Bring Your Own Codegen to Deep Learning Compiler

Zhi Chen, Cody Hao Yu, el al.
Amazon Web Services. Inc

Deep Learning Compiler Study
May 20, 2021

Hyunwoo Cho
(hyunwoocho@sogang.ac.kr)

Background

- Deep neural networks (DNNs) have been ubiquitously applied in many applications, and accelerators are emerged as an enabler to support the fast and efficient inference tasks of these applications

Problem

- From the view of hardware vendor

Time consuming reinvent process

  - ✓ To achieve high model coverage with high performance, each accelerator vendor has to develop a full compiler stack to ingest, optimize, and execute the DNNs.
  - ✓ The vendors have to contiguously update their hardware and/or software to cope with the rapid evolution of the DNN model architectures and operators.

- From the view of user
  - ✓ How can we handle not-supported or inefficiently-supported operations of specific target network in target accelerator?
  - ✓ Most of the hardware accelerators perform optimization mainly focused on specific operations such as MAC, Conv2D, etc. Can we jointly use optimized network blocks with generally optimized computational graph with Deep Learning Complier (DLC)?

Problem

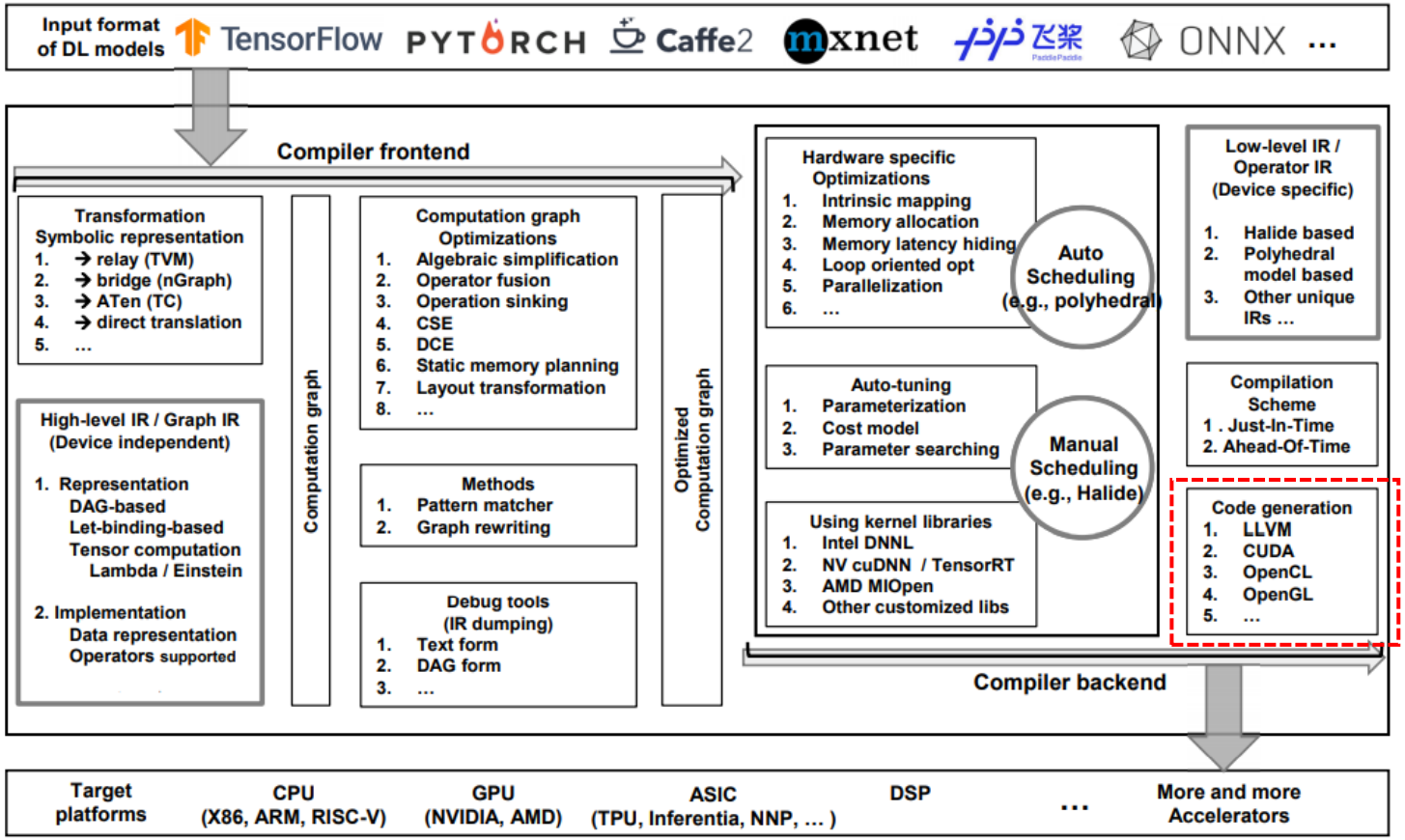
- Failing to execute
  - ✓ Due to the emergence of new models with architectural modification or new operations, existing accelerators may completely fail to support models with complicated structures.
- Inefficient to execute
  - ✓ Fail to achieve high performance for a model even if its architecture is a simple dataflow, because it may contain a portion of operators that are not executed in a determined order but with complex control logic.
- Demanding to execute
  - ✓ Even if all operators in a model can be well supported, each vendor still has to spend a considerable amount of engineering efforts to develop a full compiler stack to ingest, optimize, and compile the model

