A Task Parallelism Runtime Solution for Deep Learning Applications using MPSoC on Edge Devices

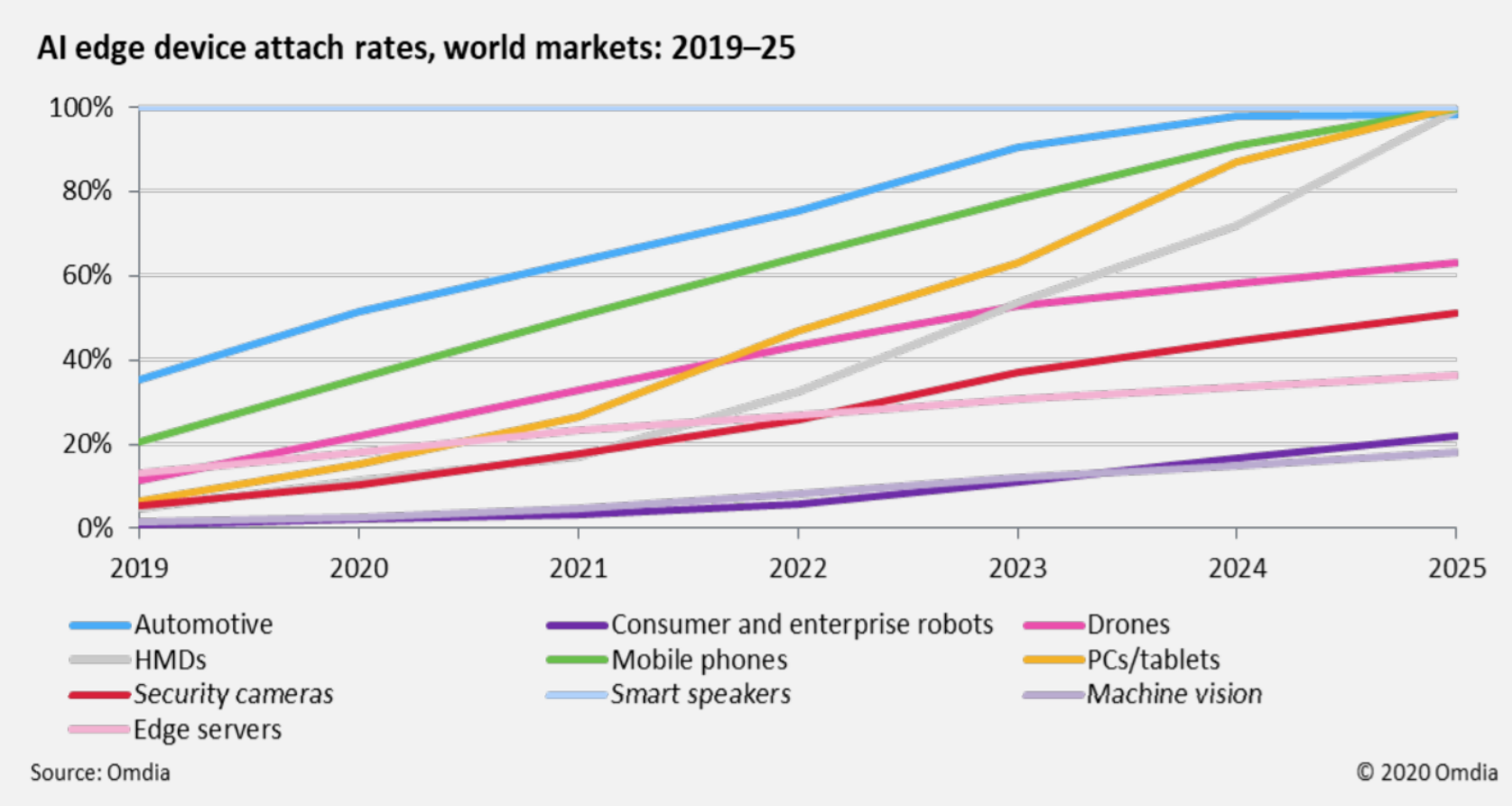
Hua Jiang, Raghav Chakravarthy, Ravikumar V C

Agenda

- Introduction and Background
- Solution Overview
- Static compilation
- Dynamic execution
- Performance
- Open-source
- Conclusion

Introduction and Background

AI On Edge Device Is Booming



Neural Network Compute Process on Edge Device

Deep Learning Neural Network Compute Process

- Design Layer
- Op Structure
- Constraint (FLOPs)

- Operator Computation
- Layer Dependency
- Parallelism

- Deploy → N to M
- Optimization
- Auto Schedule

- Backend Manage
- Control Flow
- Data Management

- Offload Alg to Hardware

CNN, GAN,RNN

AI Deployment Challenges on Edge Device

Data parallelism is challenging on heterogeneous core

Map N AI tasks to M backend heterogenous cores

Neural network task level splitting is complex

Efficient scheduler needed for heterogenous cores

- Different heterogenous backends have different compute performances
- Needs to orchestrate graph between the heterogenous cores
- Efficiently handle and manage control flow and data flow

Our Solution

Adapt task parallelism and pipelining on heterogeneous cores

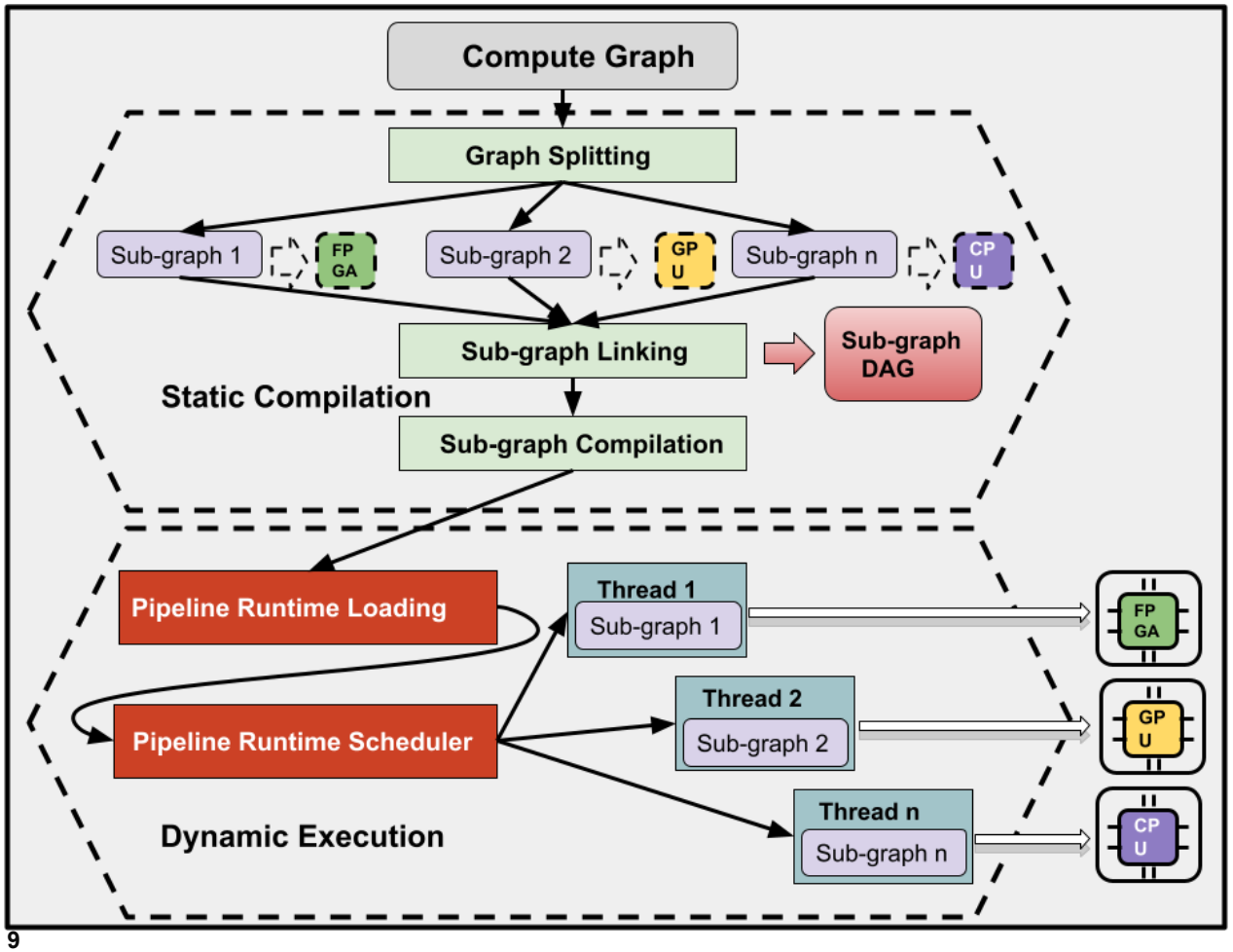
Optimize NN compiler to compile for M heterogenous backend cores.

