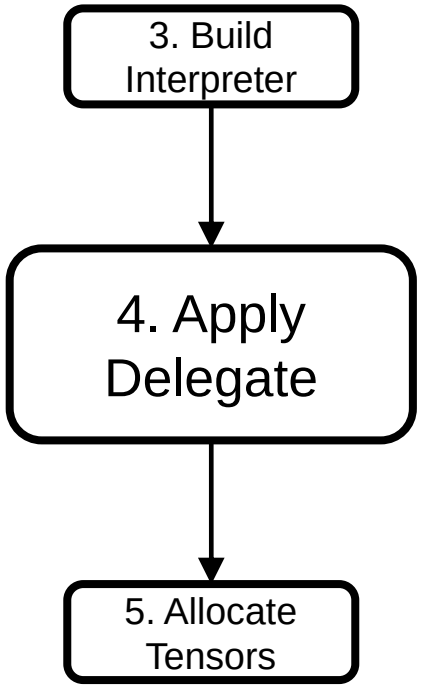
- Create delegate object(s)
  - For CPU execution optimization (XNNPACK) or GPU offloading




```
78      /* Apply either XNNPACK delegate or GPU delegate */
79      TfLiteDelegate* _litert_xnn_delegate = TfLiteXNNPackDelegateCreate(nullptr);
80      TfLiteDelegate* _litert_gpu_delegate = TfLiteGpuDelegateV2Create(nullptr);
```

4. Apply Delegate (2)

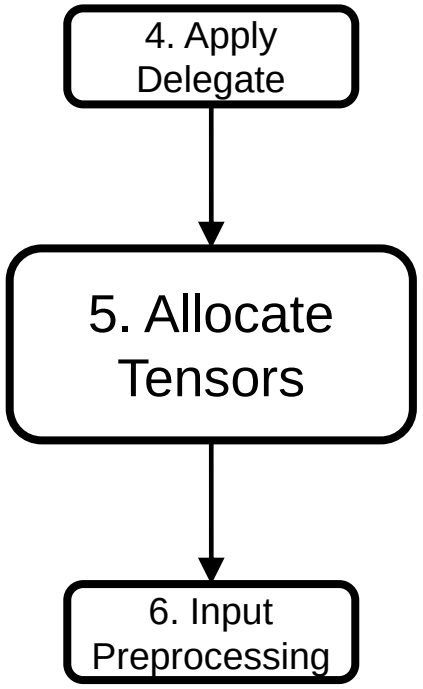
- Apply either the GPU delegate or the XNNPACK delegate
  - The delegate takes over supported parts of the model




```
81  if(gpu_usage) {
82      if (_litert_interpreter->ModifyGraphWithDelegate(_litert_gpu_delegate) == kTfLiteOk)
83      {
84          // Delete unused delegate
85          if(_litert_xnn_delegate) TfLiteXNNPackDelegateDelete(_litert_xnn_delegate);
86      } else {
87          std::cerr << "Failed to Apply GPU Delegate" << std::endl;
88      }
89  } else {
90      if (_litert_interpreter->ModifyGraphWithDelegate(_litert_xnn_delegate) == kTfLiteOk)
91      {
92          // Delete unused delegate
93          if(_litert_gpu_delegate) TfLiteGpuDelegateV2Delete(_litert_gpu_delegate);
94      } else {
95          std::cerr << "Failed to Apply XNNPACK Delegate" << std::endl;
96      }
97  }
```

5. Allocate Tensors

- Allocate memory for mutable tensors
  - Tensors that store input data, intermediate values, and final results during inference

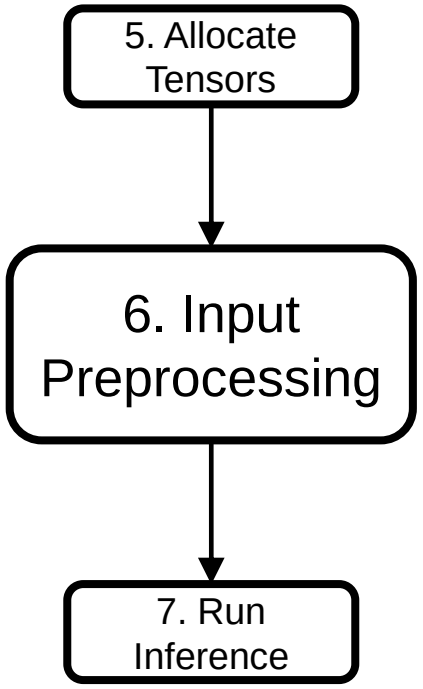



```
 99      /* Allocate Tensors */
100      if (_litert_interpreter->AllocateTensors() != kTfLiteOk)
101      {
102          std::cerr << "Failed to Allocate Tensors" << std::endl;
103          return 1;
104      }
```

- If any tensor is not allocated,
  a segmentation fault will occur during inference

6. Input Preprocessing (1)

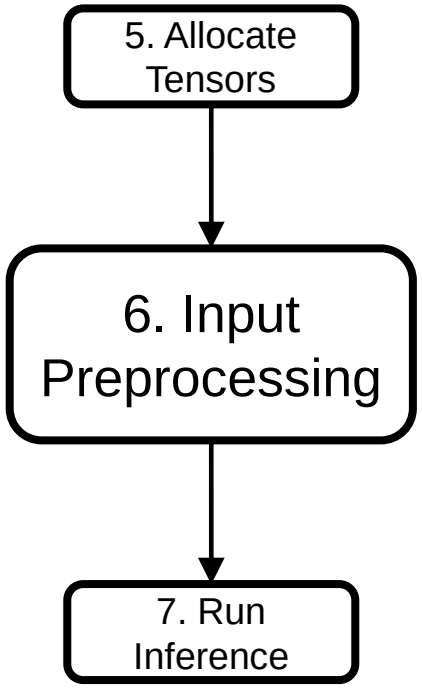
- Enters a do-while loop
  - Record the current time before starting, for use in periodic loop execution later




```
110     auto next_wakeup_time = std::chrono::high_resolution_clock::now();
111     do {
119         /* Preprocessing */
120         // Load input image
121         cv::Mat image = cv::imread(images[count]);
122         if (image.empty()) {
123             std::cerr << "Failed to load image: " << images[count] << "\n";
124             continue;
125         }
126
127         // Preprocess input data
128         cv::Mat preprocessed_image =
129                 util::preprocess_image_resnet(image, 224, 224);
130
131         // Copy preprocessed_image to input_tensor
132         float* _litert_input_tensor = _litert_interpreter->typed_input_tensor<float>(0);
133         std::memcpy(_litert_input_tensor, preprocessed_image.ptr<float>(),
134                     preprocessed_image.total() * preprocessed_image.elemSize());
```

6. Input Preprocessing (2)

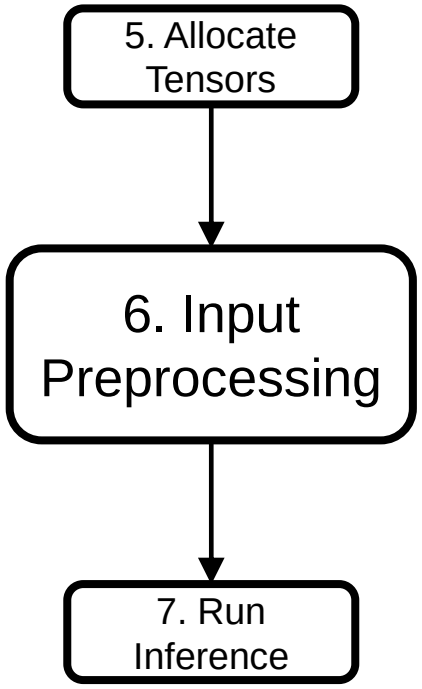
- Load an image




```
110     auto next_wakeup_time = std::chrono::high_resolution_clock::now();
111     do {
119         /* Preprocessing */
120         // Load input image
121         cv::Mat image = cv::imread(images[count]);
122         if (image.empty()) {
123             std::cerr << "Failed to load image: " << images[count] << "\n";
124             continue;
125         }
126
127         // Preprocess input data
128         cv::Mat preprocessed_image =
129                 util::preprocess_image_resnet(image, 224, 224);
130
131         // Copy preprocessed_image to input_tensor
132         float* _litert_input_tensor = _litert_interpreter->typed_input_tensor<float>(0);
133         std::memcpy(_litert_input_tensor, preprocessed_image.ptr<float>(),
134                     preprocessed_image.total() * preprocessed_image.elemSize());
```

6. Input Preprocessing (3)

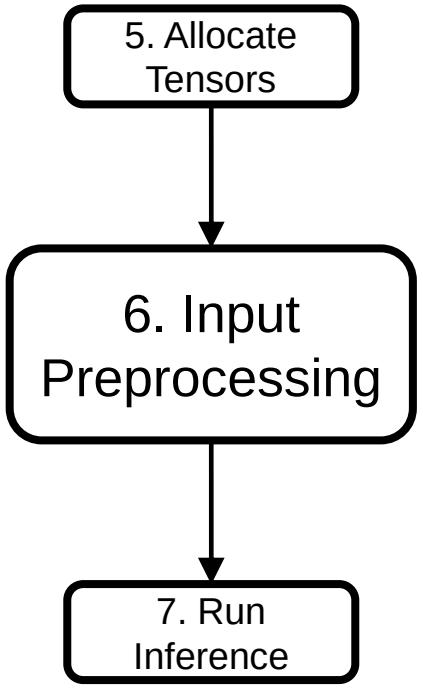
- Preprocess the loaded image




```
110     auto next_wakeup_time = std::chrono::high_resolution_clock::now();
111     do {
119         /* Preprocessing */
120         // Load input image
121         cv::Mat image = cv::imread(images[count]);
122         if (image.empty()) {
123             std::cerr << "Failed to load image: " << images[count] << "\n";
124             continue;
125         }
126
127         // Preprocess input data
128         cv::Mat preprocessed_image =
129                 util::preprocess_image_resnet(image, 224, 224);
130
131         // Copy preprocessed_image to input_tensor
132         float* _litert_input_tensor = _litert_interpreter->typed_input_tensor<float>(0);
133         std::memcpy(_litert_input_tensor, preprocessed_image.ptr<float>(),
134                     preprocessed_image.total() * preprocessed_image.elemSize());
```

6. Input Preprocessing (4)

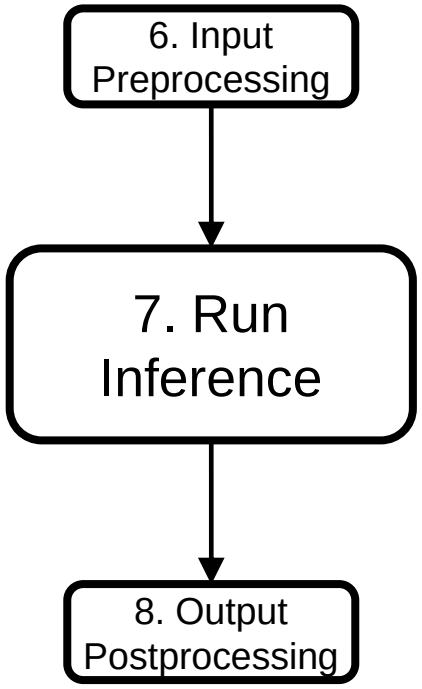
- Access the 0th input tensor as a float pointer and set its value
  - There is only one input tensor in ResNet50 FP32




```
110     auto next_wakeup_time = std::chrono::high_resolution_clock::now();
111     do {
119         /* Preprocessing */
120         // Load input image
121         cv::Mat image = cv::imread(images[count]);
122         if (image.empty()) {
123             std::cerr << "Failed to load image: " << images[count] << "\n";
124             continue;
125         }
126
127         // Preprocess input data
128         cv::Mat preprocessed_image =
129                 util::preprocess_image_resnet(image, 224, 224);
130
131         // Copy preprocessed image to input tensor
132         float* _litert_input_tensor = _litert_interpreter->typed_input_tensor<float>(0);
133         std::memcpy(_litert_input_tensor, preprocessed_image.ptr<float>(),
134                     preprocessed_image.total() * preprocessed_image.elemSize());
```

7. Run Inference

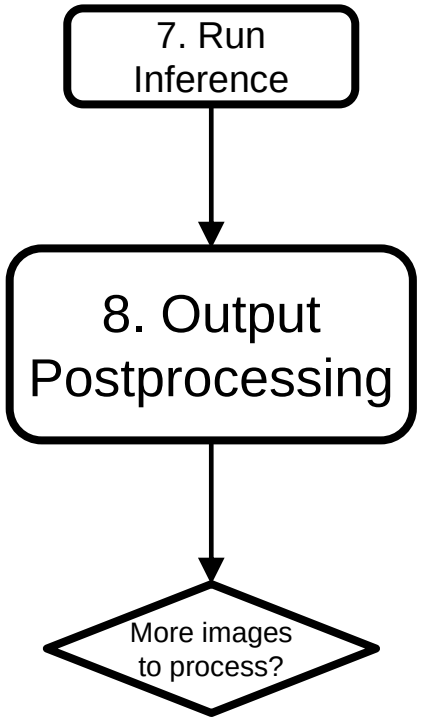
- Invoke the interpreter to run inference




```
139     /* Inference */
140     if (_litert_interpreter->Invoke() != kTfLiteOk)
141     {
142         std::cerr << "Failed to invoke the interpreter" << std::endl;
143         return 1;
144     }
```

8. Output Postprocessing (1)

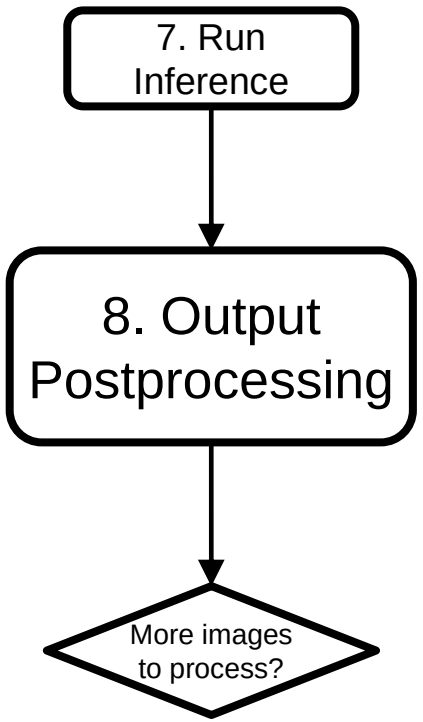
- Access the 0th output tensor as a float pointer
  - ResNet50 has only one output tensor with a shape of 1×1000




```
148     /* PostProcessing */
149     // Get output tensor
150     float *_litert_output_tensor = _litert_interpreter->typed_output_tensor<float>(0);
151     int num_classes = 1000; // Total 1000 classes
152     std::vector<float> probs(num_classes);
153     std::memcpy(probs.data(), _litert_output_tensor, sizeof(float) * num_classes);
```

8. Output Postprocessing (2)

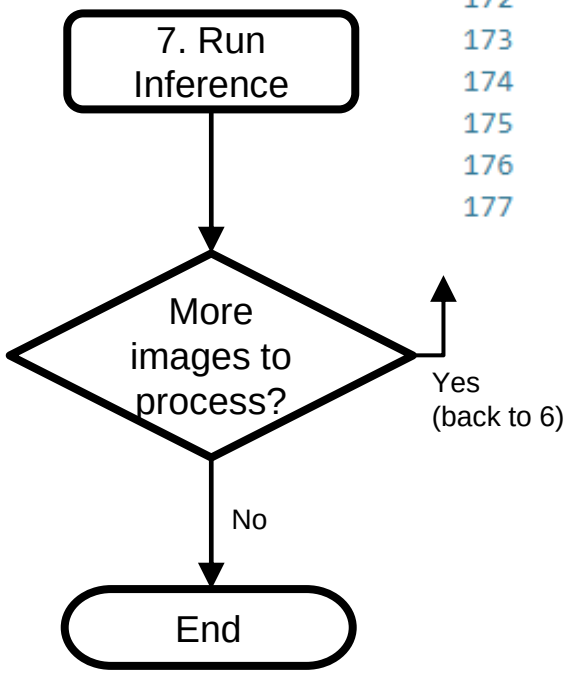
- Print the top 3 predictions with their labels




```
155         // Print Top-3 predictions every 10 iterations
156         if ((count + 1) % 10 == 0) {
157             std::cout << "\n[INFO] Top 3 predictions for image index " << count << ":"
158             << std::endl;
159             auto top_k_indices = util::get_topK_indices(probs, 3);
160             for (int idx : top_k_indices)
161             {
162                 std::string label =
163                     class_labels_map.count(idx) ? class_labels_map[idx] : "unknown";
164                 std::cout << "- Class " << idx << " (" << label << "): "
165                     << probs[idx] << std::endl;
166             }
167         }
```

Periodic Loop Execution

- Repeat step 6 at each input period until all inputs are processed
  - If the scheduled time has already passed, continue without waiting




```
172         // Sleep to control the input rate
173         // If next_wakeup_time is in the past, it will not sleep
174         next_wakeup_time += std::chrono::milliseconds(input_period_ms);
175         std::this_thread::sleep_until(next_wakeup_time);
176         ++count;
177     } while (count < images.size());
```

Hands-On: Build and Run the Inference Driver

- Objective
  - Build and run the inference driver and capture throughput
- Do
  - Follow the instructions on the next slide
- Verify
  - Check the terminal output
    - You should see the expected outputs on the next slide
- Time
  - 2 minutes

