神经网络算子调度与图模式匹配

7.1 图模式匹配

Graph Pattern Matching

一个计算图可以表示为一个由节点、边集、输入边、输出边组成的四元组 C = {N, E, I, O}。

我们往往需要在计算图中寻找**指定结构**

- 如何用一个严谨的方式定义**结构**？
- 如何设计模式匹配算法，使得其尽可能高效？

7.1 图模式匹配

Graph Pattern Matching

一个计算图可以表示为一个由节点、边集、输入边、输出边组成的四元组 C = {N, E, I, O}。

我们往往需要在计算图中寻找指定结构

- 如何用一个严谨的方式定义结构？
- 如何设计模式匹配算法，使得其尽可能高效？

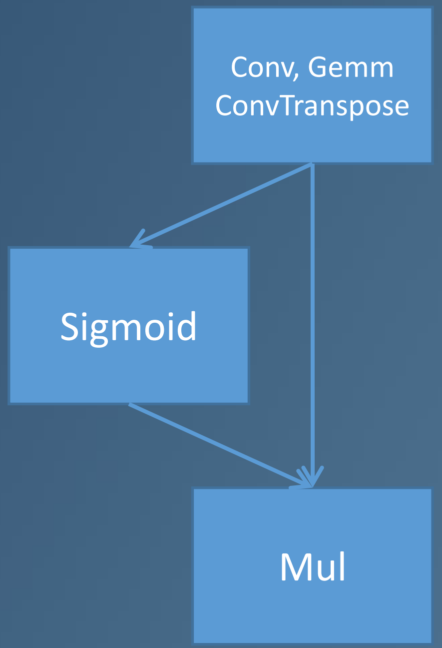
- 图模式匹配是量化算法、算子融合、算子调度的基础。

7.1 为什么需要图模式匹配

Why Graph Pattern Matching

例子1 ： Swish 算子图融合

```
search_engine = SearchableGraph(graph)
results = search_engine.pattern_matching(
            patterns = [lambda x: x.is_computing_op, 'Sigmoid', 'Mul'],
            edges = [[0, 1], [1, 2], [0, 2]],
            exclusive = True)
for computing_op, sigmoid, mul in results: ...
```

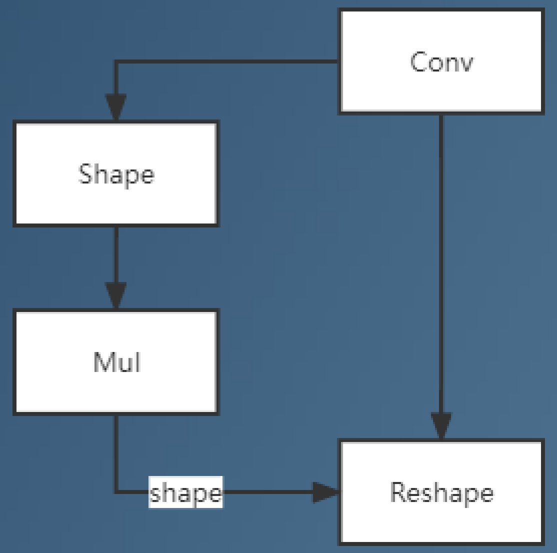


7.1 为什么需要图模式匹配

Why Graph Pattern Matching

例子2：算子调度

```
shape_backward_matching = search_engine.opset_matching(
        sp_expr = lambda x: x == shape,
        rp_expr = lamdba x, y: y.type not in {‘Reshape’, ‘Slice’, ...},
        ep_expr = lambda x: (x.type in {‘Reshape’, ‘Slice’, ...}),
        direction = 'down')
```



7.2 图模式匹配

Graph Pattern Matching

```
shape_backward_matching = search_engine.opset_matching(
            sp_expr = lambda x: x == shape,
            rp_expr = lamdba x, y: y.type not in {‘Reshape’, ‘Slice’, ...},
            ep_expr = lambda x: (x.type in {‘Reshape’, ‘Slice’, ...}),
            direction = 'down')
```

Traversal based matching

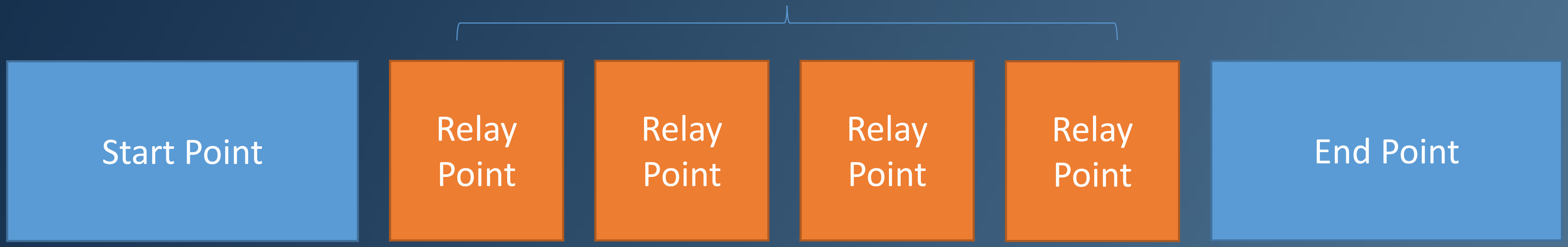
```
search_engine = SearchableGraph(graph)
results = search_engine.pattern_matching(
            patterns = [lambda x: x.is_computing_op, 'Sigmoid', 'Mul'],
            edges = [[0, 1], [1, 2], [0, 2]],
            exclusive = True)
for computing_op, sigmoid, mul in results: ...
```

Subgraph based matching

7.2.1 遍历模式匹配

Graph Traversal Pattern Matching

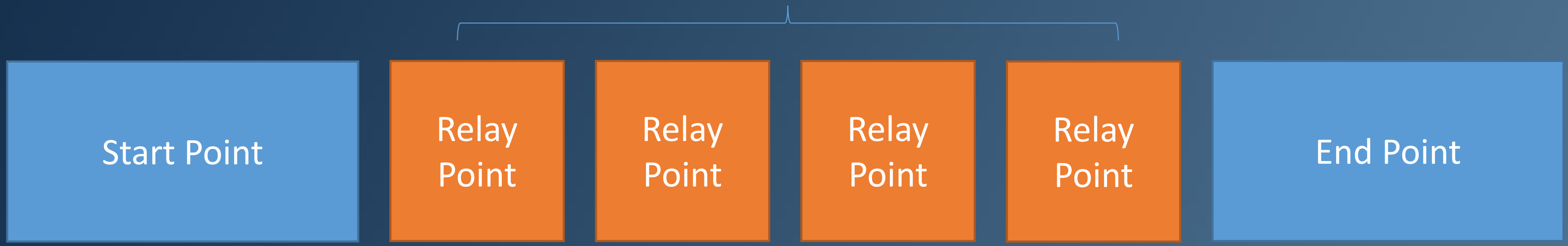
```
forward_matchings = search_engine(TraversalCommand(
    sp_expr=lambda x: ...,     (匹配起点表达式)      - - start point expression
    rp_expr=lambda x, y: ..., (匹配中继点表达式)   - - relay point expression
    ep_expr=lambda x: ...,     (匹配终点表达式)      - - end point expression
    direction='down'))
```



7.2.1 遍历模式匹配

Graph Traversal Pattern Matching

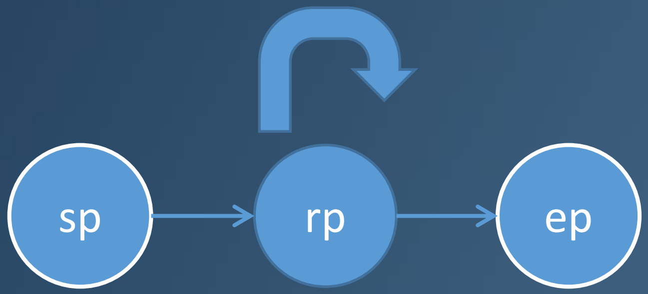
```
for operation in graph:
    if sp_expr(operation): # start point matched.
Iter:    next_op = next(operation)
         if ep_expr(next_op): return ret
         if rp_expr(next_op): goto Iter
```



