TensorFlow Lite Heterogeneous Execution with TI Deep Learning Offload

Hongmei Gou and Pekka Varis

***Abstract* – TensorFlow Lite is the most popular open source deep learning runtime to enable on-device inference for mobile and embedded devices. It has been integrated in Processor SDK Linux to run on Arm cores for all Sitara devices. TI Deep Learning (TIDL) supports high performance computation for core deep learning operators on Embedded Vision Engine (EVE) subsystems, and C66x Digital Signal Processor (DSP) cores. It allows execution of a TensorFlow Lite model only when all of its operators are supported by TIDL. As TIDL supports 20 operators and TensorFlow Lite supports 120 operators, this becomes a severe restriction for customers. This paper presents a method to enable running all TensorFlow Lite models, with operators supported by TIDL offloaded to EVE/DSP for acceleration, and operators not supported by TIDL running on Arm. This heterogeneous execution with TIDL offload is demonstrated on Sitara AM5729 family of devices.**

***Keywords*: Tensorflow Lite, TI Deep Learning, heterogeneous execution, subgraph partitioning, computation offload**

INTRODUCTION

As the most popular open source deep learning runtime for mobile and embedded devices, TensorFlow Lite (called as TFLite in short) is a lightweight solution of TensorFlow, enabling on-device inference with low latency and a small binary size [1]. TFLite currently supports a limited subset of TensorFlow operators that have been optimized for on-device use, narrowing thousands of operators in Tensorflow to around 120 operators in TFLite. In order to enable low latency inference for deployment to edge devices, TFLite provides pre-trained TFLite models [2] as a starting point. To further reduce the latency, post-training quantization can be applied to leverage faster fixed-point computation. For even lower latency and without a compromise in accuracy, quantization-aware training can be performed [3].

TI's Sitara processors form a scalable portfolio integrating Arm cores with flexible peripherals and application-specific accelerators. With single to multicore Arm processors through a unified software platform, Sitara processors provide optimal SoC solutions for various end markets. For instance, AM5729 family of devices has two Arm cores, as well as two C66x DSP cores and up to four EVE subsytems as hardware accelerators for enhanced data processing power. Along with the Arm cores, EVE/DSP accelerators can greatly boost the deep learning inference performance of convolutional neural networks on embedded devices.

TI's Processor SDK Linux provides the core foundation and building blocks for developing embedded applications and mainline Linux allows easy integration of various open source packages and applications. As one example, the open source TFLite has been integrated in Processor SDK Linux, and all TFLite models can run on Arm cores for all Sitara devices. This is shown in the dashed box on the left in Fig.1.

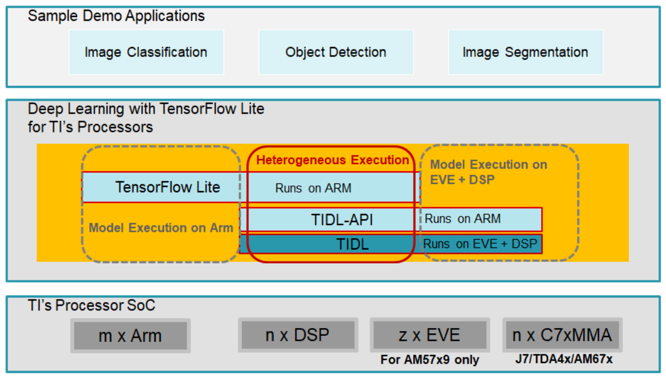


Fig. 1 Deep learning with TFLite for TI's processors: Arm only, TIDL only, and heterogeneous execution.

To utilize the processing power of EVE/DSP for accelerating deep leaning inference on TI's processors, TI Deep Learning (TIDL) [4] is developed as an efficient building block to process around 20 most computation heavy operators, such as convolution, deconvolution, pooling, and element wise operations. The operators supported by TIDL, along with any constraints for the operators' attributes, can be maintained as an allowlist. An example of a constraint is the size of the input tensor. For Sitara AM5729 devices, TIDL API [5] provides a common abstraction for computation on EVE/DSP for deep learning inference user space applications on the Arm. TIDL import tool [6] is developed to convert a deep learning model to TIDL format so that it can be dispatched to EVE/DSP. If all the operators of a TFLite model are supported by TIDL, it can be dispatched to EVE/DSP. This is shown in the dashed box on the right in Fig.1.