Run the Inference Driver

- In **`~/RTCSA25-Tutorial`**, run
  - **`make`**
  - **`./run_inference_driver.sh`**
- Expected output

```
[INFO] Top 3 predictions for image index 499:
- Class 867 (trailer_truck): 0.531833
- Class 675 (moving_van): 0.459434
- Class 569 (garbage_truck): 0.00453667

[INFO] Average E2E latency (500 runs): 48.72 ms

[INFO] Average Preprocessing latency (500 runs): 4.934 ms

[INFO] Average Inference latency (500 runs): 43.188 ms

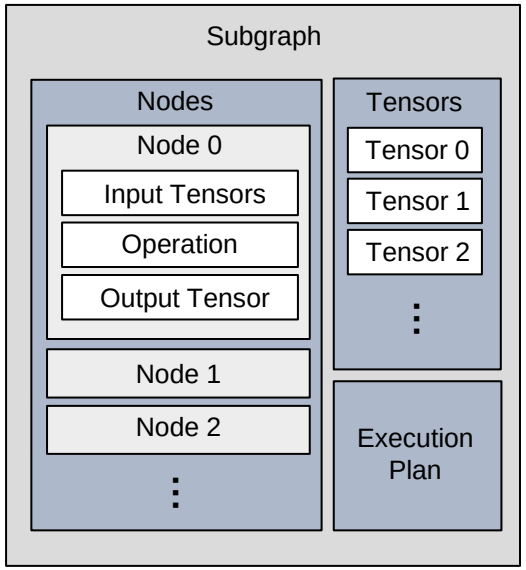
[INFO] Average Postprocessing latency (500 runs): 0 ms

[INFO] Throughput: 20.312 items/sec (500 items in 24616 ms)
```

Contents

Internal Data Structures of LiteRT Model

1. Subgraph
   - An independent computation graph with its own nodes, tensors, and execution plan
     - Multiple subgraphs are used to support advanced model features such as control flow and modular function calls
2. Tensor
   - Multi-dimensional array that holds inputs, outputs, weights and/or intermediate results
3. Node
   - Basic unit of neural operations
4. Execution plan
   - Ordered list of node indices
   - Defines execution sequence



Instrumentation Harness for LiteRT APIs

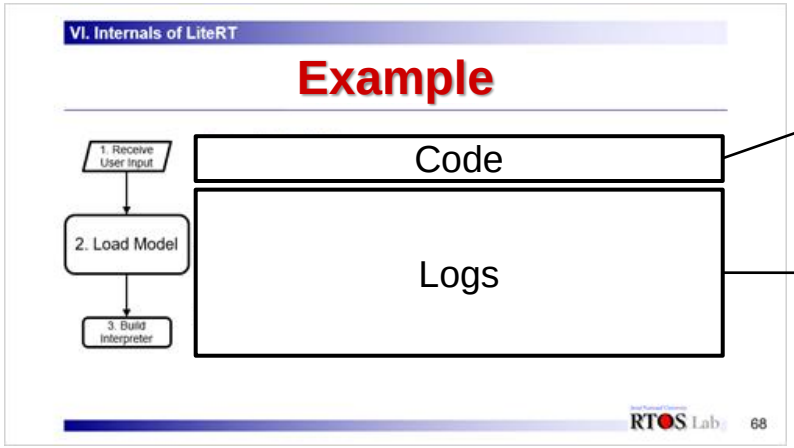
- What is an instrumentation harness?
  - Separate code that replicates the sequence of low-level LiteRT API calls invoked by high-level LiteRT APIs in an inference driver
  - Logs internal changes after each low-level LiteRT API call
- We will follow the code flow of the inference driver and understand the internals of high-level LiteRT APIs

Hands-On: Run the Instrumentation Harness

- Objective
  - Run the instrumentation harness for LiteRT APIs and inspect the logs
- Do
  - Follow the instructions on the next slide
- Verify
  - Check the terminal output
- Time
  - 20 minutes

Run the Instrumentation Harness

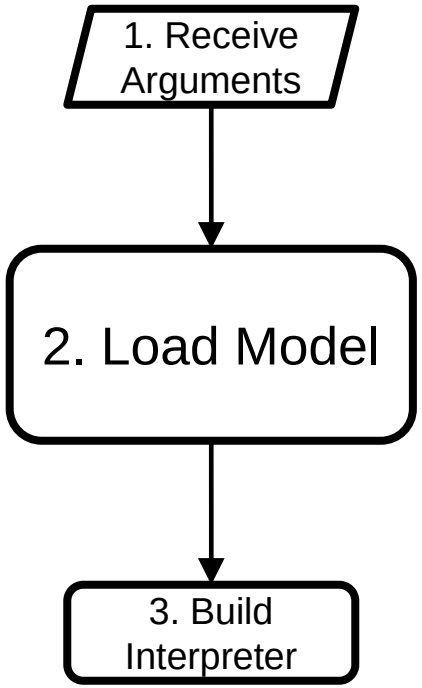
- In **`~/RTCSA25-Tutorial`**, run
  - **`./bin/instrumentation_harness ./models/resnet50.tflite`**
  - You will see an indicator `Press Enter to continue...` at each step
    - Let's go through the steps together to observe what happens internally



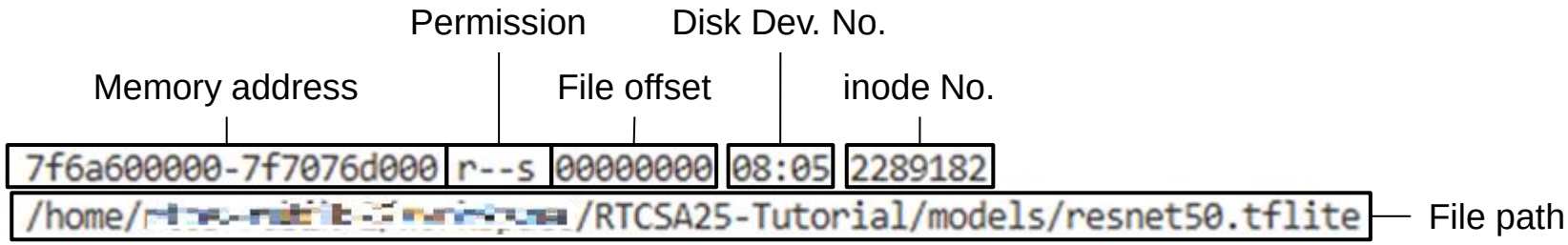
High-level LiteRT APIs invoked by the inference driver

Logs of the instrumentation harness and its explanation

Load Model

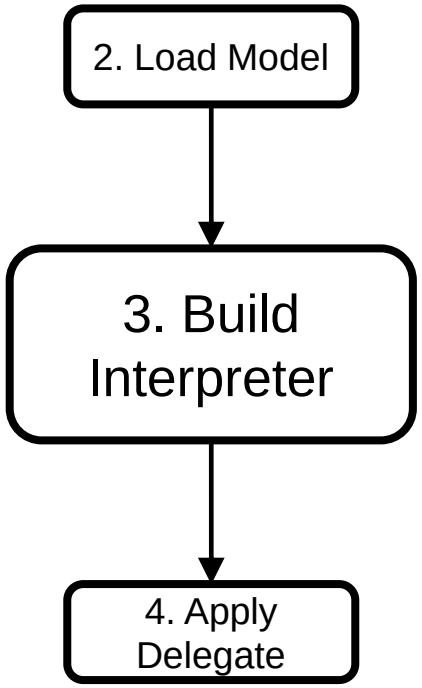



```
58    /* Load model */
59    std::unique_ptr<tflite::FlatBufferModel> _litert_model =
60        tflite::FlatBufferModel::BuildFromFile(model_path.c_str());
```



- The model file is memory-mapped into the process's virtual address space
- You can verify this by looking at **`/proc/<pid>/maps`** of the process

Build Interpreter (1)



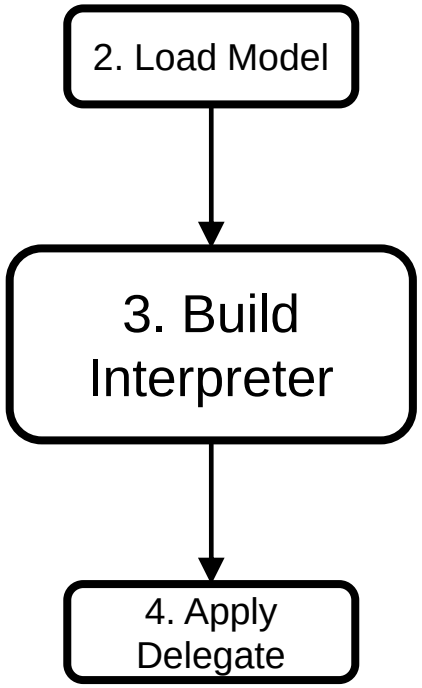

```
67      /* Build interpreter */
68      tflite::ops::builtin::BuiltinOpResolver _litert_resolver;
69      tflite::InterpreterBuilder _litert_builder(*_litert_model, _litert_resolver);
70      std::unique_ptr<tflite::Interpreter>  litert_interpreter;
71      _litert_builder(&_litert_interpreter);
```

```
Schema version of the model: 3
Supported schema version: 3
```

- What is a **schema** in LiteRT?
  - Specification that defines of how a LiteRT model is serialized in FlatBuffer format

- Validates that the model's schema version matches the runtime's supported version
- If the versions do not match, an error is returned
  - Because the runtime cannot properly parse the model file
- The schema definition for LiteRT can be found at
  `~/RTCSA25-Tutorial/external/litert/bazel-litert/external/org_tensorflow/tensorflow/compiler/mlir/lite/schema/schema.fbs`
- You can deserialize a LiteRT model using FlatBuffers, but this is beyond the scope of this tutorial

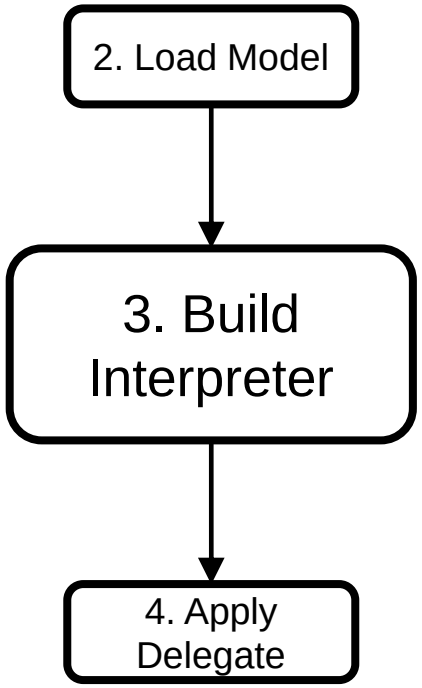
Build Interpreter (2)




```
Total 7 operators in the model
[0] PAD, version: 1, supported (Y/N): Y
[1] CONV_2D, version: 1, supported (Y/N): Y
[2] MAX_POOL_2D, version: 1, supported (Y/N): Y
[3] ADD, version: 1, supported (Y/N): Y
[4] MEAN, version: 1, supported (Y/N): Y
[5] FULLY_CONNECTED, version: 1, supported (Y/N): Y
[6] SOFTMAX, version: 1, supported (Y/N): Y
```

- Parses the operators (name and version) in the model and checks if the OpResolver supports it
- If any operator is not supported, an error is raised
  - Custom operators are handled in a more complicated way

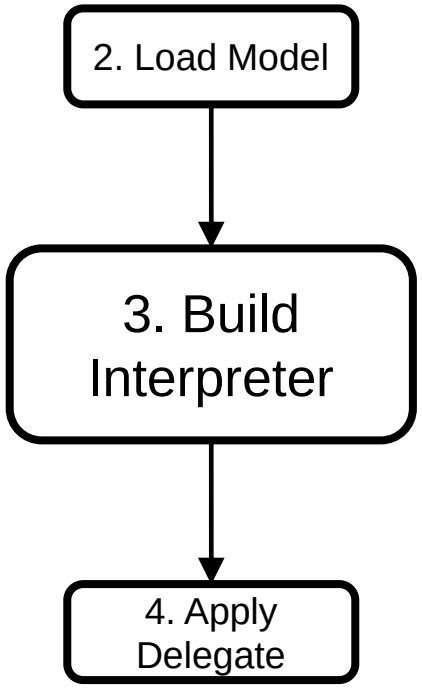
Build Interpreter (3)




```
Number of subgraphs: 1
SubGraph [0] main
Total 187 tensors in SubGraph [0]
Tensor [0] serving_default_input_1:0, type = FLOAT32, shape = [1, 224, 224, 3], buffer = 1 (empty)
Tensor [1] resnet50/conv1_bn/FusedBatchNormV3, type = FLOAT32, shape = [64], buffer = 2 (has data, size = 256)
Tensor [2] resnet50/conv2_block1_0_bn/FusedBatchNormV3, type = FLOAT32, shape = [256], buffer = 3 (has data, size = 1024)
Tensor [3] resnet50/conv2_block1_1_bn/FusedBatchNormV3, type = FLOAT32, shape = [64], buffer = 4 (has data, size = 256)
Tensor [4] resnet50/conv2_block1_2_bn/FusedBatchNormV3, type = FLOAT32, shape = [64], buffer = 5 (has data, size = 256)
Tensor [5] resnet50/conv2_block1_3_bn/FusedBatchNormV3, type = FLOAT32, shape = [256], buffer = 6 (has data, size = 1024)
Tensor [6] resnet50/conv2_block2_1_bn/FusedBatchNormV3, type = FLOAT32, shape = [64], buffer = 7 (has data, size = 256)
Tensor [7] resnet50/conv2_block2_2_bn/FusedBatchNormV3, type = FLOAT32, shape = [64], buffer = 8 (has data, size = 256)
Tensor [8] resnet50/conv2_block2_3_bn/FusedBatchNormV3, type = FLOAT32, shape = [256], buffer = 9 (has data, size = 1024)
                                                                                                    ...
Tensor [184] resnet50/avg_pool/Mean, type = FLOAT32, shape = [1, 2048], buffer = 185 (empty)
Tensor [185] resnet50/predictions/MatMul;resnet50/predictions/BiasAdd, type = FLOAT32, shape = [1, 1000], buffer = 186 (empty)
Tensor [186] StatefulPartitionedCall:0, type = FLOAT32, shape = [1, 1000], buffer = 187 (empty)
```

- Parses the number of subgraphs in the model and creates the corresponding subgraph objects
- For each subgraph, tensor information is parsed and corresponding tensor objects are instantiated

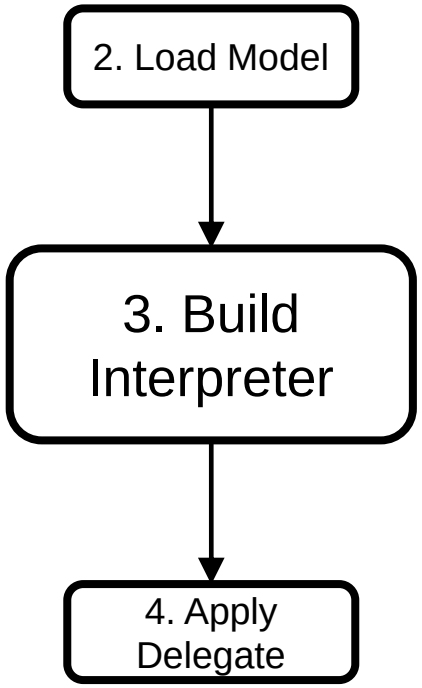
Build Interpreter (4)




```
Total 75 operators in SubGraph [0]
Node [0]: PAD
  Input tensors: 0 110
  Output tensors: 112
Node [1]: CONV_2D
  Input tensors: 112 54 1
  Output tensors: 113
Node [2]: PAD
  Input tensors: 113 108
  Output tensors: 114
Node [3]: MAX_POOL_2D
  Input tensors: 114
  Output tensors: 115
Node [4]: CONV_2D
  Input tensors: 115 55 2
  Output tensors: 116
          ...
Node [73]: FULLY_CONNECTED
  Input tensors: 184 111 107
  Output tensors: 185
Node [74]: SOFTMAX
  Input tensors: 185
  Output tensors: 186
```

- For each subgraph, node information is parsed and corresponding node objects are instantiated
- Execution plan is initialized in this step, matching the execution order to the node indices

Build Interpreter (5)

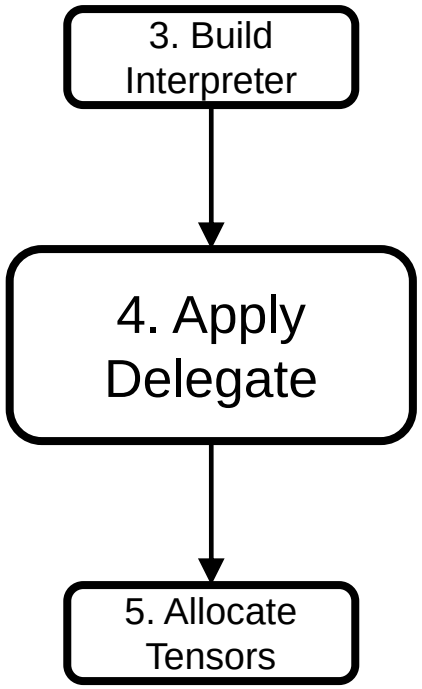



```
Number of subgraphs: 1
Number of nodes of subgraph 0: 75
Number of tensors in subgraph 0: 187
Execution plan size of subgraph 0: 75
```

- Checking whether if the interpreter is successfully created

```
Node 0: PAD
Node 1: CONV_2D
Node 2: PAD
Node 3: MAX_POOL_2D
Node 4: CONV_2D
Node 5: CONV_2D
Node 6: CONV_2D
Node 7: CONV_2D
Node 8: ADD
Node 9: CONV_2D
Node 10: CONV_2D
Node 11: CONV_2D
Node 12: ADD
Node 13: CONV_2D
Node 14: CONV_2D
Node 15: CONV_2D
Node 16: ADD
Node 17: CONV_2D
Node 18: CONV_2D
Node 19: CONV_2D
Node 20: CONV_2D
Node 21: ADD
Node 22: CONV_2D
Node 23: CONV_2D
Node 24: CONV_2D
Node 25: ADD
Node 26: CONV_2D
Node 27: CONV_2D
Node 28: CONV_2D
Node 29: ADD
Node 30: CONV_2D
Node 31: CONV_2D
Node 32: CONV_2D
Node 33: ADD
Node 34: CONV_2D
Node 35: CONV_2D
Node 36: CONV_2D
Node 37: CONV_2D
Node 38: ADD
Node 39: CONV_2D
Node 40: CONV_2D
Node 41: CONV_2D
Node 42: ADD
Node 43: CONV_2D
Node 44: CONV_2D
Node 45: CONV_2D
Node 46: ADD
Node 47: CONV_2D
Node 48: CONV_2D
Node 49: CONV_2D
Node 50: ADD
Node 51: CONV_2D
Node 52: CONV_2D
Node 53: CONV_2D
Node 54: ADD
Node 55: CONV_2D
Node 56: CONV_2D
Node 57: CONV_2D
Node 58: ADD
Node 59: CONV_2D
Node 60: CONV_2D
Node 61: CONV_2D
Node 62: CONV_2D
Node 63: ADD
Node 64: CONV_2D
Node 65: CONV_2D
Node 66: CONV_2D
Node 67: ADD
Node 68: CONV_2D
Node 69: CONV_2D
Node 70: CONV_2D
Node 71: ADD
Node 72: MEAN
Node 73: FULLY_CONNECTED
Node 74: SOFTMAX
```

