TensorLoad 张量处理指令族

1. 概述

这篇文档是关于 Tensor Reshape 类的指令扩展。这一类指令主要用于与计算无关的张量数据重组。主要应用于 Reshape 类算子的实现。例如：Reshape、Transpose、Im2Col、Compress、Concat 等 这一套指令扩展是将需要进行重排的 tensor 拆分成多个三维 block（对应一个架构寄存器），指令通过对 block 的操作，从而实现对整个 tensor 的数据重排。

- 支持多种张量操作：
  - TL.XPOSE：4 维张量任意两个维度的交换
  - TL.CONCAT：3 维张量按指定维度进行有效位拼接
  - TL.MERGE：3 维张量按指定维度进行有效位融合
  - TL.ADDI：张量元素级立即数加法运算
  - TL.MLOAD/TL.LOAD：按最高维度掩码条件载入数据
  - TL.MSTORE/TL.STORE：按最高维度掩码条件存储数据
- 原地操作：source reg 和 destination reg 相同（TL.XPOSE）
- 立即数支持：TL.ADDI 支持 12 位有符号立即数
- 内存访问：TL.MLOAD/TL.MSTORE 支持条件化内存操作

2. 编程模型

2.1 自定义 CSR

这一套指令扩展增加了若干个 CSR。CSR 的基本信息如下表所示

Address	Name	Description
TBD	ttype	用于描述三维 block 的数据类型
TBD	tshape	用于描述三维 block 的形状大小
TBD	tmask_ls	用于辅助 load/store 指令的实现
TBD	tmask_concat_1	用于辅助 concat 指令的实现, 用于指定 tlrs1 中的有效数据
TBD	tmask_concat_2	用于辅助 concat 指令的实现, 用于指定 tlrs2 中的有效数据
TBD	tmask_load_stride	用于辅助 load/store 指令的实现
TBD	tmask_load_width	用于辅助 load/store 指令的实现

2.1.1 ttype

Bits	Name	Description
31:12	0	Reserved (must be zero).
11:10	tfp32[1:0]	32-bit float point enabling.
9:8	tfp16[1:0]	16-bit float point enabling.
7:6	tfp8[1:0]	8-bit float point enabling.

Bits	Name	Description
5:4	tfp4[1:0]	4-bit float point enabling.
3	tint32	32-bit integer enabling.
2	tint16	16-bit integer enabling.
1	tint8	8-bit integer enabling.
0	tint4	4-bit integer enabling.

对于 tint4、tint8、tint16、tint32 字段，如果设置为 1，则表明当前 block 的每个元素的数据类型为：int4、int8、int16、int32

对于 tfp8 字段，如果值为 2'b01，则数据类型为 E4M3；如果值为 2'b10，则数据类型为 E5M2；如果值为 2'b11，则数据类型为 E3M4。若不支持 fp8，则该字段为 2'b00

后续补充其他浮点数据类型

2.1.2 tshape

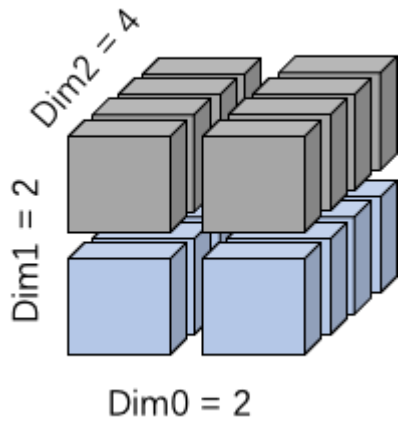
每个 block 为三维 tensor， tshape 用于描述该三维 tensor 的大小。 对于多维度 tensor 的定义与 PyTorch 保持一致。PyTorch 中，Tensor 的维度按索引从 0 开始编号，遵循“左高右低”的原则： 最高维度（Highest Dimension）： 索引 0，即最外层的维度（如 (N, H, W, C) 中的 N 维度）。 最低维度（Lowest Dimension）： 最后一个索引（即 dim=-1 或 dim=n-1，其中 n 是总维度数），即最内层的维度（如 (N, H, W, C) 中的 C 维度）。

Bits	Name	Description
31:24	0	Reserved (must be zero).
23:16	shape_dim0[7:0]	三维 block 的最外层维度的形状
15:8	shape_dim1[7:0]	三维 block 的中间维度的形状
7:0	shape_dim2[7:0]	三维 block 的最内层维度的形状

2.1.3 tmask_ls

tmask_ls 用于辅助 tl.load 和 tl.store 的执行，并且只能作用于 block 的最外层维度。因此其有效位由 tshape 决定，为 tmask_ls[shape_dim0 -1 : 0]

如图所示，如果当前设置 shape_dim0 = 2，shape_dim1 = 2，shape_dim2 = 4，则 tmask_ls 的有效位为 tmask_ls[1:0]。若 tmask_ls[1:0] = 2'b01，则表明 load/store 的有效数据为图中蓝色部分。



