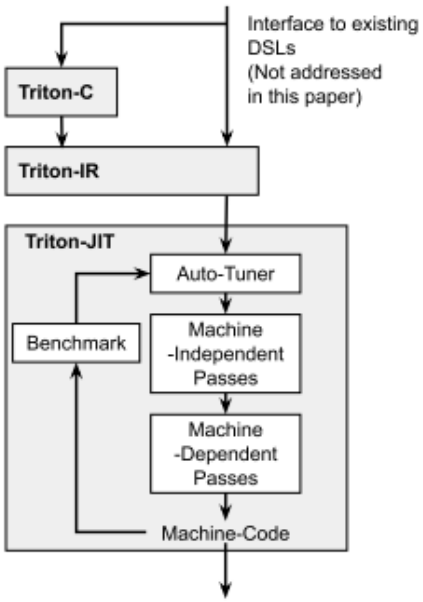
Triton backend compiler调研-HYM

简要

摆脱CUDA的一个新抽象语言，并在设计上主要解决以线程块为单位的调度。线程块内调度由自家编译器完成。整体的思路是将自己定位在TVM算子编译器和Cutlass这种vendor libraries之间。最开始是用C写的，后来用python接口接入的。



我们这里需要关注的是，他生成的LLVM IR而不是又生成了CUDA。

> a set of tile-level machine- dependent passes for generating efficient GPU-ready LLVM-IR GPU-ready LLVM-IR

作为一个编译器，也支持了tuning的功能，并且是通过实测来反馈的。

此外，作为新的框架，有很多新的算子都可以重新实现。FlashAttention最开始就是基于这个写出来的。

和CUDA在控制粒度上的对比

	CUDA	TRITON
Memory Coalescing	Manual	Automatic
Shared Memory Management	Manual	Automatic
Scheduling (Within SMs)	Manual	Automatic
Scheduling (Across SMs)	Manual	Manual

Compiler optimizations in CUDA vs Triton.

Triton语义

分块语义：主要贡献是抽象，隐藏访存合并、缓存管理细节。

Triton IR的设计

设计的语法很像LLVM。论文讲述这部分更多是一些IR的设计，告诉了用户什么情况下用什么写法来支持，具体的优化是由后面的Triton-JIT完成的。这个部分我们编程的时候可能不会用到，但是可以作为debug的参考。从计算图层到硬件层的IR表示分别是:ttir, ttgir, llvmir

对ttir的代码优化在这个位置：

https://sourcegraph.com/github.com/openai/triton/-/blob/lib/Conversion/TritonToTritonGPU/TritonToTritonGPUPass.cpp

里面有很多template写了很多pass，用来验证每个tl.op规则转换的合法性。

ATTR

Encoding主要是负责Tensor到GPU实际硬件的映射关系的。这里面有几种不同的Encoding策略，比较乱，我让Cody帮我整理了一下，这几个EncodingAttrs的区别：

```
DistributedEncodingAttr
    |-- BlockedEncodingAttr
    |-- MmaEncodingAttr
    |-- SliceEncodingAttr
    |-- DotOperandEncodingAttr

SharedEncodingAttr
```

The CTALayout indicates how the tensor is laid out across **CTAs (cooperative thread arrays)**

SharedEncodingAttr

`triton::gpu::SharedEncodingAttr`

1. It is defined in `TritonGPUAttrDefs.td` and represents an encoding where tensor elements can be accessed by different threads via shared memory.
2. `SharedEncodingAttr::get()` is called to construct an instance of this encoding.
3. The parameters passed to `get()` include:
    - `vec` : The vector size, e.g. number of elements accessed together.
    - `perPhase` : Elements per phase.
    - `maxPhase` : Maximum phase.
    - `order` : The dimension order.
    - `CTALayout` : The CTA layout.
4. These parameters define how the tensor is accessed by threads and distributed across CTAs.
5. The created `SharedEncodingAttr` instance is set as the encoding on the tensor type after conversion.
6. This indicates the tensor uses a shared memory layout accessible by different threads.

