MASE: Abstractions, Optimizations and Implementations

Aaron Zhao, Imperial College London

Introduction

Introduction - MASE

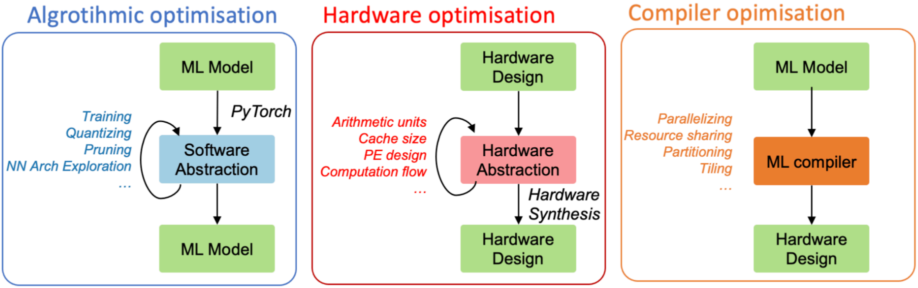
MASE (Machine Learning Accelerator System Exploration) is an open-source project that aims to automate the exploration of ML system software and hardware.

https://github.com/DeepWok/mase

Introduction - Why re-inventing the wheel?

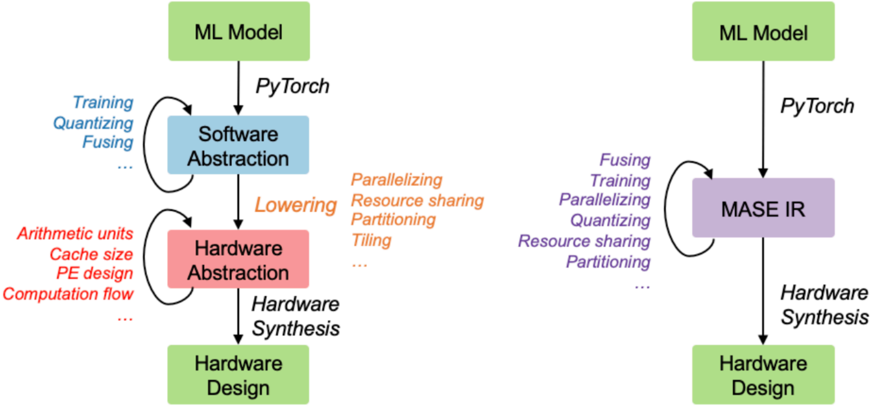
Introduction - Why re-inventing the wheel? (ii)

- Pytorch, Tensorflow (high-level python tools), algorithmic exploration, mapping mostly to CPUs and GPUs
- MLIR, TVM, compiler tools, map pre-defined network to various hardware targets
- MLIR-Circt, scheduling based HLS on top of MLIR



Introduction - Why re-inventing the wheel? (iii)

We are interested in combine these in a unified abstraction – a new graph-based MASE Intermediate Representation (IR)



Introduction - Passes

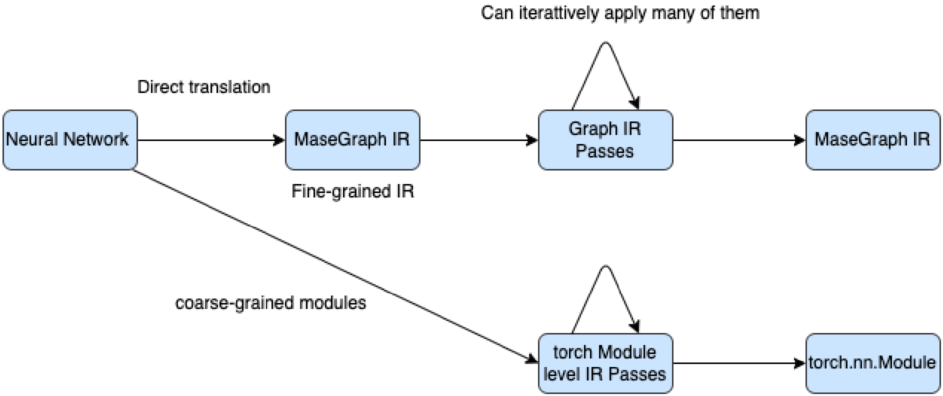
A pass (can be either a transformation or an analysis), takes in the IR of the model, and returns the IR again

```python
# pass takes a MaseGraph (the IR) and corresponding pass-related arguments
# graph: MaseGraph, pass_args: dict

# return a MaseGraph (the IR) again, and a dictionary for additional data
# graph: MaseGraph, info: dict

def pass_name(graph, pass_args):
    ...
  return graph, info
```

Introduction - Overview

- Graph-level IR system and passes (we will cover in labs)
- Module-level passes (code is there)
- The idea of summarizing workloads into a set of IRs, and apply passes on them is the same as traditional compiler systems (eg. LLVM).

