

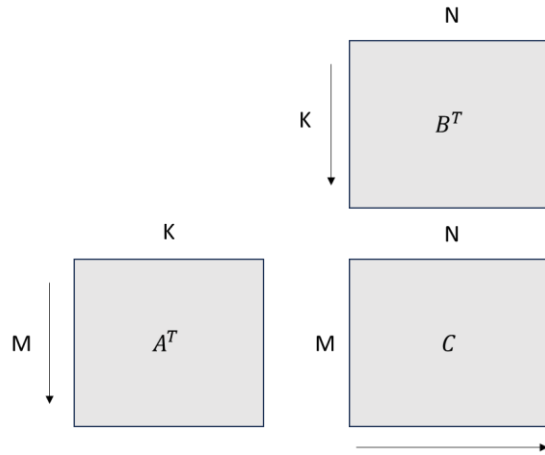
TT 模式下 GEMM 两种实现方法的对比

1. 概述

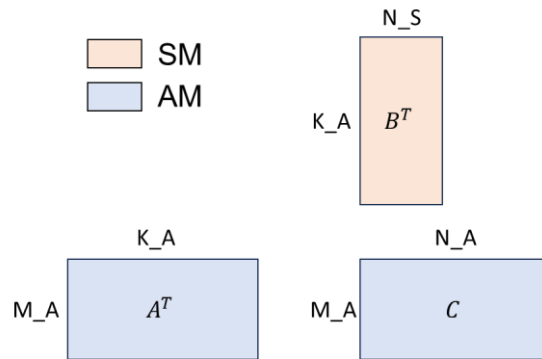
TT 模式的计算表达式为：

$$C += A^T \times B^T$$

其中矩阵 A^T 和 B^T 为列主序存储，存放原始数据和计算结果的矩阵 C 为行主序存储。为了将问题转化为 NN 模式的 GEMM，要么在计算前对输入矩阵 A^T, B^T 进行转置，要么在计算后对结果矩阵 C 进行转置。这里选择后者，一是因为只需要转置一个矩阵，二是因为在片上缓存中转置 C 能够与 C 的三缓冲策略相结合，从而无需占用额外的片上缓存。



TT 模式下输入矩阵 A^T, B^T 都是列主序存储，因此二者在计算中的定位与 NN 模式中正好相反。这体现在， B 分块存储在 SM 中，而 A 的分块存储在 AM 中。片上缓存的分配如下图所示。



其中的分块参数取值分别是：

$$K_A = 512$$

$$M_A = 64$$

$$N_G = 576$$

$$N_A = 144$$

$$N_S = 6$$

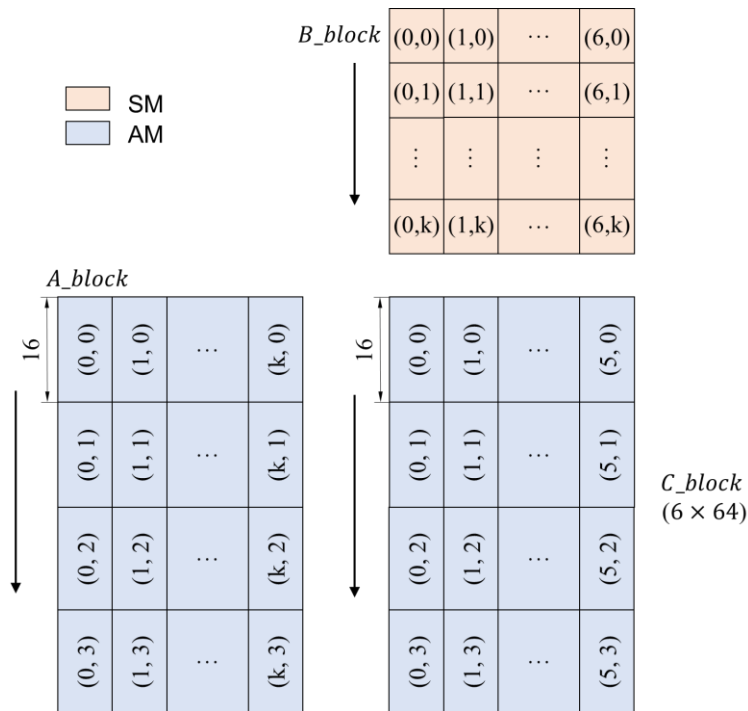
与之相匹配的循环顺序则是：

```

1  for m_i = 0:M_A:M
2
3      DMA_load(A[M_AxK_A]);
4
5      for n_i = 0:N_A:N_G
6
7          DMA_load(C[M_AxN_A]);
8
9          for n_s = 0:N_S:N_A
10
11              DMA_load(B[K_AxN_S]);
12
13              micro_kernel();
14
15          end n_s
16
17          DMA_store(C[M_AxN_A]);
18
19      end n_i
20
21 end m_i

```

其中微内核的算法与 NN 模式相同，但 A 和 B 角色互换，示意图如下：



2. 方法

对计算结果 C 进行转置操作也分为两种方法。一种是 GEMM 算子和转置算子完全分离，GEMM 先对 $A^T \times B^T$ 进行计算，结果存入 ddr，再调用转置算子将 C 从 ddr 取到片上缓存，执行转置操作后再写回 ddr。另一种方法是将转置操作融合进 GEMM 算子，每当微内核计算完毕，就对暂存在 AM 中的计算结果进行转置，然后将转置的结果与 C 的原始数据相加，再写回 ddr。这里只细说第二种方法，算法的伪代码如下。

```

for m_i = 0:M_A:M
    DMA_load(A[M_AxK_A]);
    for n_i = 0:N_A:N_G
        DMA_load(C[M_AxN_A]);
        for n_s = 0:N_S:N_A
            DMA_load(B[K_AxN_S]);
            micro_kernel();
        end n_s
        DMA_store(C[M_AxN_A]);
    end n_i
end m_i