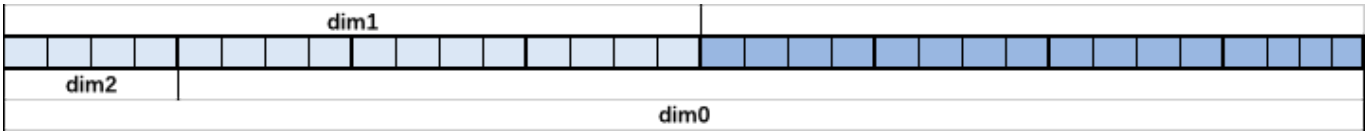
说明：浅色区表示有效切片，深色区表示无效切片；有效位数量由 `shape_dim0` 限定。

2.1.4 tmask_concat_1 / tmask_concat_2

tmask_concat_1/tmask_concat_2 用于辅助 tl.concat 的执行，用于辨明指令中指定维度中哪些数据是有效的。更多细节请参考 4.XX节。

2.2 自定义架构寄存器

这一套指令扩展增加了 32 个 TensorLoad 架构寄存器 (TLReg): 每个 tlreg 的大小为 1024 Byte，其中 tl0 为 zero 寄存器，其值恒定为 0。 tlreg中存放的 block 的数据类型及形状信息由ttype和tshape指定。 block 的大小不能超过一个 tlreg 可以表示的范围。 tlreg 的结构如下图所示：



说明：`TLReg` 容量为 1024 Byte，按 `ttype` 和 `tshape` 进行元素解释与维度映射。

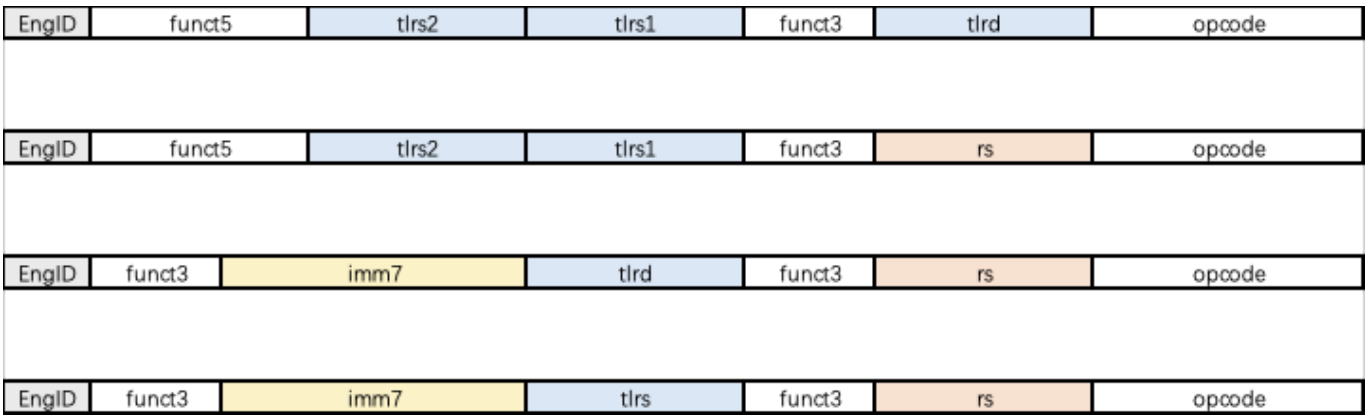
tlreg 中存储的是一个三维 block， 其尺寸信息分别为：D0，D1，D2。 则 tlreg 数据排布：

```
for i in range(D0):
    for j in range(D1):
        for k in range(D2):
            tlreg[i*D1*D2+j*D2+k] = block[i][j][k]
```

3. 指令设计

3.1 指令格式总览

Tensor Reshape 扩展指令采用 32-bit 指令编码。 指令格式概览如下：



3.2 Load/Store 指令

3.2.1 TL.LOAD / TL.MLOAD

指令功能：按block最高维度掩码从内存条件载入数据到TLReg

指令格式:

- [31:30] EngineID = 0b00(TensorLoad) / 0b01(TensorComp) / 0b10(TensorStore) / 0b11(Reshape)
- [29] st = 0b0
- [28] tm =
- [17:20] imm8 = 立即数
- [19:15] tlrd = 目标TLReg索引 [0-31]
- [14:12] funct3 = 0b000 (TL.MLOAD)
- [11:7] rs = 内存基地址GPR索引 [0-31]
- [6:0] opcode = 0b1011011 (CUSTOM-2)

掩码载入控制: TL_LOAD_MASK_CSR：32 位载入掩码寄存器

- 控制最高维度的切片需要载入的切片
- 1 = 载入该切片，0 = 跳过该切片
- 掩码位数有效值由张量最高维度大小确定

寻址方式:

