一、 构建神经网络

1.1. 神经网络量化

Resnet 是残差网络(Residual Network)的缩写,该系列网络广泛用于目标分类等领域以及作为计算机视觉任务主干经典神经网络的一部分，典型的网络有 resnet50, resnet101 等。Resnet 网络的证明网络能够向更深（包含更多隐藏层）的方向发展。

为了将神经网络部署到加速器上，我们设计的基本编译流程主要分为五个阶段，其框架如图 1 所示

图 1 神经网络的编译与映射流程

为了在硬件上部署 resnet50 网络并进行测试，我们首先通过 tvm 导入训练好的 resnet50 网络和对应的训练集，并通过 tvm.quantize.quantize 函数进行量化，获取量化后的 resnet50 网络和对应参数，其代码如图 2。

```
if target.device_name == "vta":
    # Perform quantization in Relay
    # Note: We set opt_level to 3 in order to fold batch norm
    with tvm.transform.PassContext(opt_level=3):
        with relay.quantize.qconfig(global_scale=8.0, skip_conv_layers=[0]):
            mod = relay.quantize.quantize(mod, params=params)
        # Perform graph packing and constant folding for VTA target
        assert env.BLOCK_IN == env.BLOCK_OUT
        # do device annotation if target is intelfocl or sim
        # relay_prog = graph_pack(
        #     mod["main"],
        #     env.BATCH,
        #     env.BLOCK_OUT,
        #     env.WGT_WIDTH,
        #     start_name=pack_dict[model][0],
        #     stop_name=pack_dict[model][1],
        #     device_annot=(env.TARGET == "intelfocl" or env.TARGET == "sim"),
        # )
        relay_prog = mod["main"]
else:
    relay_prog = mod["main"]

# Compile Relay program with AlterOpLayout disabled
if target.device_name != "vta":
    with tvm.transform.PassContext(opt_level=3, disabled_pass={"AlterOpLayout"}):
        graph, lib, params = relay.build(
            relay_prog, target=target, params=params, target_host=env.target_host
        )
else:
    if env.TARGET == "intelfocl" or env.TARGET == "sim":
        # multiple targets to run both on cpu and vta
        target = {"cpu": env.target_vta_cpu, "ext_dev": target}
    with vta.build_config(opt_level=3, disabled_pass={"AlterOpLayout"}):
        graph, lib, params = relay.build(
            relay_prog, target=target, params=params, target_host=env.target_host
        )
```

图 2 神经网络量化

其中，mod 为量化后得到的网络，params 为量化后得到的参数表，将 mod 打印后可以看到量化后的 resnet50 共有 700 多层，如图 2 所示。接下来我们基于量化后的 resnet50 网络来构建所需要的算子，并且考虑如何利用硬件加速器来实现各个算子的功能，以及如何将算子映射为 rocc 指令。

```
def @main(%data: Tensor[(1, 3, 224, 224), float32]) -> Tensor[(1, 1000), float32] {
  %0 = nn.conv2d(%data, meta[relay.Constant][0] /* ty=Tensor[(64, 3, 7, 7), float32] */, strides=[2, 2], padding=[3, 3, 3, 3], channels=64, ke
  %1 = add(%0, meta[relay.Constant][1] /* ty=Tensor[(64, 1, 1), float32] */) /* ty=Tensor[(1, 64, 112, 112), float32] */;
  %2 = nn.relu(%1) /* ty=Tensor[(1, 64, 112, 112), float32] */;
  %3 = nn.max_pool2d(%2, pool_size=[3, 3], strides=[2, 2], padding=[1, 1, 1, 1]) /* ty=Tensor[(1, 64, 56, 56), float32] */;

  #stage1
  #BTNK 1 1-1
  %4 = annotation.stop_fusion(%3) /* ty=Tensor[(1, 64, 56, 56), float32] */;
  %5 = multiply(%4, 16f /* ty=float32 */) /* ty=Tensor[(1, 64, 56, 56), float32] */;
  %6 = round(%5) /* ty=Tensor[(1, 64, 56, 56), float32] */;

  %7 = clip(%6, a_min=-127f, a_max=127f) /* ty=Tensor[(1, 64, 56, 56), float32] */;
  %8 = cast(%7, dtype="int8") /* ty=Tensor[(1, 64, 56, 56), int8] */;

  #fuse_op1
  %9 = nn.conv2d(%8, meta[relay.Constant][2] /* ty=Tensor[(64, 64, 1, 1), int8] */, padding=[0, 0, 0, 0], channels=64, kernel_size=[1, 1], out
  %10 = left_shift(%9, 11 /* ty=int32 */) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %11 = add(%10, meta[relay.Constant][3] /* ty=Tensor[(64, 1, 1), int32] */) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %12 = add(%11, meta[relay.Constant][4] /* ty=Tensor[(64, 1, 1), int32] */) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %13 = nn.relu(%12) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %14 = add(%13, 131072 /* ty=int32 */) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %15 = right_shift(%14, 18 /* ty=int32 */) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %16 = clip(%15, a_min=-127f, a_max=127f) /* ty=Tensor[(1, 64, 56, 56), int32] */;

  %17 = cast(%16, dtype="int8") /* ty=Tensor[(1, 64, 56, 56), int8] */;
  %18 = annotation.stop_fusion(%17) /* ty=Tensor[(1, 64, 56, 56), int8] */;

  #fuse_op2
  %19 = nn.conv2d(%18, meta[relay.Constant][5] /* ty=Tensor[(64, 64, 3, 3), int8] */, padding=[1, 1, 1, 1], channels=64, kernel_size=[3, 3], o
  %20 = add(%19, meta[relay.Constant][6] /* ty=Tensor[(64, 1, 1), int32] */) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %21 = nn.relu(%20) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %22 = add(%21, 64 /* ty=int32 */) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %23 = right_shift(%22, 7 /* ty=int32 */) /* ty=Tensor[(1, 64, 56, 56), int32] */;
  %24 = clip(%23, a_min=-127f, a_max=127f) /* ty=Tensor[(1, 64, 56, 56), int32] */;

  %25 = cast(%24, dtype="int8") /* ty=Tensor[(1, 64, 56, 56), int8] */;
  %26 = annotation.stop_fusion(%25) /* ty=Tensor[(1, 64, 56, 56), int8] */;
```

图 3 量化后的 resnet50（局部）

1.2. 卷积算子实现

为了进行卷积，首先要对 feature map 进行 padding 操作，如图 4 所示。

```
# padding for convolution
if padding:
    self.pad_data = topi.nn.pad(self.data, [0, self.padding, self.padding, 0], name="pad_data")
else:
    self.pad_data = self.data
```