Apply Delegate




```
90      |       |   if (_litert_interpreter->ModifyGraphWithDelegate(_litert_xnn_delegate) == kTfLiteOk)
```

- The delegate traverses the **execution plan** and replaces supported nodes

```
Number of nodes of subgraph 0: 76
Node 75: DELEGATE

Execution plan size of subgraph 0: 1
Node 75: DELEGATE

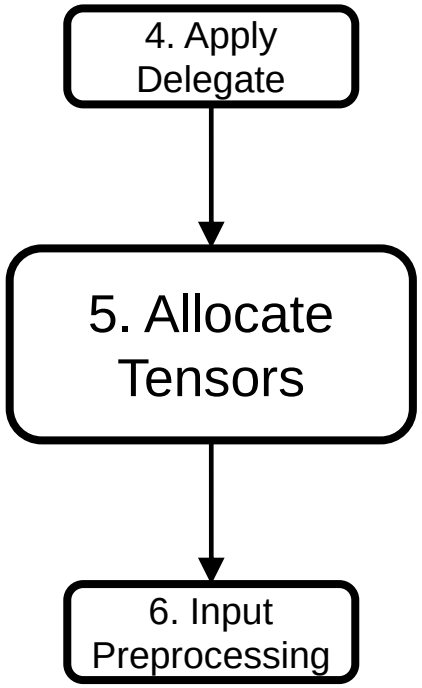
Inputs:
0 (type: FLOAT32, dims: [1, 224, 224, 3])
1 (type: FLOAT32, dims: [64])
2 (type: FLOAT32, dims: [256])
                ...
110 (type: INT32, dims: [4, 2])
111 (type: FLOAT32, dims: [1000, 2048])

Outputs:
186 (type: FLOAT32, dims: [1, 1000])

Total 113 tensors are used.
```

- As a result, a new node is created in the subgraph, which indicates the interpreter to call the delegate
- Execution plan is modified
- Tensors for intermediate results are no longer used
- The delegate manages intermediate results by itself

Allocate Tensors



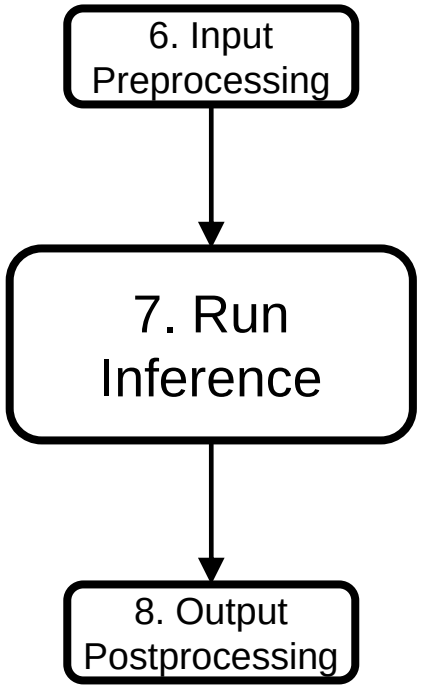

```
81      /* Allocate Tensors */
82      if (_litert_interpreter->AllocateTensors() != kTfLiteOk)
83      {
84          std::cerr << "Failed to Allocate Tensors" << std::endl;
85          return 1;
86      }
```

```
==== Before Allocate Tensors ====
Tensor 0: serving_default_input_1:0 | AllocType: kTfLiteArenaRw | Shape: [1, 224, 224, 3] | Bytes: 602112 | Address: 0
Tensor 1: resnet50/conv1_bn/FusedBatchNormV3 | AllocType: kTfLiteMmapRo | Shape: [64] | Bytes: 256 | Address: 0x7f70764bd0
Tensor 2: resnet50/conv2_block1_0_bn/FusedBatchNormV3 | AllocType: kTfLiteMmapRo | Shape: [256] | Bytes: 1024 | Address: 0x7f707647c4
...
Tensor 111: resnet50/predictions/MatMul | AllocType: kTfLiteMmapRo | Shape: [1000, 2048] | Bytes: 8192000 | Address: 0x7f6a6005c0
Tensor 186: StatefulPartitionedCall:0 | AllocType: kTfLiteArenaRw | Shape: [1, 1000] | Bytes: 4000 | Address: 0
```

- Only mutable tensors are affected

```
==== After Allocate Tensors ====
Tensor 0: serving_default_input_1:0 | AllocType: kTfLiteArenaRw | Shape: [1, 224, 224, 3] | Bytes: 602112 | Address: 0x7f61637040
Tensor 1: resnet50/conv1_bn/FusedBatchNormV3 | AllocType: kTfLiteMmapRo | Shape: [64] | Bytes: 256 | Address: 0x7f70764bd0
Tensor 2: resnet50/conv2_block1_0_bn/FusedBatchNormV3 | AllocType: kTfLiteMmapRo | Shape: [256] | Bytes: 1024 | Address: 0x7f707647c4
...
Tensor 111: resnet50/predictions/MatMul | AllocType: kTfLiteMmapRo | Shape: [1000, 2048] | Bytes: 8192000 | Address: 0x7f6a6005c0
Tensor 186: StatefulPartitionedCall:0 | AllocType: kTfLiteArenaRw | Shape: [1, 1000] | Bytes: 4000 | Address: 0x7f616ca040
```

Run Inference




```
139     /* Inference */
140     if (_litert_interpreter->Invoke() != kTfLiteOk)
141     {
142         std::cerr << "Failed to invoke the interpreter" << std::endl;
143         return 1;
144     }
```

```
0: Node 75 -> TfLiteXNNPackDelegate
```

- The interpreter traverses through the execution plan and executes the nodes in it
- The internal execution logic of each delegate varies

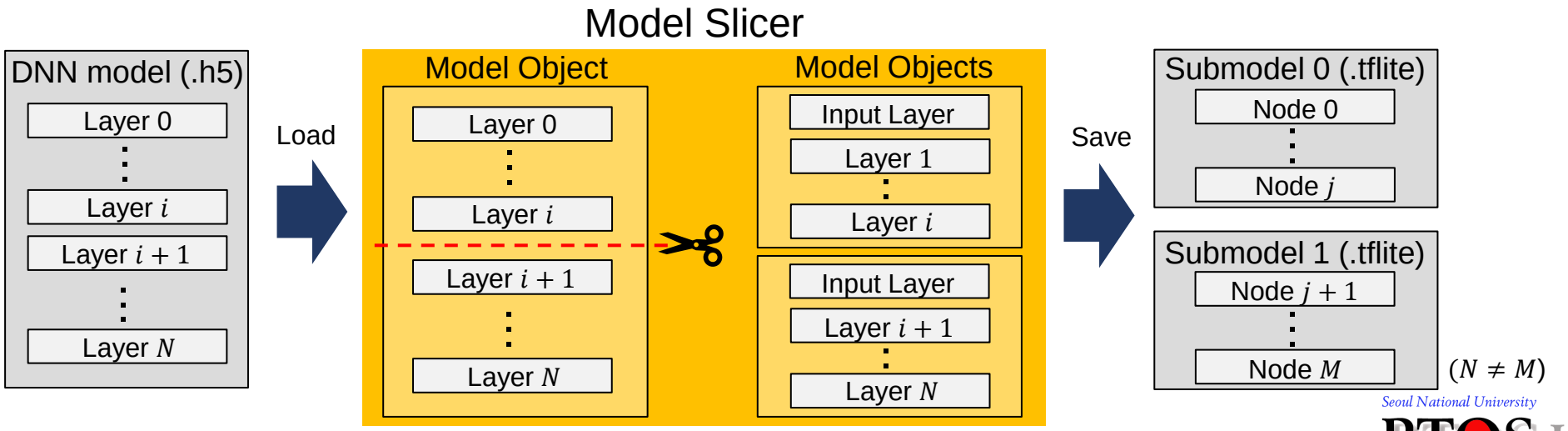
Tutorial Outline

Lecture 1: *From Inference Driver to Inference Runtime* (50 min)	
Lecturer	Seongsoo Hong
Topics	Step-by-Step Inference Driver Walkthrough
	Internals of LiteRT
Brief Break (10 min)	
Lecture 2: *Model Slicer* (50 min)	
Lecturer	Seongsoo Hong
Topic	Model Slicer: Slicing and Conversion Tool for LiteRT
Brief Break (10 min)	
Lecture 3: *Throughput Enhancement on Heterogeneous Accelerators* (1h 30 min)	
Lecturer	Namcheol Lee
Topic	Implementing a Pipelined Inference Driver for Heterogeneous Processors

Contents

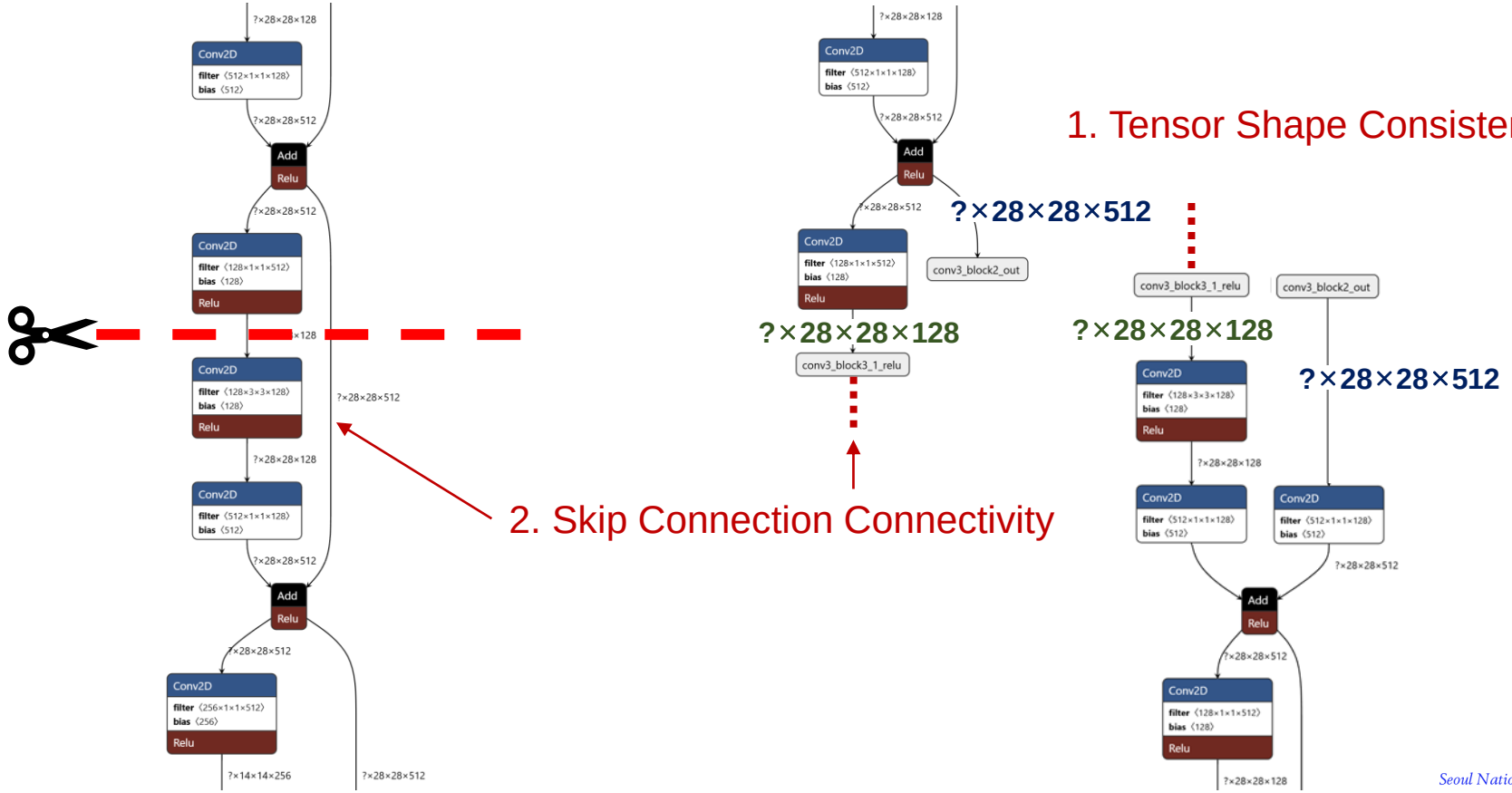
Model Slicing for DNN Pipelining (1)

- Slice a DNN model into multiple submodels
  - Accept a TensorFlow model in HDF5 (Hierarchical Data Format version 5) (.h5)
    - .h5 is a "*serialized storage format on disk*"
  - Load the model in .h5 format to create a "*model object in memory*"
  - Construct submodels from the model object by hand-crafting their model objects
  - Generate multiple LiteRT models in .tflite format



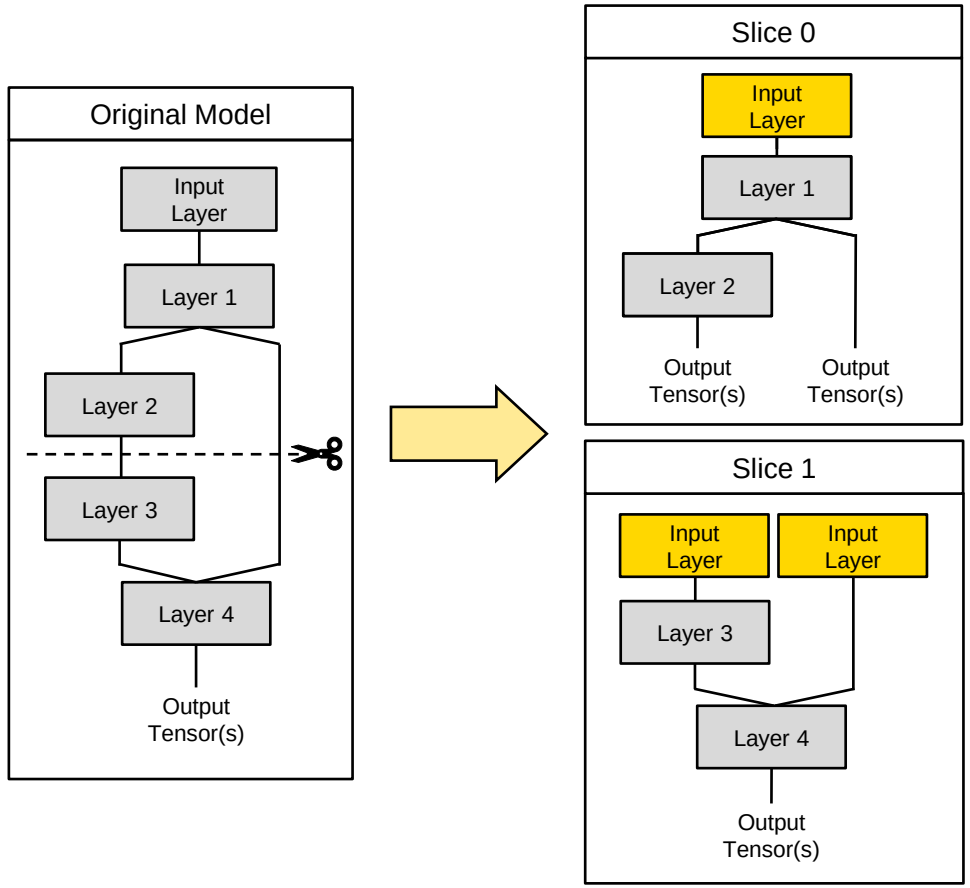
Model Slicing for DNN Pipelining (2)

- Characteristics to be preserved during slicing



Model Slicing for DNN Pipelining (3)

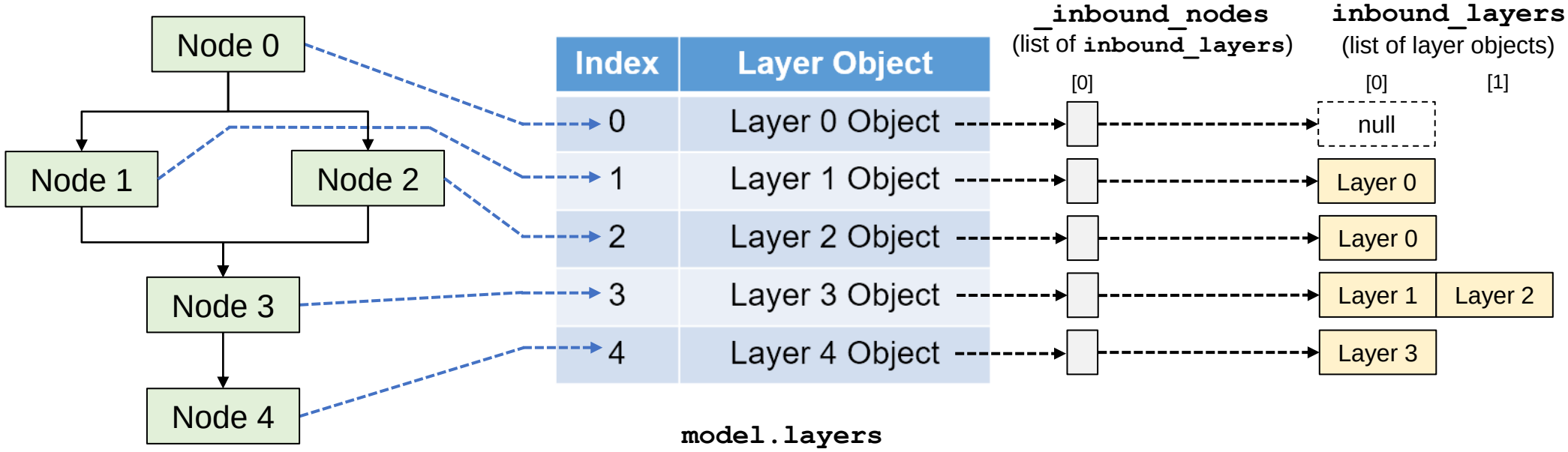
- Input layers are newly created
  - Per output (tensor(s)) of the previous slice