- TL_LOAD_STRIDE_CSR{i}：第 i 个 32 位 Load Stride 寄存器
- TL_LOAD_WIDTH_CSR：32 位 Load Data Width 寄存器

```
// mtype = int8
for i in range(D0):
  if (load_mask[i] == 1):
    tlrd[i * load_width +: load_width] = MEM[rs + (load_stride{i} + imm8)
* load_width +: load_width];
  else:
    tlrd[i * load_width +: load_width] = 0;
```

汇编语法：

```
// 配置csr
- csrrw x0, TL_LOAD_MASK_CSR,   x10  ;
- csrrw x0, TL_LOAD_STRIDE_CSR, x11  ;
- csrrw x0, TL_LOAD_WIDTH_CSR,  x12  ;

// 执行掩码载入
- tl.mload tlrd, offset(rs1)            ; 从rs1地址按上文所述的寻址方式从 L1M 加
载数据到 tlreg
```

3.2.2 TL.STORE / TL.MSTORE

指令功能：按block最高维度掩码将TLReg的数据写入 L0M 中

指令格式：

- [31:30] EngineID = 0b00(TensorLoad) / 0b01(TensorComp) / 0b10(TensorStore) / 0b11(Reshape)
- [29] st = 0b1
- [28] tm =
- [17:20] imm8 = 立即数
- [19:15] tlrs = 目标TLReg索引 [0-31]
- [14:12] funct3 = 0b010 (TL.MSTORE)
- [11:7] rs = L0M 基地址GPR索引 [0-31]
- [6:0] opcode = 0b1011011 (CUSTOM-2)

掩码存储控制 TL_STORE_MASK_CSR：32 位存储掩码寄存器

- 控制最高维度的哪些切片需要存储
- 1 = 存储该切片，0 = 跳过该切片
- 掩码位数由张量最高维度大小确定

寻址方式：

- TL_STORE_STRIDE_CSR{i}：第 i 个 32 位 Store Stride 寄存器
- TL_STORE_WIDTH_CSR：32 位 Store Data Width 寄存器

```
// mtype = int8
for i in range(D0):
    if (store_mask[i] == 1):
    MEM[rs + (store_stride{i} + imm8) * store_width +: store_width] =
tlrs[i * store_width +: store_width];
    else:
    // skip
```

汇编语法：

```
    // 配置csr
    csrrw x0, TL_STORE_MASK_CSR,   x10  ;
    csrrw x0, TL_STORE_STRIDE_CSR, x11  ;
    csrrw x0, TL_STORE_WIDTH_CSR,  x12  ;

    // 执行掩码载入
    tl.mstore tlrs, offset(rs1)            ; 从rs1地址按上文所述的寻址方式将数据
从 tlreg 写入到 L0M
```

3.3 Arithmetic & Logic 指令

3.3.1 TL.ADDI 指令

指令功能 对TLReg中的每个元素都加上一个12位有符号立即数

指令格式 (I-type变体)：

- [31:30] EngineID = 0b00(TensorLoad) / 0b01(TensorComp) / 0b10(TensorStore) / 0b11(Reshape)
- [29:28] funct2 = 0b00
- [27:20] imm8 = 立即数
- [19:15] tlrs = 源TLReg索引 [0-31]
- [14:12] funct3 = 0b010 (TL.ADDI)
- [11:7] tlrd = 目标TLReg索引 [0-31]
- [6:0] opcode = 0b1011011 (CUSTOM-2)

立即数编码 8 位有符号立即数采用二进制补码表示：

- 正数范围: 0 到 +127 (0x00 到 0x7F)
- 负数范围: -128 到 -1 (0x80 到 0xFF)
- 符号扩展: 在运算时扩展为 8 位有符号数进行计算

汇编语法

```
    tl.addi tlrd, tlrs, imm     ; rd[i] = saturate(rs1[i] + imm) for all i
    tl.addi tlr1, tlr2, 10      ; t1中每个元素 = t2对应元素 + 10
    tl.addi tlr3, tlr3, -5      ; t3中每个元素 -= 5 (原地操作)
```

3.4 Data Move 指令

3.4.1 TL.CONCAT

指令功能 对两个形状相同的3维张量按指定维度进行有效位拼接

指令格式：

- [31:30] EngineID = 0b00(TensorLoad) / 0b01(TensorComp) / 0b10(TensorStore) / 0b11(Reshape)
- [29:25] funct5 = 维度选择 + 指令标识
- [24:20] tlrs2 = 源张量2 (TLReg索引[0-31])