7.2.1 遍历模式匹配

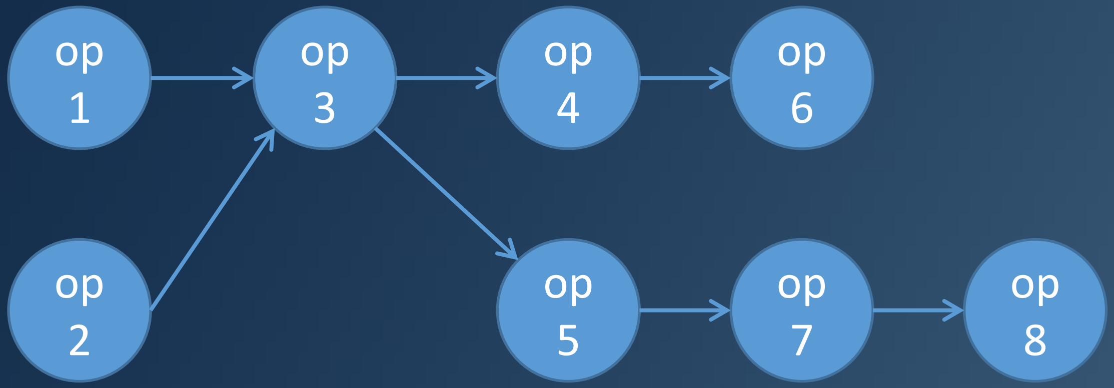
Graph Traversal Pattern Matching

```
forward_matchings = search_engine(TraversalCommand(
    sp_expr=lambda x: ...,     (匹配起点表达式)     -- start point expression
    rp_expr=lambda x, y: ..., (匹配中继点表达式)   -- relay point expression
    ep_expr=lambda x: ...,     (匹配终点表达式)     -- end point expression
    direction='down'))
```



7.2.1 遍历模式匹配

Graph Traversal Pattern Matching

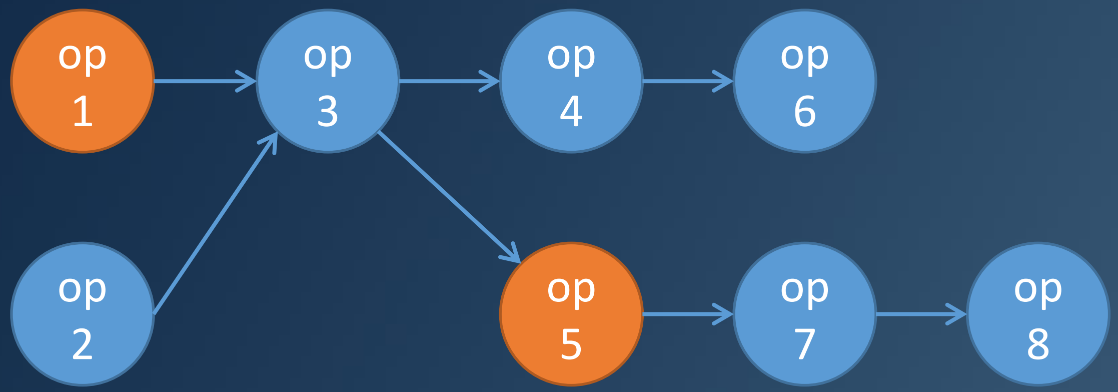


为了执行模式匹配，接下来我们需要将图逐步拆解成树，再到链。

1. 首先我们找出所有满足sp_expr的节点，将图拆解成一系列树
2. 再由树的根节点出发，将树剖分成链
3. 最后在链上执行模式匹配，期间可以利用动态规划进行优化

7.2.1 遍历模式匹配

Graph Traversal Pattern Matching

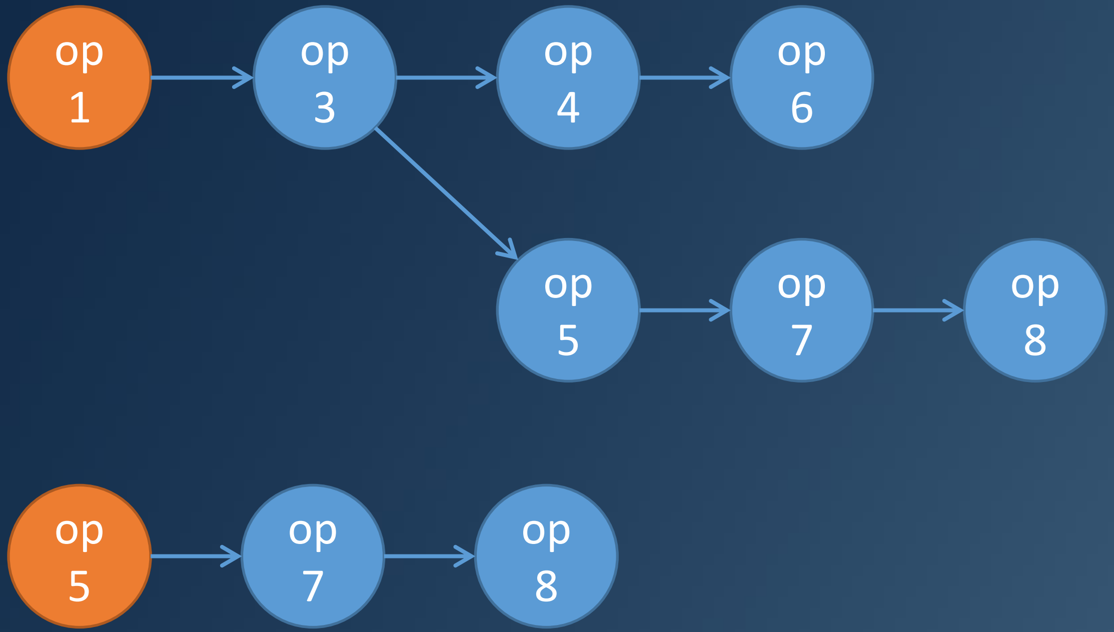


为了执行模式匹配，接下来我们需要将图逐步拆解成树，再到链。

1. 首先我们找出所有满足sp_expr的节点，将图拆解成一系列树
2. 再由树的根节点出发，将树剖分成链
3. 最后在链上执行模式匹配，期间可以利用动态规划进行优化

7.2.1 遍历模式匹配

Graph Traversal Pattern Matching

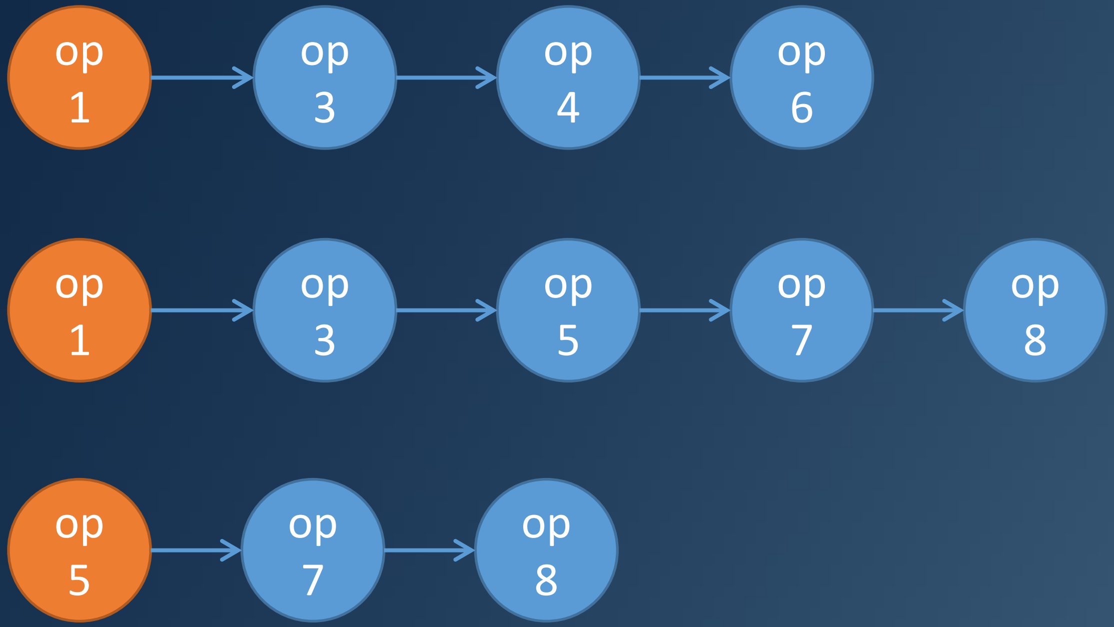


为了执行模式匹配，接下来我们需要将图逐步拆解成树，再到链。

1. 首先我们找出所有满足sp_expr的节点，将图拆解成一系列树
2. 再由树的根节点出发，将树剖分成链
3. 最后在链上执行模式匹配，期间可以利用动态规划进行优化

