一个generic swizzle函数

对一个二维空间的行和列进行swizzle $\mathrm{swizzle}(B, M, S)$：

- 2^M 个元素为一组
- swizzle的二维空间有 2^B 行
- swizzle的二维空间中 2^S 个元素为一列

每个线程用向量化指令访问128b数据，$128/32 = 4\ \mathrm{bank}$，每个线程访问4个bank，8个线程访问一条shared memory cache line。

1. 当数据类型是半精度时，$M = 3$，因为$2^3 = 8 \times 16 = 128\ \mathrm{b}$，128-bit 访存指令读取8个元素，这些元素为一组。
2. $S = 3$，1024 / 128 = 8，8个线程访问一整条shared memory cache line
3. 假如原始输入数据有形状，在内存中连续的维度是64，并且数据类型为半精度，$64 \times 16/1024 = 1$，一个连续维度就占1个shared memory cache line。

swizzle<2, 3, 3>的计算过程

swizzle<2, 3, 3>的计算过程如下：

`Bits=2, MBase=3, Shifts=3` 这样一个swizzle函数的计算过程：

1. **bits掩码**：根据 `Bits` 计算一个掩码： `bit_mask = (1 << Bits) - 1` ，Bits是几，掩码就由几个1构成。例如，Bits=3，掩码为 `111` 。Bits的长度决定了swizzle的二维空间中的列数为：2^{bits}。
2. **yyy_mask和zzz_mask**：计算 `yyy_mask` 和 `zzz_mask` 。假设 `MBase=3` ，那么会有3个比特位保持不改变。从这3个比特位的尾部位置开始，向左数 `Shifts` 个bit位数是yyy_mask，向右数 `Shifts` 个bit位数是 $\mathrm{yyy_mask}$和$\mathrm{zzz_mask}$决定了swizzle二维空间中要去交换的两个位置。

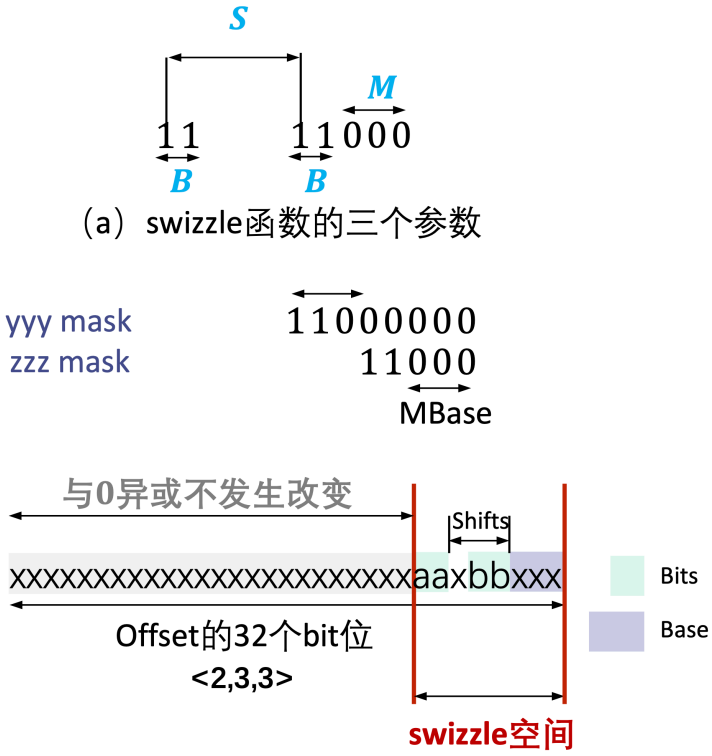

Fig. swizzle函数位运算示意

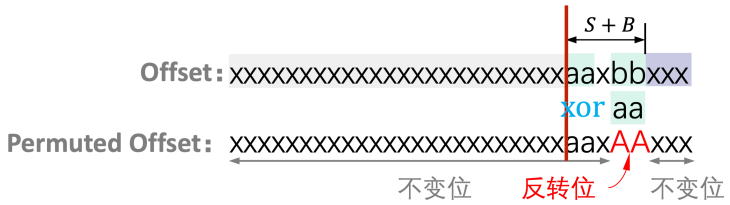

Fig. 原始的offset和swizzled offset之间的关系

3. **permute输入**： `offset ^ shiftr(offset & self.yyy_msk, self.shift)`
    i. offset的二进制表示与 `yyy_mask` 相与，右移 `Shifts` 位，结果记作offset1；offset1是将offset中yyy_mask对应位置的bit位保留原值，其余位置清零，然后取出来的部分移动到zzz_mask所在的位置。
    ii. offset与offset1进行异或。一个bit位与0异或结果不变，结果相当于offset中yyy_mask对应的bit位offset中zzz_mask对应的bit位进行异或，写入zzz_mask对应的位置