- [19:15] tlrs1 = 源张量1 (TLReg索引[0-31])
- [14:12] funct3 = 0b001 (Data Move)
- [11:7] tlrd = 目标张量 (TLReg索引[0-31])
- [6:0] opcode = 0b1011011 (CUSTOM-2)

funct5字段定义 (用于TL.CONCAT) 对于funct3=001 (Data Move)，funct5的5位编码如下：

- [4:2] = 0b000 (TL.CONCAT)
- [1:0] = concat_dim (拼接维度，2位，范围0-2，因为是3维张量)

示例funct5值：

- 0b000000 (0x0): 沿维度0拼接
- 0b000001 (0x1): 沿维度1拼接
- 0b000010 (0x2): 沿维度2拼接
- 0b000011 (0x3): 保留

双CSR有效位掩码寄存器 TL_CONACT_MASK1_CSR (): 32位有效位掩码1 (控制tlrs1)
TL_CONACT_MASK2_CSR (): 32位有效位掩码2 (控制tlrs2)

掩码位含义：

- 1 = 该位置的数据有效，参与拼接
- 0 = 该位置的数据无效，跳过

拼接算法：

tlreg 中存储的是一个三维 block， 其尺寸信息分别为：D0，D1，D2。 则 tlreg 数据排布：

```
    for i in range(D0):
        for j in range(D1):
            for k in range(D2):
                tlreg[i*D1*D2+j*D2+k] = block[i][j][k]
```

对于 TL.CONCAT，假设两个 tlrs 的融合维度为 D0，则 tlrd和tlrs1、tlrs2之间的数据关系可以描述为：

```
    i_valid = 0
    for n in range(2):
        for i in range(D0):
            for j in range(D1):
                for k in range(D2):
                    if(tl_concat_mask_{n}[i] == 1):
                        tlrd[i_valid*D1*D2+j*D2+k] = tlrs{n}
[i*D1*D2+j*D2+k]
                        i_valid = i_valid + 1
```

汇编语法：

```
// 设置双掩码
csrrw x0, TL_MASK1_CSR, x12   ; 设置rs1有效位掩码
csrrw x0, TL_MASK2_CSR, x13   ; 设置rs2有效位掩码

// 执行有效位拼接
tl.concat.0 tlr0, tlr1, tlr2        ; 沿维度0拼接 (funct7=0x40)
tl.concat.1 tlr0, tlr1, tlr2       ; 沿维度1拼接 (funct7=0x41)
tl.concat.2 tlr0, tlr1, tlr2       ; 沿维度2拼接 (funct7=0x42)
```

3.4.2 TL.MERGE

指令功能：对两个形状相同的3维张量按指定维度进行有效位融合

指令格式：

- [31:30] EngineID = 0b00(TensorLoad) / 0b01(TensorComp) / 0b10(TensorStore) / 0b11(Reshape)
- [29:25] funct5 = 维度选择 + 指令标识
- [24:20] tlrs2 = 源张量2 (TLReg索引[0-31])
- [19:15] tlrs1 = 源张量1 (TLReg索引[0-31])
- [14:12] funct3 = 0b001 (Data Move)
- [11:7] tlrd = 目标张量 (TLReg索引[0-31])
- [6:0] opcode = 0b1011011 (CUSTOM-2)

funct5字段定义 (用于TL.MERGE) 对于funct3=001 (Data Move)，funct5的5位编码如下：

- [4:2] = 0b001 （TL.MERGE）
- [1:0] = merge_dim (融合维度，2位，范围0-2，因为是3维张量)

示例funct5值：

- 0b001000 (0x0): 沿维度0融合
- 0b001001 (0x1): 沿维度1融合
- 0b001010 (0x2): 沿维度2融合
- 0b001011 (0x3): 保留

CSR有效位掩码寄存器 TL_CONACT_MASK1_CSR (): 32位有效位掩码1 (控制tlrs1和tlrs2)

掩码位含义：

- 1 = 该位置的数据来自tlrs1对应位置
- 0 = 该位置的数据来自tlrs2对应位置

融合算法：

tlreg 中存储的是一个三维 block， 其尺寸信息分别为：D0，D1，D2。 则 tlreg 数据排布：

```
for i in range(D0):
    for j in range(D1):
        for k in range(D2):
            tlreg[i*D1*D2+j*D2+k] = block[i][j][k]
```

对于 TL.MERGE，假设两个 tlrs 的融合维度为 D0，：

```
        for i in range(D0):
            for j in range(D1):
                for k in range(D2):
                    if(tl_concat_mask_{n}[i] == 0):
                        tlrd[i*D1*D2+j*D2+k] = tlrs0[i*D1*D2+j*D2+k]
                    else:
                        tlrd[i*D1*D2+j*D2+k] = tlrs1[i*D1*D2+j*D2+k]
```