7.2.1 遍历模式匹配

Graph Traversal Pattern Matching

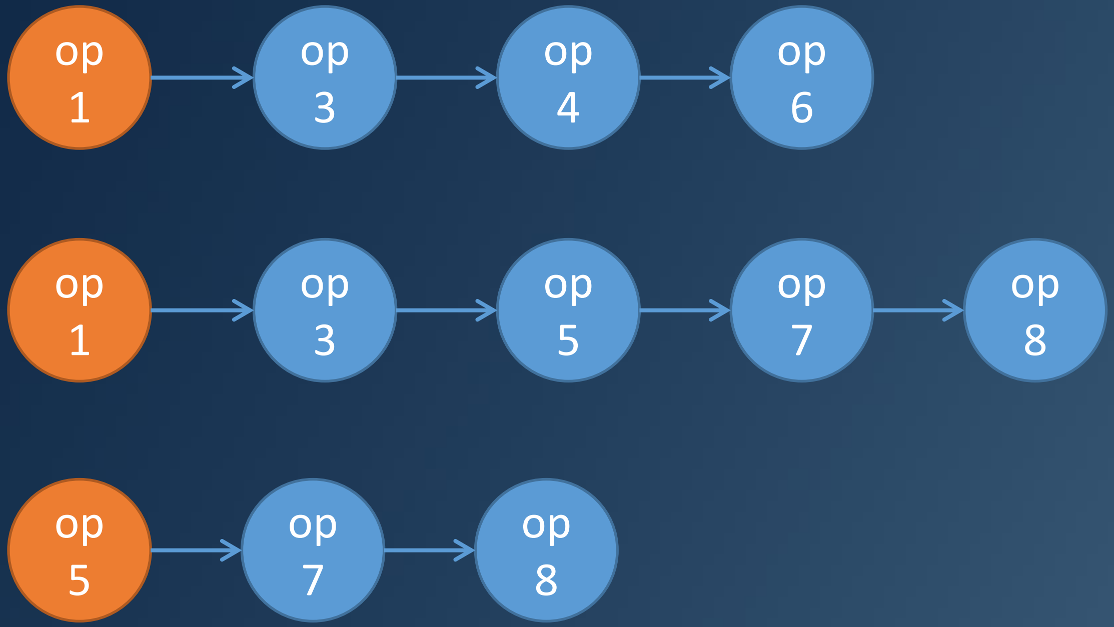


为了执行模式匹配，接下来我们需要将图逐步拆解成树，再到链。

1. 首先我们找出所有满足sp_expr的节点，将图拆解成一系列树
2. 再由树的根节点出发，将树剖分成链
3. 最后在链上执行模式匹配，期间可以利用动态规划进行优化

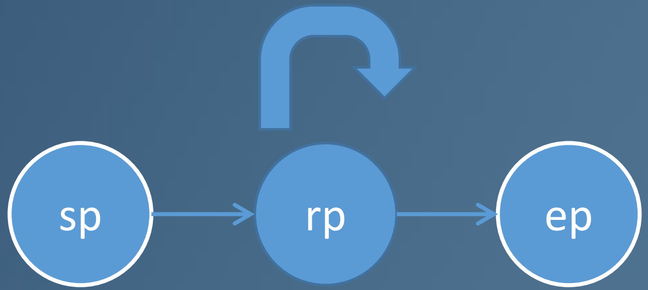
7.2.1 遍历模式匹配

Graph Traversal Pattern Matching



为了执行模式匹配，接下来我们需要将图逐步拆解成树，再到链。

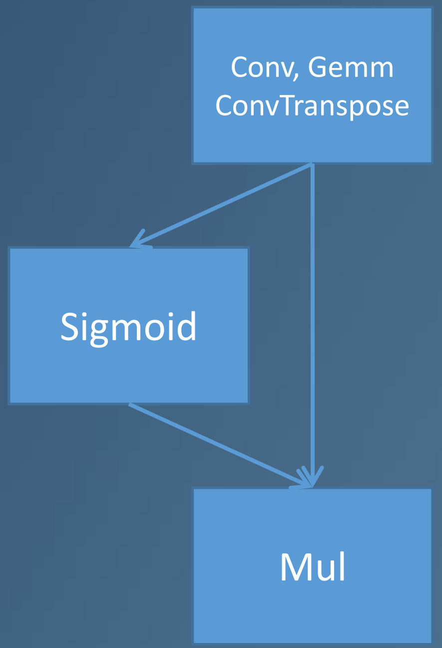
1. 首先我们找出所有满足sp_expr的节点，将图拆解成一系列树
2. 再由树的根节点出发，将树剖分成链
3. 最后在链上执行模式匹配，期间可以利用动态规划进行优化



7.2.2 子图模式匹配

SubGraph Pattern Matching

```
search_engine = SearchableGraph(graph)
results = search_engine.pattern_matching(
            patterns = [lambda x: x.is_computing_op, 'Sigmoid', 'Mul'],
            edges = [[0, 1], [1, 2], [0, 2]],
            exclusive = True)
for computing_op, sigmoid, mul in results: ...
```



很遗憾，子图同构问题是NP-hard的

PPQ提供多项式复杂度的近似算法

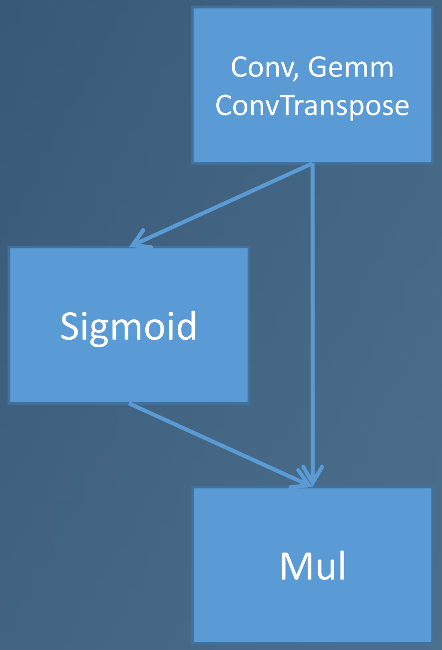
7.2.2 子图模式匹配

SubGraph Pattern Matching

```
for operation in graph:
    if operation.following == { 'Sigmoid', 'Mul' }
```

