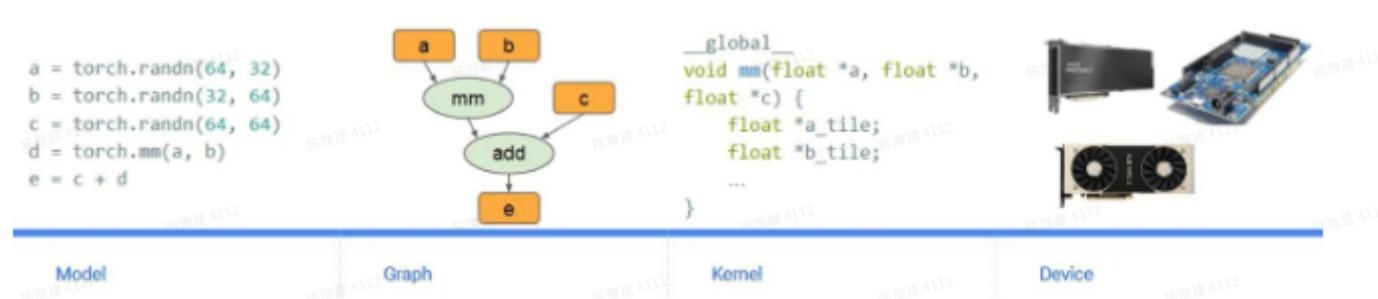


Triton 学习

概述

Triton由OpenAI、哈佛大学、IBM等机构的研究者们共同研发，并于2021年开源。

概括来说，triton是基于Python的DSL，面向GPU体系特点，自动分析和实施神经网络计算，既是语言也是编译器。



与深度学习编译器相比，深度学习编译器TVM、XLA能实现从模型到硬件端到端的优化（深度学习模型Model → 计算图Graph → 算子Kernel → 部署Device），但多数情况下性能会差于供应商算子库，而Triton则通过提供领域特定的语言和编译器，面向底层的GPU kernel开发和编译优化问题，使得开发者能够以更高的抽象层次编写高效的GPU Kernel，从而提升性能。



安装

pip 安装

代码块

```
1 pip install triton
```

源码安装

1. 下载Triton源码

代码块

```
1 git clone https://github.com/openai/triton.git
```

查看对应的LLVM版本

代码块 cat triton/cmake/llvm-hash.txt

2. 构建llvm

代码块

```
1 git clone https://github.com/llvm/llvm-project.git
2
3 cd llvm-project
4
5 git checkout xxxx
6
7 mkdir build && cd build
8 cmake -G Ninja .. llvm \
9   -DCMAKE_BUILD_TYPE=Debug \
10  -DLLVM_ENABLE_PROJECTS="mlir;llvm;lld" \
11  -DLLVM_TARGETS_TO_BUILD="host;NVPTX;AMDGPU" \
12  -DLLVM_ENABLE_ASSERTIONS=ON \
13  -DLLVM_INSTALL_UTILS=ON \
14  -DCMAKE_INSTALL_PREFIX=$HOME/workspace/llvm-install # install 位置可以改
15
16 ninja
17 ninja install
```

3. 构建triton

进入triton目录下，最新版本官方有个 `setup.py` 能自动下载依赖的东西，也不用构建llvm

代码块

```
1 pip install -r python/requirements.txt # build-time dependencies
2 pip install -e .
```

编译好后，产物在build下边的这个文件夹里，里边有很多有用的东西

```
(triton-learn) root@056c5aab901e:~/playground1/workspace/triton-learn/triton/build [main]
● # ls
  cmake.linux-x86_64-cpython-3.10
```

<https://mp.weixin.qq.com/s/PDf5E7yzhqc6CyakV3fi-g>

这个链接里有个老版本，可以分离构建，方便修改源码

kernel编写

Triton 编程模型

Triton编程是CTA级别的编程，SPMD 编程模型

一个triton的kernel表示的是一个CTA的计算逻辑，隐藏了每个warp，每个thread的工作，以及什么时候同步，什么时候用shared memory，CTA一下的细节和优化都由编译器完成，写kernel主要关心算法实现和CTA级别的任务划分。

Triton kernel示例

Vector Add

由于每个kernel表示的是一个CTA或者说是block的计算逻辑，可以首先把需要的数据进行划分

CTA0	CTA1	CTA2	CTA3	CTA4
------	------	------	------	------

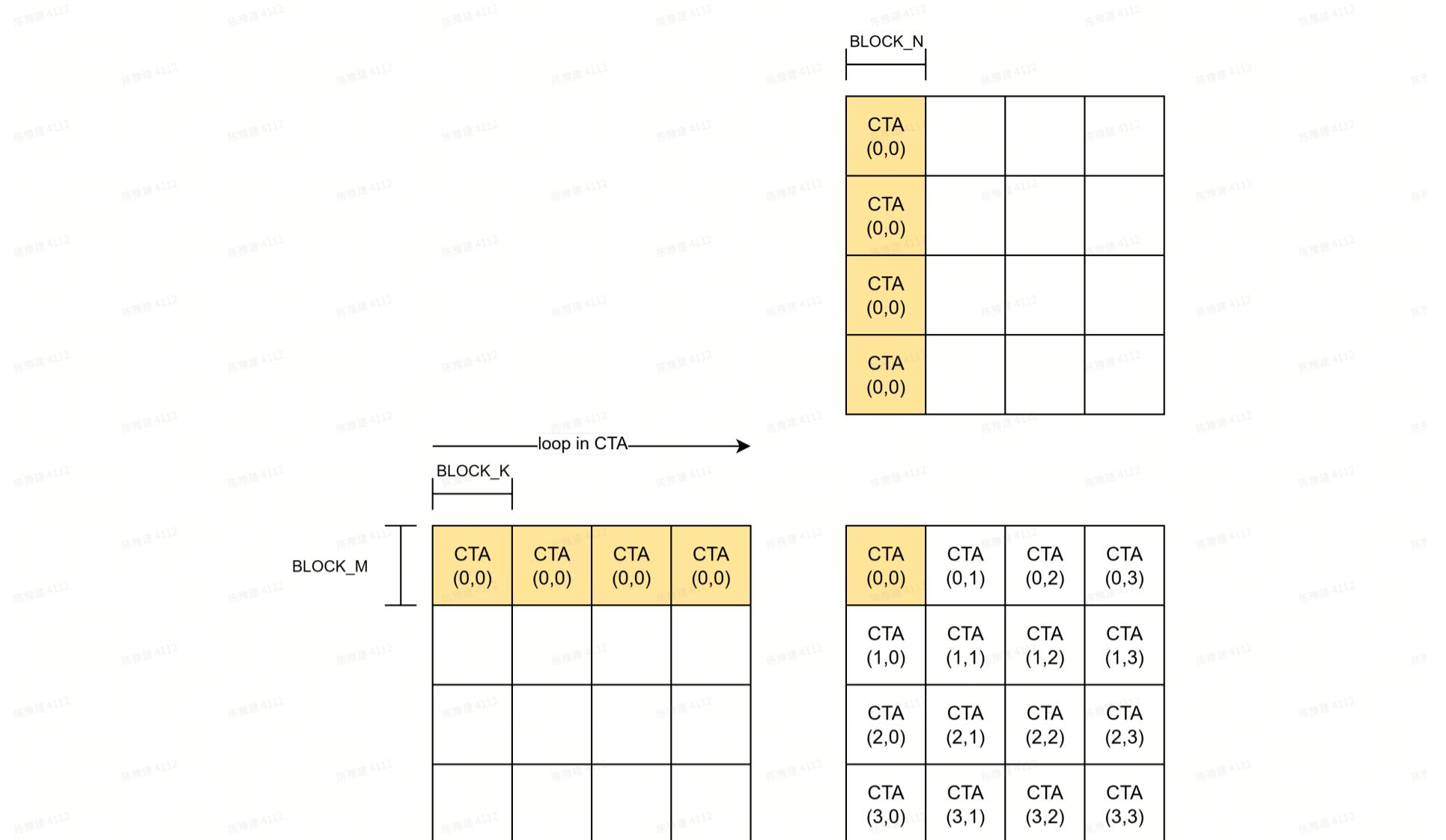
BLOCK_SIZE

代码块

```
1 @triton.jit
2 def add_kernel(x_ptr, y_ptr, out_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
3     pid = tl.program_id(axis=0)
4     block_start = pid * BLOCK_SIZE
5     offsets = block_start + tl.arange(0, BLOCK_SIZE)
6     mask = offsets < n_elements
7     x = tl.load(x_ptr + offsets, mask=mask)
8     y = tl.load(y_ptr + offsets, mask=mask)
9     output = x + y
10    tl.store(out_ptr + offsets, output, mask=mask)
11
12 def add(x: torch.Tensor, y: torch.Tensor):
13     output = torch.empty_like(x)
14     assert x.device == DEVICE and y.device == DEVICE and output.device == DEVICE
15
16     n_elements = output.numel()
17
18     def grid(meta): return (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
19     add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
20
21     return output
```