BYOC decouples a deep learning compiler (DLC) to two parts.

Shared part : The shared part can be treated as a DLC for general processors such as CPUs and GPUs.
Accelerator-exclusive part : Vendors can contribute the features required to execute new emerged models to the shared part and make them available on general devices to benefit each other.

Bring Your Own Codegen (BYOC)

- Proposed BYOC framework is a unified framework for accelerator vendors to integrate their code generators with minimum efforts. This framework aims to reuse frontend optimizations of DLC and Relay optimizations as many as possible.
- Users can use customized code generator or run-time on specific part of computational graph.



Partially replace code generator

Li, Mingzhen, et al. "The deep learning compiler: A comprehensive survey." *IEEE Transactions on Parallel and Distributed Systems* 32.3, 2020.

Bring Your Own Codegen (BYOC)

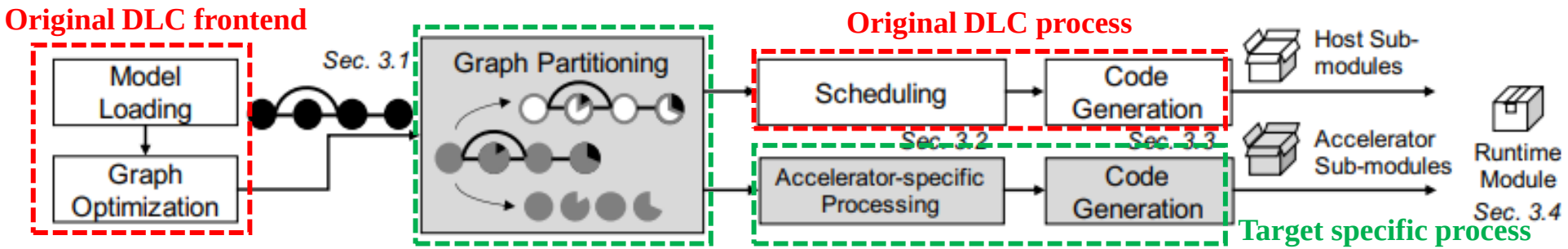


Figure 2: The *Genesis* compilation overview. The rectangles in grey are major components in *Genesis*. The deep learning model graph represents the general R-CNN. Each dot in the graph denotes a R-CNN module in Figure 1.

➢ Process of BYOC framework

1. Graph Partitioning : partition into sub-graphs and determine whether run with host or target
2. Accelerator-specific Processing : perform hardware-specific optimization
3. Code Generation : generate code into each monolithic executable module
4. Runtime : managing each executable module with execution engine

Bring Your Own Codegen (BYOC)

- Graph Partitioning
  - ✓ In order to guarantee the successful and performant execution, we design a partitioning module for users to flexibly cut their model graphs into various regions/subgraphs. Only the accelerator friendly regions are offloaded, and the rest of the graph is left to the host.

