Application Showcases for TVM with NeuroPilot on Mobile Devices

Sheng-Yuan Cheng
Department of Computer Science,
National Tsing Hua University
Hsinchu, Taiwan
sycheng@pllab.cs.nthu.edu.tw

Chun-Ping Chung
Department of Computer Science,
National Tsing Hua University
Hsinchu, Taiwan
cpchung@pllab.cs.nthu.edu.tw

Robert Lai
Mediatek Inc.
Hsinchu, Taiwan
Robert.Lai@mediatek.com

Jenq-Kuen Lee
Department of Computer Science,
National Tsing Hua University
Hsinchu, Taiwan
jklee@cs.nthu.edu.tw

ABSTRACT

With the increasing demand for machine learning inference on mobile devices, more platforms are emerging to provide AI inferences on mobile devices. One of the popular ones is TVM, which is an end-to-end AI compiler. The major drawback is TVM doesn't support all manufacturer-supplied accelerators. On the other hand, an AI solution for MediaTek's platform, NeuroPilot, offers inference on mobile devices with high performance. Nevertheless, NeuroPilot does not support all of the common machine learning frameworks. Therefore, we want to take advantage of both sides. This way, the solution could accept a variety of machine learning frameworks, including Tensorflow, Pytorch, ONNX, and MxNet and utilize the AI accelerator from MediaTek. We adopt the TVM BYOC flow to implement the solution. In order to illustrate the ability to accept different machine learning frameworks for different tasks, we used three different models to build an application showcase in this work: the face anti-spoofing model from PyTorch, the emotion detection model from Keras, and the object detection model from Tflite. Since these models have dependencies while running inference, we propose a prototype of pipeline algorithm to improve the inference performance of the application showcase.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; *Pipeline computing*.

KEYWORDS

TVM, NeuroPilot, Relay IR, Deep Learning, Inference, Pipeline

ICPP Workshops '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9445-1/22/08…$15.00
https://doi.org/10.1145/3547276.3548514

ACM Reference Format:
Sheng-Yuan Cheng, Chun-Ping Chung, Robert Lai, and Jenq-Kuen Lee. 2022. Application Showcases for TVM with NeuroPilot on Mobile Devices. In *51th International Conference on Parallel Processing Workshop (ICPP Workshops '22), August 29-September 1, 2022, Bordeaux, France.* ACM, Bordeaux, France, 8 pages. https://doi.org/10.1145/3547276.3548514

1 INTRODUCTION

Due to the booming of machine learning, there are more and more machine learning frameworks that offer convenience for researchers or developers, such as TensorFlow[1], Keras[10], and PyTorch[15], etc. However, without an AI compiler, various machine learning frameworks cannot be used end-to-end. In this sense, TVM[3] appears to bridge the gap between machine learning frameworks and applications and back-ends. Since inference needs to run on edge devices like mobile phones, Mediatek has released NeuroPilot[4], a cross-platform framework for deploying AI models from well-known frameworks to edge devices.

Nevertheless, both NeuroPilot and TVM have some drawbacks. TVM is an open-source framework that provides common front-end and back-end functionality, but TVM does not support all manufacturer-supplied hardware, such as the MediaTek AI accelerator for mobile devices. NeuroPilot, made by Mediatek, on the other hand, does not support as many machine learning frameworks as TVM. Thus, taking advantage of both sides becomes an extremely valuable idea, which could ultimately lead to a win-win outcome.

A TVM BYOC (Bring Your Own Codegen) flow provides the opportunity for the scenario. Researchers and developers can focus on their familiar framework by using a BYOC flow, which connects different runtimes through external CodeGen and runtime. It is also beneficial to developers and hardware backend providers to be able to embrace a community-wide perspective. Prior work has used TVM BYOC flow to use NNAPI for AI inference on mobile devices via NNAPI accelerator[11], with the similar idea, another prior work aimed to support NNEF execution model for NNAPI[2]. All these studies aimed to connect different frameworks to enable better AI. This work, We leveraged TVM BYOC to bridge the gap between TVM and NeuroPilot. First, we calculate the parameters for each TVM relay OP and then convert the relay AST to the corresponding Neuron IR type. We also map each TVM relay OP to

Neuron IR. We then use BYOC flow to generate runtime for TVM and NeuroPilot. In this way, the solution flow could accept more front-end machine learning frameworks from TVM frontend and use the back-end targets provided by NeuroPilot.

