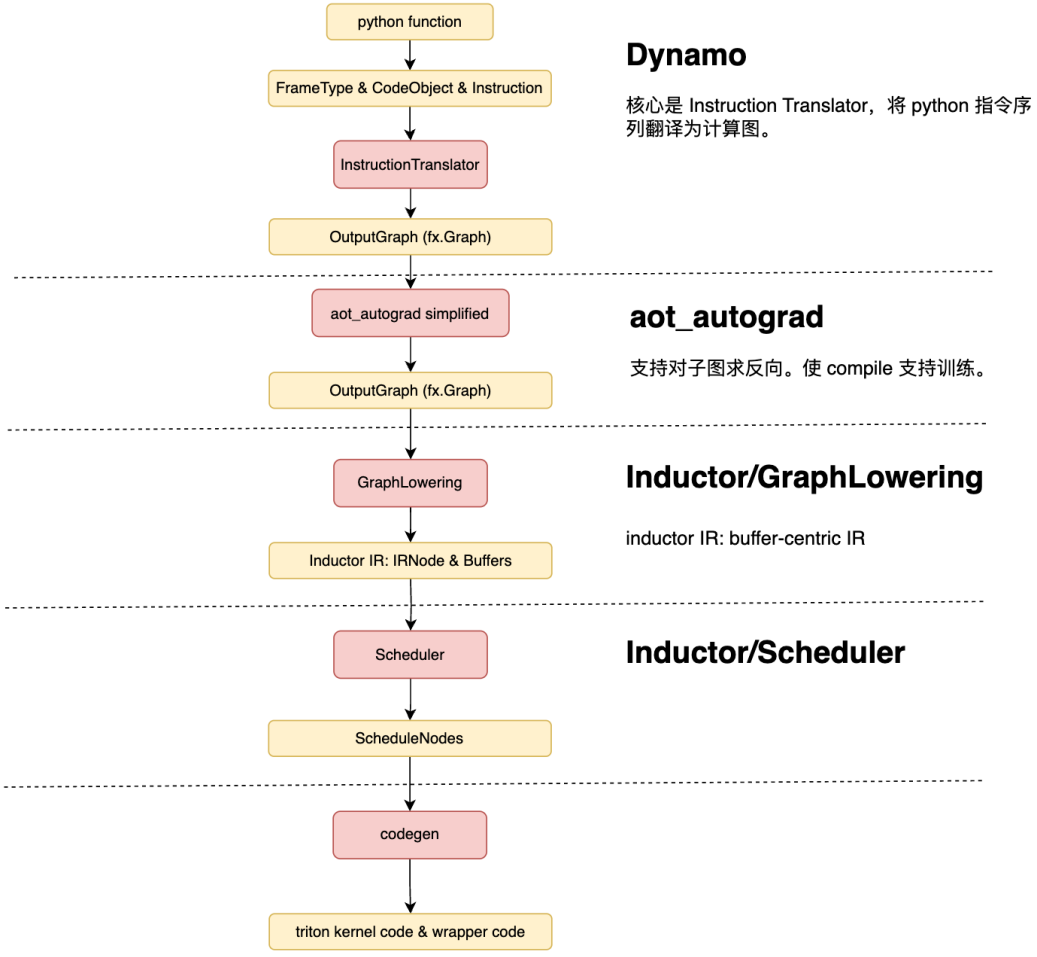
# Inductor

## Overview

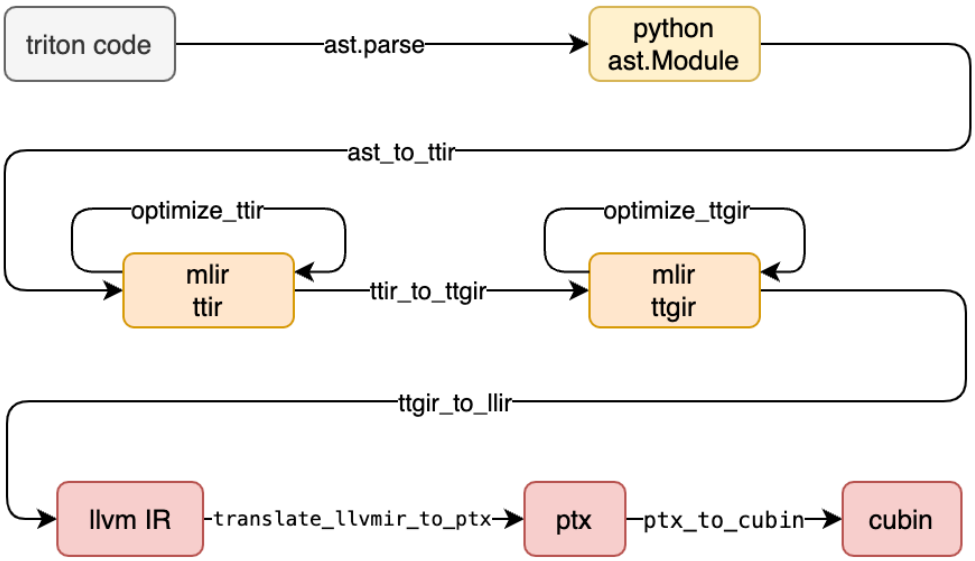
从 dynamo 编译过程中调用 call_user_compiler 函数开始，backend 接过了编译的工作。在 pytorch 的 in-tree 编译器中，目前 inductor 是默认的一个。

User compiler 的责任是将一个 fx.GraphModule 编译成一个 CompiledFn.

Stack of Dynamo + aot_autograd + inductor




Stack of triton compilation




## Aot_AutoGrad

在 dynamo 的文档里，我们提到过，inductor 外面还包着一层 aot_autograd.

Torch 的文档中也有一个教程，讲的是如何制作用于 torch.compile 的自定义后端。其中讲到两种方式集成。

1. 一种是直接集成到 dynamo 中，作为 dynamo 的后端；
2. 另一种是在外面包一层 aot_autograd 然后，作为 dynamo 后端。这样就可以直接 training.

因为在 call_user_compiler 中，此时 fx.Graph 中的算子仍然是五花八门的。为了支持训练，还需要 aot_autograd 的帮助，因为 aot_autograd 可以根据 forward 计算图 derive 出反向计算图。在传统的 pytorch eager 执行模式中，反向计算图的构建是随着 aten 函数的执行构建的（每个算子都有自己的对应的反向计算图的构建，作为非 no_grad 模式之行下的副作用）。但是通过编译执行之后，这些 aten 函数就不被执行了，那么就需要一个机制，以更大的单位（非 op，而是 graph, subgraph) 为单位构造反向计算图。这就是 aot_autograph 的作用。其实这个功能是自动微分层面的功能，和编译并没有什么关系。

aot_autograd 在其中发挥的作用是为子图生成反向计算图。至于计算图被简化到仅包含一系列的简单的算子，这只是这个过程的副作用，并且不是必须的。只是编译器也喜欢简化的算子集合。“事情就这么成了”。

经过 aot_autograd 处理的 fx.GraphModule 中的 fx.Graph 中的函数调用只包含 aten core 中的 Op 和 prim 中的 Op 了。

而官方的 inductor 其实也走的是类似其他 CustomBackend 的方式。它在设计的时候就集成了 aot_autograd.

https://pytorch.org/docs/stable/dynamo/custom-backends.html#custom-backends-after-aotautograd

P.S.: CompiledFn 只是一种 Protocol 或者 Concept, 只要能接受一个解包的 tuple of torch.Tensors, 返回一个 tuple of torch.Tensors 的 Callable，都算是 CompiledFn. Torch.nn.Module 或者 fx.GraphModule 也都符合这个 Concept.