Matmul

行主序的(M, K)矩阵乘行主序的(K, N)矩阵，输出到(M, N)行主序矩阵



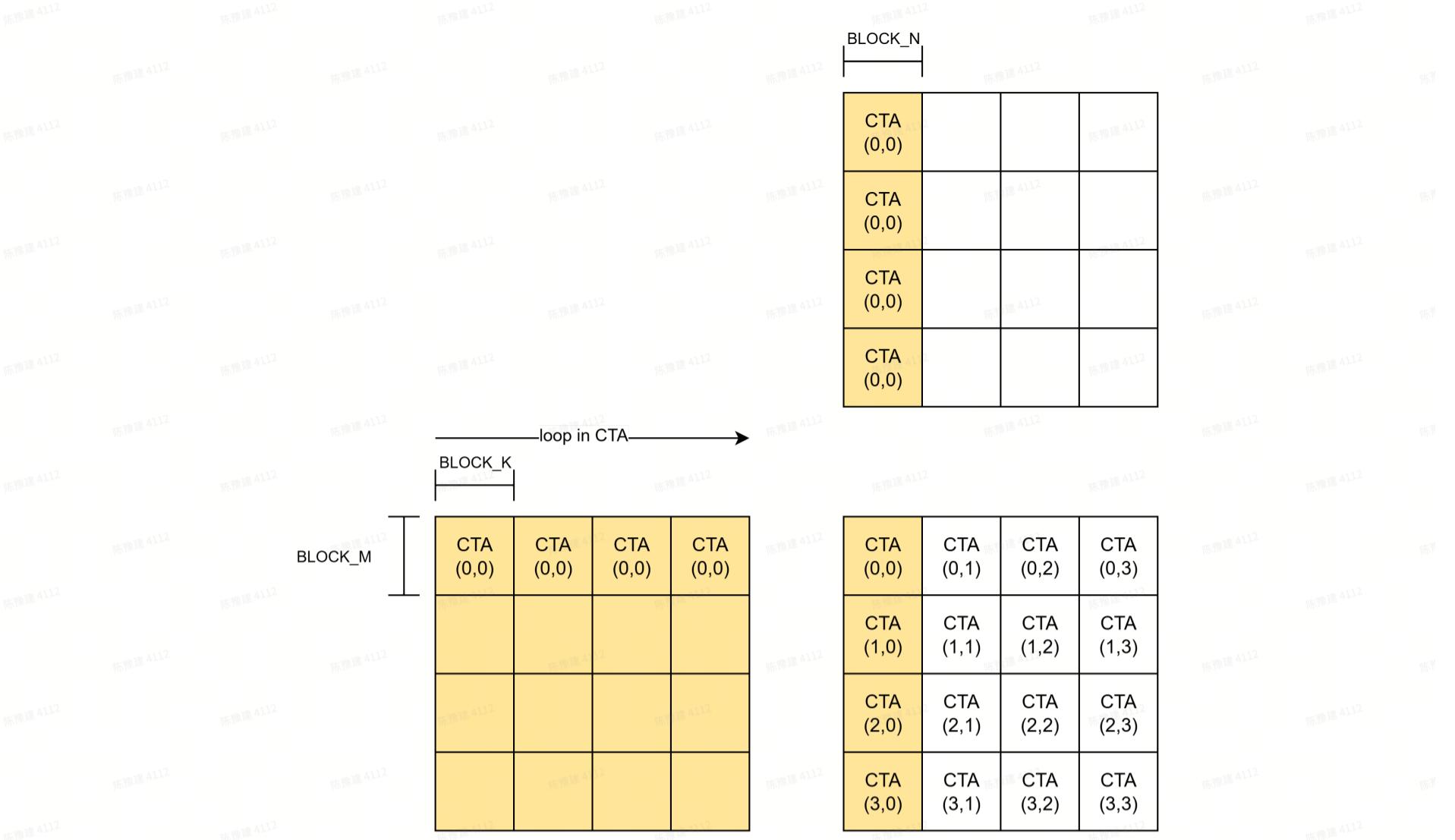
代码块

```

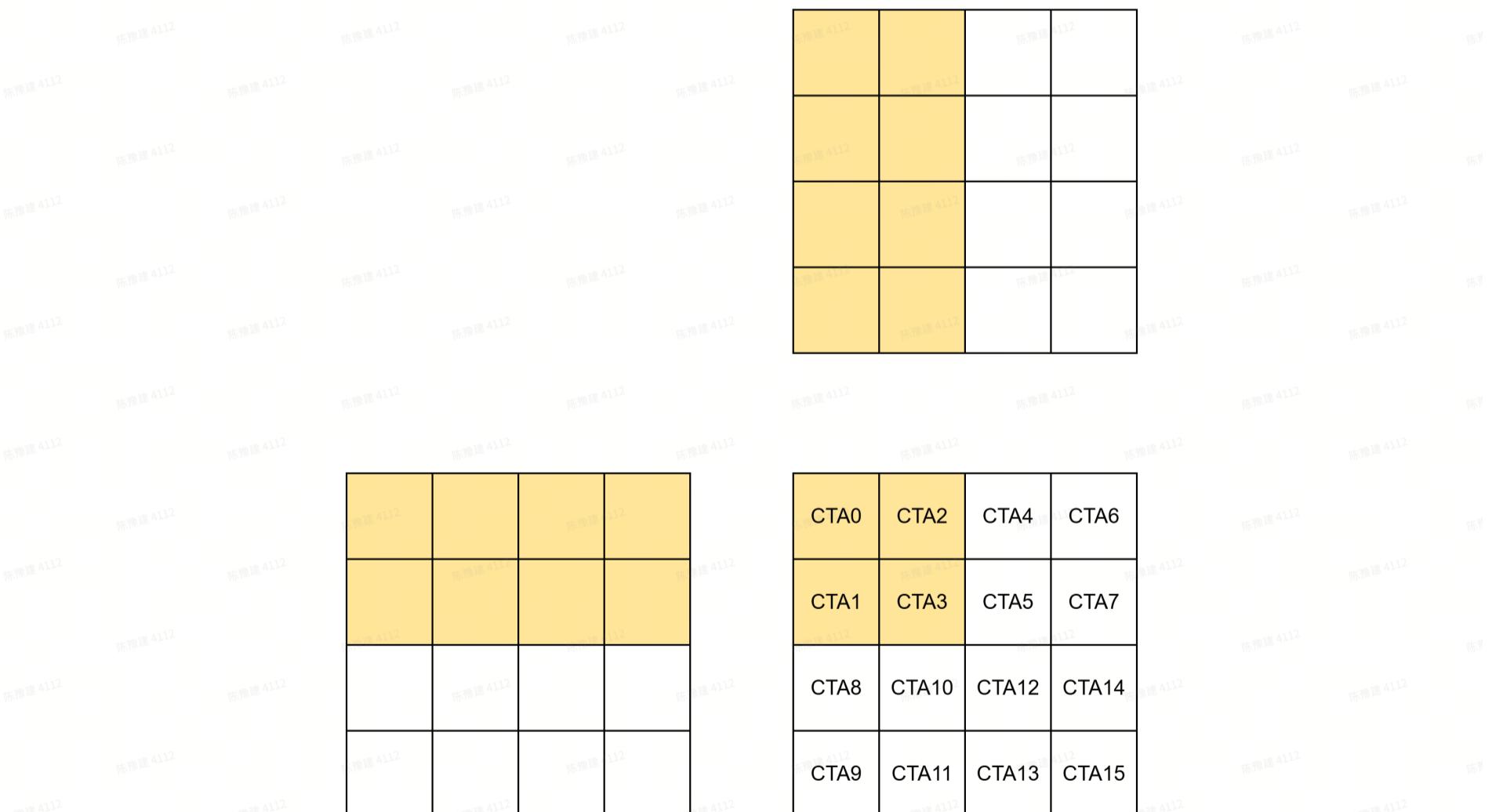
1  @triton.jit
2  def matmul_kernel1(A, B, C,
3      M, N, K,
4      BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr):
5      pid_m = tl.program_id(axis=0)
6      pid_n = tl.program_id(axis=1)
7      offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
8      offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
9      mask_m = offs_m < M
10     mask_n = offs_n < N
11
12     acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
13
14     for start_k in range(0, K, BLOCK_K):
15         offs_k = start_k + tl.arange(0, BLOCK_K)
16         a_ptrs = A + offs_m[:, None] * K + offs_k[None, :]
17         b_ptrs = B + offs_k[:, None] * N + offs_n[None, :]
18         mask_k = offs_k < K
19         mask_a = mask_m[:, None] & mask_k[None, :]
20         mask_b = mask_k[:, None] & mask_n[None, :]
21
22         a = tl.load(a_ptrs, mask=mask_a, other=0.0)
23         b = tl.load(b_ptrs, mask=mask_b, other=0.0)
24         acc += tl.dot(a, b)
25
26         c_ptrs = C + offs_m[:, None] * N + offs_n[None, :]
27         tl.store(c_ptrs, mask=mask_m[:, None] & mask_n[None, :], x=acc)

```

L2 Cache 优化



CTA编号和启动顺序有关。xyz编号的时候，x变动是最快的，也就是图中首先会计算C矩阵的一列中的四块，如果要计算图中C矩阵的四块，A矩阵要读16块，B矩阵读4块，C矩阵写4块，也就是写4块，读20块



改变读写顺序，变成上边的顺序，这个时候写C矩阵4块，只需要读A矩阵8块，读B矩阵8块。简单重排CTA就行了

CTA0	CTA2	CTA4	CTA6
CTA1	CTA3	CTA5	CTA7
CTA8	CTA10	CTA12	CTA14
CTA9	CTA11	CTA13	CTA15

GROUP0

GROUP1

代码块

```

1  @triton.jit
2  def matmul_kernel(a_ptr, b_ptr, c_ptr,
3                      M, N, K,
4                      stride_am, stride_ak,
5                      stride_bk, stride_bn,
6                      stride_cm, stride_cn,
7                      BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
8                      GROUP_SIZE_M: tl.constexpr, ACTIVATION: tl.constexpr):
9      pid = tl.program_id(axis=0)
10     num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
11     num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
12     num_pid_in_group = GROUP_SIZE_M * num_pid_n
13     group_id = pid // num_pid_in_group
14     first_pid_m = group_id * GROUP_SIZE_M
15     group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
16     pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
17     pid_n = (pid % num_pid_in_group) // group_size_m
18
19     offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
20     offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
21     offs_k = tl.arange(0, BLOCK_SIZE_K)
22     a_ptrs = a_ptr + (offs_am[:, None] * stride_am +
23                       offs_k[None, :] * stride_ak)
24     b_ptrs = b_ptr + (offs_k[:, None] * stride_bk +
25                       offs_bn[None, :] * stride_bn)
26
27     accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
28
29     for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
30         a = tl.load(a_ptrs, mask=offs_k[None, :], offset=K - k * BLOCK_SIZE_K, other=0.0)
31         b = tl.load(b_ptrs, mask=offs_k[:, None], offset=K - k * BLOCK_SIZE_K, other=0.0)
32         accumulator = tl.dot(a, b, accumulator)
33         a_ptrs += BLOCK_SIZE_K * stride_ak
34         b_ptrs += BLOCK_SIZE_K * stride_bk
35
36     if ACTIVATION == "leaky_relu":
37         accumulator = leaky_relu(accumulator)
38     c = accumulator.to(tl.float16)

```

```
41
42     offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
43     offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
44     c_ptrs = c_ptr + stride_cm * \
45               offs_cm[:, None] + stride_cn * offs_cn[None, :]
46     c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
47     tl.store(c_ptrs, c, mask=c_mask)
```

Autotune

triton jit function 的参数中有一部分是编译器常量参数，用 `tl constexpr` 标记，类似CPP中的模板参数，是直接编译进kernel中的，不会再出现在kernel参数列表里，像前边的 `BLOCK_SIZE_M` `BLOCK_SIZE_N` `BLOCK_SIZE_K` 等。还有部分不会出现在jit function定义中，例如 `num_warps`、`num_stages`，可以在调用jit function的时候传。这些参数组成了一组 `triton.Config` 用于Tunning.