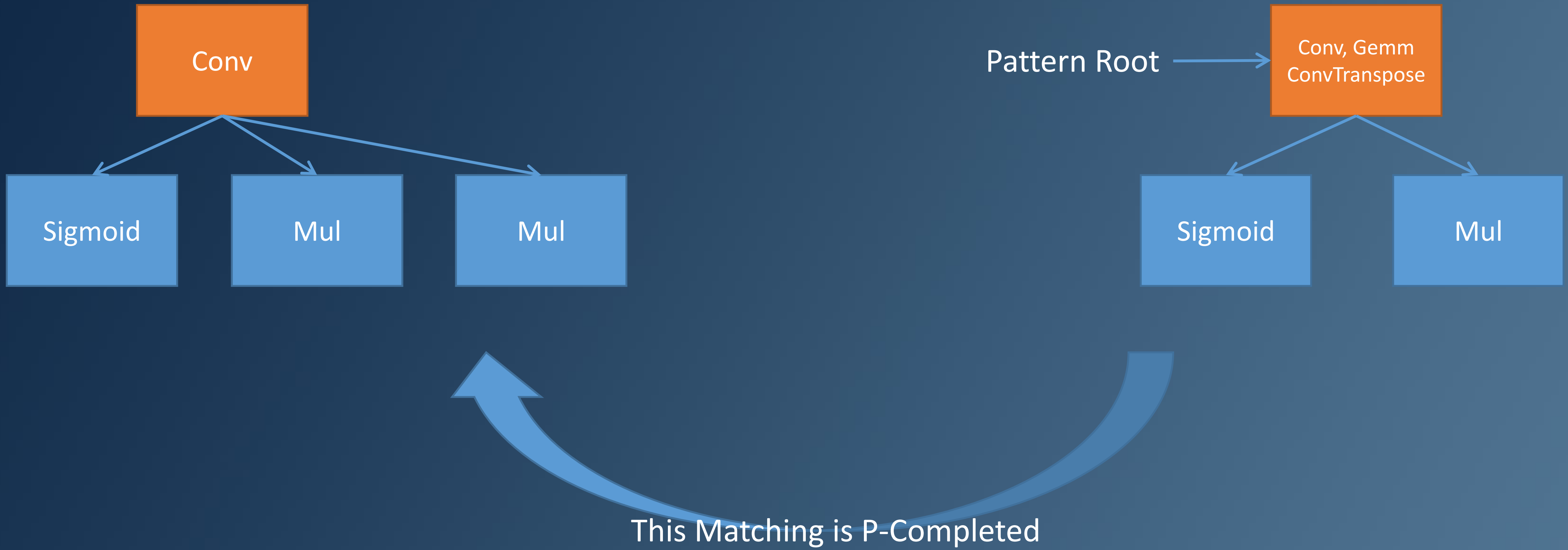
很遗憾，子图同构问题是NP-Hard的

PPQ提供多项式复杂度的近似算法



7.2.2 子图模式匹配

SubGraph Pattern Matching



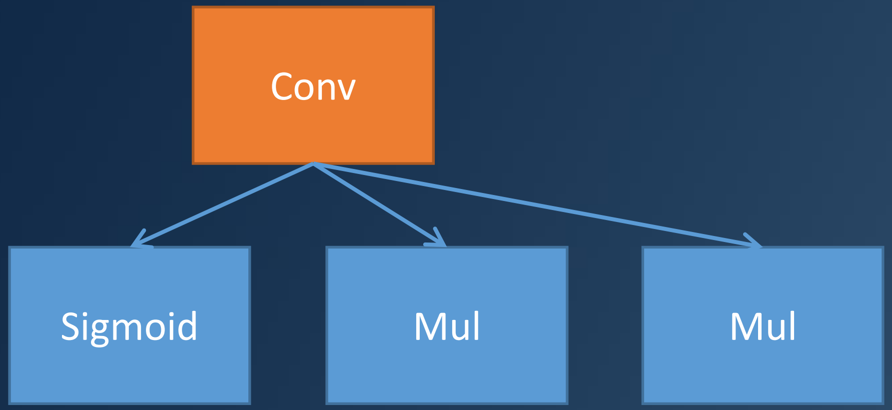
7.2.2 子图模式匹配

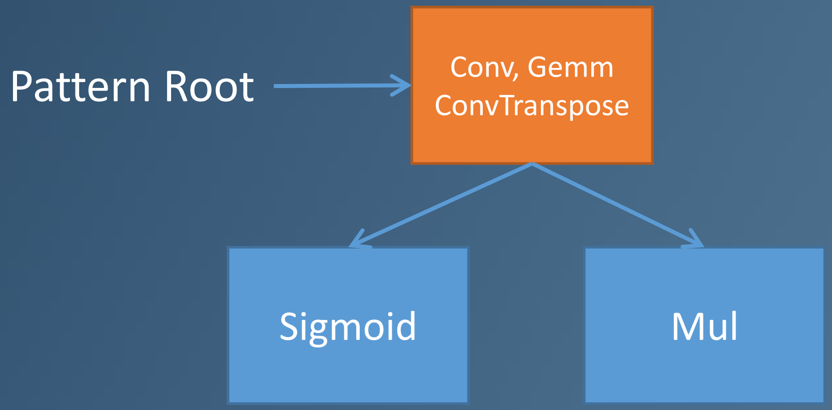
SubGraph Pattern Matching

For this situation, PPQ use Hungarian method to find an optimal matching(random).

7.2.2 子图模式匹配

SubGraph Pattern Matching






```
search_engine = SearchableGraph(graph)
results = search_engine.pattern_matching(
        patterns = [lambda x: x.is_computir _op, 'Sigmoid', 'Mul'],
        edges = [[0, 1], [1, 2], [0, 2]],
        exclusive = True)
for computing_op, sigmoid, mul in results: ...
```

