MapTools使用教程

👨‍🎓曹文旭 🏫UESTC 🗓️2023.03.19 📧1404335996@qq.com

1 获取Maptools

1.1 克隆工程

使用`git clone`获得maptools源码：

```
git clone https://gitee.com/cao-wenxu/nvcim-comm
```

进入工程根目录，创建一个名为`onnx_models`的文件夹，并将`simp-resnet18.onnx`放入`./onnx_models`目录中

1.2 配置开发环境

1.2.1 设置环境变量

将工程根目录加入到环境变量`NVCIM_HOME`，以Windows的Powershell为例：

```
$env:NVCIM_HOME="c:/your/root/directory"
```

为了检查环境变量是否设置正确，运行：

```
$env:NVCIM_HOME
```

看是否正确输出
设置完环境变量后，无需再reroot

1.2.2 安装Maptools Module

在工程根目录下，运行：

```
.\make.sh
```

观察当前目录下是否生成build文件夹，若生成，则说明maptools module安装成功。

2 映射脚本编写

2.1 MapRoutine登场

新版本的maptools新增了一个MapRoutine对象，通过MapRoutine可以快速地实现全流程映射，而不再需要冗余复杂的对象例化、函数调用和参数传递，下面展示了一个使用MapRoutine进行快速映射的例子：

```
from maptools import MapRoutine
routine = MapRoutine(mapname='newmap')
routine.run()
```

2.2 MapRoutine关键字参数

因为 `MapRoutine` 集成了映射的每一个中间环节，因此仅需调用其 `run()` 方法便能够执行全流程映射。 `MapRoutine` 的构造方法不需要位置参数，而是提供一些关键字参数用于控制映射过程，如下：

关键字参数	类型	含义
mapname	str	映射ID，默认为'newmap'。 类似于进程ID、socket等，根据不同的映射ID将映射结果分类保存到./mapsave文件夹中，比如设置mapname='xxx', 则此次映射结果保存在./mapsave/xxx文件夹下
model_dir	os.path 或 str	onnx文件的路径
arch	str	模型结构，默认为'resnet'
xbar_size	Tuple	xbar规模，默认为(256, 256*5)
noc_size	tuple	NoC规模，默认为(5, 10)
noc_map	bool	是否进行NoC映射，默认为True，若为False，则只进行xbar映射而不进行NoC映射
show_raw_graph	bool	是否显示原始模型图，默认为False
show_op_graph	bool	是否显示CIM算子图，默认为False
save_param	bool	是否保存模型参数，默认为True
toksim	bool	是否运行TokSim仿真，默认为False
show_execu	bool	是否显示TokSim的图示化结果，默认为False

关键字参数	类型	含义
show_ctg	bool	是否显示CTG图，默认为False
show_cast_path	bool	是否显示cast路径，默认为False
show_gather_path	bool	是否显示gather路径，默认为False
save_mapinfo	bool	是否保存映射结果，默认为True
calcusim	bool	是否运行CalcuSim仿真，默认为True

2.3 映射结果的保存位置

在上述`noc_map`， `save_param`和`save_mapinfo`三个关键字参数都设置为`True`（也是默认设置）的前提下，运行`MapRoutine`的`run()`方法，会自动保存模型参数和映射结果，其中：

1. 模型参数保存在`./mapsave/your-mapname/params.pkl`文件中
2. 映射结果保存在`./mapsave/your-mapname/mapinfo.pkl`文件中

3 获取配置信息

3.1 读取保存的映射结果

3.1.1 read_mapinfo()方法

Maptools提供`read_mapinfo()`方法来获得上次保存的映射信息:

```
from maptools import read_mapinfo
mapname = 'your_mapname'
mapinfo = read_mapinfo(mapname)
```

返回的`mapinfo`是一个字典，key是对应的信息类型（字符串），value是对应的信息（Any）

3.1.2 read_params()方法

Maptools提供`read_params()`方法来获得上次保存的模型参数:

```
from maptools import read_params
mapname = 'your_mapname'
params = read_params(mapname)
```