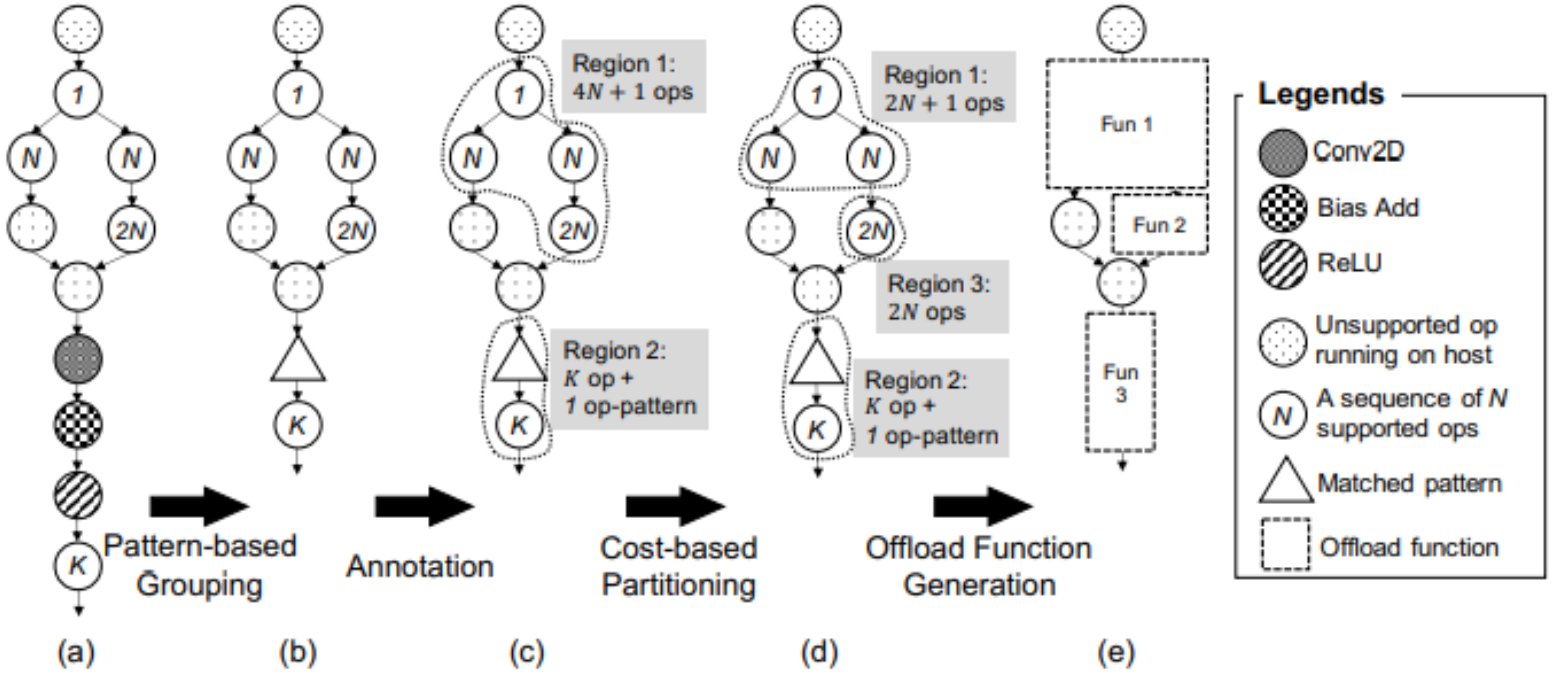
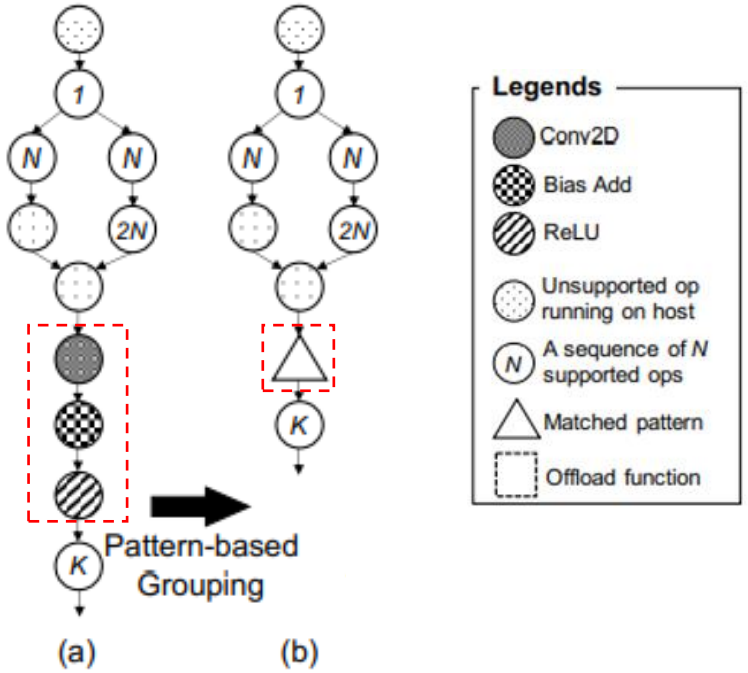


Figure 3: An illustrative example of graph partitioning.