To further demonstrate the usage of our flow, we build an application showcase.(Figure 1) This application consists of three different machine learning models which run different tasks and are from different machine learning frameworks. The first model is a mobilenet quant model from Tflite, which runs object detection jobs and boxes each object in each frame of a video. On top of that, an anti-spoofing model[6] is employed to distinguish fake faces from real ones. The model uses a Convolutional Neural Network (CNN) based framework for presentation attack detection, with deep pixel-wise supervision. Last but not least, we apply an emotion detection model to extract the emotion of the chosen faces. The TVM BYOC flow allows us to leverage models from different machine learning frameworks, which makes our application showcase unique.

In our application showcase, the three models are dependent upon each other. That is, only the faces collected by object detection models and judged as real faces by the face anti-spoofing models could be detected for the emotions. Due to the numerous back-ends provided by Mediatek NeuroPilot, including mobile CPU, GPU or AI accelerators, we developed an early pipeline prototype to improve application performance.

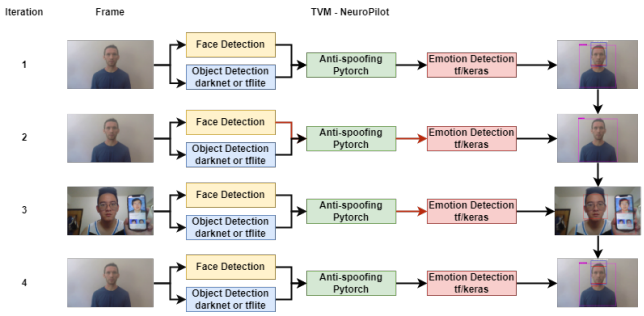

Figure 1: Execution flow of the application showcase. For each frame, the frame would be passed through a face detector and an object detector in order to detect a human and a face, then the frame would be passed through an anti-spoofing model to determine the real one from the fake. In the end, an emotion detection model is used to detect emotions.

2 BACKGROUND

2.1 MediaTek NeuroPilot

In recent years, a number of AI applications have been used in mobile devices and smart vehicles. The use of AI in the cloud or edge computing has also been discussed. Many companies have therefore developed their own platforms to run AI inference on edge devices. Intel, for example, offers an end-to-end AI framework on the edge computing called OpenVINO[7]. MediaTek also released NeuroPilot, their platform for running AI inference on mobile devices.

Even though we all know the importance of edge computing, there are still many challenges to overcome. Most of these challenges stem from physical limitations, such as power and heat problems. For this reason, NeuroPilot relies on hardware from MediaTek to resolve the problems. TVM does not easily support hardware of this kind, so we want to bridge the gap between them in order to get the best of both worlds.

NeuroPilot has two core concepts that are directly related to our work: Runtime and Compiler. As for the Compiler part, it provides high-level IR to accept various machine learning frameworks as well as mapping many AI operations. Furthermore, the compiler also provides an Execution Planner mechanism to assign operators to back-end targets. The Runtime will infer the output binary after the Compiler has completed its work.

2.2 TVM

TVM is an open-source end-to-end machine learning compiler framework. It works like a traditional compiler like LLVM[12], which is composed of three phases, the front-end, the intermediate representation, and the back-end. The ease of use and convenience in running an end-to-end AI solution have made TVM more and more popular, also some researches have developed tools for developing projects based on TVM, including NNBlocks[5]. A major difference between TVM and NeuroPilot is that TVM's frontend accepts a variety of machine learning frameworks, including TensorFlow[1], Keras[10], PyTorch[15], as well as the open standard ONNX[14]. In TVM, there are two intermediate representations (IR). One is relay IR[17], and the other is TIR.

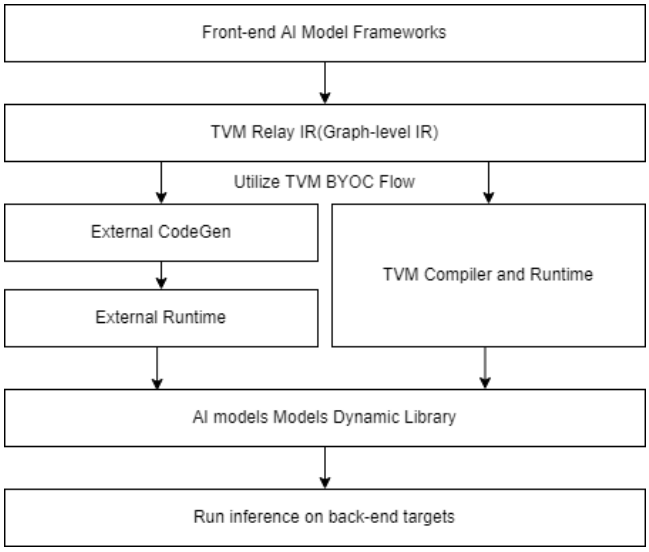

Figure 2: A brief overview of TVM BYOC architecture. As a starting point, the TVM front-end will accept AI models from a variety of frameworks. After that, TVM converts the AI model into graph-level IR, relay IR. During this process, we employ a partitioning algorithm to divide the graph into two parts, one for TVM and one for MediaTek NeuroPilot.