# Inductor Entry Point

## compile_fx

而 inductor 提供的最顶层的对外的功能是 compile_fx.

_inductor/compile_fx.py

```
def compile_fx(
    model_: torch.fx.GraphModule,
    example_inputs_: List[torch.Tensor],
    inner_compile=compile_fx_inner,
    config_patches: Optional[Dict[str, Any]] = None,
    decompositions: Optional[Dict[OpOverload, Callable]] = None,
):
```

它在设计的时候，就体现了这些。decompositions 是提供给 aot_grad 的 op 分解规则。真正的 inductor 的核心其实是 inner_compile 参数代表的函数。其默认值就是 compile_fx_inner.

## compile_fx_inner

```
def compile_fx_inner(
    gm: torch.fx.GraphModule,
    example_inputs: List[torch.Tensor],
    cudagraphs: Optional[BoxedBool] = None,
    num_fixed=0,
    is_backward=False,
    graph_id=None,
    cpp_wrapper=False,
    aot_mode=False,
    is_inference=False,
    boxed_forward_device_index=None,
    user_visible_outputs=frozenset(),
    layout_opt=None,
):
```

这才是 inductor 的本体。

其中最主要的历程是

```
    compiled_graph: CompiledFxGraph = fx_codegen_and_compile(
        *graph_args, **graph_kwargs
    )
```

其中的过程在它的名字中是显然的 fx codegen 和 compile.

## fx_codegen_and_compile

这里事实上还分几个子过程。

1. fx.Graph 转化为 Inductor IR. 亦即 Inductor IR 的构建。fx.Graph 只是 Inductor 的一种前端语言，如果不介意，其实可以直接用 Inductor 的 API 构造 IR。（目前整个系统的耦合太紧密了，不容易单个 play with）
2. Inductor IR 的优化。

3. Inductor IR 的代码生成。这里主要有 Scheduler 相关，最后才是文本生成。

下面分别讲述这几个过程。

# Inductor IR Construction

这个核心结构 GraphLowering. Inductor IR, IRNode, Buffer 等。还有一部分关于 FakeTensor 的部分，因为 torch 的历史原因，有 fake, real, symbolic 等几种 trace 模式。暂且不用管，只需要知道在 inductor 编译的这个背景下，使用的是 fake 模式即可。

核心部分的代码是

```
    with V.set_fake_mode(fake_mode):
        graph = GraphLowering(
            gm,
            shape_env=shape_env,
            num_static_inputs=num_fixed,
            graph_id=graph_id,
            cpp_wrapper=cpp_wrapper,
            aot_mode=aot_mode,
            user_visible_outputs=user_visible_outputs,
        )
        with V.set_graph_handler(graph):
            graph.run(*example_inputs)
```

1. 首先根据输入的 fx.GraphModule 构建一个 GraphLowering 对象，它继承了 fx.Interperter. 亦即，它是一种图解释器。它解释执行的副作用就是构造 Inductor IR，只要设置好一些 context 或者 environment, 就可以用直接执行 fx.Node，把副作用输出到 Inductor IR 上而不需要额外的参数。
2. GraphLowering.run(*example_inputs) 执行计算图。这类似 tree-walk interpreter 的执行。

# Interpreter

GraphLowering 继承了 torch.fx.Interpreter 因此它可以执行 fx 计算图。亦即通过 run 方法来执行计算图。而 GraphLowering 执行的是 lowering，而不是我们一般理解的 evaluation.

```
class Interpreter:
    """
    An Interpreter executes an FX graph Node-by-Node. This pattern
    can be useful for many things, including writing code
    transformations as well as analysis passes.

    Methods in the Interpreter class can be overridden to customize
    the behavior of execution. The map of overrideable methods
    in terms of call hierarchy::

        run()
            +-- run_node
                +-- placeholder()
                +-- get_attr()
                +-- call_function()
                +-- call_method()
                +-- call_module()
                +-- output()
```

run 方法会逐个执行节点。而 fx.Node 上确实记录了需要执行的操作。而且由于 fx.Node 的 op 很简单，只分为几个大类。

1. placeholder/output 输入和输出节点；
2. call_function/call_method/call_module 都是对函数，nn 模块的方法，nn 模块的调用。这里的空间就很大，具体就看执行的是什么函数了；
3. get_attr 是成员访问，实际使用比较少，一般会在对返回值 tuple 的成员访问。

fx.Interpreter 基本就是一个原原本本的执行器。

```
    @compatibility(is_backward_compatible=True)
    def call_function(self, target : 'Target', args : Tuple[Argument, ...],
kwargs : Dict[str, Any]) -> Any:
        """
        Execute a ``call_function`` node and return the result.

        Args:
            target (Target): The call target for this node. See
                `Node
<https://pytorch.org/docs/master/fx.html#torch.fx.Node>`__ for
                details on semantics
            args (Tuple): Tuple of positional args for this invocation
            kwargs (Dict): Dict of keyword arguments for this invocation

        Return
            Any: The value returned by the function invocation
        """
        assert not isinstance(target, str)
```

而 GraphLowering 则不同。

## GraphLowering

GraphLowering 的 run node 方法最主要的目的是构建 inductor IR. 主要做的事是处理输入，输出和计算。GraphLowering run 运行的过程中需要 maintain 的信息有

1. 图的输入输出，graph_inputs(str->TensorBox), graph_inputs_original(str->StorageBuffer), graph_outputs
2. 分析对图输入的 mutation 的分析， mutated_input, mutated_input_ids.
3. 所需的 buffer 的分析。buffers, removed_buffers, removed_inplace_buffer, mutated_buffer, inplaced_to_remove
4. 所需的常量。constants.
5. 符号表和 UseDef 关系。name_to_buffer, name_to_user

并非 fx.Graph 上有多少个变量就需要多少个 buffer. 有些运算可以融合在一起就省掉了 buffer. 只有不得不使用一个在 kernel launch 之间的必须持续存在的 tensor，或者是输如输出的时候，才会需要 buffer.

GraphLowering.run 对于不同的 fx.Node 的处理方式是：

## Inputs

首先会调用 fx.Interpreter 以执行模式一次。（注意：输入都是 fake 的）

1. 如果是 plain python 数值类型，用 sympy 包装好，然后加入 GraphLowering.graph_inputs.
2. 如果是 sympy Symbol 类型，直接装入 graph_inputs,
3. 如果是 torch.Tensor，尽可能计算 shape 和 stride. 然后构造 TensorBox, 装入 graph_inputs (str→ TensorBox). 实际的结构是 TensorBox 里有一个 StorageBox, Storage 里有一个 InputBuffer. 这是层包装的关系。

```python
def placeholder(self, target: str, args, kwargs):
    example = super().placeholder(target, args, kwargs)
    if isinstance(example, SymTypes):…
    elif isinstance(example, (int, bool, float)):…
    assert isinstance(example, torch.Tensor), example
    # todo(chilli): We can remove the last check once we turn buffers into
    # static shape tensors. That's a hack to workaround Inductor believing
    # the buffer should be static but us passing in a fake tensor with
    # symbolic shapes.
    if not example._has_symbolic_sizes_strides:…
    else:…
    # TODO(jansel): handle input aliasing
    tensor = TensorBox.create(
        InputBuffer(
            target,
            FixedLayout(example.device, example.dtype, sizes, strides),
        )
    )
    self.graph_inputs[target] = tensor
    self.graph_inputs_original[target] = tensor.data.data
    self.device_types.add(example.device.type)
    self.add_device_idx(example.device.index)
    return tensor
```