Contents

Internal Representation of Keras Model Object (1)

- Computational graph (DAG) defined by "*layer objects*" and "*node objects*"
  - "**`model.layers`**" works as a hashing table for the DAG



Internal Representation of Keras Model Object (2)

- Important to understand differences between "*layers*" and "*nodes*"
  - Layer objects: similar to *function definitions*
    - Static computational building blocks (units of computation)
    - "**`model.layers`**" works as a hashing table for the DAG
  - Node objects: similar to *function invocations*
    - Instances of layers in DAG
    - Created dynamically when a node needs to appear in the DAG
      - By calling the **`__call__`** method of the layer (functor **`layer()`**)
    - "**`model.layers[idx]._inbound_nodes[0]`**" is the node created first
  - *One-to-one correspondence between layers and nodes in most Keras models*
    - For the sake of simplicity

Internal Representation of Keras Model Object (3)

- Keras tensors
  - A built-in tensor object that has information for graph connections
  - Attributes: **`shape`**, **`dtype`**, and **`name`** properties

```
KerasTensor(type_spec=TensorSpec(shape=(None, 7, 7, 512), dtype=tf.float32, name=None),
            name='conv5_block3_2_conv/BiasAdd:0', description="created by layer 'conv5_block3_2_conv'")
```

`conv5_block3_2_conv` : Layer name that produced this KerasTensor
`BaisAdd` : TensorFlow op
`0` : First output tensor

- Hidden property "**`_keras_history`**"

```
KerasHistory(layer=<keras.layers.convolutional.conv2d.Conv2D object at 0x7f291f63e0>, node_index=1, tensor_index=0)
```

  - **`layer`**: layer whose functor is invoked to generate the tensor(s)
  - **`node_index`**: index of the functor invocation that generate the tensor(s)
  - **`tensor_index`**: index of the tensor in the list of the generated tensors(s)

How Model Slicer Works

- Key Idea
  1. Creates in-memory representation (a.k.a. model object) by loading .h5 file

  For each slice, handles layers one by one

  2. Accepts input tensors and calls **`layer()`** to
     - Create a node object
     - Make inbound connections
     - Execute the layer's operator
     - Create output tensors
     - Make outbound connections
  3. Process inside-ending skips and outside-ending skips

1. Create Model Object (1)

- Load HDF5 file

```
import tensorflow as tf
from tensorflow.keras.models import load_model

model = load_model(args.model_path, compile=False)
```

1. Create Model Object (2)

- "**`model.layers`**" works as a hashing table for the DAG of the model object
  - Correspondence between layers and nodes is represented by
    - **`_inbound_nodes`** (attribute of layer)
      - Each time a layer is connected to a new input, a node is added to **`layer._inbound_nodes`**
    - **`inbound_layers`** (attribute of node)
      - References to the previous layers connected through the inbound nodes
    - **`_outbound_nodes`** (attribute of layer)
      - Each time the output of a layer is used by another layer, a node is added to **`layer._outbound_nodes`**
    - **`outbound_layer`** (attribute of node)
      - References to the next layers connected through the outbound nodes

2. Call the `layer()` Functor (1)

- Understanding the **`layer()`** functor
  - **`tensors_from_current_layer = layer(tensors_to_current_layer)`**
    - Input parameter: Keras tensors that are inputs to the layer
    - Return value: Keras tensors that are outputs from the layer
  - The **`__call__`** method of the **`layer`** object is invoked

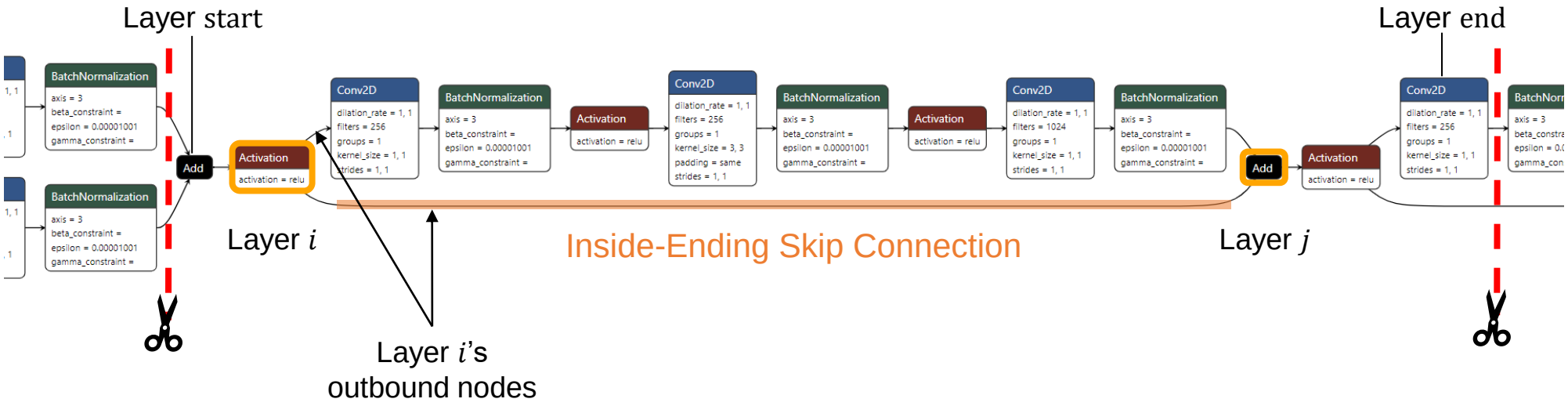
2. Call the `layer()` Functor (2)

- In **`layer()`**
  1. Reads input tensor's Keras history to identify its origin layer
  2. Creates a new node
     - Set **`Node.outbound_layer`** = current layer
     - Set **`Node.inbound_layers`** = origin layers of input tensors
  3. Record **`Node.input_tensors`** and **`Node.output_tensors`**
  4. Append the node to
     - **`inbound_nodes`** list of the current layer
     - **`outbound_nodes`** list of each input tensor's origin layer

Contents

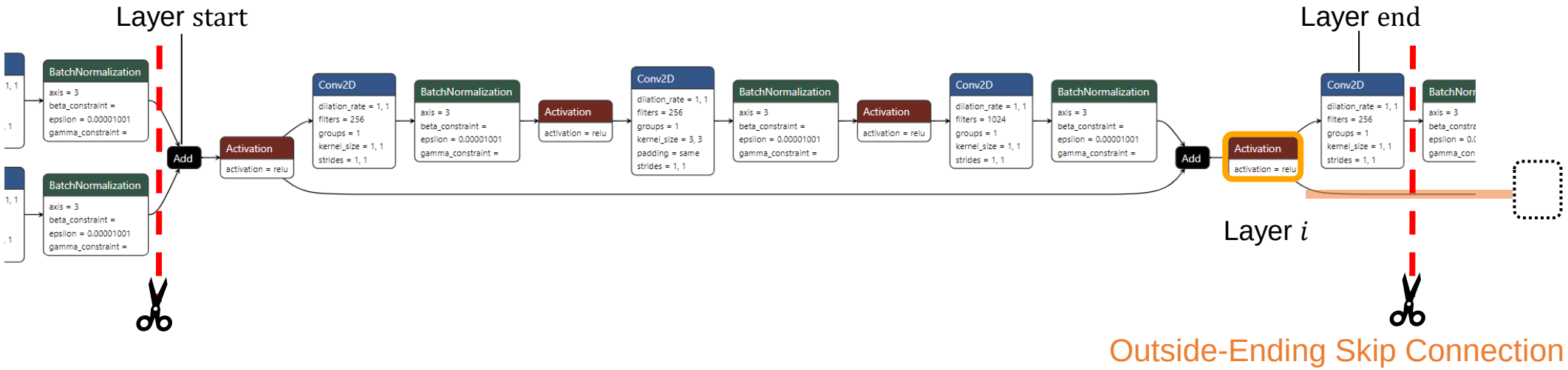
Inside- vs. Outside-Ending Skip Connections (1)

- Inside-ending skip connection of a slice with range of $[\text{start}, \text{end}]$ is
  - Among the *outbound nodes of newly created input layers*,
    a node corresponding to a layer with index j such that $j > \text{start}$ and $j \leq \text{end}$
  - And among the *outbound nodes of layer* i where $i \in [\text{start}, \text{end})$,
    a node corresponding to a layer with index j such that $j > i + 1$ and $j \leq \text{end}$



Inside- vs. Outside-Ending Skip Connections (2)

- Outside-ending skip connection of a slice with range of $[\text{start}, \text{end}]$ is
  - Among the *outbound nodes of newly created input layers*, a node corresponding to a layer with index j such that $j > \text{end}$
  - And among the *outbound nodes of layer* i where $i \in [\text{start}, \text{end})$, a node corresponding to a layer with index j such that $j > \text{end}$



Constructing Algorithm by Induction

- *A slice's output tensors are fed to the next slice as input*

1. We correctly hand-craft the input tensors to the first slice
2. We generate a slice in the middle correctly
   - Its internal structure matches the corresponding part of the model
   - It produces the correct number and shape of output tensors
3. We repeatedly generate slices from the first to the last

Cases for Model Slicer

- For a given slice, perform

1. Input layer generation
2. Input tensor initialization of start layer
3. Inside-ending skip connection initialization
4. Outside-ending skip connection initialization
5. Per-layer processing
6. Inside-ending skip connection update
7. Outside-ending skip connection update

1. Input Layer Generation

- Create input layers to the slice from the provided input tensors
  - ```
    input_layers = {}      # dictionary {layer name, KerasTensor}
    input_layers[name] =
        tf.keras.layers.Input(shape=tensor.shape[1:], name=name)
    ```
    - **`name`**: Name of the layer that generated the tensor
    - **`tensor.shape[1:]`**: Height, width, and channel of **`tensor`** as a list
    ```

2. Input Tensor Initialization of Start Layer

- Identify the outbound nodes of the input layers
- Among those, select the nodes that target the start layer
- Assemble their Keras tensors into the start layer's input
  - `tensors_to_start_layer = []`
  - `tensors_to_start_layer.append(input_layers[name])`

3. Inside-Ending Skip Connection Initialization

- Identify the outbound nodes of the input layers
- Among those, select the nodes that target the layers in the same slice, other than the start layer
- Obtain Keras tensors and register them as inside-ending skip connections
  - `inside_ending_skips = {}`
    `# dictionary {layer name, KerasTensor}`
  - `inside_ending_skips[name] = input_layers[name]`

4. Outside-Ending Skip Connection Initialization

- Identify the outbound nodes of the input layers
- Among those, select the nodes that target the layers outside the current slice
- Obtain Keras tensors and register them as outside-ending skip connections
  - `outside_ending_skips = {}`
    `# dictionary {layer name, KerasTensor}`
  - `outside_ending_skips[name] = input_layers[name]`

5. Per-Layer Processing

- Layer i ($i \in [\text{start}, \text{end}]$)
  - Call the layer's functor with
    - $i = \text{start}$: Tensor(s) initialized in Case 2
    - $i > \text{start}$: Tensor(s) from layer $i - 1$ and any required inside-ending skip tensors
      - Identify required tensors by inspecting the inbound nodes of layer i in the model

6. Inside-Ending Skip Connection Update

- After processing layer i ($i \in [\text{start}, \text{end})$), find its outbound nodes
- If an outbound node targets a layer in this slice ($\text{target} \in (i+1, \text{end}]$), register the output tensor as an inside-ending skip connection
  - `inside_ending_skips[layer.name] = tensors_from_current_layer`

7. Outside-Ending Skip Connection Update

- After processing layer i ($i \in [\text{start}, \text{end})$), find its outbound nodes
- If an outbound node targets a layer outside the current slice ($\text{target} > \text{end}$), register the output tensor as an outside-ending skip connection
  - `outside_ending_skips[layer.name] = tensors_from_current_layer`

Code for Cases 1, 2, 3, and 4

```
51      # Case 1, 2, 3, and 4
52      for name, tensor in input_tensors.items():
53          # Input layers are created
54          input_layers[name] = tf.keras.layers.Input(shape=tensor.shape[1:], name=name)
55
56          # Inspect an input layer's outbound nodes to see where the model consumes it
57          origin_layer = model.get_layer(name)
58          if len(origin_layer._outbound_nodes) > 1: # If an input layer has multiple outbound layers, its output feeds multiple layers
59              for origin_outbound_node in origin_layer._outbound_nodes:
60                  origin_outbound_layer = origin_outbound_node.outbound_layer
61                  target_idx = model.layers.index(origin_outbound_layer)
62                  if target_idx == start:
63                      tensors_to_start_layer.append(input_layers[name])
64                  elif target_idx <= end and target_idx > start:
65                      inside_ending_skips[name] = input_layers[name]
66                  elif target_idx > end:
67                      # Any outside-ending skip connections that are made here are not used in this slice
68                      outside_ending_skips[name] = input_layers[name]
69          else:
70              origin_outbound_layer = origin_layer._outbound_nodes[0].outbound_layer
71              target_idx = model.layers.index(origin_outbound_layer)
72              if target_idx == start:
73                  tensors_to_start_layer.append(input_layers[name])
74              elif target_idx <= end and target_idx > start:
75                  inside_ending_skips[name] = input_layers[name]
76              elif target_idx > end:
77                  # Any outside-ending skip connections that are made here are not used in this slice
78                  outside_ending_skips[name] = input_layers[name]
```

Code for Cases 5, 6, and 7 (1)

```
81      # Case 5, 6, and 7
82      # # Set the start layer's inputs from tensors_to_start_layer
83      if(len(tensors_to_start_layer) == 1):
84          tensors_to_current_layer = tensors_to_start_layer[0]
85      else:
86          tensors_to_current_layer = tensors_to_start_layer
87
88      for i in range(start, end+1):
89          layer = model.layers[i]
90          origin_inbound_layers = layer._inbound_nodes[0].inbound_layers
91
92          # Build current (i-th) layer
93          if isinstance(origin_inbound_layers, list): # When current layer expects multiple inputs (list of KerasTensors)
94              if(i == start):
95                  tensors_from_current_layer = layer(tensors_to_current_layer)
96              else:
97                  # NOTE: The model slicer assumes that the output of layer i is always used by layer i+1
98                  tensors_to_current_layer = [tensors_to_current_layer] if not isinstance(tensors_to_current_layer, list) else tensors_to_current_layer
99
100                 # From inbound layers, collect the required inside-ending skip tensors
101                 for origin_inbound_layer in origin_inbound_layers:
102                     inside_ending_skip = model.get_layer(origin_inbound_layer.name)
103                     if inside_ending_skip.name not in [t.name.split('/')[0] for t in tensors_to_current_layer]:
104                         tensors_to_current_layer.append(inside_ending_skips[inside_ending_skip.name])
```

Code for Cases 5, 6, and 7 (2)