Provision auto splitting and hand tuning process

Efficient Scheduling on heterogenous cores
- Separate control flow from data flow and set different hardware affinity
- Use compute intensive operator as boundary to split graph

Solution Overview

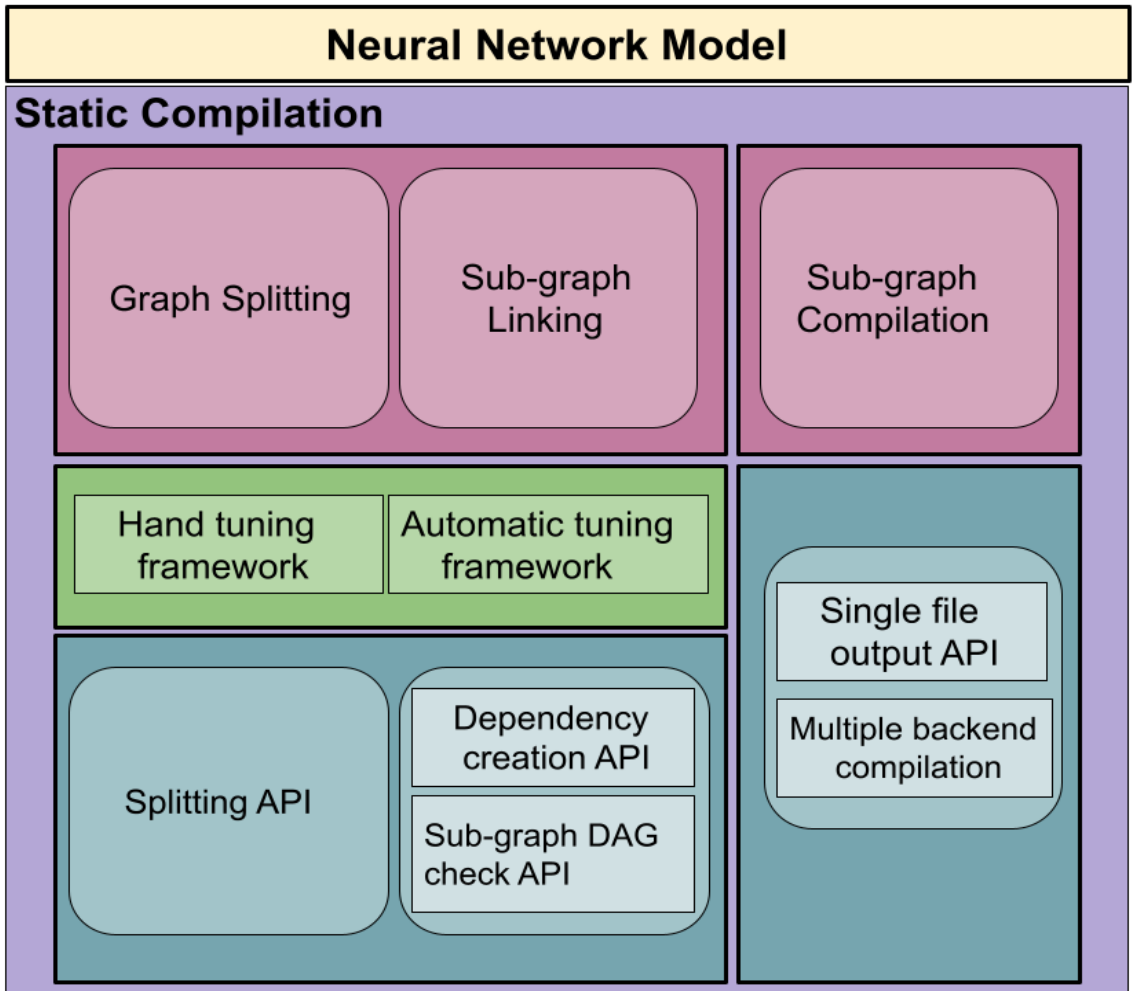
Solution Architecture



1. Static Compilation
   - Graph splitting
   - Sub-graph linking
   - Sub-graph compilation
2. Dynamic Execution
   - Runtime loading
   - Runtime scheduling