## FunctionCall

nn.ModuleCall, Tensor.MethodCall 到了这里都不复存在。

对于函数调用。对于不同的函数调用做了很多 Ifelse 处理，

1. 某些函数是用 fx.Interperter 来调用，比如对 list 和 tuple 的方括号索引，亦即 operator.getitem。因为这些运算对于 inductor IR 来说并没有意义。
2. 对于支持 lowering 的函数。（支持 lowering 指得是注册了 lowering 规则的函数。这部分可以参考 _inductor/lowering.py 中的 lowerings，这是一个全局的 map, 支持将 OpOverload 分解为一个函数调用，但是和 OpOverload 保持相同的接口形式）。形式上判断一个函数支持不支持 lowering, 就看它有没有 _inductor_lowering_function 字段。这是 register_lowering 的副作用。
3. 没有 lowering 规则的，取 op 的 base_name，亦即去掉 variant 名。比如 add.Tensor 的 base_name 就是 add. 看在 FALLBACK_ALLOW_LIST 里是否能找到。（这部分 fallback 是由 eager 执行，也就是还是执行 aten 算子。参考文件 _inductor/lowering.py）
4. 如果 fallback 都没有，就去查找 torch 的 decompositions，这可以通过 register_decompositions 来注册. 这也就是说这个算子没有 lowering 到 inductor IR，但是可以由其他 torch 算子组合出来。（注意：这是 torch._decomp 里的一套规则。是一个多层的字典。（分三部分 post_autograd, pre_autograd, meta，这里会去查 post_autograd 的。因为这已经是 aot_autograd 之后的事情了。)

   但是无论能否查找到，都会报错。因为这种事情应该由 aot_autograd 处理。

Lowering 规则就是将一个 aten core Op 的调用和一个副作用为修改 inductor IR 的函数对应起来。比如很多 pointwise 函数都是通过 register_pointwise 函数注册的。它注册上去的 lowering 函数在

_inductor/lowering.py 中 make_pointwise.inner. 它的效果是进行一份 dtype, broadcast 之类的分析之后，选择合适的函数，然后构造一个 Pointwise 节点。

```
    return Pointwise.create(
        device=device,
        dtype=dtype,
        inner_fn=inner_fn,
        ranges=ranges,
    )
```

至于这些构造出来的 IRNode 节点和 GraphLowering 运行过程中的 buffer 是什么关系还有一些过程。（这个在 GraphLowering 的一般方法 run_node 中体现。

Inductor IR 是以 Buffer 为中心的，不是所有 IRNode 都会被记录下来。它们可能以各种方式 nest 在 Buffer 为中心的 IR 表示中。最后被记录下来的 IR 就是一个 list of Buffers.

## Output

对于 output 的处理方式是

1. 先调用 fx.Interpreter 的方式处理一次得到结果。
2. 对于所有的结果 realize. 具体的做法是 apply ExternalKernel.realize_input 到每一个结果。

```
@classmethod
def realize_input(cls, x):
    if x is None:
        return NoneAsConstantBuffer()
    if isinstance(x, (sympy.Expr, sympy.logic.boolalg.Boolean, int)):
        return ShapeAsConstantBuffer(x)
    if isinstance(x, Constant):
        return V.graph.add_tensor_constant(
            torch.tensor(x.value, dtype=x.get_dtype(), device=x.get_device())
        )
    if isinstance(x, ConstantBuffer):
        return x
    if isinstance(x, TensorBox):
        return cls.realize_input(x.data)
    if isinstance(x, ReinterpretView):
        return x
    if isinstance(x, BaseView):
        x.realize()
        if is_storage_and_layout(x.unwrap_view()) and not isinstance(
            x.unwrap_view().data, ExternKernelAlloc
        ):
            try:
                return cls.convert_to_reinterpret_view(x)
            except NotImplementedError:
                pass
    if isinstance(x, StorageBox):
        # TODO(jansel): impose layout preference on realized buffer
        x.realize()
        return x
    return cls.copy_input(x)
```

这里可以看到对不同的 IRNode 有不同的处理方式，但最重要的是对 TensorBox 的 realize_input, 进而调用 StorageBox 的 realize().

StorageBox 的 realize 是对于 ComputedBuffer, InputsKernel, InputBuffer, ReinterpretView, TemplateBuffer 不做处理，对于 Pointwise 和 Reduction 添加一个 ComputedBuffer 并且注册 buffer. (P.S. 也就是说 Pointwise 和 Reduction 一定会注册 Buffer)

其他的 case 包括

1. contant 的处理，就是添加 Constant 性质的 IRNode.
2. ConstantBuffer, 直接返回
3. 对于 Constant. 也是添加 ConstantBuffer.
4. 对于 View，会调用它的 realize, 这会调用它所 view 的对象的 realize.
5. 对于其他其他情况则是 copy_input. 这会生成一个 Pointwise，然后 realize.

最终就是所有东西都只剩下了 Buffer. (ReinterpreteView 是不会对应实际运算的。)

### run_node

run_node 是执行节点的一般方法。套在上述几种 case 外层执行。

GraphLowering run_node 到某个 tensor 是输出，或者某些原因，必须 realize 的时候，会调用 StorageBuffer 的 realize 方法。而这会将它的 ComputeBuffer 注册到 GraphLowering 的 buffers 中去。

```
688         if any(
689             user.op == "output" or user.target in as_strided_ops for user in n.users
690         ) and isinstance(n.meta["val"], torch.Tensor):
```

这样就把 buffers 分析出来了。

## Inductor IR

Inductor IR 是一种面向 Buffer 的 IR。Buffer 指的是在 kernel launch 之间保持存活的在 Memory 上的一块区域。而且被赋予了某种 layout, 可以 multi-index。

其核心类型是 IRNode(inductor.ir.IRNode).

使用 TORCH_COMPILE_DEBUG=1 环境变量运行使用 torch.compile 的代码，就会在当前目录的 torch_compile_debug/<runtime>/inductor/model____xx 目录下得到 inductor 的输出。可以参考 https://pytorch.org/tutorials/intermediate/inductor_debug_cpu.html

| File | Description |
| --- | --- |
| `fx_graph_readable.py` | ... |
| `fx_graph_runnable.py` | Executable FX graph, after decomposition, before pattern match |
| `fx_graph_transformed.py` | Transformed FX graph, after pattern match |

| File | Description |
| --- | --- |
| ir_post_fusion.txt | Inductor IR before fusion |
| ir_pre_fusion.txt | Inductor IR after fusion |
| output_code.py | Generated Python code for graph, with C++/Triton kernels |

想要观察 inductor IR，则可以看 ir_pre_fusion.txt 和 ir_post_fusion.txt.

## Example: Softmax

我们可以使用一个简单的 softmax 来做实验。

```
import torch
x = torch.randn(10, 3840).cuda()

def f(x):
    return torch.softmax(x, 1)
compiled_f = torch.compile(f)
y = compiled_f(x)
y_ref = f(x)

torch.testing.assert_close(y, y_ref)
```

经过 aot_autograd 化简和 functionalize 的 fx.Graph

```
def forward(self, arg0_1):
    amax = torch.ops.aten.amax.default(arg0_1, [1], True)
    sub = torch.ops.aten.sub.Tensor(arg0_1, amax);  arg0_1 = amax = None
    exp = torch.ops.aten.exp.default(sub);  sub = None
    sum_1 = torch.ops.aten.sum.dim_IntList(exp, [1], True)
    div = torch.ops.aten.div.Tensor(exp, sum_1);  exp = sum_1 = None
    return (div,)
```