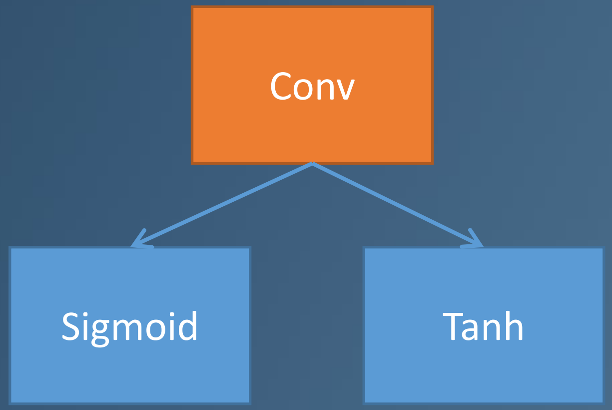
左图出现了失配节点，因此匹配直接失败。

7.2.2 子图模式匹配

SubGraph Pattern Matching

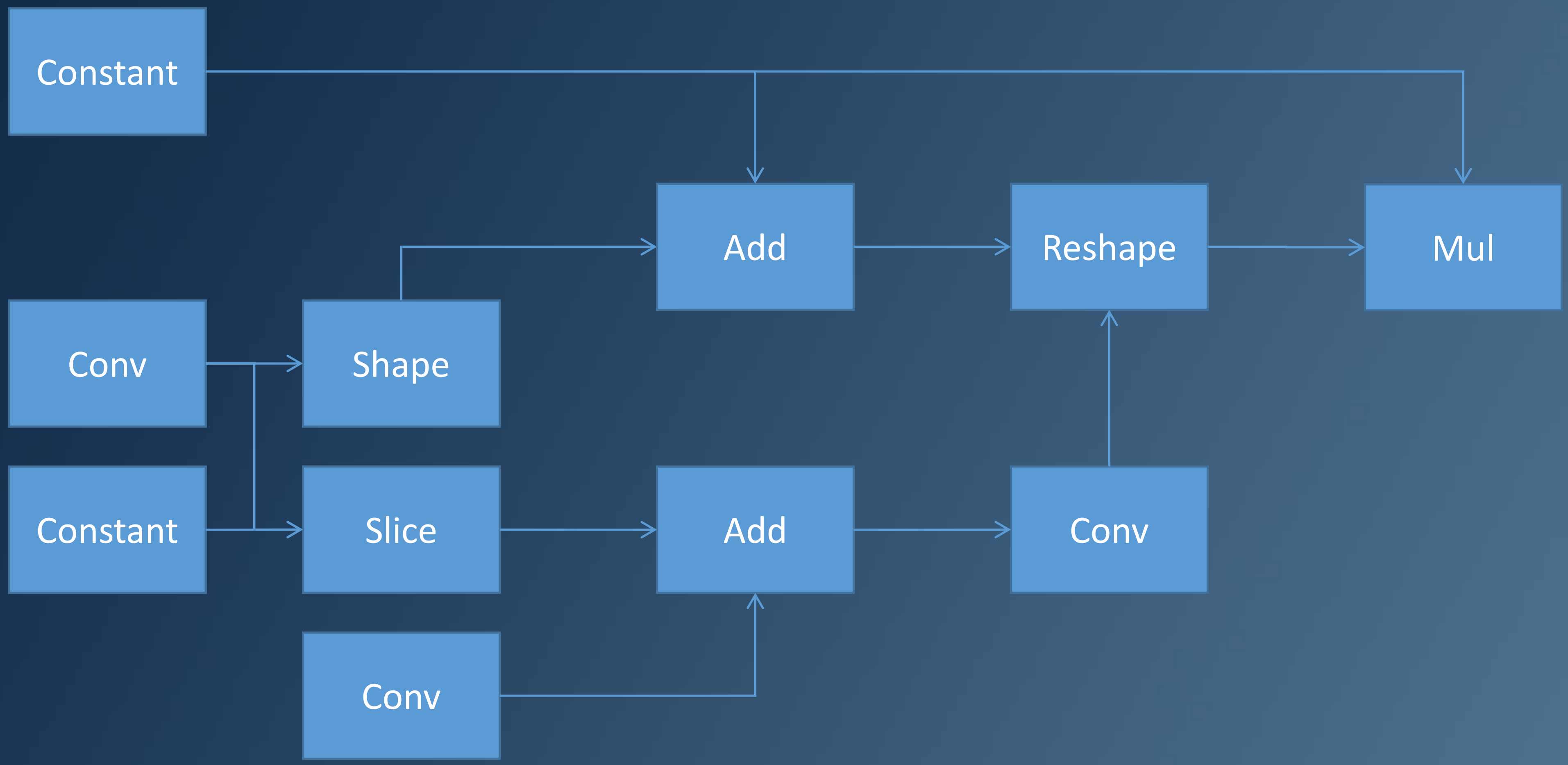
```
search_engine = SearchableGraph(graph)
results = search_engine.pattern_matching(
        patterns = [lambda x: x.is_computing_op, lambda x: x.type != 'Mul', lambda x: x.type != 'Mul'],
        edges = [[0, 1], [1, 2], [0, 2]],
        exclusive = True)
for computing_op, sigmoid, mul in results: ...
```

Do not use ambiguous pattern, it will cause random matching result, ppq will give warning about it.