Static Compilation

Static Compilation - Overview

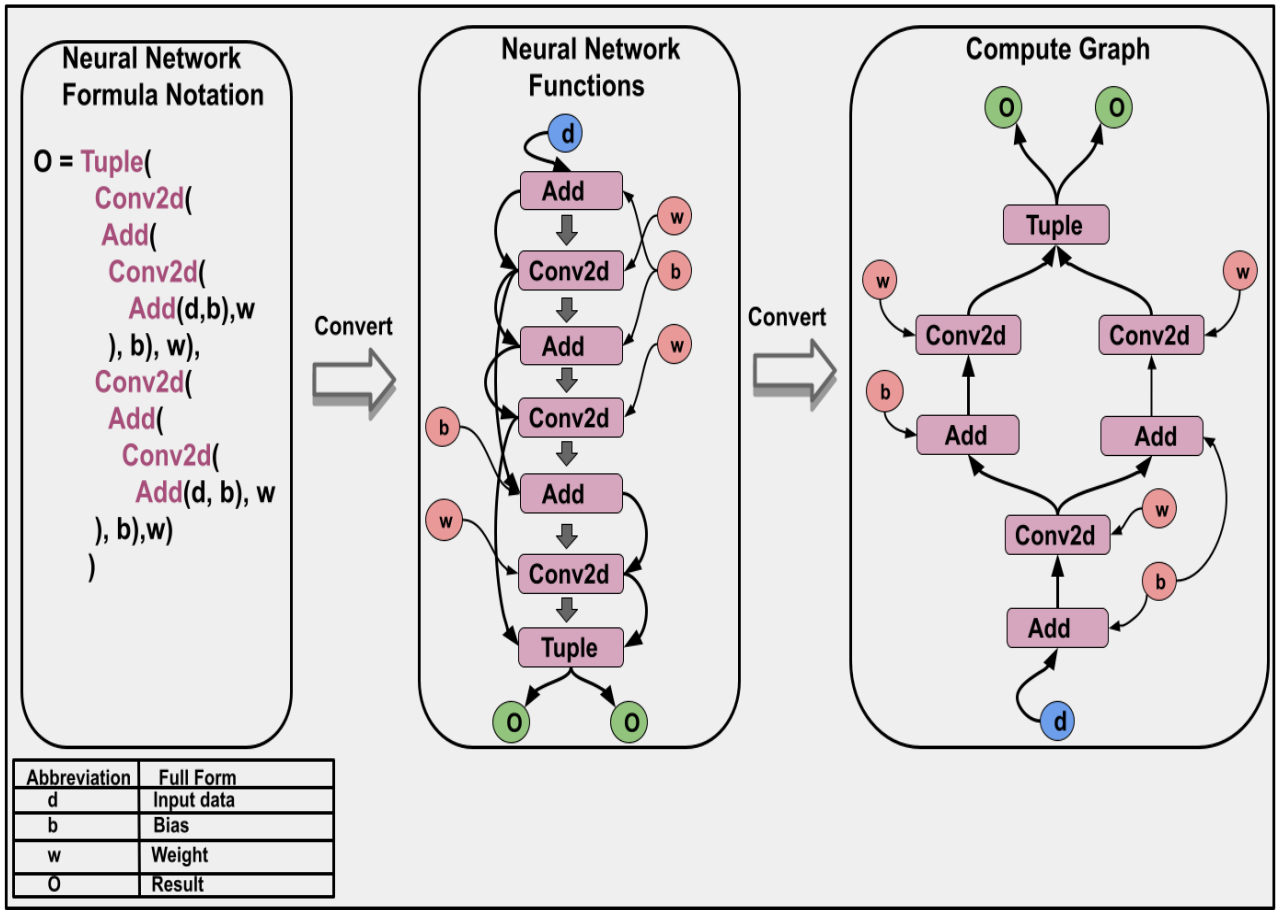


The first step of static compilation is to split compute graph into multiple sub-graph, user can either use expert mode hand tuning or automatic mode automatic tuning to do the graph split work.

The second step is to linking these sub-graph into a sub-graph DAG to pipeline or parallelism run these sub-graph, there are also two mode one is hand mode another is automatic mode.

The third step is compilation, it would create a single output file include multiple executable sub-graph and the dependency relation between the sub-graph.

Graph Splitting – Mapping neural network to Compute Graph



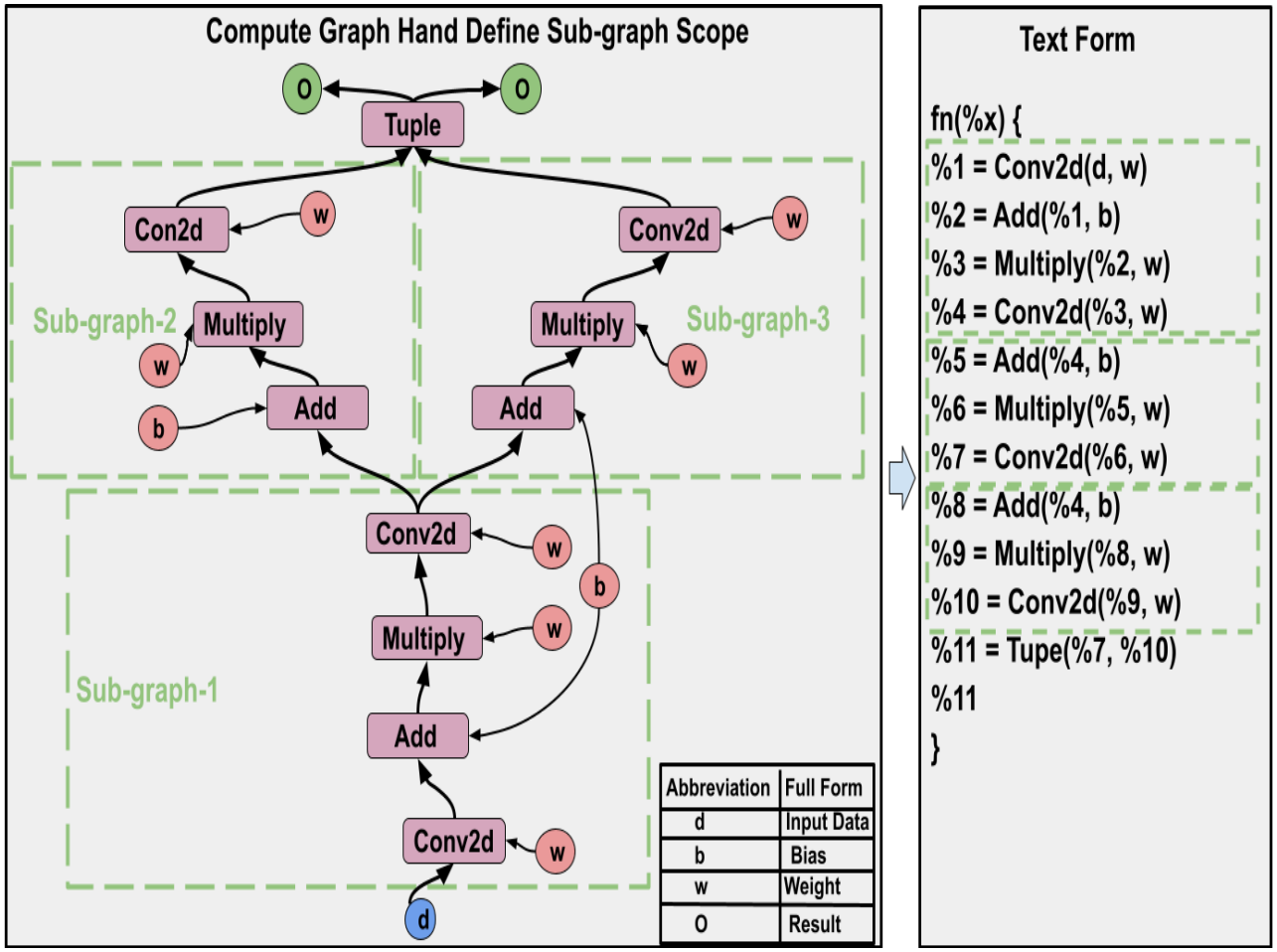
Abbreviation	Full Form
d	Input data
b	Bias
w	Weight
O	Result

Neural network start from a formula notation

Convert to a list function can split NN formula into a group of steps but lack of dependency relation of each function.

Compute graph describe the data flow reduce the model complexity and support automatic differentiation to implement generic backward propagation.

Graph Splitting- Graph Hand Splitting

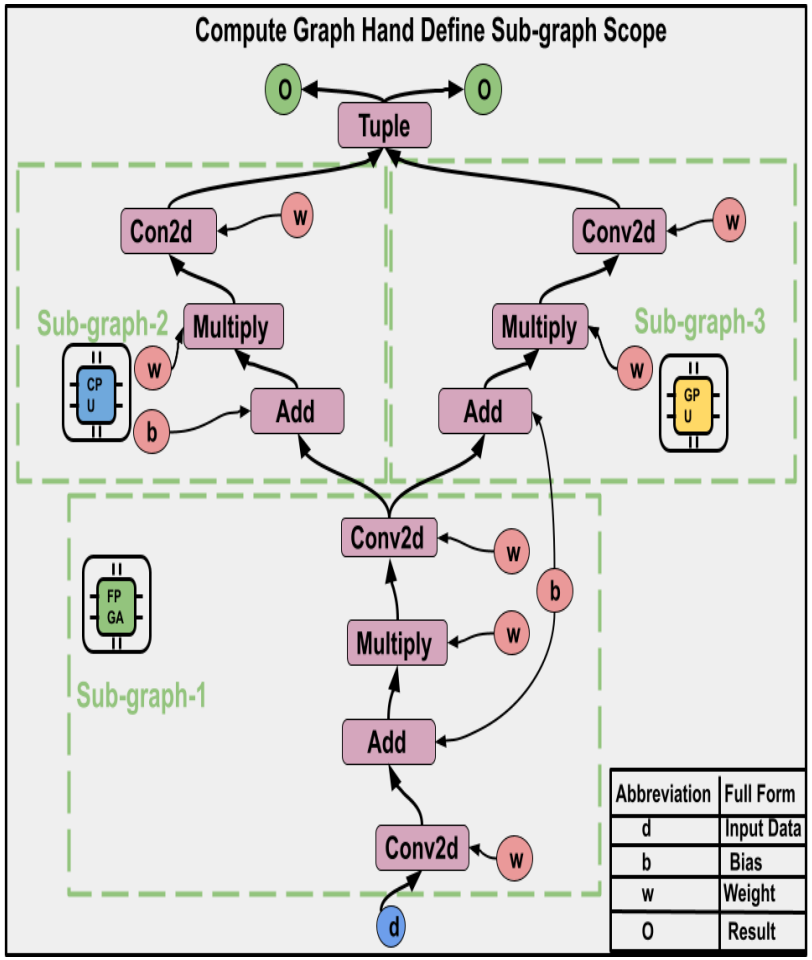


This solution provide graph split function to help user to split the graph

A Graph form can get convert into a text form, user can use a text form to define the sub-graph scope