在融合之前的 IR

```
buf0: SchedulerNode(ComputedBuffer)
buf0.writes = [MemoryDep('buf0', c0, {c0: 10})]
buf0.unmet_dependencies = []
buf0.met_dependencies = [MemoryDep('arg0_1', c0, {c0: 38400})]
buf0.users = [NodeUser(node=SchedulerNode(name='buf1'), can_inplace=True),
NodeUser(node=SchedulerNode(name='buf2'), can_inplace=False)]
buf0.group.device = cuda:0
buf0.group.iteration = (10, 3840)
buf0.sizes = ([10], [3840])
class buf0_loop_body:
    var_ranges = {z0: 10, z1: 3840}
    index0 = 3840*z0 + z1
    index1 = z0
    def body(self, ops):
        get_index = self.get_index('index0')
        load = ops.load('arg0_1', get_index)
        reduction = ops.reduction(torch.float32, torch.float32, 'max', load)
        get_index_1 = self.get_index('index1')
        store_reduction = ops.store_reduction('buf0', get_index_1, reduction)
        return store_reduction
```

```
buf1: SchedulerNode(ComputedBuffer)
buf1.writes = [MemoryDep('buf1', c0, {c0: 10})]
buf1.unmet_dependencies = [MemoryDep('buf0', c0, {c0: 10})]
buf1.met_dependencies = [MemoryDep('arg0_1', c0, {c0: 38400})]
buf1.users = [NodeUser(node=SchedulerNode(name='buf2'), can_inplace=False)]
buf1.group.device = cuda:0
buf1.group.iteration = (10, 3840)
buf1.sizes = ([10], [3840])
class buf1_loop_body:
    var_ranges = {z0: 10, z1: 3840}
    index0 = 3840*z0 + z1
    index1 = z0
    def body(self, ops):
        get_index = self.get_index('index0')
        load = ops.load('arg0_1', get_index)
        get_index_1 = self.get_index('index1')
        load_1 = ops.load('buf0', get_index_1)
        sub = ops.sub(load, load_1)
        exp = ops.exp(sub)
        reduction = ops.reduction(torch.float32, torch.float32, 'sum', exp)
        get_index_2 = self.get_index('index1')
        store_reduction = ops.store_reduction('buf1', get_index_2, reduction)
        return store_reduction


buf2: SchedulerNode(ComputedBuffer)
buf2.writes = [MemoryDep('buf2', c0, {c0: 38400})]
buf2.unmet_dependencies = [MemoryDep('buf0', c0, {c0: 10}), MemoryDep('buf1',
c0, {c0: 10})]
buf2.met_dependencies = [MemoryDep('arg0_1', c0, {c0: 38400})]
buf2.users = [NodeUser(node=OUTPUT, can_inplace=False)]
buf2.group.device = cuda:0
buf2.group.iteration = (38400, 1)
buf2.sizes = ([10, 3840], [])
class buf2_loop_body:
    var_ranges = {z0: 10, z1: 3840}
    index0 = 3840*z0 + z1
    index1 = z0
    def body(self, ops):
        get_index = self.get_index('index0')
        load = ops.load('arg0_1', get_index)
        get_index_1 = self.get_index('index1')
        load_1 = ops.load('buf0', get_index_1)
        sub = ops.sub(load, load_1)
        exp = ops.exp(sub)
        get_index_2 = self.get_index('index1')
        load_2 = ops.load('buf1', get_index_2)
        div = ops.div(exp, load_2)
        get_index_3 = self.get_index('index0')
        store = ops.store('buf2', get_index_3, div, None)
        return store
```

一共有三个 buffer. 分别用于存储 max 的结果，logsumexp 的结果，和最后的结果。每个 buffer 记录了

1. 它需要写的区域，
2. 它的 dependencies(包括已经满足的和未满足的），
3. 它的 user，以及它的 iteration 空间，

4. iteration 空间每个维度的 size，以及对应的 buffer 的 size. 一个 iteration 维度可以绵延 buffer 的多个 dimension.

因此这也是一种 Node 形式的 IR. 但这里输出的时候，是加上了 SchedulerNode 的形式。此处可以先分析 IRNode，在上面的具体 case 中是 ComputedBuffer（这是 IRNode 的子类）。

在 GraphLowering.run(*example_input) 执行完之后，可以看到 3 个 buffer.

```
ComputedBuffer(name='buf0', layout=FixedLayout('cuda', torch.float32, size=[10, 1], stride=[1, 10]), data=Reduction(
  'cuda',
  torch.float32,
  def inner_fn(index, rindex):
      i0, _ = index
      r0 = rindex
      tmp0 = ops.load(arg0_1, r0 + 3840 * i0)
      return tmp0
  ,
  ranges=[10, 1],
  reduction_ranges=[3840],
  reduction_type=max,
  origin_node=amax,
  origins={amax}
))

ComputedBuffer(name='buf1', layout=FixedLayout('cuda', torch.float32, size=[10, 1], stride=[1, 10]), data=Reduction(
  'cuda',
  torch.float32,
  def inner_fn(index, rindex):
      i0, _ = index
      r0 = rindex
      tmp0 = ops.load(arg0_1, r0 + 3840 * i0)
      tmp1 = ops.load(buf0, i0)
      tmp2 = tmp0 - tmp1
      tmp3 = ops.exp(tmp2)
      return tmp3
  ,
  ranges=[10, 1],
  reduction_ranges=[3840],
  reduction_type=sum,
  origin_node=sum_1,
  origins={sum_1, sub, exp}
))


ComputedBuffer(name='buf2', layout=FixedLayout('cuda', torch.float32, size=[10, 3840], stride=[3840, 1]), data=Pointwise(
  'cuda',
  torch.float32,
  def inner_fn(index):
      i0, i1 = index
      tmp0 = ops.load(arg0_1, i1 + 3840 * i0)
      tmp1 = ops.load(buf0, i0)
      tmp2 = tmp0 - tmp1
      tmp3 = ops.exp(tmp2)
      tmp4 = ops.load(buf1, i0)
      tmp5 = tmp3 / tmp4
      return tmp5
  ,
  ranges=[10, 3840],
  origin_node=div,
```

```
  origins={div, sub, exp}
))
```

里面记录了每个 ComputedBuffer 的信息，包括

1. name
2. ranges，形状（亦即索引的有效范围）。
3. layout. 这里指的是 multi-index 到 offset 的映射。用的是 torch 的 FixedLayout.（sizes & strides).
4. data：运算类型，Pointwise 或者 Reduction. 具体的运算函数。
5. origin_node, origins: 对应原 fxNode, 第一个 Node 和融合到它上面的 pointwise 运算对应的 Node.

## IR constructs

核心的类型是 IRNode. 具有众多子类。67 个。完整的 type hierechy 如下。

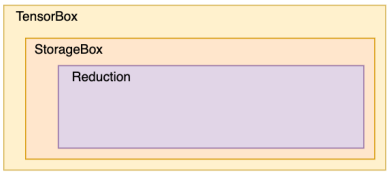
```
IRNode
    BaseConstant
        Constant
        IndexingConstant
    DynamicScalar
    ShapeAsConstantBuffer
    NoneAsConstantBuffer

    Layout
        AliasedLayout
        FixedLayout
        FlexibleLayout
        MutationLayout
    MultiOutputLayout

    MutableBox
        StorageBox
        TensorBox

    Loops
        Pointwise
            Scatter
        Reduction

    BaseView
        GenericView
            View
                SliceView
        SqueezeView
        PermuteView
        ReinterpretView
        ExpandView

    Buffer
        ComputedBuffer
        InputBuffer
            ConstantBuffer
        InputsKernel
            ExternKernel
```

```
                CollectiveKernel
                    InPlaceCollectiveKernel
                        AllReduce
                        AllReduceCoalesced
                    OutOfPlaceCollectiveKernel
                        AllGatherIntoTensor
                        AllGatherIntoTensorCoalesced
                        ReduceScatterTensor
                        ReduceScatterTensorCoalesced
                ExternKernelAlloc
                    LinearBinary
                    LinearUnary
                    MKLPackedLinear
                    MkldnnRnnLayer
                    FallbackKernel
                    Wait
                    QConv
                    ConvolutionBinary
                    ConvolutionBinaryInplace
                    ConvolutionTransposeUnary
                    ConvolutionUnary
                ExternKernelOut
                    RandomSeeds
                    DeviceCopy
                InPlaceHint
                IndexPutFallback
                ScatterFallback
                OutputBuffer
                InplaceBernoulliFallback
                MultiOutput
                    MultiOutputNoSizeAssert
            NopKernel
                ConcatKernel
        TemplateBuffer
```

