Deep Software Stack Optimization for AI-Enabled Embedded Systems

Sep. 28, 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: *Exercise Overview and Setup* (1 h 30 min)	
Lecturer	Seongsoo Hong
Topics	Motivating Example
	Development Environment Setup
Coffee Break (30 min)	
Lecture 2: *From Inference Driver to Inference Runtime* (1 h 30 min)	
Lecturer	Seongsoo Hong and Namcheol Lee
Topics	Step-by-Step Inference Driver Walkthrough
	Internals of LiteRT
Lunch Break (1 h)	
Lecture 3: *Model Slicer* (1 h 30 min)	
Lecturer	Seongsoo Hong
Topic	Model Slicer: Slicing and Conversion Tool for LiteRT
Coffee Break (30 min)	
Lecture 4: *Throughput Enhancement on Heterogeneous Accelerators* (1 h 30 min)	
Lecturer	Namcheol Lee
Topic	Implementing a Pipelined Inference Driver for Heterogeneous Processors

Contents

On-Device AI

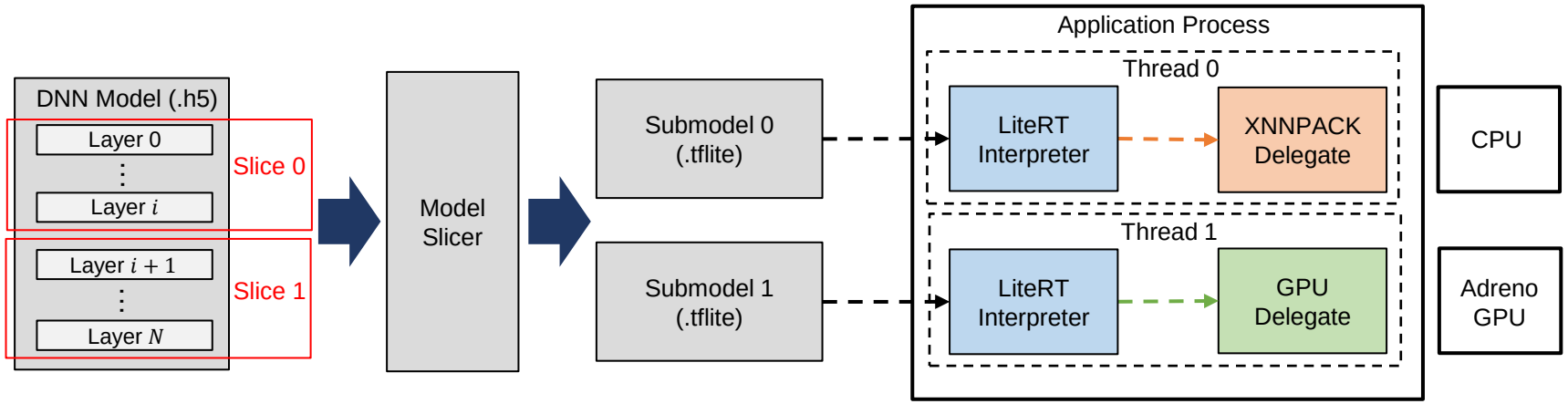
- Refers to "*executing AI models directly on edge devices*"
  - Benefits
    - Enables *real-time*, *privacy-preserving*, and *offline inference*
  - Challenge
    - Operates under limited resources available on edge devices
  - Solutions
    - Static model optimizations
      - Quantization, pruning, knowledge distillation, low-rank adaptation, weight streaming, and sparse inference
    - Dynamic optimizations
      - Pipelined inference, operator selection/scheduling, and operator fusion via capabilities
    - Dynamic resource management
      - Accelerator allocation, memory management, QoS regulation, etc.

Our Exercise Problem

- Lossless stream processing in an edge device
  - 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
  1. Partition DNN model into submodels
  2. Create worker threads that perform inference for submodels
  3. Allocate the worker threads to available, heterogeneous accelerators



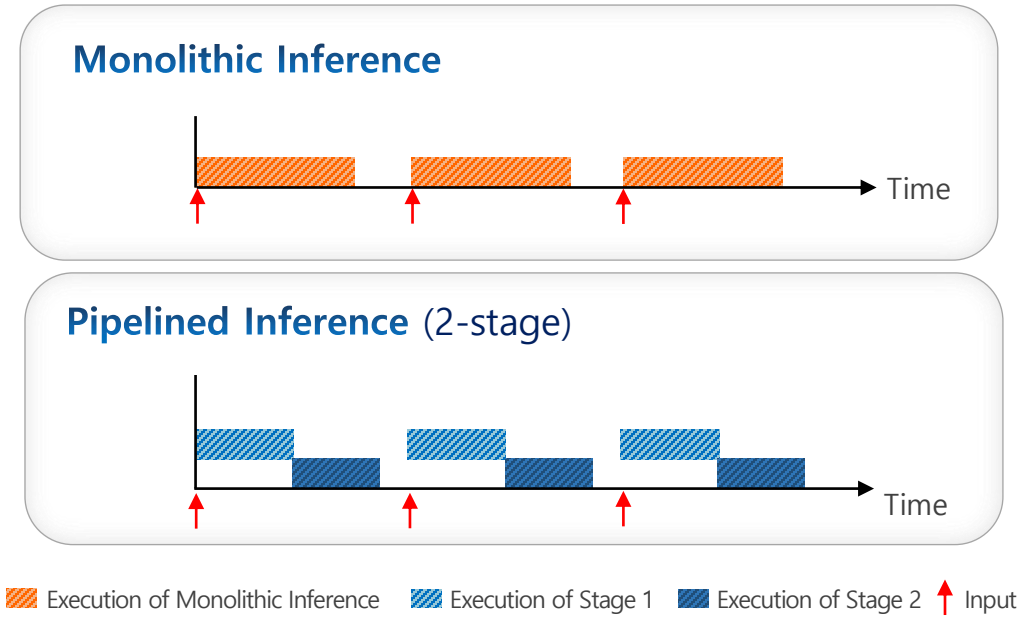
DNN Pipelining (2)

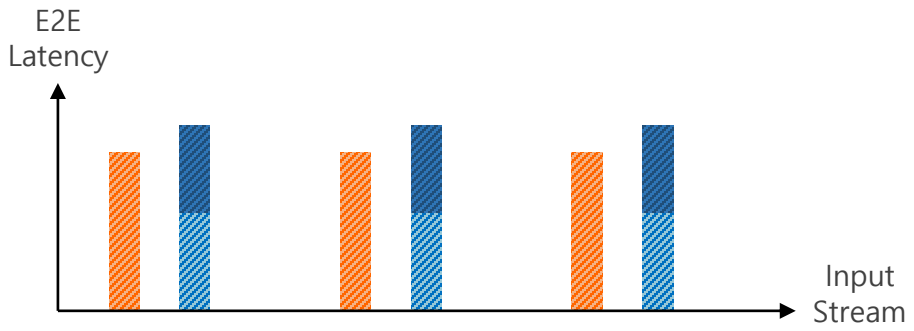
- What is the problem? And why is pipelining beneficial?

- Case analysis
  1. Normal case: *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. Exceptional case: *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)

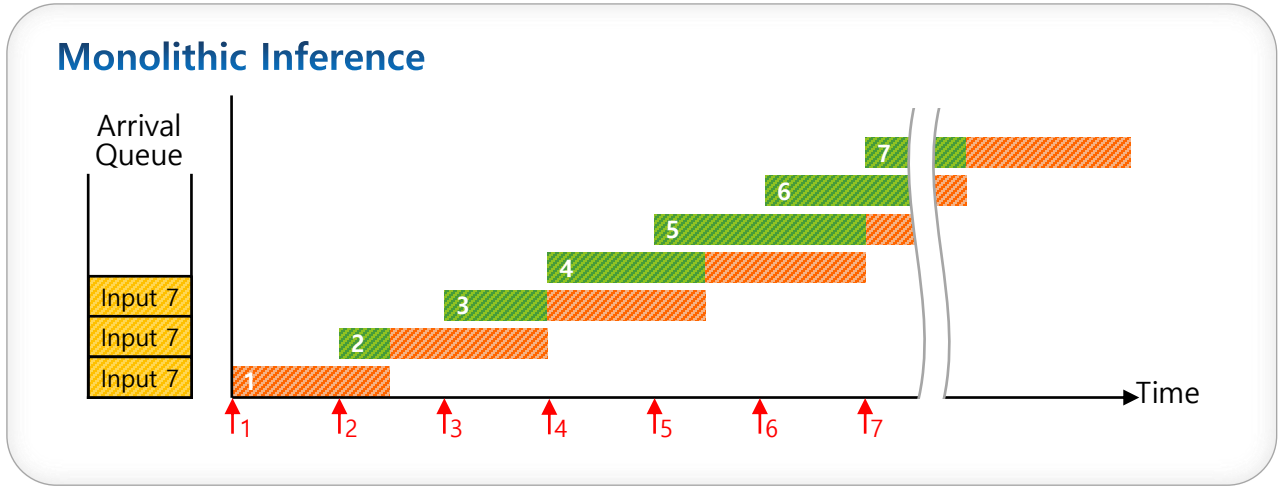
1. Normal case: *If input stream's data rate ≤ device's inference throughput*

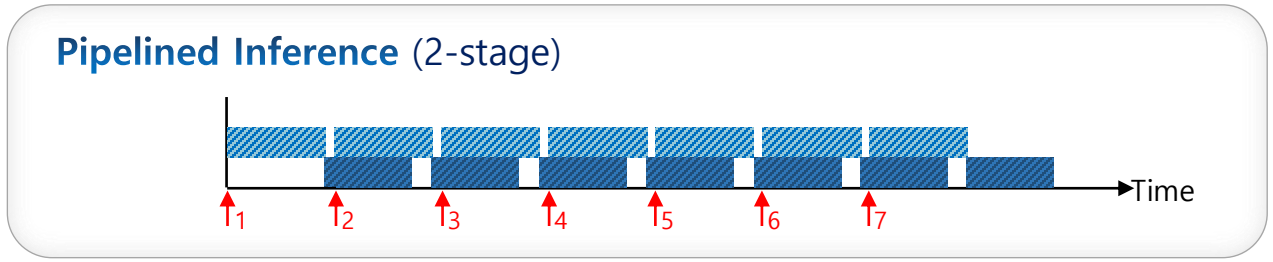


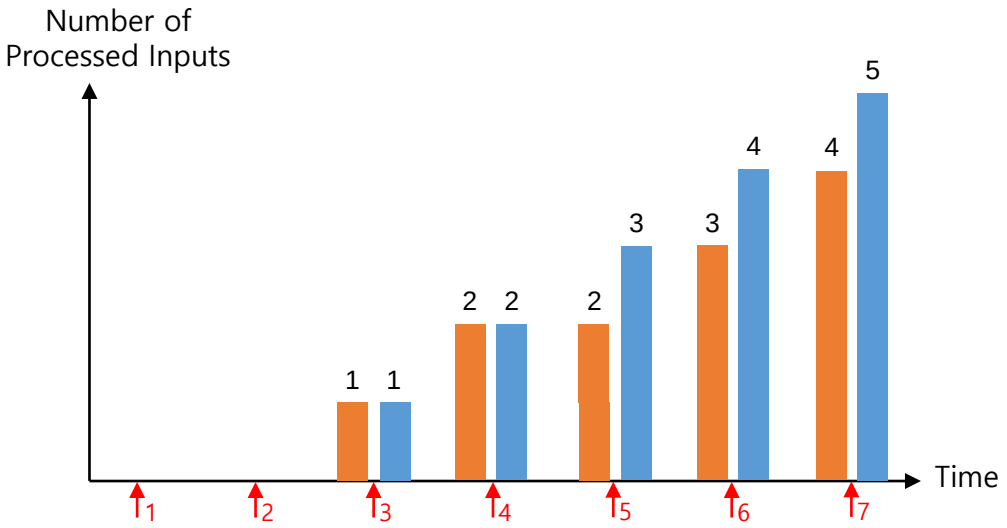


DNN Pipelining (4)

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



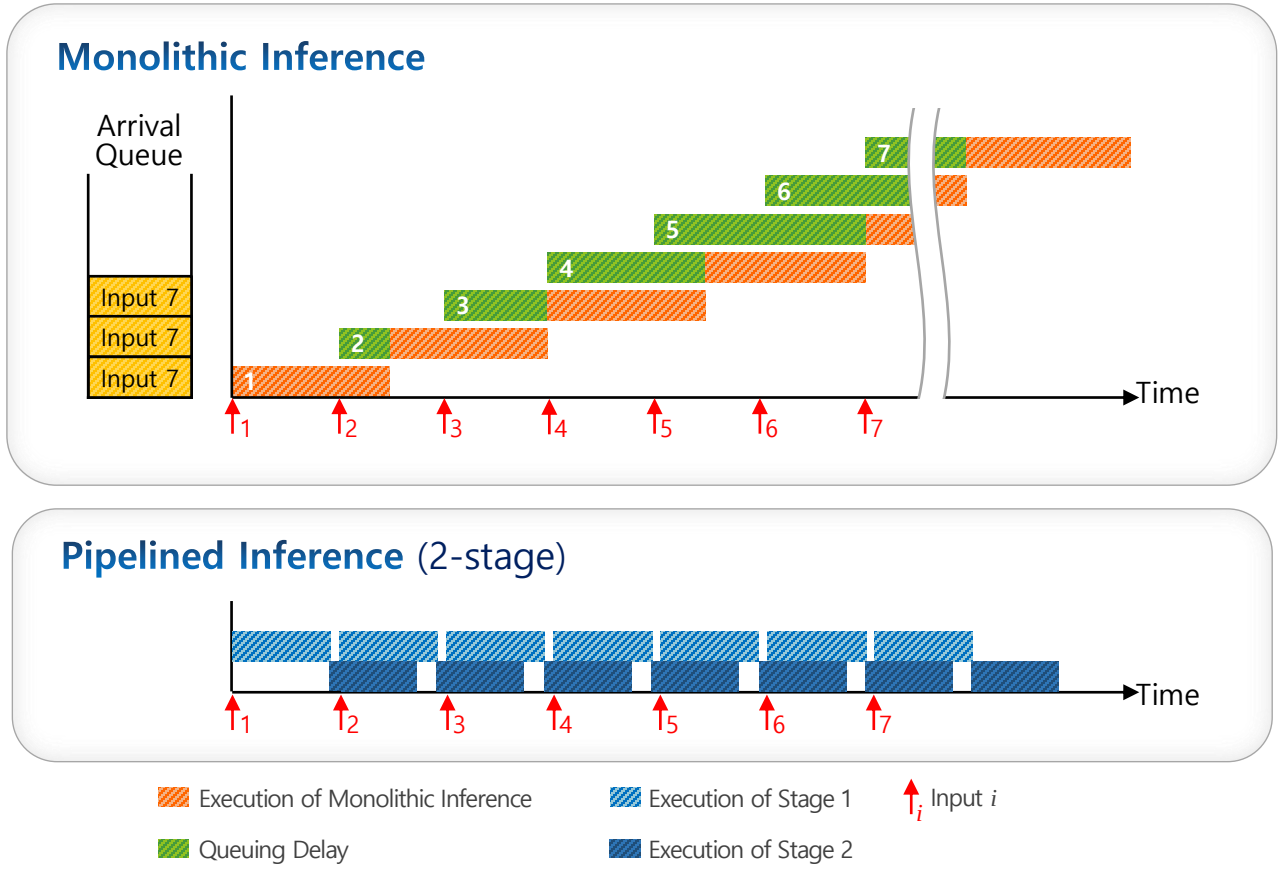


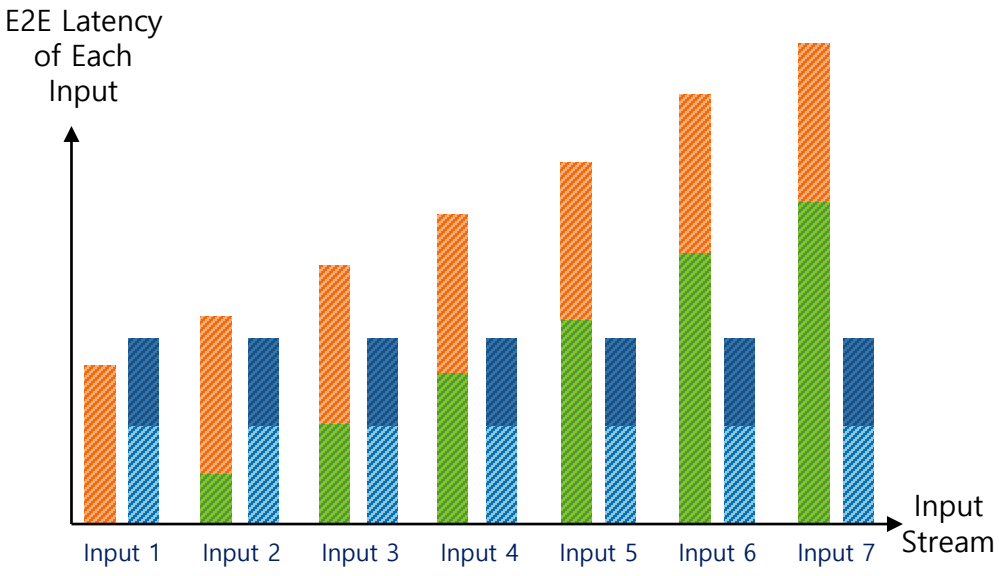


Execution of Monolithic Inference
Execution of Stage 1
$\uparrow_i$ Input i
Queuing Delay
Execution of Stage 2

DNN Pipelining (5)

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





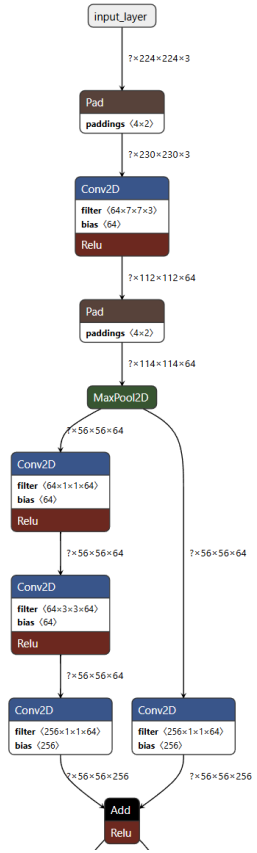
Target DNN Model: ResNet50 (1)

- 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)

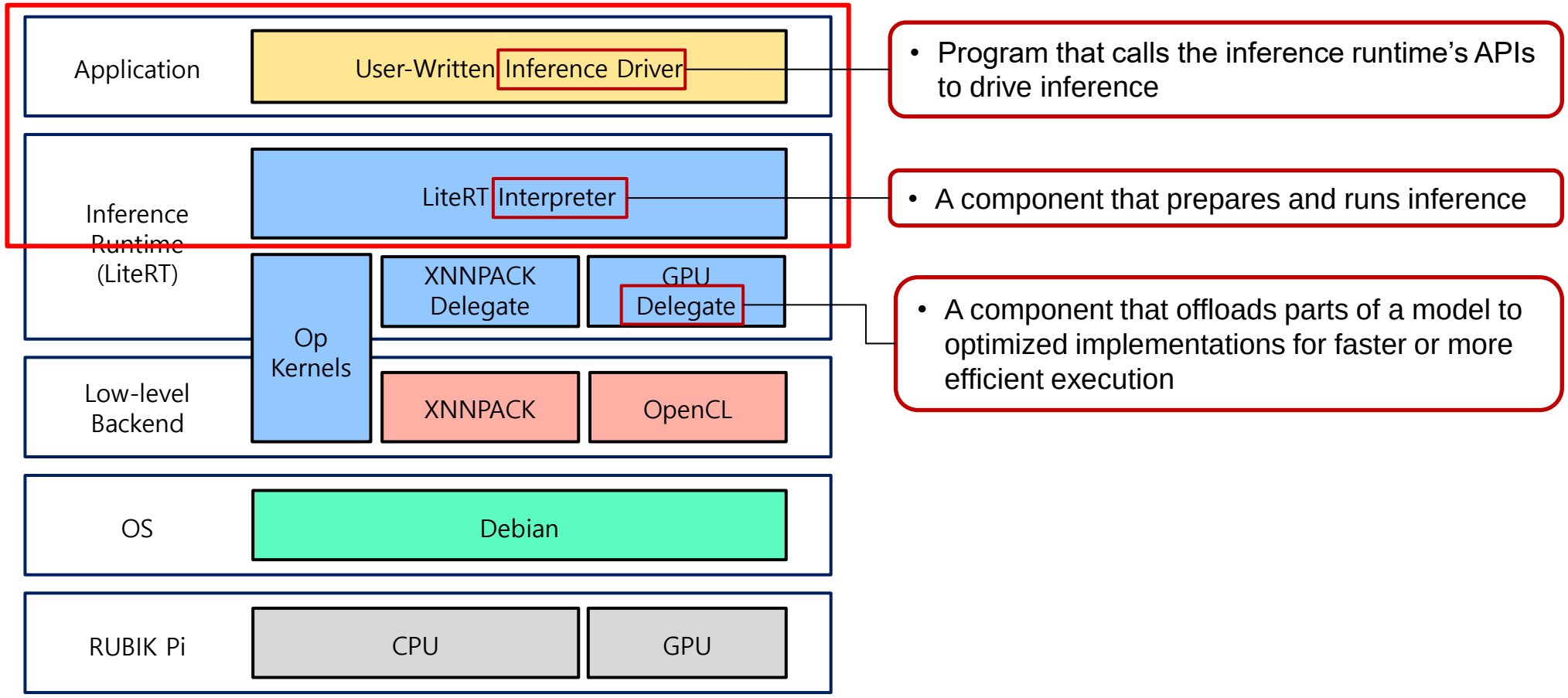
- Ideal for Hands-On learning of embedded AI software stack
  - RUBIK Pi 3 with QCS6490 SoC enables realistic, practice-oriented exercises aligned with real-world industrial applications

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



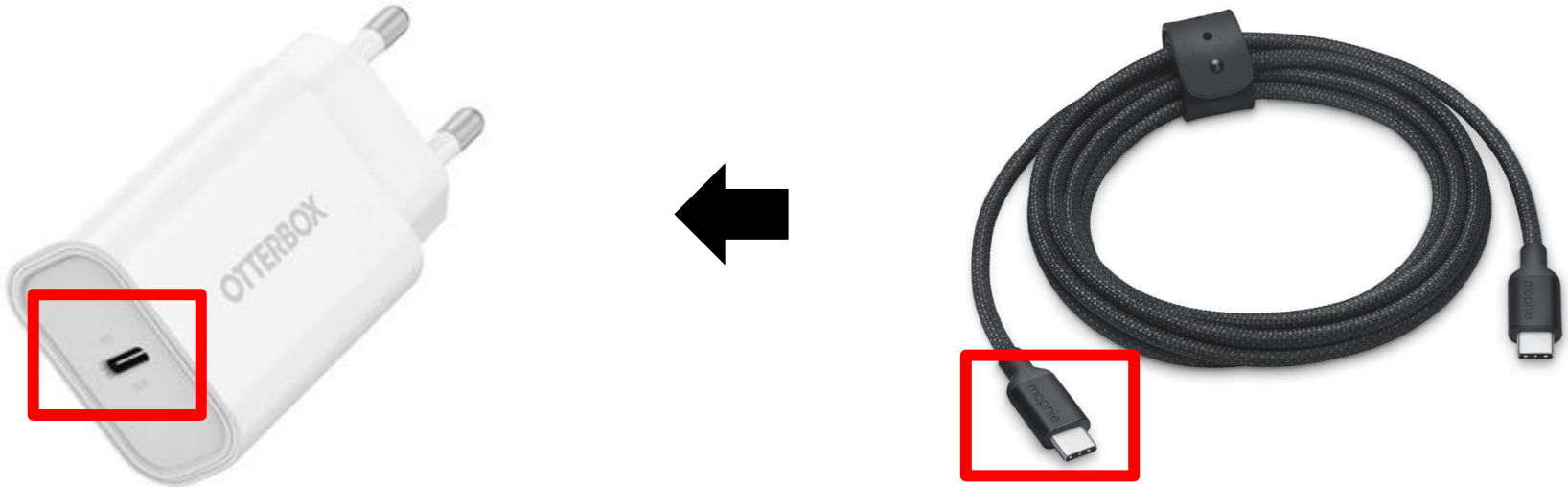
Contents

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
  - 60 minutes

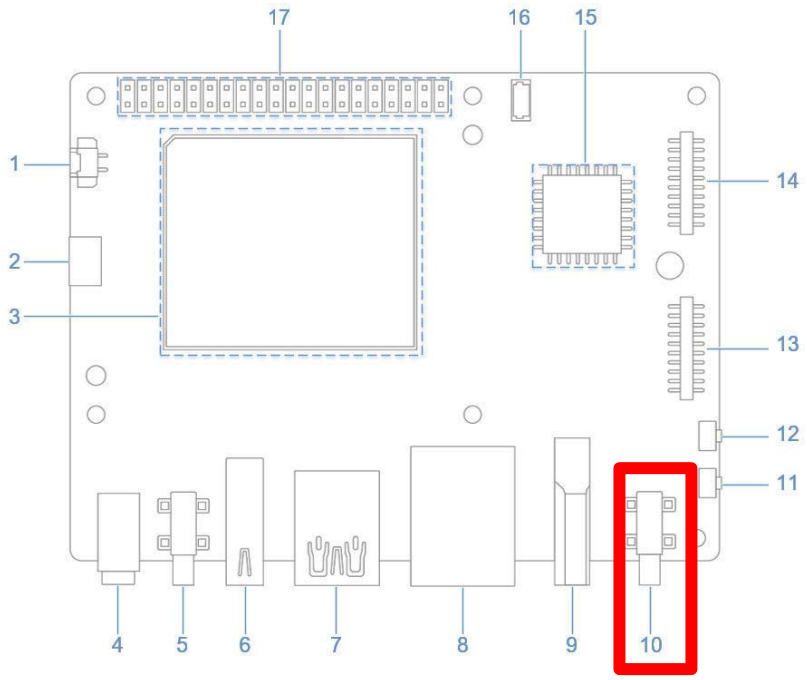
Supply Power (1)