Bring Your Own Codegen (BYOC)

- Graph Partitioning : Pattern-based Grouping
  - ✓ For example, the sequence of Conv2D, Add and ReLU can usually be mapped to a single instruction to minimize overheads in dealing with intermediate results.

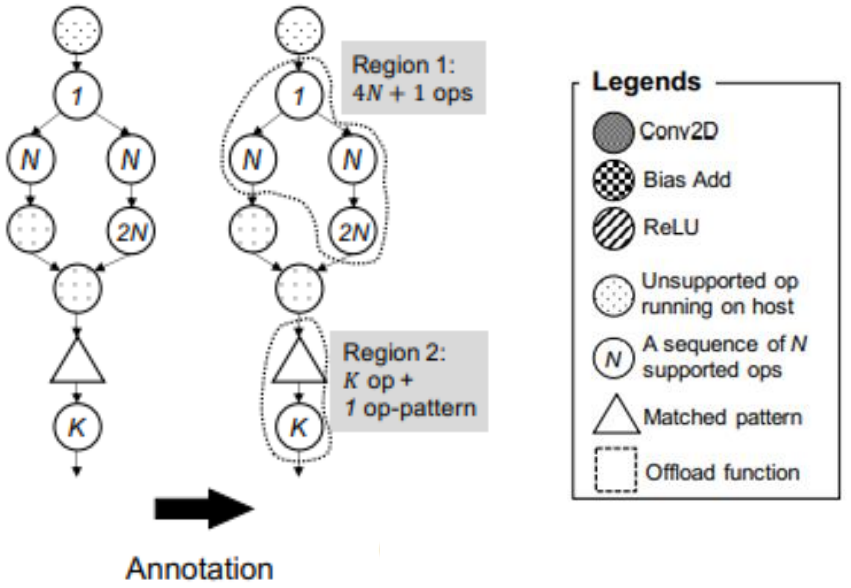
```
def conv2d_pattern():
  data, weight, bias = wildcard(), wildcard(), wildcard()
  conv = is_op("nn.conv2d")(data, weight)
  bias_optional = conv.optional( \
    lambda x: is_op("nn.bias_add")(x, bias))
  return is_op("nn.relu")(bias_optional)
pattern_table = [(conv2d_pattern, "conv2d_pattern")]
```



Bring Your Own Codegen (BYOC)

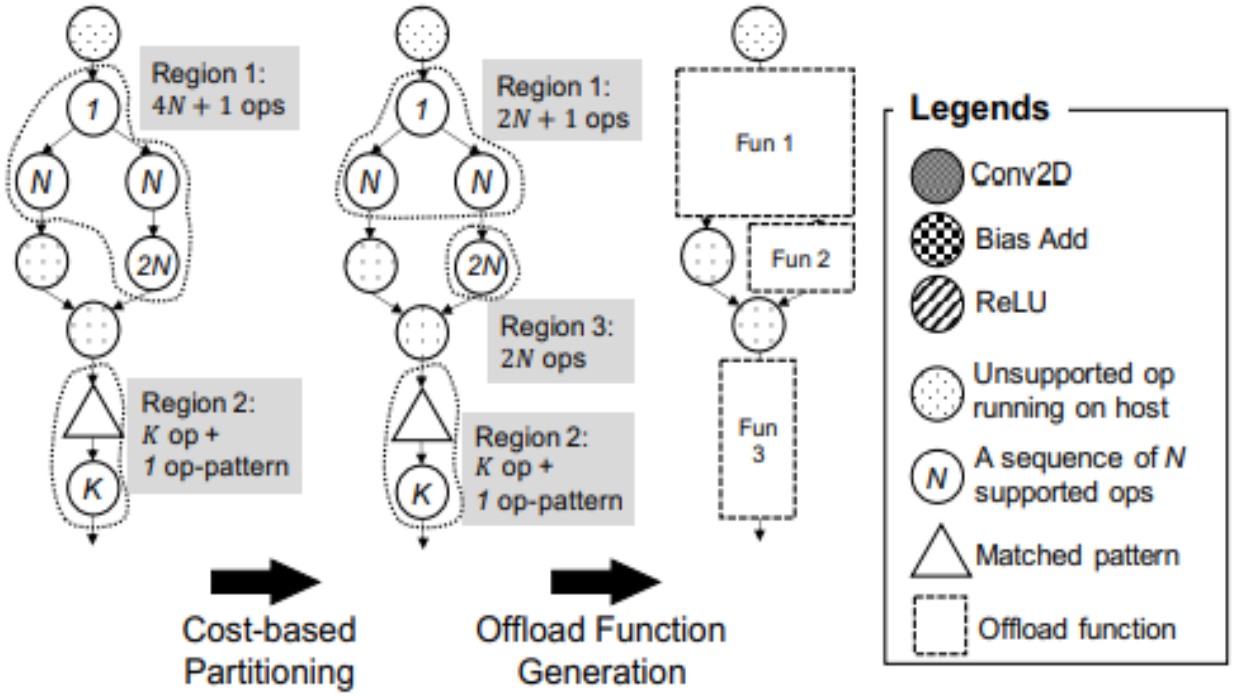
- Graph Partitioning : Annotation
  - ✓ For example, the following code snippet registers a function to indicate that all Conv2D nodes with floating point data type should be annotated and offloaded to myAccel.

```
@register_op_attr("nn.conv2d", "codegen.myAccel")
def annotate_conv2d(expr):
    attrs, args = expr.attrs, expr.args
    if any([x.dtype != "float32" for x in args]):
        return False
    return True
```