Introduction - Overview (ii)



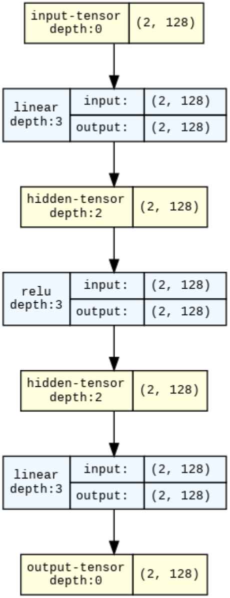
MaseGraph IR

MASEGraph IR

Core idea: we represent Neural Networks as a computation graph, where nodes are computation blocks and edges are data.

We represent Neural Networks as a computation DAG (Directed Acyclic Graph), where nodes are computation blocks and edges are data.

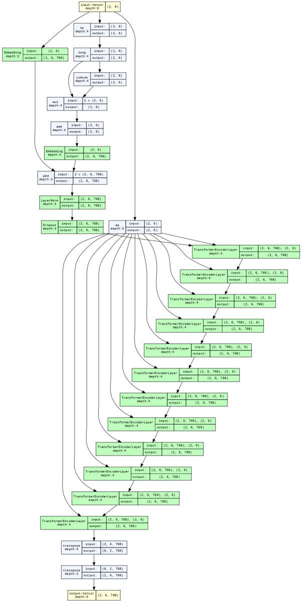
MASEGraph IR (ii)



MASEGraph IR (iii)

Visualization This can be very complex, notice the transformer layer is represented at a coarse granularity.

MASEGraph IR (iv)

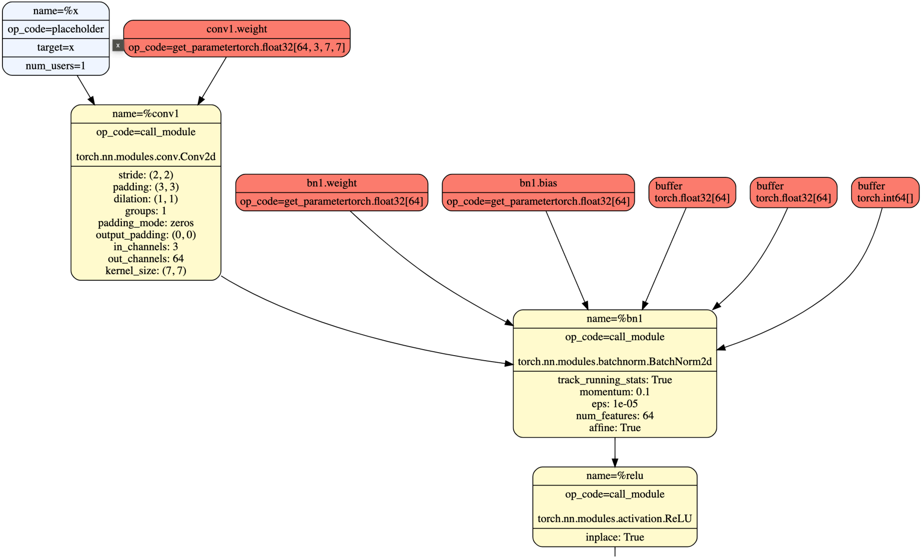


MASEGraph IR (v)

Mase takes a `torch.fx` graph representation of a model and translates it into a customised representation (Mase graph IR).

The MaseGraph IR is a lot more complex than the previous visualization. Below is a single convolution layer. To reproduce, run `masegraph.plot`.

MASEGraph IR (vi)



MASEGraph IR - Implementation

The manipulation of the model requires both access to the torch fx graph and modules, you will understand this better after having the labs.