A relay IR is a high-level IR that describes a whole machine learning model, including all the AI operations that will be used by the model but does not elaborate on how each operation is going to be implemented. TIR, on the other hand, describes how all AI operations are to be implemented. This makes relay IR a great level for optimizing since we don't have to be concerned with details, but the data flow. On TVM Relay IR, we could perform node fusion, node permutation, or even partition on the nodes. For the rest of the paper, we utilize the TVM BYOC flow to manipulate TVM and NeuroPilot. As for TVM, we used TVM Relay IR to perform graph partition and transformation.

3 NEUROPILOT SUPPORT FOR TVM

3.1 Utilizing the BYOC flow in TVM

According to our previous description, TVM is an open-source AI compiler framework that supports well-known front-end machine learning frameworks as well as various back-end targets.

Nevertheless, if a hardware manufacturer wishes to release its own hardware target, such as a machine learning accelerator, the provider may not be able to provide the same programming interface that could be integrated into current machine learning frameworks, such as TensorFlow, Pytorch, or MXNet.

Therefore, TVM BYOC was born to provide a solution to the problem. As its name suggests, BYOC stands for bring your own code generation for your accelerator, so this flow could use an external compiler and runtime to infer AI models. Figure 2 illustrates the high-level architecture. From the outset, the TVM front-end would accept AI models from various frameworks. Afterward, TVM converts the AI model to the graph-level IR, relay IR. In this process, we implement a partitioning algorithm to partition the graph into TVM's built-in part and the external part for MediaTek NeuroPilot. In the last step, we created an AI model library and deployed it across different back-ends including servers and AI accelerators on mobile devices.

3.2 Compile Relay sub-modules into equivalent IR in NeuroPilot

In order to use external compilers in TVM BYOC flow, TVM Relay IR must be converted into its NeuroPilot equivalent. Regarding how to perform this task, TVM provides a built-in structure that can be employed to traverse the AST of Relay IR, called ExprVistor. Using ExprVisitor, we perform three steps to traverse AST and map the OP between TVM and NeuroPilot.

To begin, we need to convert the parameters into tensor-oriented expressions. In particular, in order to generate IR and mapping for the QNN models, which have operator-oriented representations, we must first convert them. Adding to this, we defined a NodeEntry structure to store the inputs and outputs of each Node while using post-order DFS to traverse the Relay AST and construct the IR in NeuroPilot. Our final step is to map the relay operations into IR in NeuroPilot representations using the dictionary and object that contains the logic to convert the relay operations into IR in NeuroPilot. With this relationship and the NodeEntry, we are able to roughly convert the entire relay module to its Neuron IR counterpart. The above steps are listed in Listing 1 where we utilize NodeEntry to store information, and also use a dictionary to record.

```
def visit_var(var):
    node_entry = NodeEntry()
    neuron_input = convert_to_neuron(var)
    node_entry.inputs = [neuron_input]
    node_entry.outputs = [neuron_input]
    node_entry_dict[var] = node_entry

def visit_tuple(tuple):
    node_entry = NodeEntry()
    for field in tuple.fields:
        visit(field)
        node_entry.inputs.add(field.outputs)
    node_entry.outputs.add(node_entry.inputs)
    node_entry_dict[tuple] = node_entry

def visit_call(call):
    node_entry = NodeEntry()
    for arg in call.args:
        visit(arg)
        node_entry.inputs.add(arg.outputs)
    op_name = get_op_name(call)
    op_handler_dict[op_name]
        .create_op(call, node_entry)
    node_entry_dict[call] = node_entry
```

Listing 1: This is a brief explanation of how we implement TVM Relay's IR to IR conversion in NeuroPilot. ExprVisitor is used in TVM, and we have defined a structure called NodeEntry to store data inputs and outputs. Furthermore, a post-order DFS is used to traverse the Relay AST and build the corresponding IR.

3.3 Augment QNN flow in TVM BYOC

In order to keep pace with the developments in machine learning, we are striving to deploy machine learning models to edge devices. In any case, the models would be too large to fit inside the edge devices. As a result, some previous research has attempted to thin the AI models. The Quantized Neural Network[9] is a famous method that trains models with low precision weights and activations.