这个Attr允许了不同的threads去访问相同的张量数据。这里设计了一些策略去防止bank conflicts，他们取名叫做swizzle，实际上就是传统优化见到的内存重排。swizzle的参数如下：

swizzling parameters are:

- vec - number of elements stored contiguously as a vector
- perPhase - number of phases
- maxPhase - total number of phases，是Bank conflict的最大phase数

上述参数放在硬件存储排布的样例如下：

> A_{0, 0} A_{0, 1} A_{0, 2} A_{0, 3} ... [phase 0] \ **per_phase = 2**
>
> A_{1, 0} A_{1, 1} A_{1, 2} A_{1, 3} ... [phase 0] /
>
> groups of **vec=2** elements
>
> are stored contiguously
>
> _ _ _ _ /_ _ _ _
>
> A_{2, 2} A_{2, 3} A_{2, 0} A_{2, 1} ... [phase 1] \ **per phase = 2**
>
> A_{3, 2} A_{3, 3} A_{3, 0} A_{3, 1} ... [phase 1] /

DistributedEncodingAttr

> - It inherits from `TritonGPU_Attr<"DistributedEncoding">`, meaning it is a custom TritonGPU attribute named "DistributedEncoding".
> - Distributed encodings define a layout function that maps tensor indices to CUDA thread IDs in a distributed manner. The mapping is characterized by a parameter tensor L.
> - The attribute contains a `dim` unsigned integer parameter and a `parent` attribute parameter.
> - It has a custom assembly format parser/printer defined.
> - The description provides details on how the layout function works and gives an example mapping a 2D tensor to a 4x4 CUDA thread grid.

这里设计了一些分布Tensor设计，和上面Shard相比更General一点（毕竟还有其他的类型的需要继承该类）。上面的SharedEncodingAttr只适用于Shared memory下的一些分布设计，而且为此添加了很多参数如上面所说的vec, phase等。DistributedEncodingAttr的注释如下：

> The layout function \mathcal{L} of this layout is then defined, for an
>
> index `i` \in R^D, as follows:
>
> \mathcal{L}(A)[i_d] = L[(i_d + k_d*A.shape[d]) % L.shape[d]] \forall k_d such as i_d + k_d*A.shape[d] < L.shape[d]
>
> For example, for a tensor/layout pair
>
> A = [x x x x x x x x]
>
> [x x x x x x x x]
>
> L = [0 1 2 3]
>
> [4 5 6 7]
>
> [8 9 10 11]
>
> [12 13 14 15]
>
> L有shape要求吗?
>
> 讨论了一下，这里的map设计应该是指一个具体的线程而不是线程范围，即Tensor A第一个element应该是被映射到0和8这两个线程上而不是[0-8)线程上。

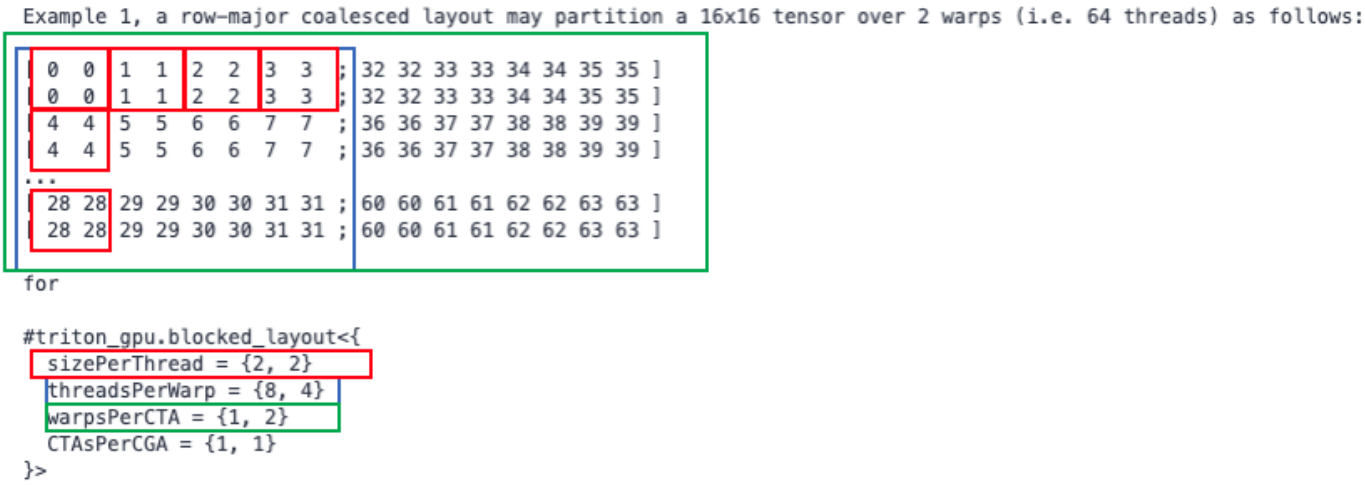
> Then the data of A would be distributed as follow between the 16 CUDA threads:
>
> L(A) = [{0,8} , {1,9} , {2,10}, {3,11}, {0,8} , {1, 9} , {2, 10}, {3, 11},
>
> {4,12}, {5,13}, {6,14}, {7,15}, {4,12}, {5, 13}, {6, 14}, {7, 15}]
>
> }];