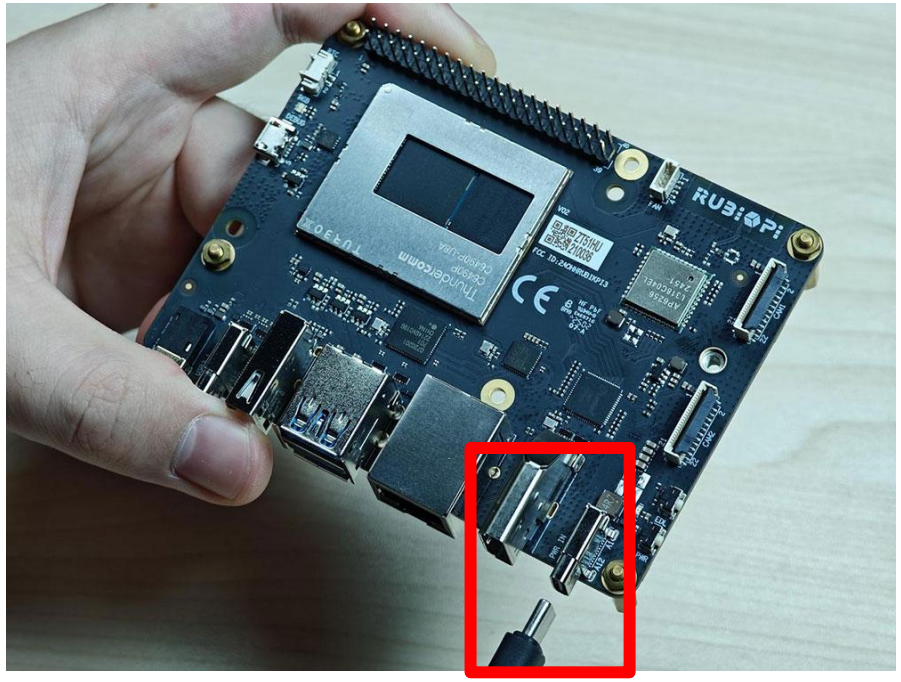
- Connect power cable to the external power supply, and make sure it is switched on



Supply Power (2)

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



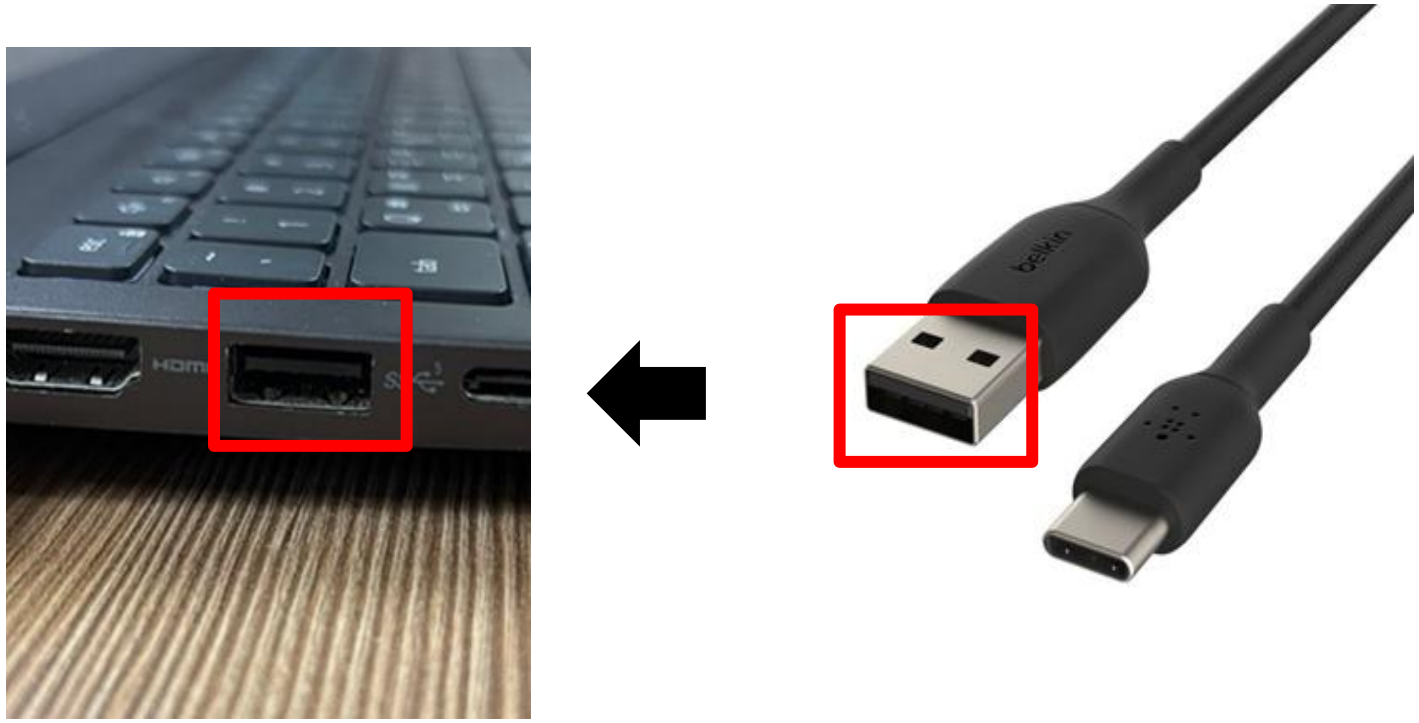


Connect RUBIK Pi to Host PC

- 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

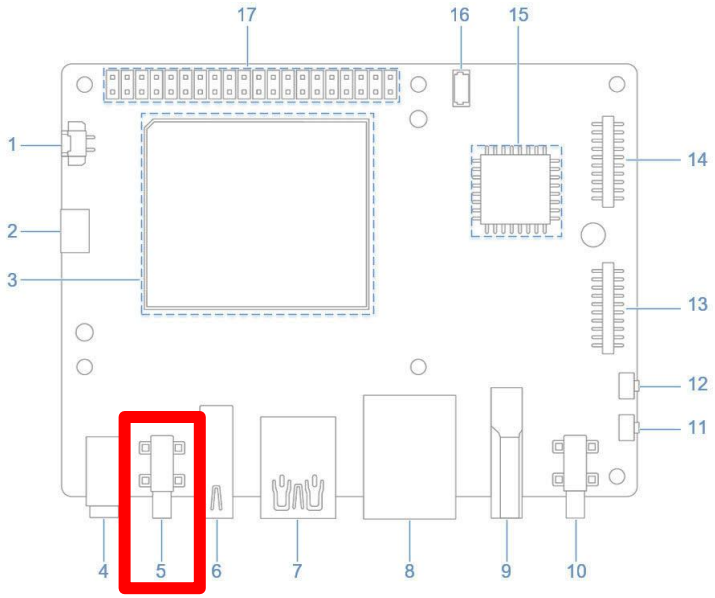
1. Connect Device to Host PC with USB-A to USB-C Cable (1)

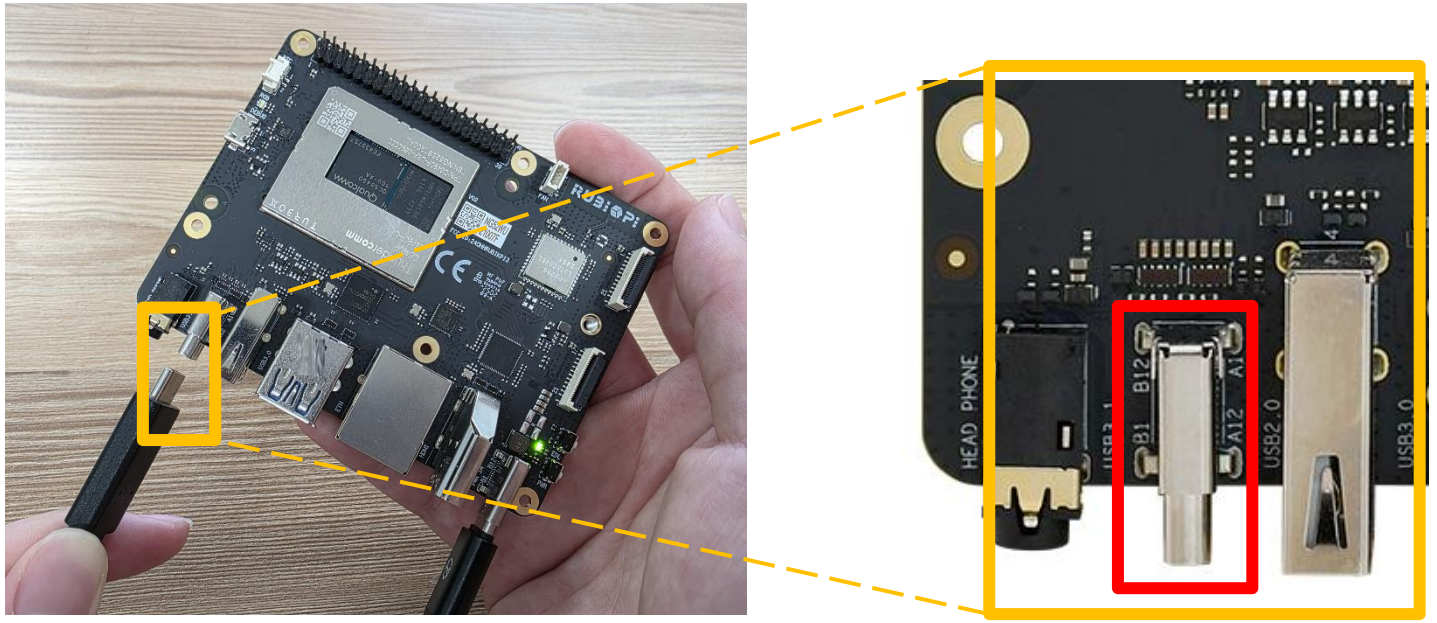
- Connect the USB-A end of the cable to host PC



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





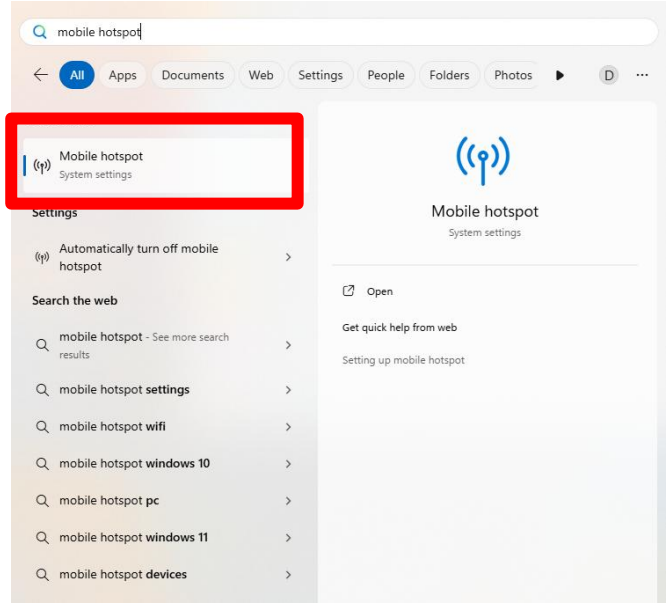
1. Connect Device to Host PC with USB-A to USB-C Cable (3)

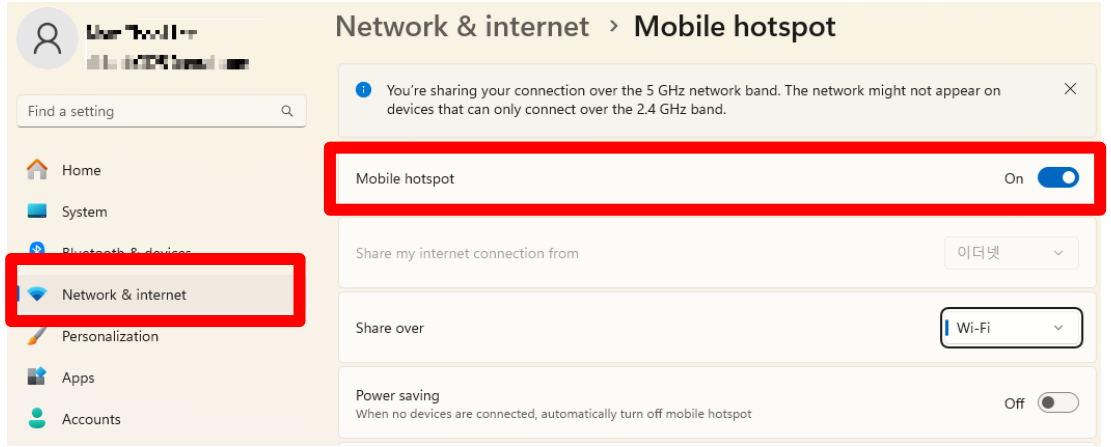
- Check the connection using ADB
  - It should show one device as below
    - The identifier of the device varies across devices

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

2. Establish Wi-Fi Connection (1)

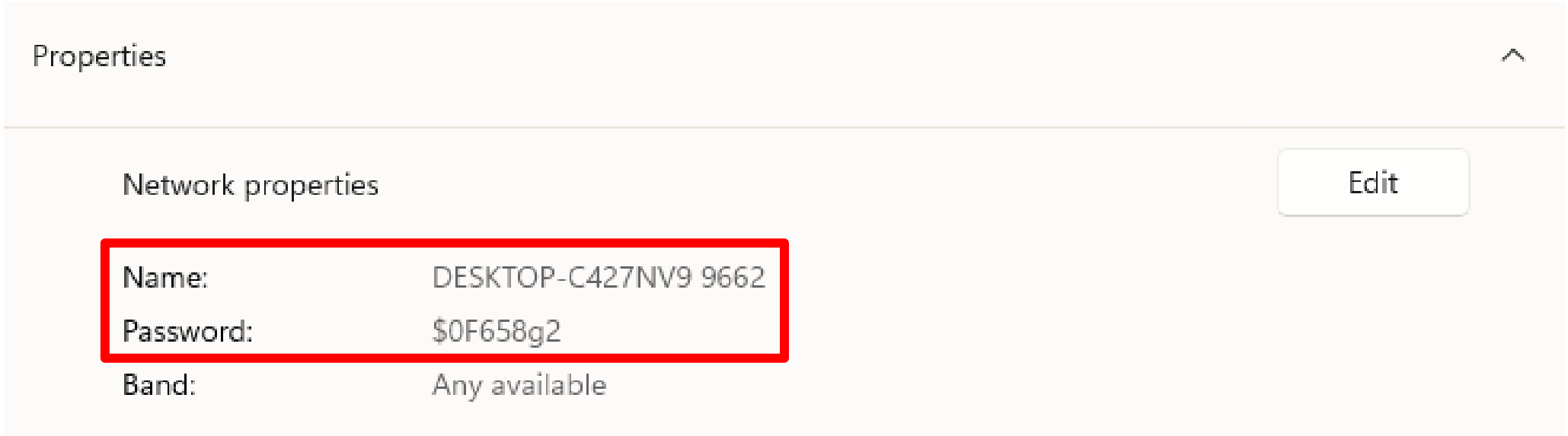
- Enable hotspot on host PC
  - Go to **Settings > Network & Internet**





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)

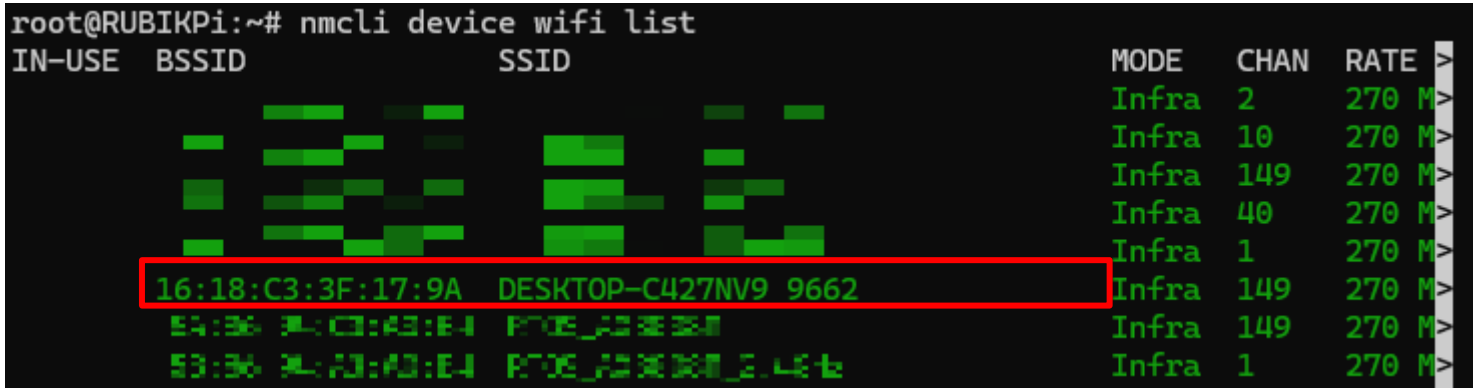
- Access the device in a command line tool with the following commands
  1. Get root access
     - **`adb root`**
  2. Enter adb shell
     - **`adb shell`**
  3. Switch to root user
     - **`su`**

```
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`**

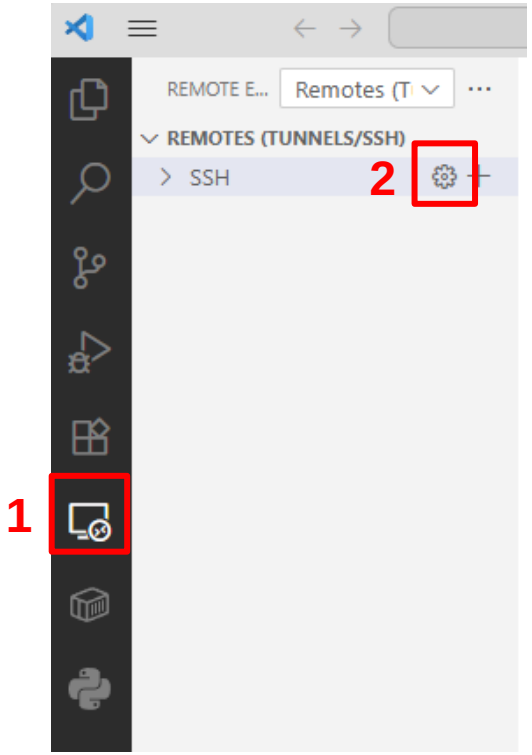
```
root@RUBIKPi:~# nmcli device wifi connect "DESKTOP-C427NV9 9662" --ask
Push of the WPS button on the router or a password is required to access the wireless
 network 'DESKTOP-C427NV9 9662'.
Password (802-11-wireless-security.psk): ********
Device 'wlan0' successfully activated with '0ea65cff-dd17-4aa1-b934-2dedbe33d58f'.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