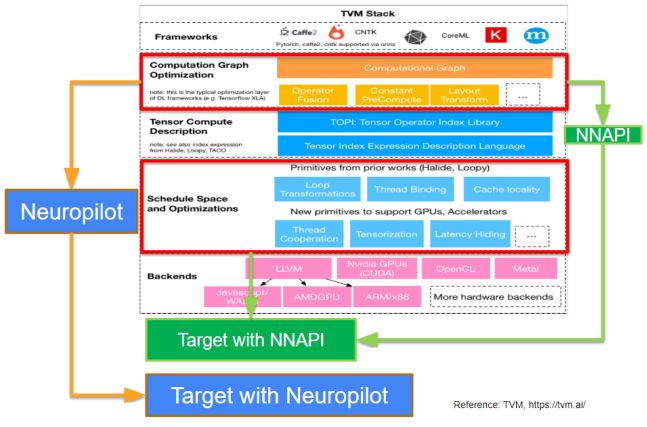

Figure 3: A high level view of the design structure. The graphs are partitioned at the computation graph level, which is built over the TVM stack. Previously, our team enabled NNAPI flow as we adopted NeuroPilot, and now we are targeting Mediatek NeuroPilot.

In this manner, the precision will decrease slightly, but the models will be relatively smaller, which is more suitable for use at the edge. To keep up with this trend, we also implemented QNN conversion. To accomplish this, some issues must be overcome. Firstly, since Relay Qnn is operator-oriented, the quantization parameters will only appear on the input of the operator with the prefix qnn, however, in Neuron these parameters will need to be carried across all Tensors. Moreover, it is important to note that even if the model has been pre-quantized, there are still some non-qnn options. When visiting such an output, we pass the output quantization parameters directly to the input and continue passing them.

In the following section of this paper, we will present our application showcase, as well as use a quentized Mobilenet SSD model to illustrate the effectiveness of augmenting QNN flow.

4 APPLICATION SHOWCASE

Based on the shoulders of giants, TVM offers a BYOC flow to adopt domain-specific AI accelerators provided by different vendors. Our team's previous work has used TVM BYOC flow to use NNAPI for AI inference on mobile devices via NNAPI accelerator[11].

NNAPI is an Android C API designed for running computationally intensive operations for machine learning on Android devices. This was followed by the creation of another flow that incorporated more back-ends such as NeuroPilot. When viewed from a higher perspective, all of these concepts can be seen in Figure 3. We partition graphs at the computation graph level, which is built above TVM stack. Previous work of our team enabled NNAPI flow as we adopted NeuroPilot.

As we were developing our application, we discovered an interesting github repository, which is the open model zoo provided by Intel OpenVINO[7]. They provided many pre-trained models and combined some subsets to develop various applications which could be applied in reality. Nonetheless, they would first convert the models into their specific IR and binary files. These files would not be used by us since we must accept models from TVM and convert to IR for NeuroPilot. In addition, if we used their specific representation of models, we could not perform model tuning and optimization.

Considering this, we compiled models from different machine learning frameworks such as Tensorflow, PyTorch, and tflite around a variety of tasks to be our application showcase. By doing so, we would be able to first prove the concept of our flow, and then perform further optimization based on this basis.

4.1 Face Anti-spoofing model

The first model is responsible for preventing face spoofing. As we all know, we have face detection, object detection kinds of models that can box where a human is located. However, it appears that the presentation attacks are using fake faces rather than real faces. Thus, many face anti-spoofing models appear to address this issue. Deep Pixel-wise Binary Supervision[6] is one of the solutions we will employ in our application.

This model comes from Pytorch, and the details of how we use it can be found in Listing 2. We first create the Pytorch model, then accept the model into TVM via a method in relay.frontend, then partition the graph into IR in NeuroPilot and run inference. In order to verify the accuracy of our results, we also ran Pytorch's original method to see if the output was the same, which indicates we had a correct answer.

```
def build_model(torch_path):
    model = DeePixBiS()
    model.load_state_dict(torch.load(torch_path))
    model.eval()

    return model

def build_on_tvm(model, use_nir):
    # We grab the TorchScripted model via tracing
    ...
    scripted_model = torch.jit.trace(model, input_data).
     eval()

    # Import the graph to Relay
    # -------------------------
    # Convert PyTorch graph to Relay graph. The input
     name can be arbitrary.
    ...
    shape_list = [(input_name, input_shape)]
    mod, params = relay.frontend.from_pytorch(
     scripted_model, shape_list)

    ...
    # Partition model to IR in neuroPilot
    mod = nir.partition_for_nir(mod, params)
    ...

    # Build the model execution library
    with tvm.transform.PassContext(opt_level=3):
        lib = relay.build(mod, target=target, params=
     params)

    # Graph Module in TVM
    m = graph_executor.GraphModule(lib["default"](dev))
    ...

def inferencing(...):
    ...
    # Set inputs
    ...
    m.set_input(input_name, tvm.nd.array(faceRegion))
    # Execute
    m.run()
    # Get outputs
    tvm_output = m.get_output(0)
    ...
```