汇编语法：

```
// 设置双掩码
csrrw x0, TL_MASK1_CSR, x12   ; 设置rs1有效位掩码
csrrw x0, TL_MASK2_CSR, x13   ; 设置rs2有效位掩码

// 执行有效位拼接
tl.merge.0 tlr0, tlr1, tlr2        ; 沿维度0拼接 (funct7=0x08)
tl.merge.1 tlr0, tlr1, tlr2       ; 沿维度1拼接 (funct7=0x09)
tl.merge.2 tlr0, tlr1, tlr2       ; 沿维度2拼接 (funct7=0x0a)
```

3.5 Transpose 指令

指令功能 对两个形状相同的3维张量组成的新张量进行指定维度的交换

指令格式：

- [31:30] EngineID = 0b00(TensorLoad) / 0b01(TensorComp) / 0b10(TensorStore) / 0b11(Reshape)
- [29:25] funct5 = 转置维度编码
- [24:20] tlrs2 = 源张量2 (TLReg索引[0-31])
- [19:15] tlrs1 = 源张量1 (TLReg索引[0-31])
- [14:12] funct3 = 0b011 (TL.XPOSE)
- [11:7] rs = dim_gpr索引 [0-31] (存储维度大小信息的GPR)
- [6:0] opcode = 0b1011011 (CUSTOM-2)

GPR维度描述格式 dim_gpr寄存器的32位布局：

- [31:24] D3_size = 第3维度大小 (8位，范围1-255)
- [23:16] D2_size = 第2维度大小 (8位，范围1-255)
- [15:8] D1_size = 第1维度大小 (8位，范围1-255)
- [7:0] D0_size = 第0维度大小 (8位，范围1-255)

约束: D0_size × D1_size × D2_size × D3_size = 2048

funct5 字段定义 (维度交换 + 指令标识) 对于 funct3=011 (TL.XPOSE)，funct5 的 5 位编码如下：

- [4] = 0b0 (TL.XPOSE 基础标识)
- [3:2] = dim1 (第二个交换维度，2位，范围 0-3)
- [1:0] = dim0 (第一个交换维度，2位，范围 0-3)

示例funct5值：

- 0b00001 (0x01): 交换维度0和1 (dim0=0, dim1=1)
- 0b00010 (0x02): 交换维度0和2 (dim0=0, dim1=2)
- 0b00011 (0x03): 交换维度0和3 (dim0=0, dim1=3)
- 0b01001 (0x09): 交换维度1和2 (dim0=1, dim1=2)
- 0b01010 (0x0A): 交换维度1和3 (dim0=1, dim1=3)
- 0b01011 (0x0B): 交换维度2和3 (dim0=2, dim1=3)

汇编语法：

```
    tl.xpose.dim0_dim1 tlreg1, tlreg2, dim_gpr

    // 具体指令变体：
    tl.xpose.01 t1, t2, x10    ; 交换维度0和1 (funct7=0x01)
    tl.xpose.02 t1, t2, x10    ; 交换维度0和2 (funct7=0x02)
    tl.xpose.03 t1, t2, x10    ; 交换维度0和3 (funct7=0x03)ha
    tl.xpose.12 t1, t2, x10    ; 交换维度1和2 (funct7=0x09)
    tl.xpose.13 t1, t2, x10    ; 交换维度1和3 (funct7=0x0A)
    tl.xpose.23 t1, t2, x10    ; 交换维度2和3 (funct7=0x0B)

    示例：
    // 设置x10仅包含维度大小信息
    li x10, 0x02081008      // D3=2, D2=8, D1=16, D0=8
    tl.xpose.01 t1, t2, x10 ; 交换维度0和1: [8,16,8,2] → [16,8,8,2]
```

4. 操作语义

4.1 TL.XPOSE 张量映射 (funct3=011)

4维张量 [D0, D1, D2, D3] 在两个TLReg中的分布：

- TLReg1: tensor[0:D0//2, :, :, :] （前半部分）
- TLReg2: tensor[D0//2:D0, :, :, :] （后半部分） 注：要求D0为偶数，以便均匀分割

4.2 TL.XPOSE 转置操作 (funct3=011)

transpose(tensor, dim0, dim1):

1. 从dim_gpr读取维度大小 [D0, D1, D2, D3]
2. 验证数据完整性：D0×D1×D2×D3 = 2048
3. 构建4维张量视图从TLReg1和TLReg2
4. 执行维度dim0和dim1的交换
5. 将结果写回TLReg1和TLReg2

4.3 TL.CONCAT有效位拼接操作 (funct3=001)

4.3.1 3维张量映射 每个TLReg存储一个完整的3维张量 [D0, D1, D2]：

- 约束：D0×D1×D2 = 1024 (单个TLReg的容量)
- rs1和rs2存储两个形状相同的源张量
- rd存储拼接后的结果张量(形状与源张量相同)

4.3.2 有效位拼接算法 concat_valid_bits(tensor1, tensor2, mask1, mask2, concat_dim):