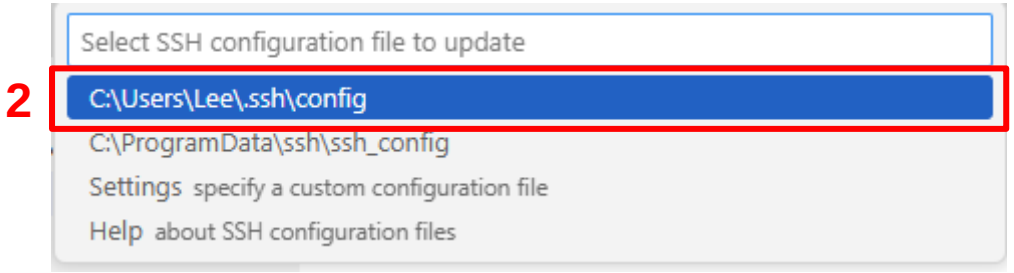
  3. Check the connection
     - **`ifconfig wlan0`**

```
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)

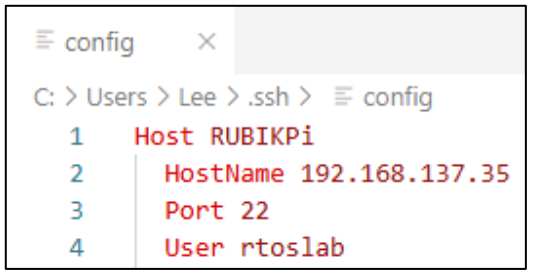
- Create SSH configuration in VS Code
  1. Open remote explorer in VS Code in the left side bar
  2. Open SSH config file



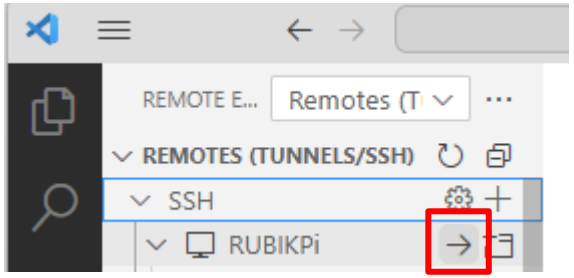


3. Make SSH Connection (2)

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

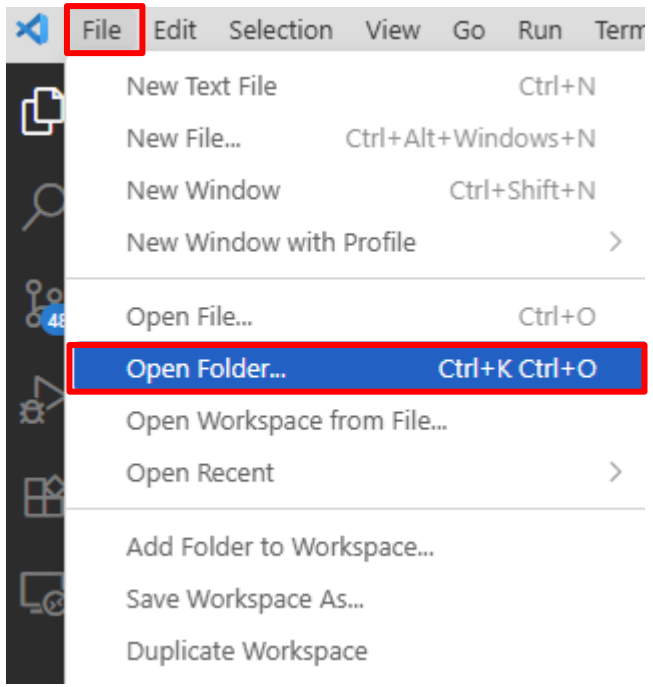


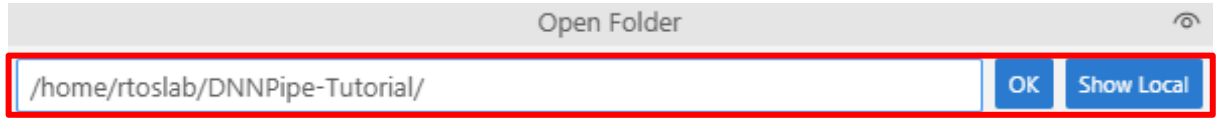
  4. Connect to the RUBIK Pi with both user name and password being "**`rtoslab`**"



3. Make SSH Connection (3)

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





Tutorial Outline

Lecture 1: *Exercise Overview and Setup* (1 h 30 min)	
Lecturer	Seongsoo Hong
Topics	Motivating Example
	Development Environment Setup
Coffee Break (30 min)	
Lecture 2: *From Inference Driver to Inference Runtime* (1 h 30 min)	
Lecturer	Seongsoo Hong and Namcheol Lee
Topics	Step-by-Step Inference Driver Walkthrough
	Internals of LiteRT
Lunch Break (1 h)	
Lecture 3: *Model Slicer* (1 h 30 min)	
Lecturer	Seongsoo Hong
Topic	Model Slicer: Slicing and Conversion Tool for LiteRT
Coffee Break (30 min)	
Lecture 4: *Throughput Enhancement on Heterogeneous Accelerators* (1 h 30 min)	
Lecturer	Namcheol Lee
Topic	Implementing a Pipelined Inference Driver for Heterogeneous Processors

Contents

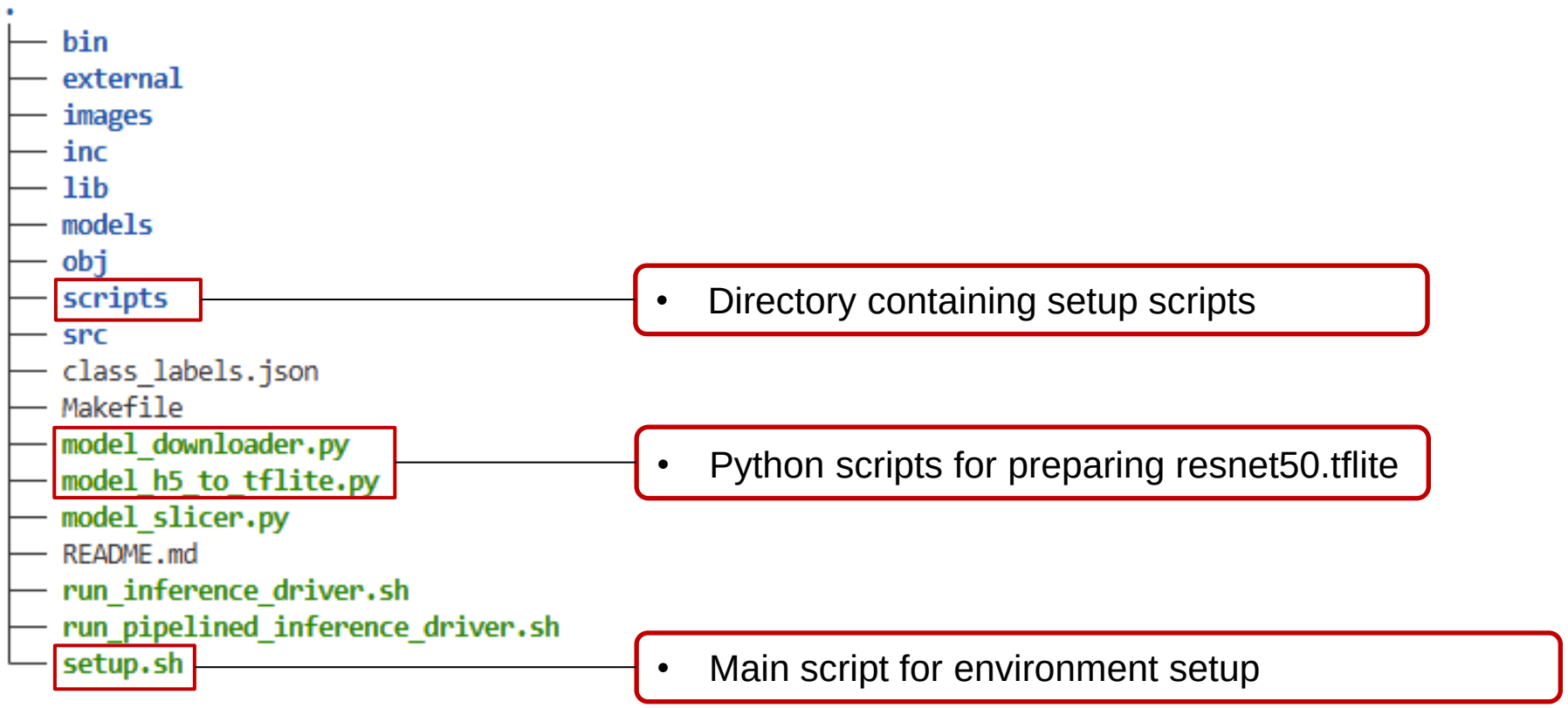
Directory Structure

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

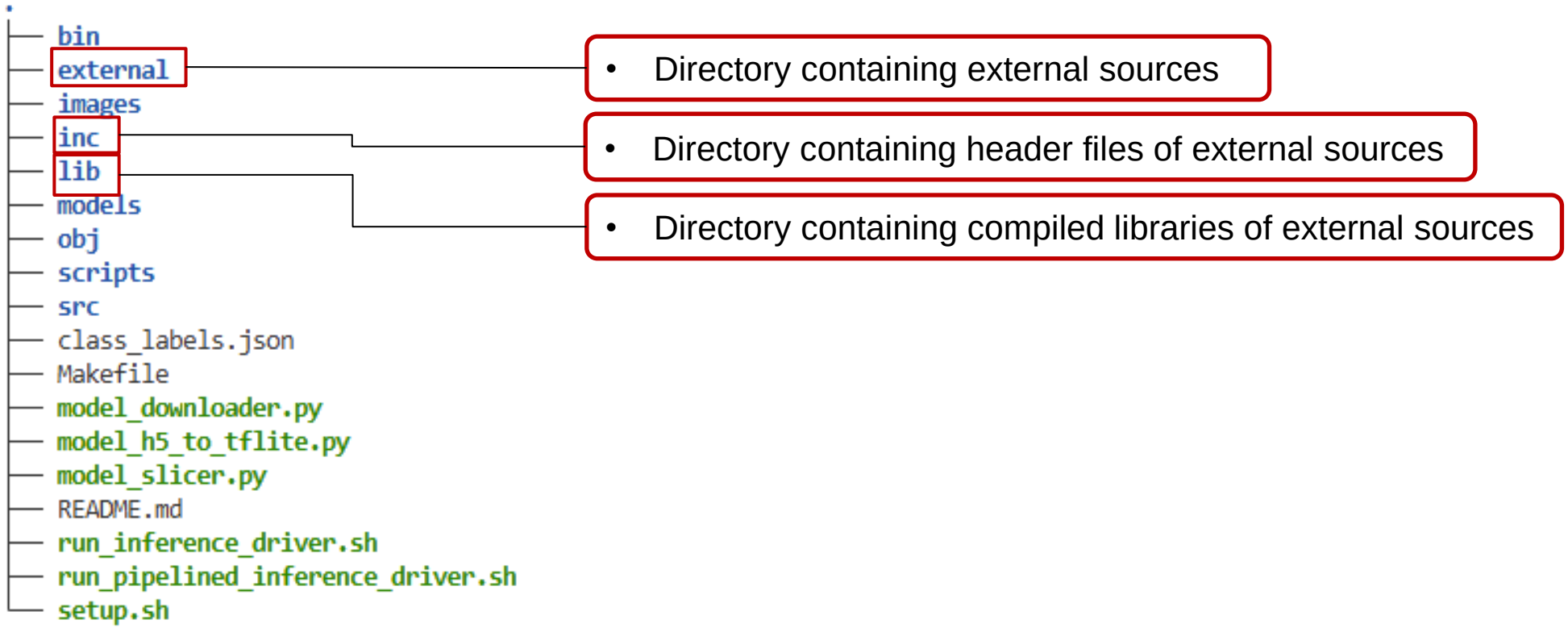
Setup Scripts

- Scripts to install software packages and prepare required files



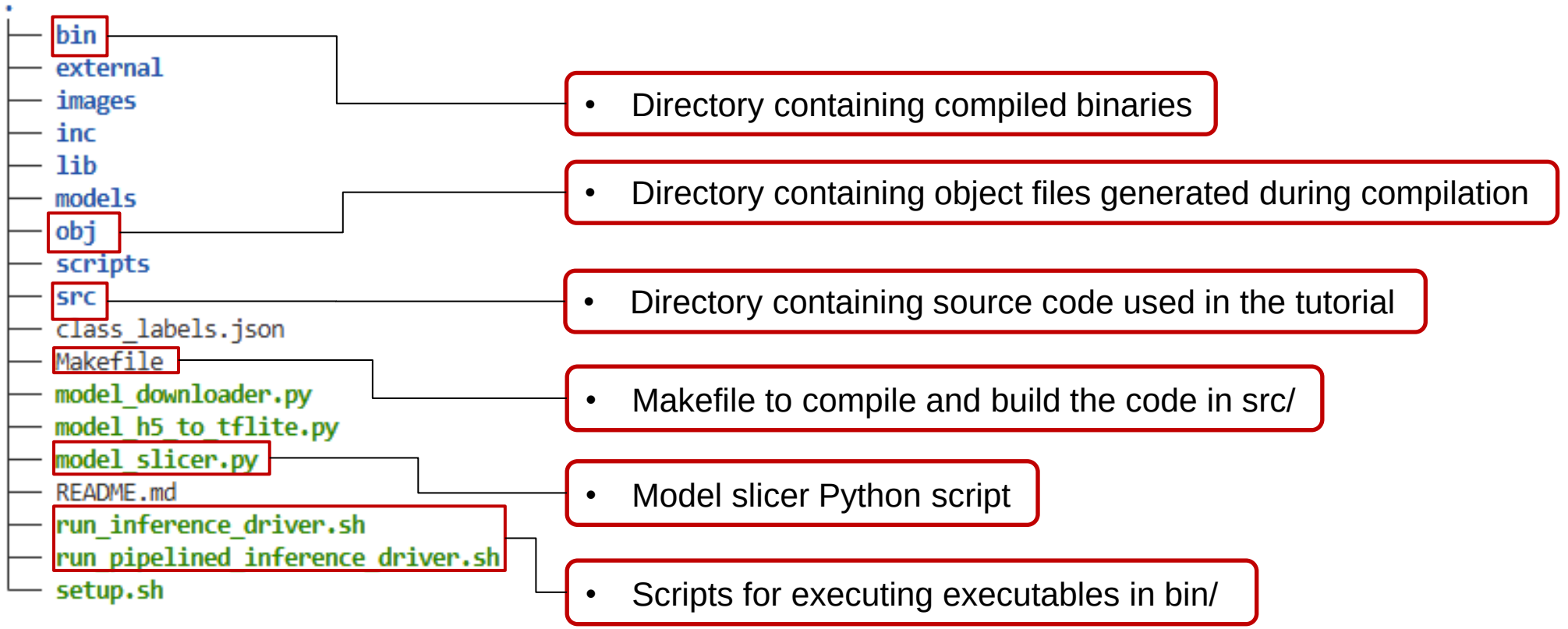
External Libraries

- Directories for externals libraries such as LiteRT



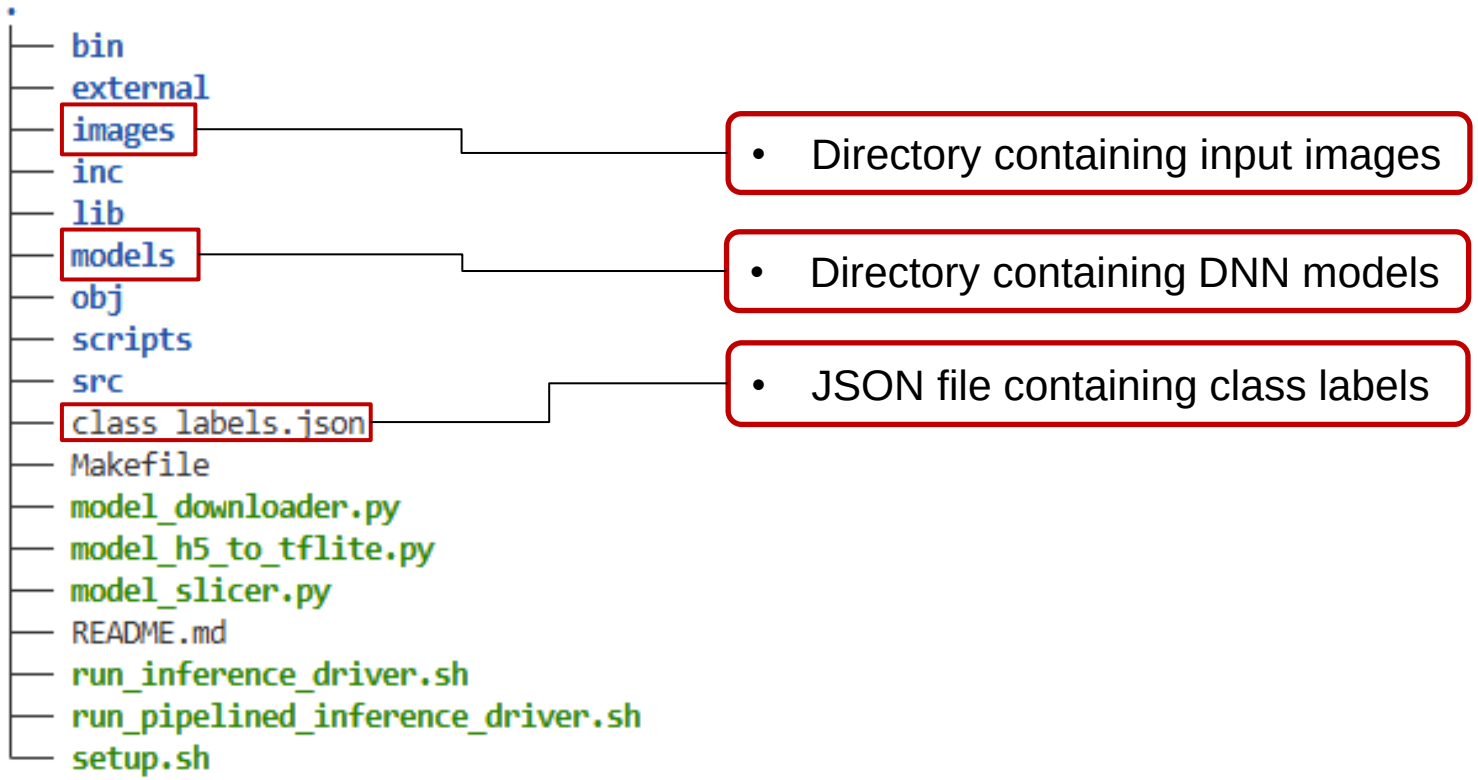
Source Code and Build File

- 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

DNNPipe-Tutorial > src > inference_driver.cpp

```
15  #include <iostream>
16  #include <thread>
17  #include <vector>
18  #include <opencv2/opencv.hpp>
19  #include "tflite/delegates/xnnpack/xnnpack_delegate.h"
20  #include "tflite/delegates/gpu/delegate.h"
21  #include "tflite/interpreter_builder.h"
22  #include "tflite/interpreter.h"
23  #include "tflite/kernels/register.h"
24  #include "tflite/model_builder.h"
25  #include "util.hpp"
```

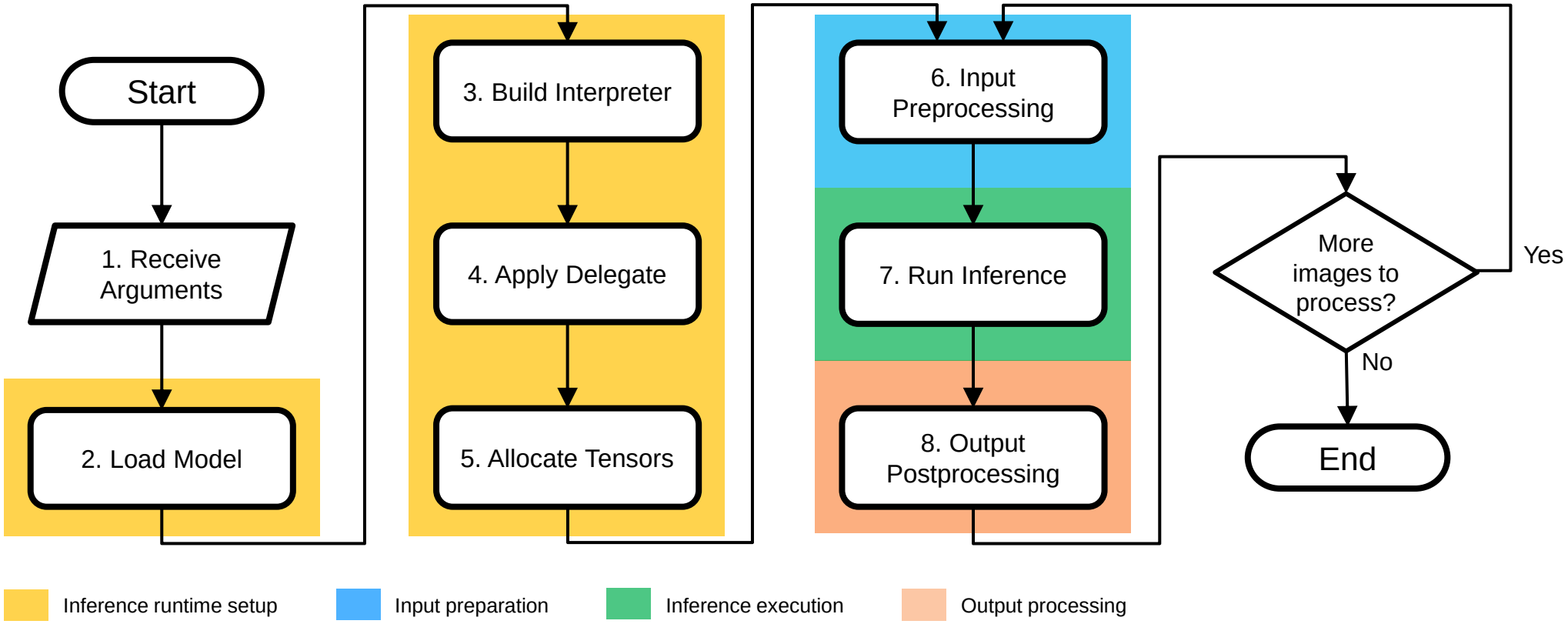
C++ standard library headers for I/O stream, thread management, and dynamic arrays

Header file for OpenCV used during input preprocessing

Header files for using LiteRT

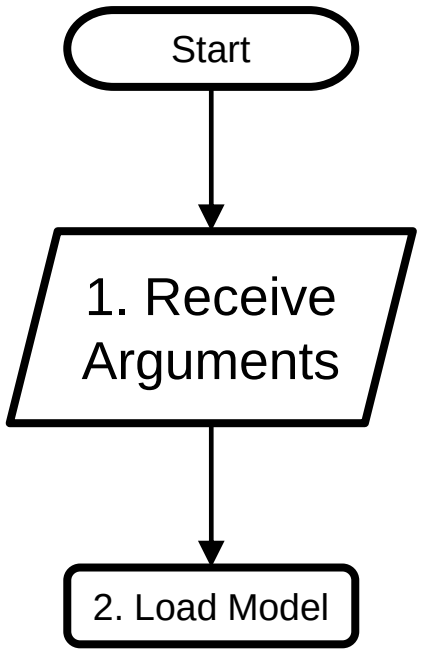
Header file for utility functions used during the tutorial

Inference Driver Code Flow



1. Receive Arguments (1)

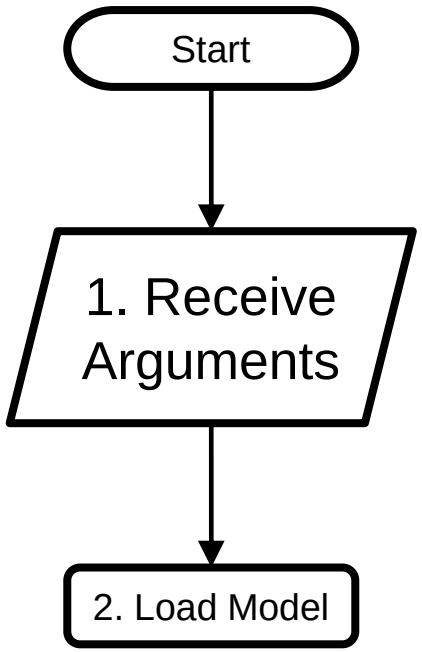
- Check the number of arguments




```
35      if (argc < 5)
36      {
37          std::cerr << "Usage: " << argv[0]
38              << "<model_path> <gpu_usage> <class_labels_path> <image_path 1> "
39              << "[image_path 2 ... image_path N] [--input-period=milliseconds]"
40              << std::endl;
41          return 1;
42      }
```

1. Receive Arguments (2)

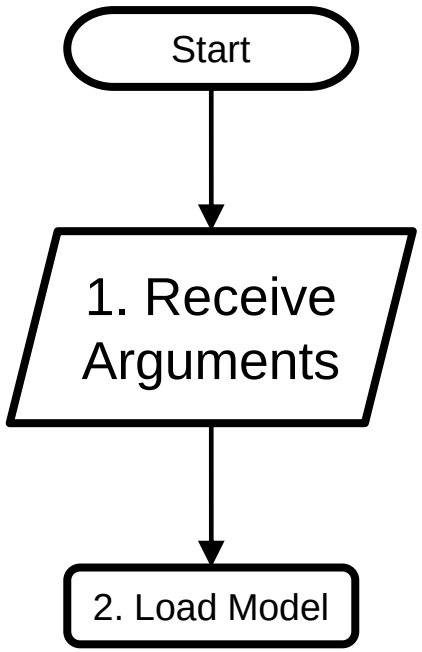
- Set variables based on the arguments




```
44      const std::string model_path = argv[1];
45
46      bool gpu_usage = false; // If true, GPU delegate is applied
47      const std::string gpu_usage_str = argv[2];
48      if(gpu_usage_str == "true"){
49          gpu_usage = true;
50      }
51
52      // Load class label mapping, used for postprocessing
53      const std::string class_labels_path = argv[3];
54      auto class_labels_map = util::load_class_labels(class_labels_path.c_str());
```

1. Receive Arguments (3)

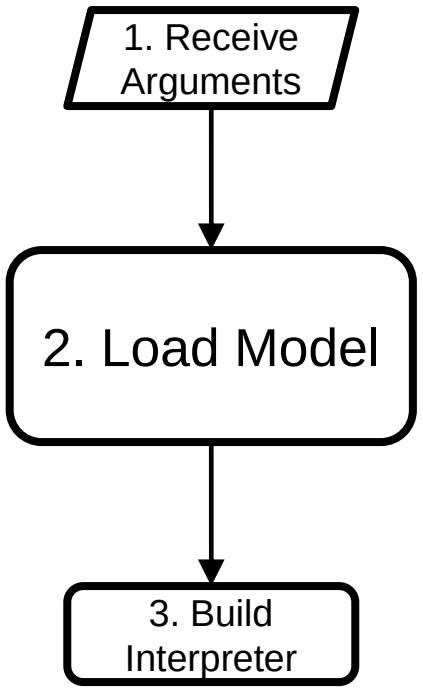
- Set variables based on the arguments (cont'd)




```
56      std::vector<std::string> images;    // List of input image paths
57      int input_period_ms = 0;            // Input period in milliseconds, default is 0 (no delay)
58      for (int i = 4; i < argc; ++i) {
59          std::string arg = argv[i];
60          if (arg.rfind("--input-period=", 0) == 0) // Check for input period argument
61              input_period_ms = std::stoi(arg.substr(15));
62          else
63              images.push_back(arg);  // Assume it's an image path
64      }
```

2. Load Model

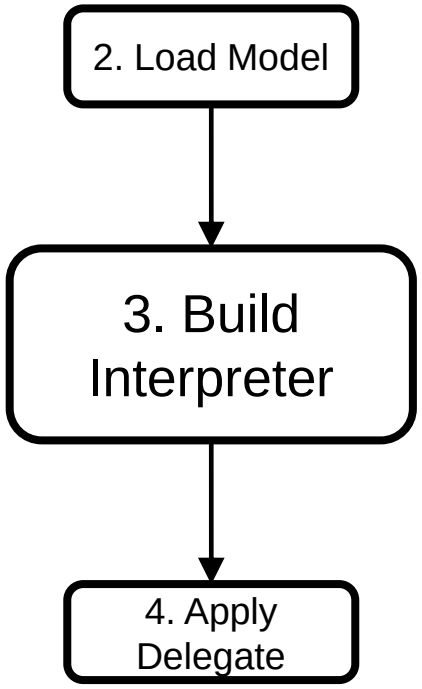
- Load a model file at **`model_path`**




```
66      /* Load model */
67      std::unique_ptr<tflite::FlatBufferModel> model =
68          tflite::FlatBufferModel::BuildFromFile(model_path.c_str());
69      if (!model)
70      {
71          std::cerr << "Failed to load model" << std::endl;
72          return 1;
73      }
```

3. Build Interpreter (1)

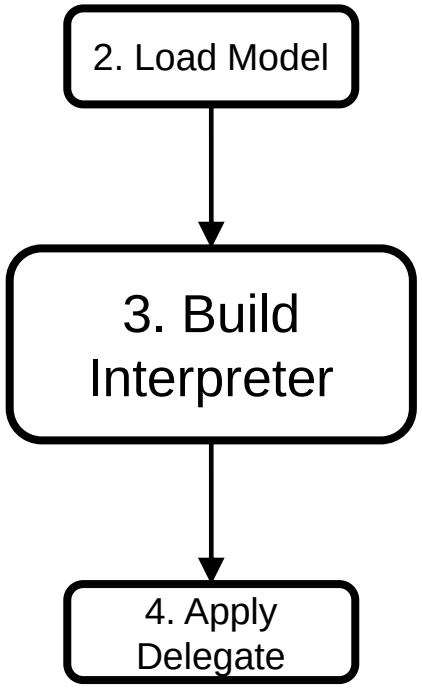
- Create an **`OpResolver`**
  - A registry that maps operator names to their corresponding implementations




```
75      /* Build interpreter */
76      tflite::ops::builtin::BuiltinOpResolver resolver;
77      tflite::InterpreterBuilder builder(*model, resolver);
78      std::unique_ptr<tflite::Interpreter> interpreter;
79      builder(&interpreter);
80      if (!interpreter)
81      {
82          std::cerr << "Failed to Initialize Interpreter" << std::endl;
83          return 1;
84      }
```

3. Build Interpreter (2)

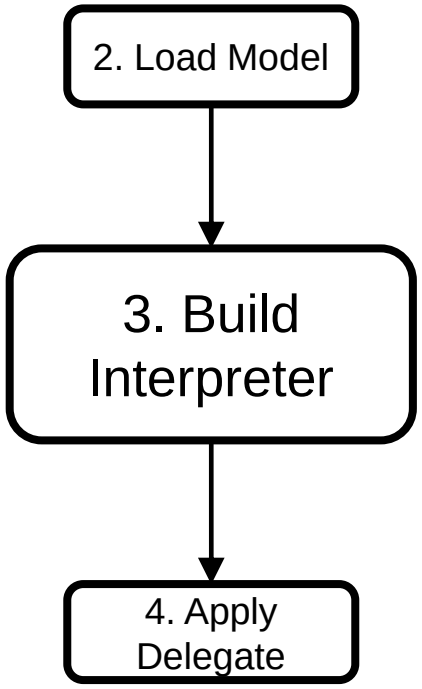
- Create an **`InterpreterBuilder`**
  - An object that constructs an interpreter instance using the builder pattern




```
75      /* Build interpreter */
76      tflite::ops::builtin::BuiltinOpResolver resolver;
77      tflite::InterpreterBuilder builder(*model, resolver);
78      std::unique_ptr<tflite::Interpreter> interpreter;
79      builder(&interpreter);
80      if (!interpreter)
81      {
82          std::cerr << "Failed to Initialize Interpreter" << std::endl;
83          return 1;
84      }
```

3. Build Interpreter (3)

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

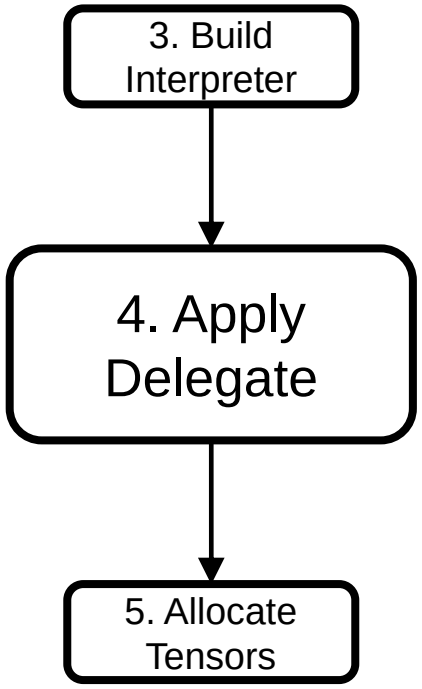