By define start operator and end operator user can define the sub-graph scope

Graph Splitting-Graph Hand Splitting, Backend associate

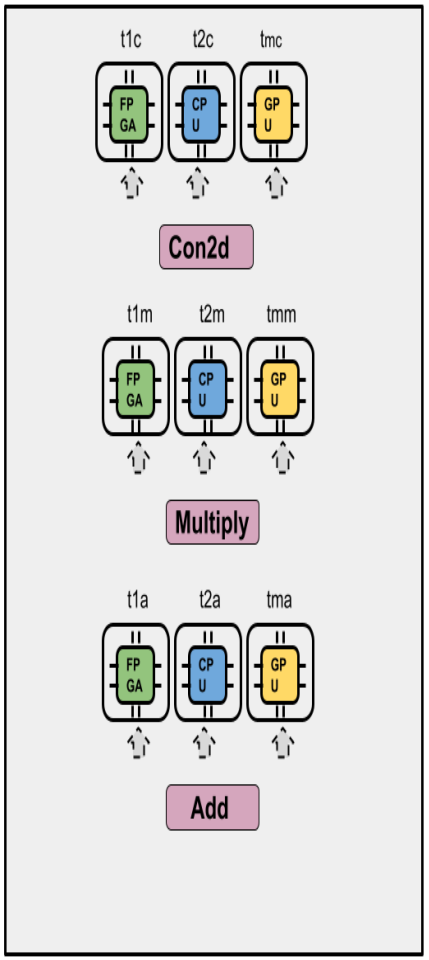


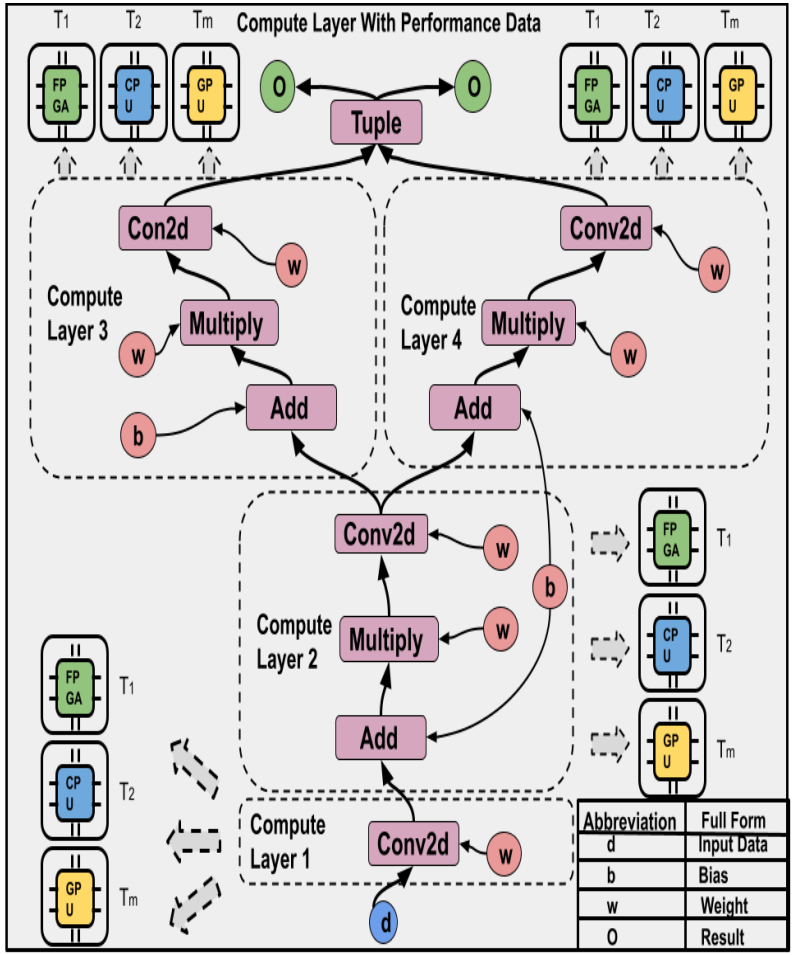

```
pipe_cfg[sub-graph1] = {
"device": "fpga()"}
pipe_cfg[sub-graph2] = {
"device": "cpu()"}
pipe_cfg[sub-graph3] = {
"device": "gpu()"}
```

User can manually create a configuration file to associate sub-graph with a backend

The backend binding will get use in compilation step

Graph Splitting-Graph Auto Splitting, Tuning



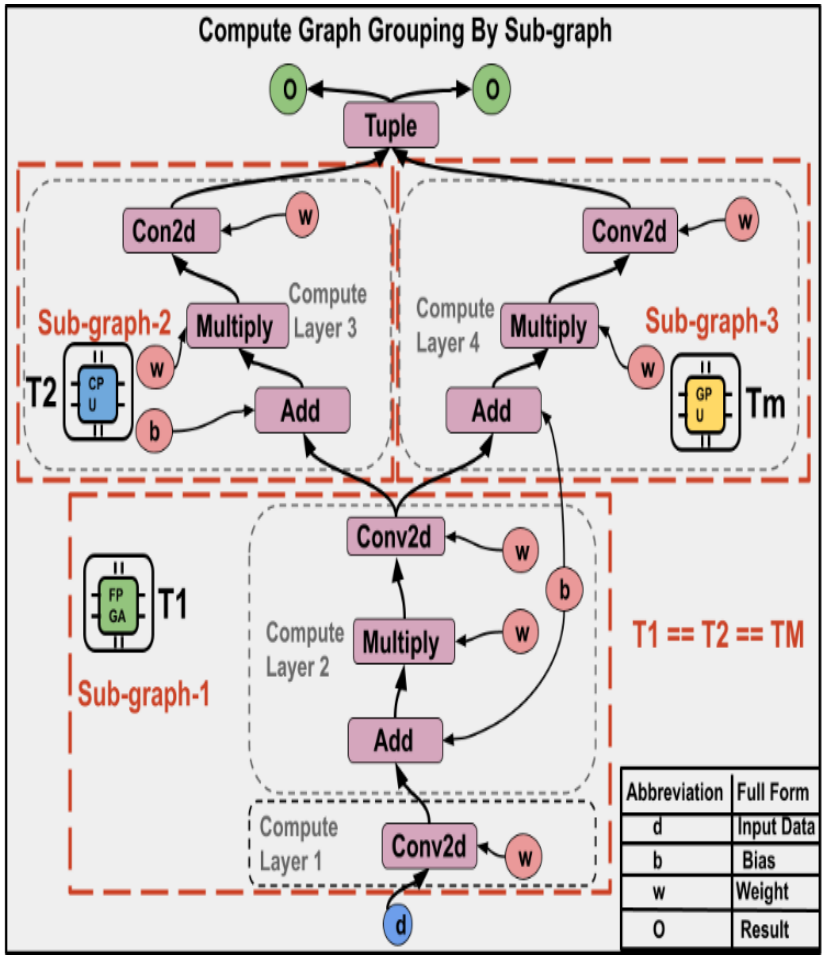


After do operator tuning, we can get each operator performance data on each backend.

Use compute intensive operator as a boundary to group operator to create a compute layer

The compute layer performance is the summarize of each operator in this compute layer.