BlockedEncodingAttr

`triton::gpu::BlockedEncodingAttr`：

1. `triton::gpu::BlockedEncodingAttr` is a class that represents a blocked encoding attribute in the Triton GPU dialect.
2. It is defined in the file `include/triton/Dialect/TritonGPU/IR/TritonGPUAttrDefs.td` as a subclass of `DistributedEncoding`.
3. `BlockedEncodingAttr::get()` is called to create an instance of this encoding attribute.
4. The parameters passed to `get()` include:
   - `origShape`: The original tensor shape
   - `retSizePerThread`: The size per thread, calculated earlier based on matrix dimensions
   - `retOrder`: The dimension order, defaulting to {1, 0}
   - `numWarps`: The number of warps, queried from the type converter
   - `threadsPerWarp`: Threads per warp, also from the type converter
   - `numCTAs`: Number of CTAs, from the type converter
5. This encoding attribute stores all the relevant GPU blocking parameters needed to correctly shape and distribute the tensor across threads/warps/CTAs.
6. The created instance is passed to `RankedTensorType::get()` to construct the new return type with this GPU blocking encoding.

这里面的参数意义找了一会规律，画图看起来更清晰一点：参数则是对应的shape。



其他例子

```
Example 2, a row-major coalesced layout may partition a 32x32 tensor over 2 warps (i.e. 64 threads) as follows:

[ 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 ]
[ 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  ; 36 36 37 37 38 38 39 39  4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]
[ 4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39  4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]
...                                                   ...
[ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63  28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]
[ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63  28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]
[ 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 ]
[ 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  ; 36 36 37 37 38 38 39 39  4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]
[ 4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39  4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]
...                                                   ...
[ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63  28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]
[ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63  28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]
for
```

```
#triton_gpu.blocked_layout<{
  sizePerThread = {2, 2}
  threadsPerWarp = {8, 4}
  warpsPerCTA = {1, 2}
  CTAsPerCGA = {1, 1}
}>

Example 3, A row-major coalesced layout may partition a 32x32 tensor over 2 warps (i.e. 64 threads) and
4 CTAs (taking 2x2 for example) as follows:

CTA [0,0]                                              CTA [0,1]
[ 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 ]
[ 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  ; 36 36 37 37 38 38 39 39 ]  [ 4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]
[ 4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]  [ 4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]
...                                                    ...
[ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]  [ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]
[ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]  [ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]

CTA [1,0]                                              CTA [1,1]
[ 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 ]
[ 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  ; 36 36 37 37 38 38 39 39 ]  [ 4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]
[ 4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]  [ 4  4  5  5  6  6  7  7  ; 36 36 37 37 38 38 39 39 ]
...                                                    ...
[ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]  [ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]
[ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]  [ 28 28 29 29 30 30 31 31 ; 60 60 61 61 62 62 63 63 ]
for

#triton_gpu.blocked_layout<{
  sizePerThread = {2, 2}
  threadsPerWarp = {8, 4}
  warpsPerCTA = {1, 2}
  CTAsPerCGA = {2, 2}
```

```
53 }>
54 }];
```

MmaEncodingAttr(matrix multiply accumulate)

这个主要是在TensorCore上做矩阵乘加速，与BlockEncodingAttr解决的问题不同

> - It inherits from triton::gpu::DistributedEncodingAttr, meaning it defines a distributed tensor layout mapping to CUDA threads.
> - The attribute contains two version parameters - major and minor - to specify the tensor core generation (Volta, Turing, etc).
> - It also contains a warpsPerCTA parameter to specify how many warps should work on the CTA/threadblock level.
> - The encoding defines how a tensor produced by tensor cores should be partitioned across the CUDA grid/threads for optimal access.
> - It provides methods to query if the encoding is for Volta or Ampere tensor cores based on the version.
> - The attribute has custom syntax parsing and printing to/from Triton IR assembly format.
> - At the core, MmaEncodingAttr encapsulates key tensor core parameters and defines a distributed mapping to CUDA threads for tensors produced by MMA ops. This enables using tensor core outputs in other GPU ops in Triton.