Listing 2: Source code for accepting a Pytorch model, dividing the grapth into IR with NeuroPilot and running the inference.

4.2 Object detection model

In terms of the object detection models, since the task is a very common one that everyone is familiar with, we initially adopt two types of models. In the first case, we have the Yolov3[16] model, which is the most commonly used model in object detection tasks. Yolov3 is based on the Darknet framework, which was developed especially for Yolo to improve its performance. By using this model, we can determine the location of the object in our application. The implementation is almost identical to that of the anti-spoofing model. We would just include another simple implementation as part of Listing 3.

```

def build_on_tvm ( model , use_nir ):
    ...
    # Import the graph to Relay
    mod, params = relay.frontend.from_darknet(net, dtype=
     dtype, shape=data.shape)

    # Setup target to run in TVM
    target = tvm.target.Target(...)
    dev = tvm.cpu(0)

    # Partition the grapth to NeuroPilot
    mod = nir.partition_for_nir(mod, params)

    ...
    # The same steps as the previous model
```

Listing 3: The source code for adopting the Yolov3 darknet model, partitioning the graph, and performing inferences.

As it is, the Yolov3 models work perfectly when our application is run on the server side. However, we would like to run our application also on mobile devices, which means we need to find another substitution that will result in a smaller application size. In view of this, we adopt a Mobilenet model. The model is based on the tflite framework, which is commonly employed to develop quantized models, and is more appropriate for mobile devices. The Mobilenet SSD model was used here so that it would also box the area where the object is located. In order to reduce the size of our model, we use a quantized version of our model that converts the weight type from float32 to int8. Using this type of model, we also examined the flow of QNN model integration. As a result, we found that the performance was similar to the original flow.

It is still the case that we build the tflite model first, and then feed it into TVM with TVM.frontend to further partition the graph and to perform inference.

4.3 Emotion detection model

The emotion detection model is developed using Keras, a high-level Tensorflow implementation. The model accepts a face and determines which of the seven basic emotions of the human face is angry, disgusted, fearful, happy, neutral, sad and surprised. Since it was built with Keras, we also promote the ease of running a custom model with Keras, just as this was a custom inference model. We will show some of the model layers in our Keras model in Listing 4 and describe the consequences of running the model in our work.

```

def build_model(weight_path):
    os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

    # Create the model
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='
     relu', input_shape=(48,48,1)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='
     relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Conv2D(128, kernel_size=(3, 3), activation=
     'relu'))
    ...
    model.load_weights(weight_path)
    # prevents openCL usage and unnecessary logging
     messages
    cv2.ocl.setUseOpenCL(False)
    return model

def build_on_tvm ( model , use_nir ):
    ...
    # Import the graph to Relay
    mod, params = relay.frontend.from_keras(model,
     shape_dict)
    # Setup target to run in TVM
    target = tvm.target.Target(...)
    dev = tvm.cpu(0)
    # Partition the grapth to NeuroPilot
    mod = nir.partition_for_nir(mod, params)
    ...
    # The same steps as the previous model
```

Listing 4: The source code for the emotion detection model, whose output would be one of the seven human emotions.

4.4 Package all models together in the application

Once the three types of tasks have been completed and verified as correct, they can be packaged into an application that is dependent upon each other. The video is fed into the application, after the video is sliced into frames, each frame would pass two methods, object detection and face detection. If the object detection model box overlapped the face detector box, we would consider it as a possible candidate for a human face, and then we would run a face anti-spoofing model to distinguish the real from the false. In the last model the faces would be detected according to their emotional state. The detailed steps are listed in Listing 5.

```

def build_model_on_TVM(use_nir): #Build models used in
     application
    ...
    anti_spoof_model = anti_spoofing.build_model(
     torch_path)
    anti_spoof_graph_module = anti_spoofing.build_on_tvm(
     anti_spoof_model, use_nir)

    ...
    emotion_model = emotions_inference.build_model(
     weight_path)
    emotion_graph_module = emotions_inference.
     build_on_tvm(emotion_model, use_nir)

    ...
    object_graph_module = yolo_darknet.build_on_tvm(
     use_nir)

    ...

def inferencing(...):
    ...
    # read video
    ret, frame = cap.read()
    ...

    # face detector & object detector condition

    for (x, y, w, h) in faces:
        # Determine real faces
```

```
        anti_out_frame, spoof_face = anti_spoofing.
    inferencing(...)
        if not spoof_face:
            # Emotion Detection
            emotion_out_frame = emotions_inference.
    inferencing(...)

    ...
```