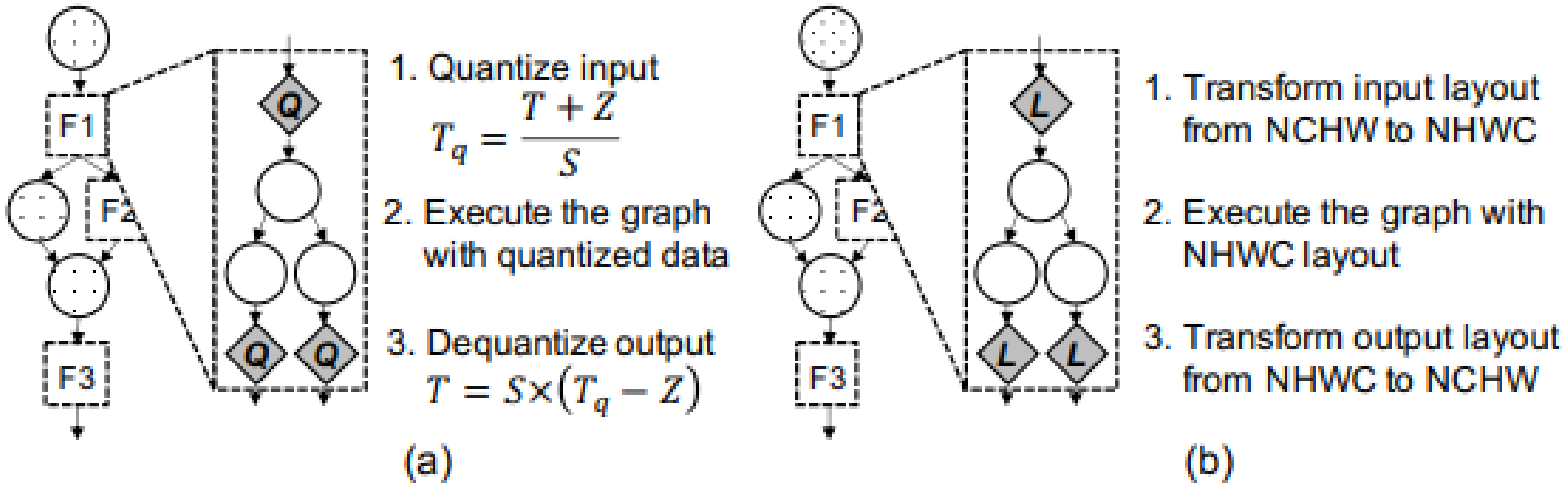
Bring Your Own Codegen (BYOC)

- Graph Partitioning : Cost-based Partitioning
  - ✓ Although greedily merging supported operators to maximize the region size is ideal, it might not be practical for some accelerators due to the resource constraints (e.g., on-chip memory size, number of compute units, etc.)



Bring Your Own Codegen (BYOC)