```
106                 # Call the functor of the current layer to build a new layer
107                 try:
108                     tensors_from_current_layer = \
109                         layer(tensors_to_current_layer)
110                 except: # When a custom layer's call signature deviates from Keras expectations
111                     raise ValueError(f"Failed to call layer {layer.name} with tensors {tensors_to_current_layer}. "
112                                      "Please check the layer's call function and the input tensors.")
113             else: # When current layer expects a single input (KerasTensor)
114                 if origin_inbound_layers.name in inside_ending_skips:
115                     tensors_from_current_layer = layer(inside_ending_skips[origin_inbound_layers.name])
116                 else:
117                     tensors_from_current_layer = layer(tensors_to_current_layer)
118 
119             # Update inside_ending_skips and outside_ending_skips based on the current layer's outbound nodes
120             if i < end: # Ensure the current layer is not the slice's last layer
121                 for origin_outbound_node in layer._outbound_nodes:
122                     skip_target_idx = model.layers.index(origin_outbound_node.outbound_layer)
123                     if skip_target_idx <= end and skip_target_idx > i + 1:
124                         inside_ending_skips[layer.name] = tensors_from_current_layer
125                     elif skip_target_idx > end:
126                         outside_ending_skips[layer.name] = tensors_from_current_layer
127 
128             tensors_to_current_layer = tensors_from_current_layer # End of for i in range(start, end+1)
```

Contents

The Model Slicer

- A Python script tool
  - Developed at RTOSLab. of Seoul National University
- Derived from the original implementation of **`DNNPipe`**

DNNPipe: Dynamic Programming-based Optimal DNN Partitioning for Pipelined Inference on IoT Networks

Woobean Seo[1], Saehwa Kim[2]*, and Seongsoo Hong[1]

[1]Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, Korea
[2]Department of Information Communications Engineering, Hankuk-University of Foreign Studies, Yongin 17035, Korea

ABSTRACT– Pipeline parallelization is an effective technique that enables the efficient execution of deep neural network (DNN) inference on resource-constrained IoT devices. To enable pipeline parallelization across computing nodes with asymmetric performance profiles, interconnected via low-latency, high-bandwidth networks, we propose DNNPipe, a DNN partitioning algorithms that constructs a pipeline plan for a given DNN. The primary objective of DNNPipe is to maximize the throughput of DNN inference while minimizing the runtime overhead of DNN partitioning, which is repeatedly executed online in dynamically changing IoT environments. To achieve this, DNNPipe uses dynamic programming (DP) with pruning techniques that preserve optimality to explore the search space and find the optimal pipeline plan whose maximum stage time is no greater than that of any other possible pipeline plan. Specifically, it aggressively prunes suboptimal pipeline plans using two pruning techniques: *upper-bound-based pruning* and *under-utilized-stage pruning*. Our experimental results demonstrate that pipelined inference using an obtained optimal pipeline plan improves DNN throughput by up to 1.78 times compared to the highest performing single device and DNNPipe achieves up to 98.26% lower runtime overhead compared to PipeEdge, the fastest known optimal DNN partitioning algorithm.

KEYWORDS: IoT systems, embedded AI, DNN partitioning, pipeline parallelization, inference throughput optimization.

1. INTRODUCTION

The proliferation of Internet of Things (IoT) technology and the adoption of AI in various IoT applications [2, 3] has led to an increased demand for real-time processing of streaming data on deep neural networks (DNNs). However, the limited memory size of IoT devices often poses a challenge to the feasibility of executing even simple AI models. Even when a model can be loaded, the low processing power of such devices results in limited throughput, which cause the throughput of DNN inference to fall below the input stream's data rate. In such cases, unprocessed input data accumulates in a queue, leading to increasing queueing delays and eventually unbounded end-to-end (E2E) latency. A common workaround is to drop excess input data to avoid system overload, but this leads to data loss, which is unacceptable in many applications that require full stream integrity.

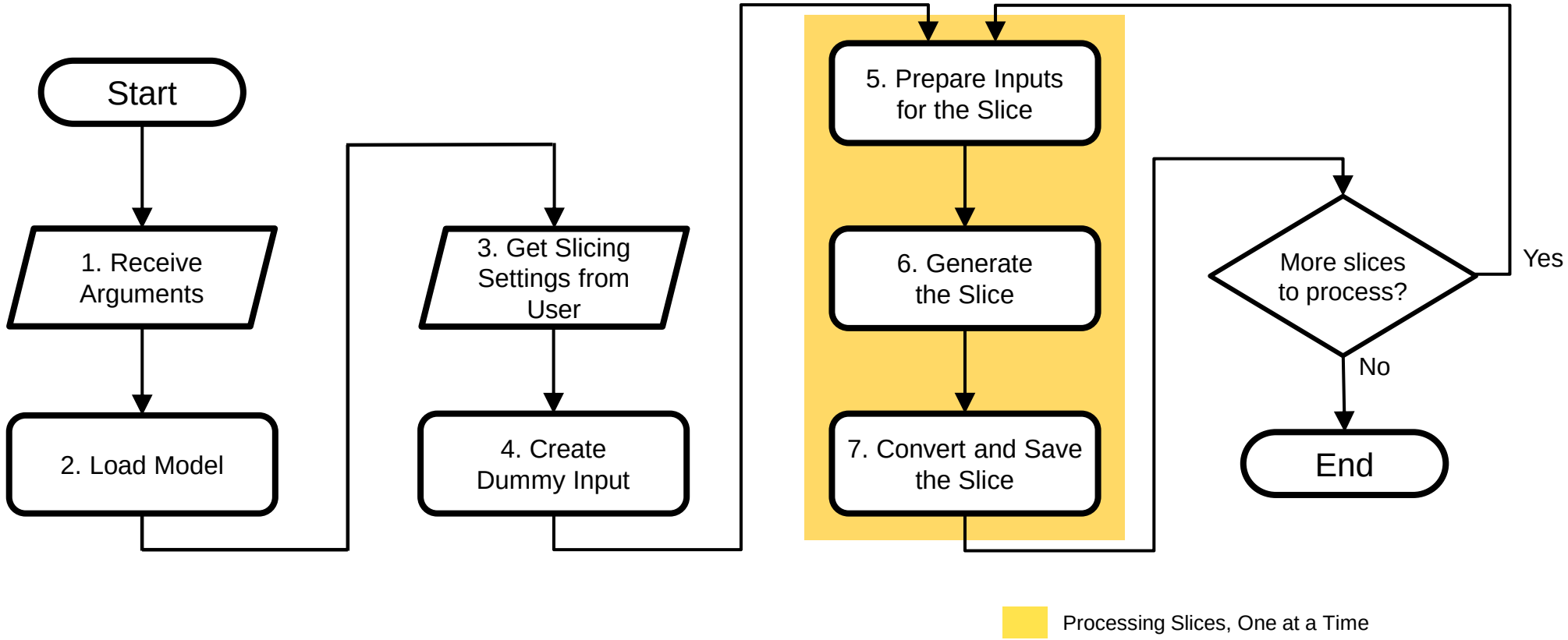
To address these challenges, pipeline parallelization has emerged as a promising solution. It not only reduces per-device memory requirements, but also enables the efficient execution of DNN inference on resource-constrained IoT devices.

Pipeline parallelization relies on DNN partitioning, which divides a DNN model into multiple stages, each consisting of a non-empty, contiguous subset of layers, and distributes their workload across computing nodes with asymmetric performance profiles. It can accelerate DNN inference through parallel and pipelined execution.

* Corresponding author.
E-mail address: ksaehwa@hufs.ac.kr (S. Kim).
This paper is an extended version of the paper that appeared in the Ph.D. Symposium at the 2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS 2024) [1].

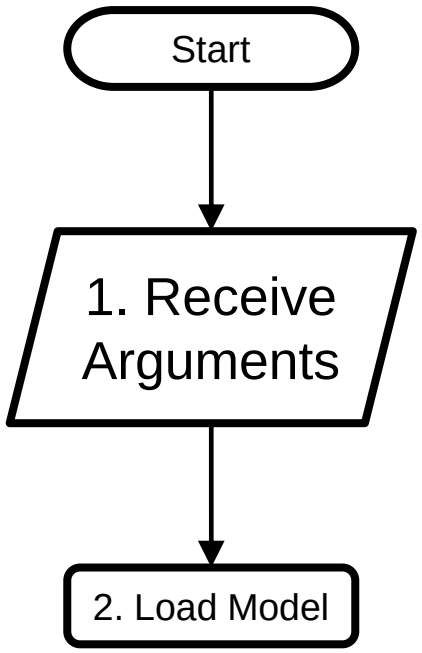
Woobean Seo, Saehwa Kim, and Seongsoo Hong, https://doi.org/10.1016/j.sysarc.2025.103462."DNNPipe: Dynamic Programming–based Optimal DNN Partitioning for Pipelined Inference on IoT Networks," Journal of Systems Architecture, vol. 166, 2025, 103462, ISSN 1383–7621,

Model Slicer Code Flow



1. Receive Arguments

- Parse command-line arguments for the model path and output directory

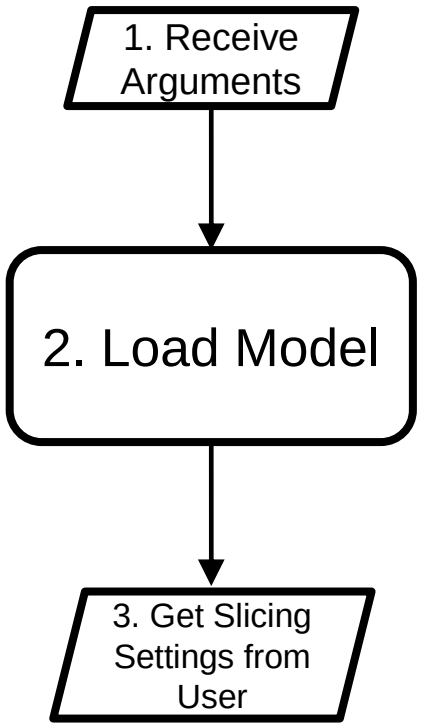



```
207  def main():
208      # Parse command line arguments
209      args = parse_arguments()
```

```
162  # Parse command line arguments
163  def parse_arguments():
164      parser = argparse.ArgumentParser()
165      parser.add_argument('--model-path', type=str, required=True, help='Path to a model file (.h5)')
166      parser.add_argument('--output-dir', type=str, default='./models')
167      return parser.parse_args()
```

2. Load Model

- Load .h5 model file into memory as Keras model object
  - Slicing is performed on model object, not on .h5 model file

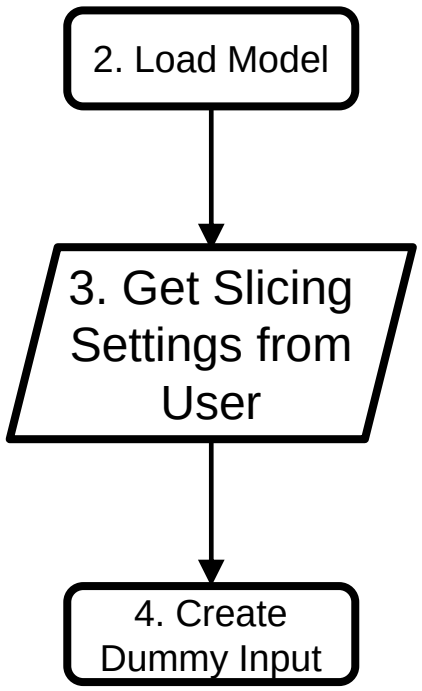



```
17  import tensorflow as tf
18  import numpy as np
19  import os
20  from tensorflow.keras.models import load_model
21  import argparse
```

```
208     # Load the model from the given path without compilation (for inference/slicing only)
209     model = load_model(args.model_path, compile=False)
```

3. Get Slicing Settings from User

- Get the number of submodels
  and the index of the first layer in each submodel

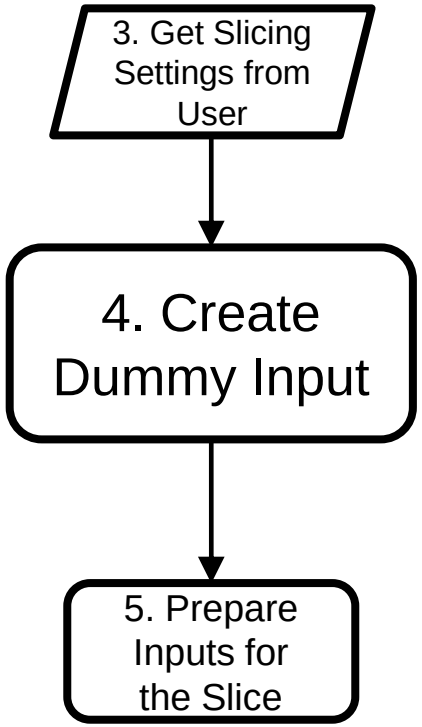



```
211     # Ask the user for the number of slices and the index of the last layer in each slice
212     num_layers = len(model.layers)
213     num_slices, starts = get_slice_starts(num_layers)
```

```
(.venv) rubikpi@RUBIKPi:~/RTCSA25-Tutorial$ python model_slicer.py --model-path ./models/resnet50.h5
How many submodels? 2
Generated submodels look like: (1, x1-1), (x1, 176)
Enter x1: 23
Layer index ranges for each submodel: [(1, 22), (23, 176)]
Saved LiteRT model to: ./models/submodel_0.tflite
Saved LiteRT model to: ./models/submodel_1.tflite
```

4. Create Dummy Input

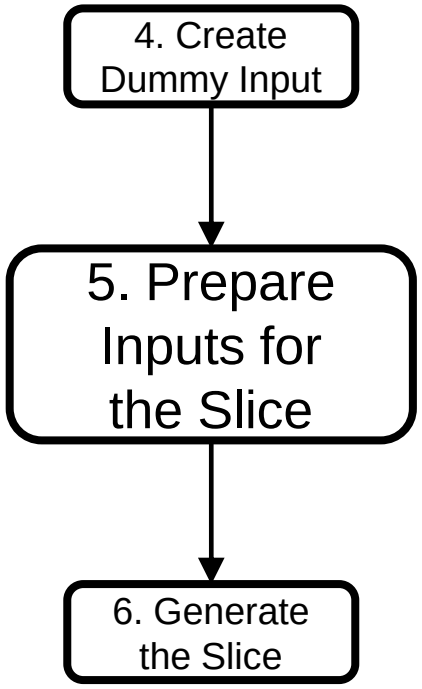
- Create a dummy input tensor that has the same shape as the original model's input tensor




```
215    # Create a dummy input tensor for the first slice
216    input_shape = model.layers[0].input_shape[0][1:]
217    dummy_input = np.random.rand(1, *input_shape)
```

5. Prepare Inputs for the Slice

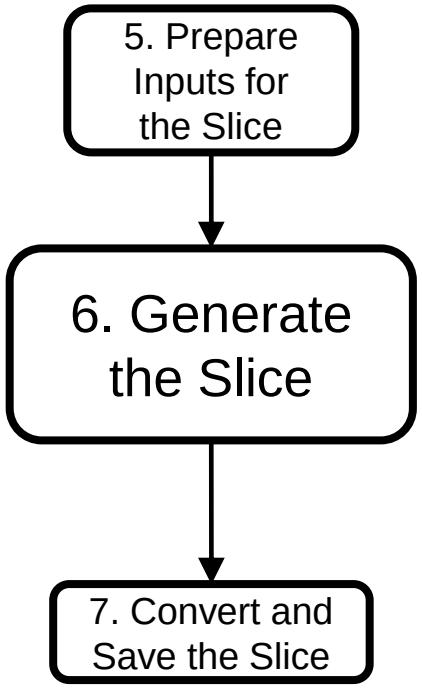
- Dictionary for inputs to the slice is created
  - {input layer name: input tensor}




```
219     # Perform slicing and conversion
220     slices = []
221     for i in range(num_slices):
222         # Prepare inputs for each slice
223         if i == 0:
224             slice_inputs = {model.layers[0].name: dummy_input}
225         else:
226             slice_inputs = prepare_slice_inputs(slices[i-1])
227
228         # Slice the model using slice_dnn
229         slice = slice_dnn(model, starts[i], starts[i+1]-1, slice_inputs)
230         slices.append(slice)
231
232         # Convert and save the slice to a LiteRT model
233         convert_save_slice(args.output_dir, slice, i)
```

6. Generate the Slice

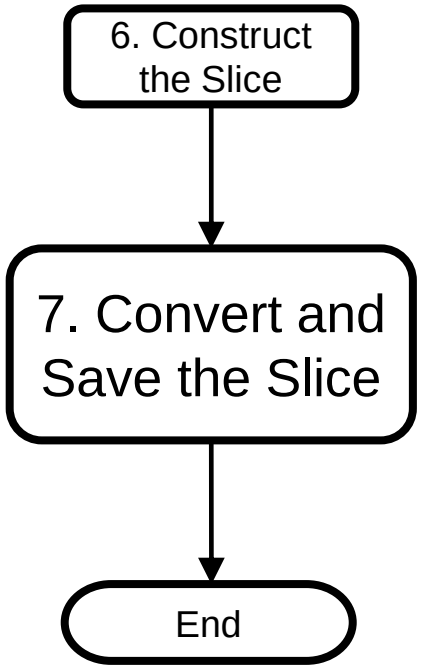
- For the given model, create a slice, one at a time
  - According to user-provided slicing settings




```
219     # Perform slicing and conversion
220     slices = []
221     for i in range(num_slices):
222         # Prepare inputs for each slice
223         if i == 0:
224             slice_inputs = {model.layers[0].name: dummy_input}
225         else:
226             slice_inputs = prepare_slice_inputs(slices[i-1])
227
228         # Slice the model using slice_dnn
229         slice = slice_dnn(model, starts[i], starts[i+1]-1, slice_inputs)
230         slices.append(slice)
231
232         # Convert and save the slice to a LiteRT model
233         convert_save_slice(args.output_dir, slice, i)
```

7. Convert and Save the Slice

- Convert and save the slice into a LiteRT model




```
219     # Perform slicing and conversion
220     slices = []
221     for i in range(num_slices):
222         # Prepare inputs for each slice
223         if i == 0:
224             slice_inputs = {model.layers[0].name: dummy_input}
225         else:
226             slice_inputs = prepare_slice_inputs(slices[i-1])
227
228         # Slice the model using slice_dnn
229         slice = slice_dnn(model, starts[i], starts[i+1]-1, slice_inputs)
230         slices.append(slice)
231
232         # Convert and save the slice to a LiteRT model
233         convert_save_slice(args.output_dir, slice, i)
```

Contents

Hands-On: Run Model Slicer

- Objective
  - Run model slicer python script and see the outputs
- Do
  - Slice ResNet50 into two submodels
- Verify
  - Observe submodel structure using Netron
- Time
  - 5 minutes

Slicing ResNet50 (1)

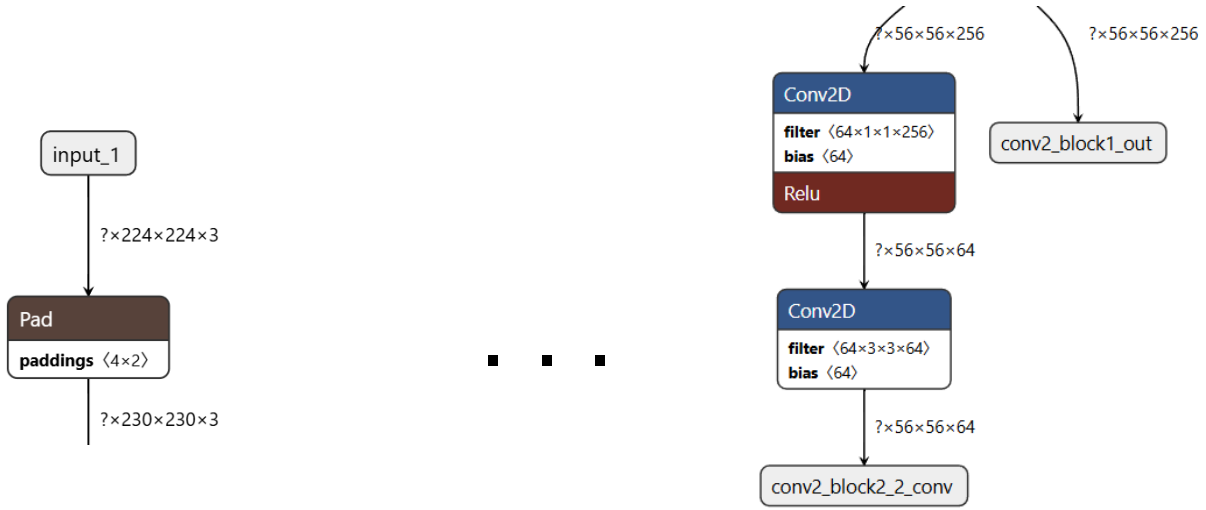
- Command to slice the target DNN model (ResNet50)
  - **`$ python model_slicer.py --model-path ./models/resnet50.h5`**

- For lecture 3, slice ResNet50 into two submodels
  - **`submodel_0`**: From layer 1 to layer 22
  - **`submodel_1`**: From layer 23 to layer 176