Listing 5: The steps of application showcase. Object detection and face detection would be applied to each frame. We would consider it a possible human face if the detector box overlapped the face detector box, after which we would run a face anti-spoofing model to tell the true faces from the fakes. Faces were detected based on emotion in the last model.

4.5 Cross compile and deploy to android

As soon as we finish the application on the server side, we would like to bring it to edge devices, such as Android phones. This can be accomplished in two ways. The first objective is to develop an android application that displays the end-to-end result and the second objective is to develop a binary executable to run the prediction algorithm. On Android devices, we prefer to use the latter one to run executable binary files.

To accomplish this, we must build the TVM runtime library at the beginning of the process. TVM stack consists of two critical components, TVM compiler and TVM runtime. Using TVM compiler, we could compile and optimize the model and run it on specific targets using TVM runtime. Therefore, we must build TVM modules on the server side using the TVM compiler, and then we can use TVM runtime on the target devices. Based on the above, the only thing we need to build from TVM is the TVM runtime. This could be done by cross-compiling the TVM runtime for the architectures of our devices.

We use TVM's runtime to run inferencing via inferencing. It contains all the necessary components. The AI models must also be exported. The reason for this is that Python cannot run directly on an Android environment; rather, the library must be exported and imported through a C++ API that is supplied by TVM(Listing 6). After following the above steps, we are able to set up the input and run the model inference on an Android environment.

```

def build_model_on_TVM(use_nir):

    ...
    # Setup target
    target = tvm.target.Target(...)
    dev = tvm.cpu(0)

    ...
    mod, params = relay.frontend.from_keras(model,
     shape_dict)

    ...
    # Partition grapth and setup targets
    mod = nir.partition_for_nir(mod, params, nir_targets
     =[...], nir_debug=True)


    ...
    with tvm.transform.PassContext(opt_level=0):
        lib = relay.build(mod, target=target, params=
     params)


    # After built, export the library for run on android
     environment
    ...
    lib.export_library(dylib_path, ndk.create_shared)

    ...
```

Listing 6: The way to export the library.

5 THE SCHEDULING FOR OPTIMIZING PERFORMANCE

After packaging the models into applications and setting up the android environment for the models, we were inspired by the Efficient Video Captioning on Heterogeneous System Architecture research[8]. This research provides optimization when executing multiple models. Prior to defining the problem, it is essential to determine the time it takes for each model to run on the target devices. As for NeuroPilot's backend, we used a mobile CPU and a mobile APU, which is an AI accelerator. Therefore, we have seven types of permutations: TVM-only, TVM BYOC with mobile CPU, TVM BYOC with mobile APU, TVM BYOC with both mobile CPU and APU, NeuroPilot-only with mobile CPU, NeuroPilot-only with mobile APU, NeuroPilot-only with both mobile CPU and APU, respectively.

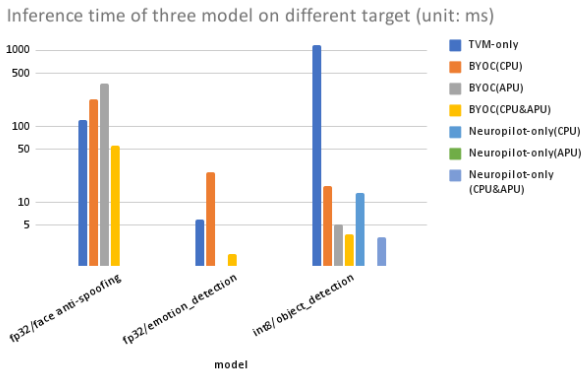

Figure 4: Inference time for different targets. In NeuroPilot-only, since it does not support as many AI operations as TVM, there may have been a lack of statistics regarding the inference time. In the meantime, TVM-only inferences appear to be taking longer than those using NeuroPilot backends. Utilizing TVM BYOC flow is, we believe, a solution which benefits both TVM and NeuroPilot.