```
75      /* Build interpreter */
76      tflite::ops::builtin::BuiltinOpResolver resolver;
77      tflite::InterpreterBuilder builder(*model, resolver);
78      std::unique_ptr<tflite::Interpreter> interpreter;
79      builder(&interpreter);
80      if (!interpreter)
81      {
82          std::cerr << "Failed to Initialize Interpreter" << std::endl;
83          return 1;
84      }
```

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

4. Apply Delegate (1)

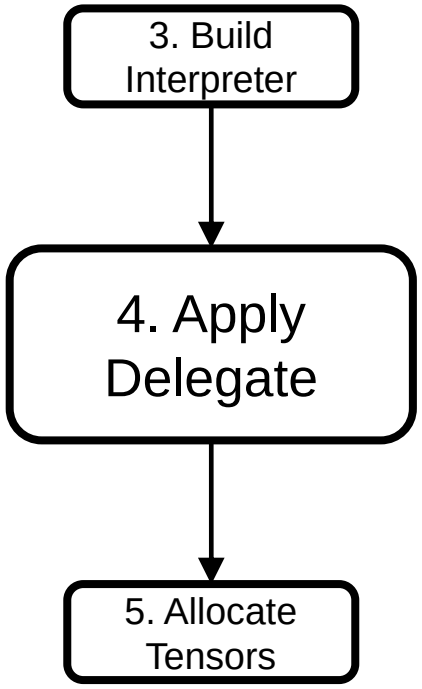
- Create delegate object(s)
  - For CPU execution optimization (XNNPACK) or GPU offloading




```
86      /* Apply either XNNPACK delegate or GPU delegate */
87      TfLiteDelegate* xnn_delegate = TfLiteXNNPackDelegateCreate(nullptr);
88      TfLiteDelegate* gpu_delegate = TfLiteGpuDelegateV2Create(nullptr);
```

4. Apply Delegate (2)

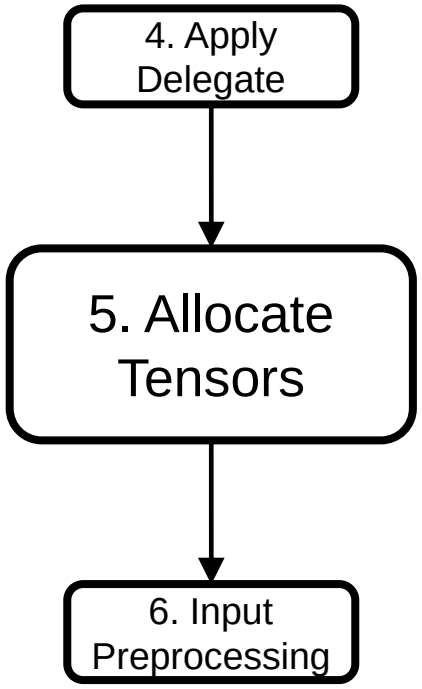
- Apply either the GPU delegate or the XNNPACK delegate
  - The delegate takes over supported parts of the model




```
89  if(gpu_usage) {
90      if (interpreter->ModifyGraphWithDelegate(gpu_delegate) == kTfLiteOk)
91      {
92          // Delete unused delegate
93          if(xnn_delegate) TfLiteXNNPackDelegateDelete(xnn_delegate);
94      } else {
95          std::cerr << "Failed to Apply GPU Delegate" << std::endl;
96      }
97  } else {
98      if (interpreter->ModifyGraphWithDelegate(xnn_delegate) == kTfLiteOk)
99      {
100         // Delete unused delegate
101         if(gpu_delegate) TfLiteGpuDelegateV2Delete(gpu_delegate);
102     } else {
103         std::cerr << "Failed to Apply XNNPACK Delegate" << std::endl;
104     }
105 }
```

5. Allocate Tensors

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

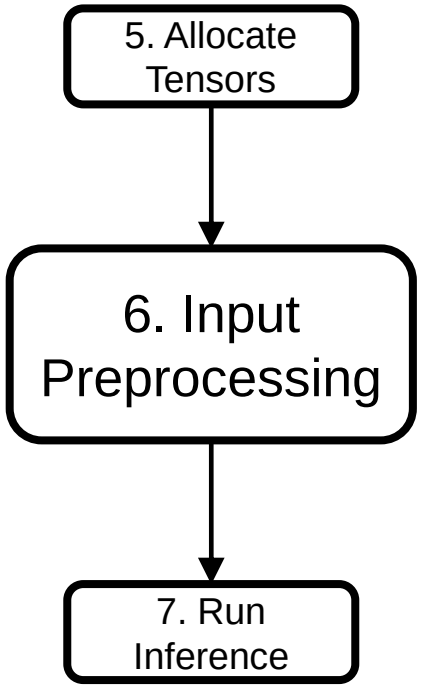



```
107     /* Allocate Tensors */
108     if (interpreter->AllocateTensors() != kTfLiteOk)
109     {
110         std::cerr << "Failed to Allocate Tensors" << std::endl;
111         return 1;
112     }
```

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

6. Input Preprocessing (1)

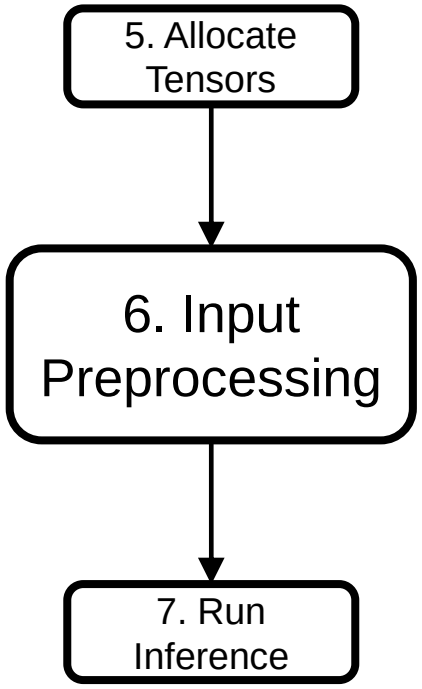
- Enters a do-while loop
  - Record the current time before starting, for use in periodic loop execution later




```
118     auto next_wakeup_time = std::chrono::high_resolution_clock::now();
119     do {
127         /* Preprocessing */
128         // Load input image
129         cv::Mat image = cv::imread(images[count]);
130         if (image.empty()) {
131             std::cerr << "Failed to load image: " << images[count] << "\n";
132             continue;
133         }
134
135         // Preprocess input data
136         cv::Mat preprocessed_image =
137                 util::preprocess_image_resnet(image, 224, 224);
138
139         // Copy preprocessed_image to input_tensor
140         float* input_tensor = interpreter->typed_input_tensor<float>(0);
141         std::memcpy(input_tensor, preprocessed_image.ptr<float>(),
142                     preprocessed_image.total() * preprocessed_image.elemSize());
```

6. Input Preprocessing (2)

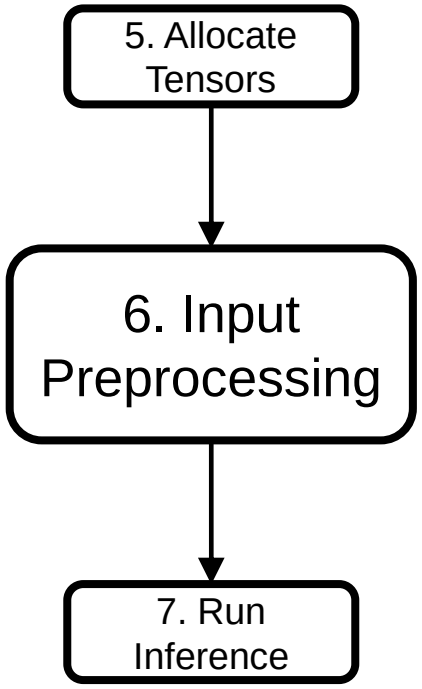
- Load an image




```
118     auto next_wakeup_time = std::chrono::high_resolution_clock::now();
119     do {
127         /* Preprocessing */
128         // Load input image
129         cv::Mat image = cv::imread(images[count]);
130         if (image.empty()) {
131             std::cerr << "Failed to load image: " << images[count] << "\n";
132             continue;
133         }
134
135         // Preprocess input data
136         cv::Mat preprocessed_image =
137                 util::preprocess_image_resnet(image, 224, 224);
138
139         // Copy preprocessed_image to input_tensor
140         float* input_tensor = interpreter->typed_input_tensor<float>(0);
141         std::memcpy(input_tensor, preprocessed_image.ptr<float>(),
142                 preprocessed_image.total() * preprocessed_image.elemSize());
```

6. Input Preprocessing (3)

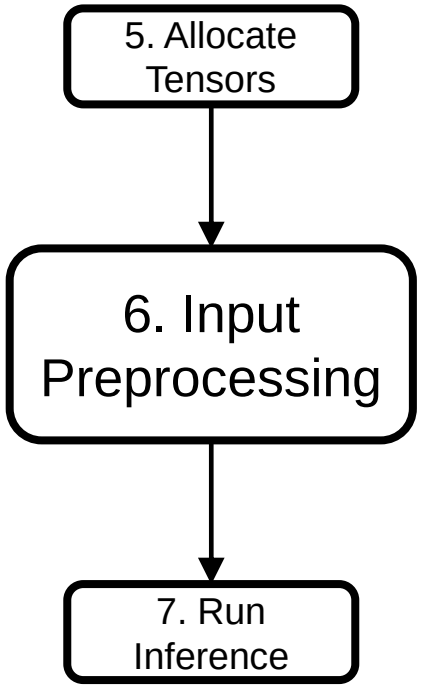
- Preprocess the loaded image




```
118     auto next_wakeup_time = std::chrono::high_resolution_clock::now();
119     do {
127         /* Preprocessing */
128         // Load input image
129         cv::Mat image = cv::imread(images[count]);
130         if (image.empty()) {
131             std::cerr << "Failed to load image: " << images[count] << "\n";
132             continue;
133         }
134
135         // Preprocess input data
136         cv::Mat preprocessed_image =
137                 util::preprocess_image_resnet(image, 224, 224);
138
139         // Copy preprocessed_image to input_tensor
140         float* input_tensor = interpreter->typed_input_tensor<float>(0);
141         std::memcpy(input_tensor, preprocessed_image.ptr<float>(),
142                     preprocessed_image.total() * preprocessed_image.elemSize());
```

6. Input Preprocessing (4)

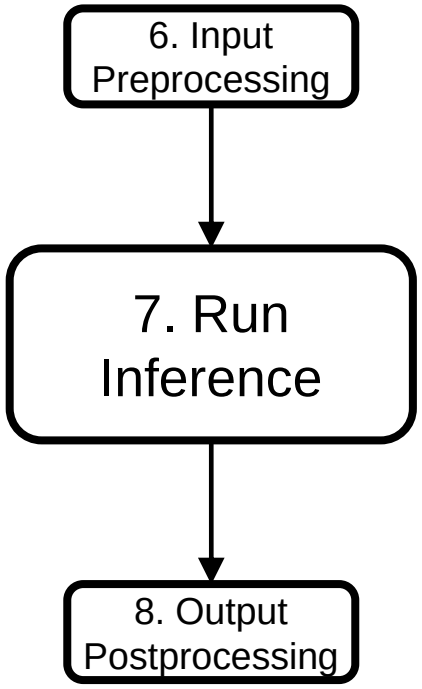
- Access the 0th input tensor as a float pointer and set its value
  - There is only one input tensor in ResNet50 FP32




```
118     auto next_wakeup_time = std::chrono::high_resolution_clock::now();
119     do {
127         /* Preprocessing */
128         // Load input image
129         cv::Mat image = cv::imread(images[count]);
130         if (image.empty()) {
131             std::cerr << "Failed to load image: " << images[count] << "\n";
132             continue;
133         }
134
135         // Preprocess input data
136         cv::Mat preprocessed_image =
137                 util::preprocess_image_resnet(image, 224, 224);
138
139         // Copy preprocessed_image to input_tensor
140         float* input_tensor = interpreter->typed_input_tensor<float>(0);
141         std::memcpy(input_tensor, preprocessed_image.ptr<float>(),
142                     preprocessed_image.total() * preprocessed_image.elemSize());
```

7. Run Inference

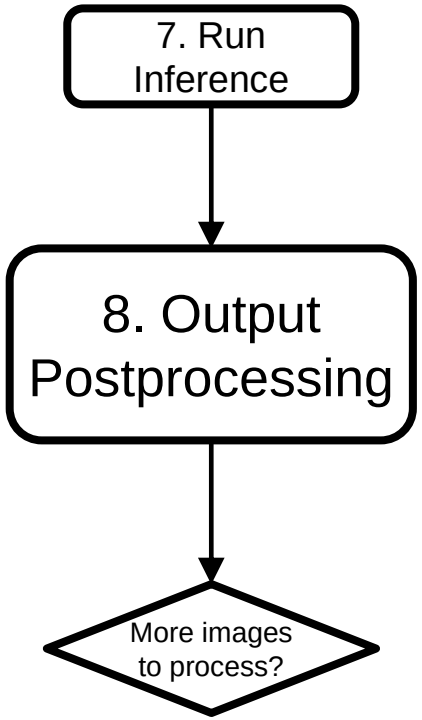
- Invoke the interpreter to run inference




```
147         /* Inference */
148         if (interpreter->Invoke() != kTfLiteOk)
149         {
150             std::cerr << "Failed to invoke the interpreter" << std::endl;
151             return 1;
152         }
```

8. Output Postprocessing (1)

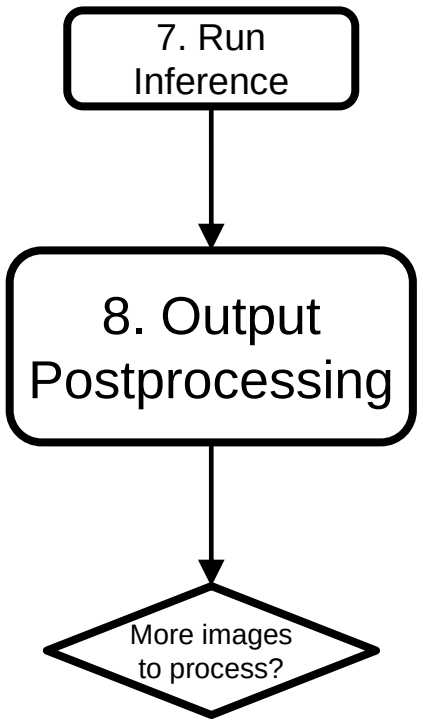
- Access the 0th output tensor as a float pointer
  - ResNet50 has only one output tensor with a shape of 1×1000




```
156     /* PostProcessing */
157     // Get output tensor
158     float *output_tensor = interpreter->typed_output_tensor<float>(0);
159     int num_classes = 1000; // Total 1000 classes
160     std::vector<float> probs(num_classes);
161     std::memcpy(probs.data(), output_tensor, sizeof(float) * num_classes);
```

8. Output Postprocessing (2)

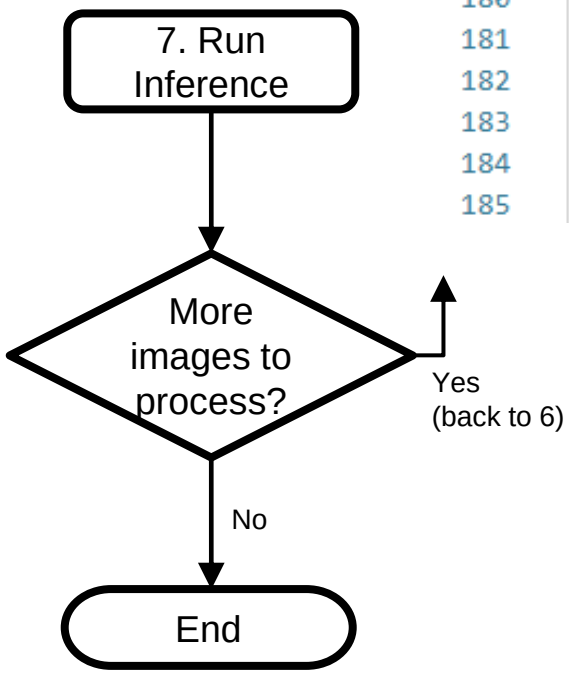
- Print the top 3 predictions with their labels




```
163         // Print Top-3 predictions every 10 iterations
164         if ((count + 1) % 10 == 0) {
165             std::cout << "\n[INFO] Top 3 predictions for image index " << count << ":"
166             << std::endl;
167             auto top_k_indices = util::get_topK_indices(probs, 3);
168             for (int idx : top_k_indices)
169             {
170                 std::string label =
171                     class_labels_map.count(idx) ? class_labels_map[idx] : "unknown";
172                 std::cout << "- Class " << idx << " (" << label << "): "
173                     << probs[idx] << std::endl;
174             }
175         }
```

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




```
180         // Sleep to control the input rate
181         // If next_wakeup_time is in the past, it will not sleep
182         next_wakeup_time += std::chrono::milliseconds(input_period_ms);
183         std::this_thread::sleep_until(next_wakeup_time);
184         ++count;
185     } while (count < images.size());
```

Contents

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 **`~/DNNPipe-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

Two Key Entities of LiteRT

1. Model object
   - Static description of the model stored in a .tflite file
2. Interpreter
   - Runtime instance built from the model object

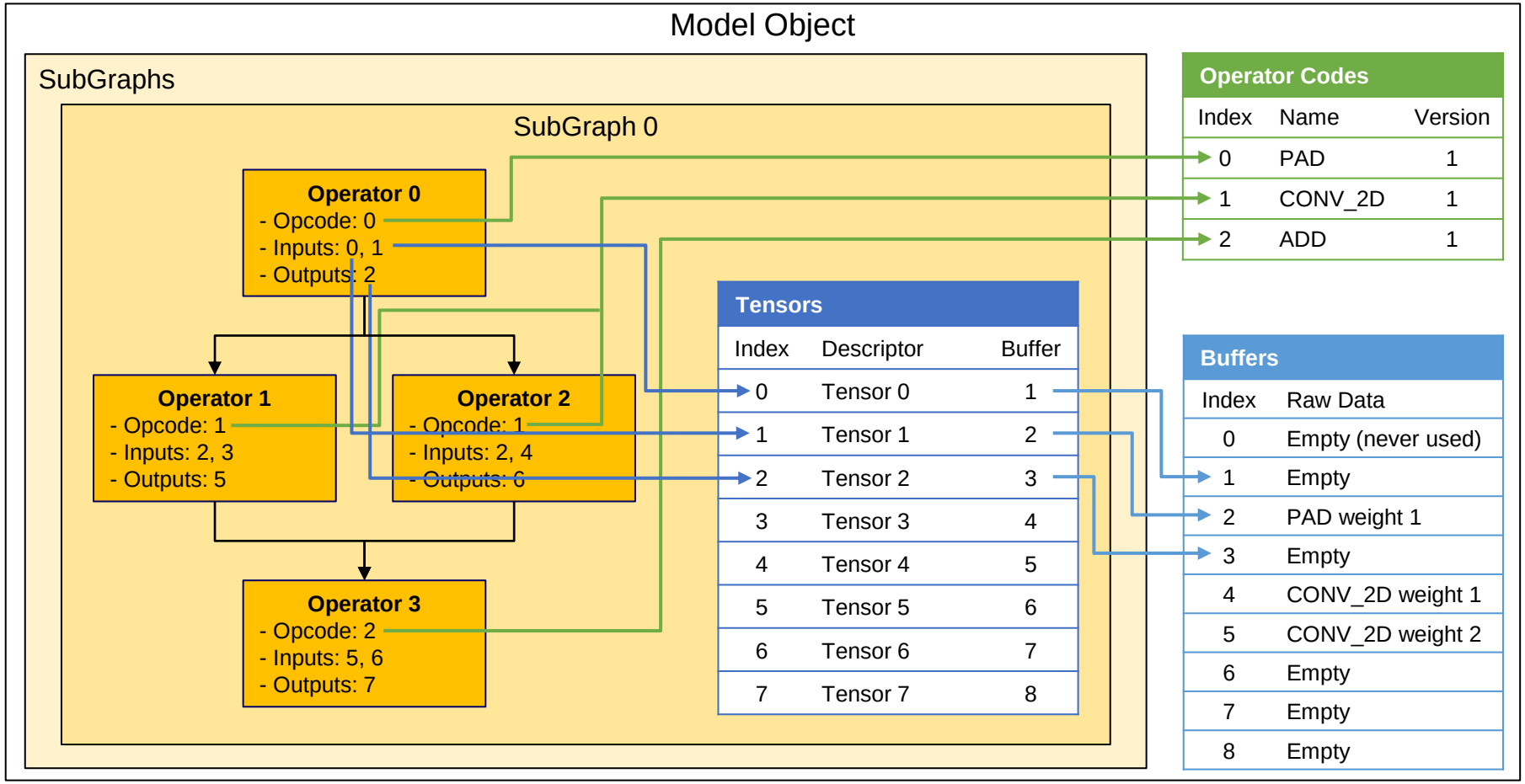
1. Model Object (1)

1. SubGraph
   - Computation graph with its own operators and tensors
     - Multiple subgraphs are used to support advanced features such as control flow and modular function calls
2. Operator
   - Basic unit of neural operations
   - Each operator references an operator code and tensors it uses
3. Operator code
   - Identifier that defines the type of neural operation

1. Model Object (2)

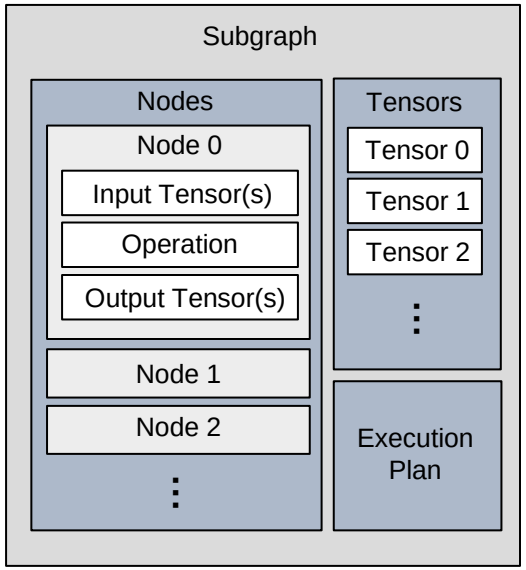
4. Tensor
   - Descriptor of a multi-dimensional array instantiated at runtime
     - Defines name, shape, data type, and buffer reference
5. Buffer
   - Container of raw constant data referenced by immutable tensors
     - e.g., weights and biases
     - Empty for mutable tensors

1. Model Object (3)



2. Interpreter

1. Subgraph
   - Computation graph created by interpreter builder
     from the SubGraph (nodes, tensors, and execution plan) of the model object
2. Node
   - Runtime unit of computation
     corresponding to an operator in the model object
3. Tensor
   - Multi-dimensional array that holds
     inputs, outputs, weights, biases, and/or intermediate results
4. Execution plan
   - Ordered list of node indices
   - Defines the sequence of execution for the subgraph



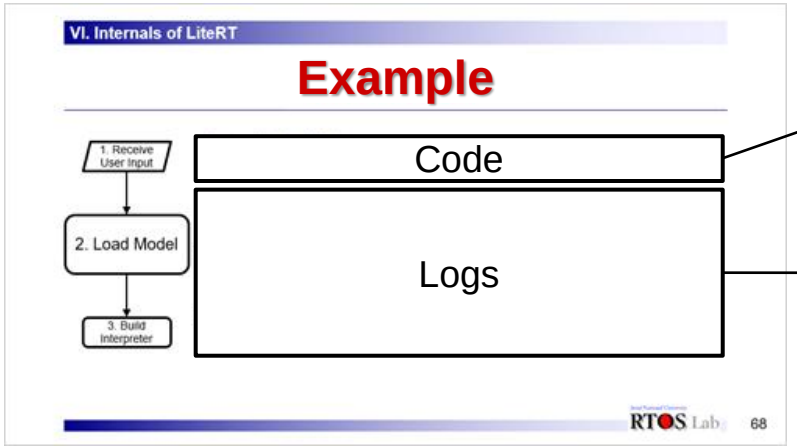
Contents

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
- To understand the internals of high-level LiteRT APIs,
  - Follow the code flow of the inference driver and
  - Visualize logs that the implementation harness collects

Run the Instrumentation Harness

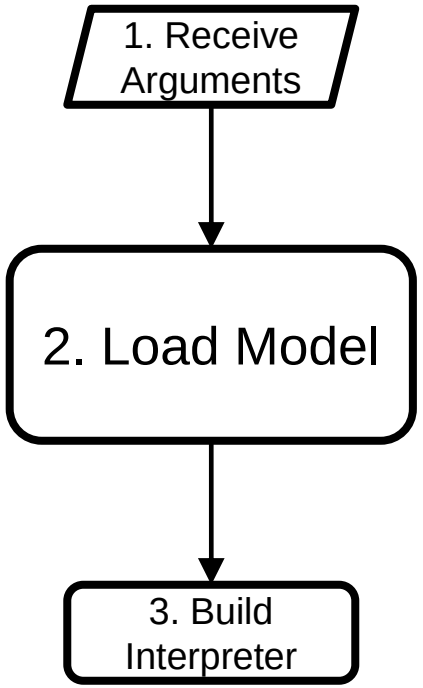
- In **`~/DNNPipe-Tutorial`**, run
  - **`./bin/instrumentation_harness ./models/resnet50.tflite`**
  - You will see an indicator `Press Enter to continue...` at each step
    - Let's go through the steps together to observe what happens internally



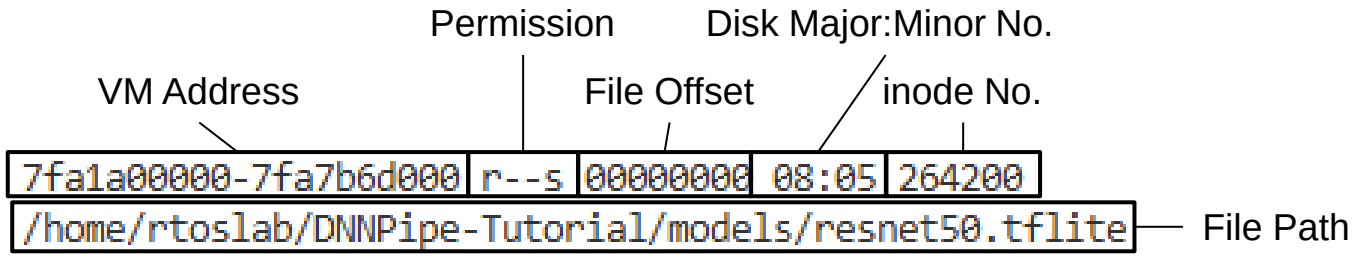
High-level LiteRT APIs invoked by the inference driver

Logs of the instrumentation harness and its explanation

2. Load Model

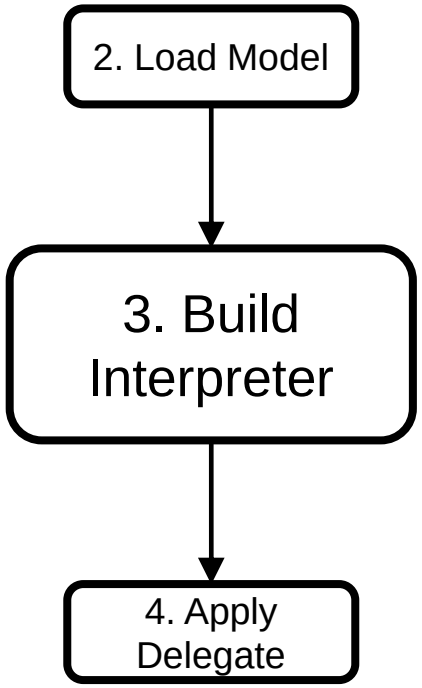