```
(.venv) rubikpi@RUBIKPi:~/RTCSA25-Tutorial$ python model_slicer.py --model-path ./models/resnet50.h5
How many submodels? 2
Generated submodels look like: (1, x1-1), (x1, 176)
Enter x1: 23
Layer index ranges for each submodel: [(1, 22), (23, 176)]
Saved LiteRT model to: ./models/submodel_0.tflite
Saved LiteRT model to: ./models/submodel_1.tflite
```

Slicing ResNet50 (2)

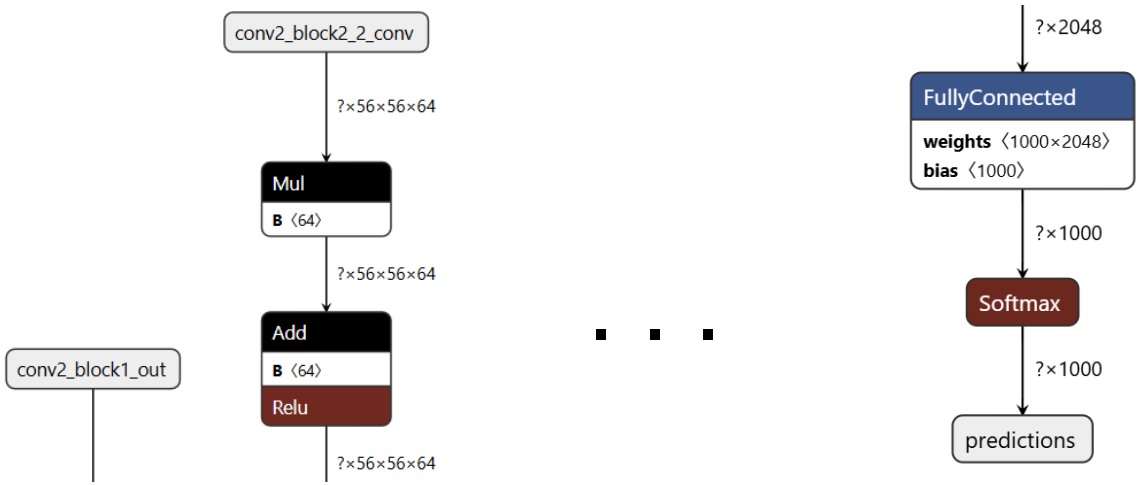
- `netron ./models/submodel_0.tflite`



submodel	Input	Output	Saved As
0	input_1	**conv2_block2_2_conv, conv2_block1_out**	submodel_0.tflite
1	**conv2_block2_2_conv, conv2_block1_out**	Predictions (final output)	submodel_1.tflite

Slicing ResNet50 (3)

- `netron ./models/submodel_1.tflite`



submodel	Input	Output	Saved As
0	input_1	**conv2_block2_2_conv, conv2_block1_out**	submodel_0.tflite
1	**conv2_block2_2_conv, conv2_block1_out**	Predictions (final output)	submodel_1.tflite

Tutorial Outline

Lecture 1: *From Inference Driver to Inference Runtime* (50 min)	
Lecturer	Seongsoo Hong
Topics	Step-by-Step Inference Driver Walkthrough
	Internals of LiteRT
Brief Break (10 min)	
Lecture 2: *Model Slicer* (50 min)	
Lecturer	Seongsoo Hong
Topic	Model Slicer: Slicing and Conversion Tool for LiteRT
Brief Break (10 min)	
Lecture 3: *Throughput Enhancement on Heterogeneous Accelerators* (1h 30 min)	
Lecturer	Namcheol Lee
Topic	Implementing a Pipelined Inference Driver for Heterogeneous Processors

Contents

Goal

- In this lecture, we will
  - Implement a pipelined inference driver
  - Measure throughput and compare it with the throughput of the inference driver

```
[INFO] Throughput: 20.312 items/sec (500 items in 24616 ms)
```

```
[INFO] Throughput: [      ?      ] (500 items in [   ?   ])
```

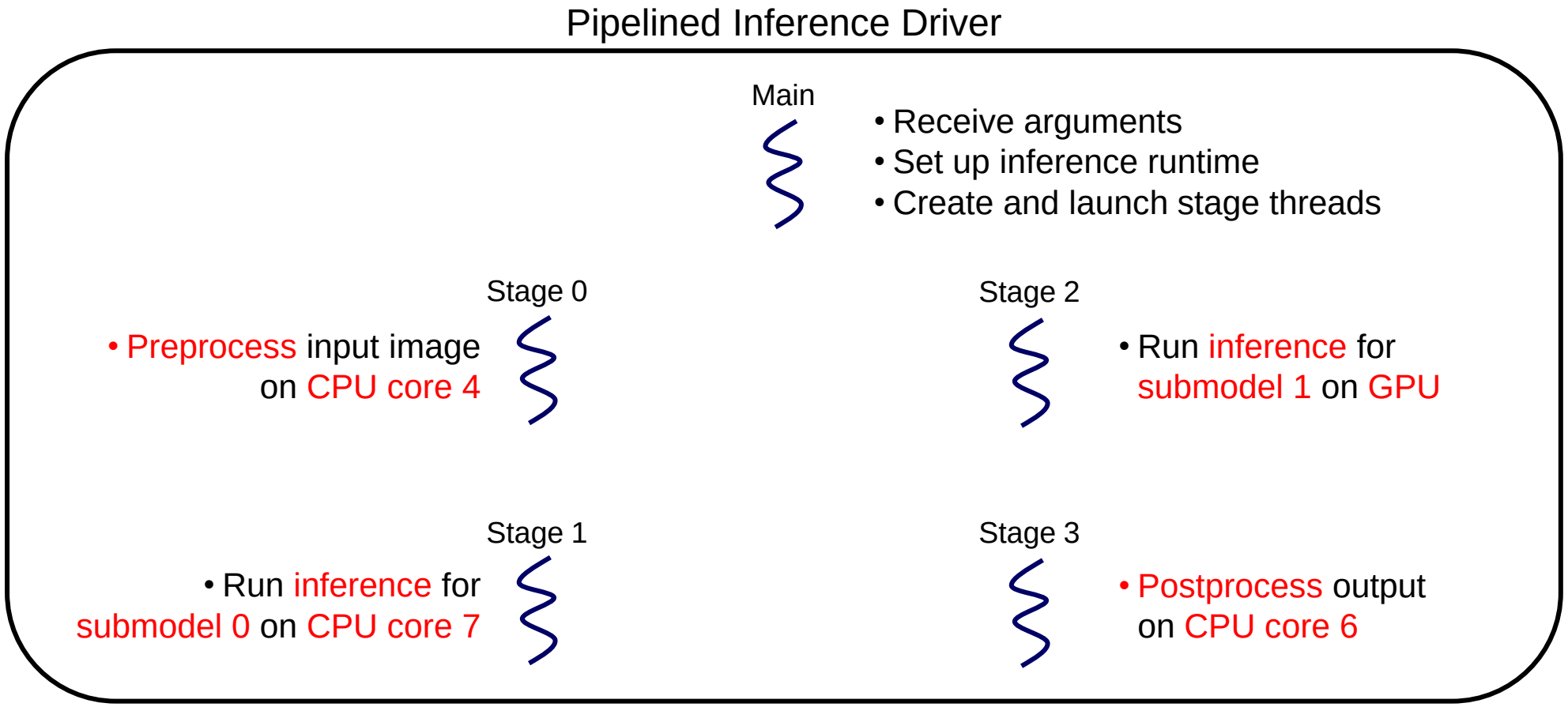
Materials

- Skeleton code
  - **`src/pipelined_inference_driver.cpp`**
- Submodels
  - **`models/submodel_0.tflite`**
    - Layers from 1 to 22 of **`models/resnet50.h5`**
  - **`models/submodel_1.tflite`**
    - Layers from 23 to 176 of **`models/resnet50.h5`**

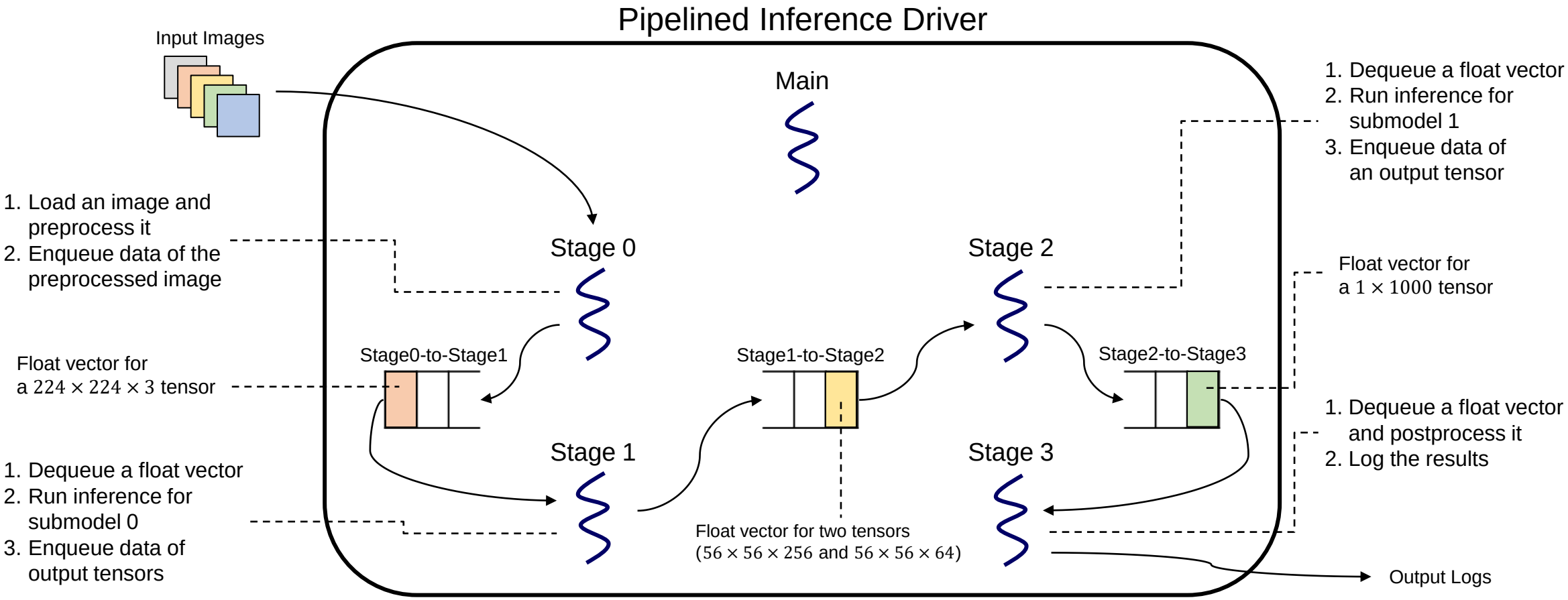
Multithreaded Structure (1)

- Pipelined inference driver with two submodels consists of
  - One main thread
  - Four worker threads
    - One for preprocessing, another for postprocessing
    - Two threads for two submodels, one for each
  - Equivalently, four stages
    - A worker thread becomes a *stage* when it is allocated onto a dedicated accelerator

Multithreaded Structure (2)



Multithreaded Structure (3): Operational Flow



Data Structure (1)

`InterStageQueue`

- Class implementing a thread-safe queue for transferring **`StageOutput`** variables between stages

```
template <typename T>
class InterStageQueue
{
public:
    void push(T item) // Push an item to the queue
    {…
    }

    bool pop(T &item) // Pop an item from the queue
    {…
    }

    void signal_shutdown() // Signal that no more items will be pushed
    {…
    }

    size_t size() // Get the number of items in the queue
    {…
    }
```

Data Structure (2)

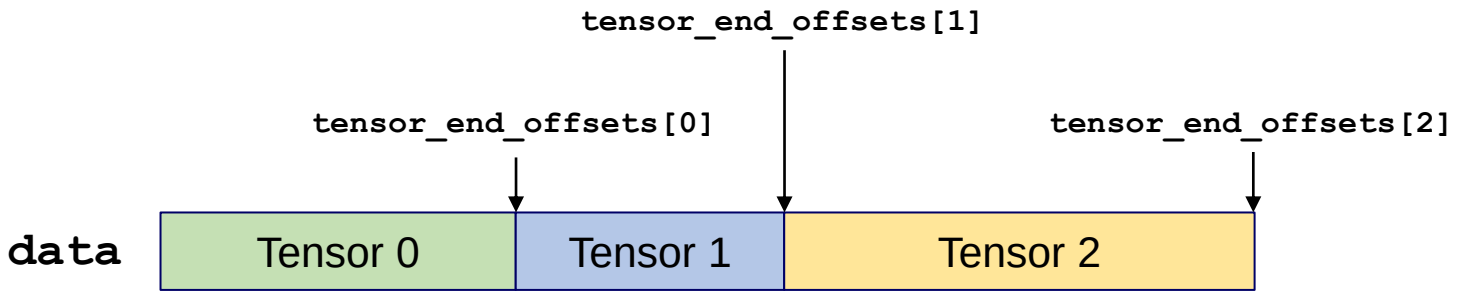
- **`InterStageQueue`** (cont'd)
  - In the pipelined inference driver
    - Three instances of **`InterStageQueue`** are defined as global variables

```
34  // === Queues for inter-stage communication ===
35  // stageX_to_stageY_queue: from stageX to stageY
36  InterStageQueue<IntermediateTensor> stage0_to_stage1_queue;
37  InterStageQueue<IntermediateTensor> stage1_to_stage2_queue;
38  InterStageQueue<IntermediateTensor> stage2_to_stage3_queue;
```

Queue Item Structure

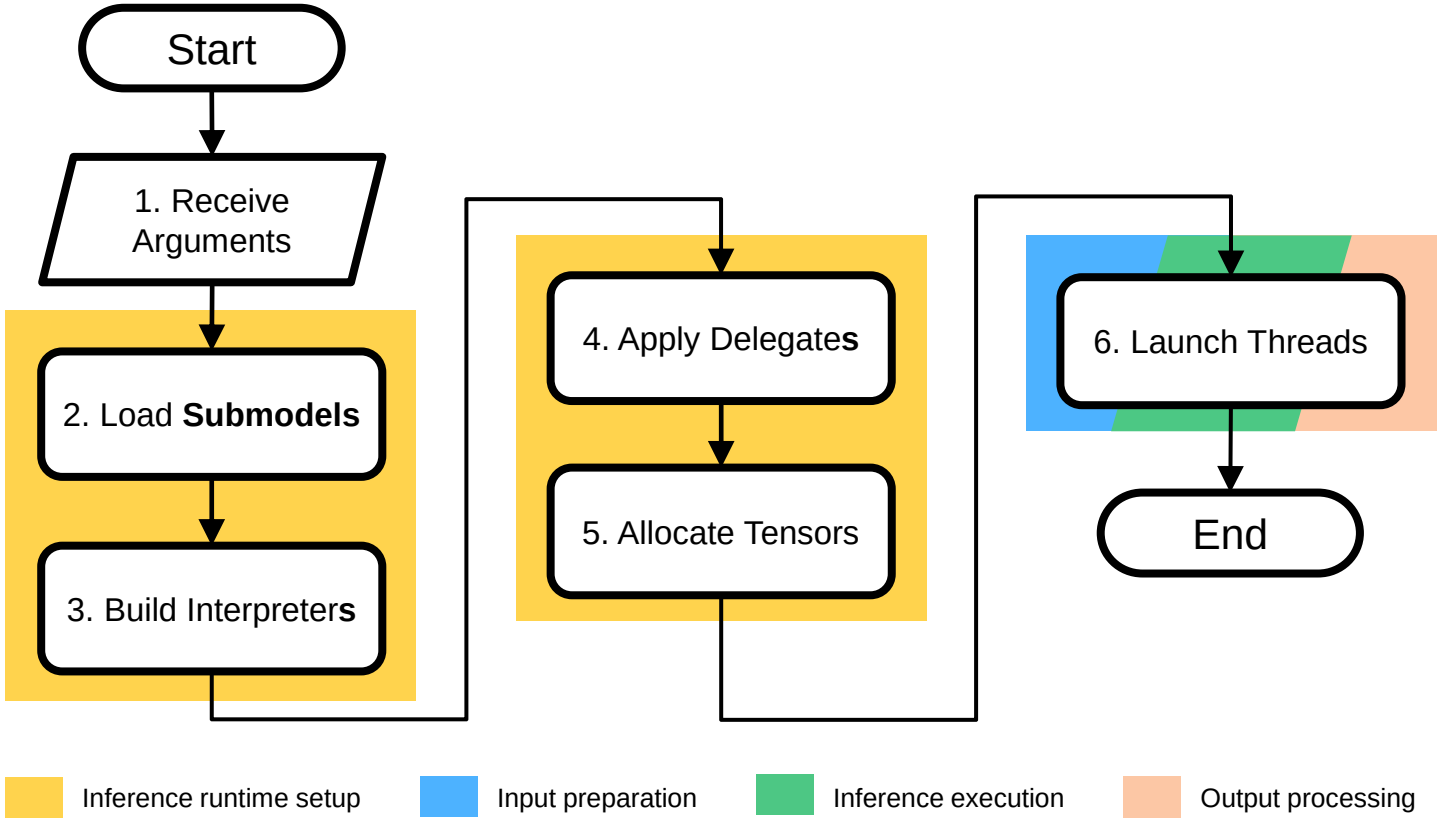
StageOutput