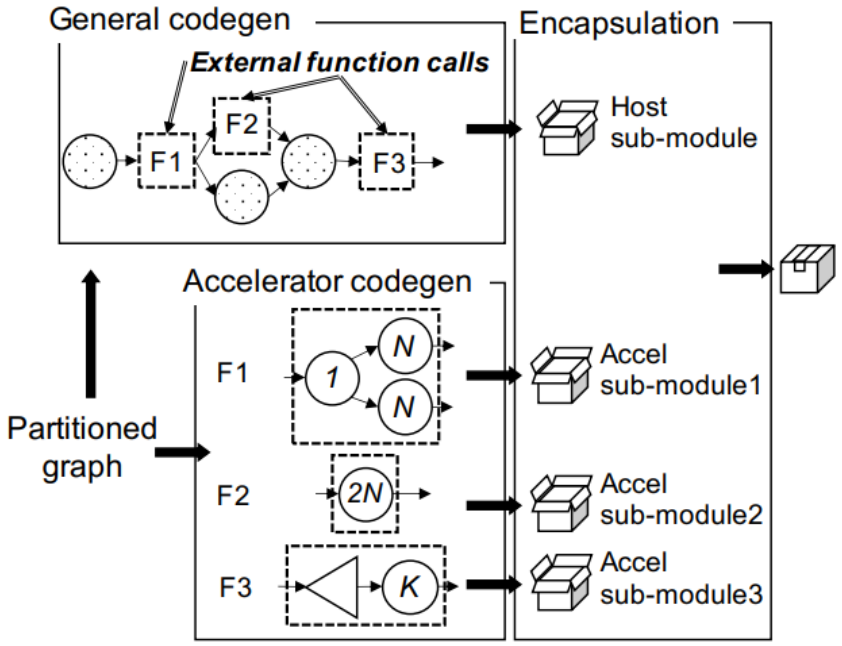
- Accelerator-specific Processing
  - ✓ The regions offloaded to accelerators may require some hardware-dependent optimizations (e.g., fusion, substitution, layout transformation, quantization, etc.)



Bring Your Own Codegen (BYOC)

➢ Code Generation

- ✓ Given a deep learning model graph with partitioned functions, the framework aims to generate a monolithic executable module for model inference.
- ✓ When traversing to a node that remains on the host, we leverage the code generation in existing deep learning compilers, such as TVM and XLA. Most of them are capable of generating code for general devices (e.g., CPU and GPU).
- ✓ When traversing to a node (i.e., a partitioned function) that is annotated with the specific target, we simply generate an external function call as a hook for kernel invocation at runtime.

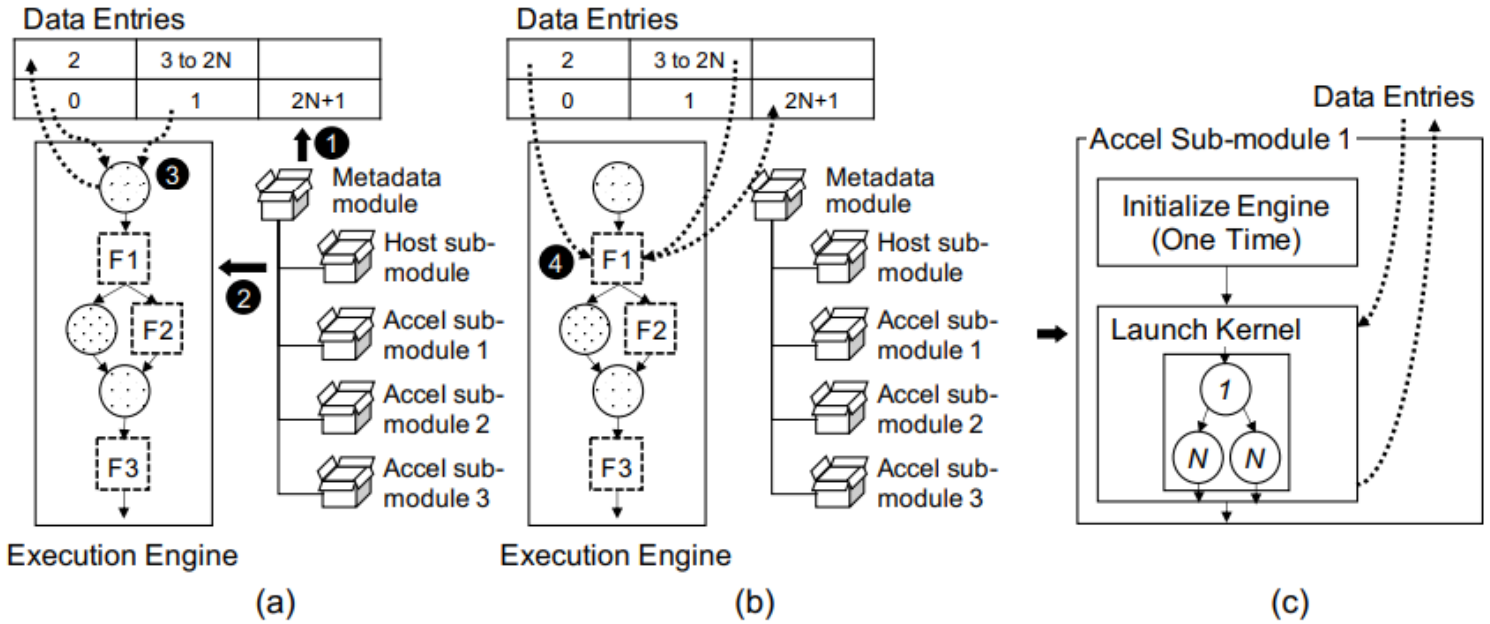


Possible options to represent the generated code format.