代码块

```
1 def get_cuda_autotune_config():
2     return [
3         triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 256, 'BLOCK_SIZE_K': 64, 'GROUP_SIZE_M': 8}, num_stages=3,
4                         num_warps=8),
5         triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 256, 'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
6                         num_warps=4),
7         triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
8                         num_warps=4),
9         triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 64, 'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
10                         num_warps=4),
11         triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
12                         num_warps=4),
13         triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 32, 'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
14                         num_warps=4),
15         triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 32, 'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=5,
16                         num_warps=2),
17         triton.Config({'BLOCK_SIZE_M': 32, 'BLOCK_SIZE_N': 64, 'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=5,
18                         num_warps=2),
19         # Good config for fp8 inputs.
20         triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 256, 'BLOCK_SIZE_K': 128, 'GROUP_SIZE_M': 8}, num_stages=3,
21                         num_warps=8),
22         triton.Config({'BLOCK_SIZE_M': 256, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 128, 'GROUP_SIZE_M': 8}, num_stages=3,
23                         num_warps=8),
24         triton.Config({'BLOCK_SIZE_M': 256, 'BLOCK_SIZE_N': 64, 'BLOCK_SIZE_K': 128, 'GROUP_SIZE_M': 8}, num_stages=4,
25                         num_warps=4),
26         triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 256, 'BLOCK_SIZE_K': 128, 'GROUP_SIZE_M': 8}, num_stages=4,
27                         num_warps=4),
28         triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 128, 'GROUP_SIZE_M': 8}, num_stages=4,
29                         num_warps=4),
30         triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 64, 'BLOCK_SIZE_K': 64, 'GROUP_SIZE_M': 8}, num_stages=4,
```

```

31             num_warps=4),
32         triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 64, 'GROUP_SIZE_M': 8},
33         num_stages=4,
34             num_warps=4),
35         triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 32, 'BLOCK_SIZE_K': 64, 'GROUP_SIZE_M': 8},
36         num_stages=4,
37             num_warps=4)
38     ]
39
40     @triton.autotune(
41         configs=get_autotune_config(),
42         key=['M', 'N', 'K']
43     )

```

key 指明jit function的哪些运行时参数影响配置选取

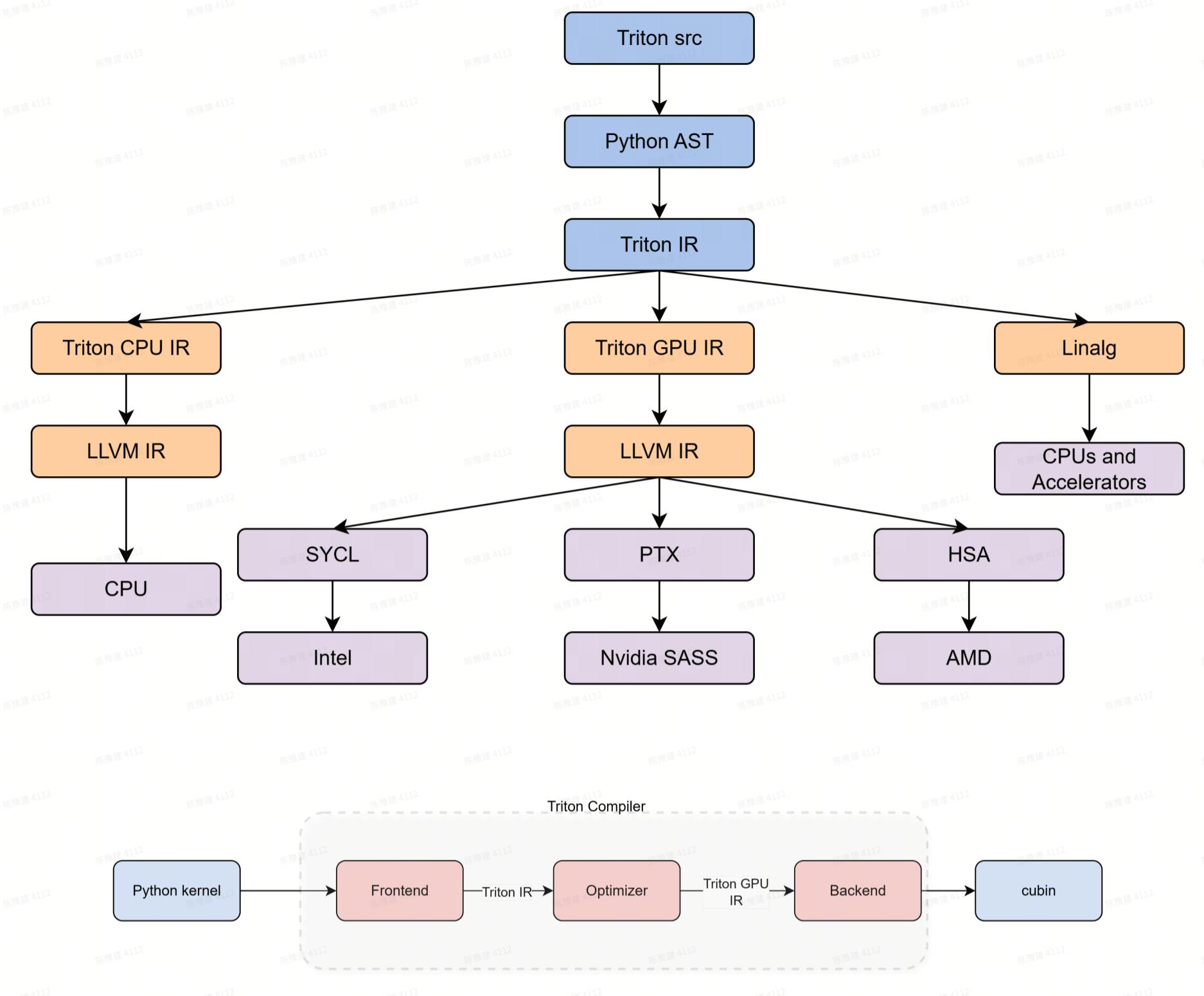
用autotune后，kernel启动的时候不用传自动配置的参数

测试性能

matmul-performance-fp16:					
	M	N	K	cUBLAS	Triton
0	256.0	256.0	256.0	3.640889	3.640889
1	384.0	384.0	384.0	10.053818	11.059200
2	512.0	512.0	512.0	21.845333	21.845333
3	640.0	640.0	640.0	36.571428	39.384616
4	768.0	768.0	768.0	63.195428	52.043293
5	896.0	896.0	896.0	70.246402	78.051553
6	1024.0	1024.0	1024.0	99.864382	80.659693
7	1152.0	1152.0	1152.0	129.825388	106.642284
8	1280.0	1280.0	1280.0	157.538463	132.129034
9	1408.0	1408.0	1408.0	151.438217	115.995231
10	1536.0	1536.0	1536.0	172.631417	144.446699
11	1664.0	1664.0	1664.0	181.796207	157.875646
12	1792.0	1792.0	1792.0	172.914215	184.252856
13	1920.0	1920.0	1920.0	206.328356	148.645157
14	2048.0	2048.0	2048.0	226.719125	169.466833
15	2176.0	2176.0	2176.0	223.596085	179.675426
16	2304.0	2304.0	2304.0	246.266731	202.439587
17	2432.0	2432.0	2432.0	216.111261	187.296418
18	2560.0	2560.0	2560.0	239.182490	206.088047
19	2688.0	2688.0	2688.0	211.916520	176.432825
20	2816.0	2816.0	2816.0	229.548456	191.290382
21	2944.0	2944.0	2944.0	244.294279	188.060491
22	3072.0	3072.0	3072.0	229.243346	202.225366
23	3200.0	3200.0	3200.0	240.601514	214.765101
24	3328.0	3328.0	3328.0	230.004143	195.628517
25	3456.0	3456.0	3456.0	246.549147	205.667272
26	3584.0	3584.0	3584.0	245.670475	205.286289
27	3712.0	3712.0	3712.0	234.500800	216.228019
28	3840.0	3840.0	3840.0	237.832263	199.985526
29	3968.0	3968.0	3968.0	233.762322	202.026384
30	4096.0	4096.0	4096.0	248.091925	193.397306

Triton 编译器

详细的内容可以看这个文档：<https://opendeep.wiki/triton-lang/triton/triton-compiler>



浅析下降过程

以Nvidia下的 `vector_add` 为例，简单探索一下下降的过程

AST 获取并转换为ttir

调用Triton Kernel (`python/tutorials/01-vector-add.py`) : `add(x, y)`

↓

JIT调用编译器(`triton/runtime/jit.py`): `kernel = self.compile()`

↓

调用`make_ir(triton/compiler/compiler.py)`: `src.make_ir()`

↓

调用`ast去生成ttir(triton/compiler/compiler.py)`: `ast_to_ttir()`, 逻辑位于`triton/compiler/code_generator.py`文件中, 具体地进行CodeGen的遍历完成Triton Python到Triton IR (ttir) 的转换

代码块

```

1 #loc = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":9:0)
2 #loc15 = loc("x_ptr"(#loc))
3 #loc16 = loc("y_ptr"(#loc))
4 #loc17 = loc("out_ptr"(#loc))
  
```

```

5 #loc18 = loc("n_elements"(#loc))
6 module {
7   tt.func public @add_kernel(%x_ptr: !tt.ptr<f32> {tt.divisibility = 16 : i32} loc("x_ptr"(#loc)),
8   %y_ptr: !tt.ptr<f32> {tt.divisibility = 16 : i32} loc("y_ptr"(#loc)), %out_ptr: !tt.ptr<f32>
9   {tt.divisibility = 16 : i32} loc("out_ptr"(#loc)), %n_elements: i32 {tt.divisibility = 16 : i32}
10  loc("n_elements"(#loc))) attributes {noinline = false} {
11    %c1024_i32 = arith.constant 1024 : i32 loc(#loc1)
12    %pid = tt.get_program_id x : i32 loc(#loc19)
13    %block_start = arith.muli %pid, %c1024_i32 : i32 loc(#loc20)
14    %offsets = tt.make_range {end = 1024 : i32, start = 0 : i32} : tensor<1024xi32> loc(#loc21)
15    %offsets_0 = tt.splat %block_start : i32 -> tensor<1024xi32> loc(#loc22)
16    %offsets_1 = arith.addi %offsets_0, %offsets : tensor<1024xi32> loc(#loc22)
17    %mask = tt.splat %n_elements : i32 -> tensor<1024xi32> loc(#loc23)
18    %mask_2 = arith.cmpi slt, %offsets_1, %mask : tensor<1024xi32> loc(#loc23)
19    %x = tt.splat %x_ptr : !tt.ptr<f32> -> tensor<1024x!tt.ptr<f32>> loc(#loc24)
20    %x_3 = tt.addptr %x, %offsets_1 : tensor<1024x!tt.ptr<f32>>, tensor<1024xi32> loc(#loc24)
21    %x_4 = tt.load %x_3, %mask_2 : tensor<1024x!tt.ptr<f32>> loc(#loc25)
22    %y = tt.splat %y_ptr : !tt.ptr<f32> -> tensor<1024x!tt.ptr<f32>> loc(#loc26)
23    %y_5 = tt.addptr %y, %offsets_1 : tensor<1024x!tt.ptr<f32>>, tensor<1024xi32> loc(#loc26)
24    %y_6 = tt.load %y_5, %mask_2 : tensor<1024x!tt.ptr<f32>> loc(#loc27)
25    %output = arith.addf %x_4, %y_6 : tensor<1024xf32> loc(#loc28)
26    %0 = tt.splat %out_ptr : !tt.ptr<f32> -> tensor<1024x!tt.ptr<f32>> loc(#loc12)
27    %1 = tt.addptr %0, %offsets_1 : tensor<1024x!tt.ptr<f32>>, tensor<1024xi32> loc(#loc12)
28    tt.store %1, %output, %mask_2 : tensor<1024x!tt.ptr<f32>> loc(#loc13)
29    tt.return loc(#loc14)
30  } loc(#loc)
31  #loc1 = loc(unknown)
32  #loc2 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":10:24)
33  #loc3 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":11:24)
34  #loc4 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":12:41)
35  #loc5 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":12:28)
36  #loc6 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":13:21)
37  #loc7 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":14:24)
38  #loc8 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":14:16)
39  #loc9 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":15:24)
40  #loc10 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":15:16)
41  #loc11 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":16:17)
42  #loc12 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":17:23)
43  #loc13 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":17:32)
44  #loc14 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":17:4)
45  #loc19 = loc("pid"(#loc2))
46  #loc20 = loc("block_start"(#loc3))
47  #loc21 = loc("offsets"(#loc4))
48  #loc22 = loc("offsets"(#loc5))
49  #loc23 = loc("mask"(#loc6))
50  #loc24 = loc("x"(#loc7))
51  #loc25 = loc("x"(#loc8))
52  #loc26 = loc("y"(#loc9))
53  #loc27 = loc("y"(#loc10))
54  #loc28 = loc("output"(#loc11))