对16x16数据块进行swizzle

有一个16x16的数据块，进一步以1×8为粒度被分成了16×2个块（**这里我们先模糊行优先/列优先，具体对应到行优先/列优先时，只需要对这两个维度做相应的调整和适配**）。我们的目标是将这个16×16的数据块以bank-conflict free的方式存储在shared memory中。

这里我们考虑以下假设：

1. 将GPU的shared memory看作由8个bank构成，于是每个bank位宽128 bits，正好对应了上面提到的大小为1×8的一段数据；
2. 数据是以半精度存储，于是$1024/16 = 64$个半精度正好存储在一条shared memory cache line里面
3. 单线程访问128bit，于是8线程并发访存一次的数据恰好可以写入一整行shared memory cache line，而这一点是我们需要保证的，这八个线程写入shared memory的bank id必须是0~8这个8个bank id的一个permutation，不可以落入同一个bank。

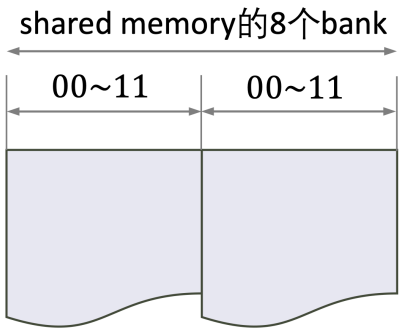

Fig. <2,3,3>设置下shared memory的逻辑编号

$< B = 2, M = 3, S = 3 >$这样一个swizzle函数会对：$2^2 * 2^3 * 2^3 = 4 * 8 * 8 = 16 * 16 = 256$个元素进行permute，也就是恰好对16x16的半精度进行permute。

下表是swizzle函数要去操作的bit位，红色位置的bit位进行异或（后三位总是当成是一个元素）。

-	-	-	x	x	-	-	-
x	x	-	-	-	-	-	-

Bits=2决定了bank id用2个bit位表示，也就是shared memory的8个bank被分成了两组。

xor	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

从上面这个表是$B = 2$时的异或表，可以看到异或具有封闭性；

swizzle的二维index空间中共有$2^2 \times 2^3 = 4 \times 8 = 32$个坐标，我们在下表中表示中这个swizzle空间中所有index的十进制（上方）和对应的二进制（下方）：

0 00000	1 00001	2 00010	3 00011
4 00100	5 00101	6 00110	7 00111
8 01000	9 01001	10 01010	11 01011
12 01100	13 01101	14 01110	15 01111
16 10000	17 10001	18 10010	19 10011
20 10100	21 10101	22 10110	23 10111
24 11000	25 11001	26 11010	27 11011
28 11100	29 11101	30 11110	31 11111

swizzled index（**下面的表格用红色和黑色将数据分成了两部分，可以看出来换序仅仅发生在同色的数据块之内**）：

bank-id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Access-0	0	1	2	3	4	5	6	7
Access-1	9	8	11	10	13	12	15	14
Access-2	18	19	16	17	22	23	20	21
Access-3	27	26	25	24	31	30	29	28

T_0 (0)	T_1 (1)
T_2 (2)	T_3 (3)
T_4 (4)	T_5 (5)
T_6 (6)	T_7 (7)
T_8 (8)	T_9 (9)
T_{10} (10)	T_{11} (11)
T_{12} (12)	T_{13} (13)
T_{14} (14)	T_{15} (15)
T_{16} (16)	T_{17} (17)
T_{18} (18)	T_{19} (19)
T_{20} (20)	T_{21} (21)
T_{22} (22)	T_{23} (23)
T_{24} (24)	T_{25} (25)
T_{26} (26)	T_{27} (27)
T_{28} (28)	T_{29} (29)
T_{30} (30)	T_{31} (31)

源地址空间（Global Memory）线程读取数据方式

T_0 (0)	T_1 (1)	T_2 (2)	T_3 (3)	T_4 (4)	T_5 (5)	T_6 (6)	T_7 (7)
T_9 (9)	T_8 (8)	T_{11} (11)	T_{10} (10)	T_{13} (13)	T_{12} (12)	T_{15} (15)	T_{14} (14)
T_{18} (18)	T_{19} (19)	T_{16} (16)	T_{17} (17)	T_{22} (22)	T_{23} (23)	T_{20} (20)	T_{21} (21)
T_{27} (27)	26 (26)	T_{25} (25)	T_{24} (24)	T_{31} (31)	T_{30} (30)	T_{29} (29)	T_{28} (28)

目标地址空间（Shared Memory）线程写入数据的方式

Fig. 以Global Memory上row major的16x16数据块为源，线程分数据方式以及shared memory中数据存储顺序；

Reference

1. What does bitwise XOR (exclusive OR) mean?
2. DEVELOPING CUDA KERNELS TO PUSH TENSOR CORES TO THE ABSOLUTE LIMIT ON NVIDIA A100
3. cute 之 Swizzle