1. 验证tensor1和tensor2形状相同：shape1 == shape2
2. 从TL_MASK1_CSR和TL_MASK2_CSR读取32位掩码值
3. 根据concat_dim确定拼接维度：
    - concat_dim=0: 沿D0维度拼接
    - concat_dim=1: 沿D1维度拼接
    - concat_dim=2: 沿D2维度拼接
4. 执行有效位收集和拼接： valid_data1 = collect_valid_data(tensor1, mask1, concat_dim) valid_data2 = collect_valid_data(tensor2, mask2, concat_dim) result = concatenate(valid_data1, valid_data2, concat_dim)
5. 将结果写入rd指定的TLReg

4.3.3 有效位收集策略 collect_valid_data(tensor, mask, concat_dim):

- concat_dim=0: 收集mask[i]=1对应的tensor[i][j][k]
- concat_dim=1: 收集mask[j]=1对应的tensor[i][j][k]
- concat_dim=2: 收集mask[k]=1对应的tensor[i][j][k]

4.3.4 拼接示例 (您的例子) 假设concat_dim=2 (最低维度)，维度大小=4：

- rs1数据: [a, b, c, d]
- rs2数据: [e, f, g, h]
- mask1: 0011 (位置2,3有效)
- mask2: 1100 (位置0,1有效)
- 结果: {rs1[2], rs1[3], rs2[0], rs2[1]} = {c, d, e, f}

4.4 TL.ADDI立即数加法操作 (funct3=010)

4.4.1 元素级运算 对TLReg中的每个字节元素执行立即数加法：

- 源数据: rs1[0..1023] (1024个字节元素)
- 立即数: imm (12位有符号数，范围-2048到+2047)
- 目标数据: rd[0..1023]

4.4.2 饱和加法算法 saturated_add(element, imm):

1. 将8位无符号元素转换为16位进行计算
2. 计算 result = element + imm
3. 应用饱和逻辑： if result > 255: result = 255 if result < 0: result = 0
4. 返回8位无符号结果

4.4.3 并行处理 for i in range(1024): rd[i] = saturated_add(rs1[i], imm)

4.4.4 运算示例

- rs1[0] = 200, imm = 100 → rd[0] = 255 (饱和)
- rs1[1] = 50, imm = -100 → rd[1] = 0 (饱和)
- rs1[2] = 128, imm = 10 → rd[2] = 138 (正常)
- rs1[3] = 30, imm = -20 → rd[3] = 10 (正常)

4.5 TL.MLOAD掩码载入操作 (funct3=011)

4.5.1 最高维度掩码载入 根据张量维度描述和掩码，条件载入最高维度的数据切片：

- 维度描述: rs1 GPR包含张量形状 [D0, D1, D2, D3]
- 内存基址: rs2 GPR包含内存起始地址
- 载入掩码: TL_LOAD_MASK_CSR控制D0维度的载入
- 目标寄存器: rd TLReg接收载入的数据

4.5.2 掩码载入算法 masked_load(base_addr, shape, mask, target_tlreg):

1. 解析张量形状 [D0, D1, D2, D3] 从rs1
2. 验证 D0×D1×D2×D3 = 1024 (单TLReg容量)
3. 从TL_LOAD_MASK_CSR读取32位掩码
4. 对于最高维度D0的每个切片i (i ∈ [0, D0-1]): if mask[i] == 1: slice_size = D1×D2×D3 slice_addr = base_addr + i×slice_size target_tlreg[i×slice_size:(i+1)×slice_size] = memory[slice_addr:slice_addr+slice_size] else: target_tlreg[i×slice_size:(i+1)×slice_size] = 0 (或保持不变)

4.5.3 载入示例 张量形状 [8, 16, 8, 1]，掩码 0b11001100：

- 载入切片 0,1,4,5 (mask位为1)
- 跳过切片 2,3,6,7 (mask位为0)
- 每个切片大小 = 16×8×1 = 128字节

4.6 TL.MSTORE掩码存储操作 (funct3=100)

4.6.1 最高维度掩码存储 根据张量维度描述和掩码，条件存储最高维度的数据切片：

- 源寄存器: rs1 TLReg包含待存储数据
- 内存基址: rs2 GPR包含内存起始地址
- 维度描述: rd GPR包含张量形状 [D0, D1, D2, D3]
- 存储掩码: TL_STORE_MASK_CSR控制D0维度的存储

4.6.2 掩码存储算法 masked_store(source_tlreg, base_addr, shape, mask):

1. 解析张量形状 [D0, D1, D2, D3] 从rd
2. 验证 D0×D1×D2×D3 = 1024 (单TLReg容量)
3. 从TL_STORE_MASK_CSR读取32位掩码
4. 对于最高维度D0的每个切片i (i ∈ [0, D0-1]): if mask[i] == 1: slice_size = D1×D2×D3 slice_addr = base_addr + i×slice_size memory[slice_addr:slice_addr+slice_size] = source_tlreg[i×slice_size:(i+1)×slice_size] else: 跳过该切片的存储操作