Graph Splitting-Graph Auto Splitting, Sub-graph balance



List Of Sub-graph

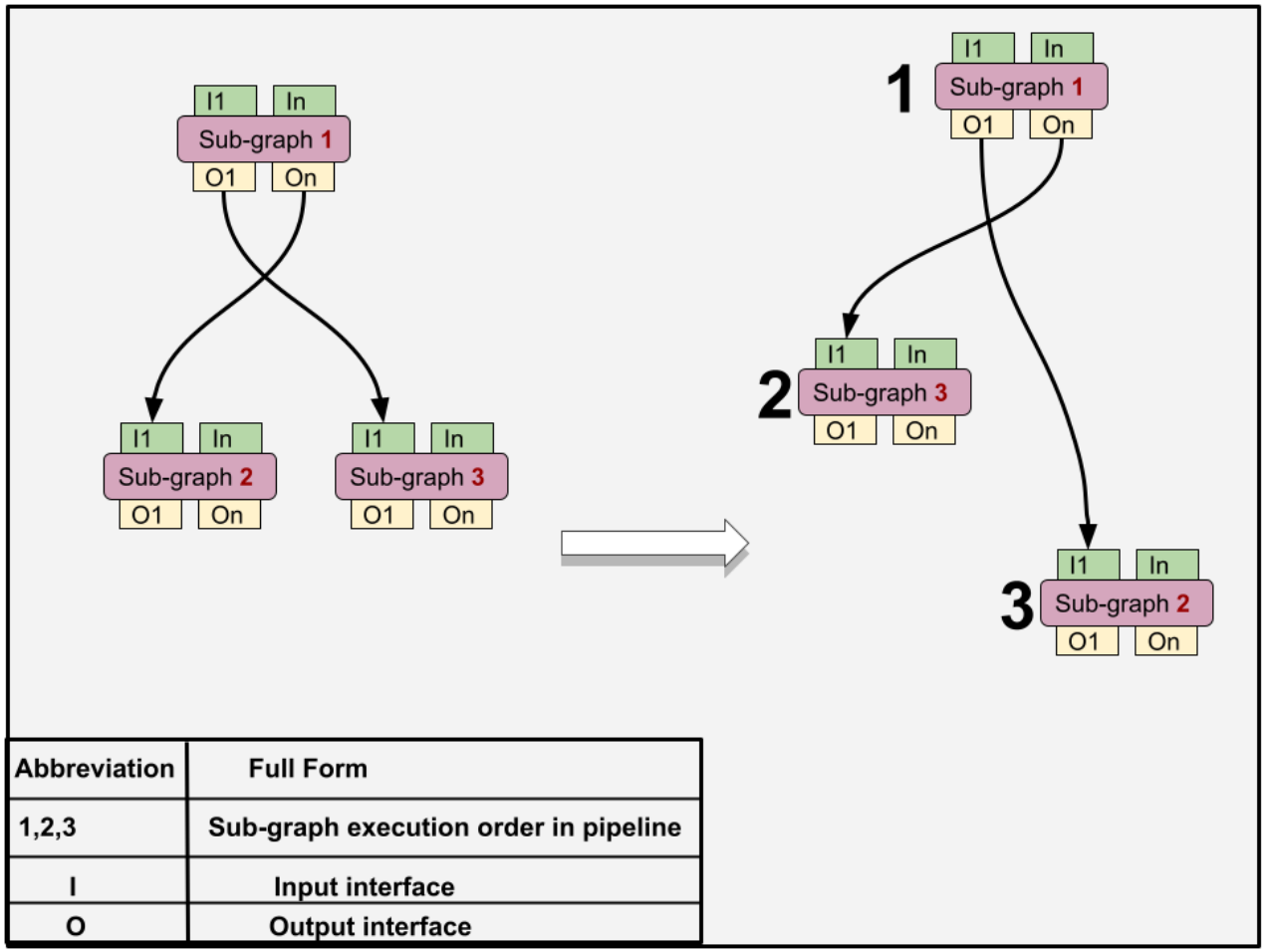
Sub-graph backend association

Sub-graph DAG

Auto splitting is an automatic process, this process will base on the tuning performance data to group the compute layer and create sub-graph .

After creating sub-graph, auto splitting will balance every sub-graph to make sure after backend binding each subgraph spend similar time

Sub-graph Linking - Overview



Abbreviation	Full Form
1,2,3	Sub-graph execution order in pipeline
I	Input interface
O	Output interface

After do graph splitting, the sub-graph parallelism become possible , but the original compute logic get broken, we use a sub-graph DAG to re-create the original compute logic

Use Sub-graph DAG to describe dependency between sub-graph and define sub-graph compute scope and find the execution order.

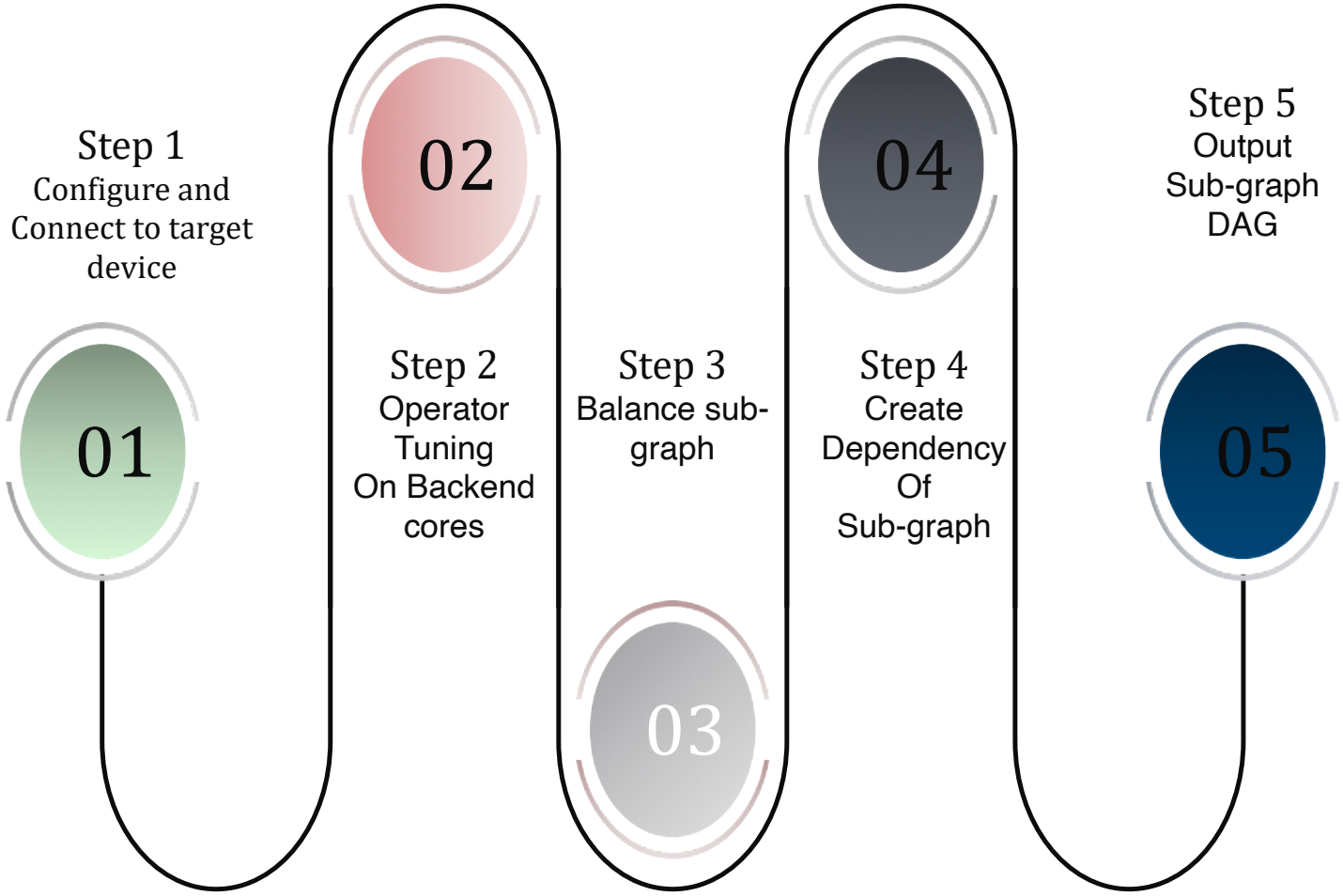
Sub-graph Linking -Create Sub-graph DAG, Hand Linking

```
mod1, mod2, mod3 = my_manual_partitioner()
pipe_cfg = PipelineModuleConfig()

# Define pipeline inputs. Here I assume two inputs of mod1 and one input of mod3 are the pipeline inputs.
pipe_config.connect(pipe_config.pipe_input("data_0"), pipe_config[mod1].input("data_0"))
pipe_config.connect(pipe_config.pipe_input("data_1"), pipe_config[mod1].input("data_1"))
pipe_config.connect(pipe_config.pipe_input("data_2"), pipe_config[mod3].input("data_1"))

# Define pipeline outputs to be the first output of mod3.
pipe_config.connect(pipe_config[mod3].output(0), pipe_config.pipe_output("0"))

# Define connections.
pipe_config.connect(pipe_config[mod1].output(0), pipe_config[mod2].input("data_0")) # mod1.output(0) -> mod2.data_0
pipe_config.connect(pipe_config[mod2].output(0), pipe_config[mod3].input("data_1")) # mod2.output(0) -> mod3.data_1

# Print config for debugging
print(str(pipe_cfg))
# Inputs:
#   |- data_0: mod1.data_0
#   |- data_1: mod1.data_1
#   |- data_2: mod3.data_0
# Outputs:
#   |- output(0): mod3.output(0)
# Connections:
#   |- mod1.output(0) -> mod2.data_0
#   |- mod2.output(0) -> mod3.data_1
```