The inference time for each of the seven permutations is shown in Figure 4. There are not all the statistics shown on the NeuroPilot-only part, because NeuroPilot does not support as many AI operations as TVM, so there may not be any statistics to show the inference time. It is for this reason that we developed TVM's BYOC

functionality for NeuroPilot. Meanwhile, inference time for TVM-only appears to be longer than others utilizing NeuroPilot backends, which is another reason we believe this is a win-win solution for both TVM and NeuroPilot.

5.1 Computation Scheduling

Since we discovered that TVM BYOC flow is more efficient than TVM-only, and because NeuroPilot does not support as many AI operators as TVM, there is a computation scheduling problem as to where we should assign the graph to the target that is more efficient. As shown in Figure 4, both the face anti-spoofing model and the object detection model are more efficient on the combination of mobile CPU and APU, while the emotion detection model is more efficient on APU alone.

Therefore we could assign them to targets that are more efficient, and this type of computation scheduling is a simple method since it is on the model-level to perform the scheduling task. Another perspective is operation-level, which means we should assign the corresponding efficient targets to each operation. Compared to the model-level, this is more difficult since we need to break the models apart and also consider the I/O time while transferring data between targets. It will be the purpose of our future work to do a harder computation scheduling algorithm. Additionally, we find that the inference time of the anti-spoofing model is longer than the other two. We believe that this is caused by the large number of subgraphs in the model, which we will further investigate to avoid.

5.2 Pipeline Scheduling

In order to handle pipeline scheduling, we observed that there are dependencies among the three models intra-frame, and from the above computation scheduling section, we assigned the various models to their most efficient targets. Dependencies are as follows; the anti-spoofing model waits for the output of object detection model, and the input of emotion detection model is the output of anti-spoofing model. Another perspective that should be considered is the resource perspective. Models could not utilize the same resources at the same time, including mobile CPU and APU.

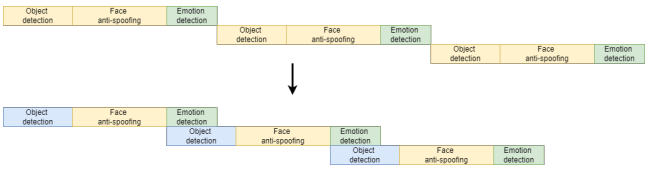

Figure 5: An early prototype of pipeline scheduling among the models. The color yellow indicates that we run at CPU+APU, the color green indicates that we run at APU-only, and the color blue indicates that we run at CPU only. By changing the object detection model from CPU+APU to CPU only, we could guarantee the exclusive use of resources.

Our solution to this problem is to draw upon the concept of the concatenation algorithm[13], and present an early prototype for pipeline scheduling, which can be seen in Figure 5. A problem with the face anti-spoofing model is that we partition it into too many subgraphs, so we should still use the combination of mobile APU and CPU. For the object detection model, we have selected to run it in mobile CPU-only mode. In this manner, we will be able to ensure exclusive resource usage, that is, we will not use the same resource simultaneously for multiple models. By doing so, we would be able to implement a pipeline algorithm that could parallelize the execution of object detection and emotion detection models.

6 EXPERIMENTS

Our experiments evaluated a variety of models of the inference time from the seven permutations, including TVM-only, TVM BYOC with mobile CPU, TVM BYOC with mobile APU, TVM BYOC with both mobile CPU and APU, NeuroPilot-only with mobile CPU, NeuroPilot-only with mobile APU, NeuroPilot-only with both mobile CPU and APU, respectively. Aside from the three models used in our application showcases, we also tested the inference time of other models under different conditions. Models include densenet, inception resnet v2, inception v3, inception v4, mobilenet v2, and nasnet. Additionally, we test the quantized versions of the models of Inception v3 and Mobilenet v1 and v2. Table 1 illustrates the models and their data types, which are either float32 or int8.

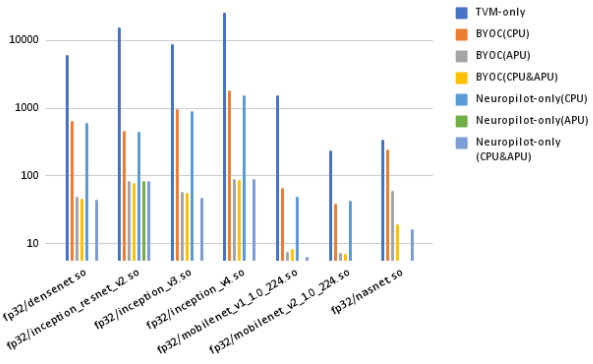

Figure 6: Inference time for more models. Results show the same pattern. There may have been an absence of statistics regarding the inference time in NeuroPilot-only since it does not support as many AI operations as TVM. TVM-only inferences appear to be taking longer than those utilizing NeuroPilot backends. TVM BYOC is an ideal solution that both TVM and NeuroPilot can enjoy.