```
struct StageOutput {
    int index;                          // Index of the input image (used for tracking)
    std::vector<float> data;            // Flattened data of input/output tensors
    std::vector<int> tensor_end_offsets; // Offsets marking the end of each tensor in the flattened data
};
```



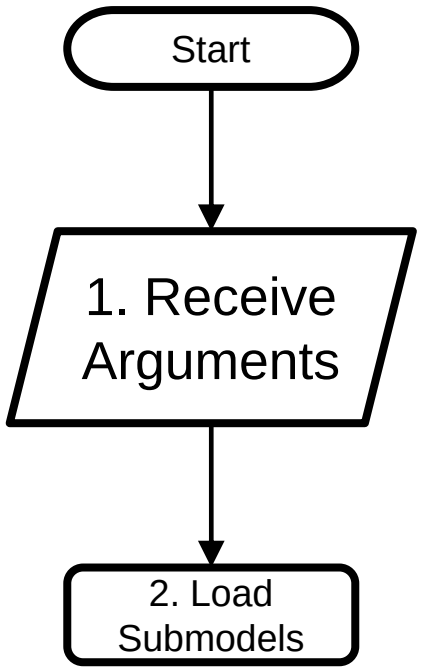
Contents

Code Flow


Start
1. Receive Arguments
2. Load **Submodels**
3. Build Interpreters
4. Apply Delegates
5. Allocate Tensors
6. Launch Threads
End
Inference runtime setup
Input preparation
Inference execution
Output processing

1. Receive Arguments (1)

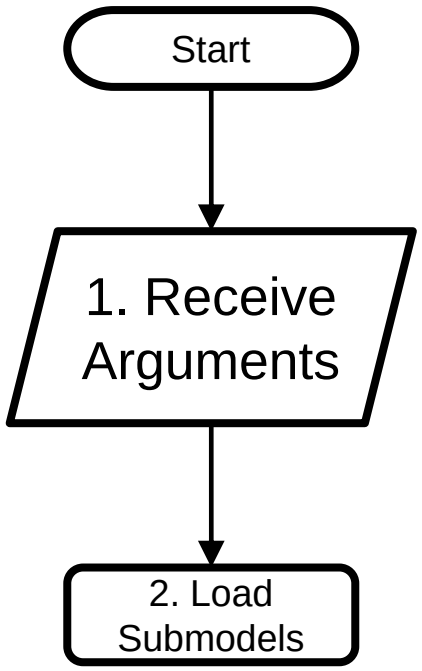
- Check the number of arguments




```
228 int main(int argc, char* argv[]) {
229     /* Receive user input */
230     if (argc < 7)
231     {
232         std::cerr << "Usage: " << argv[0]
233             << "<submodel0_path> <submodel0_gpu_usage> <submodel1_path> "
234             << "<submodel1_gpu_usage> <class_labels_path> <image_path 1> "
235             << "[image_path 2 ... image_path N] [--input-period=milliseconds]"
236             << std::endl;
237         return 1;
238     }
```

1. Receive Arguments (2)

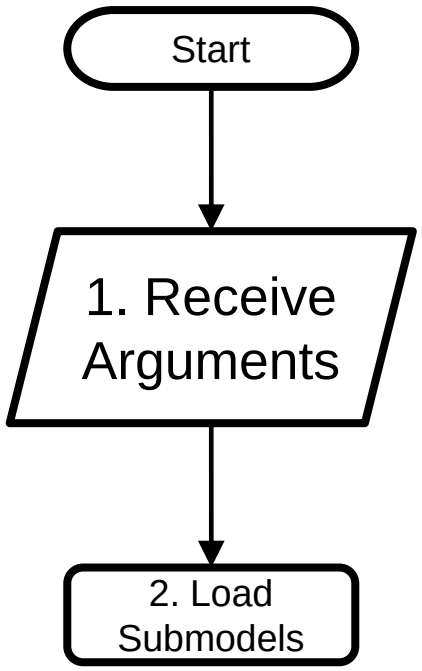
- Set variables based on the arguments




```
240     const std::string submodel0_path = argv[1];
241     bool submodel0_gpu_usage = false;
242     const std::string gpu_usage_str1 = argv[2];
243     if(gpu_usage_str1 == "true"){
244         submodel0_gpu_usage = true;
245     }
246
247     const std::string submodel1_path = argv[3];
248     bool submodel1_gpu_usage = false;
249     const std::string gpu_usage_str2 = argv[4];
250     if(gpu_usage_str2 == "true"){
251         submodel1_gpu_usage = true;
252     }
253
254     // Load class label mapping, used for postprocessing
255     const std::string class_labels_path = argv[5];
256     auto class_labels_map = util::load_class_labels(class_labels_path.c_str());
```

1. Receive Arguments (3)

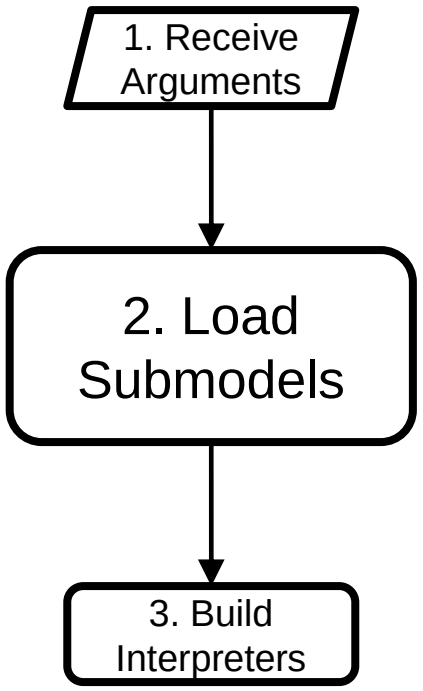
- Set variables based on the arguments (cont'd)




```
258     std::vector<std::string> images;    // List of input image paths
259     int input_period_ms = 0;            // Input period in milliseconds, default is 0 (no delay)
260     for (int i = 6; i < argc; ++i) {
261         std::string arg = argv[i];
262         if (arg.rfind("--input-period=", 0) == 0)
263             input_period_ms = std::stoi(arg.substr(15));
264         else
265             images.push_back(arg);  // Assume it's an image path
266     }
```

2. Load Submodels

- Load submodel 0 and 1

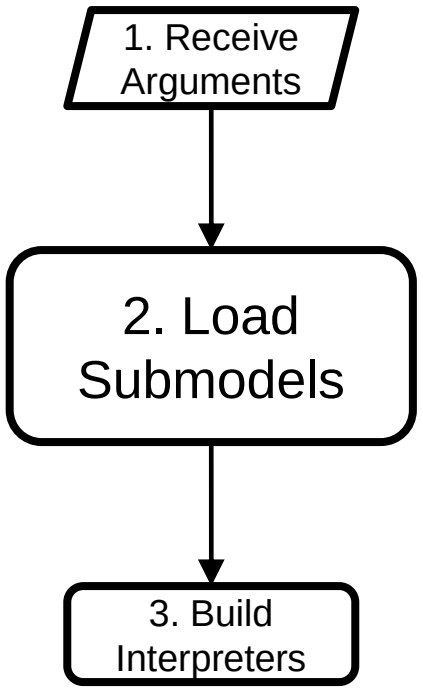



```
268   /* Load models */
269   // 1. Create a std::unique_ptr<tflite::FlatBufferModel> for each submodel
270   // ======= Write your code here =======
271
272
273
274
275
276
277
278
279   // ====================================
```

?

2. Load Submodels

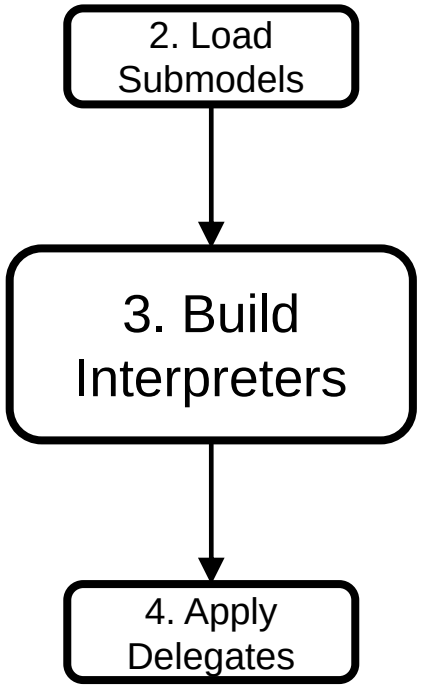
- Load submodel 0 and 1




```
268    /* Load models */
269    // 1. Create a std::unique_ptr<tflite::FlatBufferModel> for each submodel
270    // ======= Write your code here =======
271    std::unique_ptr<tflite::FlatBufferModel> submodel0_model =
272        tflite::FlatBufferModel::BuildFromFile(submodel0_path.c_str());
273    std::unique_ptr<tflite::FlatBufferModel> submodel1_model =
274        tflite::FlatBufferModel::BuildFromFile(submodel1_path.c_str());
275    if (!submodel0_model || !submodel1_model) {
276        std::cerr << "Failed to load one or both models" << std::endl;
277        return 1;
278    }
279    // ====================================
```

3. Build Interpreters

- Build interpreters for submodel 0 and submodel 1

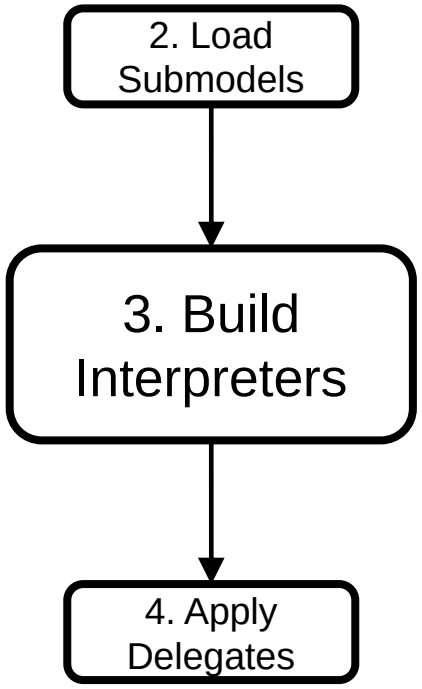



```
281   /* Build interpreters */
282   // 1. Create an OpResolver
283   // 2. Create two interpreter builders, one for each submodel
284   // 3. Build interpreters using the interpreter builders
285   // ======= Write your code here =======
286
287
288
289
290
291
292
293
294
295
296
297   // ====================================
```

?

3. Build Interpreters

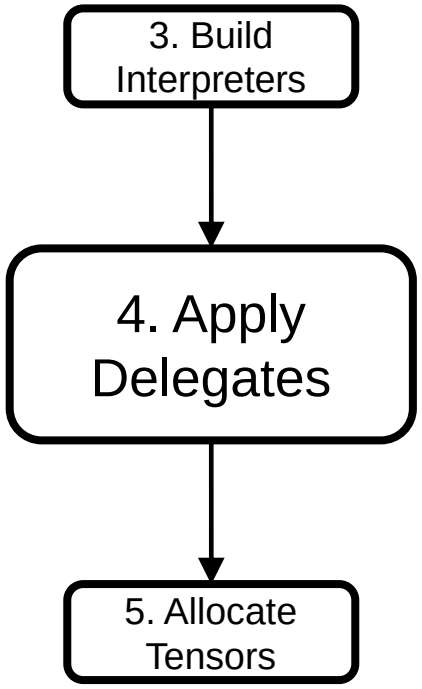
- Build interpreters for submodel 0 and submodel 1




```
281     /* Build interpreters */
282     // 1. Create an OpResolver
283     // 2. Create two interpreter builders, one for each submodel
284     // 3. Build interpreters using the interpreter builders
285     // ======= Write your code here =======
286     tflite::ops::builtin::BuiltinOpResolver resolver;
287     tflite::InterpreterBuilder submodel0_builder(*submodel0_model, resolver);
288     tflite::InterpreterBuilder submodel1_builder(*submodel1_model, resolver);
289     std::unique_ptr<tflite::Interpreter> submodel0_interpreter;
290     std::unique_ptr<tflite::Interpreter> submodel1_interpreter;
291     submodel0_builder(&submodel0_interpreter);
292     submodel1_builder(&submodel1_interpreter);
293     if (!submodel0_interpreter || !submodel1_interpreter) {
294         std::cerr << "Failed to initialize interpreters" << std::endl;
295         return 1;
296     }
297     // ====================================
```

4. Apply Delegates

- Create and apply delegates to the interpreters

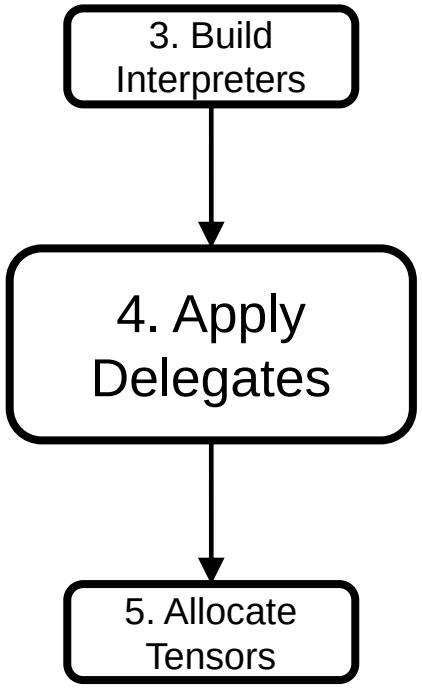



```
299   /* Apply delegate */
300   // 1. Create a XNNPACK delegate
301   // 2. Apply the delegate to the submodel0 interpreter
302   // 3. Create a GPU delegate
303   // 4. Apply the GPU delegate to the submodel1 interpreter
304   // ======= Write your code here =======
305
306
307
308
309
310
311
312
313
314
315   // ====================================
```

?

4. Apply Delegates

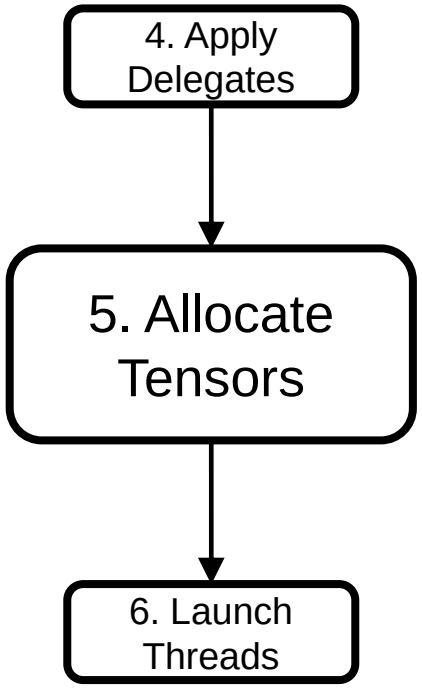
- Create and apply delegates to the interpreters




```
299     /* Apply delegate */
300     // 1. Create a XNNPACK delegate
301     // 2. Apply the delegate to the submodel0 interpreter
302     // 3. Create a GPU delegate
303     // 4. Apply the GPU delegate to the submodel1 interpreter
304     // ======= Write your code here =======
305     TfLiteDelegate* xnn_delegate = TfLiteXNNPackDelegateCreate(nullptr);
306     if(submodel0_interpreter->ModifyGraphWithDelegate(xnn_delegate) != kTfLiteOk) {
307         std::cerr << "Failed to apply XNNPACK delegate to submodel0" << std::endl;
308         return 1;
309     }
310     TfLiteDelegate* gpu_delegate = TfLiteGpuDelegateV2Create(nullptr);
311     if(submodel1_interpreter->ModifyGraphWithDelegate(gpu_delegate) != kTfLiteOk) {
312         std::cerr << "Failed to apply GPU delegate to submodel1" << std::endl;
313         return 1;
314     }
315     // ====================================
```

5. Allocate Tensors

- Allocate tensors for both interpreters

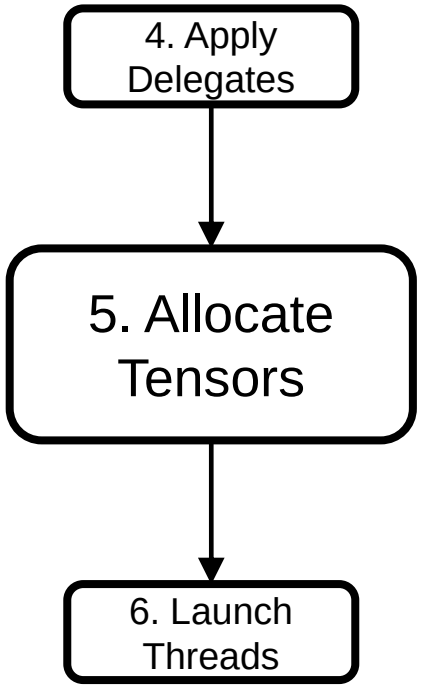



```
317   /* Allocate tensors */
318   // 1. Allocate tensors for both interpreters
319   // ======= Write your code here =======
320
321
322
323
324
325
326
327
328   // ====================================
```

?

5. Allocate Tensors

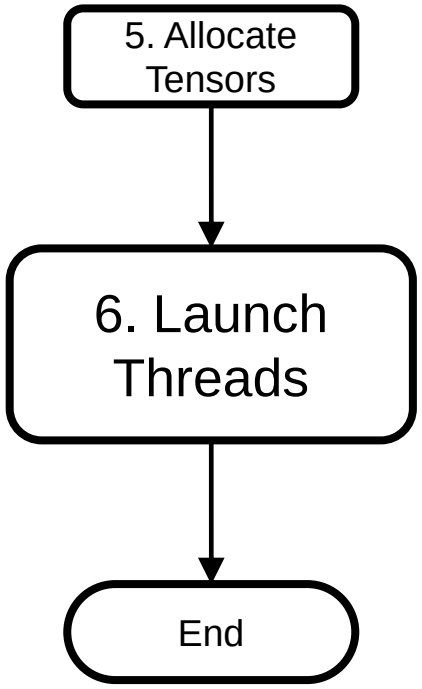
- Allocate tensors for both interpreters




```
317     /* Allocate tensors */
318     // 1. Allocate tensors for both interpreters
319     // ======= Write your code here =======
320     if (submodel0_interpreter->AllocateTensors() != kTfLiteOk) {
321         std::cerr << "Failed to allocate tensors for submodel0" << std::endl;
322         return 1;
323     }
324     if (submodel1_interpreter->AllocateTensors() != kTfLiteOk) {
325         std::cerr << "Failed to allocate tensors for submodel1" << std::endl;
326         return 1;
327     }
328     // ====================================
```

6. Launch Threads

- Create and launch threads for each stage