```

ttir下降到ttgir

Backend 系统

Triton 在高层提供了一个 `compile` (src, target, options) 这样的统一编译函数(在 `python/triton/compiler/compiler.py` 文件中)。但具体如何把 `ttir` 转成可执行文件 (PTX / Cubin / GCN / etc.) 则由各个后端模块在 `backends` 目录下完成, 如英伟达 GPU 的后端实现目录 `:third_party/nvidia/backend`。

`python/triton/backends` 目录下的 `init.py`、`compiler.py` (BaseBackend 类)、`driver.py` (DriverBase 类) 代码实现了跨硬件平台支持，提供了一套子类，定义了一系列抽象方法 `abstractmethod` 统一接口，用于将 Triton 的中间表示 (IR) 或 AST 编译成不同 GPU 平台所需的可执行内核。

Triton 定义了一个抽象后端 `BaseBackend` 类，定义所有硬件后端必须实现的接口：

```
代码块
1
2 class BaseBackend(metaclass=ABCMeta):
3     supports_native_tensor_specialization = True
4
5     def __init__(self, target: GPUTarget) -> None:
6         self.target = target
7         assert self.supports_target(target)
8
9     @staticmethod
10    @abstractmethod
11    def supports_target(target: GPUTarget):
12        raise NotImplementedError
13
14    @abstractmethod
15    def hash(self) -> str:
16        """Returns a unique identifier for this backend"""
17        raise NotImplementedError
18
19    @abstractmethod
20    def parse_options(self, options: dict) -> object:
21        """
22            Converts an `options` dictionary into an arbitrary object and returns it.
23            This function may contain target-specific heuristics and check the legality of the provided
24            options
25        """
26        raise NotImplementedError
27
28    @abstractmethod
29    def add_stages(self, stages: dict, options: object) -> None:
30        """
31            Populates `stages` dictionary with entries of the form:
32            ir_name [str] => Function[(src: str, metadata: dict) -> str|bytes]
33            The value of each entry may populate a `metadata` dictionary.
34            Stages will be run sequentially (in insertion order) and can communicate using `metadata`.
35            All stages are expected to return a `str` object, except for the last stage which returns
36            a `bytes` object for execution by the launcher.
37        """
38        raise NotImplementedError
39
40    @abstractmethod
41    def load_dialects(self, context):
42        """
43            Load additional MLIR dialects into the provided `context`
44        """
45        raise NotImplementedError
46
47    @abstractmethod
48    def get_module_map(self) -> Dict[str, ModuleType]:
49        """
50            Return a map of interface modules to their device-specific implementations
51        """
52        raise NotImplementedError
53
54    @staticmethod
```

```

54     def parse_attr(desc):
55         assert isinstance(desc, str)
56         ret = []
57         if "D" in desc:
58             ret += [{"tt.divisibility": 16}]
59         return ret
60
61     @staticmethod
62     def get_int_specialization(arg, **kwargs):
63         if arg % 16 == 0 and kwargs.get("align", False):
64             return "D"
65         return ""
66
67     @staticmethod
68     def get_tensor_specialization(arg, **kwargs):
69         if arg.data_ptr() % 16 == 0 and kwargs.get("align", False):
70             return "D"
71         return ""
72

```

Nvidia 平台下降包含了下边几个stage

```

def add_stages(self, stages, options, language):
    capability = self._parse_arch(options.arch)
    if language == Language.TRITON:
        stages["ttir"] = lambda src, metadata: self.make_ttir(src, metadata, options, capability)
        stages["ttgir"] = lambda src, metadata: self.make_ttgir(src, metadata, options, capability)
    elif language == Language.GLUON:
        stages["ttgir"] = lambda src, metadata: self.gluon_to_ttgir(src, metadata, options, capability)
    stages["llir"] = lambda src, metadata: self.make_llir(src, metadata, options, capability)
    stages["ptx"] = lambda src, metadata: self.make_ptx(src, metadata, options, self.target.arch)
    stages["cubin"] = lambda src, metadata: self.make_cubin(src, metadata, options, self.target.arch)

```

其中ttir下降到ttgir在stages['ttgir']中，具体其中的一个pass

```

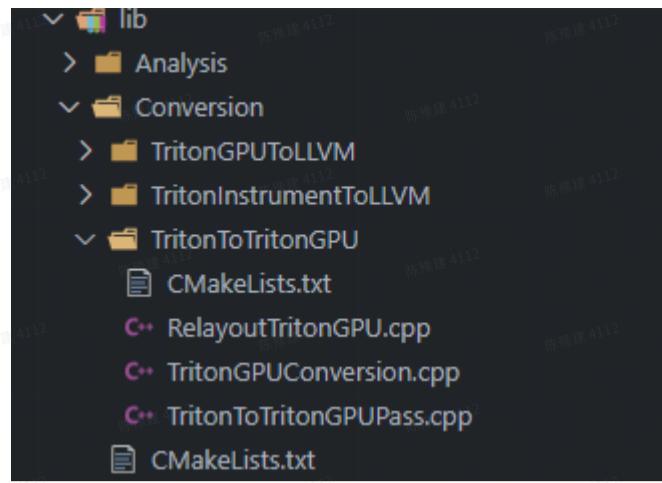
    return mod

    @staticmethod
    def make_ttgir(mod, metadata, opt, capability):
        # Set maxnreg on all kernels, if it was provided.
        if opt.maxnreg is not None:
            mod.set_attr("ttg.maxnreg", ir.builder(mod.context).get_int32_attr(opt.maxnreg))

        cluster_info = nvidia.ClusterInfo()
        if opt.cluster_dims is not None:
            cluster_info.clusterDimX = opt.cluster_dims[0]
            cluster_info.clusterDimY = opt.cluster_dims[1]
            cluster_info.clusterDimZ = opt.cluster_dims[2]
        pm = ir.pass_manager(mod.context)
        dump_enabled = pm.enable_debug()
        passes.tтир.add_convert_to_ttgpuir(pm, f"cuda:{capability}", opt.num_warps, 32, opt.num_ctas)
        # optimize TTGIR

```

这个pass对应的CPP实现在 `triton/lib/Conversion/TritonToTritonGPU` 目录下



下降后的ttgir如下

代码块

```
1 #blocked = #ttg.blocked<{sizePerThread = [4], threadsPerWarp = [32], warpsPerCTA = [4], order = [0]}>
2 #loc = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":9:0)
3 #loc15 = loc("x_ptr"(#loc))
4 #loc16 = loc("y_ptr"(#loc))
5 #loc17 = loc("out_ptr"(#loc))
6 #loc18 = loc("n_elements"(#loc))
7 module attributes {"ttg.num-ctas" = 1 : i32, "ttg.num-warps" = 4 : i32, ttg.target = "cuda:80",
"ttg.threads-per-warp" = 32 : i32} {
8     tt.func public @add_kernel(%x_ptr: !tt.ptr<f32> {tt.divisibility = 16 : i32} loc("x_ptr"(#loc)),
%y_ptr: !tt.ptr<f32> {tt.divisibility = 16 : i32} loc("y_ptr"(#loc)), %out_ptr: !tt.ptr<f32>
{tt.divisibility = 16 : i32} loc("out_ptr"(#loc)), %n_elements: i32 {tt.divisibility = 16 : i32}
loc("n_elements"(#loc))) attributes {noinline = false} {
9         %c1024_i32 = arith.constant 1024 : i32 loc(#loc1)
10        %pid = tt.get_program_id x : i32 loc(#loc19)
11        %block_start = arith.muli %pid, %c1024_i32 : i32 loc(#loc20)
12        %offsets = tt.make_range {end = 1024 : i32, start = 0 : i32} : tensor<1024xi32, #blocked> loc(#loc21)
13        %offsets_0 = tt.splat %block_start : i32 -> tensor<1024xi32, #blocked> loc(#loc22)
14        %offsets_1 = arith.addi %offsets_0, %offsets : tensor<1024xi32, #blocked> loc(#loc22)
15        %mask = tt.splat %n_elements : i32 -> tensor<1024xi32, #blocked> loc(#loc23)
16        %mask_2 = arith.cmpi slt, %offsets_1, %mask : tensor<1024xi32, #blocked> loc(#loc23)
17        %x = tt.splat %x_ptr : !tt.ptr<f32> -> tensor<1024x!tt.ptr<f32>, #blocked> loc(#loc24)
18        %x_3 = tt.addptr %x, %offsets_1 : tensor<1024x!tt.ptr<f32>, #blocked>, tensor<1024xi32, #blocked>
loc(#loc24)
19        %x_4 = tt.load %x_3, %mask_2 : tensor<1024x!tt.ptr<f32>, #blocked> loc(#loc25)
20        %y = tt.splat %y_ptr : !tt.ptr<f32> -> tensor<1024x!tt.ptr<f32>, #blocked> loc(#loc26)
21        %y_5 = tt.addptr %y, %offsets_1 : tensor<1024x!tt.ptr<f32>, #blocked>, tensor<1024xi32, #blocked>
loc(#loc26)
22        %y_6 = tt.load %y_5, %mask_2 : tensor<1024x!tt.ptr<f32>, #blocked> loc(#loc27)
23        %output = arith.addf %x_4, %y_6 : tensor<1024xf32, #blocked> loc(#loc28)
24        %0 = tt.splat %out_ptr : !tt.ptr<f32> -> tensor<1024x!tt.ptr<f32>, #blocked> loc(#loc12)
25        %1 = tt.addptr %0, %offsets_1 : tensor<1024x!tt.ptr<f32>, #blocked>, tensor<1024xi32, #blocked>
loc(#loc12)
26        tt.store %1, %output, %mask_2 : tensor<1024x!tt.ptr<f32>, #blocked> loc(#loc13)
27        tt.return loc(#loc14)
28    } loc(#loc)
29 } loc(#loc)
30 #loc1 = loc(unknown)
31 #loc2 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":10:24)
32 #loc3 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":11:24)
33 #loc4 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":12:41)
34 #loc5 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":12:28)
35 #loc6 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":13:21)
36 #loc7 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":14:24)
37 #loc8 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":14:16)
38 #loc9 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":15:24)
39 #loc10 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":15:16)
```

```
40 #loc11 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":16:17)
41 #loc12 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":17:23)
42 #loc13 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":17:32)
43 #loc14 = loc("/root/playground1/workspace/triton-learn/learn-demo/vector_add.py":17:4)
44 #loc19 = loc("pid"(#loc2))
45 #loc20 = loc("block_start"(#loc3))
46 #loc21 = loc("offsets"(#loc4))
47 #loc22 = loc("offsets"(#loc5))
48 #loc23 = loc("mask"(#loc6))
49 #loc24 = loc("x"(#loc7))
50 #loc25 = loc("x"(#loc8))
51 #loc26 = loc("y"(#loc9))
52 #loc27 = loc("y"(#loc10))
53 #loc28 = loc("output"(#loc11))
```