SliceEncodingAttr

略

DotOperandEncodingAttr

MMA的子操作，是GEMM的更细粒度操作。

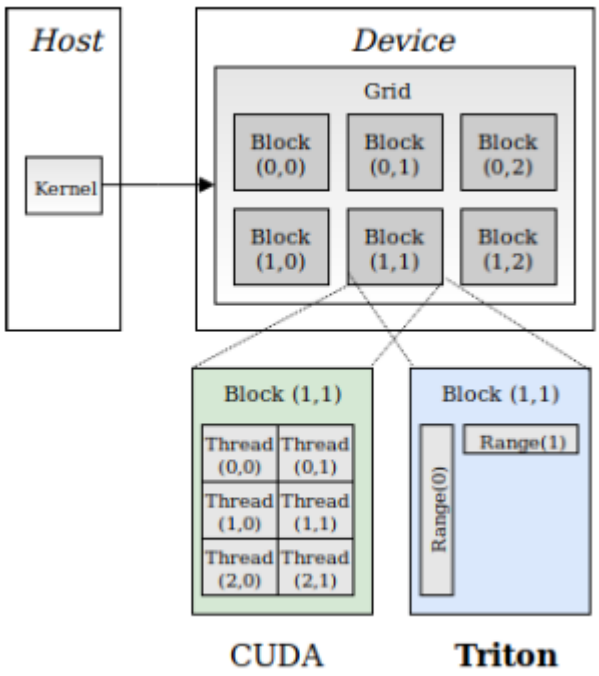
三个参数

- opIdx: 0 for operand a, 1 for operand b
- parent: Encoding layout of the output matrix（一般就是MmaEncodingAttrs）
- kWidth: Width of the K dimension

有比较多的辅助函数都是用来读取上层parent MMA的一些信息用来做特殊优化

```
getMMAv1IsRow: Whether operand is used in MMA row or column op
getMMAv1IsVec4: Whether operand uses vec4 layout
getMMAv1Rep: Get tile shape
getMMAv1ShapePerWarp: Get warp tile shape
getMMAv1Vec: Get vec size
getMMAv1NumOuter: Get number of outer tiles
```

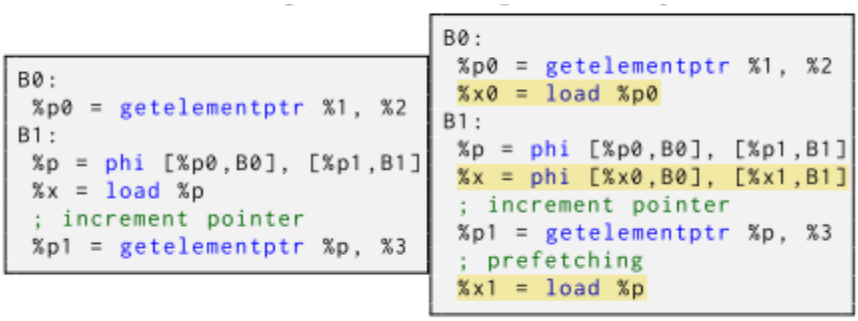
编程模型的不同



triton是通过tl.arange这种方法去做线程索引的。

硬件无关优化

这里的优化都在编译原理上看到过，当然如果觉得这上面有做的不好的，也可以改进。



```
B0:
 %p0 = getelementptr %1, %2
B1:
 %p = phi [%p0,B0], [%p1,B1]
 %x = load %p
 ; increment pointer
 %p1 = getelementptr %p, %3
```

```
B0:
 %p0 = getelementptr %1, %2
 %x0 = load %p0
B1:
 %p = phi [%p0,B0], [%p1,B1]
 %x = phi [%x0,B0], [%x1,B1]
 ; increment pointer
 %p1 = getelementptr %p, %3
 ; prefetching
 %x1 = load %p
```