4.6.3 存储示例 张量形状 [4, 32, 8, 1]，掩码 0b1010：

- 存储切片 1,3 (mask位为1)

- 跳过切片 0,2 (mask位为0)
- 每个切片大小 = 32×8×1 = 256字节

4.7 地址计算

4维张量元素 tensor[i][j][k][l] 的线性地址： addr = i×(D1×D2×D3) + j×(D2×D3) + k×D3 + l

3维张量元素 tensor[i][j][k] 的线性地址： addr = i×(D1×D2) + j×D2 + k

5. 使用示例

5.1 简化的基本示例

示例1：交换维度0和1，张量形状[8,16,8,2]

D0=8, D1=16, D2=8, D3=2

li x10, 0x02081008 # D3=2, D2=8, D1=16, D0=8 tl.xpose.01 t0, t1, x10 # 执行转置：[8,16,8,2] → [16,8,8,2]

示例2：交换维度2和3，张量形状[16,8,8,2]

D0=16, D1=8, D2=8, D3=2

li x11, 0x02080810 # D3=2, D2=8, D1=8, D0=16 tl.xpose.23 t0, t1, x11 # 执行转置：[16,8,8,2] → [16,8,2,8]

5.2 常见转置模式 (TL.XPOSE)

矩阵转置：[32,64,1,1] → [64,32,1,1]

D0=32, D1=64, D2=1, D3=1

li x11, 0x01014020 # D3=1, D2=1, D1=64, D0=32 tl.xpose.01 t2, t3, x11 # 执行矩阵转置

批量转置：[4,8,8,4] → [4,8,4,8]

D0=4, D1=8, D2=8, D3=4

li x12, 0x04080804 # D3=4, D2=8, D1=8, D0=4 tl.xpose.23 t4, t5, x12 # 执行批量转置

5.3 有效位拼接示例 (TL.CONCAT)

示例1：您的例子 - 最低维度拼接 [8,8,4]

rs1=[a,b,c,d], rs2=[e,f,g,h], 期望结果={c,d,e,f}

li x20, 0x0000000C # mask1: 0011 (位置2,3有效) li x21, 0x00000003 # mask2: 1100 (位置0,1有效) csrrw x0, TL_MASK1_CSR, x20 # 设置rs1有效位掩码 csrrw x0, TL_MASK2_CSR, x21 # 设置rs2有效位掩码 tl.concat.2 t10, t11, t12 # 沿维度2拼接有效位

示例2：沿维度0稀疏拼接 [16,8,8]

选择rs1的奇数位置和rs2的偶数位置

li x22, 0x0000AAAA # mask1: 奇数位有效 (1010...) li x23, 0x00005555 # mask2: 偶数位有效 (0101...) csrrw x0, TL_MASK1_CSR, x22 csrrw x0, TL_MASK2_CSR, x23
tl.concat.0 t13, t14, t15 # 沿维度0拼接

示例3：沿维度1部分拼接 [8,16,4]

选择rs1前4个和rs2后4个

li x24, 0x0000000F # mask1: 前4位有效 (0000...1111) li x25, 0x0000F000 # mask2: 后4位有效 (1111...0000) csrrw x0, TL_MASK1_CSR, x24 csrrw x0, TL_MASK2_CSR, x25 tl.concat.1 t16, t17, t18 # 沿维度1拼接

5.4 立即数加法示例 (TL.ADDI)

示例1：图像亮度调整 - 所有像素增加50

tl.addi t1, t0, 50 # t1[i] = saturate(t0[i] + 50) for all i

示例2：数据归一化偏移 - 减去均值128

tl.addi t2, t1, -128 # t2[i] = saturate(t1[i] - 128)

示例3：原地操作 - 对比度增强

tl.addi t3, t3, 20 # t3[i] = saturate(t3[i] + 20) (原地)

示例4：边界处理 - 测试饱和效果

假设t4中有元素值为[250, 10, 128, 200]

tl.addi t5, t4, 100 # 结果: [255, 110, 228, 255] (250+100和200+100都饱和到255) tl.addi t6, t4, -50 # 结果: [200, 0, 78, 150] (10-50饱和到0)

示例5：批量数据预处理

对1024字节的张量数据进行偏移校正

tl.addi t10, t11, -32 # 减去偏移量32

5.5 掩码载入示例 (TL.MLOAD)

示例1：稀疏矩阵载入 - 张量形状 [8,16,8,1]