图 4 padding 操作

Padding 之后，由于默认 batch=1，为了能够使得 feature map 数据能够被适配到 8*8 的 gemm 中，需要做进一步的 padding，融合 h 和 w 两个维度，并将 h*w 补齐至能够被 8 所整除。下图代码是计算需要对 feature map 的 hw 融合维度补多少 0 才能被 8 整除。

```
# calculate padding parameter
if (oheight * owidth) % gemm_size == 0:
    padded_length = oheight * owidth
    middle_pad = 0
else:
    padded_length = ((oheight * owidth) // gemm_size + 1) * gemm_size
    middle_pad = padded_length - oheight * owidth
```

图 5 计算所需补 0 数

对 feature map 和 weight 都做 img2col 变换，变换后数据维度变成了 2 维，然后对 2 维数据做 padding，让所有边都能够被 8 整除，方便分块。然后对参数进行分块，分块使用的是 tvm 的 topi.reshape API，分块后，用 topi.transpose API 对数据的维度进行调整，使得数据分块后的排布方式是按照大 Z 小 z 的方式排布。

```
self.mdata = te.compute(
    (ishape[0], oheight, owidth, kshape[1], kshape[2], input_channel),
    lambda n, o1, o2, k1, k2, c: self.pad_data[0, k1 + o1 * strides[0], k2 + o2 * strides[1], c],
    name="mdata"
)

# reshape data and weight to 2d
self.mdata_2d = topi.reshape(self.mdata, (ishape[0] * oheight * owidth, input_channel * window_size))
self.kernel_2d = topi.reshape(self.kernel, (output_channel, input_channel * window_size))

# pad data_2d to match [16, 16]
self.mdata_pad = topi.nn.pad(self.mdata_2d, [0, 0], [middle_pad, 0], name="mdata_pad")

# split data and weight with 16
self.mdata_pad_4d = topi.reshape(self.mdata_pad,
                                 (padded_length // gemm_size, gemm_size, input_channel * window_size // gemm_size, gemm_size))
self.kernel_4d = topi.reshape(self.kernel_2d, (output_channel // gemm_size, gemm_size, input_channel * window_size // gemm_size, gemm_size))

# transpose data and weight to move two axis of 16 to the end
self.A = topi.transpose(self.mdata_pad_4d, (0, 2, 1, 3))
self.B = topi.transpose(self.kernel_4d, (0, 2, 1, 3))
```

图 6 参数分块

这里是由于 feature buffer 和 weight buffer 比较小，因此还要再对数据做切割，feature buffer 和 weight buffer 能存下 144 个 8*8 的数据，这里需要对 c*k*k 的值进行分类，尽量保证分块的大小不要留余数。

```
reduction_repeat = A_shape[1]
if(reduction_repeat>128):
    if(reduction_repeat%128==0):
        self.A1 = topi.reshape(self.A, (A_shape[0], reduction_repeat // 128, 128, gemm_size, gemm_size))
        self.B1 = topi.reshape(self.B, (B_shape[0], reduction_repeat // 128, 128, gemm_size, gemm_size))
    elif(reduction_repeat%72==0):
        self.A1 = topi.reshape(self.A, (A_shape[0], reduction_repeat // 72, 72, gemm_size, gemm_size))
        self.B1 = topi.reshape(self.B, (B_shape[0], reduction_repeat // 72, 72, gemm_size, gemm_size))
    else:
        print("error")
        self.A1 = topi.reshape(self.A, (A_shape[0], 1, A_shape[1], gemm_size, gemm_size))
        self.B1 = topi.reshape(self.B, (B_shape[0], 1, B_shape[1], gemm_size, gemm_size))
else:
    self.A1 = topi.reshape(self.A, (A_shape[0], 1, A_shape[1], gemm_size, gemm_size))
    self.B1 = topi.reshape(self.B, (B_shape[0], 1, B_shape[1], gemm_size, gemm_size))
```

图 7 数据切割

```
self.k0 = te.reduce_axis((0, A1_shape[1]), name="k0")
self.k1 = te.reduce_axis((0, A1_shape[2]), name="k1")
self.k2 = te.reduce_axis((0, gemm_size), name="k2")

self.middle_buffer = te.compute(
    (A1_shape[0], B1_shape[0], gemm_size, gemm_size),
    lambda i, x, j, y: te.sum(
        self.feature[i, self.k0, self.k1, j, self.k2].astype("int32") * self.weight[x, self.k0, self.k1, y, self.k2].astype("int32"),
        axis=[self.k0, self.k1, self.k2]
    ),
    name="middle_buffer"
)
```

图 8 矩阵乘分块操作

分块乘之后的结果依旧是大 Z 小 z 的格式，为了进行后续的操作，这里需要对数据进行格式还原，并且把之前为了能够匹配 8*8 做的 padding 的数据也进行消除，最终还原成 nhwc 的格式：

```
self.middle_tran = topi.transpose(self.middle, (0, 2, 1, 3))
self.middle_reshape = topi.reshape(self.middle_tran, (padded_length, output_channel))
self.middle_unpad = topi.nn.pad(self.middle_reshape, [0, 0], [-middle_pad, 0], name="middle_unpad")

self.out = topi.reshape(self.middle_unpad, (1, oheight, owidth, output_channel))
```

图 9 数据格式还原

1.3. Add 算子实现

Add 算子主要分为 add_num 算子和 add_tensor 算子两部分。其中 add_num 算子主要负责将左操作数张量中的每个元素都加上右操作数。为了实现这个过程，首先我们需要将右操作数存储在 input_buffer 中的某个区域，当需要调用它时，则将右操作数通过 DMA 操作扩张到与左操作数相同的规模，再搬移到 mid_buffer 中，同时另一个 DMA 将左操作数也 load 到 mid_buffer 中，当搬移操作完成后，就执行 ve 操作，将两个张量相加并将结果存储到 output_buffer 中，最后再等待复用或者搬出到 dram 中

```
class add_num:
    def __init__(self, data1,data2):
        self.data1=data1
        self.data2=data2

        dshape = topi.utils.get_const_tuple(self.data1.shape)
        self.data2_i = te.compute(dshape, lambda n, h, w, c: self.data2, "data2_i")
        self.data1_in = te.compute(dshape, lambda n, h, w, c: self.data1[n, h, w, c], "data1_in")
        self.data2_in = te.compute(dshape, lambda n, h, w, c: self.data2_i[n,h,w,c], "data2_in")

        self.cout = te.compute(
        dshape,
        lambda n, h, w, c: self.data1_in[n, h, w, c] + self.data2_in[n, h, w, c],
        name="cout")

        self.out = te.compute(dshape, lambda n, h, w, c: self.cout[n, h, w, c], "out")
```