Model	Data Type
densenet	float32
inception resnet v2	float32
inception v3	float32
inception v4	float32
mobilenet v1	float32
mobilenet v2	float32
nasnet	float32

Table 1: Models used for testing and their data types.

Our experiments were performed on OPPO Reno4 Z 5G, with detailed specifications in Table 2.

OS	Android 11
Chipset	MediaTek MT6873V Dimensity 800
CPU	4x2.0 GHz Cortex-A76 & 4x2.0 GHz Cortex-A55
GPU	Mali-G57 MC4
APU	MediaTek APU 3.0

Table 2: Specifications of experiment environment, OPPO Reno4 Z 5G

The inference time of those is depicted in Figure 6. The results indicate that TVM is applicable to a variety of machine learning frameworks and a wide range of AI operations. However, TVM is not able to make use of vendor-supplied hardware. Hence, combining them via TVM BYOC flow provides a win-win situation. Aside from the results of the experiment, we packaged the models into an application showcase that will allow inferences to be performed on mobile devices.

7 CONCLUSION

Given the trend of running inference on edge devices, we spot that it is necessary to bridge the gap between TVM, the popular AI compiler, and NeuroPilot, which is supported by vendor-supplied accelerators. The TVM BYOC flow provides us with the opportunity to enable the solution.

Apart from that, we developed an application showcase that contains a variety of models derived from various machine learning frameworks and deployed them to the mobile environment. In order to optimize our performance, we propose an early idea for a pipeline scheduling method, and we are currently developing the algorithm for automatically pipeline scheduling of different models and even operations within models in order to enhance efficiency.

ACKNOWLEDGMENTS

This work is supported in part by Taiwan MOST and Mediatek.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Yuan-Ming Chang, Chia-Yu Sung, Yu-Chien Sheu, Meng-Shiun Yu, Min-Yih Hsu, and Jenq-Kuen Lee. 2021. Support NNEF execution model for NNAPI. *The Journal of Supercomputing* 77, 9 (2021), 10065–10096.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[4] Tung-Chien Chen, Wei-Ting Wang, Kloze Kao, Chia-Lin Yu, Code Lin, Shu-Hsin Chang, and Pei-Kuei Tsung. 2019. NeuroPilot: A cross-platform framework for edge-AI. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 167–170.

[5] Tai-Liang Chen, Yi-Ru Chen, Meng-Shiun Yu, and Jenq-Kuen Lee. 2021. NNBlocks: a Blockly framework for AI computing. *The Journal of Supercomputing* 77, 8 (2021), 8622–8652.

[6] Anjith George and Sébastien Marcel. 2019. Deep pixel-wise binary supervision for face presentation attack detection. In *2019 International Conference on Biometrics (ICB)*. IEEE, 1–8.

[7] Yury Gorbachev, Mikhail Fedorov, Iliya Slavutin, Artyom Tugarev, Marat Fatekhov, and Yaroslav Tarkan. 2019. Openvino deep learning workbench: Comprehensive analysis and tuning of neural networks inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 0–0.

[8] Horng-Ruey Huang, Ding-Yong Hong, Jan-Jan Wu, Pangfeng Liu, and Wei-Chung Hsu. 2021. Efficient Video Captioning on Heterogeneous System Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1035–1045.

[9] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[10] keras team. [n.d.]. *Home - Keras Documentation*. keras-team. Retrieved April 11, 2020 from https://keras.io/

[11] Ming-Yi Lai, Chia-Yu Sung, Jenq-Kuen Lee, and Ming-Yu Hung. 2020. Enabling android nnapi flow for tvm runtime. In *49th International Conference on Parallel Processing-ICPP: Workshops*. 1–8.

[12] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[13] Pangfeng Liu and Jan-Jan Wu. 2019. Task scheduling techniques for deep learning in heterogeneous environment. In *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE, 141–147.

[14] ONNX. [n.d.]. *ONNX | Home*. onnx. Retrieved April 20, 2020 from https://onnx.ai/

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8026–8037. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[16] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[17] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A New IR for Machine Learning Frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 58–68. https://doi.org/10.1145/3211346.3211348