- ➢ Option 1: Standard graph representation.
  - ✓ JSON as the default graph format because it has gathered notable popularity in the deep learning community. (TensorRT, ARM CL, etc.)
- ➢ Option 2: Standard C code.
  - ✓ Offers a standard C code generator that can support accelerator's proprietary kernel libraries via emitting the kernel library function calls and linking them together with the host sub-module.
- ➢ Option 3: Custom graph representation.
  - ✓ However, certain accelerators can only load the customized graph representation format which is different from the aforementioned ones. (ARM Ethos-N and Xilinx Vitis AI, etc.)

Bring Your Own Codegen (BYOC)

- Runtime
  - ✓ The runtime module is in charge of executing the model graph and dispatching the operators and subgraphs to the target platform.
  - ✓ A unified module, namely metadata module is designed as a hierarchical module that contains all constants, the host sub-module, and the accelerator sub-modules. Upon the initialization, metadata module loads the constants to the runtime data entries, which are a set of pre-allocated memory buffers on the host or the accelerator. In addition to the constants, data entries also maintain the model inputs and outputs as well as the intermediate results.
  - ✓ The initiated execution engine then starts executing the graph nodes sequentially. The host execution engine now attempts to execute the node F1, which is an external function call to the accelerator.



Bring Your Own Codegen (BYOC)

- Case Study
  - ✓ How many efforts to import code generator to DLC(tested on TVM)?
    - ✓ Arm Ethos-N processor : 2,405 LOC
    - ✓ Xilinx Vitis-AI with Deep Learning Processor Unit : 1,924 LOC
    - ✓ NVIDIA TensorRT : 4,403 LOC
    - ✓ Arm Compute Library : 2,188 LOC
    - ✓ Texas Instruments Deep Learning 3,085 LOC
    - ✓ CoreML : 840 LOC
  - ✓ Performance between partitioned graph (TVM+TensorRT) and end-to-end DLC (TVM)

Model	Offload Ratio (%)		Full-Precision		Half-Precision	
	Node	MAC	Latency (ms)	Speedup	Latency (ms)	Speedup
ResNet-18	100%	100%	4.12	1.07	2.14	2.06
ResNet-50 v1b/v2	100%	100%	9.66	1.53	4.12	3.58
Inception V3	100%	100%	16.13	2.00	6.10	5.30
DarkNet-53	100%	100%	12.81	3.73	5.43	8.81
MobileNet V2 1.0	100%	100%	3.32	1.19	2.22	1.78
VGG19	100%	100%	24.26	0.92	9.78	2.29
SSD 512 MobileNet 1.0	36%	100%	18.43	2.24	11.06	3.73
SSD 300 ResNet-34	41%	100%	34.31	48.44	15.29	108.68
Faster R-CNN ResNet50 v1b	63%	100%	404.51	6.29	99.72	25.53

Table 2: Subgraph number and latency comparison with and without cost-based partitioning of TensorRT integration on NVIDIA Jetson Xavier.

Model	Without		With	
	Total Subgraph #	Latency (ms)	Total Subgraph #	Latency (ms)
ResNet-18	1	4.12	1	4.12
ResNet-50 v1b/v2	1	9.66	1	9.66
Inception V3	1	16.13	1	16.13
DarkNet-53	1	12.81	1	12.81
MobileNet V2 1.0	1	3.32	1	3.32
VGG19	1	24.26	1	24.26
SSD 512 MobileNet 1.0	6	48.13	1	18.43
SSD 300 ResNet-34	6	79.30	1	34.31
Faster R-CNN ResNet50 v1b	21	407.15	2	404.51

Bring Your Own Codegen (BYOC)

- Case Study
  - ✓ Performance between partitioned graph (TVM+Xilinx Vitis AI) and end-to-end DLC (TVM)

Table 3: End-to-end performance of full precision models on Xilinx U250 cloud FPGA. The baseline is compiled by the TVM builtin LLVM code generation and runs on Intel(R) Xeon(R) Gold 6252 CPU.