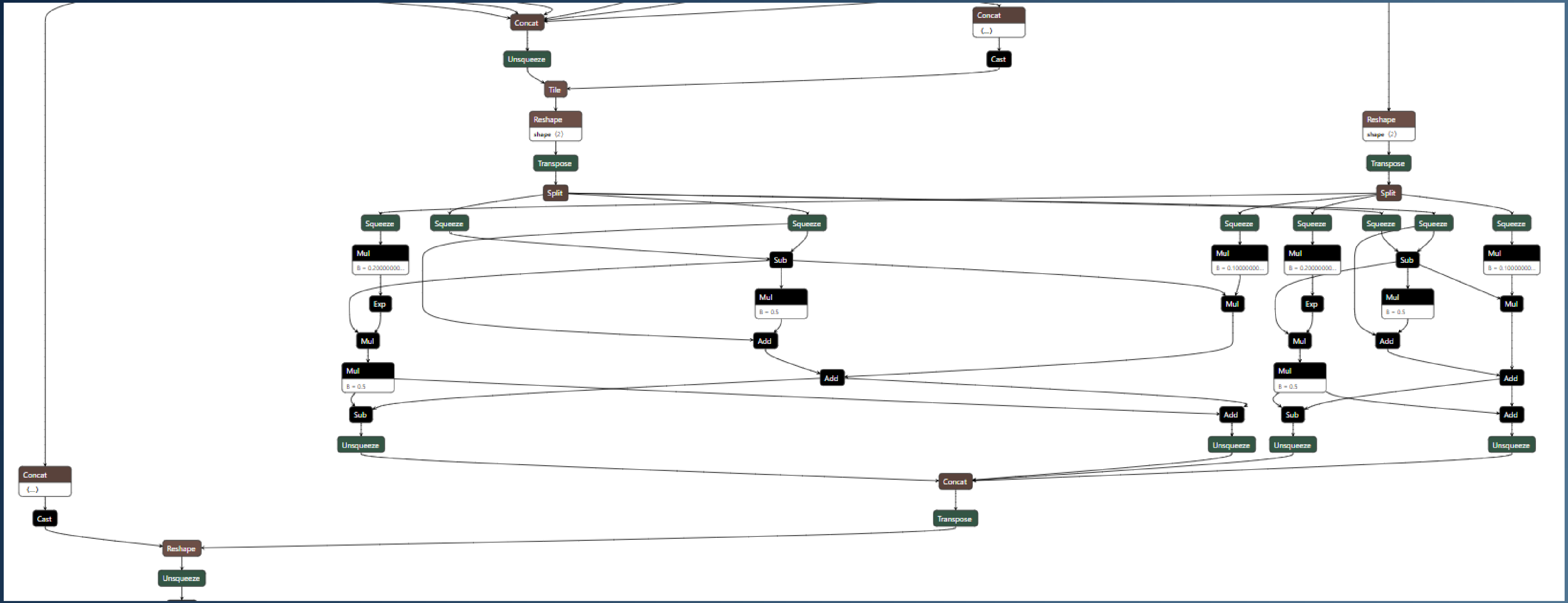


7.3 算子调度

Graph Dispatching

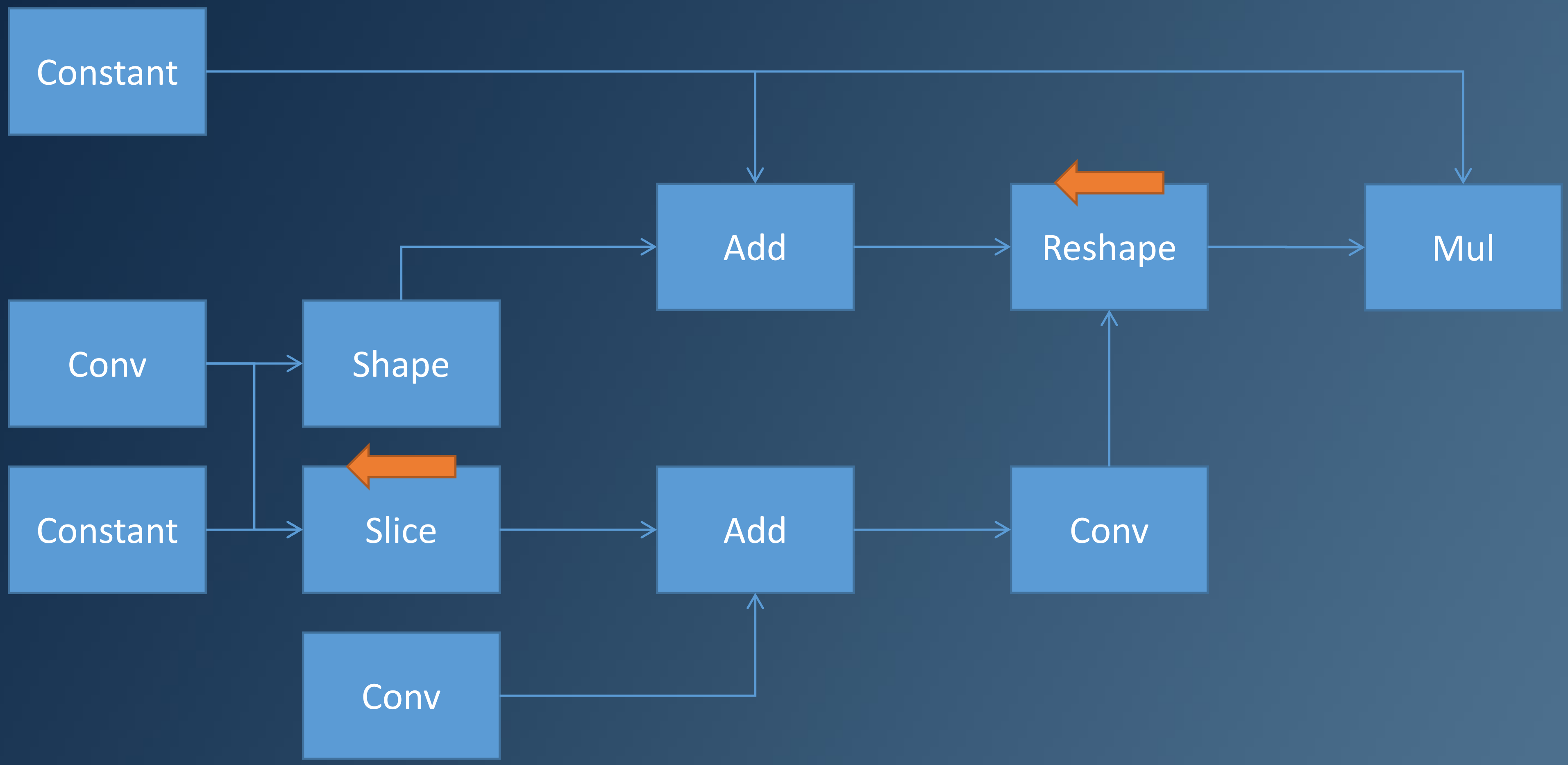


7.3 算子调度
Graph Dispatching



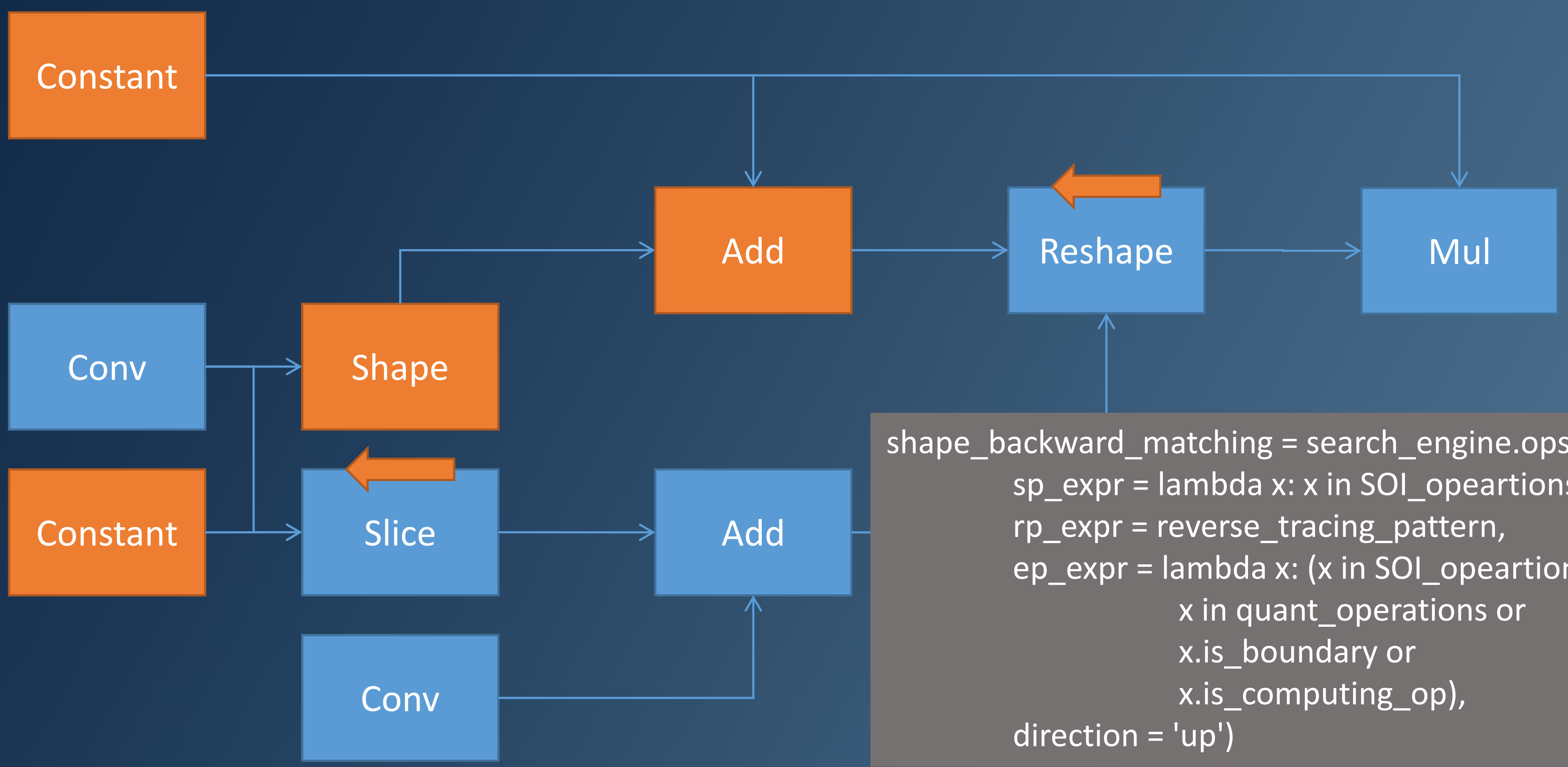
7.3 算子调度

Graph Dispatching



7.3.1 SOI 反向传播

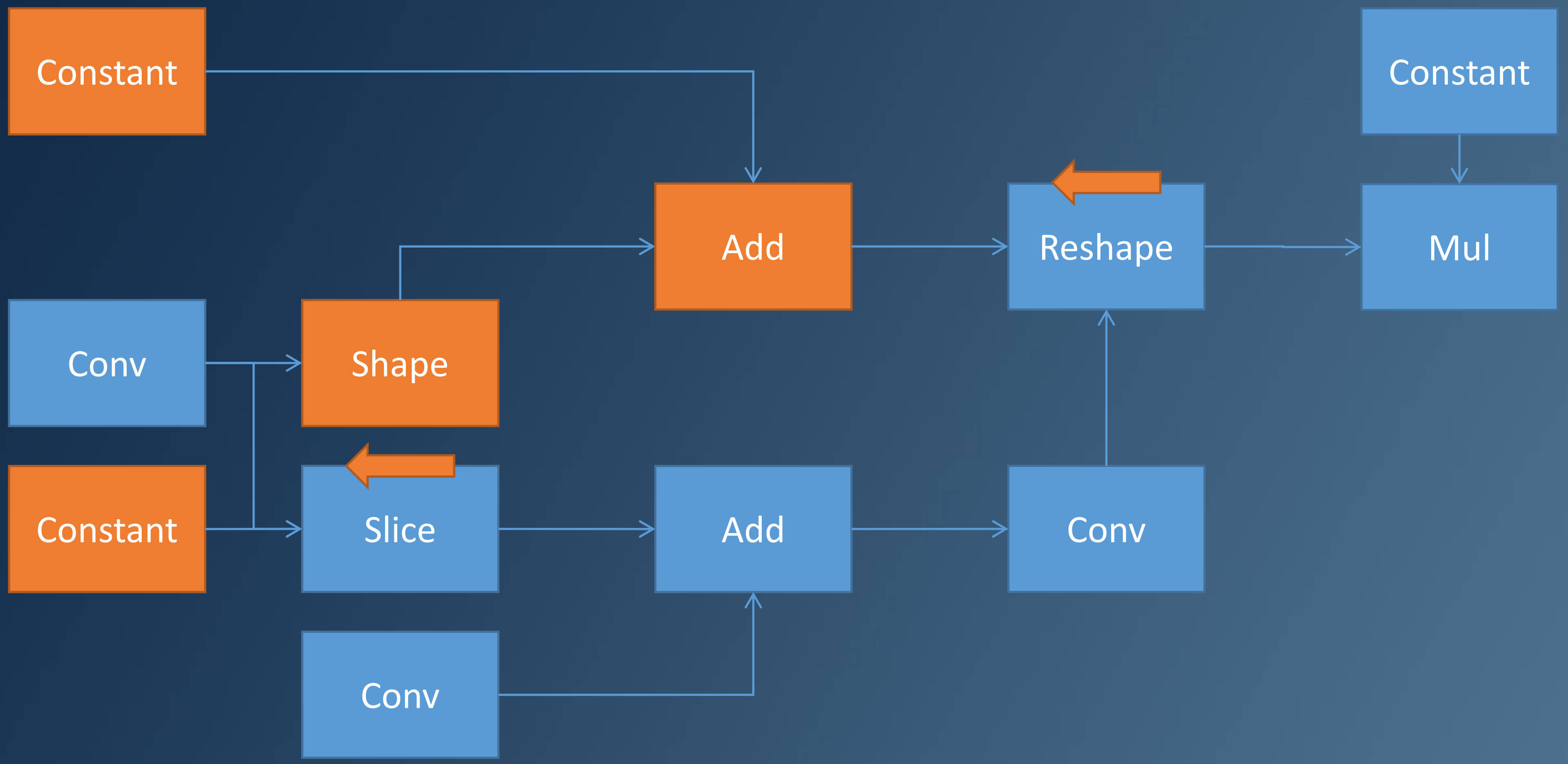
SOI backwards




```
shape_backward_matching = search_engine.opset_matching(
        sp_expr = lambda x: x in SOI_opeartions,
        rp_expr = reverse_tracing_pattern,
        ep_expr = lambda x: (x in SOI_opeartions or
                    x in quant_operations or
                    x.is_boundary or
                    x.is_computing_op),
        direction = 'up')
```