图 10 加法算子实现

与 add_num 不同，add_tensor 操作主要负责两个张量相加，这又包含两种不同的操作类型。其一是直接将两个规模相等的张量相加，此操作只需要将左操作数和右操作数分别 load 到 mid_buffer 中再相加即可；另一种操作则是要将右操作数按 channel 加到左操作数上，这种操作同样需要先通过 DMA 将右操作数按 channel 扩张到与左操作数相同规模，然后再搬移到 mid_buffer 中进行 ve 操作，最后结果输出到 output_buffer 等待复用或者移出到 dram

```
class add_tensor:
    def __init__(self, data1, data2):
        self.data1 = data1
        self.data2 = data2

        print(type(self.data1))
        dshape1 = topi.utils.get_const_tuple(self.data1.shape)
        dshape2 = topi.utils.get_const_tuple(self.data2.shape)
        # n=int(n)
        # h=int(h)
        # w=int(w)
        # c=int(c)
        self.data2_i = te.compute(dshape1, lambda n, h, w, c: self.data2[c, 0, 0], "data2_i")

        self.data1_in = te.compute(dshape1, lambda n, h, w, c: self.data1[n, h, w, c], "data1_in")
        self.data2_in = te.compute(dshape1, lambda n, h, w, c: self.data2_i[n, h, w, c], "data2_in")

        self.cout = te.compute(
            dshape1,
            lambda n, h, w, c: self.data1_in[n, h, w, c] + self.data2_in[n, h, w, c],
            name="cout")

        self.out = te.compute(dshape1, lambda n, h, w, c: self.cout[n, h, w, c], "out")
```

图 11 加法算子实现（二）

1.4. Mul 算子实现

Mul 算子主要负责张量的乘法，其功能是将右操作数按 channel 与左操作数相乘。该算子的执行过程主要是将右操作数先通过 DMA 按 channel 扩张到与左操作数一样的规模，再将其搬移到 mid_buffer 中，同时通过 DMA 将左操作数也通过 DMA 搬移到 mid_buffer 中，当搬移完成后，通过 ve 指令将其对应位置相乘，并将结果输出到 output_buffer，等待后续的复用或者搬出到 dram。

```python
class mul_tensor:
    def __init__(self, data1, data2):
        self.data1 = data1
        self.data2 = data2

        dshape1 = topi.utils.get_const_tuple(self.data1.shape)
        dshape2 = topi.utils.get_const_tuple(self.data2.shape)

        self.data2_i = te.compute(dshape1, lambda n, h, w, c: self.data2[c, 0, 0], "data2_i")

        self.data1_in = te.compute(dshape1, lambda n, h, w, c: self.data1[n, h, w, c], "data1_in")
        self.data2_in = te.compute(dshape1, lambda n, h, w, c: self.data2_i[n, h, w, c], "data2_in")

        self.cout = te.compute(
            dshape1,
            lambda n, h, w, c: self.data1_in[n, h, w, c] * self.data2_in[n, h, w, c],
            name="cout")

        self.out = te.compute(dshape1, lambda n, h, w, c: self.cout[n, h, w, c], "out")
```

图 12　mul 算子实现

1.5. Shift 算子实现

Shift 算子主要负责移位操作，即将左操作数中的每个元素放缩等同于右操作数数值的二进制位。该操作分为左移和右移，分别对应数据放大和缩小。当执行该算子时，首先将右操作数通过 DMA 扩张到与左操作数相同规模，再搬移到 mid_buffer 中，同时通过 DMA 将右操作数也搬移到 mid_buffer 中，待搬移完成后，执行对应的 ve 操作，并将结果输出到 output_buffer 中，等待后续复用或者搬出到 dram。

```python
class left_shift:
    def __init__(self, lhs,rhs):
        self.data1=lhs
        self.data2=rhs

        dshape = topi.utils.get_const_tuple(self.data1.shape)
        self.data2_i = te.compute(dshape, lambda n, h, w, c: self.data2, "data2_i")
        self.data1_in = te.compute(dshape, lambda n, h, w, c: self.data1[n, h, w, c], "data1_in")
        self.data2_in = te.compute(dshape, lambda n, h, w, c: self.data2_i[n,h,w,c], "data2_in")

        self.cout = te.compute(
        dshape,
        lambda n, h, w, c: self.data1_in[n, h, w, c] << self.data2_in[n, h, w, c],
        name="cout")

        self.out = te.compute(dshape, lambda n, h, w, c: self.cout[n, h, w, c], "out")
```

图 13　shift 算子实现

```python
class right_shift:
    def __init__(self, lhs,rhs):
        self.data1=lhs
        self.data2=rhs

        dshape = topi.utils.get_const_tuple(self.data1.shape)
        self.data2_i = te.compute(dshape, lambda n, h, w, c: self.data2, "data2_i")
        self.data1_in = te.compute(dshape, lambda n, h, w, c: self.data1[n, h, w, c], "data1_in")
        self.data2_in = te.compute(dshape, lambda n, h, w, c: self.data2_i[n,h,w,c], "data2_in")

        self.cout = te.compute(
        dshape,
        lambda n, h, w, c: self.data1_in[n, h, w, c] >> self.data2_in[n, h, w, c],
        name="cout")

        self.out = te.compute(dshape, lambda n, h, w, c: self.cout[n, h, w, c], "out")
```

图 14　shift 算子实现（二）

1.6. Relu 算子实现

Relu 算子主要负责比较操作，即将左操作数中的每个元素与右操作数数值相比，保留比右操作数大的值，并将小于右操作数的元素变为与右操作数相同值。当执行该算子时，首先将右操作数通过 DMA 扩张到与左操作数相同规模，再搬移到 mid_buffer 中，同时通过 DMA 将右操作数也搬移到 mid_buffer 中，待搬移完成后，执行对应的 ve 操作，并将结果输出到 output_buffer 中，等待后续复用或者搬出到 dram。

```
class relu:
    def __init__(self, data):
        self.data1 = data
        self.data2 = tir.const(0)

        dshape = topi.utils.get_const_tuple(self.data1.shape)
        self.zeros = te.compute(dshape, lambda n, h, w, c: self.data2, "data2_in")
        self.data1_in = te.compute(dshape, lambda n, h, w, c: self.data1[n, h, w, c], "data1_in")
        self.data2_in = te.compute(dshape, lambda n, h, w, c: self.zeros[n, h, w, c], "data2_in")
        self.cout = te.compute(
            dshape,
            lambda n,h,w,c: tvm.te.max(self.data1_in[n,h,w,c], self.data2_in[n,h,w,c]),
            name="cout"
        )
        self.out = te.compute(dshape, lambda n, h, w, c: self.cout[n, h, w, c], "out")
```