可以发现，ttgir添加了block内部相关的信息，例如layout相关信息。

ttgir 下降到 llir

ttgir 下降到 llir 主要由 `make_llir` 实现。这个转换流程包含了两个步骤，首先是把 ttgir 优化转换成 MLIR 表示的 LLVM IR，再进一步转换为实际的 LLVM 模块并返回其字符串形式。

```
def make_llir(self, src, metadata, options, capability):
    ptx_version = get_ptx_version_from_options(options, self.target.arch)

    mod = src
    # TritonGPU -> LLVM-IR (MLIR)
    pm = ir.pass_manager(mod.context)
    pm.enable_debug()

    passes.ttgpuir.add_combine_tensor_select_and_if(pm)
    passes.ttgpuir.add_allocate_warp_groups(pm)
    passes.convert.add_scf_to_cf(pm)
    passes.gluon.add_inliner(pm)
    nvidia.passes.ttgpuir.add_allocate_shared_memory_nv(pm, capability, ptx_version)
    nvidia.passes.ttnvgpuir.add_allocate_tensor_memory(pm)
    if knobs.compilation.enable_experimental_consan:
        # Call ConcurrencySanitizerPass here, before allocating global scratch memory but after allocating tensor and shared
        passes.ttgpuir.add_concurrency_sanitizer(pm)
    passes.ttgpuir.add_allocate_global_scratch_memory(pm)
    nvidia.passes.ttnvgpuir.add_proxy_fence_insertion(pm, capability)
    # instrumentation point here so we can override IRs above (e.g., ttir and ttgir)
    if CUDABackend.instrumentation:
        CUDABackend.instrumentation.patch("ttgpuir_to_llvmir", pm, mod.context)
    nvidia.passes.ttgpuir.add_to_llvmir(pm, capability, ptx_version)
    passes.common.add_canonicalizer(pm)
    passes.common.add_cse(pm)
    nvidia.passes.ttnvgpuir.add_nvgpu_to_llvm(pm)
    nvidia.passes.ttnvgpuir.add_warp_specialize_to_llvm(pm)
    passes.common.add_canonicalizer(pm)
    passes.common.add_cse(pm)
    passes.common.add_symbol_dce(pm)
    passes.convert.add_nvvm_to_llvm(pm)
    if not knobs.compilation.disable_line_info:
        passes.llvmir.add_di_scope(pm)
    if CUDABackend.instrumentation:
        CUDABackend.instrumentation.patch("llvmir_to_llvm", pm, mod.context)

    pm.run(mod)
    # LLVM-IR (MLIR) -> LLVM-IR (LLVM)
    llvm.init_targets()
    context = llvm.context()
    if knobs.compilation.enable_asan:
        raise RuntimeError(
            "Address Sanitizer Error: Address sanitizer is currently only supported on the AMD backend")
```

llir 下降到 PTX

这一步是在 `make_ptx` 中完成的，将此前编译产生的 LLVM IR (在字符串形式) 翻译成 **PTX 汇编代码**，并对生成的 PTX 进行一定的后处理 (如插入正确的 .version、去除 debug 标记等)。它主要依赖 `llvmlite` 或 Triton 自带的 `llvm.translate_to_asm` 接

口，将 LLVM IR 转换为可在 NVIDIA GPU 上使用的 PTX 代码。

```
def make_ptx(self, src, metadata, opt, capability):
    ptx_version = get_ptx_version_from_options(opt, self.target.arch)

    triple = 'nvptx64-nvidia-cuda'
    proc = sm_arch_from_capability(capability)
    features = get_features(opt, self.target.arch)
    ret = llvm.translate_to_asm(src, triple, proc, features, [], opt.enable_fp_fusion, False)
    # Find kernel names (there should only be one)
    names = re.findall(r".visible .entry ([a-zA-Z_][a-zA-Z0-9_]*)", ret)
    assert len(names) == 1
    metadata["name"] = names[0]
    # post-process
    ptx_version = f'{ptx_version//10}.{ptx_version%10}'
    ret = re.sub(r'\.version \d+\.\d+', f'.version {ptx_version}', ret, flags=re.MULTILINE)
    ret = re.sub(r'\.target sm_\d+', f'.target sm_{capability}', ret, flags=re.MULTILINE)
    # Remove the debug flag that prevents ptxas from optimizing the code
    ret = re.sub(r"\s*debug|debug\s*", "", ret)
    if knobs.nvidia.dump_nvptx:
        print("// -----// NVPTX Dump //----- //")
        print(ret)
    return ret
```

最后通过 `make_cubin` 得到NVIDIA GPU 的二进制可执行文件。

ttir 优化 pass

代码块

```
1 def make_ttir(mod, metadata, opt, capability):
2     pm = ir.pass_manager(mod.context)
3     pm.enable_debug()
4     passes.common.add_inliner(pm)
5     passes.ttir.add_rewrite_tensor_pointer(pm)
6     if capability // 10 < 9:
7         passes.ttir.add_rewrite_tensor_descriptor_to_pointer(pm)
8     passes.common.add_canonicalizer(pm)
9     passes.ttir.add_combine(pm)
10    passes.ttir.add_reorder_broadcast(pm)
11    passes.common.add_cse(pm)
12    passes.common.add_symbol_dce(pm)
13    passes.ttir.add_loop_unroll(pm)
14    pm.run(mod)
15    return mod
```

`passes.common` 开头的是MLIR通用的优化pass, `passed.ttir` 开头的是 Triton IR层次的优化pass.

common

inliner pass 内联优化

inliner pass 能将 function call 给 inline 到主 kernel 中来(不用散落在不同的 func 中)，方便后序代码的优化。

canonicalizer 规范化操作

规范化pass的功能是将代码转换为一种规范化的形式，消除冗余和不必要的复杂结构，以便为后续的优化 Pass 提供便利。

canonicalize 的准则并不是性能优化，而是为了后续 ir 分析更加高效。所以经常用 canonicalize 把一些 constant 的 arith 计算给 fold 掉，减短一些 ir 链。

cse pass 公共子表达式消除

CSE全称为Common Subexpression Elimination，功能是识别和消除重复计算的表达式，优化程序的运行时间和空间。

symbol_dce pass 死代码消除

死代码消除作用是移除中间表示中没有使用的符号定义。

licm pass 循环不变量外提

licm全称为Loop Invariant Code Motion，功能是循环内部不随循环变化的代码提到循环外面。新版本triton这个pass好像去掉了

ttir

rewrite_tensor_pointer pass

重写张量指针。识别并重写那些涉及张量指针 (tensor pointers) 的操作，以提高性能和内存使用效率。将带有 tensor pointers(由 `tt.make_tensor_ptr` 和 `tt.advance`) 的 load、store 以及可能用到 tensor pointer 的 loop 重写为特定的模式。

重写后，方便之后分析 `AxisInfo`。

combine pass

Op融合优化。对一些Op操作进行融合。pass中定义了`CombineDotAddIPattern`、`CombineDotAddFPattern`、`CombineDotAddIRevPattern`、`CombineDotAddFRevPattern`、`CombineSelectMaskedLoadPattern`、`CombineAddPtrPattern`、`CombineBroadcastMulReducePattern`几种融合规则。

reorder_broadcast pass

功能是通过调整广播操作和其他Op间的调用顺序关系，来优化广播操作的使用，提升程序性能，该pass中定义了 `elementwise(splat(a), splat(b), ...) => splat(elementwise(a, b, ...))`、`elementwise(broadcast(a)) => broadcast(elementwise(a))` 等几种模式。

loop_unroll pass

循环展开。根据load操作中指定的循环因子对循环进行相应次数的展开。

详细细节可以看这篇文章：<https://zhuanlan.zhihu.com/p/718936671>，<https://tfruan2000.github.io/posts/triton-source-code-1/>

ttgir 优化pass

代码块

```
1 def make_ttgir(mod, metadata, opt, capability):
2     # Set maxnreg on all kernels, if it was provided.
3     if opt.maxnreg is not None:
4         mod.set_attr("ttg.maxnreg", ir.builder(mod.context).get_int32_attr(opt.maxnreg))
5
6     cluster_info = nvidia.ClusterInfo()
7     if opt.cluster_dims is not None:
8         cluster_info.clusterDimX = opt.cluster_dims[0]
9         cluster_info.clusterDimY = opt.cluster_dims[1]
10        cluster_info.clusterDimZ = opt.cluster_dims[2]
11    pm = ir.pass_manager(mod.context)
12    dump_enabled = pm.enable_debug()
13    passes.tтир.add_convert_to_ttgpuir(pm, f"cuda:{capability}", opt.num_warps, 32, opt.num_ctas)
```

```
14     # optimize TTGIR
15     passes.ttgpuir.add_coalesce(pm)
16     if capability // 10 >= 8:
17         passes.ttgpuir.add_f32_dot_tc(pm)
18     # TODO(Qingyi): Move PlanCTAPass to the front of CoalescePass
19     nvidia.passes.ttnvgpuir.add_plan_cta(pm, cluster_info)
20     passes.ttgpuir.add_remove_layout_conversions(pm)
21     passes.ttgpuir.add_optimize_thread_locality(pm)
22     passes.ttgpuir.add_accelerate_matmul(pm)
23     passes.ttgpuir.add_remove_layout_conversions(pm)
24     passes.ttgpuir.add_optimize_dot_operands(pm, capability >= 80)
25     nvidia.passes.ttnvgpuir.add_optimize_descriptor_encoding(pm)
26     passes.ttir.add_loop_aware_cse(pm)
27     if capability // 10 in [8, 9]:
28         passes.ttgpuir.add_fuse_nested_loops(pm)
29         passes.common.add_canonicalizer(pm)
30         passes.ttir.add_triton_licm(pm)
31         passes.common.add_canonicalizer(pm)
32         passes.ttgpuir.add_combine_tensor_select_and_if(pm)
33         nvidia.passes.hopper.add_hopper_warpsspec(pm, opt.num_stages, dump_enabled)
34         passes.ttgpuir.add_assign_latencies(pm, opt.num_stages)
35         passes.ttgpuir.add_schedule_loops(pm)
36         passes.ttgpuir.add_pipeline(pm, opt.num_stages, dump_enabled)
37     elif capability // 10 >= 10:
38         passes.ttgpuir.add_fuse_nested_loops(pm)
39         passes.common.add_canonicalizer(pm)
40         passes.ttir.add_triton_licm(pm)
41         passes.ttgpuir.add_optimize_accumulator_init(pm)
42         passes.ttgpuir.add_hoist_tmem_alloc(pm, False)
43         nvidia.passes.ttnvgpuir.add_promote_lhs_to_tmem(pm)
44         passes.ttgpuir.add_assign_latencies(pm, opt.num_stages)
45         passes.ttgpuir.add_schedule_loops(pm)
46         passes.ttgpuir.add_warp_specialize(pm, opt.num_stages)
47         passes.ttgpuir.add_pipeline(pm, opt.num_stages, dump_enabled)
48         passes.ttgpuir.add_combine_tensor_select_and_if(pm)
49         # hoist again and allow hoisting out of if statements
50         passes.ttgpuir.add_hoist_tmem_alloc(pm, True)
51         nvidia.passes.ttnvgpuir.add_remove_tmem_tokens(pm)
52     else:
53         passes.ttir.add_triton_licm(pm)
54         passes.common.add_canonicalizer(pm)
55         passes.ttir.add_loop_aware_cse(pm)
56         passes.ttgpuir.add_prefetch(pm)
57         passes.ttgpuir.add_optimize_dot_operands(pm, capability >= 80)
58         passes.ttgpuir.add_coalesce_async_copy(pm)
59         nvidia.passes.ttnvgpuir.add_optimize_tmem_layouts(pm)
60         passes.ttgpuir.add_remove_layout_conversions(pm)
61         nvidia.passes.ttnvgpuir.add_interleave_tmem(pm)
62         passes.ttgpuir.add_reduce_data_duplication(pm)
63         passes.ttgpuir.add_reorder_instructions(pm)
64         passes.ttir.add_loop_aware_cse(pm)
65         passes.common.add_symbol_dce(pm)
66     if capability // 10 >= 9:
67         nvidia.passes.ttnvgpuir.add_tma_lowering(pm)
68         nvidia.passes.ttnvgpuir.add_fence_insertion(pm, capability)
69         nvidia.passes.ttnvgpuir.add_lower_mma(pm)
70         passes.common.add_sccp(pm)
71         passes.common.add_cse(pm)
72         passes.common.add_canonicalizer(pm)
73
74     pm.run(mod)
```