Model	Latency (ms)	Speedup	Offload Ratio (%)	
			Node	MAC
ResNet-18	2.66	3.91×	96%	99%
ResNet-50 v1b/v2	5.59	4.48×	98%	99%
Inception V3	7.51	4.14×	99%	99%
DarkNet-53	6.78	5.07×	98%	100%
VGG19	15.82	5.52×	80%	99%
SSD 300 ResNet-34.	43.21	2.42×	33%	100%
Faster R-CNN ResNet50 v1b	1,080	1.08×	46%	20%

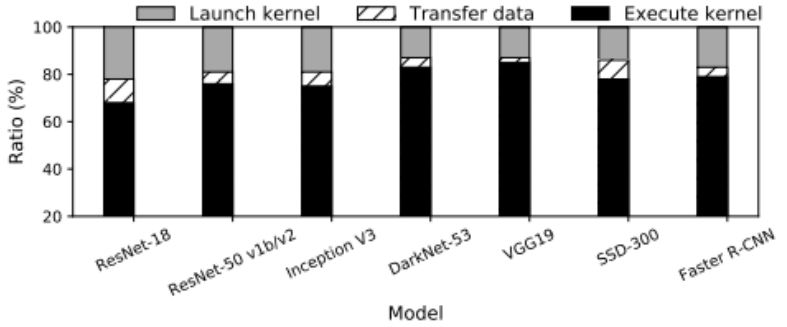


Figure 7: The breakdown of kernel execution on Xilinx U250 cloud FPGA. The data transfer and kernel invocation time take on average 6% and 20%, respectively.

Table 4: End-to-end performance of full precision models on Xilinx Zynq UltraScale+ edge FPGA. The baseline is compiled by the TVM builtin LLVM code generation and runs on ARM Cortex-A53 CPU.

Model	Latency (ms)	Speedup	Offload Ratio (%)	
			Node	MAC
ResNet-18	5.83	34.05×	96%	100%
ResNet-50 v1b/v2	14.7	44.51×	98%	99%
Inception V3	19.47	77.34×	98%	100%
DarkNet-53	18.29	75.01×	98%	100%
MobileNet V2 1.0	4.72	19.11×	99%	100%
VGG19	132.78	10.22×	80%	99%
SSD 512 MobileNet 1.0	352.69	5.47×	28%	100%
SSD 300 ResNet-34	467.91	12.7×	34%	100%
Faster R-CNN ResNet50	49,016	1.29×	46%	20%.

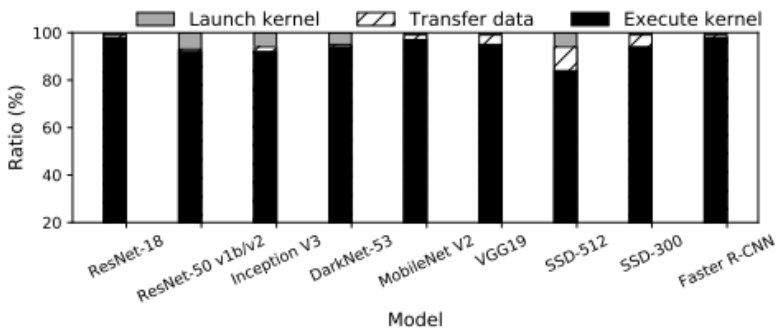


Figure 8: The breakdown of kernel execution on Xilinx Zynq UltraScale+ edge FPGA. Both data transfer and kernel invocation time take on average 3%.

Bring Your Own Codegen (BYOC)

- Usage Case
  - ✓ In case of performing task upon specific acceleration hardware and target task has special operation not supported by the accelerator.
  - ✓ When some operations compiled by DLC showed considerable performance drop and the target hardware accelerator has high-performance kernel libraries to handle those operations.
  - ✓ To test and evaluate custom kernel or code generator.
  - ✓ When performing tasks with special architecture or operations on a hardware accelerator.

Bring Your Own Codegen (BYOC)

- Reference

[1] Chen, Zhi, et al. "Bring Your Own Codegen to Deep Learning Compiler." arXiv preprint arXiv:2105.03215, 2021.

[2] Li, Mingzhen, et al. "The deep learning compiler: A comprehensive survey." IEEE Transactions on Parallel and Distributed Systems 32.3, 2020.

- Demo, Tutorial
  - ✓ Tutorial to integrate TensorRT with TVM
    - ✓ TVM Conf 2020 - Tutorials - Bring Your Own Codegen to TVM – YouTube
  - ✓ Official Tutorial of BYOC with TVM
    - ✓ How to Bring Your Own Codegen to TVM (apache.org)
    - ✓ Bring Your Own Codegen To TVM — tvm 0.8.dev0 documentation (apache.org)

Q & A