图 15　relu 算子实现

1.7. Clip 算子实现

Clip 算子主要负责比较操作，与 relu 操作不同，clip 的操作包含两个步骤，即将左操作数中的每个元素与设定最小值相比，保留比设定最小值大的值，并将小于设定最小值的元素变为与设定最小值相同值；接下来再将第一步得到的结果作为输入，与设定最大值比较，并保留小于设定最大值的元素，并将大于设定最大值的元素替换为设定最大值。当执行该算子时，首先将最小值通过 DMA 扩张到与左操作数相同规模，再搬移到 mid_buffer 中，同时通过 DMA 将右操作数也搬移到 mid_buffer 中，待搬移完成后，执行对应的 ve 操作，并将结果输出到 output_buffer 中；接下来将最大值通过 DMA 扩张到与待操作数同规模的张量，并搬移到 mid_buffer 中，同时通过 DMA 将 output_buffer 中的值搬移到 mid_buffer 中，待搬移完成后，再进行 ve 操作，并将结果输出到 output_buffer 中，等待复用或搬出到 dram。

```
################### clip ########################

self.max_value = te.compute(dshape, lambda n, h, w, c: tvm.tir.const(127.), "max_value")
self.min_value = te.compute(dshape, lambda n, h, w, c: tvm.tir.const(-127.), "min_value")

self.max_value_buffer = te.compute(dshape, lambda n, h, w, c: self.max_value[n,h,w,c], "max_value_buffer")
self.min_value_buffer = te.compute(dshape, lambda n, h, w, c: self.min_value[n,h,w,c], "min_value_buffer")

self.clip_max_out = te.compute(
    dshape,
    lambda n, h, w, c: tvm.te.min(self.rshift_out_buffer[n,h,w,c],  self.max_value_buffer[n,h,w,c]),
    name="clip_max_out")

self.clip_middle_buffer = te.compute(dshape, lambda n, h, w, c: self.clip_max_out[n,h,w,c], "clip_middle_buffer")

self.clip_min_out = te.compute(
    dshape,
    lambda n, h, w, c: tvm.te.max(self.clip_middle_buffer[n,h,w,c],  self.min_value_buffer[n,h,w,c]),
    name="clip_min_out")

self.out = te.compute(dshape, lambda n, h, w, c: self.clip_min_out[n, h, w, c], name="out")
```

图 16　clip 算子实现

1.8. 算子融合实现

算子融合可以显著减少计算过程中中间结果读出和取入的次数，在对 resnet50 的整体结构进行分析后，对网络中的部分层进行融合。采用以下融合算法：

Algorithm 2 Operator Fusing

Input: The Workload Model
Output: *Schedule* - a schedule after deciding which whether feature map or weight to be reused during the computation

1: $m \leftarrow$ *the total number of operators in the model*
2: $FusedModel \leftarrow \emptyset$
3: for $i = 0, 1, 2$ to m do
4: if $operator[i]$ is not float computation then
5: if $operator[i]$ is Convolution operator then
6: $FusedOp \cup FusedModel$
7: $FusedOp \leftarrow \emptyset$
8: end if
9: $operator[i] \cup FusedOp$
10: else
11: $operator[i] \cup FusedModel$
12: end if
13: end for

图 17　融合算法

算法 2 的思想在于融合卷积算子以及在它之后的 element wise 的算子。由于卷积是一个 reduction 计算过程，不方便多个卷积相互融合。

在对融合后的网络进行分析后，可以得到主要有以下 4 个融合算子：

•Conv-lshift-add-relu-add-rshift-clip

•Conv-add-relu-add-rshift-clip

•Conv-lshift-add-add-rshift-clip-multi-add-add-rshift-clip

•Conv-add-rshift-clip

卷积算子映射到 gemm 单元上进行，其他的 element wise 的算子映射到 ve 单元上执行。后续会对这几个融合算子进行分块操作。

二、 Transform 实现

2.1. DMA 指令的插入与替换

插入 DMA 指令主要使用了 tvm 中的 pragma 函数和 pass 功能。为了能顺利生成 DMA 指令，我们需要找到结构树中对应的循环节点，然后获取地址，数据长度和偏置的参数，最后再实现指令的替换。其大致结构如图 18 所示。

图 18　DMA 映射流程

首先，利用 pragma 函数执行标记插入，即通过搜寻对应变量名，在对应的变量的循环根节点插入 DMA 的指令标记。这个插入指令的过程我们封装在 schedule.py 中，通过调用对应算子的 schedule 函数来在整个结构树中插入每个算子需要调用的 DMA 指令，以 conv2d 为例，其对应的调度如图 19 所示。

```python
def conv_schedule(s, conv):
    s[conv.feature_input].set_scope(env.input_scopeA)
    s[conv.weight_input].set_scope(env.input_scopeB)
    s[conv.feature].set_scope(env.inp_scope)
    s[conv.weight].set_scope(env.wgt_scope)
    s[conv.middle_buffer].set_scope(env.mid_scopeA)

    s[conv.feature_input].pragma(s[conv.feature_input].op.axis[0], env.dma_copy)
    s[conv.weight_input].pragma(s[conv.weight_input].op.axis[0], env.dma_copy)
    s[conv.feature].pragma(s[conv.feature].op.axis[2], env.dma_copy)
    s[conv.weight].pragma(s[conv.weight].op.axis[2], env.dma_copy)
    s[conv.middle].pragma(s[conv.middle].op.axis[2], env.dma_copy)

    i, x, j, y = s[conv.middle_buffer].op.axis
    s[conv.middle_buffer].reorder(i, x, conv.k0, conv.k1, j, y, conv.k2)

    s[conv.feature].compute_at(s[conv.middle_buffer], conv.k0)
    s[conv.weight].compute_at(s[conv.middle_buffer], conv.k0)
    s[conv.middle_buffer].compute_at(s[conv.middle], s[conv.middle].op.axis[1])

    s[conv.middle_buffer].tensorize(conv.k1, env.gemm)

    return s
```

图 19　插入 DMA 指令标记和 scope 分配

接下来再通过 pass 遍历 schedule 树节点执行替换，将对应指令标记下的循环结构替换为 DMA 指令。调用 pass 的选择在 build_module.py 中，通过自定义 pass list，可以选择性插入不同的 pass 并设定其执行等级，即只对一定等级以内的分支执行 pass，以此实现更深度的分类功能，具体调用如图 20 所示

```python
pass_list = [
    # (0, transform.InjectConv2DTransposeSkip()),
    (1, transform.InjectDMAIntrin()),
    # (1, transform.InjectSkipCopy()),
    #(1, transform.AnnotateALUCoProcScope()),
    # (1, tvm.tir.transform.LiftAttrScope("coproc_uop_scope")),
    # (1, transform.LiftAllocToScopeBegin()),
    # (1, tvm.tir.transform.LiftAttrScope("coproc_scope")),
    # (1, transform.InjectCoProcSync()),
    # (1, EarlyRewrite()),
]

# if debug_flag:
#     pass_list.append((1, add_debug))
pass_list.append((2, transform.InjectALUIntrin()))
# pass_list.append((3, tvm.tir.transform.LowerDeviceStorageAccessInfo()))
# pass_list.append((3, transform.FoldUopLoop()))
# pass_list.append((3, transform.CPUAccessRewrite()))
config = {"tir.add_lower_pass": pass_list}
if kwargs.get("config"):
    config.update(kwargs[config])
    del kwargs["config"]

return tvm.transform.PassContext(config=config, **kwargs)
```