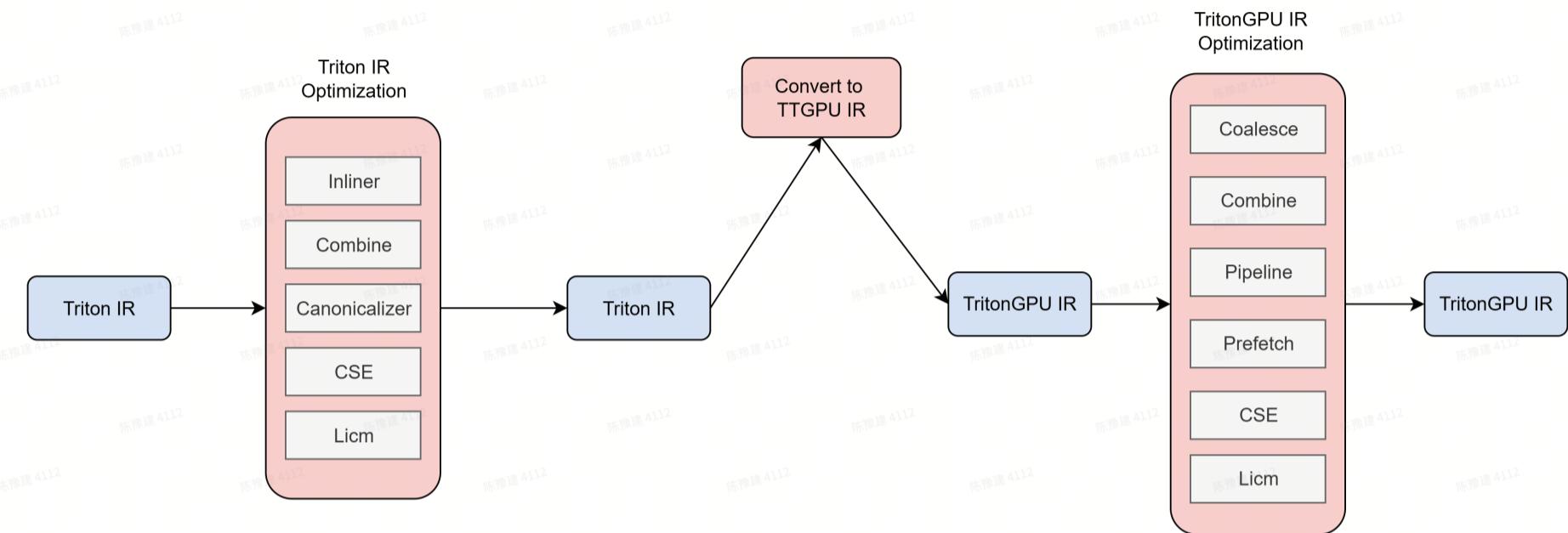
```

75     metadata["cluster_dims"] = (cluster_info.clusterDimX, cluster_info.clusterDimY,
76                                 cluster_info.clusterDimZ)
77     tensordesc_meta = mod.get_tensordesc_metadata()
78     metadata["tensordesc_meta"] = tensordesc_meta
    return mod

```

这个优化pass很多，会根据不同的架构设置不同的pass，详细还没研究。

Triton Layout



- Triton IR上的优化主要是计算本身，和硬件无关
- TritonGPU IR优化在计算本身优化外，新增GPU相关的优化，增加了TritonGPU 特有的Layout

GPU多层次架构

层级 / 模块	NVIDIA	AMD
顶层模块	GPC (Graphics Processing Cluster)	XCD (Accelerator Complex Die)
中间模块	TPC (Texture Processing Cluster)	Shader Engine (可选)
基本计算单元	SM (Streaming Multiprocessor)	CU (Compute Unit)
线程调度粒度	CTA (Cooperative Thread Array)	Workgroup (包含多个 frontend)
最小执行单元	warp (32 个 thread)	wavefront (64 个 workitem)
基本线程	thread	workitem
矩阵计算加速单元	Tensor Core	Matrix Core
内存层级	全局内存、L1/L2 Cache、共享内存、寄存器文件	全局内存、L1/L2 Cache、共享内存、寄存器文件

- 什么是CTA

- 并行线程执行 (PTX) 编程模型

PTX 编程模型是显式并行的：一个 PTX 程序定义了并行线程数组中某个线程的执行方式。

- 协作线程数组 (Cooperative Thread Array, 简称 CTA) 是一组并发或并行执行内核的线程构成的数组。
- CTA 内的线程可以相互通信。为了协调 CTA 内线程的通信，开发者可以指定同步点——线程会在这些同步点等待，直到 CTA 中的所有线程都到达该点。
- CTA 内的线程会以SIMT (单指令多线程) 的方式，按“线程束 (warp)”为组执行。一个 warp 是“来自单个 CTA 的线程的最大子集”，且该子集中的线程会同时执行相同的指令。warp 内的线程会被顺序编号。warp 的大小是一个与机器相关的常量，通常一个 warp 包含 32 个线程。

• 什么是CGA

CGA的概念是在Hopper(sm90)架构引入的

- 一组并发或并行运行的 CTA (协作线程数组)，这些 CTA 可通过共享内存相互同步与通信。执行中的 CTA 必须确保：在通过共享内存与对等 CTA 通信前，对等 CTA 的共享内存已存在；且在完成共享内存操作前，对等 CTA 尚未退出。
- Threads within the different CTAs in a cluster can synchronize and communicate with each other via shared memory. Cluster-wide barriers can be used to synchronize all the threads within the cluster. Each CTA in a cluster has a unique CTA identifier within its cluster (*cluster_ctaid*). Each cluster of CTAs has 1D, 2D or 3D shape specified by the parameter *cluster_nctaid*. Each CTA in the cluster also has a unique CTA identifier (*cluster_ctarank*) across all dimensions. The total number of CTAs across all the dimensions in the cluster is specified by *cluster_nctarank*. Threads may read and use these values through predefined, read-only special registers `%cluster_ctaid`, `%cluster_nctaid`, `%cluster_ctarank`, `%cluster_nctarank`.

Triton Layout概念

在Triton中，Layout定义了数据是如何被GPU线程处理的，不同的Layout可以看作是不同的映射函数，代表不同的访问模式。也就是说，Layout是用于确定各层级内存中的Tensor到线程Thread之间的映射关系，即Tensor中某个坐标的元素，被哪些GPU所拥有。

例如：

$L(0, 0) = \{0, 4\}$: Tensor[0, 0]被thread 0 和 thread 4 拥有

$L(0, 1) = \{1, 5\}$: Tensor[0, 1]被thread 1 和 thread 5 拥有

$L(1, 0) = \{2, 6\}$: Tensor[1, 0]被thread 2 和 thread 6 拥有

$L(1, 1) = \{3, 7\}$: Tensor[1, 1]被thread 3 和 thread 7 拥有

选择合理的Layout，可以优化数据的访问模式，减少线程间的竞争，提高并行计算的效率。

Triton中主要有两种Layout类型：`Distributed Layout` 和 `Shared Layout`

`Distributed Layout` 表示数据被分散到多个线程中

`Shared Layout` 表示数据被多个线程共享

Distributed Layout	Blocked Layout	映射函数会将特定的Tensor交给特定的Thread去处理，达到分配的效果
	MMA Layout	
	DotOperand Layout	
	Slice Layout	
Shared Layout	Shared Layout	映射函数被定义为任意Tensor->任意Thread

Blocked Layout

Block Layout 表示thread间平均分配workload的情况，每个线程拥有一块内存上连续的数据处理，用来描述一个warp如何映射到目标张量的连续区域，主要参数有：

- sizePerThread: 每个thread访问tensor子集的shape
- threadsPerWarp: 每个warp在不同维度上的线程数
- warpsPerCTA: 每个CTA中warp的shape
- order: 访问顺序，order=[1,0]为按行访问，order=[0,1]为按列访问
- CTALayout: 用来描述Tensor如何分布在一个CTA cluster(CGA)的多个CTA中
 - CTAsPerCGA: 代表一个Tensor分布在几个CTA中，以及CTA的shape
 - CTASplitNum: 代表Tensor每个维度切分成几片，每个CTA处理的子张量大小为tensor_shape / CTASplitNum

例如

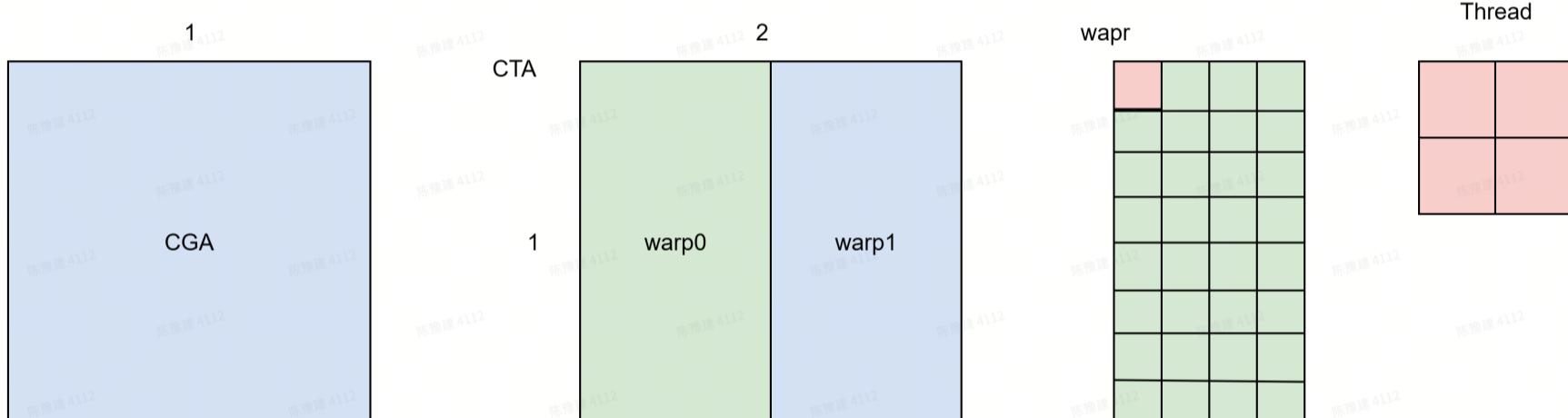
sizePerThread = {2, 2}

threadsPerWarp = {8, 4}

warpsPerCTA = {1, 2}

CTAsPerCGA = {1, 1}

CTASplitNum = {1, 1}



如果指定order = [1, 0] 即按行访问，一个16x16 fp16的Tensor

thread0															
0	0	1	1	2	2	3	3	32	32	33	33	34	34	35	35
0	0	1	1	2	2	3	3	32	32	33	33	34	34	35	35
4	4	5	5	6	6	7	7								
4	4	5	5	6	6	7	7								
28	28	29	29	30	30	31	31								
28	28	29	29	30	30	31	31								