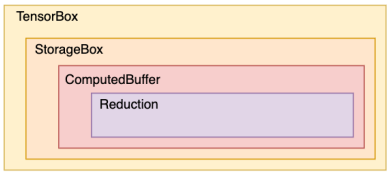
不仅 Buffer 是 IRNode，Reduction, Pointwise 是 IRNode, Layout 也是 IRNode. 但这些子类并不是平坦的，功能存在很大的区别。主要的子类分为四种：

1. 表示在全局内存上的对象的 Buffer;
2. 表示 Buffer 内存布局的 Layout;
3. 表示运算的 Loops 和表示视图变换的 BaseView;
4. 表示张量的 TensorBox.

常见的几种节点内的数据结构如下。




运算节点




运算 Buffer 节点


输入 Buffer 节点

Layout 是属于 buffer 的属性。当一个运算节点被创建出来的时候，它是一个 TensorBox → StorageBox → Reduction/Pointwise 类型的结构，里面是没有 buffer 的，自然也没有 layout. 一切都发生在 StorageBox 被 realize 的时候。但是 StorageBox 节点的 realize 时机有很多个。

关于代码生成的类型，inductor 只会生成 triton 的 pointwise, reduction, persistent_reduction 和 template 代码。ExternalKernel 是不会有代码生成的，只会有 wrapper code 去调用它。

1. Pointwise: 就是我们理解的那个 pointwise
2. reduction: inductor 写的一种特定的 strided reduction. 是用 for 循环写的
3. persistent_reduction, 调用 triton.language.reduce
4. template: 使用 triton 模板。目前只有 for_each, bmm 和 mm 之类的 kernel 会用到 template. 可以参见 _inductor/select_algorithm.TritonTemplate

# Buffer & Buffer Selection

## Buffer

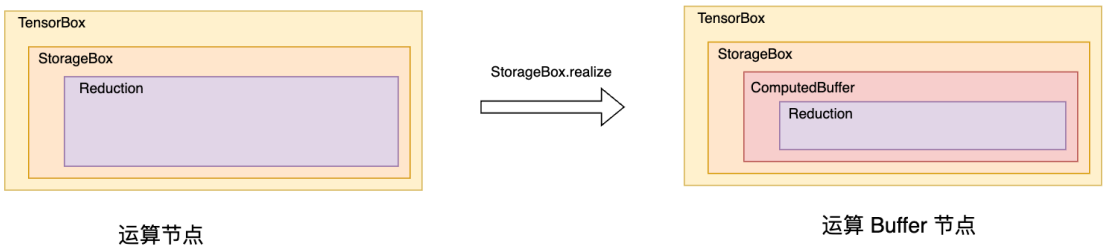
在 inductor IR 中，buffer 是一篇可以 multi-index 的内存区域。有几个大的子类型

1. ComputedBuffer，表示 Pointwise, Reduction 运算结果的 Buffer;
2. TemplateBuffer，表示 Template 类型运算的 Buffer;
3. InputBuffer，表示输入 Buffer;
4. InputsKernel，表示用于外部 Kernel 的 Buffer. (我揣度的意义）

这和 MLIR 中的 bufferization 差别在什么地方?

## Realization

在 inductor IR 中主要以 TensorBox 内包含一个 StorageBox, StorageBox 内包含一个表示运算的节点的方式来组织成 DAG. 但并不是所有的 StorageBox 都会被选择为 buffer. 所谓的选择为 buffer 指的是构造从一个运算节点变成一个运算 buffer 节点。




这是通过 StorageBox.realize 来实现的。本来 StorageBox.data 是一个运算 Node, 调用之后就变成了一个 ComputedBuffer 节点，它的内部才是原本的运算节点。

```
def realize(self):
    if isinstance(
        self.data,
        (
            ComputedBuffer,
            InputsKernel,
            InputBuffer,
            ReinterpretView,
            TemplateBuffer,
        ),
    ):
        return self.data.get_name()
    assert isinstance(self.data, (Pointwise, Reduction)), type(self.data)
    origin_node = self.data.get_origin_node()
    traceback = self.data.get_traceback()
    self.data = ComputedBuffer(
        name=None,
        layout=FlexibleLayout(
            device=self.data.get_device(),
            dtype=self.data.get_dtype(),
            size=self.data.get_size(),
        ),
        data=self.data,
    )
    self.data.name = V.graph.register_buffer(self.data)
    self.data.origins = self.origins
    self.data.origin_node = origin_node
    self.data.traceback = traceback
    return self.data.name
```

另外存在一种定位 realize. realize_into.

## Registration

至于 buffer 的注册（被添加到 GraphLowering.buffers 列表中），则是通过 register_buffer 接口完成。

虽然 StorageBox.realize 中会注册 buffer. 但这不是注册 Buffer 的唯一时机，还有一些其他的时机可以注册 buffer.

## Input As Border

对于计算图的输入，都是需要 buffer 的。但是这也不归这个子图来决定。比如 softmax 运算子图中的输入是一个装着 InputBuffer 的 TensorBox.

```
TensorBox(StorageBox(
  InputBuffer(name='arg0_1', layout=FixedLayout('cuda', torch.float32, size=[4096, 1024], stride=[1024, 1]))
))
```

这是在 GraphLowering 对 placeholder 节点的处理方式中决定的。

```
# TODO(jansel): handle input aliasing
tensor = TensorBox.create(
    InputBuffer(
        target,
        FixedLayout(example.device, example.dtype, sizes, strides),
    )
)
self.graph_inputs[target] = tensor
```

## Output As Border

对于子图的输出，也必须选定为 buffer, 因为它们可能被 pytorch 的其他算子作为输入使用。这是从这个编译子图返回到 torch 的出口。这是在 GraphLowering 对 output 节点的处理方式中决定的。

```
def output(self, target, args, kwargs):
    result = super().output(target, args, kwargs)
    assert isinstance(result, (tuple, list)), type(result)
    assert all(
        isinstance(
            x,
            (
                TensorBox,
                ir.Constant,
                type(None),
                ir.ConstantBuffer,
                sympy.Expr,
                sympy.logic.boolalg.Boolean,
                int,
            ),
        )
        for x in result
    ), result
    self.graph_outputs = [ir.ExternKernel.realize_input(x) for x in result]
```

所有的输出都会被当作 ExternalKernel 的输入被 realize. 这个 function 最主要的功能仍然是调用 StorageBox.realize. （此外还有一些对常量的处理，以及 fallback 的情况是创建一个 Pointwise 来 copy 结果，并 realize 之，但不是此处的重点。）

## ExternKernel As Border

ExternKernel 以及它的子类 ExternKernelOut, ExternKernelAlloc, InplaceBernoulliFallback, ScatterFallback, IndexPutFallback, MultiOutput, CollectiveKernel, InPlaceHint, OutputBuffer 作为外部 Kernel, 它们的 buffer 本身会被注册。P.S. ExtenKernel 本身就继承了 Buffer (ExternKernel ← InputsKernel ← Buffer). 所以它们不是像 ComputedBuffer 里包装着一个 Reduce 或者 Pointwise 那样。他们本身就是 Buffer, 不需要 realize. 只需要考虑 registration.