图 20　pass 设置

为了区分不同 buffer 之间的 DMA 指令，我们定义了不同的 scope，每个 scope 对应相应的 buffer，我们定义了 input_scope、feature_scope、weight_scope、middle_scope 和 output_scope，并区分了 pingpong buffer，通过确定执行 DMA 操作的源 buffer 和目的 buffer 的 scope，即可确定我们需要生成的指令中的源地址和目的地址。DMA 指令的具体实现在 transform 中，我们首先通过确定 src.scope 和 dst.scope 来进行分支选择，从 environment.py 中载入 scope 对应的参数，即对应地址和 buffer 的数据类型等，其实现如图 21 所示。

```
if dst.scope == "global":
    # Store
    if pad_before or pad_after:
        raise RuntimeError("Do not support copy into DRAM with pad")
    if src.scope == env.mid_scopeA:
        elem_width = env.ACC_WIDTH
        elem_bytes = env.ACC_ELEM_BYTES*8
        mem_type = env.dev.MEM_ID_OUT
        data_type = "int%d" % env.ACC_WIDTH
        task_qid = env.dev.QID_STORE_OUT

    elif src.scope == env.mid_scopeB:
        elem_width = env.ACC_WIDTH
        elem_bytes = env.ACC_ELEM_BYTES*8
        mem_type = env.dev.MEM_ID_OUT
        data_type = "int%d" % env.ACC_WIDTH
        task_qid = env.dev.QID_STORE_OUT
    elif src.scope == env.inp_scope:
        elem_width = env.INP_WIDTH
        elem_bytes = env.INP_ELEM_BYTES
        mem_type = env.dev.MEM_ID_INP
        data_type = "int%d" % env.INP_WIDTH
        task_qid = env.dev.QID_LOAD_INP
    elif src.scope == env.wgt_scope:
        elem_width = env.WGT_WIDTH
        elem_bytes = env.WGT_ELEM_BYTES
        mem_type = env.dev.MEM_ID_WGT
        data_type = "int%d" % env.WGT_WIDTH
        task_qid = env.dev.QID_LOAD_WGT
    elif src.scope == env.input_scopeA:
        elem_width = env.INP_WIDTH
        elem_bytes = env.INPUT_ELEM_BYTES
```

图 21　获取 scope 参数

确定了源地址和目的地址之后，我们通过读取替换掉的 schedule 树中对应的循环结构，获取对应的每层循环的 range，再配合上读取 scope 对应的 dtype，我们就可以确定整个 DMA 操作的 length,这样单次 dma 操作就基本实现了。在具体实现中，我们通过调取一个 fold 函数来计算总的数据个数，同时通过输入数据的 dtype 来换算总的数据需要的 bit 数，最后再换算成指令对应的 length，其实现过程如图 22

```
shape = list(x for x in shape)
base = buf.shape[-1]
strides = list(x for x in strides)
def raise_error():
    """Internal function to raise error """
    raise RuntimeError(
        (
            "Scope[%s]: cannot detect 2d pattern with elem_block=%d:"
            + " shape=%s, strides=%s"
        )
        % (scope, 0, buf.shape, buf.strides)
    )

ndim = len(shape)

# Check if the inner-tensor is already flat
flat = utils.equal_const_int(shape[-1], shape[-1])

if flat:
    if not utils.equal_const_int(strides[-1], 1):
        raise_error()

    if ndim == 1:
        x_size0 = 0
        x_size1 = 0
        x_size2 = 0
        x_size3 = 0
        y_size= 1
        return x_size0, x_size1,x_size2,x_size3, y_size,base
    # if not utils.equal_const_int(strides[-2] - elem_block, 0):
    #     raise_error()

    if ndim == 2:
        x_size0 = shape[-2]
        x_size1 = 0
```

图 22　计算 DMA 指令 length

当数据规模增大时，feature buffer 和 weight buffer 不足以存储下所有数据（即达到了 gemm 操作的最大规模），这时我们需要通过外层循环来实现多次 DMA 操作，以匹配 gemm 运算，也就需要给 DMA 操作的源地址和目的地址加上适当的偏置。为了配合 tvm 的 c codegen 的功能，我们选择使用 tvm 内部的偏置函数获取对应循环的偏置值，修正后附加于指令的的源地址和目的地址，以实现 DMA 指令的外部循环，如图 23 所示。

```
    irb = tvm.tir.ir_builder.create()
    irb.scope_attr(env.dev.zte_axis, "coproc_scope", env.dev.get_task_qid(task_qid))
    if src.scope == env.output_scopeA:
        irb.emit(
            tvm.tir.call_extern(
                "int64",
                "store",
                # tvm.tir.const(dst.handle.value, "int64"),
                # tvm.tir.Var(dst.handle.value, "int64"),
                dst.data,
                # src.data,
                tvm.tir.const(0x20080000, "int64"),
                base * y_size//base0,
                4*dst.elem_offset
            )
        )
    elif src.scope == env.output_scopeB:
        irb.emit(
            tvm.tir.call_extern(
                "int64",
                "store",
                # tvm.tir.const(dst.handle.value, "int64"),
                # tvm.tir.Var(dst.handle.value, "int64"),
                dst.data,
                # src.data,
                tvm.tir.const(0x20090000, "int64"),
                base * y_size // base0,
                4*dst.elem_offset
            )
        )
```