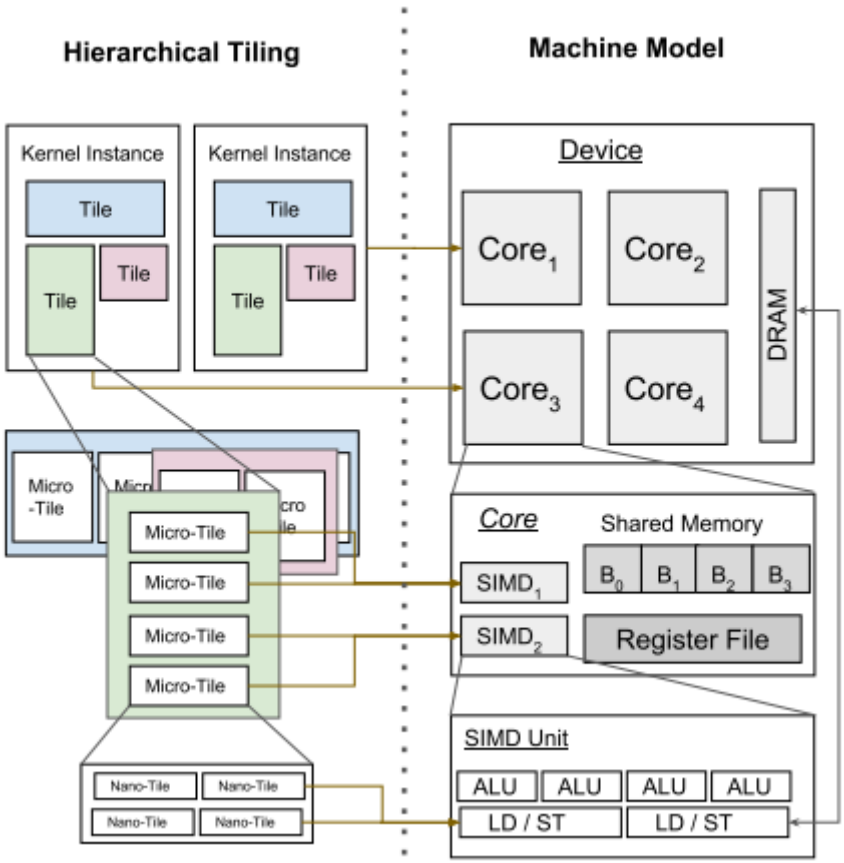
这段代码中的预取操作可以分为以下几个步骤：

1. B0 阶段：
    - `%p0` 是一个指向内存中某个位置的指针，通过 `getelementptr` 指令计算得到。
    - `%x0` 是从 `%p0` 所指向的内存位置加载的数据。
2. B1 阶段：
    - `%p` 是一个 phi 节点，它根据两个来源来选择 `%p0`（来自 B0 阶段）或 `%p1`（来自 B1 阶段）的值。这个 phi 节点的目的是在两个阶段之间选择适当的指针值。
    - `%x` 是另一个 phi 节点，它根据两个来源来选择 `%x0`（来自 B0 阶段）或 `%x1`（来自 B1 阶段）的值。同样，这个 phi 节点用于在两个阶段之间选择适当的数据值。
3. Pointer Increment（指针增加）：
    - `%p1` 通过 `getelementptr` 指令计算得到，基于当前 `%p` 指向的位置。这表示 `%p1` 是 `%p` 的下一个位置，即预取的目标地址。
4. Prefetching 阶段：
    - `%x1` 是从 `%p1` 所指向的内存位置加载的数据。这里的目的是在实际需要使用 `%x1` 数据之前，预先加载到高速缓存中。

这段代码中的内存预取通过在 B1 阶段中预先加载下一个数据项（`%p1` 和 `%x1`）来优化内存访问性能。这可以减少因等待内存访问而导致的计算延迟，提高程序的整体性能。具体效果取决于具体的硬件和访问模式，但总体目标是减少内存访问的等待时间，从而提高程序的吞吐量。