```
66    /* Load model */
67    std::unique_ptr<tflite::FlatBufferModel> model =
68        tflite::FlatBufferModel::BuildFromFile(model_path.c_str());
```



- 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

3. Build Interpreter (1)



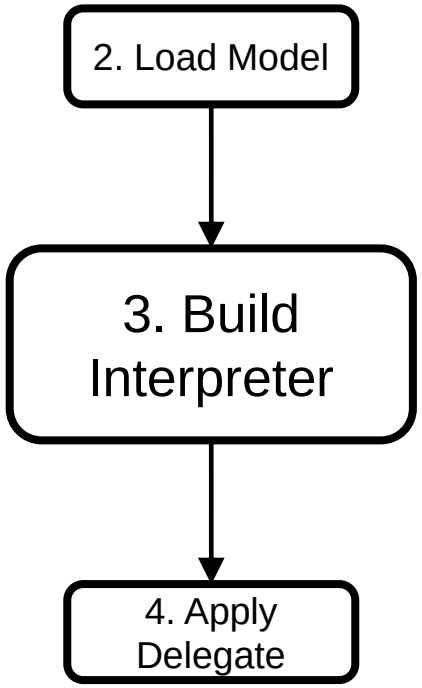

```
75    /* Build interpreter */
76    tflite::ops::builtin::BuiltinOpResolver resolver;
77    tflite::InterpreterBuilder builder(*model, resolver);
78    std::unique_ptr<tflite::Interpreter> interpreter;
79    builder(&interpreter);
```

```
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
  `~/DNNPipe-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

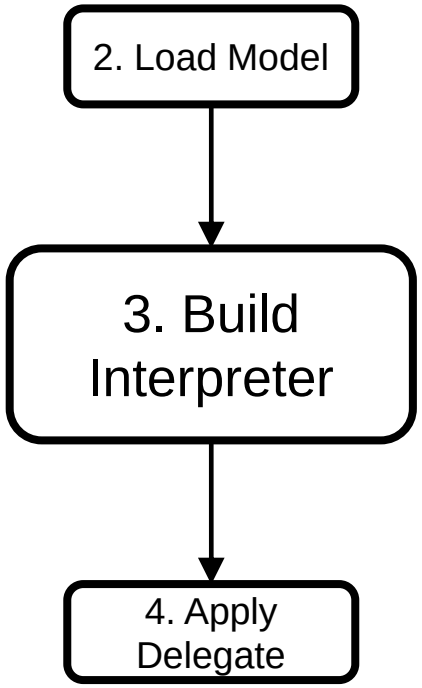
3. Build Interpreter (2)




```
Total 7 operator codes 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 operator code in the model and checks if the OpResolver supports it
- If any operator code is not supported, an error is raised
  - Custom operators are handled in a more complicated way

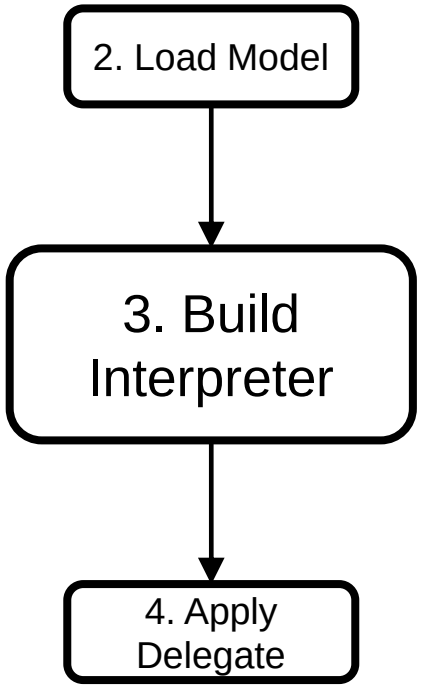
3. Build Interpreter (3)




```
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

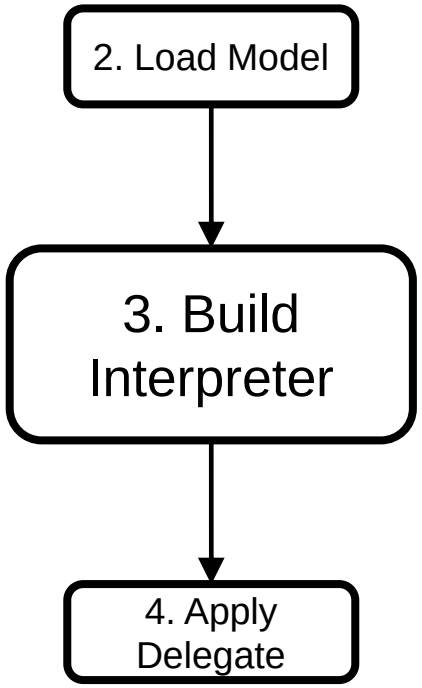
3. Build Interpreter (4)




```
Total 75 nodes 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

3. Build Interpreter (5)

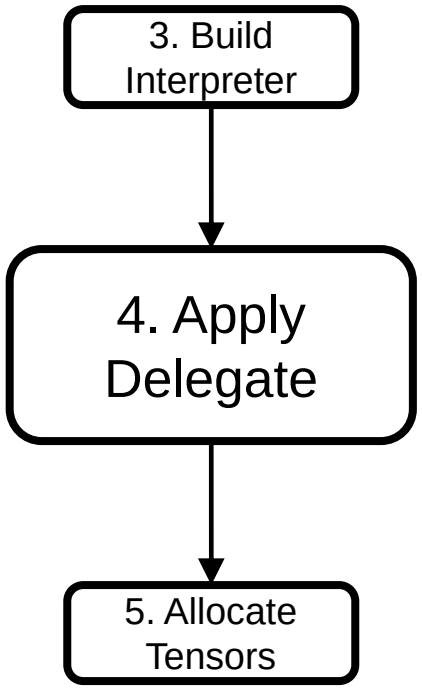



```
Number of subgraphs: 1
Number of nodes of subgraph 0: 75
Number of tensors in subgraph 0: 187
Execution plan size of subgraph 0: 75
```

- 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
```

4. Apply Delegate




```
98        if (interpreter->ModifyGraphWithDelegate(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

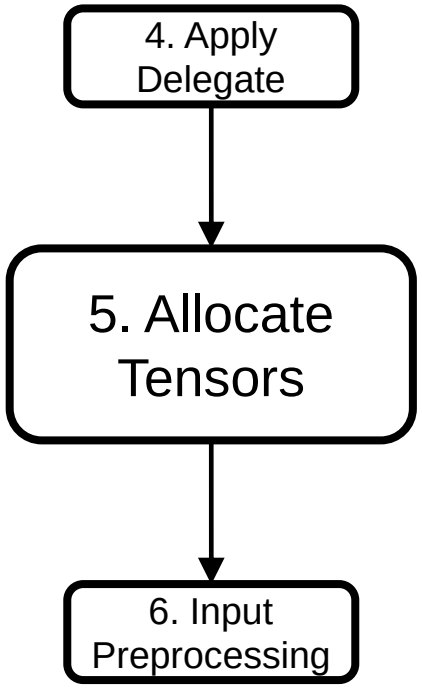
Inputs:
0 (type: FLOAT32, dims: [1, 224, 224, 3])
1 (type: FLOAT32, dims: [64])
2 (type: FLOAT32, dims: [256])
                ...
110 (type: INT32, dims: [4, 2])
111 (type: FLOAT32, dims: [1000, 2048])

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

5. Allocate Tensors



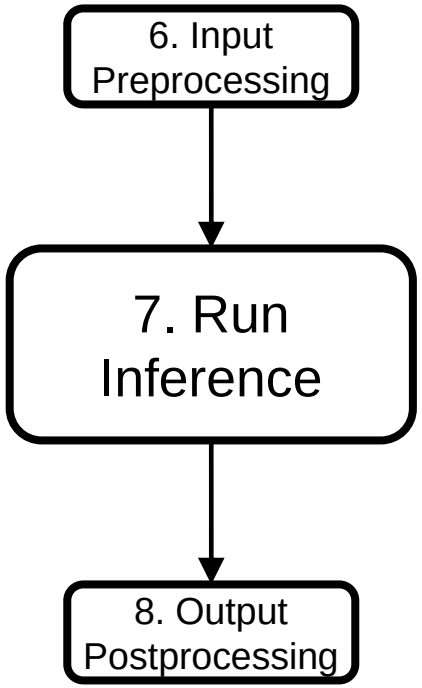

```
107     /* Allocate Tensors */
108     if (interpreter->AllocateTensors() != kTfLiteOk)
109     {
110         std::cerr << "Failed to Allocate Tensors" << std::endl;
111         return 1;
112     }
```

```
==== 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
```

7. Run Inference




```
147         /* Inference */
148         if (interpreter->Invoke() != kTfLiteOk)
149         {
150             std::cerr << "Failed to invoke the interpreter" << std::endl;
151             return 1;
152         }
```

```
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

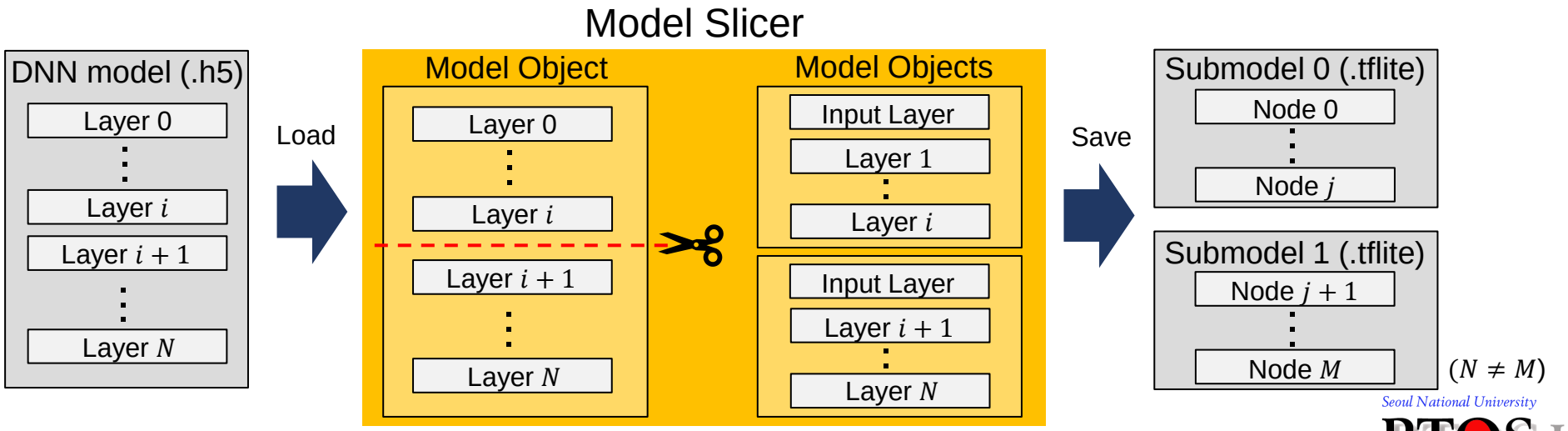
Tutorial Outline

Lecture 1: *Exercise Overview and Setup* (1 h 30 min)	
Lecturer	Seongsoo Hong
Topics	Motivating Example
	Development Environment Setup
Coffee Break (30 min)	
Lecture 2: *From Inference Driver to Inference Runtime* (1 h 30 min)	
Lecturer	Seongsoo Hong and Namcheol Lee
Topics	Step-by-Step Inference Driver Walkthrough
	Internals of LiteRT
Lunch Break (1 h)	
Lecture 3: *Model Slicer* (1 h 30 min)	
Lecturer	Seongsoo Hong
Topic	Model Slicer: Slicing and Conversion Tool for LiteRT
Coffee Break (30 min)	
Lecture 4: *Throughput Enhancement on Heterogeneous Accelerators* (1 h 30 min)	
Lecturer	Namcheol Lee
Topic	Implementing a Pipelined Inference Driver for Heterogeneous Processors

Contents

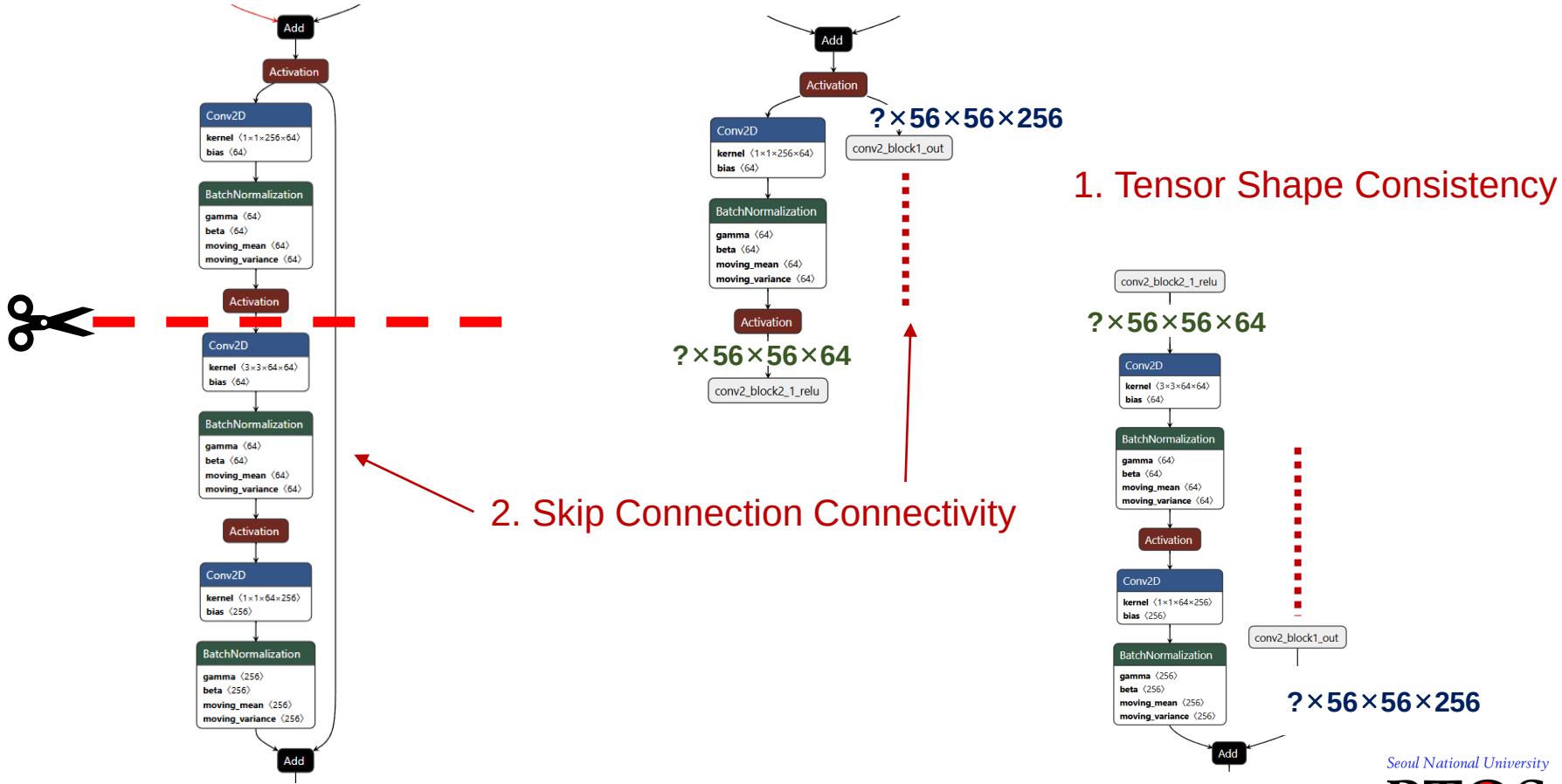
Model Slicing for DNN Pipelining (1)

- 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



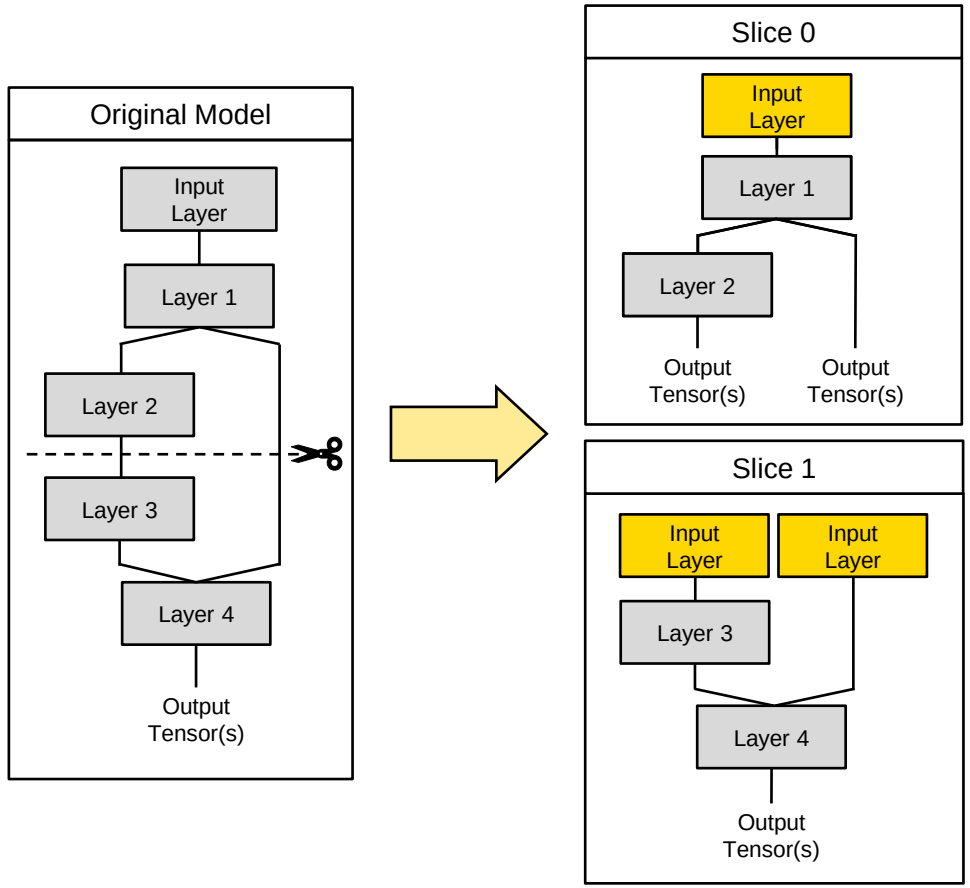
Model Slicing for DNN Pipelining (2)

- Characteristics to be preserved during slicing



Model Slicing for DNN Pipelining (3)

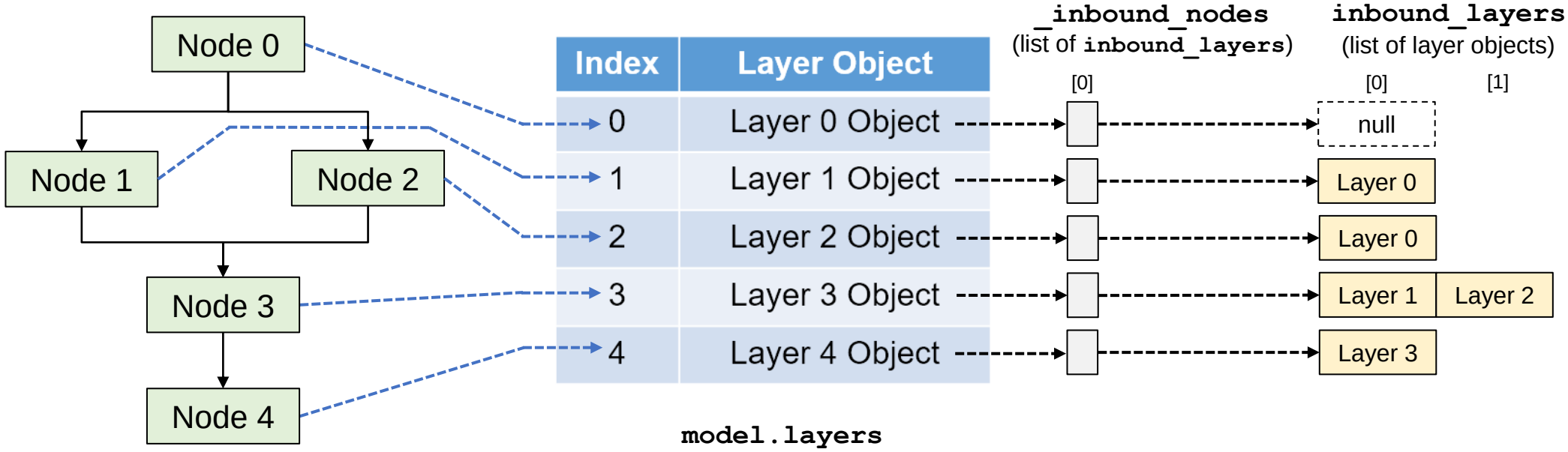
- Input layers are newly created
  - Per output (tensor(s)) of the previous slice



Contents

1. 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



1. 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

1. 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)