图 23　输入参数和附加偏置

当这一系列流程都完成后，DMA 指令的替换就完成了，以一个融合算子为例，最终实现的效果如下图所示，这样就完成了算子中的数据搬移指令替换过程。

```
for (int32_t nó = 0; nó < 32; ++nó) {
  for (int32_t hó = 0; hó < 2; ++hó) {
    (void)gemm((int64_t)537001984, (int64_t)537133056, (int64_t)537264128, (int64_t)1, (int64_t)1, (int64_t)1, (int64_t)1);
    (void)load(mdata, (int64_t)536870912, 64, (nó * 512));
    (void)move((((int64_t)0 * (int64_t)4) + (int64_t)536870912), (((int64_t)0 * (int64_t)2) + (int64_t)537001984), 64, 0);
    (void)load(T_reshape1, (int64_t)536936448, 64, (hó * 512));
    (void)move((((int64_t)0 * (int64_t)4) + (int64_t)536936448), (((int64_t)0 * (int64_t)16) + (int64_t)537133056), 64, 0);
    (void)gemm((int64_t)537001984, (int64_t)537133056, (int64_t)537264128, (int64_t)1, 8, (int64_t)1, (int64_t)0);
    (void)move((int64_t)537264128, (((int64_t)0 * (int64_t)8) + (int64_t)537329664), 32, 0);
    (void)load(pad_data, (int64_t)537264128, 32, ((nó * 128) + (hó * 64)));
    (void)ve(1, (int64_t)537264128, (int64_t)537329664, (int64_t)537395200, 2, (int64_t)0);
    (void)move((int64_t)537395200, (int64_t)537329664, 128, 0);
    (void)load(zeros, (int64_t)537264128, 32, ((nó * 128) + (hó * 64)));
    (void)ve(5, (int64_t)537264128, (int64_t)537329664, (int64_t)537395200, 2, (int64_t)0);
    (void)move((int64_t)537395200, (int64_t)537329664, 128, 0);
    (void)load(add_num_4d, (int64_t)537264128, 32, ((nó * 128) + (hó * 64)));
    (void)ve(1, (int64_t)537264128, (int64_t)537329664, (int64_t)537395200, 2, (int64_t)0);
    (void)move((int64_t)537395200, (int64_t)537264128, 128, 0);
    (void)load(data2_in, (int64_t)537329664, 32, ((nó * 128) + (hó * 64)));
    (void)ve(9, (int64_t)537264128, (int64_t)537329664, (int64_t)537395200, 2, (int64_t)0);
    (void)move((int64_t)537395200, (int64_t)537329664, 128, 0);
    (void)load(max_value, (int64_t)537264128, 32, ((nó * 128) + (hó * 64)));
    (void)ve(6, (int64_t)537264128, (int64_t)537329664, (int64_t)537395200, 2, (int64_t)0);
    (void)move((int64_t)537395200, (int64_t)537329664, 128, 0);
    (void)load(min_value, (int64_t)537264128, 32, ((nó * 128) + (hó * 64)));
    (void)ve(5, (int64_t)537264128, (int64_t)537329664, (int64_t)537395200, 2, (int64_t)0);
    (void)store(out_buffer, (int64_t)537395200, 32, ((nó * 512) + (hó * 256)));
```

图 24　替换后的指令（局部）

当然，为了能将指令转换为 rocc 指令，还需要在编译过程中加入对应指令的 rocc 指令映射文件，以实现最终到后端指令的映射。

```
void move(uint64_t src,uint64_t des,uint64_t length,uint64_t mask2){
  uint64_t rs1 = (des << 30) + src;
  uint64_t rs2 = (mask2 << 16) + length;
  // printf("src    : 0x%" PRIx64 "\n", src);
  // printf("des    : 0x%" PRIx64 "\n", des);
  // printf("length : 0x%" PRIx64 "\n", length);
  //printf("rs2 : 0x%" PRIx64 "\n", rs2);
  // printf("\n");
  ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, 0x78);
}

void load(void* mem_addr,uint64_t rocc_addr,uint64_t length,uint64_t bias){
  uint64_t mask_load_ex  = mask2(1,0,1,1,0,0);
  uint64_t mem_addr_int=(uint64_t)mem_addr+bias;
  write_reg(0,mem_addr_int);
  asm volatile("fence");
  move(0,rocc_addr,length,mask_load_ex);
  asm volatile("fence");
}

void store(void* mem_addr,uint64_t rocc_addr,uint64_t length,uint64_t bias){
  uint64_t mask_store_ex = mask2(1,0,0,1,0,1);
  uint64_t mem_addr_int=(uint64_t)mem_addr+bias;
  write_reg(1,mem_addr_int);
  asm volatile("fence");
  move(rocc_addr,0,length,mask_store_ex);
  asm volatile("fence");
}
```

图 25　rocc 指令映射

```
#define ROCC_INSTRUCTION_R_R_R(X, rd, rs1, rs2, funct)                  \
    do {                                                                \
        __asm__ __volatile__ (                                          \
            ".insn r CUSTOM_" #X ", %3, %4, %0, %1, %2\n\t"             \
            : "=r" (rd)                                                 \
            : "r" (rs1), "r" (rs2),                                     \
              "i" (ROCC_XD | ROCC_XS1 | ROCC_XS2), "i" (funct));        \
    } while (0)

#define ROCC_INSTRUCTION_R_R_I(X, rd, rs1, rs2, funct)                  \
    do {                                                                \
        __asm__ __volatile__ (                                          \
            ".insn r CUSTOM_" #X ", %3, %4, %0, %1, x%2\n\t"            \
            : "=r" (rd)                                                 \
            : "r" (rs1), "K" (rs2),                                     \
              "i" (ROCC_XD | ROCC_XS1), "i" (funct));                   \
    } while (0)

#define ROCC_INSTRUCTION_R_I_I(X, rd, rs1, rs2, funct)                  \
    do {                                                                \
        __asm__ __volatile__ (                                          \
            ".insn r CUSTOM_" #X ", %3, %4, %0, x%1, x%2\n\t"           \
            : "=r" (rd)                                                 \
            : "K" (rs1), "K" (rs2),                                     \
              "i" (ROCC_XD), "i" (funct));                              \
    } while (0)

#define ROCC_INSTRUCTION_I_R_R(X, rd, rs1, rs2, funct)                  \
    do {                                                                \
        __asm__ __volatile__ (                                          \
            ".insn r CUSTOM_" #X ", %3, %4, x%0, %1, %2\n\t"            \
            :                                                           \
            : "K" (rd), "r" (rs1), "r" (rs2),                           \
              "i" (ROCC_XS1 | ROCC_XS2), "i" (funct));                  \
```

图 26　rocc 指令映射（二）

2.2. Ve 指令的插入与替换

Ve 指令的插入与替换过程与 DMA 指令类似，同样是通过 pass 遍历整个结构树来实现指令的插入和替换。首先，需要利用 pragma 指令在结构树中搜索对应变量并插入 ALU 标记，这个插入指令也封装在 schedule.py 中，见图*。同时，我们在 build_module 中的 pass list 中添加 ALU 插入的 pass，这样当进行 tvm.lower 时，就会在结构树中需要替换 alu 指令的位置插入标记。

当 lower 插入标记完成后，我们就需要进行 build 来替换指令，ALU 指令替换的具体实现过程也封装在 transform 中。替换中我们首先需要定位到插入标记的位置，接下来就通过逐层向里读取循环来获取赋值操作和运算操作符，并记录每层循环的 range，如图 27 所示

```
if _match_pragma(stmt, "alu"):
    # Get to the innermost loop body
    loop_body = stmt.body
    nest_size = 0
    while isinstance(loop_body, tvm.tir.For):
        print("nest_size",nest_size)
        print("loop_body",loop_body)
        if isinstance(loop_body.body, tvm.tir.stmt.SeqStmt):
            # print("loop_body.body.value",loop_body.body.value)
            loop_body = loop_body.body.seq[1]
            nest_size += 1
        else:
            loop_body = loop_body.body
            nest_size += 1
```