可以运行下边的命令进行查看

```
代码块
1 ./triton-tensor-layout -l "#ttg.blocked<{sizePerThread = [2, 2], threadsPerWarp = [8, 4], warpsPerCTA = [1, 2], order = [1, 0]}>" -t "tensor<16x16xf16>"
```

```
(triton-learn) root@056c5aab901e:~/playground1/workspace/triton-learn/triton/build/cmake.linux-x86_64-cpython-3.10/bin [main]
● # ./triton-tensor-layout -l "#ttg.blocked<{sizePerThread = [2, 2], threadsPerWarp = [8, 4], warpsPerCTA = [1, 2], order = [1, 0]}>" -t "tensor<16x16xf16>"
```

Print layout attribute: #ttg.blocked<{sizePerThread = [2, 2], threadsPerWarp = [8, 4], warpsPerCTA = [1, 2], order = [1, 0]}>

```
[[ T0:0, T0:1, T1:0, T1:1, T2:0, T2:1, T3:0, T3:1, T32:0, T32:1, T33:0, T33:1, T34:0, T34:1, T35:0, T35:1]
[ T0:2, T0:3, T1:2, T1:3, T2:2, T2:3, T3:2, T3:3, T32:2, T32:3, T33:2, T33:3, T34:2, T34:3, T35:2, T35:3]
[ T4:0, T4:1, T5:0, T5:1, T6:0, T6:1, T7:0, T7:1, T36:0, T36:1, T37:0, T37:1, T38:0, T38:1, T39:0, T39:1]
[ T4:2, T4:3, T5:2, T5:3, T6:2, T6:3, T7:2, T7:3, T36:2, T36:3, T37:2, T37:3, T38:2, T38:3, T39:2, T39:3]
[ T8:0, T8:1, T9:0, T9:1, T10:0, T10:1, T11:0, T11:1, T40:0, T40:1, T41:0, T41:1, T42:0, T42:1, T43:0, T43:1]
[ T8:2, T8:3, T9:2, T9:3, T10:2, T10:3, T11:2, T11:3, T40:2, T40:3, T41:2, T41:3, T42:2, T42:3, T43:2, T43:3]
[ T12:0, T12:1, T13:0, T13:1, T14:0, T14:1, T15:0, T15:1, T44:0, T44:1, T45:0, T45:1, T46:0, T46:1, T47:0, T47:1]
[ T12:2, T12:3, T13:2, T13:3, T14:2, T14:3, T15:2, T15:3, T44:2, T44:3, T45:2, T45:3, T46:2, T46:3, T47:2, T47:3]
[ T16:0, T16:1, T17:0, T17:1, T18:0, T18:1, T19:0, T19:1, T48:0, T48:1, T49:0, T49:1, T50:0, T50:1, T51:0, T51:1]
[ T16:2, T16:3, T17:2, T17:3, T18:2, T18:3, T19:2, T19:3, T48:2, T48:3, T49:2, T49:3, T50:2, T50:3, T51:2, T51:3]
[ T20:0, T20:1, T21:0, T21:1, T22:0, T22:1, T23:0, T23:1, T52:0, T52:1, T53:0, T53:1, T54:0, T54:1, T55:0, T55:1]
[ T20:2, T20:3, T21:2, T21:3, T22:2, T22:3, T23:2, T23:3, T52:2, T52:3, T53:2, T53:3, T54:2, T54:3, T55:2, T55:3]
[ T24:0, T24:1, T25:0, T25:1, T26:0, T26:1, T27:0, T27:1, T56:0, T56:1, T57:0, T57:1, T58:0, T58:1, T59:0, T59:1]
[ T24:2, T24:3, T25:2, T25:3, T26:2, T26:3, T27:2, T27:3, T56:2, T56:3, T57:2, T57:3, T58:2, T58:3, T59:2, T59:3]
[ T28:0, T28:1, T29:0, T29:1, T30:0, T30:1, T31:0, T31:1, T60:0, T60:1, T61:0, T61:1, T62:0, T62:1, T63:0, T63:1]
[ T28:2, T28:3, T29:2, T29:3, T30:2, T30:3, T31:2, T31:3, T60:2, T60:3, T61:2, T61:3, T62:2, T62:3, T63:2, T63:3]]
```

访存合并

<https://opendeep.wiki/triton-lang/triton/manual-performance-tuning#%E5%86%85%E5%AD%98%E5%90%88%E5%B9%B6%E4%BC%98%E5%8C%96-coalesce>

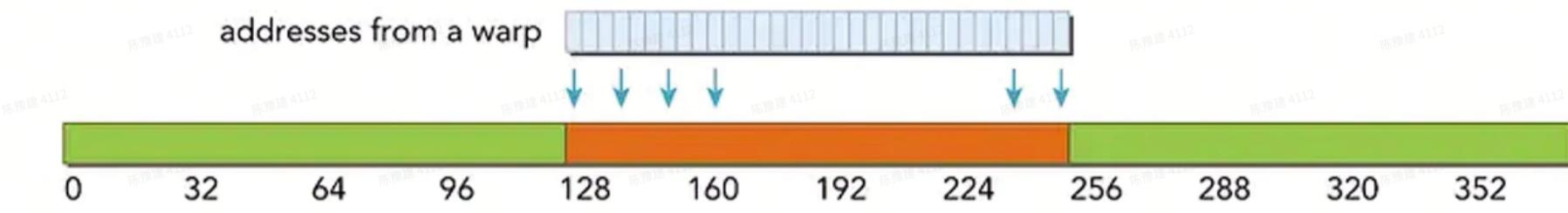
访存合并是针对全局内存访问的内存优化技术，通过对线程束同一时刻发出的多条访存请求进行合并，使处理器可以在最少的访存周期内完成多个数据的存取工作，并通过使用宽字节访存指令进行一步减少访存指令数。

- 缓存行：GPU访问内存中的数据时，会以缓存行为单位进行数据的读取和存储。缓存行大小通常是固定的，与架构有关。
- 内存事务：线程束发起一个访存请求到硬件响应并返回缓存行数据的整个过程。

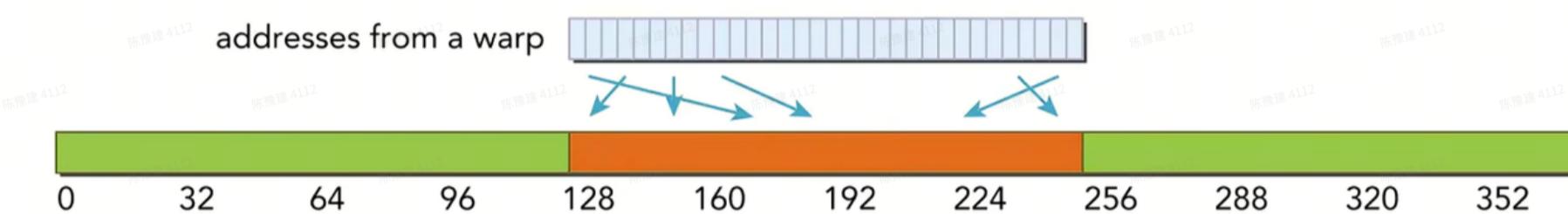
全局内存访问模式

- 对齐访问：线程束需要访问的内存块首地址是缓存行大小的整数倍
- 合并访问：线程束中线程访问的内存地址是一个连续的内存块

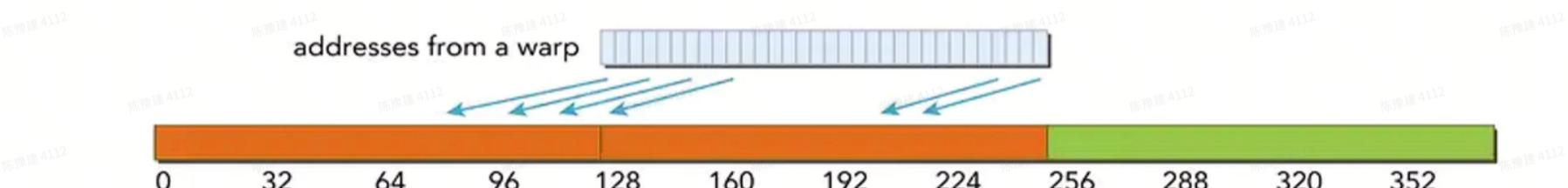
对齐合并访问，利用率 100%



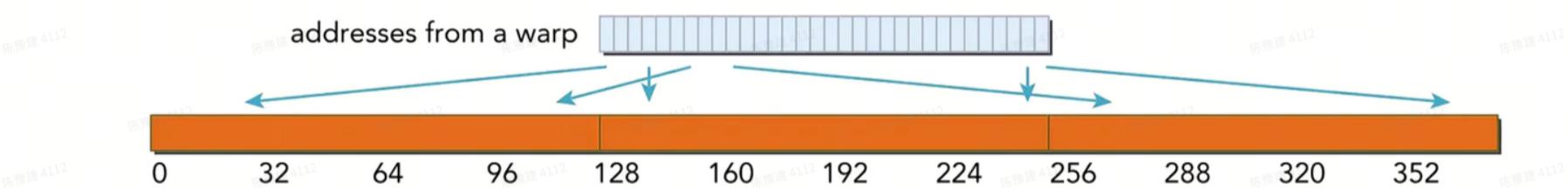
对齐访问，访问地址不连续，利用率 100%



非对齐但连续访问，利用率 50%



非对齐且不连续，N次缓存读取，利用率1/N



在Triton中，算子的访存合并由编译器自动实现。通过使用**访存合并优化pass**对TritonGPU中间表示进行线程布局变换，实现算子的合并访存和向量化访存。

(1) 未合并访存

属性	大小
sizePerThread	[1,1]
threadsPerWarp	[32,1]
warpsPerCTA	[2,2]
order	[0,1]



(2) 合并访存

属性	大小
sizePerThread	[1,1]
threadsPerWarp	[1,32]
warpsPerCTA	[2,2]
order	[0,1]



(3) 向量化访存

属性	大小
sizePerThread	[1,4]
threadsPerWarp	[2,16]
warpsPerCTA	[4,1]
order	[0,1]



Triton中的访存合并优化遍是应用于TritonGPU方言的第一个优化遍，通过对块布局信息进行优化实现核函数的合并访存和向量化访存。

轴分析优化遍 Axisinfo pass

块布局计算 BlockedLayout

块布局设置 BlockedLayout

访存合并优化

详细细节可以看这篇文章：<https://zhuanlan.zhihu.com/p/687394750>

MMA Layout

MMA Layout 用来描述使用GPU的**专用矩阵计算单元**计算后的结果张量布局，这个Layout完全由**专用矩阵计算单元**计算决定，而不是由用户决定。