```
333     /* Create and launch threads */
334     // Hint: std::thread thread_name(function name, arguments...);
335     // 1. Launch stage0_worker in a new thread with images and input_period_ms
336     // 2. Launch stage1_worker in a new thread with submodel0 interpreter
337     // 3. Launch stage2_worker in a new thread with submodel1 interpreter
338     // 4. Launch stage3_worker in a new thread with class_labels_map
339     // ======= Write your code here =======
340     std::thread stage0_thread(stage0_worker, images, input_period_ms);
341     std::thread stage1_thread(stage1_worker, submodel0_interpreter.get());
342     std::thread stage2_thread(stage2_worker, submodel1_interpreter.get());
343     std::thread stage3_thread(stage3_worker, class_labels_map);
344     // ====================================
```

Contents

Stage Workers

- Functions executed by stage threads

1. `void stage0_worker(…);`
2. `void stage1_worker(…);`
3. `void stage2_worker(…);`
4. `void stage3_worker(…);`

Stage0 Worker (1)

- Specification
  - Input
    - **`images`**: Vector of image file paths
    - **`input_period_ms`**: Input period in milliseconds
  - Return value
    - None
  - Behavior
    1. Load and preprocess an image from the image file path vector
    2. Convert the preprocessed image into an **`StageOutput`**
    3. Push the **`StageOutput`** into **`stage0_to_stage1_queue`**
    4. Sleep until the next input period, unless already behind schedule
       - If no images remain, signal shutdown and return

Stage0 Worker (2)

1. Load and preprocess an image from the image file path vector

```
40  void stage0_worker(const std::vector<std::string>& images, int input_period_ms) {
41      auto next_wakeup_time = std::chrono::high_resolution_clock::now();
42      size_t idx = 0;
43      do {
44          std::string label = "Stage0 " + std::to_string(idx);
45          util::timer_start(label);
46          /* Preprocessing */
47          // Load image
48          cv::Mat image = cv::imread(images[idx]);
49          if (image.empty()) {
50              std::cerr << "[Stage0] Failed to load image: " << images[idx] << "\n";
51              util::timer_stop(label);
52              continue;
53          }
54
55          // Preprocess image
56          cv::Mat preprocessed_image = util::preprocess_image_resnet(image, 224, 224);
57          if (preprocessed_image.empty()) {
58              std::cerr << "[Stage0] Preprocessing failed: " << images[idx] << "\n";
59              util::timer_stop(label);
60              continue;
61          }
```

Stage0 Worker (3)

2. Convert the preprocessed image into an **`StageOutput`**

```
63      /* Create an StageOutput, copy preprocessed_image data into it,
64      *  and push it into stage0_to_stage1_queue */
65      // Hint: std::memcpy(destination_ptr, source_ptr, num_bytes);
66      StageOutput stage_output;
67      // ======= Write your code here =======
68
69
70
71
72
73
74
75      // ====================================
```

?

Stage0 Worker (3)

2. Convert the preprocessed image into an **`StageOutput`**

```
63      /* Create an StageOutput, copy preprocessed_image data into it,
64      *  and push it into stage0_to_stage1_queue */
65      // Hint: std::memcpy(destination_ptr, source_ptr, num_bytes);
66      StageOutput stage_output;
67      // ======= Write your code here =======
68      stage_output.index = idx;
69      stage_output.data.resize(
70          preprocessed_image.total() * preprocessed_image.channels());
71      std::memcpy(stage_output.data.data(), preprocessed_image.ptr<float>(),
72          stage_output.data.size() * sizeof(float));
73      stage_output.tensor_end_offsets =
74          {static_cast<int>(stage_output.data.size())};
75      // ====================================
```

Stage0 Worker (4)

3. Push the **`StageOutput`** into **`stage0_to_stage1_queue`**

```
76          stage0_to_stage1_queue.push(std::move(stage_output));
77          ++idx;
```

4. Sleep until the next input period, unless already behind schedule
   - If no images remain, signal shutdown and return

```
81          // Sleep to control the input rate
82          // If next_wakeup_time is in the past, it will not sleep
83          next_wakeup_time += std::chrono::milliseconds(input_period_ms);
84          std::this_thread::sleep_until(next_wakeup_time);
85      } while (idx < images.size());
86
87      // Notify stage1_thread that no more data will be sent
88      stage0_to_stage1_queue.signal_shutdown();
89  } // end of stage0_worker
```

Stage1 Worker (1)

- Specification
  - Input
    - **`interpreter`**: Pointer to an **`tflite::Interpreter`** object
  - Return value
    - None
  - Behavior
    1. Pop a **`StageOutput`** from **`stage0_to_stage1_queue`**
    2. Copy its data into the input tensor of the **`interpreter`**
    3. Run inference using the **`interpreter`**
    4. Extract data from the **`interpreter's`** output tensors and copy into a **`StageOutput`**
    5. Push the **`StageOutput`** into **`stage1_to_stage2_queue`**
    6. Repeat until **`stage0_to_stage1_queue`** signals shutdown and is empty
       - Then signal shutdown **`stage1_to_stage2_queue`** and return

Stage1 Worker (2)

1. Pop an **`StageOutput`** from **`stage0_to_stage1_queue`**

```
91  void stage1_worker(tflite::Interpreter* interpreter) {
92      StageOutput stage_output;
93
94      while (stage0_to_stage1_queue.pop(stage_output)) {
```

Stage1 Worker (3)

2. Copy its data into the input tensor of the **`interpreter`**

```
98    /* Access the 0th input tensor of the interpreter as a float pointer
99    *  and copy the contents of stage_output.data into it */
100   // Hint: std::memcpy(destination_ptr, source_ptr, num_bytes);
101   // ======= Write your code here =======
102   ?
103   ?
104
105   // ====================================
```

3. Run inference using the **`interpreter`**

```
107   /* Inference */
108   // ======= Write your code here =======
109   ?
110   // ====================================
```

Stage1 Worker (3)

2. Copy its data into the input tensor of the **`interpreter`**

```
98      /* Access the 0th input tensor of the interpreter as a float pointer
99      *  and copy the contents of stage_output.data into it */
100     // Hint: std::memcpy(destination_ptr, source_ptr, num_bytes);
101     // ======= Write your code here =======
102     float *input_tensor = interpreter->typed_input_tensor<float>(0);
103     std::memcpy(input_tensor, stage_output.data.data(),
104         stage_output.data.size() * sizeof(float));
105     // ====================================
```

3. Run inference using the **`interpreter`**

```
107     /* Inference */
108     // ======= Write your code here =======
109     interpreter->Invoke();
110     // ====================================
```

Stage1 Worker (4)

4. Extract data from the **`interpreter's`** output tensors and copy into a **`StageOutput`**

```
112         /* Extract data from the interpreter's output tensors and copy into a StageOutput */
113         // Clear data in it for reuse
114         stage_output.data.clear();
115         stage_output.tensor_end_offsets.clear();
116         // ======= Write your code here =======
117         for (size_t i = 0; i < interpreter->outputs().size(); ++i) {
118             // Get i-th output tensor object
119                                         ?
120
121             // Calculate the number of elements in the tensor
122             int num_elements = 1;
123             for (int d = 0; d < output_tensor->dims->size; ++d)
124                                         ?
```

Stage1 Worker (4)

4. Extract data from the **`interpreter's`** output tensors and copy into a **`StageOutput`**

```
112         /* Extract data from the interpreter's output tensors and copy into a StageOutput */
113         // Clear data in it for reuse
114         stage_output.data.clear();
115         stage_output.tensor_end_offsets.clear();
116         // ======= Write your code here =======
117         for (size_t i = 0; i < interpreter->outputs().size(); ++i) {
118             // Get i-th output tensor object
119             TfLiteTensor* output_tensor = interpreter->output_tensor(i);
120 
121             // Calculate the number of elements in the tensor
122             int num_elements = 1;
123             for (int d = 0; d < output_tensor->dims->size; ++d)
124                 num_elements *= output_tensor->dims->data[d];
```

Stage1 Worker (5)

4. Extract data from the **`interpreter's`** output tensors and copy into a **`StageOutput`** (cont'd)

```
126             // Resize stage_output.data and copy output tensor data into it
127             int current_data_length = stage_output.data.size();
128             stage_output.data.resize(current_data_length + num_elements);
129             std::memcpy(stage_output.data.data() + current_data_length,
130                 output_tensor->data.f,
131                 num_elements * sizeof(float));
132             stage_output.tensor_end_offsets.push_back(current_data_length + num_elements);
133         } // end of for loop
134         // ====================================
```

Stage1 Worker (6)

5. Push the **`StageOutput`** into **`stage1_to_stage2_queue`**

```
136         stage1_to_stage2_queue.push(std::move(stage_output));
137
138         util::timer_stop(label);
139     } // end of while loop
```

6. Repeat until **`stage0_to_stage1_queue`** signals shutdown and is empty
   - Then signal shutdown **`stage1_to_stage2_queue`** and return

```
141     // Notify stage2_thread that no more data will be sent
142     stage1_to_stage2_queue.signal_shutdown();
143 } // end of stage1_worker
```

Stage2 Worker (1)

- Specification
  - Input
    - **`interpreter`**: Pointer to an **`tflite::Interpreter`** object
  - Return value
    - None
  - Behavior
    1. Pop a **`StageOutput`** from **`stage1_to_stage2_queue`**
    2. Copy its data into the input tensors of the **`interpreter`**
    3. Run inference using the **`interpreter`**
    4. Extract data from the **`interpreter's`** output tensors and copy into a **`StageOutput`**
    5. Push the **`StageOutput`** into **`stage2_to_stage3_queue`**
    6. Repeat until **`stage1_to_stage2_queue`** signals shutdown and is empty
       - Then signal shutdown **`stage2_to_stage3_queue`** and return

Stage2 Worker (2)

1. Pop a **`StageOutput`** from **`stage1_to_stage2_queue`**

```
145 void stage2_worker(tflite::Interpreter* interpreter) {
146     StageOutput stage_output;
147 
148     while (stage1_to_stage2_queue.pop(stage_output)) {
```

Stage2 Worker (3)

2. Copy its data into the input tensors of the **`interpreter`**

```
152         /* Extract tensor data from stage_output.data and copy into
153         *  the corresponding input tensors of the interpreter */
154         // ======= Write your code here =======
155         for (size_t i = 0; i < interpreter->inputs().size(); ++i) {
156             // Get i-th input tensor from the interpreter as a float pointer
157                                          ?
158
159             // Copy data from stage_output to i-th input tensor
160                                          ?
161
162                                          ?
163
164         } // end of for loop
165         // ====================================
```

Stage2 Worker (3)

2. Copy its data into the input tensors of the **`interpreter`**

```
152         /* Extract tensor data from stage_output.data and copy into
153         *  the corresponding input tensors of the interpreter */
154         // ======= Write your code here =======
155         for (size_t i = 0; i < interpreter->inputs().size(); ++i) {
156             // Get i-th input tensor from the interpreter as a float pointer
157             float* input_data = interpreter->typed_input_tensor<float>(i);
158 
159             // Copy data from stage_output to i-th input tensor
160             int start_idx = (i == 0) ? 0 : stage_output.tensor_end_offsets[i-1];
161             int end_idx = stage_output.tensor_end_offsets[i];
162             std::memcpy(input_data, stage_output.data.data() + start_idx,
163                 (end_idx - start_idx) * sizeof(float));
164         } // end of for loop
165         // ====================================
```

Stage2 Worker (4)

3. Run inference using the **`interpreter`**

```
167     /* Inference */
168     // ======= Write your code here =======
169                         ?
170     // ====================================
```

Stage2 Worker (5)

4. Extract data from the **`interpreter's`** output tensors and copy into a **`StageOutput`**

```
172         /* Extract data from the interpreter's output tensors and copy into a StageOutput */
173         // Clear data in it for reuse
174         stage_output.data.clear();
175         stage_output.tensor_end_offsets.clear();
176         // ======= Write your code here =======
177         for (size_t i = 0; i < interpreter->outputs().size(); ++i) {
178             // Get i-th output tensor object
179                                     ?
180
181             // Calculate the number of elements in the tensor
182             int num_elements = 1;
183             for (int d = 0; d < output_tensor->dims->size; ++d)
184                                     ?
```

Stage2 Worker (5)

4. Extract data from the **`interpreter's`** output tensors and copy into a **`StageOutput`**

```
172         /* Extract data from the interpreter's output tensors and copy into a StageOutput */
173         // Clear data in it for reuse
174         stage_output.data.clear();
175         stage_output.tensor_end_offsets.clear();
176         // ======= Write your code here =======
177         for (size_t i = 0; i < interpreter->outputs().size(); ++i) {
178             // Get i-th output tensor object
179             TfLiteTensor* output_tensor = interpreter->output_tensor(i);
180 
181             // Calculate the number of elements in the tensor
182             int num_elements = 1;
183             for (int d = 0; d < output_tensor->dims->size; ++d)
184                 num_elements *= output_tensor->dims->data[d];
```

Stage2 Worker (6)

4. Extract data from the **`interpreter's`** output tensors and copy into a **`StageOutput`** (cont'd)

```
186             // Resize stage_output.data and copy output tensor data into it
187             int current_data_length = stage_output.data.size();
188             stage_output.data.resize(current_data_length + num_elements);
189             std::memcpy(stage_output.data.data() + current_data_length,
190                 output_tensor->data.f,
191                 num_elements * sizeof(float));
192             stage_output.tensor_end_offsets.push_back(current_data_length + num_elements);
193         } // end of for loop
194         // ====================================
```

Stage2 Worker (7)

5. Push the **`StageOutput`** into **`stage2_to_stage3_queue`**

```
195         stage2_to_stage3_queue.push(std::move(stage_output));
196         util::timer_stop(label);
197     } // end of while loop
```

6. Repeat until **`stage1_to_stage2_queue`** signals shutdown and is empty
   - Then signal shutdown **`stage2_to_stage3_queue`** and return

```
199     stage2_to_stage3_queue.signal_shutdown();
200 } // end of stage2_worker
```

Stage3 Worker (1)

- Specification
  - Input
    - **`class_labels_map`**: Map of class labels
  - Return value
    - None
  - Behavior
    1. Pop a **`StageOutput`** from **`stage2_to_stage3_queue`**
    2. Postprocess the output tensor data in the **`StageOutput`**
    3. Repeat until **`stage2_to_stage3_queue`** signals shutdown and is empty

Stage3 Worker (2)

1. Pop an **`StageOutput`** from **`stage2_to_stage3_queue`**

```
202 void stage3_worker(std::unordered_map<int, std::string> class_labels_map) {
203     StageOutput stage_output;
204
205     while (stage2_to_stage3_queue.pop(stage_output)) {
```

Stage3 Worker (3)

2. Postprocess the output tensor data in the **`StageOutput`**

```
209         const std::vector<float>& probs = stage_output.data;
210
211         if ((stage_output.index + 1) % 10 == 0) {
212             auto top_k_indices = util::get_topK_indices(probs, 3);
213             std::cout << "\n[stage3] Top-3 prediction for image index "
214                       << stage_output.index << ":\n";
215             for (int idx : top_k_indices) {
216                 std::string label = class_labels_map.count(idx)
217                     ? class_labels_map.at(idx)
218                     : "unknown";
219                 std::cout << "- Class " << idx << " (" << label
220                           << "): " << probs[idx] << std::endl;
221             }
222         }
```

3. Repeat until **`stage2_to_stage3_queue`** signals shutdown and is empty

```
225     } // end of while loop
226 } // end of stage3_worker
```

Contents

Hands-On: Build and Run the Pipelined Inference Driver

- Objective
  - Build and run the pipelined inference driver and measure throughput
- Do
  - Follow the instructions in the next slide
- Verify
  - Check the terminal output
    - You should see the expected outputs on the next slide
- Time
  - 2 minutes

Run Pipelined Inference Driver

- In **`~/RTCSA25-Tutorial`**, run
  - **`make pipelined`**
  - **`./run_pipelined_inference_driver.sh`**
- Expected output

```
[INFO] Top 3 predictions for image index 499:
- Class 867 (trailer_truck): 0.531833
- Class 675 (moving_van): 0.459434
- Class 569 (garbage_truck): 0.00453667
[INFO] Average Stage0 latency (500 runs): 2.136 ms

[INFO] Average Stage1 latency (500 runs): 46.2 ms

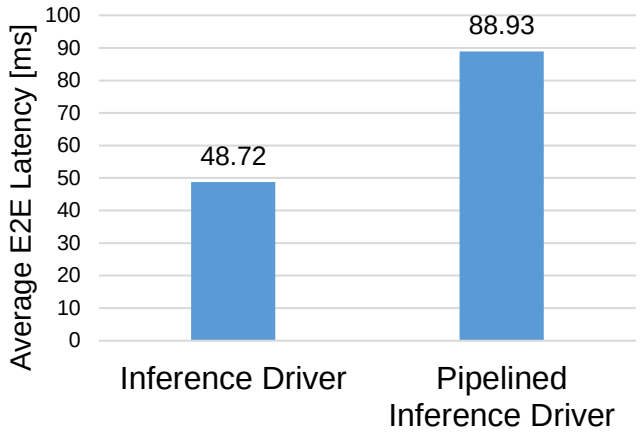
[INFO] Average Stage2 latency (500 runs): 40.594 ms

[INFO] Average Stage3 latency (500 runs): 0 ms

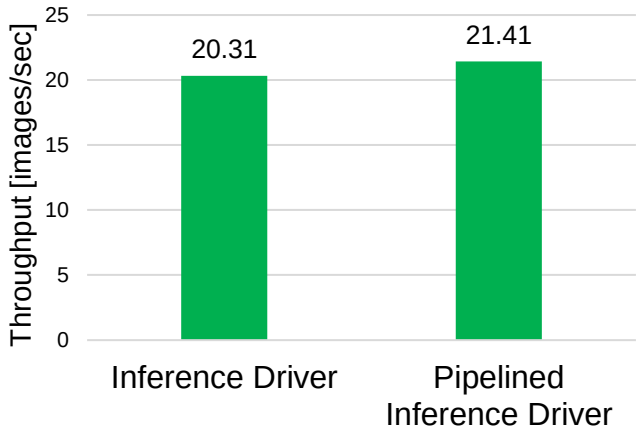
[INFO] Throughput: 21.4096 items/sec (500 items in 23354 ms)
```

Inference Driver vs. Pipelined Inference Driver

- End-to-End latency
  - Increased by about 1.83 times



- Throughput
  - Increased by about 5.4%



Q&A