2. How the 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

2-1. 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)
```

2-1. 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

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

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

2-2. 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

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

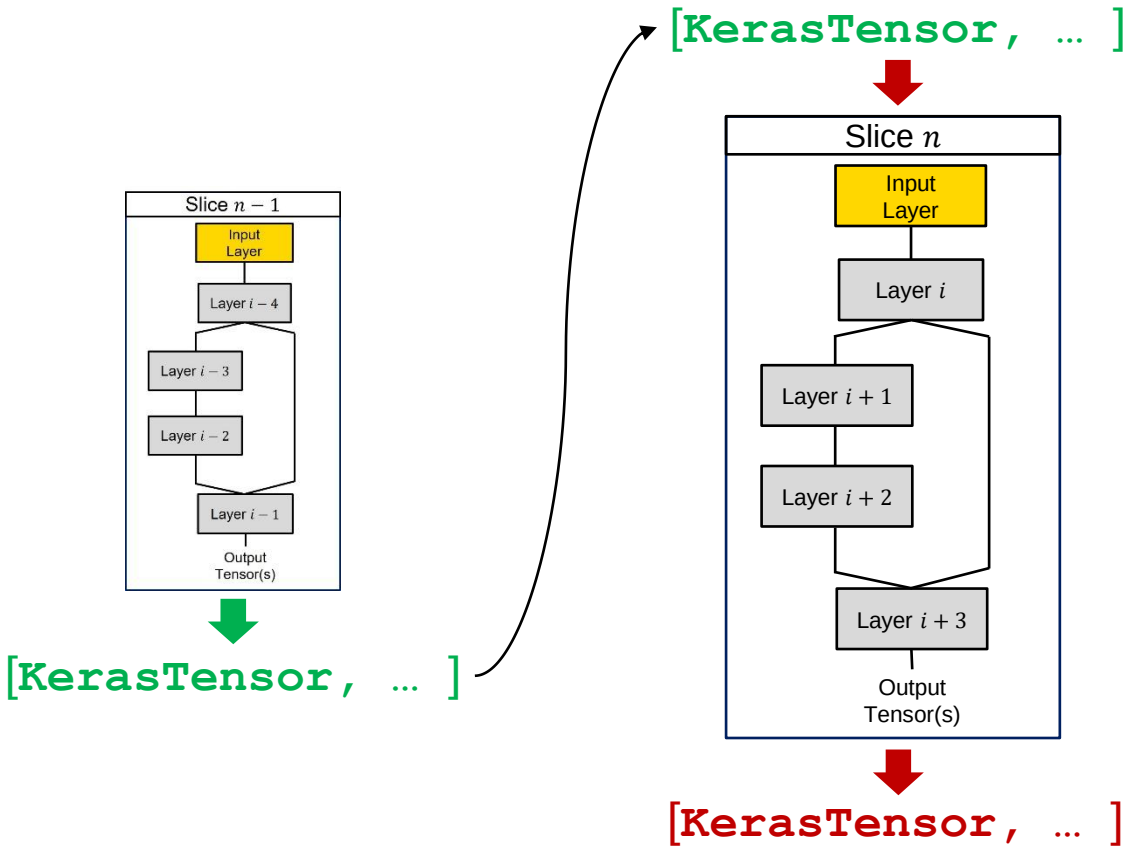
7 Cases for the Model Slicer

❖ 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

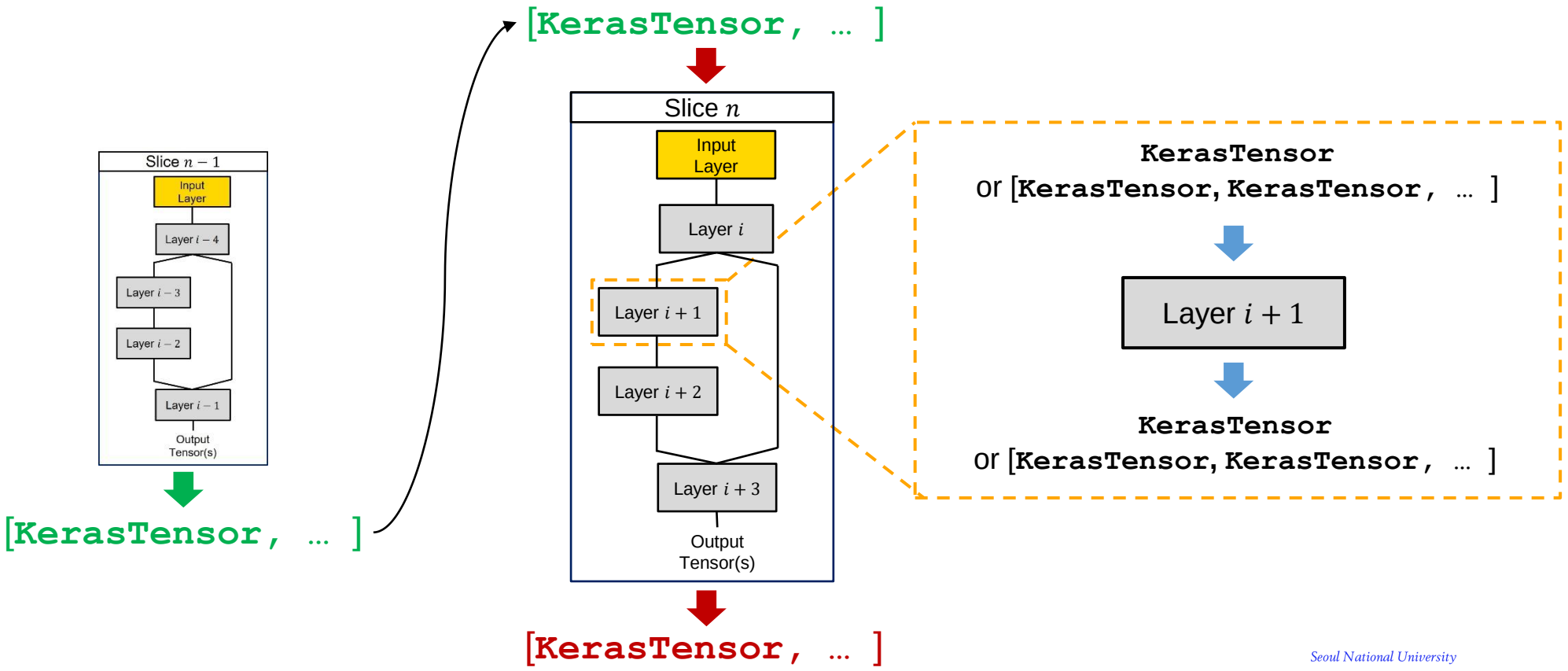
Priori Knowledge: 1. Input and Output of Slice

- A list containing one or more Keras tensors



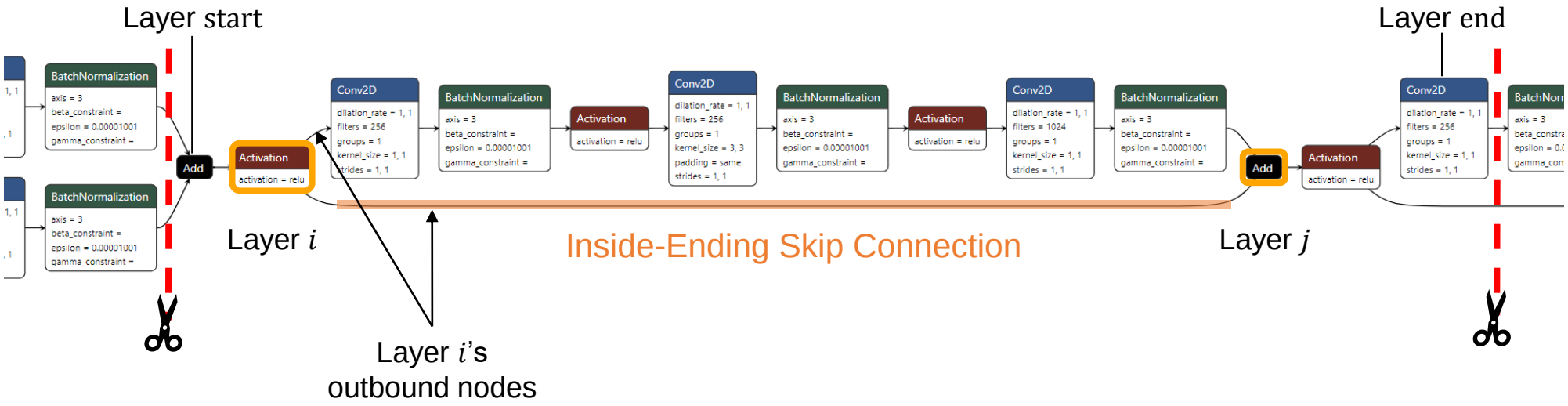
Priori Knowledge: 2. Input and Output of Layer

- Either a single Keras tensor or a list containing multiple Keras tensors



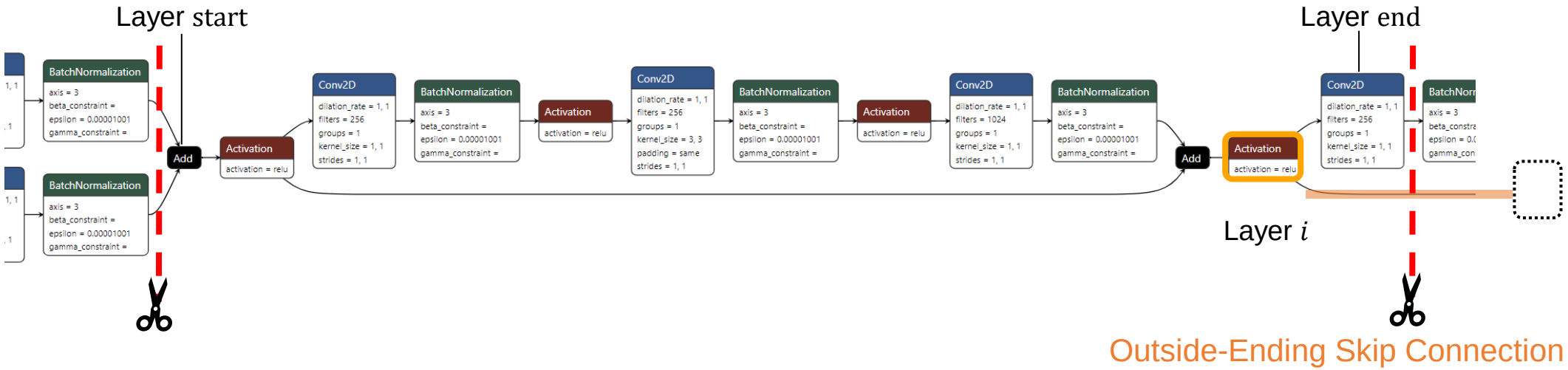
Priori Knowledge: 3. Inside-Ending Skip Connections

- 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 $j \leq \text{end}$



Priori Knowledge: 4. Outside-Ending Skip Connections

- 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 $j > \text{end}$



Case 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
    ```

Case 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])`

Case 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]`

Case 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]`

Case 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

Case 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_layer`

Case 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_layer`

Code for Cases 1, 2, 3, and 4

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

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

```
28  def slice_dnn(model, start, end, input_tensors):
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_layer = tensors_to_start_layer[0]
85      else:
86          tensors_to_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          # Skip the layer if it is an InputLayer
93          if isinstance(layer, tf.keras.layers.InputLayer):
94              continue
95
96          # Build current (i-th) layer
97          if isinstance(origin_inbound_layers, list): # When current layer expects multiple inputs (list of KerasTensors)
98              if(i == start):
99                  tensors_from_layer = layer(tensors_to_layer)
100             else:
101                 # Check if layer i's output tensor(s) is used in layer i+1
102                 if model.layers[i-1] in origin_inbound_layers:
103                     tensors_to_layer = [tensors_to_layer] if not isinstance(tensors_to_layer, list) else tensors_to_layer
104                 else:
105                     tensors_to_layer = []
```

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

```
107                 # From inbound layers, collect the required inside-ending skip tensors
108                 for origin_inbound_layer in origin_inbound_layers:
109                     inside_ending_skip = model.get_layer(origin_inbound_layer.name)
110                     if inside_ending_skip.name not in [t.name.split('/')[0] for t in tensors_to_layer]:
111                         tensors_to_layer.append(inside_ending_skips[inside_ending_skip.name])
112 
113                 # Call the functor of the current layer to build a new layer
114                 try:
115                     tensors_from_layer = \
116                         layer(tensors_to_layer)
117                 except: # When a custom layer's call signature deviates from Keras expectations
118                     raise ValueError(f"Failed to call layer {layer.name} with tensors {tensors_to_layer}. "
119                                      "Please check the layer's call function and the input tensors.")
120             else: # When current layer expects a single input (KerasTensor)
121                 if origin_inbound_layers.name in inside_ending_skips:
122                     tensors_from_layer = layer(inside_ending_skips[origin_inbound_layers.name])
123                 else:
124                     tensors_from_layer = layer(tensors_to_layer)
125 
126             # Update inside_ending_skips and outside_ending_skips based on the current layer's outbound nodes
127             if i < end: # Ensure the current layer is not the slice's last layer
128                 for origin_outbound_node in layer._outbound_nodes:
129                     skip_target_idx = model.layers.index(origin_outbound_node.outbound_layer)
130                     if skip_target_idx <= end and skip_target_idx > i + 1:
131                         inside_ending_skips[layer.name] = tensors_from_layer
132                     elif skip_target_idx > end:
133                         outside_ending_skips[layer.name] = tensors_from_layer
134 
135             tensors_to_layer = tensors_from_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.

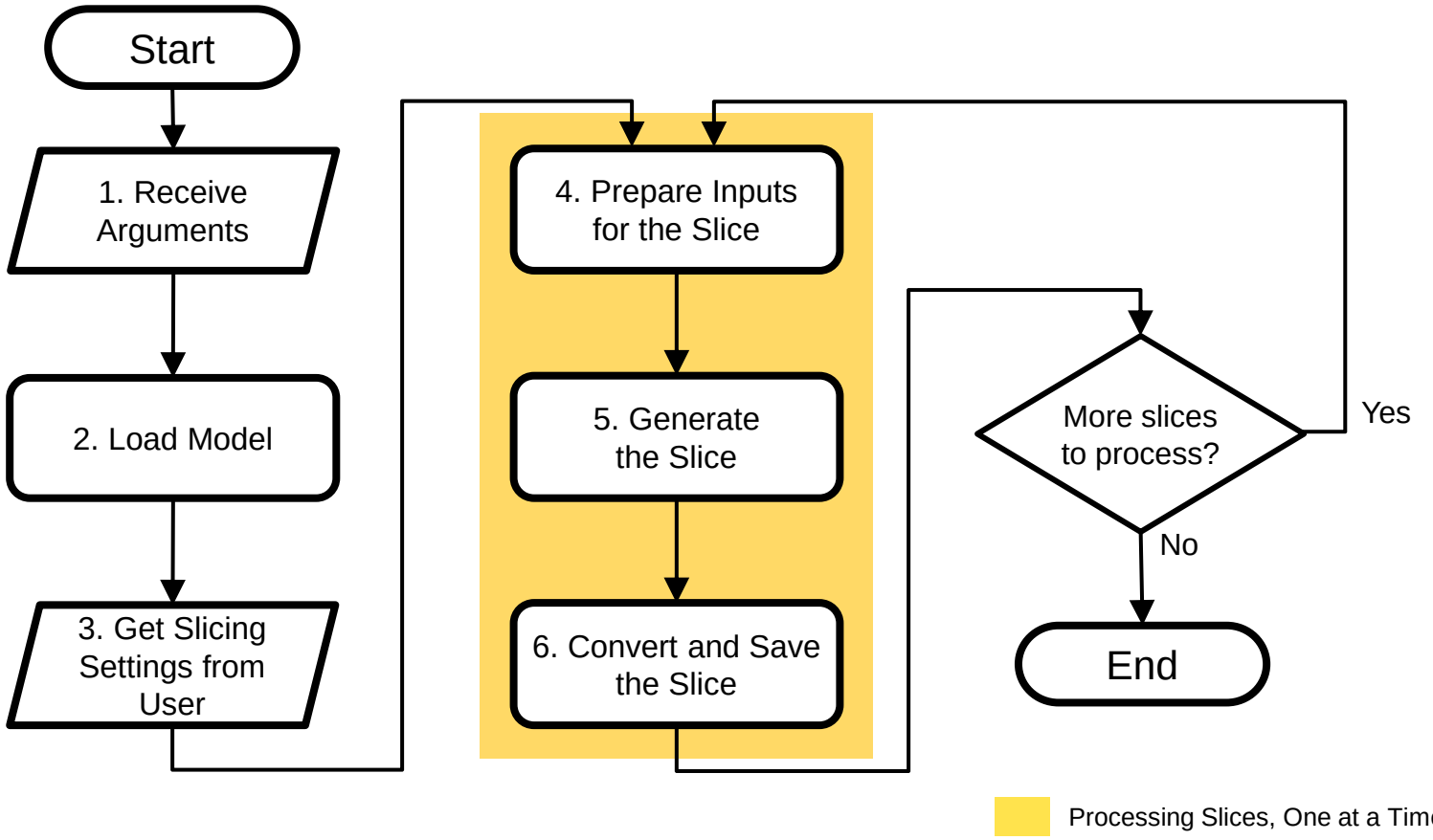
To address these challenges, pipeline parallelization has emerged as a promising solution. It not only reduces per-device memory requirements, but also enables the efficient execution of DNN inference on resource-constrained IoT devices.

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].

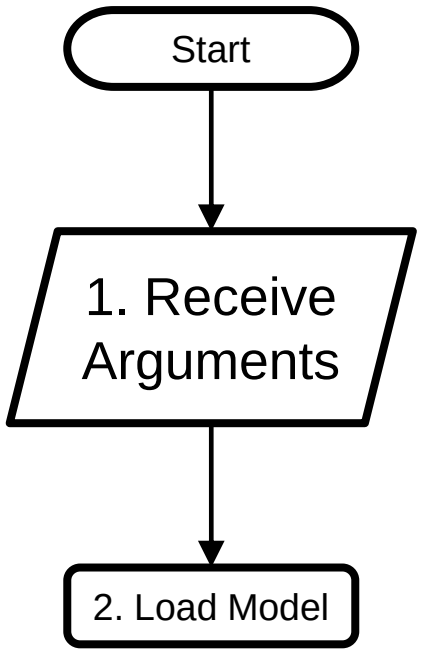
Woobean Seo, Saehwa Kim, and Seongsoo Hong, https://doi.org/10.1016/j.sysarc.2025.103462."DNNPipe: Dynamic Programming–based Optimal DNN Partitioning for Pipelined Inference on IoT Networks," Journal of Systems Architecture, vol. 166, 2025, 103462, ISSN 1383–7621,

Model Slicer Code Flow



1. Receive Arguments

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

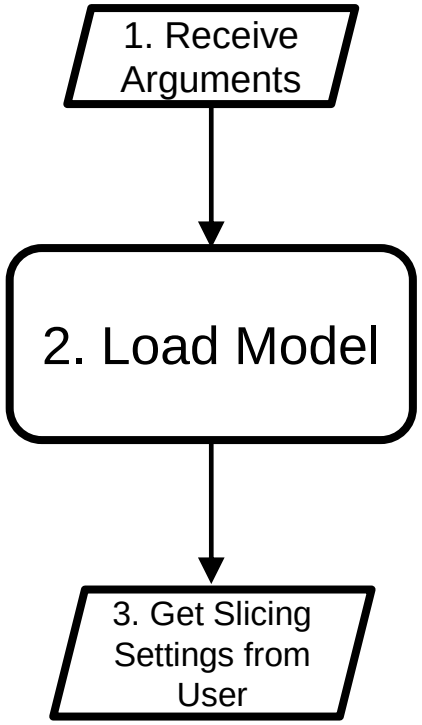



```
239  def main():
240      # Parse command line arguments
241      args = parse_arguments()
```

```
163  # Parse command line arguments
164  def parse_arguments():
165      parser = argparse.ArgumentParser()
166      parser.add_argument('--model-path', type=str, required=True, help='Path to a model file (.h5)')
167      parser.add_argument('--output-dir', type=str, default='./models')
168      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

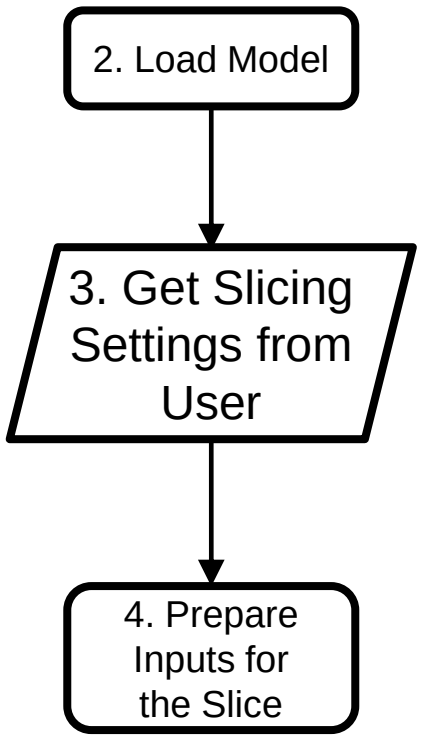



```
16  import tensorflow as tf
17  import numpy as np
18  import os
19  from tensorflow.keras.models import load_model
20  import argparse
```

```
243     # Load the model from the given path without compilation (for inference/slicing only)
244     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

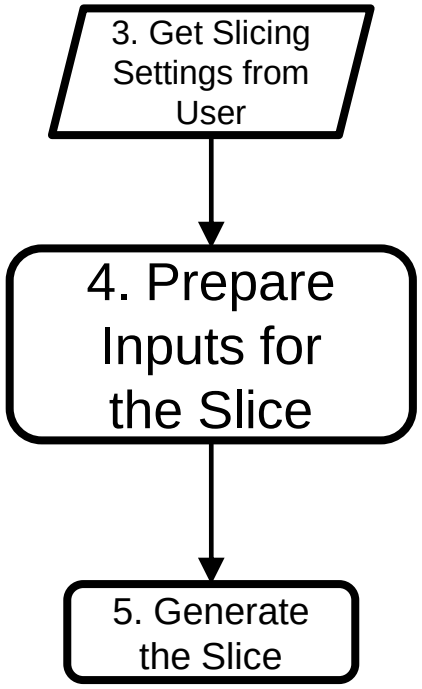



```
246     # Ask the user for the number of slices and the index of the last layer in each slice
247     num_layers = len(model.layers)
248     num_slices, starts = get_slice_starts(num_layers)
```

```
(.venv) rtoslab@RUBIKPi:~/DNNPipe-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: 89
Layer index ranges for each submodel: [(1, 88), (89, 176)]
Saved LiteRT model to: ./models/submodel_0.tflite
Saved LiteRT model to: ./models/submodel_1.tflite
```

4. Prepare Inputs for the Slice (1)

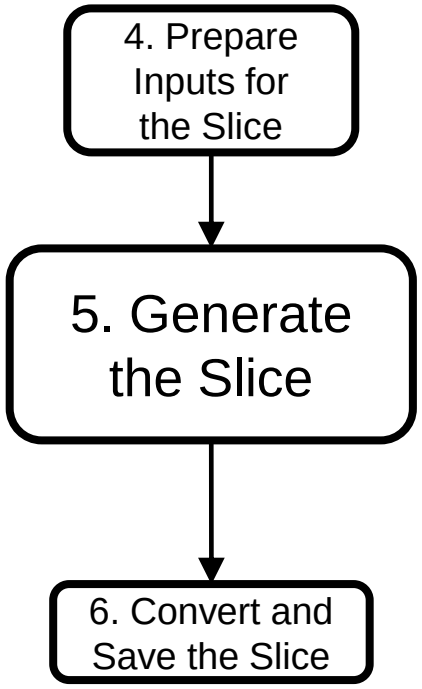
- List of Keras tensor(s) is created for the input to the slice




```
250     # Perform slicing and conversion
251     slices = []
252     slice_inputs = []
253     for i in range(num_slices):
254         # Prepare inputs for each slice
255         if i == 0:
256             slice_inputs = model.inputs
257         else:
258             slice_inputs = slices[i-1].outputs
259
260         # Slice the model using slice_dnn
261         slice = slice_dnn(model, starts[i], starts[i+1]-1, slice_inputs)
262         slices.append(slice)
263
264         # Convert and save the slice to a LiteRT model
265         convert_save_slice(args.output_dir, slice, i)
```

5. Generate the Slice

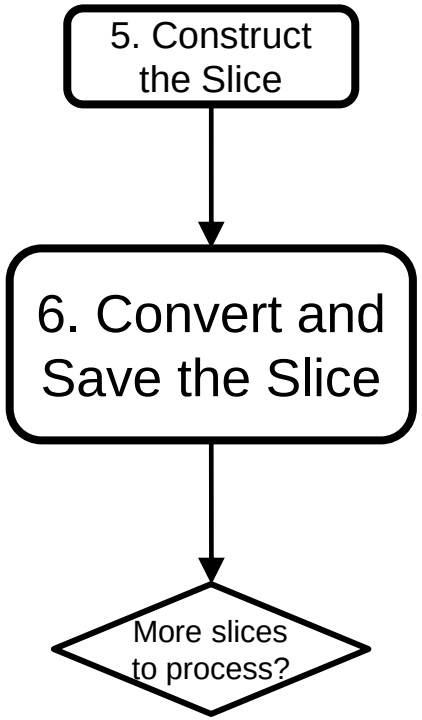
- For the given model, create a slice, one at a time
  - According to user-provided slicing settings




```
250     # Perform slicing and conversion
251     slices = []
252     slice_inputs = []
253     for i in range(num_slices):
254         # Prepare inputs for each slice
255         if i == 0:
256             slice_inputs = model.inputs
257         else:
258             slice_inputs = slices[i-1].outputs
259
260         # Slice the model using slice_dnn
261         slice = slice_dnn(model, starts[i], starts[i+1]-1, slice_inputs)
262         slices.append(slice)
263
264         # Convert and save the slice to a LiteRT model
265         convert_save_slice(args.output_dir, slice, i)
```

6. Convert and Save the Slice

- Convert and save the slice into a LiteRT model




```
250     # Perform slicing and conversion
251     slices = []
252     slice_inputs = []
253     for i in range(num_slices):
254         # Prepare inputs for each slice
255         if i == 0:
256             slice_inputs = model.inputs
257         else:
258             slice_inputs = slices[i-1].outputs
259
260         # Slice the model using slice_dnn
261         slice = slice_dnn(model, starts[i], starts[i+1]-1, slice_inputs)
262         slices.append(slice)
263
264         # Convert and save the slice to a LiteRT model
265         convert_save_slice(args.output_dir, slice, i)
```

Contents

Run the 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)

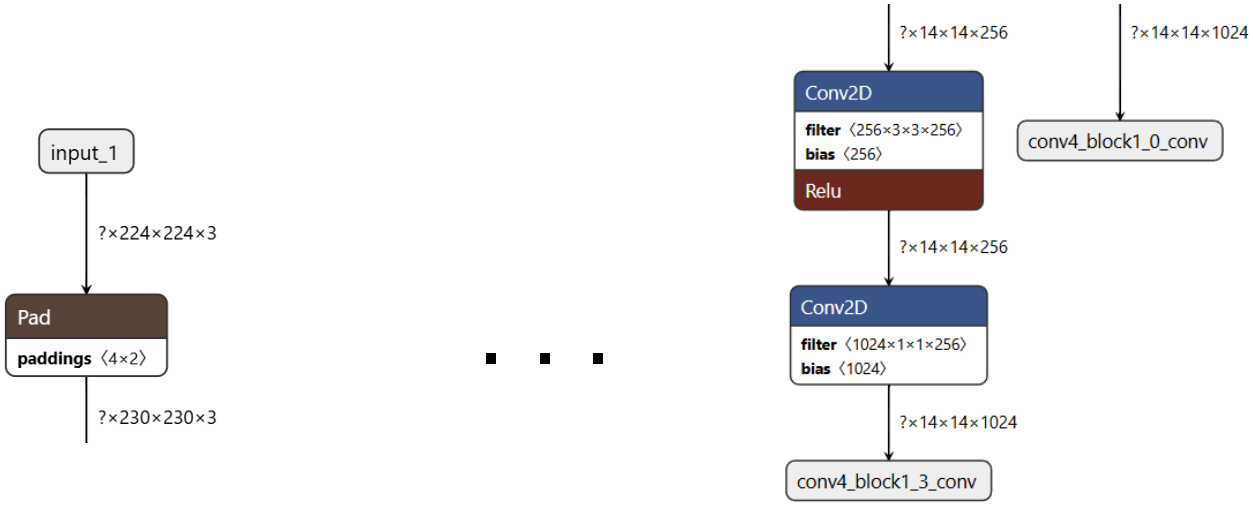
- Command to slice the target DNN model (ResNet50)
  - **`$ python model_slicer.py --model-path ./models/resnet50.h5`**