## TemplateBuffer

TemplateBuffer ← Buffer 的输入本身也需要注册.

## Pointwise

对于 pointwise 函数的 lowering 方法一般使用 make_pointwise 函数生成。它不会注册 Buffer。

## Reduction

对于 reduce 函数的 lowering 方法一般使用 make_reduce 函数生成。它会注册 buffer.

## Inplace Mutation

1. MutationLayout 和写入有关。当一个 buffer 需要被写时，realize 它的所有 users. 参见 mark_buffer_mutated.
2. 对于会修改输入的函数的 lowering 会注册新的 buffer. aten.index_put_, aten.scatter_reduce_.

## Cost Consideration

可以参考

```
def has_exceeded_max_reads(self):
    return isinstance(self.data, Pointwise) and (
        self.num_reads() > config.realize_acc_reads_threshold
        or self.inner_fn_str_len() > config.realize_bytes_threshold
    )
```

对于 Pointwise 运算，一方面的限制来自 user 个数，这是为了避免过多的重计算；

另一个限制来自函数体长度，如果函数的文本长度过长，可能也意味着运算量渐渐增大。

以及 mark_reuse, 里面有针对运算量是不是 heavy 的判断。（用的真的是字符串搜索）

```
def mark_reuse(self, users):
    """
    A heuristic to decide if we should realize a tensor
    that is used multiple times.
    """
```

---

```
    def should_realize_on_cpu(loops: Union[Pointwise, Reduction]):
        """
        The heuristic for realizing reused result of heavy ops on cpu
        """
        heavy_ops = ["exp"]  # a list of heavy ops
        fn_str = loops.inner_fn_str()
        return any((op + "(") in fn_str for op in heavy_ops)

    if (
        users > 1
        and isinstance(self.data, (Pointwise, Reduction))
        and (
            self.num_reads() > config.realize_reads_threshold
            or len(self.inner_fn_str()) > config.realize_bytes_threshold
            or (is_cpu(self.data) and should_realize_on_cpu(self.data))
        )
    ):
        self.realize()
```

和 realize_hint

```
def realize_hint(self):
    """
    Called on buffers we expect to be forced to realize later.
    """
    if (
        isinstance(self.data, (Pointwise, Reduction))
        and self.num_reads() > 1
        and self.is_pointwise_non_scalar_tensor_num_reads_larger_than_one()
    ):
        self.realize()
```

## Operations requiring realized input

部分 op 要求输入是 realized input. 也就是说不能和前面的 Op fuse 在一起。但它们是在 aten op 的层面进行的判断，和在 inductor IR 层面的 ExternKernel 不同。

**torch._inductor.lowering.needs_realized_inputs** 集合中包含了所有需要 realized input 的 op. 这里包含了很多的 op. 比如 mm, bmm, convolution, sort, fft, 矩阵分解等， 基本上不是那么简单地可以分级为 Pointwise 和 Reduction 的运算都在这里。

```
for op in lowering.needs_realized_inputs:
    if isinstance(op, torch._ops.OpOverload):
        print(op.name())
    elif isinstance(op, torch._ops.OpOverloadPacket):
        for o in op.overloads():
            print(f"{op}.{o}")
```

## Operation requiring explicit layout

部分 op 要求输出是某种确定的排布，必须 realize

1. aten.as_strided, empty_strided

部分的 op 实现得不够 general, 必须要求输入是一定的 FixedLayout. 这是 inductor dev 内部也存在争议的部分。

1. torch.ops.aten.convolution_backward.default
2. torch.ops.aten.mm.default
3. torch.ops.aten._int_mm.default
4. ...

# Layout & Layout Decision

在 inductor 中 buffer 有多种 layout. 这里我们涉及 FixedLayout 和 FlexibleLayout，前者是不可更改的，而后者是可以更改的。

在 GraphLowering.run 构造 inductor IR 的结束阶段，会调用 finalize 方法，在这里对每一个 buffer 调用 decide_layout 将 FlexibleLayout 确定为 FixedLayout.

```
    def finalize(self):
        for buf in self.buffers:
            buf.decide_layout()
```

这里只能对 FlexibleLayout 做处理。

```
    def decide_layout(self):
        if isinstance(self.layout, FlexibleLayout):
            order = self.get_fill_order()
            if order:
                self.freeze_layout_with_fill_order(order)
            else:
                self.freeze_layout()
```

但这不是唯一一处会调用 decide_layout 的地方。还有其他地方会调用 decide_layout 去修改 Buffer 的 layout. 下面主要讲述 FlexibleLayout 和 FixedLayout 的功能。

## FlexibleLayout

FlexibleLayout 表示一种可以更改的 Layout. 构造的时候只需要 device, dtype, size 就可以构造。并没有指定 stride 具体是多少。它的 flexible 也只是在 stride 上 flexible.

FlexibleLayout 除了有 initializer 之外有多个 staticmethod 可以用于构造实例。主要有几种构造方式都是通过计算 stride 得到的：

1. contiguous_strides: 给定形状，构造 C-contiguous 排布的 stride. 比如 (3, 4, 5) 的 stride 是 (20,5,1)
2. fill_order: 给定形状和 axis 沿着 stride 从小到大的排列。比如 (N, C, H, W) 的形状，但是实际排布中 C 维度的 stride 最小，W, H, N 依次增大（亦即 channel-last 布局或者 NHWC 格式），那么 fill_order 就是 (1, 3, 2, 0).
3. stride_order: 给定形状和每个 axis 在 stride 从小到大排序中的 order，计算出 stride. 比如 (N, C, H, W) 形状的 NHWC 格式，那么 stride order 就是 (3, 0, 2, 1). 这和 fill_order 之间可以互相转换。
4. same_order：给定形状和 stride。根据 stride 从小到大的算出排序结果，然后用 fill_order 的方式计算 stride.

默认情况下创建 Pointwise 和 Reduction 对应的 ComputedBuffer 的时候，layout 就是 FlexibleLayout.

比如：

```
reduce_amax = register_lowering(aten.amax)(make_reduction("max"))
```

aten.amax 对应的 lowering 函数行为是先创建计算节点，然后 realize.

```
def make_reduction(reduction_type: str, override_return_dtype=None):
    def inner(x, axis=None, keepdims=False, *, dtype=None):
        kwargs = _make_reduction_inner(
            x,
            axis=axis,
            keepdims=keepdims,
            dtype=dtype,
            override_return_dtype=override_return_dtype,
        )
        result = Reduction.create(reduction_type=reduction_type, **kwargs)
        if isinstance(
            result.data.data, Reduction
        ):  # Only realize if reduction isn't unrolled
            result.realize()
        return result

    return inner
```

而一个包含着计算节点的 TensorBox 的 realize 方法会调用 StorageBox 的 realize 方法。它会在 Pointwise 和 Reduction 计算的外层再套上一层 ComputedBuffer, 此时附加的 layout 是 FlexibleLayout.

```
    self.data = ComputedBuffer(
        name=None,
        layout=FlexibleLayout(
            device=self.data.get_device(),
            dtype=self.data.get_dtype(),
            size=self.data.get_size(),
        ),
        data=self.data,
    )
    self.data.name = V.graph.register_buffer(self.data)
```

而 pointwise 运算的 lowering 函数则不会 realize. 具体可以参考 make_pointwise 函数返回的 closure.

