MNSIM Latency Simulation Manual on Network-on-Chip

MNSIM Modification

To support the latency simulation based on Network-on-Chip (NoC), we update the interface between the MNSIM and the neural networks. Now, you can use function "_init_evaluation_interface" to initialize the neural network instance in the MNSIM.

```
__TestInterface = _init_evaluation_interface(args.NN, "imagenet",
args.hardware_description, None, args.device)
```

As for the input parameters of this function:

- the first param.: *args.NN*, the network type of the neural network, you can choose from {lenet, resnet18, alexnet, vgg8, vgg16} and so on.
- the second param.: *"imagenet:*, the dataset type, you can choose from {cifar10, cifar100, and imagenet}.
- the third param.: *args.hardware_description*, the hardware description file path.
- the fourth param.: *weight_path*, weight path of the per-trained model.
- the last params.: *args.device*, the device value, -1 is for CPU and others are for GPU device indexes.

The functions of the new __TestInterface are all the same as the original TrainTestInterface. You can utilize __TestInterface to evaluate the accuracy and get the structure based on the member function "_get_structure".

Data preparation for NoC simulation

We support the NoC simulation based on MNSIM_NoC, a upcoming open-source github repository. And we utilize the MNSIM to generate the data for the NoC simulation, we call it noc_data. In the new EvaluationInterface (the class of __TestInterface), we add a member function called "noc_data" to get the noc data. The function goes as follows:

```python
def noc_data(self):
    """
    get the noc data
    """
    tile_max_time = 0.
    # get key structure and key layers
    key_layer_info_list = list(…
    key_layers = list(…
    assert len(key_layer_info_list) == len(key_layers), \…
    in_buf_size = 4
    out_buf_size = 4
    intra_tile_bandwidth = 20
    # generate tile behavior
    tile_behavior_list = list()
    for layer_id, (key_layer_info, key_layer) in enumerate(zip(key_layer_info_list, key_layers)):
        if key_layer_info["type"] == "conv":…
        elif key_layer_info["type"] == "fc":…
        elif key_layer_info["type"] == "pooling":…
        elif key_layer_info["type"] == "element_sum":…
        else:…
    # max tile time
    print(f"max tile time: {tile_max_time} ns")
    print(f"need {len(tile_behavior_list)} tiles")
    # link tile, true tile id
    def _get_true_tile_id(layer_index):
        for tile_behavior in tile_behavior_list:
            if tile_behavior["layer_id"] == layer_index and tile_behavior["merge_flag"]:…
        return None
    for layer_id, (key_layer_info, key_layer) in enumerate(zip(key_layer_info_list, key_layers)):
        # for element sum
        input_index_list = [layer_id + i for i in key_layer_info["Inputindex"]]
        input_tile_list = [_get_true_tile_id(i) for i in input_index_list]
        target_tile_list = []
        if key_layer_info["type"] == "element_sum":…
        else:…
        # change for output
        for tile_behavior in tile_behavior_list:
            if tile_behavior["layer_id"] in input_index_list and tile_behavior["merge_flag"]:
                tile_behavior["target_tile_id"] = tile_behavior["target_tile_id"] + target_tile_list
    # last layer
    for tile_behavior in tile_behavior_list:
        if tile_behavior["layer_id"] == len(key_layer_info_list) - 1 and tile_behavior["merge_flag"]:
            tile_behavior["target_tile_id"] = [-1]
    # modify for th first fc
    for layer_id, (key_layer_info, key_layer) in enumerate(zip(key_layer_info_list, key_layers)):
        if key_layer_info["type"] == "fc":…
    # return
    return tile_behavior_list
```

It is hard to explain the function in words, and more details can be found in "MNSIM/Interface/evaluation.py". The function will generate the tile_behavior_list, which describes the behavior of each tile in the NoC.

tile_behavior_list is a list. Each element in the list corresponds to one and only corresponding tile. Element is a dictionary. The dictionary has the following key and corresponding values:

- task_ID: used to mark corresponding tasks and distinguish different tasks
- layer_ID: used to mark which layer of the corresponding task is the data processed by this tile. The number starts from 0 and increases in turn
- tile_ID: used to mark which tile this is, and this is in the task_ID number inside the task_ID
- target_tile_ID: used to mark the tile to which the data of this tile should be *sent* after calculation
- Dependence: which is the key information of the tile, stores the *behavior* and *calculation delay* of the tile. Behavior means what the input data and output data of the tile are, and how long it takes to

calculate the output data. Dependency is a list, in which each dependency corresponds to an output of the tile. Each element is a dictionary, consisting of the following keys and values

- Wait. The wait data is a list, and each element in the list is a tuple. Each tuple contains *nine* data, which are composed of layers in the following format (x, y, start, end, bit, length, image_id, layer_id, in_id). layer_id represents which layer generated this group of data. image_id is used to mark which image of this calculation. in_ids are used to represent different input sources for the merge node. Bit and length are used to represent the bit width and total number of channels of this data. x, y, start, and end represent the horizontal and vertical axes respectively, as well as the start and end points of the channel, which are left open and right closed. The reason why the design is so complex is to ensure that all data has its uniqueness and ensure the correctness of data transmission. Currently, 9 tuples have been changed to 10 tuples, and the last one is tile_id, which can accurately indicate the tile from which the tuple data is generated, and record its number.
- Output. Output is also a list, but the length is 1, representing the tuple of output data
- Drop. Drop is also a list, representing tuples that can be taken from the input buffer after the data is completed. To ensure that all buffers are empty after all runs are completed
- latency, representing the time of completing this operation

All data interactions are conducted in the form of tuples, which are unique within the task.

Examples

You can follow the following steps to run the example:

```
    evaluation_interface = _init_evaluation_interface(
        "vgg16_imagenet", "imagenet", "SimConfig.ini", None, -1,
        # "MNSIM/Interface/zoo/cifar10_resnet18_SGD_FIX_TRAIN_FIX_TRAIN.pth", 0
    )
    tile_behavior_list = evaluation_interface.noc_data()
    with open("tmp.pkl", "wb") as f:
        pickle.dump(tile_behavior_list, f)
```

More details can be found in "MNSIM/Interface/test/test.py". It should be noticed that, we recommend you to use pickle to dump the noc data to a binary file, and you should open the target file in "wb" mode. Alternatively, you can simply use the test.py file.

```
pytest -k test_evaluation_interface -s MNSIM/Interface/test/test.py
```

Before running the example, you should make sure that you have in the main directory of the MNSIM.

MNSIM_NoC

The explanation of the MNSIM_NoC is upcoming in github.