As TFLite supports around 120 operators while TIDL supports around 20 and with constraints on some operators' attributes, it is likely that there are many TFLite models which cannot run with TIDL alone. To enable execution of any TFLite models on TI's processors while utilizing the EVE/DSP hardware accelerators to boost the inference performance, this paper presents a method of heterogeneous execution of a TFLite model on the Arm and on the hardware accelerators. A TFLite model is first compiled offline to be compatible with the TIDL format, and then re-serialized as a standard TFLite model. This compiled TFLite model can then run through the TFLite runtime interface on the Arm, offloading the operators supported by TIDL to EVE/DSP for acceleration while executing the operators which are not supported by TIDL on the Arm. For a user this looks like TFLite on ARM that just runs faster. This is shown in the solid box in the middle in Fig. 1, and implemented for Sitara AM5729 family of devices.

Prior art of this work is Edge TPU, Google's purpose-built ASIC designed to run deep leaning inference at the edge [7]. Edge TPU supports TFLite models that are 8-bit quantized and use only the operations listed in [8]. A TFLite model is first compiled by the Edge TPU Compiler. At the first point in the model graph where an unsupported operation occurs, the compiler partitions the graph into two parts. The first part of the graph that contains only supported operations is compiled into a custom operation, and everything else remains the same [9]. After this offline step, inference is performed on a model which has been compiled by the Edge TPU compiler. The Edge TPU custom operation inside the compiled model is registered with TFLite, and dispatched from TFLite runtime to the Edge TPU [10]. The unsupported operations execute on the host CPU.

APPROACH

A. Overview

To enable TIDL offload of a TFLite model to hardware accelerators of TI's processors, a two-stage approach similar to Edge TPU is used, as shown in Fig. 2. The blocks on the left illustrate the offline TIDL compilation for a TFLite model, by enhancing the TIDL import tool with subgraph partitioning and reserialization to create a TIDL compatible TFLite model. The blocks on the right show the TFLite inference on the compiled TFLite model. The TIDL subgraph is registered as a custom operator of TFLite, and executed on EVE/DSP hardware accelerators via TIDL API. The subsections below describe details for both stages.

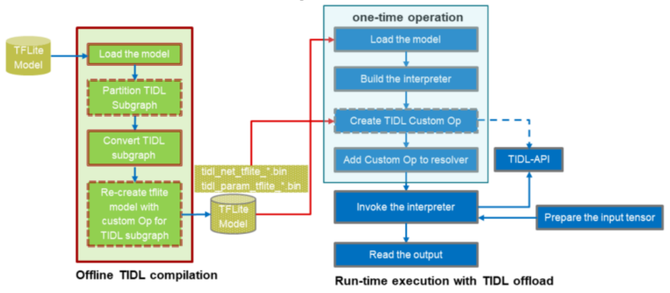
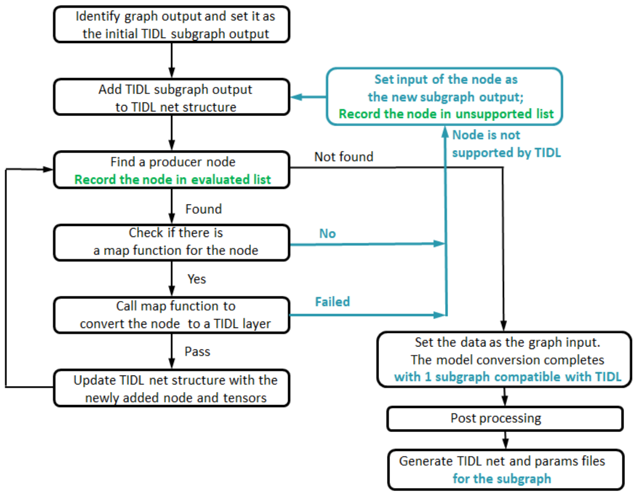


Fig. 2: Framework for TFLite heterogeneous execution with TIDL.

B. Offline TIDL compilation of TFLite model

At the first stage, as part of the software build process the TIDL import tool is enhanced to compile a TFLite model. It takes a TFLite model as the input, and generates a model compatible with TIDL and still in the TFLite model format.

As shown in Fig.3, the TIDL import process starts from the graph output, and retrieves its producer node. If this node is supported by TIDL, it will be converted into the TIDL format, and the search continues upward to find another producer node. If there is a node which is not supported by TIDL, the old import process just exits and reports the import failure. With the target of heterogeneous execution which allows operators not supported by TIDL, the import process continues but uses the input of this unsupported node as the output of the updated TIDL subgraph. The same process continues until no producer node can be found. This completes the model conversion and subgraph partitioning in a single-pass process. One TIDL subgraph is identified and converted into the TIDL format, described by a net description file and a parameter description file. In the import process, the nodes which have been evaluated and the nodes which are not supported by TIDL are recorded to facilitate recreation of the TFLite model later.