## FixedLayout

这表示一种不可更改的 layout. 构造 FixedLayout 需要 **device, dtype, size, stride, offset**. 内部布局就已经完全确定了。

1. fx.Graph 中的 placeholder 节点对转化为一个 TensorBox, 内部是一个 FixedLayout 的 InputBuffer. 具体参见 GraphLowering.placeholder.
2. ExpandView, PermuteView, SqueezeView, View, SliceView 的静态 create 方法都有可能返回一个带有 FixedLayout 的 ReinterpretView. 一系列的 View 性质的 op 的 lowering 方法中都会用到它们。因为这些视图变换类的 op 产出的可能是一个 ReinterpretView，但是这个 View 的 layout 是有固定要求的，否则它就不能是一个 ReinterpretView.
3. ConcatKerne.create 输出的 StorageBox 是 FixedLayout.
4. ExternalKernel 一般对输入或者输出有 Layout 的要求，会需要将输入转换为 ReInterpretView. (convert_to_reinterpret_view)
5. 部分 op 的语义本身包含对 layout 的说明。比如 as_stride, random_stride.

下面的一些 case 是将 FlexibleLayout 转换为 FixedLayout 的方法。

1. GraphLowering 运行的最后阶段会调用 finalize, 使用 decide_layout 来修改 FlexibleLayout.
2. as_storage_and_layout: 这个函数会修改 Buffer 或者 ReinterpretView 的 layout . 此举会将 FlexibleLayout 修改为 FixedLayout. 在创建 ExpandView 等的 View 类 IR Construct 的时候会调用这个方法。

```
def as_storage_and_layout(x, freeze=True, want_contiguous=False, stride_order=None):
    """Try to simplify x into a StorageBox and a Layout"""
    if isinstance(x, TensorBox):
        return as_storage_and_layout(
            x.data,
            freeze=freeze,
            want_contiguous=want_contiguous,
            stride_order=stride_order,
        )
    if isinstance(x, StorageBox) and isinstance(x.data, Buffer):
        if freeze:
            if want_contiguous:
                x.data.freeze_layout()
                assert x.data.layout.is_contiguous()
            elif stride_order is not None:
                x.data.freeze_layout_with_stride_order(stride_order)
            else:
                x.data.decide_layout()
        return x, x.data.layout
    if isinstance(x, ReinterpretView):
        # making the base of x contiguous or stride_ordered will not necessarily make
        # the ReinterpretedView either, so dont pass along those arguments
        buffer, _ = as_storage_and_layout(
            x.data,
            freeze=freeze,
        )
        return buffer, x.layout
    raise NotImplementedError
```

free_layout 是直接将 stride 固定下来。freeze_layout_with_stride_order 是根据 stride_order 从小到大算出 stride. decide layout 则是根据 fill order 计算出 stride。所以从这里来看，buffer 的 layout 决定算法并没有太多的全局考虑。而是选择一种连续成片的 buffer。(最小地址和最大地址之间的每一个地址都有至少一个 multi-index 对应它。）

# Inductor IR Optimization

Actually no explicit optimization steps.

# Inductor IR Code Generation

借着上文中，完成 inductor IR 的构建之后，就是进行代码生成的过程了。（

```
    with V.set_graph_handler(graph):
        graph.run(*example_inputs)
        context = torch._guards.TracingContext.get()
        if context is not None and context.output_strides is not None:
            # Return the output strides to the caller via TracingContext
            assert len(context.output_strides) == 0
            for out in graph.graph_outputs:
                if hasattr(out, "layout"):
                    context.output_strides.append(
                        tuple(
                            V.graph.sizevars.size_hint(s) for s in out.layout.stride
                        )
                    )
                else:
                    context.output_strides.append(None)
        compiled_fn = graph.compile_to_fn()
        compiled_graph = CompiledFxGraph(
            compiled_artifact=compiled_fn,
            cache_key=graph.cache_key,
            artifact_path=graph.cache_path,
            cache_linemap=graph.cache_linemap,
            device_types=graph.device_types,
            device_idxs=graph.device_idxs,
            mutated_inputs=graph.mutated_inputs,
        )
    return compiled_graph
```

核心的函数是 GraphLowering.compile_to_fn → GraphLowering.compile_to_module

进而调用 **GraphLowering.codegen** 函数。

```
    def codegen(self):
        from .scheduler import Scheduler

        self.init_wrapper_code()

        self.scheduler = Scheduler(self.buffers)
        assert self.scheduler is not None  # mypy can't figure this out
        self.scheduler.codegen()
        assert self.wrapper_code is not None
        return self.wrapper_code.generate()
```

其中分为 kernel 生成和 wrapper 生成。wrapper 是指生成的代码中的 call 函数，它负责去调用多个 triton kernel. 而 kernel 函数则是 triton kernel 本体。

其中对于 kernel 的生成分为两部分。

1. 根据上一步生成的 buffers 构造 Scheduler,
2. 再调用 Scheduler.codegen().

由于不喜欢太多的嵌套。所以把下面两节的 title 提升了一番。

# Scheduler

Scheduler 初始化的过程是给定一系列 ComputedBuffer, 分析出一系列 ScheduleNode 的过程，其中会进行融合，裁剪等分析。

首先是创建 ScheduleNode.

```
self.nodes = [self.create_scheduler_node(n) for n in nodes]
```

构造 ScheduleNode 的过程需要使用针对设备的 Schedule 方案。

```
    def create_scheduler_node(self, node):
        assert (
            node.origins is not None
        ), "All nodes passed to scheduling must have an origin"
        if node.is_no_op():
            return NopKernelSchedulerNode(self, node)
        elif isinstance(node, (ir.ComputedBuffer, ir.TemplateBuffer)):
            group_fn = self.get_backend(node.get_device()).group_fn
            return SchedulerNode(self, node, group_fn)
        elif isinstance(node, ir.ExternKernel):
            return ExternKernelSchedulerNode(self, node)
        else:
            raise NotImplementedError(node)
```

从前文的分析来看，这里的 node 只会是 ComputedBuffer, InputsKernel, InputBuffer, ReinterpretView, TemplateBuffer 中的一种。

首先会根据 node 所在的设备，选择 group_fn. 这里会设计另一个另一个比较重要的概念 TritonScheduling，这是 triton 代码生成用的 Scheduling. 但是对于 cpu 则有其他的 CppScheduling. 这部分可以参考 Scheduler.create_backend。它会填充 Scheduler.backends 字典，每种设备一个 Scheduling.

## ScheduleNode

ScheduleNode 基类 BaseScheduleNode。其中包含的信息如下

```
class BaseSchedulerNode:
    def __init__(self, scheduler: "Scheduler", node: ir.Buffer):
        self.scheduler: Scheduler = scheduler
        self.node: ir.Buffer = node
        self.users: Optional[List[NodeUser]] = None
        self.inverse_users: List[BaseSchedulerNode] = []
        self.set_read_writes(node.get_read_writes())
        self.recursive_predecessors: Optional[Set[str]] = None
        self.min_order: Optional[int] = None
        self.max_order: Optional[int] = None
        self.last_usage: Set[str] = None  # buffers that won't be used after this kernel
        self.written = False
```

但是除了 node 之外，其他的信息都是等待填充的。后续的其他的字段（没有写在 __init__ 中的）包括

1. read_writes: 节点的读写
2. unmet_dependencies：未满足的依赖，就是 read_writes 中的 reads.