硬件相关优化

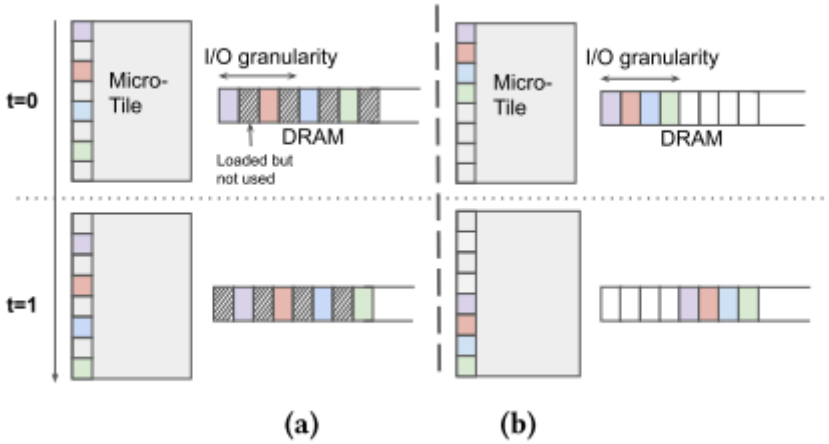
水平分块



这里说一下水平分块的优势：

1. 并行性：将计算任务或数据分解成小块可以充分利用计算机的并行计算能力。这意味着可以同时处理多个小块，从而提高计算速度。
2. 访存优化：将数据分块可以更好地利用计算机的内存层次结构，包括高速缓存和主存。较小的块更容易适应高速缓存，并且减少了不必要的内存传输，从而降低了内存访问延迟。
3. 自动化和可扩展性：根据Triton-IR 的结构，这种分块策略可以自动化地生成和优化分块配置，而无需复杂的多边形计算机制。这使得优化过程更加容易且可扩展，适用于各种类型的程序，而不仅仅是那些需要复杂优化技巧的程序。
4. 适应不同硬件：不同的计算机和硬件架构具有不同的计算能力和内存结构。**分块策略**可以根据特定硬件的特性来定制，以获得最佳性能。这种适应性使得分块策略在各种不同的硬件平台上都能够发挥作用。

访存合并



左图是访存未合并的情况，右图是访存合并的情况。

访存合并主要通过以下方式来优化：

1. 数据合并：在GPU上，线程通常以线程块（thread block）的形式运行，而每个线程块包含多个线程。为了减少对全局内存的访问次数，GPU会尝试将多个线程的内存访问请求合并成更大的内存块一次性读取。这样，多个线程可以共享一次内存访问，减少了内存访问的总次数。
2. 连续内存访问：GPU会尽量将具有连续内存地址的数据项合并成一次访问。这种合并可以减少内存访问的延迟，因为连续的内存地址通常可以以更高的带宽从内存中读取数据。
3. 减少访问冲突：访存合并还有助于减少内存访问冲突。当多个线程尝试同时访问相邻的内存位置时，可能会导致冲突和延迟。通过合并访问请求，可以减少这种冲突，提高效率。
4. 提高数据局部性：访存合并有助于提高数据局部性，使得GPU能够更好地利用高速缓存。当多个线程合并访问相同的内存块时，这些数据可能会被缓存在高速缓存中，从而加速后续的访问。

源码实现访存合并的位置在lib/Dialect/TritonGPU/Transforms/Coalesce.cpp

`runOnOperation()` 函数会对合适的module添加访存合并Coalesce（通过 `setCoalescedEncoding` 方法）

`setCoalescedEncoding` 这个函数会分析得到最合适的映射关系，主要是通过更改 `layoutMap` 这一个映射关系，后来要通过 `coalesceOp()` 这个函数进行替换。一个**识别->生成layout->转换**的过程。

识别过程通过已有warps, threads和所有访问的tensor顺序等信息，统一调度。特别是对store(写到Global memory上）的操作；对于load操作由于有L1的存在，不用考虑所谓的gaps。