Identify graph output and set it as the initial TIDL subgraph output
Add TIDL subgraph output to TIDL net structure
Set input of the node as the new subgraph output; Record the node in unsupported list
Node is not supported by TIDL
Not found
Find a producer node
Record the node in evaluated list
Found
Check if there is a map function for the node
No
Yes
Call map function to convert the node to a TIDL layer
Failed
Pass
Update TIDL net structure with the newly added node and tensors
Set the data as the graph input. The model conversion completes with 1 subgraph compatible with TIDL
Post processing
Generate TIDL net and params files for the subgraph
Fig. 3: TIDL import execution flow with enhancement for subgraph partitioning (marked in bold and in blue/green color).

The left side of Fig.4 shows the subgraph partitioning result after applying the algorithm above to an example TFLite model, Deeplab v3 pixel segmentation [11]. The tool Netron [12] is used to generate the graphical representation of the networks. This model has three nodes of the ResizeBilinear operator (black box in the red explosion), which is not supported by TIDL. It also has an AveragePool2D node (green box in the red explosion) which has a kernel size of 33x33 while TIDL supports up to 9x9 kernels for pooling. The algorithm traverses from the graph output to the AveragePool2D node, and then does the partitioning right up to this unsupported node. The partitioned TIDL subgraph is shown with the larger gray outline box at the top left. The compute heavy nodes such as Conv2D on larger tensors inside the cloud are not drawn to limit the size of the figure. This pattern can be repeated enabling multiple TIDL subgraphs to be offloaded.

After the TIDL subgraph is partitioned and converted, the next step is to create a TFLite model which represents the TIDL subgraph as a single custom operator and keeps the operators outside of the subgraph (if there are any) as in the original model. The right side of Fig. 4 shows what a TIDL compiled TFLite model looks like for the Deeplab v3 model.

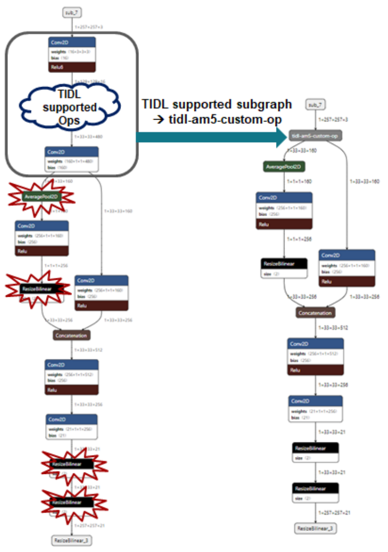


Fig. 4 Original and TIDL compiled Deeplab v3 TFLite model.

From the subgraph partitioning and the original TFLite model, we have known the information listed below.

1) Nodes in the TIDL subgraph
2) Nodes outside the TIDL subgraph
3) Input and output of the TIDL subgraph
4) Input and output of the graph

To create the TIDL compiled TFLite model which is in the FlatBuffers format [13], the network needs to be serialized. As shown in Fig. 5, the major tasks here include exporting tensors, buffers, and operators, as well as creating the opcode table. A model is created after all these elements are established.

The tensor exporting loops through all the tensors in the original model and writes a tensor in the output model only when it connects to a node outside the offloaded TIDL subgraph, or it is the input or output of the subgraph. Using this process, the number of tensors can be greatly reduced, and the tensors are re-indexed. When a written tensor has buffer data, the buffer is also exported with a new buffer index assigned.

The first exported operator is called tidl-am5-custom-op, which encapsulates all the nodes in the TIDL subgraph. File names for the two TIDL format files created from the import process for the subgraph are written as custom options of this custom operator. Rather than the file names, binary data of these files can also be written as custom options inside this operator to have the output TFLite model contain the complete information needed by the runtime. Following the custom operator are the rest of the nodes of the original network model outside the TIDL subgraph, with built-in options copied from the original model. Input and output tensors for each of these operators (including tidl-am5-custom-op) are identified and associated with the operator with the new tensor indices from the tensor exporting above.

For the opcode table, "tidl-am5-custom-op" is added along with BuiltinOperator_CUSTOM type for the TIDL custom operator. For the nodes outside the TIDL subgraph, their built-in codes (e.g., BuiltinOperator_CONV_2D) remain and are added to the table.