The definition can be found at `mase/chop/ir/graph/mase_graph.py`

```python
class MaseGraph:
  def __init__(self, model, cf_args) -> None: ...

  def fx_graph(self): ... return self.model.graph

  def modules(self): ... return dict(self.model.named_modules())
```

MASEGraph IR

MASEGraph IR - types IR types are for nodes in the MaseGraph

- placeholder: for inputs
- module: for pytorch nn.Module
- module_related_func: some functions have the same functionality as a module, for instance, torch.nn.Conv2d (Module) and torch.nn.functional.conv2d are the same.
- builtin_func: what fx considers as builtin_funcs
- implicit_func: all other funcs that are not builtin
- get_attr: normally used for retrieving a parameter
- output: for outputs

A complete definitions of these and also supported nodes are in `mase/chop/passes/graph/common.py`

Metadata and Analysis Passes

MASEMetadata - why?

IR only carries type information and node relations.

We normally need more information to perform complex operations, such information is called metadata and they are added to each node.

```
class MaseMetadata: ...
```

- How do we add such a class to the MaseGraph IR?

Instantiate Metadata Pass Implemented as a pass!

Traverse each node and append the MaseMetadata object to each node. `chop/passes/graph/analysis/init_metadata.py`

```
def init_metadata_analysis_pass(graph, pass_args):
  for node in graph.fx_graph.nodes:
    node.meta["mase"] = MaseMetadata(
      node=node, model=graph.model)
    ...
  return graph,
```

MASEMetadata - why? (ii)

Analysis Passes Optimizations or information gathering are implemented as Passes that traverse the whole or some portion of a network to either collect information or transform the network.

Generally, analysis passes are used for collect extra information of the network for later transformation passes.

- add_common_metadata
- add_software_metadata
- add_hardware_metadata

Passes are summarised at `chop/passes/graph/__init__.py`

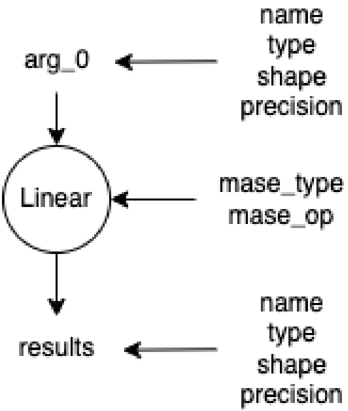
`add_common_metadata` passes add a bunch of commonly used metadata to each node. This includes

- `mase_type`: (`module_related_func`, `implicit_func` ...)
- `mase_op`: (`linear`, `relu` ...)
- `args`: (name, type, shape and precision for all input arguments)

MASEMetadata - why? (iii)

- `results`: (name, type, shape and precision for all all results)

`add_common_metadata` passes



MASEMetadata - why? (iv)

`add_software_metadata` and `add_hardware_metadata` passes do the same thing but add metadata for software and hardware respectively.

```
{
  "common": "mase_type": "module_related_func",
  "mase_op": "linear",
  "args":
  "data_in_0": {
    "shape": [1, 784],
    "type": "float",
    "precision": [32] ...},
  "weight": {
    "type": "float",
    "precision": [32],
    "shape": [784, 784] ... },
  "bias": {
    "type": "float",
    "precision": [32],
```

MASEMetadata - why? (v)

```
    "shape": [784] ... },
  "data_out_0": {
    "type": "float",
    "precision": [32]...
  }
  "torch_dtype": torch.float32,
  "software": ...,
  "hardware": ...,
}
```

Transform Passes

Transform Passes

Transform passes take a MaseGraph (or a network) as an input and perform certain modifications to it as an input and perform certain modifications to it.

I will use the quantization transform pass as an example.

`chop/passes/graph/transforms/quantize/quantize.py`

Quantize Transform Passes

```python
def quantize_transform_pass(graph, pass_args=None):
  by = pass_args.pop("by")
  match by:
    case "type":
      graph = graph_iterator_quantize_by_type(
        graph, pass_args)
    case "name":
      graph = graph_iterator_quantize_by_name(
        graph, pass_args)
    case "regex_name":
      graph = graph_iterator_quantize_by_regex_name(
        graph, pass_args)
    ...
```

Quantize Transform Passes

- Transformation is also implemented as a traverse to the MaseGraph.
- You can use pass arguments to control your logic.

```
def graph_iterator_quantize_by_name(graph, config):
    ...
    for node in graph.fx_graph.nodes:
        ...
        # create the new quantized module
        ori_module = get_node_actual_target(node)
        # take the parent node based on the graph hierarchy
parent_name,
        new_module = create_new_module(...)
        name = get_parent_name(node.target)
        # update meta data accordingly
        setattr(graph.modules[parent_name], name, new_module)
        update_quant_meta_param(node, node_config,
get_mase_op(node))
```

What's next?

- You will go through the quantization pass and learn MASE in labs.
- Lecture 3 - 4 will cover more on MASE and the labs.