对于Nvidia MMA Layout

- versionMajor: 指定张量核心的代数，1代表第一代张量核心(Volta), 2代表第二代张量核心(Turing/Ampere)
- versionMinor: 表示张量核心生成的具体Layout, 每一代TensorCore有好几种不同的Layout, 例如Volta可能有多种用0、1、2等标注的Layout
- blockTileSize: 表示Tensor数据如何被切分到warp中。

以Ampere为例，mma.m16n8k16.fp16.fp16.fp16.fp32指令产生的FP32 C矩阵， $C_{16 \times 8} = A_{16 \times 16} \times B_{16 \times 8}$

warpTileSize 为 [16, 8], blockTileSize = [32, 16], 共分配到4个warp中，其中每个thread负责存储结果矩阵C中的4个FP32元素，需要4个寄存器。（warpTileSize 由底层硬件指令隐式确定的，表示一个warp中的32个线程协同计算一个16x8的输出块）

0	0	1	1	2	2	3	3
4	4	5	5	6	6	7	7
...
28	28	29	29	30	30	31	31
0	0	1	1	2	2	3	3
4	4	5	5	6	6	7	7
...
28	28	29	29	30	30	31	31

32	32	33	33	34	34	35	35
36	36	37	37	38	38	39	39
...
60	60	61	61	62	62	63	63
32	32	33	33	34	34	35	35
36	36	37	37	38	38	39	39
...
60	60	61	61	62	62	63	63

64	64	65	65	66	66	67	67
68	68	69	69	70	70	71	71
...
92	92	93	93	94	94	95	95
64	64	65	65	66	66	67	67
68	68	69	69	70	70	71	71
...
92	92	93	93	94	94	95	95

96	96	97	97	98	98	99	99
100	100	101	101	102	102	103	103
...
124	124	125	125	126	126	127	127
96	96	97	97	98	98	99	99
100	100	101	101	102	102	103	103
...
124	124	125	125	126	126	127	127

DotOperand Layout

DotOperand Layout 规定了A和B矩阵的Layout，即它们在thread的寄存器里是怎么分布的。

对于Triton的矩阵乘计算 $D = \text{tt.dot } A, B, C$ ，要求dot操作的两个Operand都必须在执行dot前被转换为DotOperand Layout，进而执行Dot操作。

DotOperand Layout 包含两个参数

- opldx: 0/1, 0代表矩阵A, 1代表矩阵B
- parent: 代表结果矩阵D的Layout，当parent Layout 是 MMA Layout时，降级到PTX对应MMA指令；Blocked Layout时，降级到PTX对应FMA指令。

需要注意：

- Nvidia GPU 在 Hopper之前的架构，tensorcore计算时，A和B都是存在寄存器里的。所以A和B的layout都是DotOperand Layout
- Nvidia GPU 在 Hopper架构中，tensorcore计算时，B矩阵总是直接从shared memory读取，B的layout是Shared Layout，A矩阵可以从shared memory 或 register 读取，所以A的layout可以是Shared Layout，也可以是DotOperand Layout。（大部分情况是Shared Layout）

Slice Layout

Slice Layout 用于压缩给定布局的某个维度，通过在给定维度 dim 上对张量进行分片，然后将这些分片分配给不同的处理线程来工作。就是给定一个 "parent" layout 和一个 "dim"，在 "parent" layout 中挤压给定的 "dim"，并根据新 layout 在 tensor T 中广播分布。

例如

代码块

```
1 T = [x x x x x x x x x]
2 L_parent = [0 1 2 3]
3 [4 5 6 7]
4 [8 9 10 11]
5 [12 13 14 15] (with 16 CUDA threads)
```

L_parent 中的数字代表 thread id.

当 dim = 0，挤压 L_parent 的 dim0，得到：

代码块

```
1 L = [{0,4,8,12}, {1,5,9,13}, {2,6,10,14}, {3,7,11,15}]
```

因此 Tensor T 在 16 个 thread 中按照如下的 layout 来访问/存储 8 个元素：

代码块

```
1 L(T) = [ {0,4,8,12} , {1,5,9,13} , ... {3,7,11,15} , {0,4,8,12} , ... , {3,7,11,15} ]
```

其中 {0,4,8,12} 代表 Tensor 的 index=0 的元素，被 thread 0、4、8、12 共同访问/存储。

当 dim = 1，挤压 L_parent 的 dim1，得到：

代码块

```
1 L = [ {0,1,2,3}, {4,5,6,7}, {8,9,10,11}, {12,13,14,15} ]
```

因此 Tensor T 在 16 个 thread 中按照如下的 layout 来访问/存储 8 个元素：

代码块

```
1 L = [ {0,1,2,3}, {4,5,6,7}, ... , {12,13,14,15} , {0,1,2,3} , ... , {12,13,14,15} ]
```

其中 {0,1,2,3} 代表 Tensor 的 index=0 的元素，被 thread 0、1、2、3 共同访问/存储。

Shared Layout & Bank Conflict

共享内存可以被同一个 Block 内所有线程访问，Shared Layout 就是用来描述共享内存中的 Tensor 如何被不同的线程访问，需要使用 Swizzle 方式去避免不同线程在并发访问共享内存时的 Bank 冲突。

Shared memory 由 32 个 bank 组成，每个 bank 32-bit 位宽，连续的 32 个 32bit 地址会映射到 32 个连续的 bank，bank=(addr/4)%32

Bank Conflict 是指在同一时间内，一个 warp 中的多个线程尝试访问共享内存中同一个 bank 的不同地址时发生的冲突。

Shared Layout 包含如下字段：

- vec: swizzle 的单位。每次访问内存时，将对 vec 个连续的数据元素进行处理或重新排列。
- perPhase: 每个 phase 跨的行数。也就是每 perPhase 行使用的是一个异或值。
- maxPhase: tensor 总共包含多少个 phase。对不同的 phase 的同一 bank 访问不会造成冲突。
- Order: axis 的次序，[0, 1] 表示列主序，[1, 0] 表示行主序。

其中 vec、perPhase、maxPhase 是用于避免 bank conflict 的 swizzle 操作所需要的参数

例如

```
#shared<{vec=1, perPhase=1, maxPhase=4, order=[1,0]}>
```

[0, 1, 2, 3] // xor 0	[0, 1, 2, 3]
[4, 5, 6, 7] // xor 1	[5, 4, 7, 6]
[8, 9, 10, 11] // xor 2	[10, 11, 8, 9]
[12, 13, 14, 15] // xor 3	[15, 14, 13, 12]

```
#shared<{vec=1, perPhase=2, maxPhase=4, order=[1,0]}>
```

```
out[r][c] = in[r][c ^ (r // 2)]
```

[0, 1, 2, 3] // phase 0 (xor 0)	[0, 1, 2, 3]
[4, 5, 6, 7]	[4, 5, 6, 7]
[8, 9, 10, 11] // phase 1 (xor 1)	[9, 8, 11, 10]
[12, 13, 14, 15]	[13, 12, 15, 14]

```
#shared<{vec=1, perPhase=1, maxPhase=2, order=[1,0]}>
```

```
out[r][c] = in[r][c ^ ((r // perPhase) % maxPhase)]
```

[0, 1, 2, 3] // phase 0 (xor 0)	[0, 1, 2, 3]
[4, 5, 6, 7] // phase 1 (xor 1)	[5, 4, 7, 6]
[8, 9, 10, 11] // phase 0 (xor 0)	[8, 9, 10, 11]
[12, 13, 14, 15] // phase 1 (xor 1)	[13, 12, 15, 14]

```
#shared<{vec=2, perPhase=1, maxPhase=4, order=[1,0]}>
```

```
out[r][c] = in[r][(c // vec) ^ phase] * vec + (c % vec)
```

[0, 1, 2, 3, 4, 5, 6, 7] // xor 0	[0, 1, 2, 3, 4, 5, 6, 7]
[8, 9, 10, 11, 12, 13, 14, 15] // xor 1	[10, 11, 8, 9, 14, 15, 12, 13]
[16, 17, 18, 19, 20, 21, 22, 23] // xor 2	[20, 21, 22, 23, 16, 17, 18, 19]
[24, 25, 26, 27, 28, 29, 30, 31] // xor 3	[30, 31, 28, 29, 26, 27, 24, 25]

Linear Layout

与定制布局相比，线性布局统一了表示形式，有助于简化代码库并提升CodeGen的性能

就是从硬件地址映射到张量逻辑索引的函数。

假设 2 维张量 T 存储在 GPU 的寄存器中，使用函数 L 表示它的布局：

$L(t, w) = (x, y)$ 表示 warp w 中 thread t 中的某个寄存器中就保存着 $T[x, y]$ 的值。

Linear Layout 的关键在于：

- 映射中只需要指定某些特定点，就能通过线性规则推导出其它所有点的值。
- 任意 M 维到任意 N 维的整数元组映射，满足 GPU 寄存器布局 (block-id, warp-id, thread-id, reg-id)
- 更换不同的基向量就能表达出不同的布局，进而可以消除特殊情况，写出更通用的代码

假设有 4 个 warp，每个 warp 有 4 个线程，张量 T 是 4×4 ，我们可以指定以下 4 个值来定义 L，即所谓的基向量 (basis vectors)：

$$L(0,1)=(0,1), L(0,2)=(0,2), L(1,0)=(1,1), L(2,0)=(2,2)$$

t/w	0	1	2	3
0	?	(0, 1)	(0, 2)	?
1	(1, 1)	?	?	?
2	(2, 2)	?	?	?
3	?	?	?	?

$(t, w) \rightarrow (t, w^t)$ 可以得到结果

t/w	0	1	2	3
0	(0, 0)	(0, 1)	(0, 2)	(0, 3)
1	(1, 1)	(1, 0)	(1, 3)	(1, 2)
2	(2, 2)	(2, 3)	(2, 0)	(2, 1)
3	(3, 3)	(3, 2)	(3, 1)	(3, 0)

应该是实现《Linear Layouts: Robust Code Generation of Efficient Tensor Computation Using F2》这篇论文里的内容。

https://mp.weixin.qq.com/s/PDFshzgcj_udAFu3aJr1tQ

Layout 转换

Triton 提供了一些操作来转换这些布局，以便在不同的内存层次之间移动数据：

- ConvertLayoutOp(#blocked to #shared): Registers to Shared Memory
- ConvertLayoutOp(#shared to #dotOp): Shared Memory to Registers
- ConvertLayoutOp(#mma to #blocked): Registers to Shared to Registers
-

Triton 中 ConvertLayoutOp 转换为 LLVM IR 的模式类，通过分析源布局和目标布局的线性映射关系，判断数据是否在 CGA、CTA、warp、lane 或线程内进行搬运，并选择对应的转换策略（如使用共享内存、warp shuffle 或寄存器重排），最终生成等价的低层操作。

Triton 生态

- FlagGemm: <https://github.com/FlagOpen/FlagGems>
- FlagTree: <https://github.com/FlagTree/flagtree>
- 华为: <https://gitee.com/ascend/triton-ascend/>
- Triton-shared: <https://github.com/microsoft/triton-shared>
- Triton-Linalg: <https://github.com/Cambricon/triton-linalg>