User can manually link sub-graph to create a sub-graph DAG, this solution provide a list of user-friendly API to help for such work.

The link need to link global input with sub-graph input and global output with sub-graph output, it also need to link sub-graph to build dependency relation for data flow

Sub-graph Linking-Create Sub-graph DAG, Auto Linking



This solution also provide an automatic solution automatically creating sub-graph DAG.

Automatically Link Sub-graph steps as the diagram shown

Sub-graph Compilation-Configuration and Compile

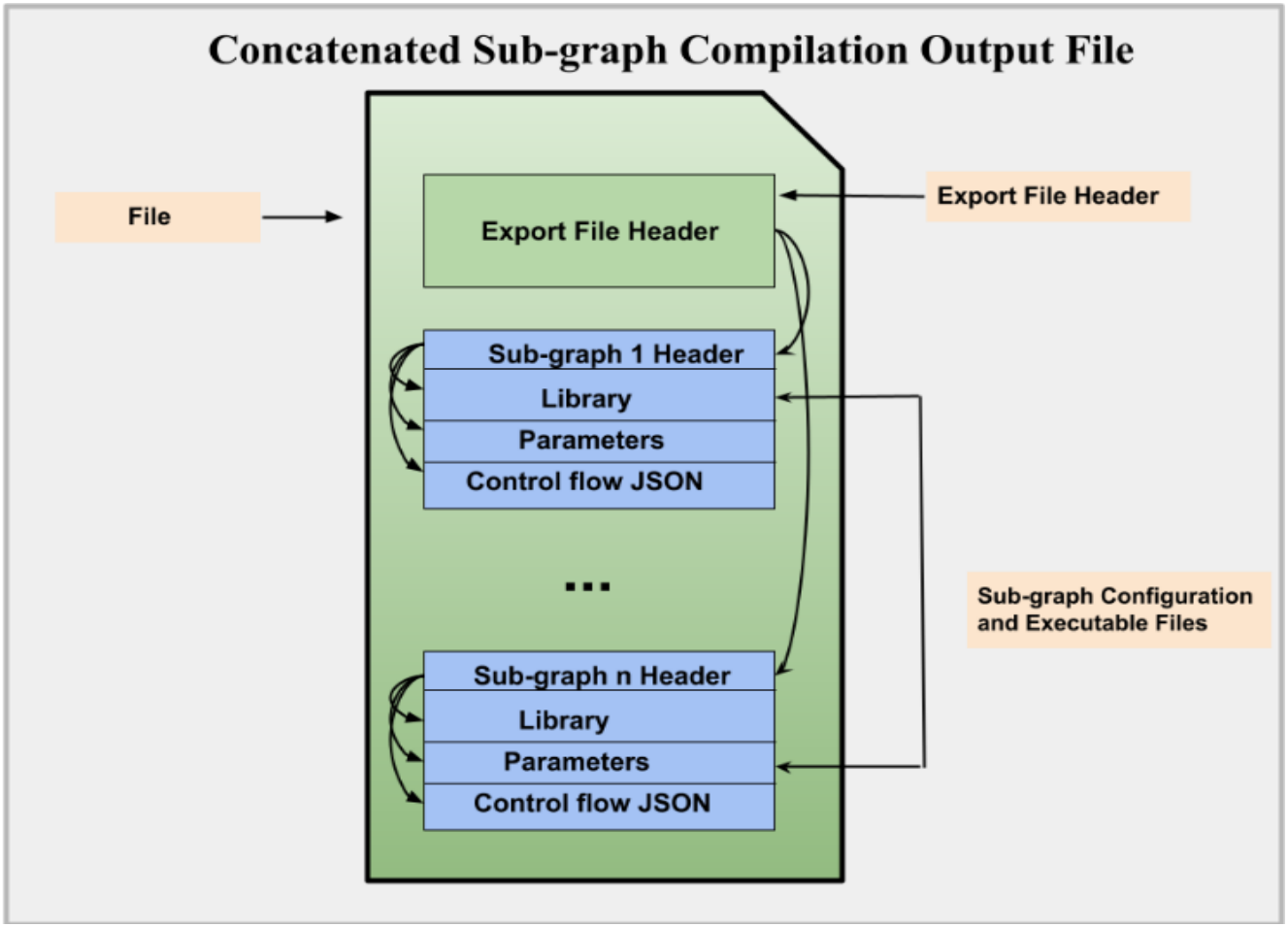
```
pipe_cfg[sub-graph0] = { "target": "llvm",
                         "customized_build_func": None,
                         "device": "cpu"
                        }
pipe_cfg[sub-graph1] = { "target": "opencl",
                         "customized_build_func":None,
                         "device": "gpu"
                        }
pipe_cfg[sub-graph2] = { "target": armcc,
                         "customized_build_func": vta_build,
                         "device": "fpga"
                        }

# Use the config to build a pipeline executor
with relay.build_config(opt_level=3):
    lib = pipeline_executor.build_pipeline(pipe_cfg)
```

Compile Multiple backend in one place.

Manually define the mapping of sub-graph and backend.

This solution also provide an automatically solution to automatically generate the configure.