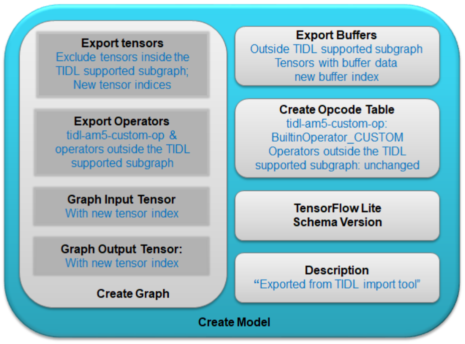
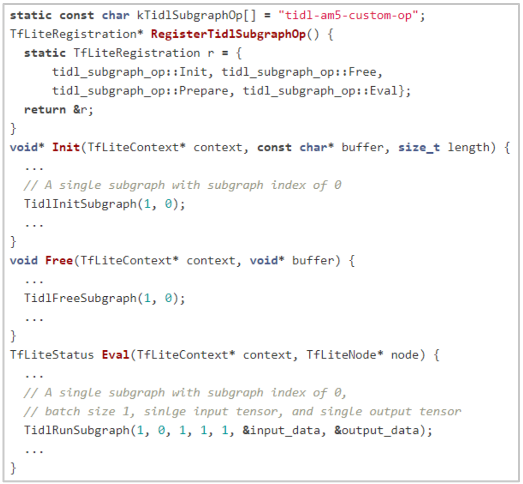


Fig. 5 Serialization to create the output model in FlatBuffers format.

C. TFLite runtime inference with TIDL offload

Once the compilation has been completed offline the part that is executed on the embedded device is running inference with the TIDL compiled TFLite model. For typical vision applications the input tensor is a frame or a cropped region in the frame. In order to let the TFLite runtime recognize the newly created custom operator (tidl-am5-custom-op), a custom kernel [14] is created and registered with TFLite runtime. As shown in Fig. 6, the kernel defines Init(), Free(), Prepare(), and Eval() functions. The Init(), Free(), and Eval() functions include calls to TIDL APIs to initialize, free, and run the subgraph on EVE/DSP.


```
static const char kTidlSubgraphOp[] = "tidl-am5-custom-op";
TfLiteRegistration* RegisterTidlSubgraphOp() {
  static TfLiteRegistration r = {
      tidl_subgraph_op::Init, tidl_subgraph_op::Free,
      tidl_subgraph_op::Prepare, tidl_subgraph_op::Eval};
  return &r;
}
void* Init(TfLiteContext* context, const char* buffer, size_t length) {
  ...
  // A single subgraph with subgraph index of 0
  TidlInitSubgraph(1, 0);
  ...
}
void Free(TfLiteContext* context, void* buffer) {
  ...
  TidlFreeSubgraph(1, 0);
  ...
}
TfLiteStatus Eval(TfLiteContext* context, TfLiteNode* node) {
  ...
  // A single subgraph with subgraph index of 0,
  // batch size 1, sinlge input tensor, and single output tensor
  TidlRunSubgraph(1, 0, 1, 1, 1, &input_data, &output_data);
  ...
}
```
Fig. 6: Register tidl-am5-custom-op and interface it with TIDL APIs.

After the custom kernel for tidl-am5-custom-op is registered with TFLite, it needs to be added to the model resolver (as shown in Fig. 7) when building the TFLite interpreter. The other operators in the TIDL compiled TFLite model are built-in operators of TFLite, and can be interpreted natively on at Arm core. Now the typical TFLite inference steps [15] can be used to run the model as listed below:

1) load the model (initialization)
2) build the interpreter (initialization)
3) set the input tensor
4) invoke inference
5) read output tensor values

Steps 3 through 5 are run per each input tensor, e.g., frame in a sequence of images. Under the hood, the tidl-am5-custom-op will be dispatched to EVE/DSP for acceleration, while the other operators will be run on the Arm.

```
std::unique_ptr<tflite::Interpreter> BuildTfliteInterpreter(
    const tflite::FlatBufferModel& model, int num_threads) {
  tflite::ops::builtin::BuiltinOpResolver resolver;
#ifdef TIDL_OFFLOAD
  resolver.AddCustom(tidl::kTidlSubgraphOp, tidl::RegisterTidlSubgraphOp());
#endif
```

Fig. 7: Add tidl-am5-custom-op to the model resolver.