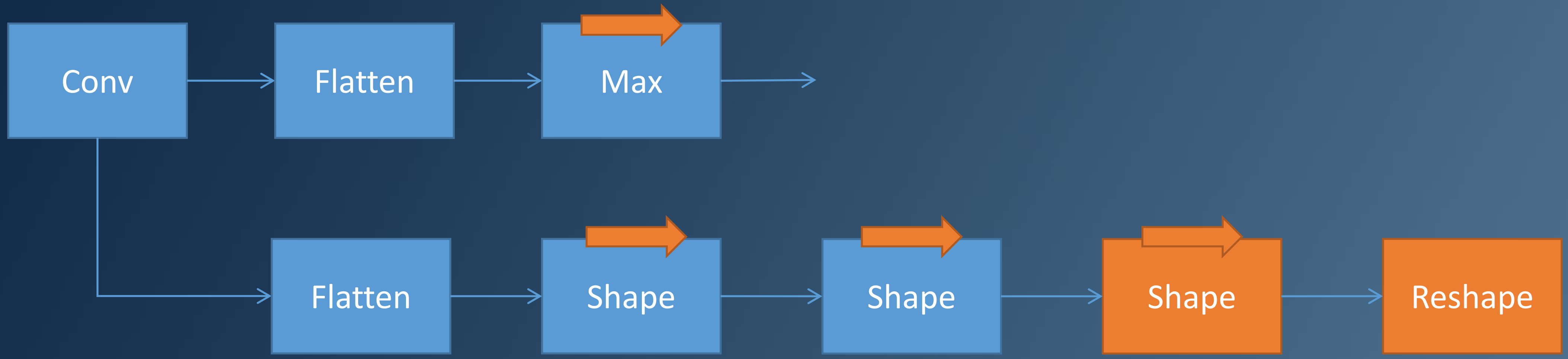
7.3.2 常量分裂

Constant Split



7.3.3 SOI 正向传播

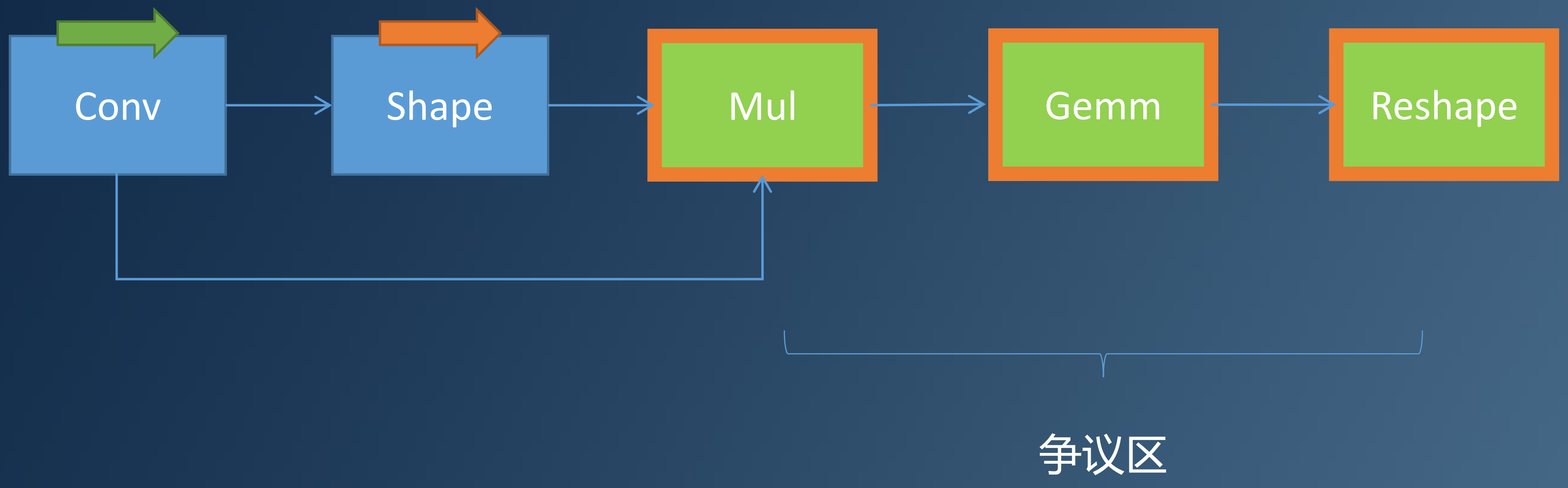
SOI forwards




```
shape_forward_matching = search_engine.opset_matching(
    sp_expr = lambda x: x in generators and x.type not in {'Constant'},
    rp_expr = value_tracing_pattern,
    ep_expr = lambda x: x in recivers or x in quant_operations or x.is_boundary,
    direction = 'down')
```