其中转换过程的code如下：

```
1 /***
```

```
 2 For each memory op that has a layout L1:
 3 1. Create a coalesced memory layout L2 of the pointer operands
 4 2. Convert all operands from layout L1 to layout L2
 5 3. Create a new memory op that consumes these operands and produces a tensor
   with layout L2
 6 4. Convert the output of this new memory op back to L1
 7 5. Replace all the uses of the original memory op by the new one
 8 ***/
 9 void coalesceOp(Attribute encoding, Operation *op) {
10     OpBuilder builder(op);
11     // Convert operands
12     // For load/store with tensor pointers, we don't have to change the
13     // operands' type, we do this by changing the outputs' type of
14     // `make_tensor_ptr`
15     SmallVector<Value, 4> newArgs;
16     for (auto operand : op->getOperands()) {
17       auto tensorType = operand.getType().dyn_cast<RankedTensorType>();
18       if (tensorType &&
19           !tensorType.getEncoding().isa<triton::gpu::SharedEncodingAttr>()) {
20         Type newType = getNewType(tensorType, encoding);
21         // Convert layout here
22         newArgs.push_back(builder.create<triton::gpu::ConvertLayoutOp>(
23             op->getLoc(), newType, operand));
24       } else {
25         newArgs.push_back(operand);
26       }
27     }
28
29     // Convert output types
30     SmallVector<Type, 4> newTypes;
31     for (auto t : op->getResultTypes()) {
32       bool isAsync = isa<triton::gpu::InsertSliceAsyncOp>(op);
33       newTypes.push_back(isAsync ? t : getNewType(t, encoding));
34     }
35
36     // Construct new op with the new encoding
37     Operation *newOp =
38         builder.create(op->getLoc(), op->getName().getIdentifier(), newArgs,
39                        newTypes, op->getAttrs());
40
41     // Cast the results back to the original layout
42     for (size_t i = 0; i < op->getNumResults(); i++) {
43       Value newResult = newOp->getResult(i);
44       if (newTypes[i] != op->getResultTypes()[i]) {
45         newResult = builder.create<triton::gpu::ConvertLayoutOp>(
46             op->getLoc(), op->getResult(i).getType(), newResult);
47       }
```

```
48        op->getResult(i).replaceAllUsesWith(newResult);
49      }
50      op->erase();
51    }
```

我理解是计算过程通过新的layout进行，但是计算结果返回的时候按照旧的layout返回。（有没有同学更正一下说的对不对）

指令重排

指令重排的标准有一个很重要的指标就是是否会增加Register Pressure。

```
1  static inline bool
2  willIncreaseRegisterPressure(triton::gpu::ConvertLayoutOp op) {
3    auto srcType = op.getOperand().getType().cast<RankedTensorType>();
4    auto dstType = op.getResult().getType().cast<RankedTensorType>();
5    auto srcEncoding = srcType.getEncoding();
6    auto dstEncoding = dstType.getEncoding();
7    if (srcEncoding.isa<triton::gpu::SharedEncodingAttr>())
8      return true;
9    if (dstEncoding.isa<triton::gpu::DotOperandEncodingAttr>())
10     return true;
11   return false;
12 }
```

判断的标准其实挺显而易见的，只有两种情况能够提升Register Pressure，此时返回True。

1. 输入的encoding是SharedEndoingAttr，这意味着源数据来自共享内存。从共享内存加载数据到寄存器会增加寄存器强度。
2. 输出encoding是DotOperandEncodingAttr，这种编码用于矩阵乘法dot操作的操作数。dot操作在寄存器中执行计算，因此转换到这种布局会增加寄存器压力。

除此之外，还有一些和提升Register Pressure无关的一些重排。总结一下重排的规则是：（按顺序）

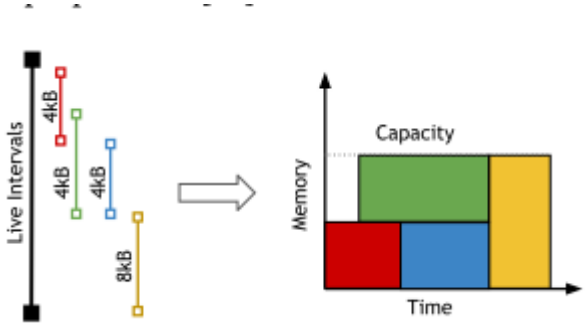
1. 遍历所有ConvertLayoutOps，检查其是否提升寄存器强度。如果提升判断是否是否有user不在同一循环内，则插入user_begin前面
2. 遍历所有ConvertLayoutOps，检查是否转换到shared memory上，如果是的就放在操作定义op后
3. 遍历TransOp，全部放在这些操作的定义op后
4. 最后一步很复杂，没想通。大佬们可以帮忙看一下：

```cpp
// Move `dot` operand so that conversions to opIdx=1 happens after
// conversions to opIdx=0
m.walk([&](triton::gpu::ConvertLayoutOp op) {
  auto dstType = op.getResult().getType().cast<RankedTensorType>();
  auto dstEncoding =
      dstType.getEncoding().dyn_cast<triton::gpu::DotOperandEncodingAttr>();
  if (!dstEncoding)
    return;
  int opIdx = dstEncoding.getOpIdx();
  if (opIdx != 1)
    return;
  if (op->getUsers().empty())
    return;
  auto dotUser = dyn_cast<triton::DotOp>(*op->user_begin());
  if (!dotUser)
    return;
  auto AOp =
      dotUser.getOperand(0).getDefiningOp<triton::gpu::ConvertLayoutOp>();
  if (!AOp)
    return;
  // Check that the conversion to OpIdx=1 happens before and can be moved
  // after the conversion to OpIdx=0.
  if (!dom.dominates(op.getOperation(), AOp.getOperation()))
    return;
  moveAfter(op, AOp);
});
```