Sub-graph Compilation-Output

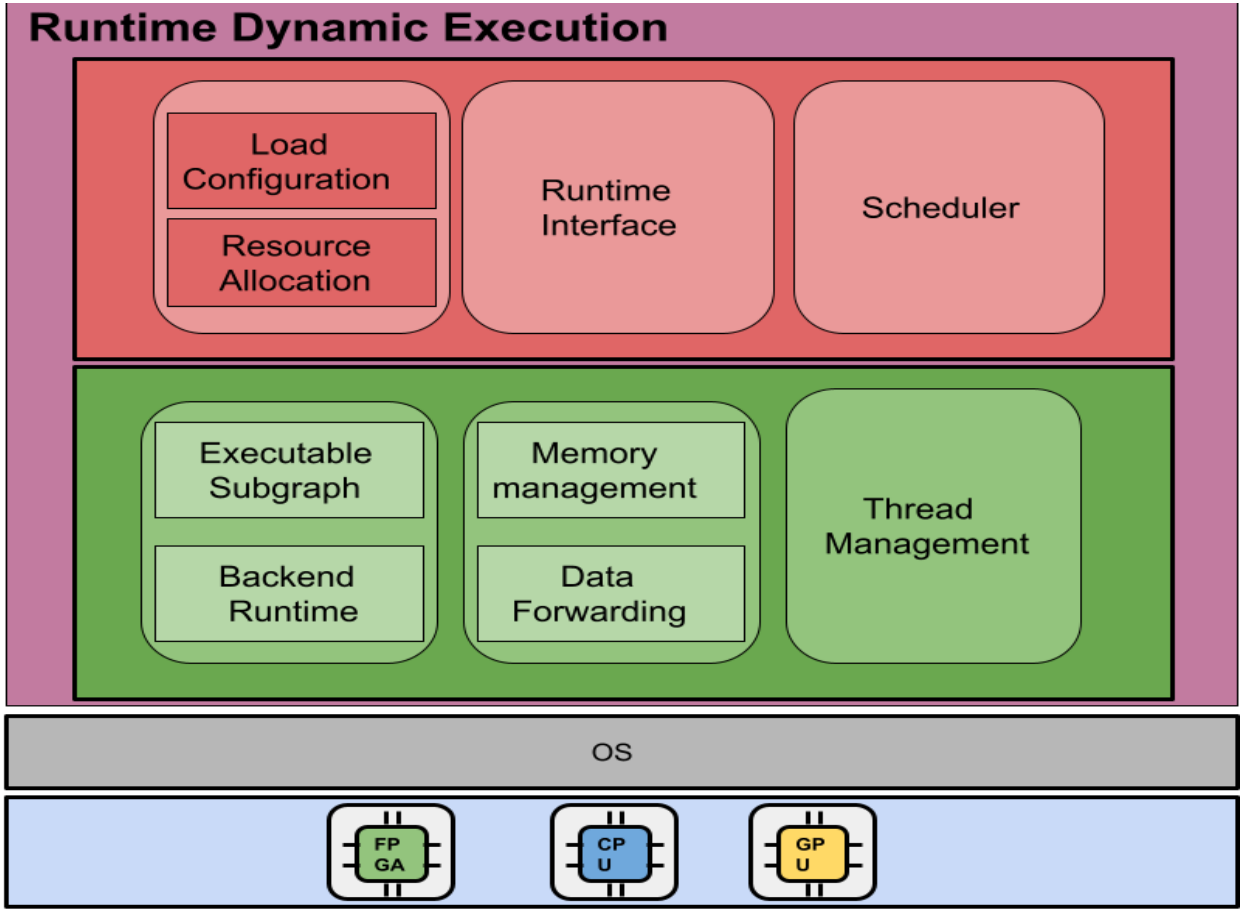


Output compile result in single file

This file is the input of JIT Running, it include executable sub-graph binary and model per sub-graph parameters and control flow infroamtion.

Dynamic Execution

Dynamic Execution- Overview



Schedule multiple backends and forward data between these backends

The first step is to load the output file of "Static compilation" module, then user calling this runtime interface to set and get data ,internally the scheduler will schedule the sub-graph on different backend.

Loading configuration and initialization

Load executable sub-graph

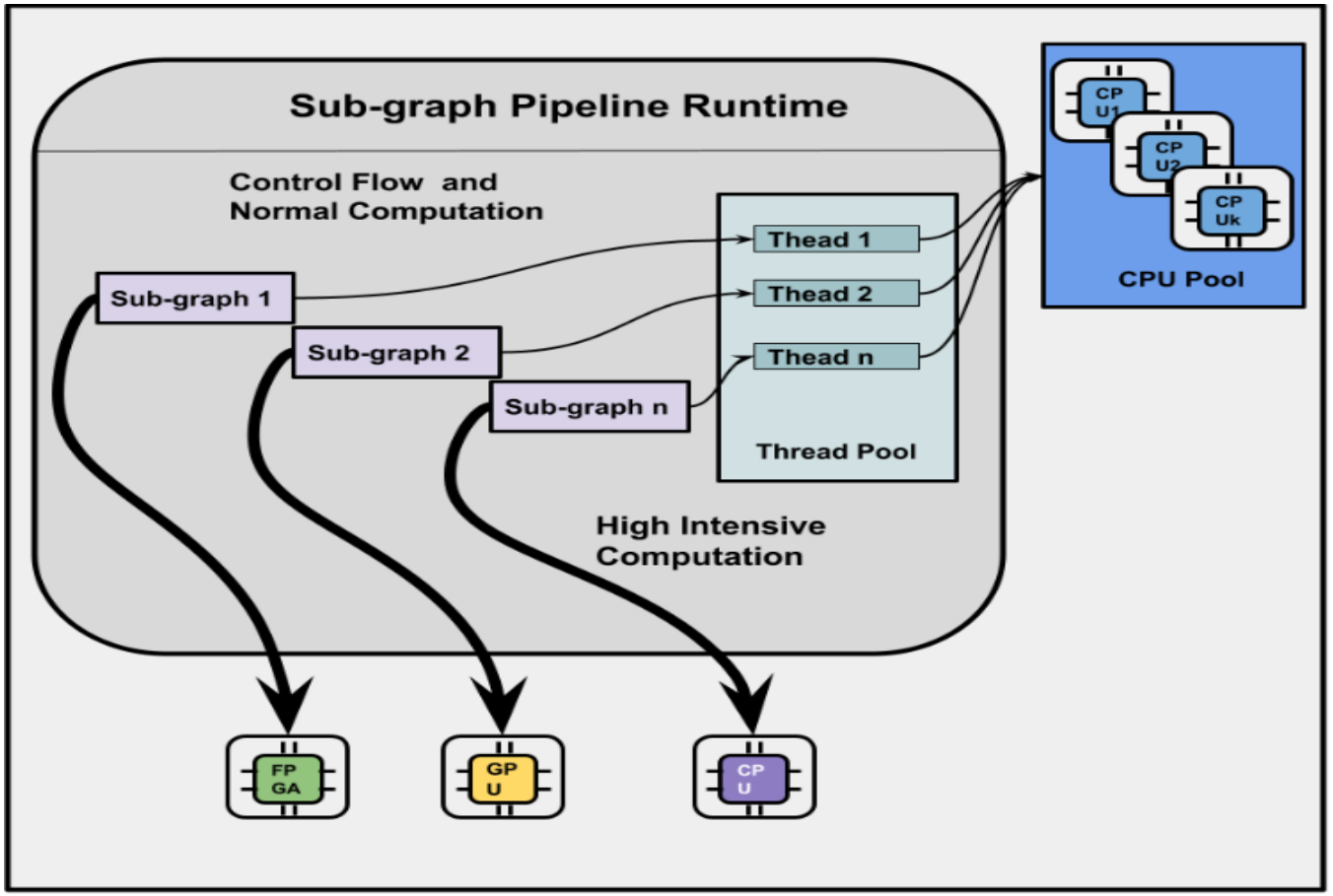
Load neural network per sub-graph parameters