- For lecture 4, slice ResNet50 into two submodels
  - **`submodel_0`**: From layer 1 to layer x1 – 1
  - **`submodel_1`**: From layer x1 to layer 176
  - Choose x1 freely

```
(.venv) rtoslab@RUBIKPi:~/DNNPipe-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: 89
Layer index ranges for each submodel: [(1, 88), (89, 176)]
Saved LiteRT model to: ./models/submodel_0.tflite
Saved LiteRT model to: ./models/submodel_1.tflite
```

Slicing ResNet50 (2)

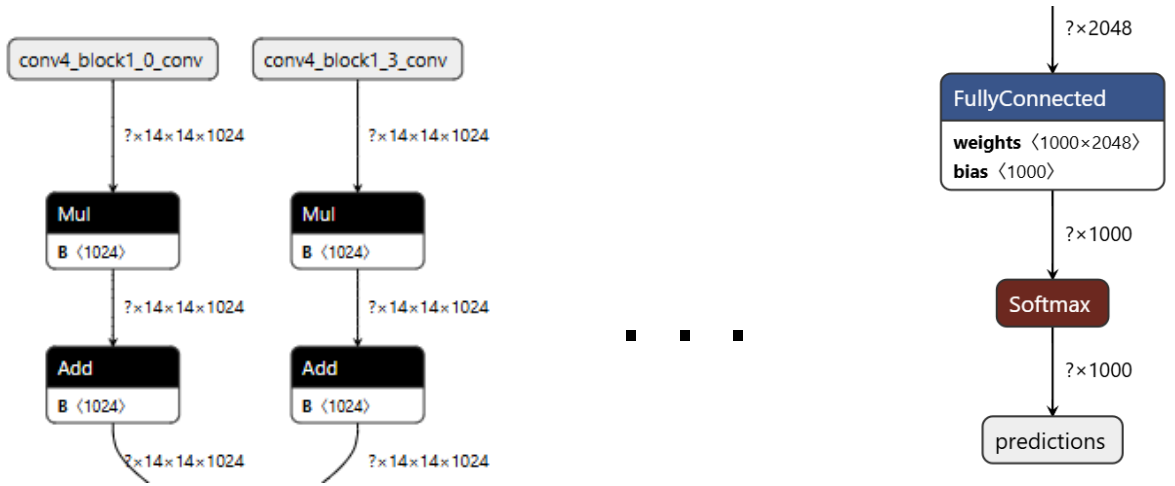
- `netron ./models/submodel_0.tflite`



submodel	Input	Output	Saved As
0	input_1	**conv4_block1_3_conv, conv4_block1_0_conv**	submodel_0.tflite
1	**conv4_block1_3_conv, conv4_block1_0_conv**	Predictions (final output)	submodel_1.tflite

Slicing ResNet50 (3)

- `netron ./models/submodel_1.tflite`



submodel	Input	Output	Saved As
0	input_1	**conv4_block1_3_conv, conv4_block1_0_conv**	submodel_0.tflite
1	**conv4_block1_3_conv, conv4_block1_0_conv**	Predictions (final output)	submodel_1.tflite

Tutorial Outline

Lecture 1: *Exercise Overview and Setup* (1 h 30 min)	
Lecturer	Seongsoo Hong
Topics	Motivating Example
	Development Environment Setup
Coffee Break (30 min)	
Lecture 2: *From Inference Driver to Inference Runtime* (1 h 30 min)	
Lecturer	Seongsoo Hong and Namcheol Lee
Topics	Step-by-Step Inference Driver Walkthrough
	Internals of LiteRT
Lunch Break (1 h)	
Lecture 3: *Model Slicer* (1 h 30 min)	
Lecturer	Seongsoo Hong
Topic	Model Slicer: Slicing and Conversion Tool for LiteRT
Coffee Break (30 min)	
Lecture 4: *Throughput Enhancement on Heterogeneous Accelerators* (1 h 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 [ ? ])
```

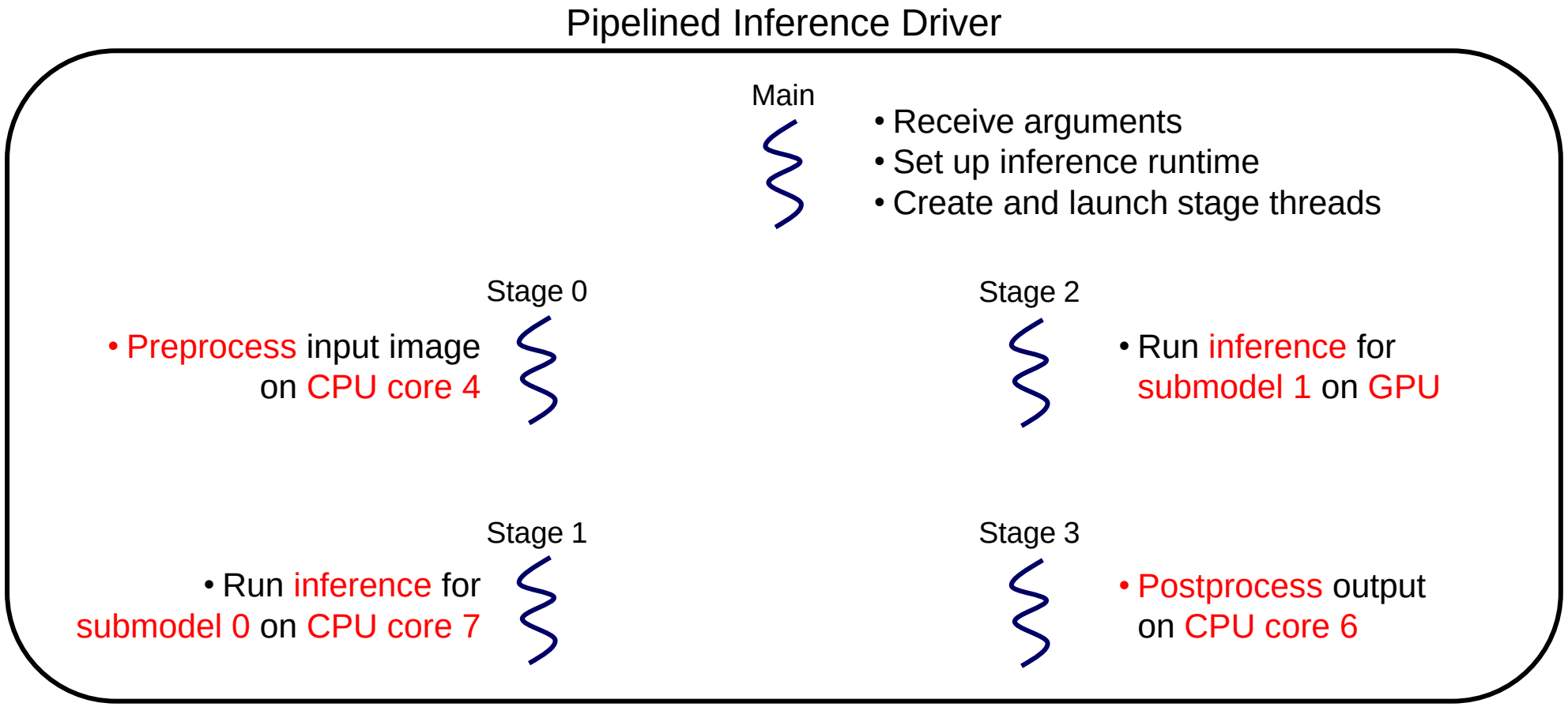
Materials

- Skeleton code
  - **`src/pipelined_inference_driver.cpp`**
- Submodels
  - **`models/submodel_0.tflite`**
    - Layers from 1 to x1 – 1 of **`models/resnet50.h5`**
  - **`models/submodel_1.tflite`**
    - Layers from x1 to 176 of **`models/resnet50.h5`**

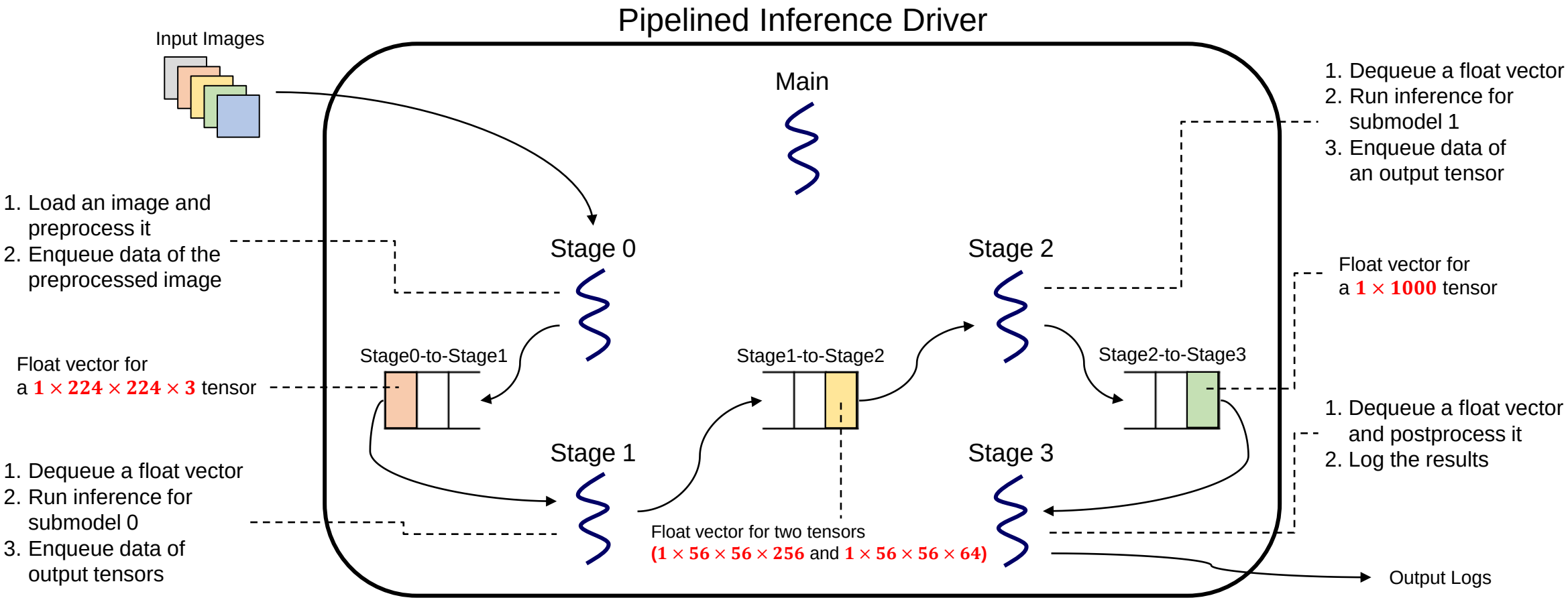
Multithreaded Structure (1)

- 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 **`StagePayload`** 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)

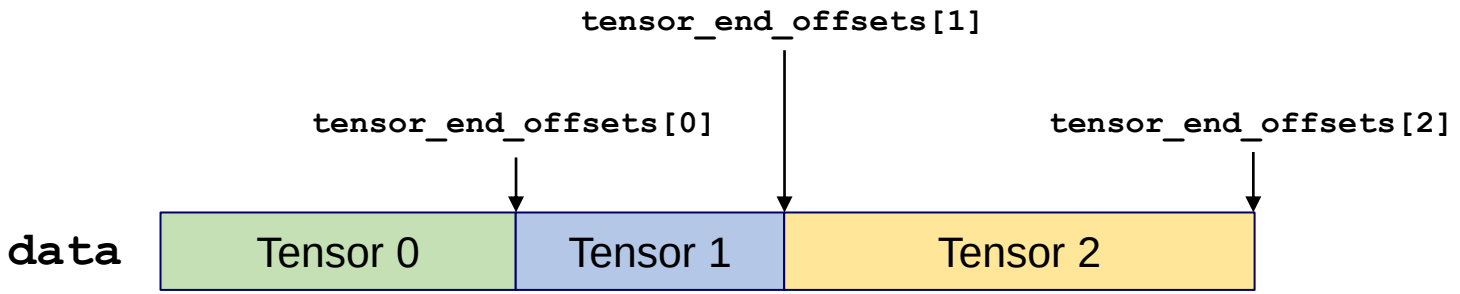
- **`InterStageQueue`** (cont'd)
  - In the pipelined inference driver
    - Three instances of **`InterStageQueue`** are defined as global variables

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

Queue Item Structure

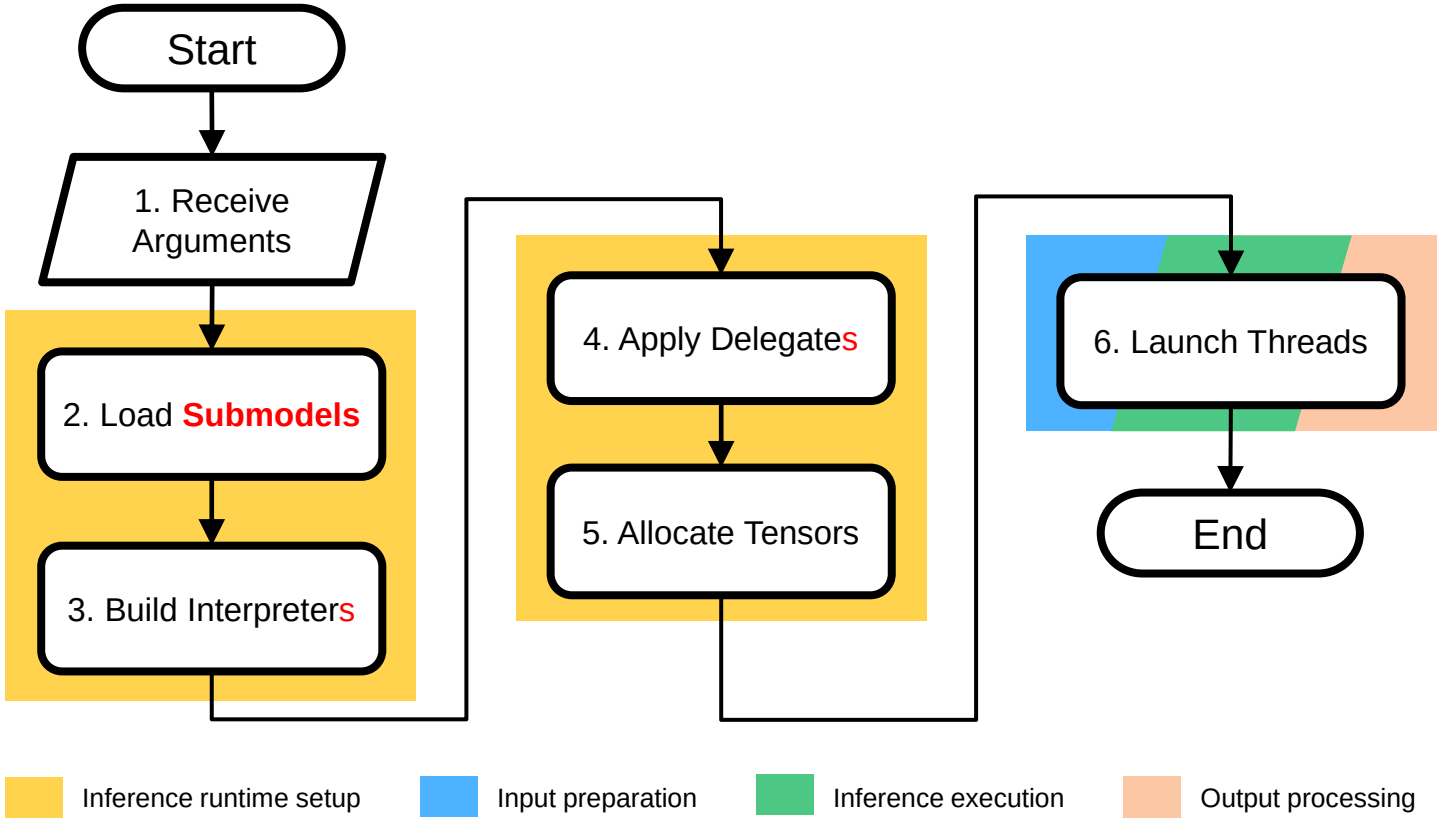
StagePayload

```
struct StagePayload {
    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
};
```



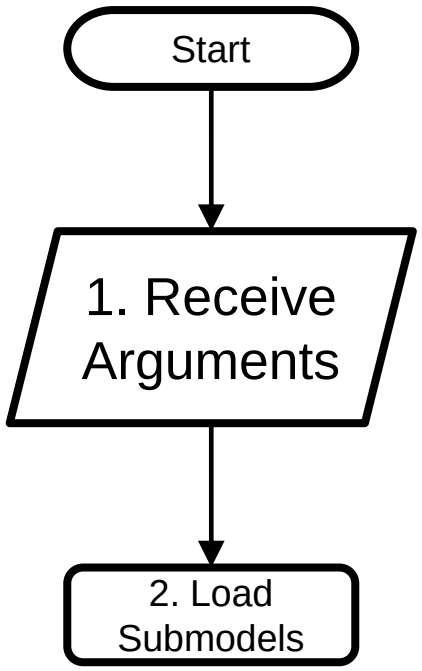
Contents

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)

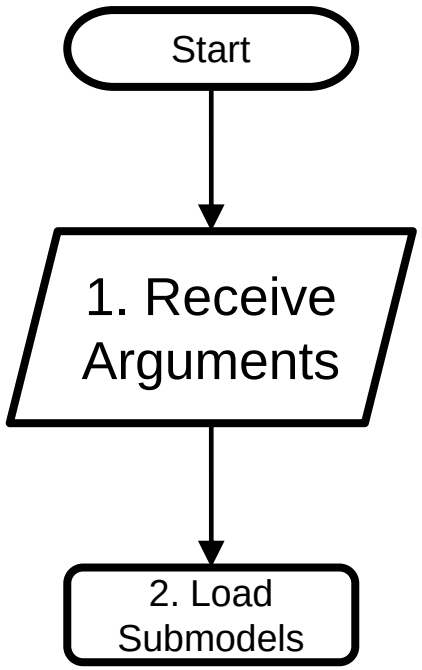
- Check the number of arguments




```
294  int main(int argc, char* argv[]) {
295      /* Receive user input */
296      if (argc < 7)
297      {
298          std::cerr << "Usage: " << argv[0]
299              << "<submodel0_path> <submodel0_gpu_usage> <submodel1_path> "
300              << "<submodel1_gpu_usage> <class_labels_path> <image_path 1> "
301              << "[image_path 2 ... image_path N] [--input-period=milliseconds]"
302              << std::endl;
303          return 1;
304      }
```

1. Receive Arguments (2)

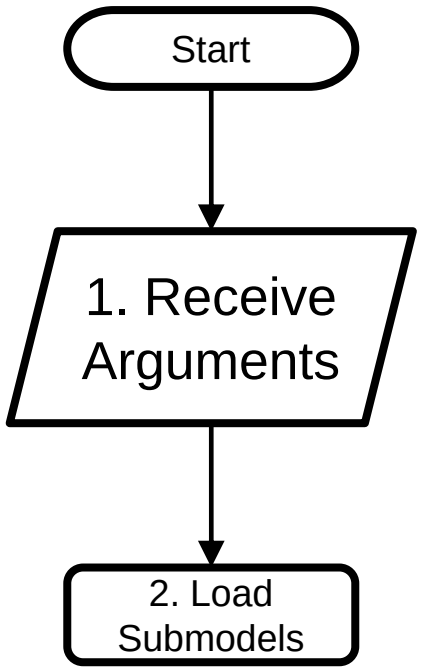
- Set variables based on the arguments




```
306     const std::string submodel0_path = argv[1];  // Path to submodel 0
307     bool submodel0_gpu_usage = false;
308     const std::string gpu_usage_str1 = argv[2];
309     if(gpu_usage_str1 == "true"){
310         submodel0_gpu_usage = true;
311     }
312
313     const std::string submodel1_path = argv[3];  // Path to submodel 1
314     bool submodel1_gpu_usage = false;
315     const std::string gpu_usage_str2 = argv[4];
316     if(gpu_usage_str2 == "true"){
317         submodel1_gpu_usage = true;
318     }
319
320     // Load class label mapping, used for postprocessing
321     const std::string class_labels_path = argv[5];
322     auto class_labels_map = util::load_class_labels(class_labels_path.c_str());
```

1. Receive Arguments (3)

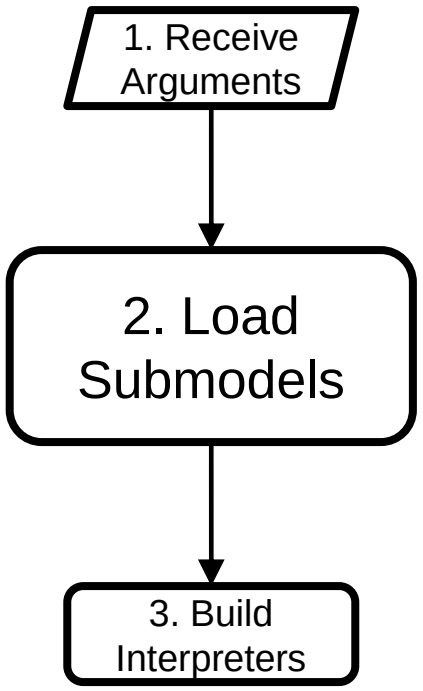
- Set variables based on the arguments (cont'd)




```
324     std::vector<std::string> images;    // List of input image paths
325     int input_period_ms = 0;            // Input period in milliseconds, default is 0 (no delay)
326     for (int i = 6; i < argc; ++i) {
327         std::string arg = argv[i];
328         if (arg.rfind("--input-period=", 0) == 0)
329             input_period_ms = std::stoi(arg.substr(15));
330         else
331             images.push_back(arg);  // Assume it's an image path
332     }
```

2. Load Submodels

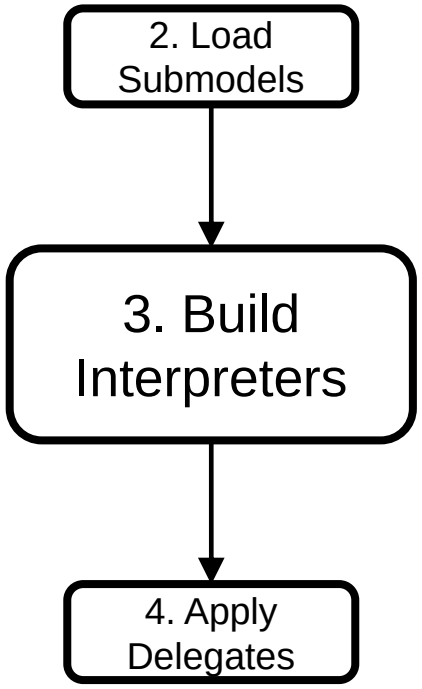
- Load submodel 0 and 1




```
334     /* Load models */
335     // 1. Create a std::unique_ptr<tflite::FlatBufferModel> for each sub-model
336     // ======= Write your code here =======
337     std::unique_ptr<tflite::FlatBufferModel> submodel0_model =
338         tflite::FlatBufferModel::BuildFromFile(submodel0_path.c_str());
339     std::unique_ptr<tflite::FlatBufferModel> submodel1_model =
340         tflite::FlatBufferModel::BuildFromFile(submodel1_path.c_str());
341     if (!submodel0_model || !submodel1_model) {
342         std::cerr << "Failed to load one or both models" << std::endl;
343         return 1;
344     }
345     // ====================================
```

3. Build Interpreters

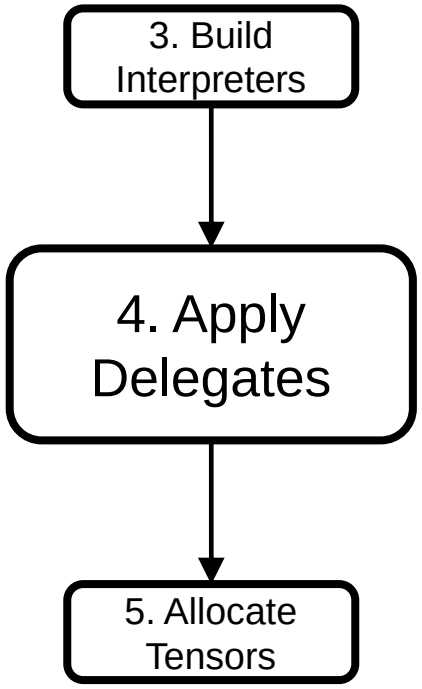
- Build interpreters for submodel 0 and submodel 1