Demo

To demonstrate the proposed method, the mobilenet v1 model (*mobilenet_v1_1.0_224.tflite*) is chosen as it has been validated with the TIDL library. Although all operators of this model are supported by TIDL, to test partitioning a specific tensor can be passed to the TIDL import tool from the command line to determine the cutoff for a subgraph. Fig. 8 shows the example command and the output TFLite model for creating a subgraph of all nodes before the tensor called *MobilenetV1/MobilenetV1/Conv2d_13_pointwise/Relu6*.

```
#1. Prepare the input: tensorflow lite model
cd /usr/share/ti/tidl/utils
mkdir -p ./test/testvecs/config/tflite_models
mkdir -p ./test/testvecs/config/tidl_models/tflite/
cp /usr/share/tensorflow-lite/demos/mobilenet_v1_1.0_224.tflite ./test/testvecs/config/tflite_models
#2. Run tidl_model_import.out with the output tensor specified
tidl_model_import.out ./test/testvecs/config/import/tflite/tidl_import_mobileNetv1.txt
MobilenetV1/MobilenetV1/Conv2d_13_pointwise/Relu6
```

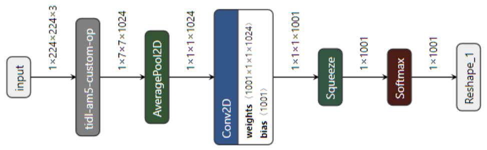


Fig. 8: Compile mobilenet v1 model for heterogeneous execution.

Along with the output model, the compilation creates two TIDL format files for the subgraph:

- *tidl_net_tflite_mobilenet_v1_1.0_224.bin*
- *tidl_param_tflite_mobilenet_v1_1.0_224.bin*

The TIDL API subgraph handling requires supplying a TIDL specific subgraph0.cfg file which specifies the two TIDL format files above, as well as the boundary conversion parameters related to model quantization for the TIDL subgraph. Top of Fig. 9 shows the example subgraph0.cfg.

To evaluate the performance of heterogeneous execution with TIDL offload, we have built a classification demo using TFLite runtime with the tidl-am5-custom-op kernel added to the interpreter as discussed before. Bottom of Fig. 9 shows the example command to run classification using the compiled mobilenet v1 model (*mobilenet_v1_1.0_224_tidl_am5.tflite*), with subgraph0.cfg above consumed by the TIDL API. The original mobilenet v1 model can also be supplied to the classification demo to obtain the Arm only performance for comparison. As shown in Table 1, the inference time using heterogeneous execution is 32% from the Arm only execution, with slight accuracy loss of 0.03. This demonstrates the performance advantage of our method for enabling an arbitrary TFLite model on AM5729 while leveraging TIDL offload.

```
netBinFile      = /usr/share/ti/tidl/utils/test/testvecs/config/tidl_models/
                  /tflite/tidl_net_tflite_mobilenet_v1_1.0_224.bin
paramsBinFile   = /usr/share/ti/tidl/utils/test/testvecs/config/tidl_models/
                  tflite/tidl_param_tflite_mobilenet_v1_1.0_224.bin
# The input is in NHWC format and ranges [-1,1]
inConvType = 0
inIsSigned = 1
inScaleF2Q = 128
inIsNCHW = 0
# The output is in NHWC format and ranges [0,6]
outConvType = 0
outIsSigned = 0
outScaleF2Q = 42.5
outIsNCHW = 0
```

```
cd /usr/share/tensorflow-lite/demos
./tflite_classification -m /usr/share/ti/tidl/utils/test/testvecs/config/ \
                           tflite_models/mobilenet_v1_1.0_224_tidl_am5.tflite \
                        -i ../examples/grace_hopper.bmp -l ../examples/labels.txt -p 1
```

Fig. 9: Example subgraph0.cfg and command to run classification.

TABLE I
PERFORMANCE COMPARSION

	Accuracy	Inference Time (ms)
Arm only	0.86	308.14
Heterogeneous	0.83	98.89

Conclusion

In this paper, we have provided a solution to run any TFLite models with TIDL offload to hardware accelerators, through the TFLite runtime interface. By enhancing the TIDL import tool and leveraging TIDL API, we have demonstrated that the TIDL offload can be inserted in TFLite nearly seamlessly in a development flow analogous to Google's Edge TPU.

References

[1] TFLite guide, http://www.tensorflow.org/lite/guide
[2] TFLite hosted models
[3] TFLite model optimization
[4] Deep Learning Inference For Embedded Applications
[5] TIDL API
[6] TIDL import tool
[7] Edge TPU, https://cloud.google.com/edge-tpu
[8] Edge TPU supported operations
[9] Edge TPU Compiler, https://coral.ai/docs/edgetpu/compiler/
[10] Edge TPU inference, https://coral.ai/docs/edgetpu/inference/
[11] TFLite segmentation model
[12] Netron tool, https://github.com/lutzroeder/netron
[13] FlatBuffers, https://google.github.io/flatbuffers/
[14] TFLite custom operators
[15] TFLite inference