Load sub-graph DAG

Initialize forwarding queue and threads pool

Loading and initialize

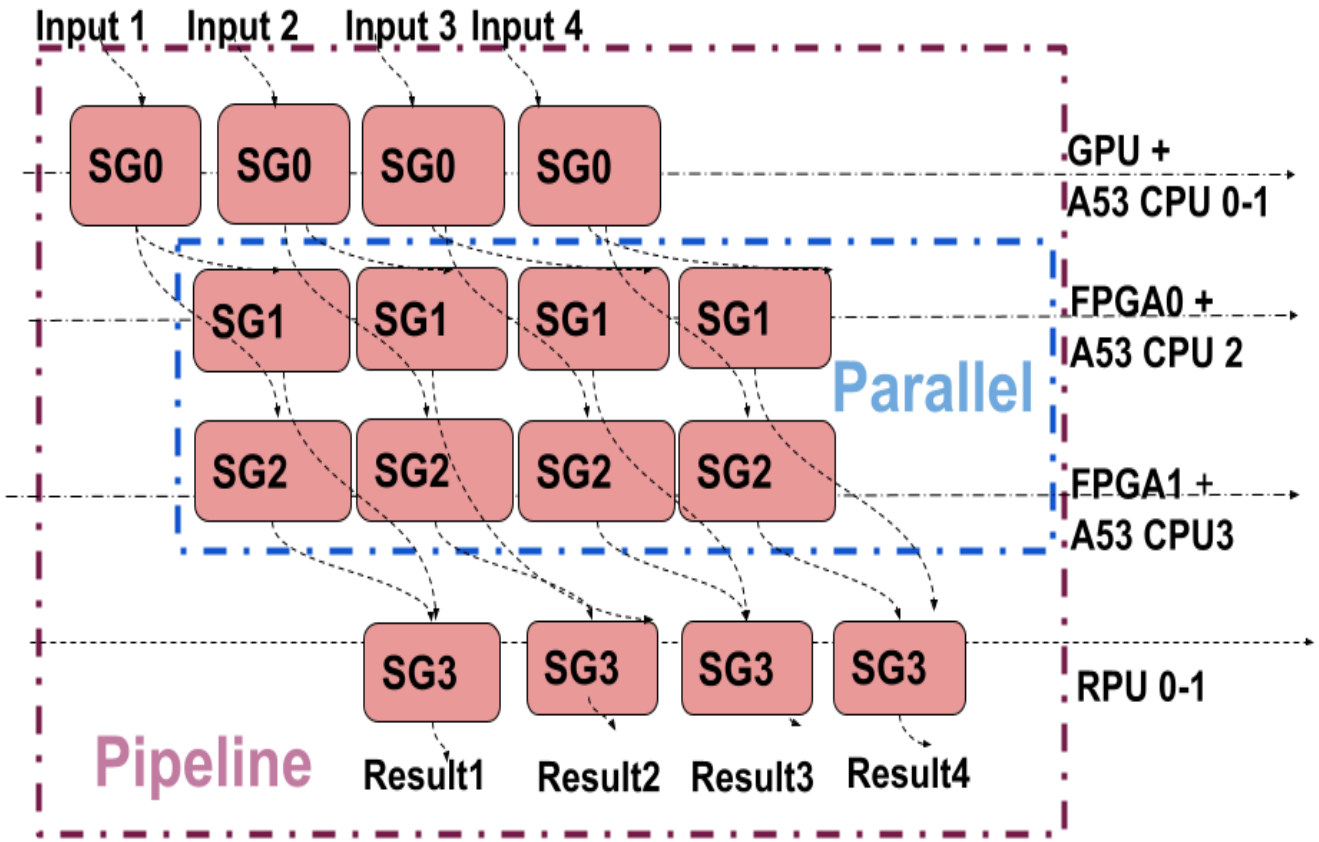
Threads Management



Running control flow and normal computation on CPU and a run the highly intensive computation on accelerator.

Each sub-graph would have it own working threads pool binding with specify CPU resource.

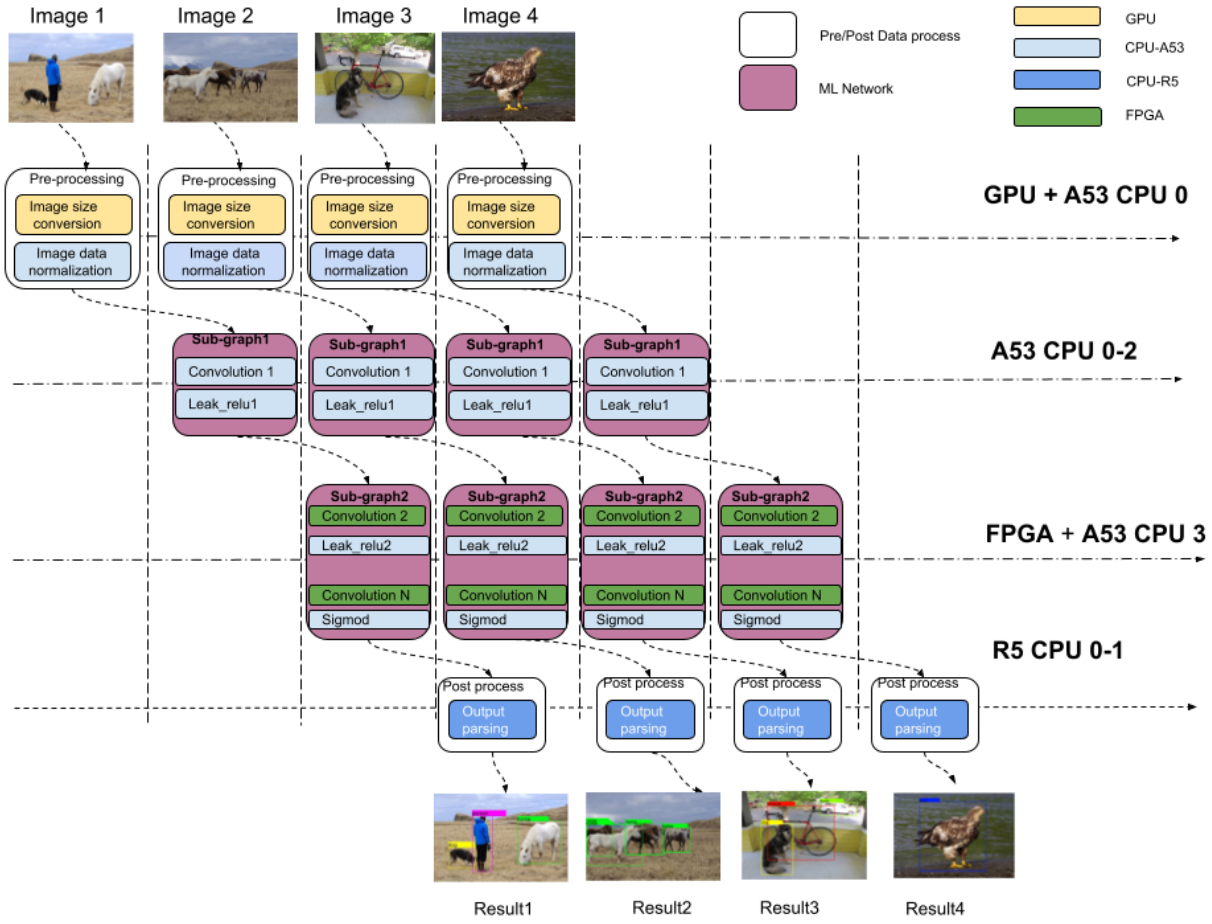
Scheduler



The non-dependent sub-graphs can get schedule in parallel mode and these sub-graphs have dependency will execute in pipeline

Scheduler using the dependency queue to control the sub-graph execution is pipeline or parallelism.

Example Object detection



Pipelining aids parallel processing of multiple image at the same time and improves performance

Performance

Performance

Performance data on Xilinx Ultra96 board comparing our proposed solution and TVM-VTA.

Network Type	Architecture Type	
	TVM-VTA	Proposed Solution
Resnet18	12FPS	16FPS
Yolov3-tiny	6FPS	8FPS

Pipelining and parallelism shows ~1.3x performance improvement compared to sequential processing in heterogenous system

Open-source

Open-source

Part of This Solution already Merged into Upstream TVM Codebase

https://github.com/apache/tvm/pull/8702

https://github.com/apache/tvm/pull/9108

Conclusion

Conclusion

- Achieved significant performance improvements and flexibility compared to existing frameworks
- The framework is completely open-source and we hope community can contribute further to extend it capability
- The work for sub-graph pipelining and parallelism upstream to the TVM community, Several of our patches have been accepted and the remaining patches are planned to be completed soon.