返回的`params`是一个字典，key是参数名（字符串），value是对应的参数张量（numpy.array）

3.2 获取xbar配置信息

xbar的配置信息存储在`mapinfo['xbar_config']`中，可以使用下面的方法来获得xbar配置信息：

```
for physical_xbar, cfg_info in mapinfo['xbar_config'].items():
    # physical_xbar : Tuple[2] is the physical position of the
xbar
    # cfg_info : Dict is the configuration information of the
corresponding xbar
```

上述`cfg_info`的组织结构如下：

key	对应value
op_type	表示当前xbar负责的计算的类型。是由'Conv'（卷积）,'Add'（Gather in）,'Act'（激活）,'Pool（池化）','Bias（偏置）'四个基本操作组合而成的，比如'Conv-Act-Pool'，'Conv-Pool'，'Conv-Add-Act-Bias'，当然也可以是单独的'Conv'。不管四个基本操作的组合顺序如何，其在硬件上的执行顺序永远是'Conv'→'Bias'→'Add'→'Act'->'Pool'。 op_type至少包含'Conv'，也就是说，每个xbar都要执行卷积运算。
xbar_icfg	和之前的icfg相同
xbar_ocfg	和之前的ocfg相同
conv_kernel_size	卷积核尺寸，[Height, Width]
conv_pads	卷积padding值，[Top, Right, Bottom, Left]
conv_input_size	卷积输入尺寸，[Height, Width], 不含padding
conv_output_size	卷积输出尺寸，[Height, Width]
conv_strides	卷积滑动步长，[Vertical, Horizontal]
conv_weights	卷积权重的指针，是个字符串，作为上述`params`字典的key使用
conv_bias	卷积偏置的指针，是个字符串，作为上述`params`字典的key使用

key	对应value
xbar_num_ichan	当前xbar负责的卷积输入通道个数
xbar_num_ochan	当前xbar负责的卷积输出通道个数
act_mode	激活运算的类型，只有当op_type包含'Act'时才存在这个key。可能值：['Relu','PRelu','HardSigmoid']，在resnet中暂且只考虑'Relu'
pool_input_size	Pooling输入尺寸，只有当op_type包含'Pool'时才存在这个key。[Height, Width], 不含padding
pool_output_size	Pooling输出尺寸，只有当op_type包含'Pool'时才存在这个key。[Height, Width]
pool_mode	Pooling运算的类型，只有当op_type包含'Pool'时才存在这个key。可能值：['MaxPool','AveragePool']
pool_kernel_size	Pooling窗口尺寸，只有当op_type包含'Pool'时才存在这个key。[Height, Width]
pool_pads	Pooling padding值，只有当op_type包含'Pool'时才存在这个key。[Top, Right, Bottom, Left]
pool_strides	Pooling滑动步长, 只有当op_type包含'Pool'时才存在这个key。[Vertical, Horizontal]

3.3 获取模型参数

对于某个xbar， 可以使用下面的方法获取其负责的卷积层的参数：

```
for physical_xbar, cfg_info in mapinfo['xbar_config'].items():
        wp = cfg_info['conv_weight']
        weight: numpy.array = params[wp]
        if 'conv_bias' in cfg_info:
                bp = cfg_info['conv_bias']
                bias: numpy.array = params[bp]
```

注意，这里获得的`weight`和`bias`是整层的参数，所以在将参数部署到xbar之前那，还需要根据`cfg_info['xbar_icfg']`和`cfg_info['xbar_ocfg']`对获得的参数进行切片

3.4 获取通信连接信息

cast，merge和gather三种通信的连接信息分别保存在`mapinfo['p2p_casts']`，`mapinfo['p2p_merges']`，`mapinfo['p2p_gathers']`三个List中。

3.4.1 获取cast通信连接信息

```
for item in mapinfo['p2p_casts']:
    src = item[0] # src is the root node of the cast tree
    dsts = item[1] # dsts is a list of the destination nodes of
the cast tree
```

3.4.2 获取merge通信连接信息

```
for item in mapinfo['p2p_merges']:
    srcs = item[0] # srcs is a list of the source nodes of the
merge tree (including the root node)
    dst = item[1] # dst is the root node if the merge tree
```

❗注意，merge的源节点列表包含根节点。

3.4.3 获取gather通信连接信息

```
for item in mapinfo['p2p_gathers']:
    src = item[0] # src is the source node of the gather
connection
    dst = item[1] # dst is the destination node of the gather
connection
```