7.3.5 调度争议区

Dispatching Conflict Area

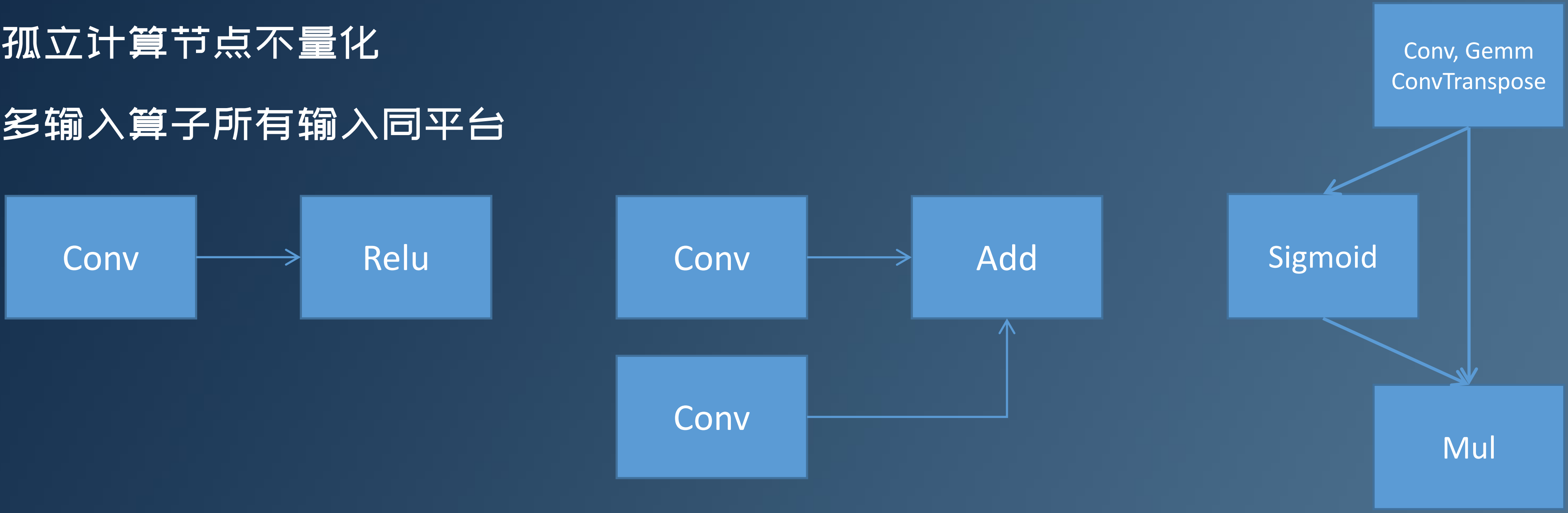


- AggresiveDispatcher: 所有争议区算子视为可量化算子
- ConservativeDispatcher: 所有争议区算子视作不可量化算子

7.3.6 调度约束

Dispatching Principles

- 激活函数与计算节点保持统一平台
- NMS，SHAPE，TOPK，MAX与计算节点保持同一平台
- 参与图融合的算子保持统一平台
- 孤立计算节点不量化
- 多输入算子所有输入同平台



7.3.6 调度约束

Dispatching Principles

- 激活函数与计算节点保持统一平台
- NMS, SHAPE, TOPK, MAX与计算节点保持同一平台
- 参与图融合的算子保持统一平台
- 孤立计算节点不量化
- 多输入算子所有输入同平台

7.3.7 手动调度

PPQ dispatching table

```
quant_setting = QuantizationSettingFactory.default_setting()
quant_setting.equalization = True # use layerwise equalization algorithm.
quant_setting.dispatcher   = 'conservative' # dispatch this network in conservertive way.

reports = layerwise_error_analyse(
    graph=quantized, running_device=DEVICE, collate_fn=collate_fn,
    dataloader=calibration_dataloader)
```

```
Conv_4:   | ████████████████████ |
0.015058
Conv_138: | █████                | 0.003621
Conv_32:  | ████                 | 0.002722
Conv_133: | ███                  | 0.002622
Conv_22:  | ███                  | 0.001980
Conv_142: | ███                  | 0.001965
Conv_109: | ██                   | 0.001746
Conv_123: | ██                   | 0.001680
Conv_119: | ██                   | 0.001645
Conv_100: | ██                   | 0.001411
Conv_13:  | ██                   | 0.001378
```

7.3.7 手动调度

PPQ dispatching table

```
quant_setting = QuantizationSettingFactory.default_setting()
quant_setting.equalization = True # use layerwise equalization algorithm.
quant_setting.dispatcher   = 'conservative' # dispatch this network in conservertive way.

reports = layerwise_error_analyse(
    graph=quantized, running_device=DEVICE, collate_fn=collate_fn,
    dataloader=calibration_dataloader)

quant_setting.dispatching_table.append( 'Conv4', TargetPlatform.FP32)
quant_setting.dispatching_table.append( 'Conv138', TargetPlatform.FP32)
```

7.3.7 手动调度

PPQ dispatching table

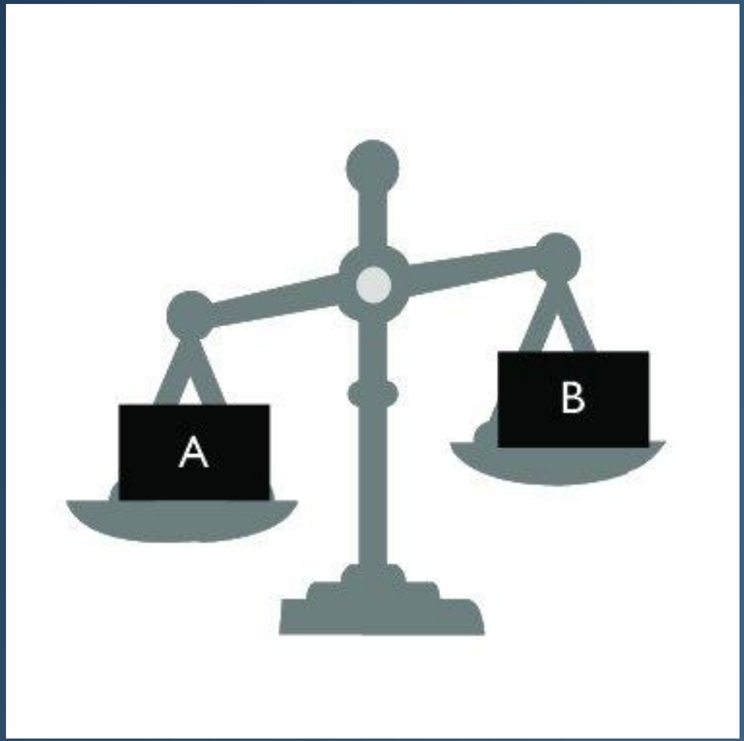
你需要知道手动调度对于量化而言意味着什么：

- 对于CPU, GPU而言，被调度到FP32平台的算子将牺牲计算时间来换取计算精度，在量化子图与非量化子图之间将存在精度转换节点。在这样的硬件上，网络的量化其实只是在权衡网络精度与速度。
- 对于FPGA, DSP而言，硬件端没有浮点运算能力，手动调度算子可能造成网络无法执行。在最优情况下，浮点算子将被调度到主机端进行运算，为此你的推理框架需要对此有相应实现，并且有设备流水线的功能。
- 混合精度量化调度极度复杂，如果你对 1-8 bit 混合精度量化感兴趣，请注意真实的计算绝不是简单地将conv, gemm等调度到到不同平台，而是牵连着许多算子一同调度，其过程是难以工业化的。

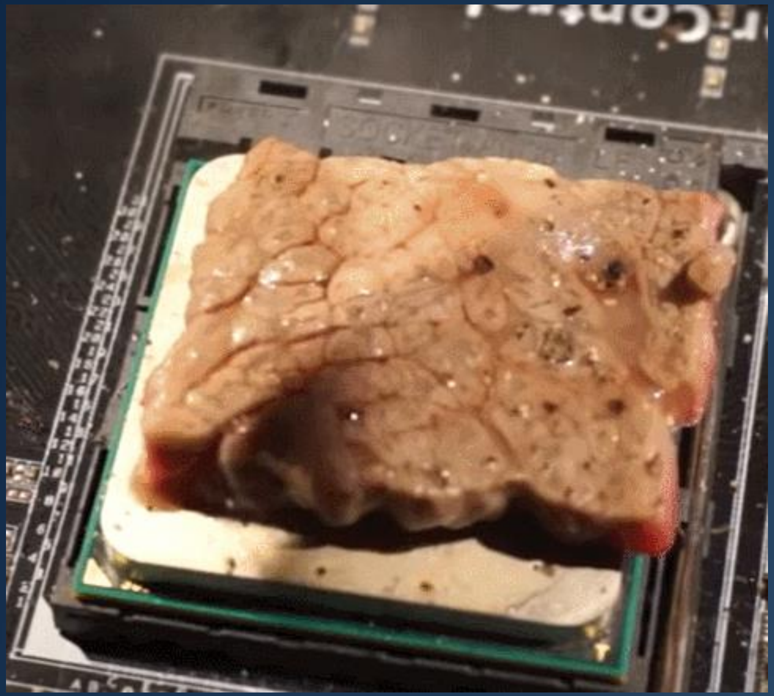
7.3.7 手动调度

PPQ dispatching table

- 在真正的部署过程中，量化实际上是一个执行速度与网络精度权衡问题。
- 网络调度是量化走向工业化所必须的特性，是稳定部署所必须的。



联系我们 https://github.com/openppl-public



广告位招租

微信群

QQ群（入群密令OpenPPL）