```
347   /* Build interpreters */
348   // 1. Create an OpResolver
349   // 2. Create two interpreter builders, one for each sub-model
350   // 3. Build interpreters using the interpreter builders
351   // ======= Write your code here =======
352   tflite::ops::builtin::BuiltinOpResolver resolver;
353   tflite::InterpreterBuilder submodel0_builder(*submodel0_model, resolver);
354   tflite::InterpreterBuilder submodel1_builder(*submodel1_model, resolver);
355   std::unique_ptr<tflite::Interpreter> submodel0_interpreter;
356   std::unique_ptr<tflite::Interpreter> submodel1_interpreter;
357   submodel0_builder(&submodel0_interpreter);
358   submodel1_builder(&submodel1_interpreter);
359   if (!submodel0_interpreter || !submodel1_interpreter) {
360       std::cerr << "Failed to initialize interpreters" << std::endl;
361       return 1;
362   }
363   // ====================================
```

4. Apply Delegates (1)

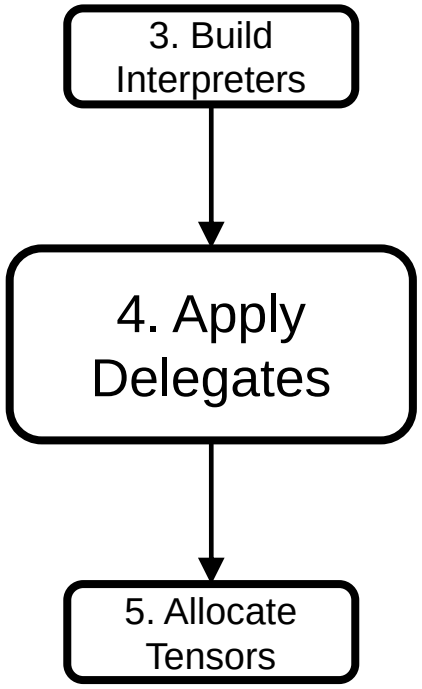
- Create and apply delegates to the interpreters




```
365     /* Apply delegate */
366     // 1. Create a XNNPACK or GPU delegate
367     // 2. Apply the delegate to the submodel0 interpreter
368     // 3. Repeat for submodel1 interpreter
369     // ======= Write your code here =======
370     TfLiteDelegate* xnn_delegate0 = TfLiteXNNPackDelegateCreate(nullptr);
371     TfLiteDelegate* gpu_delegate0 = TfLiteGpuDelegateV2Create(nullptr);
372     if(submodel0_gpu_usage) {
373         if(submodel0_interpreter->ModifyGraphWithDelegate(gpu_delegate0) == kTfLiteOk) {
374             if(xnn_delegate0) TfLiteXNNPackDelegateDelete(xnn_delegate0);
375         }
376     } else {
377         if(submodel0_interpreter->ModifyGraphWithDelegate(xnn_delegate0) == kTfLiteOk) {
378             if(gpu_delegate0) TfLiteGpuDelegateV2Delete(gpu_delegate0);
379         }
380     }
```

4. Apply Delegates (2)

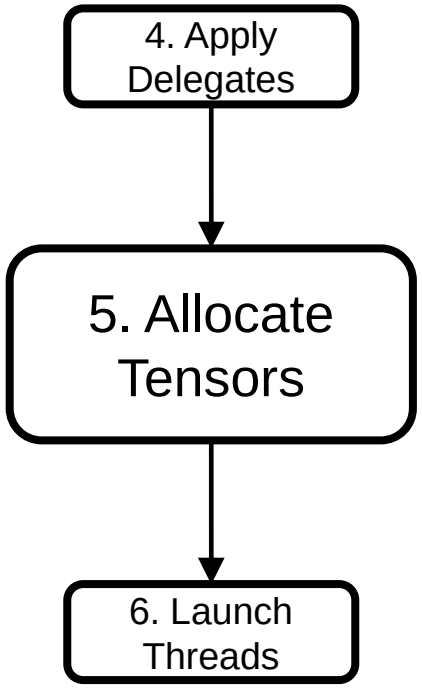
- Create and apply delegates to the interpreters (cont'd)




```
382     TfLiteDelegate* xnn_delegate1 = TfLiteXNNPackDelegateCreate(nullptr);
383     TfLiteDelegate* gpu_delegate1 = TfLiteGpuDelegateV2Create(nullptr);
384     if(submodel1_gpu_usage) {
385         if(submodel1_interpreter->ModifyGraphWithDelegate(gpu_delegate1) == kTfLiteOk) {
386             if(xnn_delegate1) TfLiteXNNPackDelegateDelete(xnn_delegate1);
387         }
388     } else {
389         if(submodel1_interpreter->ModifyGraphWithDelegate(xnn_delegate1) == kTfLiteOk) {
390             if(gpu_delegate1) TfLiteGpuDelegateV2Delete(gpu_delegate1);
391         }
392     }
393     // ====================================
```

5. Allocate Tensors

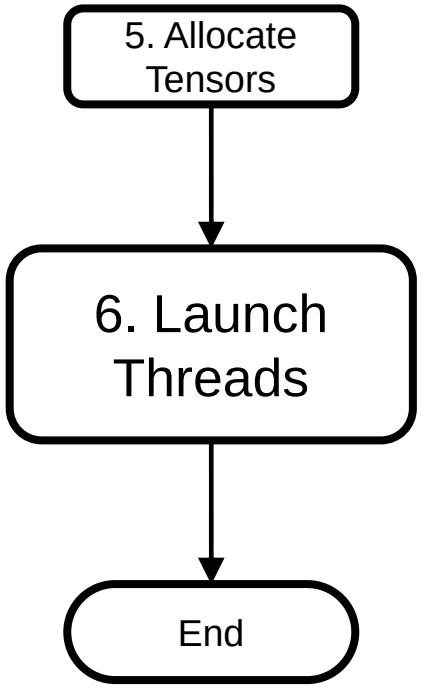
- Allocate tensors for both interpreters




```
395     /* Allocate tensors */
396     // 1. Allocate tensors for both interpreters
397     // ======= Write your code here =======
398     if (submodel0_interpreter->AllocateTensors() != kTfLiteOk) {
399         std::cerr << "Failed to allocate tensors for submodel0" << std::endl;
400         return 1;
401     }
402     if (submodel1_interpreter->AllocateTensors() != kTfLiteOk) {
403         std::cerr << "Failed to allocate tensors for submodel1" << std::endl;
404         return 1;
405     }
406     // ====================================
```

6. Launch Threads

- Create and launch threads for each stage




```
411     /* Create and launch threads */
412     // 1. Launch stage0_worker in a new thread with images and input_period_ms
413     // 2. Launch stage1_worker in a new thread with submodel0 interpreter
414     // 3. Launch stage2_worker in a new thread with submodel1 interpreter
415     // 4. Launch stage3_worker in a new thread with class_labels_map
416     // ======= Write your code here =======
417     std::thread stage0_thread(stage0_worker, images, input_period_ms);
418     std::thread stage1_thread(stage1_worker, submodel0_interpreter.get());
419     std::thread stage2_thread(stage2_worker, submodel1_interpreter.get());
420     std::thread stage3_thread(stage3_worker, class_labels_map);
421     // ====================================
```

Contents

Four Stage Workers

- Functions executed by stage threads

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

1. Stage0 Worker (1)

- Specification
  - Inputs
    - Vector of image file paths (**`images`**)
    - Input period in milliseconds (**`input_period_ms`**)
  - Output
    - **`StagePayload`** containing a preprocessed image of shape $224 \times 224 \times 3$
  - Behavior
    1. Load and preprocess an image from the image file path vector
    2. Convert the preprocessed image into an **`StagePayload`**
    3. Push the **`StagePayload`** into **`stage0_to_stage1_queue`**
    4. Sleep until the next input period, unless already behind schedule
       - If no images remain, signal shutdown and return

1. 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          }
```

1. Stage0 Worker (3)

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

```
63          /* Create an StagePayload, 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          StagePayload stage_payload;
67          // ======= Write your code here =======
68          stage_payload.index = idx;
69          stage_payload.data.resize(
70              preprocessed_image.total() * preprocessed_image.channels());
71          std::memcpy(stage_payload.data.data(), preprocessed_image.ptr<float>(),
72              stage_payload.data.size() * sizeof(float));
73          stage_payload.tensor_end_offsets =
74              {static_cast<int>(stage_payload.data.size())};
75          // ====================================
```

1. Stage0 Worker (4)

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

```
76          stage0_to_stage1_queue.push(std::move(stage_payload));
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
```

2. Stage1 Worker (1)

- Specification
  - Input
    - `StagePayload` containing one tensor of shape $1 \times 224 \times 224 \times 3$
  - Outputs
    - `StagePayload` containing two tensors of shape $1 \times 56 \times 56 \times 256$ and $1 \times 56 \times 56 \times 64$

2. Stage1 Worker (2)

- Specification (cont'd)
  - Behavior
    1. Pop a **`StagePayload`** 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 **`StagePayload`**
    5. Push the **`StagePayload`** 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

2. Stage1 Worker (3)

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

```
91  void stage1_worker(tflite::Interpreter* interpreter) {
92      StagePayload stage_payload;
93      bool print_tensor_shape = true;
94
95      while (stage0_to_stage1_queue.pop(stage_payload)) {
```

2. Stage1 Worker (4)

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

```
99      /* Extract tensor data from stage_payload.data and copy into
100     *  the corresponding input tensors of the interpreter */
101     // ======= Write your code here =======
102     if(print_tensor_shape) std::cout << "Input Tensor(s) of Stage 1" << std::endl;
103     for (size_t i = 0; i < interpreter->inputs().size(); ++i) {
104         // Get i-th input tensor object
105         TfLiteTensor* input_tensor = interpreter->input_tensor(i);
```

2. Stage1 Worker (5)

2. Copy its data into the input tensor of the **`interpreter`** (cont'd)

```
107         // Calculate the required number of elements for the i-th input tensor based on its dimensions
108         int num_elements = 1;
109         if(print_tensor_shape) std::cout << "- Shape of input tensor " << i << ": [";
110         for (int d = 0; d < input_tensor->dims->size; ++d) {
111             num_elements *= input_tensor->dims->data[d];
112             if(print_tensor_shape) {
113                 std::cout << input_tensor->dims->data[d];
114                 if(d != input_tensor->dims->size - 1) std::cout << ", ";
115             }
116         }
117         if(print tensor shape) std::cout << "], # of elements = " << num elements << std::endl;
```

```
Input Tensor(s) of Stage 1
- Shape of input tensor 0: [1, 224, 224, 3], # of elements = 150528
```

- Shape of a **`TfLiteTensor*`** is stored under **`TfLiteTensor->dims->data`**
  - It is an integer array with one entry per dimension in order

2. Stage1 Worker (6)

2. Copy its data into the input tensor of the **`interpreter`** (cont'd)

```
119             // Copy data from stage_payload to i-th input tensor
120             int start_idx = (i == 0) ? 0 : stage_payload.tensor_end_offsets[i-1];
121             int end_idx = stage_payload.tensor_end_offsets[i];
122             int given_elements = end_idx - start_idx;
123 
124             if(num_elements != given_elements){
125                 std::cerr << "[Stage 1] Input tensor " << i << " size mismatch! " << std::endl;
126                 std::cerr << "Expected " << num_elements << ", got " << given_elements << std::endl;
127             } else {
128                 std::memcpy(input_tensor->data.f, stage_payload.data.data() + start_idx,
129                             (given_elements) * sizeof(float));
130             }
131         } // end of for loop
132         // ====================================
```

2. Stage1 Worker (7)

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

```
134     /* Inference */
135     // ======= Write your code here =======
136     interpreter->Invoke();
137     // ====================================
```

2. Stage1 Worker (8)

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

```
139         /* Extract data from the interpreter's output tensors and copy into a StagePayload */
140         // Clear data in it for reuse
141         stage_payload.data.clear();
142         stage_payload.tensor_end_offsets.clear();
143         if(print_tensor_shape) std::cout << "Output Tensor(s) of Stage 1" << std::endl;
144         // ======= Write your code here =======
145         for (size_t i = 0; i < interpreter->outputs().size(); ++i) {
146             // Get i-th output tensor object
147             TfLiteTensor* output_tensor = interpreter->output_tensor(i);
```

2. Stage1 Worker (9)

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

```
149             // Calculate the number of elements in the i-th output tensor
150             if(print_tensor_shape) std::cout << "- Shape of output tensor " << i << ": [";
151             int num_elements = 1;
152             for (int d = 0; d < output_tensor->dims->size; ++d) {
153                 num_elements *= output_tensor->dims->data[d];
154                 if(print_tensor_shape) {
155                     std::cout << output_tensor->dims->data[d];
156                     if(d != output_tensor->dims->size - 1) std::cout << ", ";
157                 }
158             }
159             if(print_tensor_shape) std::cout << "], # of elements = " << num_elements << std::endl;
```

```
Output Tensor(s) of Stage 1
- Shape of output tensor 0: [1, 56, 56, 256], # of elements = 802816
- Shape of output tensor 1: [1, 56, 56, 64], # of elements = 200704
```

2. Stage1 Worker (10)

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

```
161             // Resize stage_payload.data and copy output tensor's data into it
162             int current_data_length = stage_payload.data.size();
163             stage_payload.data.resize(current_data_length + num_elements);
164             std::memcpy(stage_payload.data.data() + current_data_length,
165                 output_tensor->data.f,
166                 num_elements * sizeof(float));
167             stage_payload.tensor_end_offsets.push_back(current_data_length + num_elements);
168         } // end of for loop
169         print_tensor_shape = false;
170         // ====================================
```

2. Stage1 Worker (11)

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

```
172         stage1_to_stage2_queue.push(std::move(stage_payload));
173
174         util::timer_stop(label);
175     } // 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

```
177     // Notify stage2_thread that no more data will be sent
178     stage1_to_stage2_queue.signal_shutdown();
179 } // end of stage1_worker
```

3. Stage2 Worker (1)

- Specification
  - Inputs
    - `StagePayload` containing two tensors of shape $1 \times 56 \times 56 \times 256$ and $1 \times 56 \times 56 \times 64$
  - Output
    - `StagePayload` containing one tensor of shape 1×1000

3. Stage2 Worker (2)

- Specification (cont'd)
  - Behavior
    1. Pop a **`StagePayload`** 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 **`StagePayload`**
    5. Push the **`StagePayload`** 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

3. Stage2 Worker (3)

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

```
181 void stage2_worker(tflite::Interpreter* interpreter) {
182     StagePayload stage_payload;
183     bool print_tensor_shape = true;
184 
185     while (stage1_to_stage2_queue.pop(stage_payload)) {
```

3. Stage2 Worker (4)

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

```
189         /* Extract tensor data from stage_payload.data and copy into
190         *  the corresponding input tensors of the interpreter */
191         if(print_tensor_shape) std::cout << "Input Tensor(s) of Stage 2" << std::endl;
192         // ======= Write your code here =======
193         for (size_t i = 0; i < interpreter->inputs().size(); ++i) {
194             // Get i-th input tensor object
195             TfLiteTensor* input_tensor = interpreter->input_tensor(i);
```

3. Stage2 Worker (5)

2. Copy its data into the input tensors of the **`interpreter`** (cont'd)

```
197             // Calculate the required number of elements for the i-th input tensor based on its dimensions
198             int num_elements = 1;
199             if(print_tensor_shape) std::cout << "- Shape of input tensor " << i << ": [";
200             for (int d = 0; d < input_tensor->dims->size; ++d) {
201                 num_elements *= input_tensor->dims->data[d];
202                 if(print_tensor_shape) {
203                     std::cout << input_tensor->dims->data[d];
204                     if(d != input_tensor->dims->size - 1) std::cout << ", ";
205                 }
206             }
207             if(print_tensor_shape) std::cout << "], # of elements = " << num_elements << std::endl;
```

```
Input Tensor(s) of Stage 2
- Shape of input tensor 0: [1, 56, 56, 256], # of elements = 802816
- Shape of input tensor 1: [1, 56, 56, 64], # of elements = 200704
```

3. Stage2 Worker (6)

2. Copy its data into the input tensors of the **`interpreter`** (cont'd)

```
209             // Copy data from stage_payload to i-th input tensor
210             int start_idx = (i == 0) ? 0 : stage_payload.tensor_end_offsets[i-1];
211             int end_idx = stage_payload.tensor_end_offsets[i];
212             int given_elements = end_idx - start_idx;
213 
214             if(num_elements != given_elements){
215                 std::cerr << "[Stage 2] Input tensor " << i << " size mismatch! " << std::endl;
216                 std::cerr << "Expected " << num_elements << ", got " << given_elements << std::endl;
217             } else {
218                 std::memcpy(input_tensor->data.f, stage_payload.data.data() + start_idx,
219                             (given_elements) * sizeof(float));
220             }
221         } // end of for loop
222         // ====================================
```

3. Stage2 Worker (7)

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

```
224     /* Inference */
225     // ======= Write your code here =======
226     interpreter->Invoke();
227     // ====================================
```

3. Stage2 Worker (8)

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

```
229         /* Extract data from the interpreter's output tensors and copy into a StagePayload */
230         // Clear data in it for reuse
231         stage_payload.data.clear();
232         stage_payload.tensor_end_offsets.clear();
233         if(print_tensor_shape) std::cout << "Output Tensor(s) of Stage 2" << std::endl;
234         // ======= Write your code here =======
235         for (size_t i = 0; i < interpreter->outputs().size(); ++i) {
236             // Get i-th output tensor object
237             TfLiteTensor* output_tensor = interpreter->output_tensor(i);
```

3. Stage2 Worker (9)

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

```
239             // Calculate the number of elements in the tensor
240             if(print_tensor_shape) std::cout << "- Shape of output tensor " << i << ": [";
241             int num_elements = 1;
242             for (int d = 0; d < output_tensor->dims->size; ++d) {
243                 num_elements *= output_tensor->dims->data[d];
244                 if(print_tensor_shape) {
245                     std::cout << output_tensor->dims->data[d];
246                     if(d != output_tensor->dims->size - 1) std::cout << ", ";
247                 }
248             }
249             if(print_tensor_shape) std::cout << "], # of elements = " << num_elements << std::endl;
```

```
Output Tensor(s) of Stage 2
- Shape of output tensor 0: [1, 1000], # of elements = 1000
```

3. Stage2 Worker (10)

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

```
251             // Resize stage_payload.data and copy output tensor data into it
252             int current_data_length = stage_payload.data.size();
253             stage_payload.data.resize(current_data_length + num_elements);
254             std::memcpy(stage_payload.data.data() + current_data_length,
255                 output_tensor->data.f,
256                 num_elements * sizeof(float));
257             stage_payload.tensor_end_offsets.push_back(current_data_length + num_elements);
258         } // end of for loop
259         print_tensor_shape = false;
260         // ====================================
```

3. Stage2 Worker (11)

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

```
261         stage2_to_stage3_queue.push(std::move(stage_payload));
262         util::timer_stop(label);
263     } // 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

```
265     stage2_to_stage3_queue.signal_shutdown();
266 } // end of stage2_worker
```

4. Stage3 Worker (1)

- Specification
  - Inputs
    - Map of class labels (`class_labels_map`)
    - `StagePayload` containing one tensor of shape 1×1000
  - Output
    - Prediction result
  - Behavior
    1. Pop a `StagePayload` from `stage2_to_stage3_queue`
    2. Postprocess the output tensor data in the `StagePayload`
    3. Repeat until `stage2_to_stage3_queue` signals shutdown and is empty

4. Stage3 Worker (2)

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

```
268   void stage3_worker(std::unordered_map<int, std::string> class_labels_map) {
269       StagePayload stage_payload;
270
271       while (stage2_to_stage3_queue.pop(stage_payload)) {
```

4. Stage3 Worker (3)

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

```
275         const std::vector<float>& probs = stage_payload.data;
276
277         if ((stage_payload.index + 1) % 10 == 0) {
278             auto top_k_indices = util::get_topK_indices(probs, 3);
279             std::cout << "\n[stage3] Top-3 prediction for image index "
280                       << stage_payload.index << ":\n";
281             for (int idx : top_k_indices) {
282                 std::string label = class_labels_map.count(idx)
283                     ? class_labels_map.at(idx)
284                     : "unknown";
285                 std::cout << "- Class " << idx << " (" << label
286                           << "): " << probs[idx] << std::endl;
287             }
288         }
```

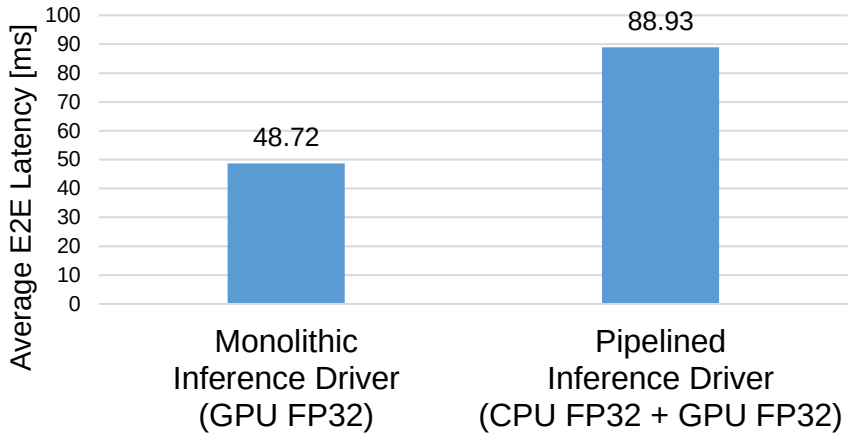
3. Repeat until **`stage2_to_stage3_queue`** signals shutdown and is empty

```
291     } // end of while loop
292 } // end of stage3_worker
```

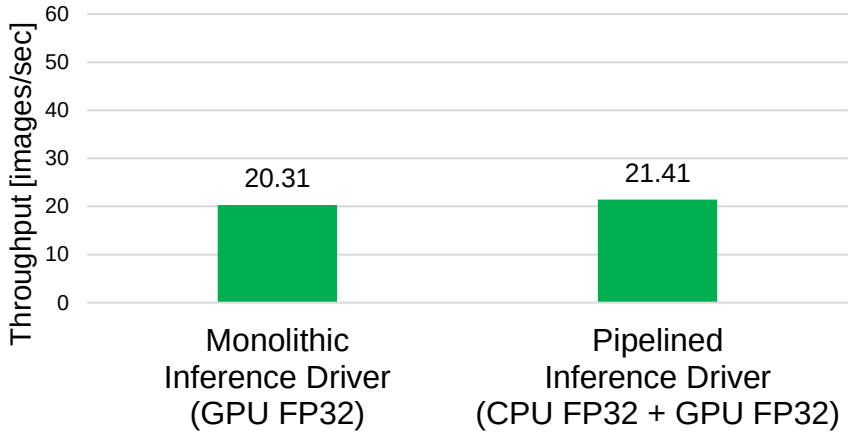
Contents

Monolithic vs. Pipelined Inference Drivers (1)

- End-to-End latency
  - Travel time of a single sample
  - Increased by about 1.83 times

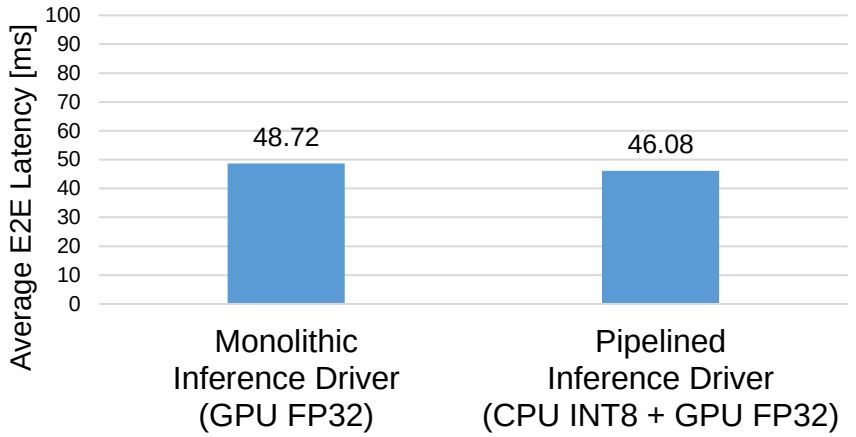


- Throughput
  - Frames per second
  - Increased by about 5.4%

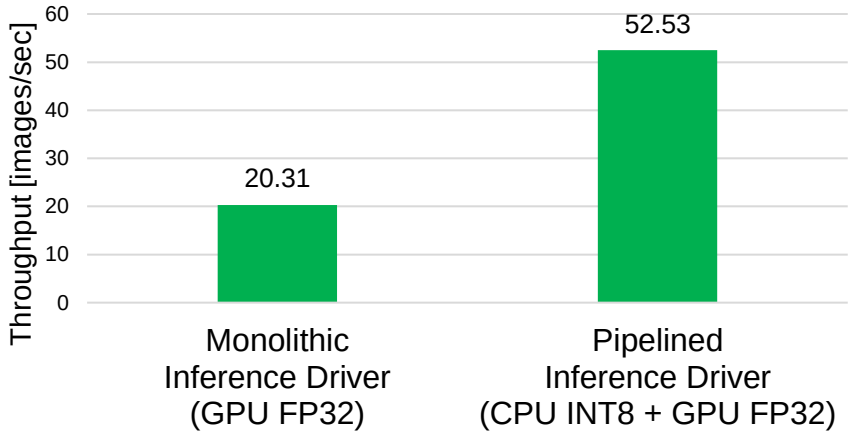


Monolithic vs. Pipelined Inference Drivers (2)

- End-to-End latency
  - Travel time of a single sample
  - Decreased by about 0.95 times



- Throughput
  - Frames per second
  - Increased by about 158.64%
    - When 6 out of 8 cores utilized



Contents

Balance Pipeline Stages

- Objective
  - Balance latency between pipeline stages, especially stage 1 and stage 2
- Do
  - Follow the instructions in the next slides
- Verify
  - Check the logs printed in the terminal
- Time
  - 15 minutes

Parameters to Adjust

- Number of layers in each submodel
  - Decide how many layers go into submodel 0 and submodel 1
- Accelerator assignment
  - Select which accelerator (CPU or GPU) runs stage 1 and stage 2
    - Revise configurations in **~/DNNPipe-Tutorial/run_pipelined_inference_driver.sh**

```
19  submodel0="./models/submodel_0.tflite"
20  submodel0_gpu_usage="false" # adjust as needed
21  submodel1="./models/submodel_1.tflite"
22  submodel1_gpu_usage="true" # adjust as needed
```

Run Pipelined Inference Driver

- In **`~/DNNPipe-Tutorial`**, run
  - **`make`**
  - **`./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)
```