图 27　读取循环内部的运算

当读取到循环最里层之后，就先读取运算符来判断对应的运算指令，判断通过分支指令进行，并将运算的左右操作数保存起来，等待进行运算。其执行代码如图 28 所示。

```
if isinstance(loop_body.value, tvm.tir.Add):
    alu_opcode = 1
    lhs = loop_body.value.a
    rhs = loop_body.value.b
elif isinstance(loop_body.value, tvm.tir.Sub):
    alu_opcode = 2
    lhs = loop_body.value.a
    rhs = loop_body.value.b
elif isinstance(loop_body.value, tvm.tir.Mul):
    alu_opcode = 3
    lhs = loop_body.value.a
    rhs = loop_body.value.b
elif isinstance(loop_body.value, tvm.tir.Min):
    alu_opcode = 6
    lhs = loop_body.value.a
    rhs = loop_body.value.b
elif isinstance(loop_body.value, tvm.tir.Max):
    alu_opcode = 5
    lhs = loop_body.value.a
    rhs = loop_body.value.b
elif isinstance(loop_body.value, tvm.tir.And):
    alu_opcode = 7
    lhs = loop_body.value.a
    rhs = loop_body.value.b
elif isinstance(loop_body.value, tvm.tir.Or):
    alu_opcode = 8
    lhs = loop_body.value.a
    rhs = loop_body.value.b
elif isinstance(loop_body.value, tvm.tir.Call):
    if loop_body.value.op.name == "tir.shift_left":
        alu_opcode = 10
        lhs = loop_body.value.args[0]
        rhs = analyzer.simplify(-loop_body.value.args[1])
    elif loop_body.value.op.name == "tir.shift_right":
```

图 28　判断 ALU 操作类型

当获取了操作对应的序号后，就通过之前读取的 range 来计算 ve 操作的 length，并判断需要输出的数据格式，如果需要将数据转换为 int8，则额外记录标志位，否则直接执行普通的 ve 操作

```
    extent = reduce(lambda x,y,:x*y,,extents)
    irb = tvm.tir.ir_builder.create()
    # for idx, extent in enumerate(extents):
        # irb.emit(
        #     tvm.tir.call_extern(
        #         "int32", "zteUopLoopBegin", extent, dst_coeff[idx], src_coeff[idx], 0
        #     )
        # )
    # use_imm = int(use_imm)
    irb.emit(
        tvm.tir.call_extern(
            "int64",
            "ve",
            alu_opcode,
            tvm.tir.const(0x20060000, "int64"),
            tvm.tir.const(0x20070000, "int64"),
            tvm.tir.const(0x20080000, "int64"),

            extent//32,
            tvm.tir.const(0, "int64")
        )
    )
    # for extent in extents:
    #     irb.emit(tvm.tir.call_extern("int32", "zteUopLoopEnd"))
    return irb.get()
return stmt
```

图 29　VE 指令替换

经过替换后的 ve 指令将在编译后转换为 rocc 指令，其方法与 DMA 指令类似，其对应的 rocc 指令编译过程如图 30 和图 31 所示。

```
void ve(int funct,uint64_t rm,uint64_t rn,uint64_t rd,uint64_t length,uint64_t mask1){
  rm = rm & 0xfffff;
  rn = rn & 0xfffff;
  rd = rd & 0xfffff;
  uint64_t rs1 = (rd << 40) + (rm << 20) + rn;
  uint64_t rs2 = (mask1 << 16) + length;
  switch(funct){
    case ADD:ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, ADD << 2);break;
    case SUB:ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, SUB << 2);break;
    case MUL:ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, MUL << 2);break;
    case MAX:ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, MAX << 2);break;
    case MIN:ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, MIN << 2);break;
    case AND:ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, AND << 2);break;
    case OR :ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, OR  << 2);break;
    case SR :ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, SR  << 2);break;
    case SL :ROCC_INSTRUCTION_I_R_R(3, 0, rs1, rs2, SL  << 2);break;
  }

}
```

图 30　ve 指令的 rocc 指令映射

```
#define ROCC_INSTRUCTION_R_I_I(X, rd, rs1, rs2, funct)                  \
    do {                                                                \
        __asm__ __volatile__ (                                          \
            ".insn r CUSTOM_" #X ", %3, %4, %0, x%1, x%2\n\t"           \
            : "=r" (rd)                                                 \
            : "K" (rs1), "K" (rs2),                                     \
              "i" (ROCC_XD), "i" (funct));                              \
    } while (0)

#define ROCC_INSTRUCTION_I_R_R(X, rd, rs1, rs2, funct)                  \
    do {                                                                \
        __asm__ __volatile__ (                                          \
            ".insn r CUSTOM_" #X ", %3, %4, x%0, %1, %2\n\t"            \
            :                                                           \
            : "K" (rd), "r" (rs1), "r" (rs2),                           \
              "i" (ROCC_XS1 | ROCC_XS2), "i" (funct));                  \
    } while (0)

#define ROCC_INSTRUCTION_I_R_I(X, rd, rs1, rs2, funct)                  \
    do {                                                                \
        __asm__ __volatile__ (                                          \
            ".insn r CUSTOM_" #X ", %3, %4, x%0, %1, x%2\n\t"           \
            :                                                           \
            : "K" (rd), "r" (rs1), "K" (rs2),                           \
              "i" (ROCC_XS1), "i" (funct));                             \
    } while (0)

#define ROCC_INSTRUCTION_I_I_I(X, rd, rs1, rs2, funct)                  \
    do {                                                                \
        __asm__ __volatile__ (                                          \
            ".insn r CUSTOM_" #X ", %3, %4, x%0, x%1, x%2\n\t"          \
            :                                                           \
            : "K" (rd), "K" (rs1), "K" (rs2),                           \
              "i" (0), "i" (funct));                                    \
    } while (0)
```

图 31　ve 指令的 rocc 指令映射（二）

三、 Tensorize 实现

TVM 中对于 gemm 指令的插入是通过 tensorize API，tensor intrinsic 提供了一个可以替换的模板，需要定义好 tensorize 的输入的 shape，计算过程，以及 tensor 关联的 scope。

Tensor intrinsic 还支持宏指令的定义，可以设置 tensor shape 为变量。在主程序中调用 tensorize 的时候，会自动获取需要替换的 tensor 的 shape，并可以在生成的 C 中输出出来。。

```
wgt = te.placeholder(
    (wgt_shape[0], wgt_shape[1], wgt_shape[2]), dtype="int%d" % env.WGT_WIDTH, name=env.wgt_scope
)
inp = te.placeholder(
    (inp_shape[0], inp_shape[1], inp_shape[2]), dtype="int%d" % env.INP_WIDTH, name=env.inp_scope
)
k1 = te.reduce_axis((0, wgt_shape[0]), name="k1")
k2 = te.reduce_axis((0, wgt_shape[2]), name="k2")
out_dtype = "int%d" % env.ACC_WIDTH
out = te.compute(
    (out_shape[0], out_shape[1]),
    lambda i, j: te.sum(
        inp[k1, i, k2].astype(out_dtype) * wgt[k1, j, k2].astype(out_dtype),
        axis=[k1, k2]),
    name="out",
)
```