因此，总结如下：

- 从共享内存到寄存器的转换会增加寄存器压力。
- 转换为dot操作数布局会增加寄存器压力，因为dot操作在寄存器中进行计算。

内存分配和同步



经典编译器算法优化问题了。不细说，看起来没有做的和传统编译器有差别。

$$in_s^{(RAW)} = \bigcup_{p \in \text{pred}(s)} out_p^{(RAW)}$$

$$in_s^{(WAR)} = \bigcup_{p \in \text{pred}(s)} out_p^{(WAR)}$$

$$out_s^{(RAW)} = \begin{cases} \emptyset & \textbf{if } in_s^{(RAW)} \cap read(s) \neq \emptyset \text{ (barrier)} \\ in_s^{(RAW)} \cup write(s) & \textbf{otherwise} \end{cases}$$

$$out_s^{(WAR)} = \begin{cases} \emptyset & \textbf{if } in_s^{(WAR)} \cap write(s) \neq \emptyset \text{ (barrier)} \\ in_s^{(WAR)} \cup read(s) & \textbf{otherwise} \end{cases}$$

编译器防止读写冲突的必要条件，以确定硬件计算正确。

Auto-tuner

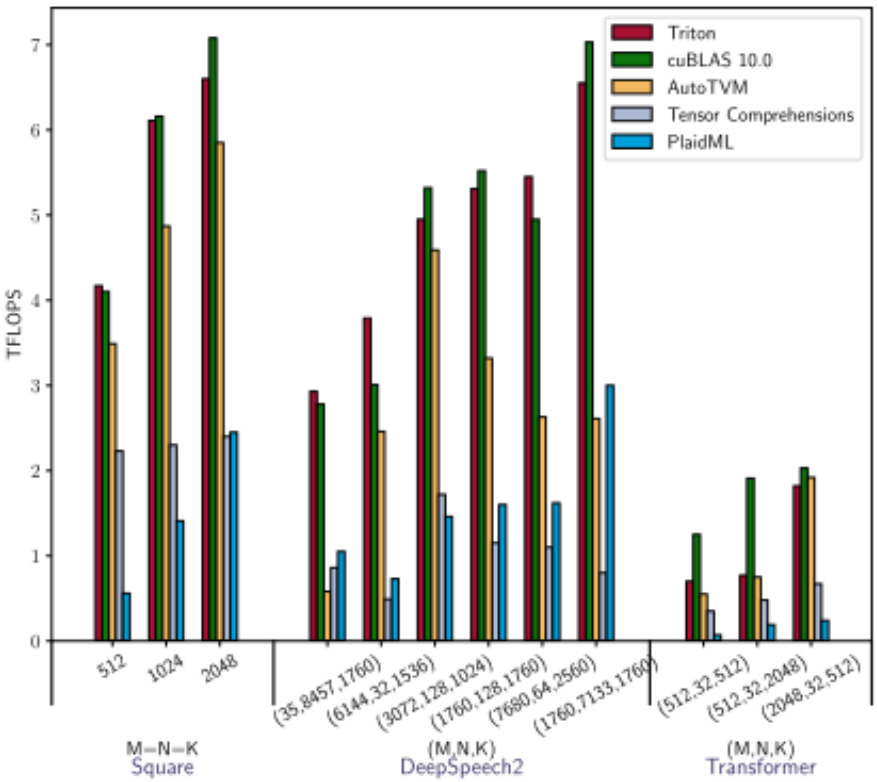
这项工作中，Triton仅考虑了分层分块（Hierarchical Tiling）步骤，每个维度每个块最多有3个分块参数。然后，使用幂次方为2的穷尽搜索来优化这些参数，范围分别为：

(a) tile在32到128之间

(b) micro-tile大小在8到32之间

(c) nano-tile大小在1到4之间

实验



其实和很多文章介绍过一样，triton在tvm和cutlass这两种框架上做了一个取舍，选择了一个折衷方案。但如果triton编译优化的更好，且利用TorchInductor这类框架编译生成，问题可能就会被解决。