只载入第0,1,4,5行 (掩码 0b11001100)

li x10, 0x01081008 # 张量形状: D3=1, D2=8, D1=16, D0=8 li x11, 0x1000 # 内存基地址 li x12, 0x000000CC # 载入掩码: 0b11001100 csrrw x0, TL_LOAD_MASK_CSR, x12 # 设置载入掩码 tl.mload t1, x10, x11 # 条件载入到t1

示例2：边界处理载入 - 张量形状 [4,32,8,1]

只载入前两个和最后一个切片 (掩码 0b1011)

li x13, 0x01082004 # 张量形状: D3=1, D2=8, D1=32, D0=4 li x14, 0x2000 # 内存基地址 li x15, 0x0000000B # 载入掩码: 0b1011 csrrw x0, TL_LOAD_MASK_CSR, x15 tl.mload t2, x13, x14 # 条件载入到t2

5.6 掩码存储示例 (TL.MSTORE)

示例1：稀疏结果存储 - 张量形状 [16,8,8,1]

只存储奇数索引切片 (掩码 0b1010101010101010)

li x16, 0x01081010 # 张量形状: D3=1, D2=8, D1=8, D0=16 li x17, 0x3000 # 内存基地址 li x18, 0x0000AAAA # 存储掩码: 奇数位 csrrw x0, TL_STORE_MASK_CSR, x18 # 设置存储掩码 tl.mstore t3, x17, x16 # 条件存储t3到内存

示例2：部分更新存储 - 张量形状 [8,16,8,1]

只存储前4个切片 (掩码 0b00001111)

li x19, 0x01081008 # 张量形状: D3=1, D2=8, D1=16, D0=8 li x20, 0x4000 # 内存基地址 li x21, 0x0000000F # 存储掩码: 前4位 csrrw x0, TL_STORE_MASK_CSR, x21 tl.mstore t4, x20, x19 # 条件存储t4到内存

示例3：组合操作 - 载入->处理->存储

载入部分数据，处理后存储到不同位置

li x22, 0x000000F0 # 载入掩码: 中间4位 li x23, 0x0000000F # 存储掩码: 前4位 csrrw x0, TL_LOAD_MASK_CSR, x22 tl.mload t5, x10, x11 # 载入中间切片 tl.addi t5, t5, 50 # 数据处理 csrrw x0, TL_STORE_MASK_CSR, x23 tl.mstore t5, x20, x10 # 存储到前面位置

6. 异常和错误处理

6.1 异常条件

TL.XPOSE (funct3=000):

- 非法TLReg索引 (>31)
- 非法维度索引 (>3)
- 维度乘积不等于2048
- dim0 == dim1 (无操作，但不报错)
- D0不为偶数 (无法均匀分割到两个TLReg)

TL.CONCAT (funct3=001):

- 非法TLReg索引 (>31)
- 非法拼接维度 (concat_dim > 2)
- 3维张量大小超过1024字节
- 源张量形状不匹配
- TL_MASK1_CSR或TL_MASK2_CSR未初始化
- 有效位数量超出目标张量容量

TL.ADDI (funct3=010):

- 非法TLReg索引 (>31)
- 立即数超出范围 (不在-2048到+2047之间，但硬件应自动截断)
- 无其他异常条件 (饱和运算不抛出异常)

TL.MLOAD (funct3=011):

- 非法TLReg索引 (>31)
- 非法GPR索引 (>31)
- 张量维度乘积不等于1024
- 内存访问越界或对齐错误
- TL_LOAD_MASK_CSR未初始化
- 最高维度大小超过32 (掩码位数限制)

TL.MSTORE (funct3=100):

- 非法TLReg索引 (>31)
- 非法GPR索引 (>31)
- 张量维度乘积不等于1024
- 内存访问越界或对齐错误
- TL_STORE_MASK_CSR未初始化
- 最高维度大小超过32 (掩码位数限制)

6.2 错误处理

- 维度验证失败：触发非法指令异常
- 访问越界：触发存储访问异常
- 正常情况：指令正常完成，无状态更新

7. 实现注意事项

7.1 性能优化

- 可以使用硬件加速的数据重排单元
- 支持流水线操作，提高吞吐量
- 缓存友好的数据访问模式

7.2 硬件要求

- 需要专用的TLReg寄存器文件
- 高带宽的数据重排引擎
- 与通用寄存器文件的高速接口

8. 扩展性

8.1 未来扩展方向

- 支持不同数据类型（int16, int32, float32等）
- 支持更大的张量（多对TLReg组合）
- 支持更复杂的张量变换（reshape, permute等）

8.2 指令变体 (使用CUSTOM-2空间的不同funct7值)

- TL.XPOSE.W: 32位元素版本 (funct7 = 0b0110010)
- TL.XPOSE.H: 16位元素版本 (funct7 = 0b0110011)
- TL.XPOSE.D: 双精度浮点版本 (funct7 = 0b0110100)
- 基础版本: 8位元素版本 (funct7 = 0b0110001)