图 32 定义计算

```
wgt_layout = tvm.tir.decl_buffer(
    wgt.shape,
    wgt.dtype,
    env.wgt_scope,
    scope=env.wgt_scope,
    offset_factor=wgt_lanes,
    data_alignment=wgt_lanes,
    # strides=[strides_wgt, 1]
)

strides_inp = tvm.tir.Var("s_inp", "int32");
inp_layout = tvm.tir.decl_buffer(
    inp.shape,
    inp.dtype,
    env.inp_scope,
    scope=env.inp_scope,
    offset_factor=inp_lanes,
    data_alignment=inp_lanes,
    # strides=[strides_inp, 1, 1, 1]
    # strides=[strides_inp, strides_inp, 8, 8]
)

strides_out = tvm.tir.Var("s_out", "int32");
out_layout = tvm.tir.decl_buffer(
    out.shape,
    out.dtype,
    env.mid_scopeA,
    scope=env.mid_scopeA,
    offset_factor=out_lanes,
    data_alignment=out_lanes,
    # strides=[strides_out, 1]
)
```

图 33 定义 tensor 的 scope

```python
def _body():
    irb = tvm.tir.ir_builder.create()
    dev = env.dev
    irb.scope_attr(dev.zte_axis, "zte_uop_scope", tvm.tir.StringImm("zte_gemm"))
    irb.emit(
        tvm.tir.call_intrin(
            "int32",
            "tir.zte_gemm_fp16",
            tvm.tir.const(0x20020000, "int64"),
            tvm.tir.const(0x20040000, "int64"),
            tvm.tir.const(0x20060000, "int64"),
            # m,
            tvm.tir.const(1, "int64"),
            k,
            tvm.tir.const(1, "int64"),
            # n,
            tvm.tir.const(0, "int64"),
        )
    )
    return irb.get()
```

图 34 定义待替换的指令

```
(void)gemm((int64_t)537001984, (int64_t)537133056, (int64_t)537264128, (int64_t)1, 72, (int64_t)1, (int64_t)0);
```

图 35 生成的 C 代码

四、 C 代码编程

TVM 的程序通过 tvm.build 会生成 IRmodule，然后用 save 这个 tvm 的 API，可以把 C 代码保存下来，保存下来的代码是一个函数。如果要执行的话，需要给这个函数写一个 main 函数来调用它，获取输入得到输出结果。

```
int main()
{
  printf("1.define\n");
    int num_args = 4;
  int cn = 128;
  int f_shape[4]={1, 56, 56, 64};
  int p9_shape[4]={64,3,3,64};
  int p10_shape[3]={64, 1, 1};
  // int p4_shape[3]={64, 1, 1};
  // int w_shape[4]={cn, 3, 3, cn};
  int o_shape[4]={1, 56, 56, 64};
  int f_length = f_shape[0]*f_shape[1]*f_shape[2]*f_shape[3];
  int p9_length = p9_shape[0]*p9_shape[1]*p9_shape[2]*p9_shape[3];
  int p10_length = p10_shape[0]*p10_shape[1]*p10_shape[2];
  // int p4_length = p4_shape[0]*p4_shape[1]*p4_shape[2];
  int o_length = o_shape[0]*o_shape[1]*o_shape[2]*o_shape[3];

    TVMValue * args = new TVMValue[num_args];

    int32_t * arg_type_ids = new int32_t[num_args];

    DLTensor * arg0 = new DLTensor;
    DLTensor * arg1 = new DLTensor;
    DLTensor * arg2 = new DLTensor;
    DLTensor * arg3 = new DLTensor;
    DLTensor * arg4 = new DLTensor;


    args[0].v_handle = arg0;
    args[1].v_handle = arg1;
    args[2].v_handle = arg2;
    args[3].v_handle = arg3;
    // args[4].v_handle = arg4;


    arg_type_ids[0] = kTVMDLTensorHandle;
    arg_type_ids[1] = kTVMDLTensorHandle;
```

图 36 main 函数（局部）

五、 总结

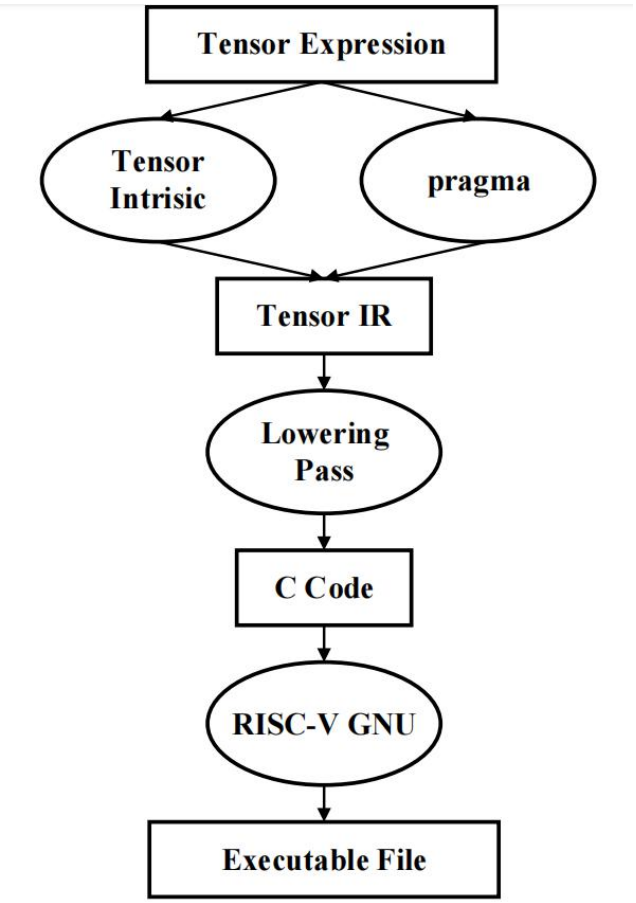

图 37 编译流程

我们实现 resnet50 在加速器上的运行主要包含以下几个编译阶段：

首先用 tvm 的程序写出 resnet50 的计算过程

定义 resnet50 的调度过程，指定 tensor 的 scope，并且插入 dma、alu pragma，使用 tensor instrinsic 标出替换 SIMD 指令的位置

通过在 tvm.build 的过程中遍历的 lowering pass，将循环替换成对应的 SIMD 指令，并保存生成的 C 代码

为 C 代码写 main 函数，输入 image 和参数，获得结果

用 RISC-V 的交叉编译器编译程序，将可执行文件拷贝到板子上执行。