3.5 获取逻辑-物理映射信息

- 逻辑xbar是一个四元元组 (layer, region, block, index_in_block)
- 物理xbar是一个二元元组 (x, y)

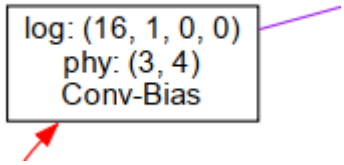
由于映射过程包含随机因素，所以逻辑-物理的映射信息在每次映射中都不同，为了获得该信息，可以通过mapinfo或CTG两个途径实现

3.5.1 通过mapinfo获取逻辑-物理映射信息

逻辑-物理的映射信息保存在`mapinfo['match_dict']`这个字典中，其key是逻辑xbar，value是物理xbar。

3.5.2 通过CTG获取逻辑-物理映射信息

CTG可以直观地展示逻辑-物理的映射信息，如图：



为了实现这一点，只需在创建 `MapRoutine` 对象时把 `show_ctg` 这个关键字参数设置为 `True`，如下：

```
from maptools import MapRoutine
routine = MapRoutine(mapname='newmap', show_ctg=True)
routine.run()
```

这样每次映射，都会在作出的CTG中显示每个xbar的逻辑ID和物理ID，该CTG以PDF的形式被自动保存到 `./mapsave/your-mapname/ctg/` 目录下

❗注意：一定要保证每次映射生成的mapinfo.pkl和CTG文件都保持一致，如果下次运行映射时你关闭了 `show_ctg` 这个关键字参数，还是会生成最新的mapinfo.pkl文件，但这个文件和上次生成的CTG文件的信息就不一致了。所以，要么就不要轻易运行映射程序，一旦运行，最好每次都打开 `show_ctg` 这个关键字参数。

4 计算结果对比

Maptools提供CalcuSim仿真器用于仿真调试时的结果比对，CalcuSim能够模拟每个xbar的计算任务，并将每个xbar中参与运算的中间结果都保存到.pkl文件中。

4.1 使用CalcuSim

在例化 `MapRoutine` 时，将 `calcusim` 这个关键字参数设置为 `True`，然后设置 `input` 这个关键字参数为输入数据，之后再运行 `MapRoutine` 的 `run()` 方法时，便会自动运行CalcuSim仿真并自动保存中间结果。
在获取输入数据时，可以直接使用MapTools提供的 `get_input` 方法，下面举了一个例子：

```
from maptools import *
mapname = 'your-mapname'
img = get_input('your/path/to/test.jpg', resize=(224, 224))
routine = MapRoutine(mapname=mapname,calcusim=True,show_ctg=True,
input=img)
routine.run()
```

运行上述程序后，中间结果被自动保存到 `./mapsave/your-mapname/calcusim/results.pkl` 文件中

4.2 读取中间结果

Maptools提供 `read_results()` 方法用于读取CalcuSim上次保存的中间结果，使用方法如下：

```
from maptools import read_results
mapname = 'your-mapname'
results = read_results(mapname)
```

上面的程序中，得到的 `results` 是一个字典，其key是逻辑xbar(四元元组)，value也是一个字典，保存着和该xbar计算有关的的中间结果，它有4个key，分别是 `'cast_in'`， `'merge_in'`， `'gather_in'`，和 `'data_out'`，分别对应cast输入tensor（每个xbar都有），merge输入tensor（只有merge xbar才可能有），gather输入tensor（只有merge xbar才可能有）和当前xbar的输出tensor（即cast_out，每个xbar都有）。对于没有的情况，读出来的值是 `None`。

比如，如果我要获得(3,1,0,0)这个xbar的中间结果：

```
cast_in = results[(3,1,0,0)]['cast_in']
merge_in = results[(3,1,0,0)]['merge_in']
gather_in = results[(3,1,0,0)]['gather_in']
data_out = results[(3,1,0,0)]['data_out']
```

另外 `results` 还有一个特殊的key，为 `'output'`，是xbar阵列的总输出，是经过拼接之后的。

比如，我要获得xbar阵列的总输出：

```
output = results['output']
```

❗注意： `results` 中所有的tensor都是4维（[N, C, H, W]）的 `torch.Tensor`