ScheduleNode 扩展的字段有 _sizes, _body, group. 其中关于 _sizes 和 _body. 有一个比较重要的变换是根据数据的 layout 进行化简，ComputedBuffer.simplify_and_reorder 和 TemplatedBuffer.simplify_and_reorder。

```
class SchedulerNode(BaseSchedulerNode):
    def __init__(self, scheduler: "Scheduler", node: ir.ComputedBuffer, group_fn):
        super().__init__(scheduler, node)
        (
            self._sizes,
            self._body,
        ) = node.simplify_and_reorder()

        self.group = (node.get_device(), group_fn(self._sizes))

        if self.is_template():
            self.set_read_writes(node.normalized_read_writes())
        else:
            self.set_read_writes(
                dependencies.extract_read_writes(
                    self._body, *self._sizes, normalize=True
                )
            )
```

其中主要是 ComputedBuffer 的 simplify_and_reorder 是有意义的。因为对于 TemplatedBuffer，基本没有操作。 ComputedBuffer 的 simplify_and_reorder 包含的变换有：

1. 去除 size 为 1 的维度，这就是徒增维度；
2. 合并连续的维度。（当 stride[i+1] * sizes[i +1] == stride[i] 的时候 i+1 轴可以合并到 I 维度上；
3. 根据 stride 从大到小重排维度。

比如在这里例子里，_sizes 就是 ([10], [3840]).

由于会进行化简，所以当创建一些配置的时候，会发现 inductor 生成的 Schedule 比较智能。比如可以合并连续维度。

```
def f(A, B):
   return A + B
```

然后给定输入的 A 和 B 都是 (30, 40) 矩阵。其次，经过化简，_sizes 是 ([1200], []).

对于 reduction on inner most dimension 且 size 大于一定的值的情况，reduce 会被分多次执行。比如 A 是 (10, 384000, 5) 的矩阵。沿着最后一维 reduce sum. 会在 inductor IR 构造的阶段就分成两份。

```
10, 38400 (10, 47, 8171)→ 10, 47 → 10
```

这样在 ScheduleNode 构造的时候，第一步 reduce 的 _size 就会是 ([10, 47], [8171]). group 则是分组乘起来 (470, 8171).

## Optimization

其次还有不少的分析与变换，比如 dependency 分析，这会填充每个 ScheduleNode 节点的 user 和 inverse_user. 进而进行 topological_sort 和 dce.

```
self.compute_dependencies()
self.topological_sort_schedule()
self.compute_predecessors()
self.dead_node_elimination()
```

## Fuse

其次是 fuse_nodes

```
        metrics.ir_nodes_pre_fusion += len(self.nodes)
        V.debug.ir_pre_fusion(self.nodes)
        self.num_orig_nodes = len(self.nodes)
        self.name_to_fused_node = {n.get_name(): n for n in self.nodes}
        self.create_foreach_nodes()
        self.topological_sort_schedule()
        self.fuse_nodes()
        self.compute_last_usage()
        V.debug.ir_post_fusion(self.nodes)
        V.debug.graph_diagram(self.nodes)
        self.debug_draw_graph()
```

这会修改 name_to_fused_node. 其中 fuse 的条件和 fuse 变换的部分则是另一个题目了。一个例子是，对于 size 比较小的 softmax on inner most dimension, 可以看到三个 buffer 融合成一个 ScheduleNode, 对于 softmax on outer simension, 则可以看到三个 buffer 融合成两个 ScheduleNode.

Inductor scheduler 的 fuse 也是采用一种贪心的 fuse 方式，走一步算一步。最多走十步或者提前因为无事可做而停止。

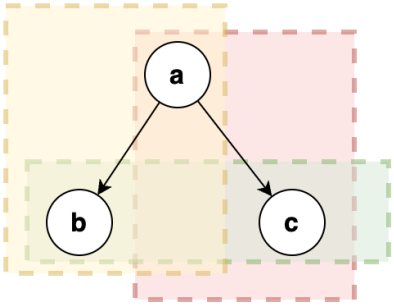
```
def fuse_nodes(self):
    """
    Mutates self.nodes to combine nodes into FusedSchedulerNodes.
    """
    for _ in range(10):
        old_len = len(self.nodes)
        self.fuse_nodes_once()
        if len(self.nodes) == old_len:
            break
```

其中 fuse_nodes_once 的基本流程是，找出可以 fuse 的 op pairs, 按分数排序，然后只要满足 can_fuse 的条件且不会引入环，就 fuse.

```
    def fuse_nodes_once(self):
        """…
        fused_nodes = set(self.nodes)
        for node1, node2 in self.get_possible_fusions():
            node1 = self.name_to_fused_node[node1.get_first_name()]
            node2 = self.name_to_fused_node[node2.get_first_name()]
            if self.can_fuse(node1, node2) and not self.will_fusion_create_cycle(
                node1, node2
            ):
                node3 = fuse(node1, node2)
                fused_nodes.remove(node1)
                fused_nodes.remove(node2)
                fused_nodes.add(node3)
                self.name_to_fused_node.update(
                    {n.get_name(): node3 for n in node3.get_nodes()}
                )
```

列出所有 **candidates**

inductor 的 fusion 是基于对 buffer 的使用来分析的。如果同一个 buffer 会被多个 ScheduleNode 使用(包括读和写），那么就会考虑将这两个 ScheduleNode 融合(但仍然需要考虑 can_fuse 条件是否满足）。这确实是减少 kernel launch 的一种方式, 这里包含了 vertical fuse 和 horizontal fuse.

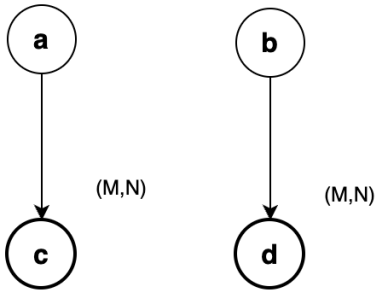



**fusion based on buffer access**

每一个 ScheduleNode 都会 write 本身对应的 buffer. 所以以三种 fuse 都会被考虑。

a writes a, c reads a -> consider (a, c)
a writes a, b reads a -> consider(a, b)
b reads a, c reads a -> consider(b, c)

此外，还有一种是当两个 ScheduleNode 的 group 一致，也就是说它们的 task 索引空间一致的情况下，也会考虑将这两个 ScheduleNode 融合。这对应的是 loop fusion.




**fusion based on iteration pattern**

c has group (M, N), d has group (M, N) -> consider (c, d)

关于 fusion 的评分有量方面的考虑，参见 Scheduler.score_fusion

1. Estimate of the saved memory operations
2. Fusions closer together in original order (和节点序有关，min_order 和 max_order)

判断 **can_fuse**

can_fuse 是一系列的 if-else, 写了很多情况不可以 fuse. 只有剩余的情况可以 fuse. 在排除了不能 fuse 的情况下。由 Scheduler 判断是否可以 vertical 或者 horizontal fuse, 在 Scheduler 并不明确不能 fuse 的情况下，最终需要由 backend 的 Scheduling(TritonScheduling 和 CppScheduling) 来决定是否可以 vertical fuse 或者 horizontal fuse.

作为例子：(10, 3840) softmax over axis 1 的 ScheduleNode 只剩下了一个。

```
buf0_buf1_buf2: FusedSchedulerNode(NoneType)
buf0_buf1_buf2.writes =
    [   MemoryDep('buf0', c0, {c0: 10}),
        MemoryDep('buf1', c0, {c0: 10}),
        MemoryDep('buf2', c0, {c0: 38400})]
buf0_buf1_buf2.unmet_dependencies = []
buf0_buf1_buf2.met_dependencies = [MemoryDep('arg0_1', c0, {c0: 38400})]
buf0_buf1_buf2.users = None
buf0_buf1_buf2.snodes = ['buf0', 'buf1', 'buf2']
```

但并非所有的 softmax 都会被 fuse 成这样。比如

## Scheduler Codegen

完成了 fusion 之后，到了真正的 codegen 阶段。就是生成 triton 代码字符串的阶段。包含 kernel 和 wrapper 两个部分。其中涉及到一个比较重要的类型是 IterationRange, 这是和符号运算相关的部分。稍候整理补充。

## InductorBackend Compile

跳转到 Triton 编译文章。