```

```

dma_load(spm_C[0]);

cnt_c = 0;

for n_i = 0:N_A:N_G
    cnt_c_1 = (cnt_c + 1) % 2;

    for n_s = 0:N_S:N_A // [0]
        dma_load(spm_B);
        micro_kernel(spm_A, spm_B, spm_C[2]);
    end n_s

    dma_wait(ch_c1[cnt_c]); // [1]
    transpose(spm_C[2], spm_C[cnt_c]); // [2]
    dma_wait(ch_cs[cnt_c_1]); // [3]

    if(C has next)
        dma_load(spm_C[cnt_c_1]); // [4]
        dma_store(spm_C[cnt_c]); // [5]
    end

    cnt_c = cnt_c_1;
end n_i

dma_wait(ch_cs[(cnt_c+1)%2]);

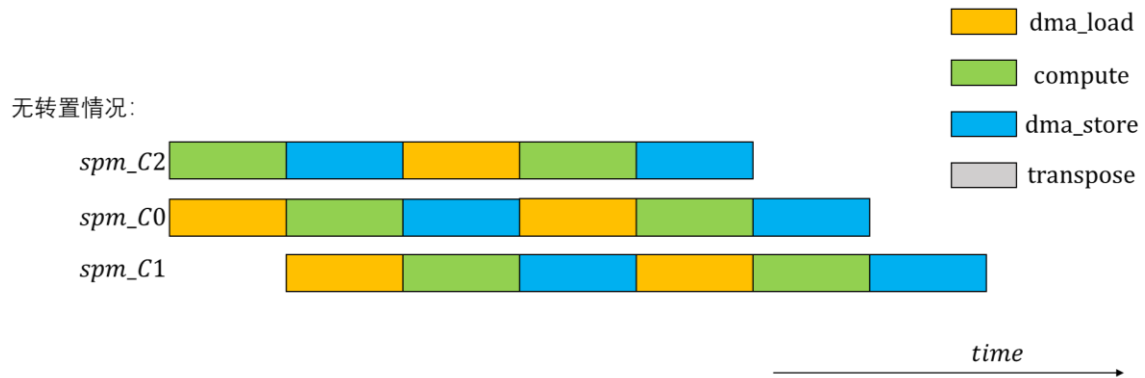
```

因为转置的对象是 C 分块暂存在 AM 中的数据，而 C 分块的读入和写回都发生在红框中的这层循环，所以主要对这层循环中的操作顺序进行修改。右图中编号[0-5]对应的分别是：

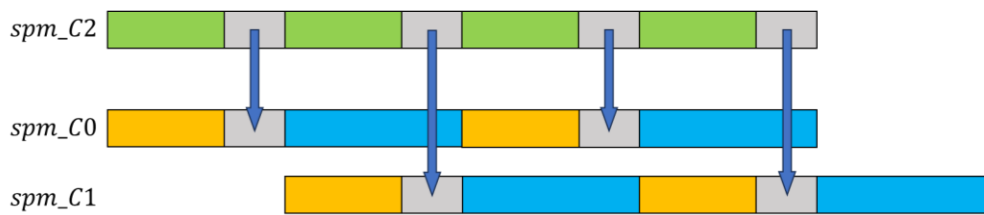
- [0]内层循环(从 gsm 读入 B 分块+微内核执行)；
- [1]等待当前 C 分块的原始数据写入 AM；
- [2]对微内核的计算结果做转置，并与当前 C 分块的原始数据相加；
- [3]等待上一个 C 分块写回 ddr 完成；
- [4]加载下一个 C 分块的原始数据；
- [5]写回当前 C 分块的数据。

与修改前相比，算法的一个不同在于，将计算结果与 C 分块的原始数据相加发生在转置函数[2]中，因此微内核不会用到 C 分块的原始数据，也就可以在原始数据完成写入[1]之前率先执行。微内核的计算结果会写入一个单独的缓冲区 spm_C2，待 C 分块的原始数据读入 AM 完毕[1]，即可执行转置操作[2]，将 spm_C2 中的数据转置后与 C 分块的原始数据相加。此外，[3]之所以放在[4]之前是因为上一个 C 分块与下一个 C 分块使用同一个缓冲区，必须等待旧数据写回完成才能开始装入新数据。[5]是将[2]的结果写回 ddr，这一操作也可以放在[3, 4]之前。

算法的时空图如下，其中 spm_C0/1/2 分别是 C 分块占用的三个缓冲区。无转置的情况下，三个缓冲区地位相同，同一时刻分别在执行读入、计算和写回，交替往复。加入转置操作以后，spm_C2 缓冲区专供微内核写入计算结果，而 spm_C0/1 则交替地执行原始数据的读入、转置和写回。图中的箭头表示微内核的计算结果暂存到 spm_C2 之后，会经过转置和累加，得到最终结果写入 spm_C0/1。



对AM中的计算结果做转置的情况:



3. 性能测试

设输入矩阵的大小是 $M=1536, K=2048, N=2304$ 。对 TT 模式的两种实现方法 (GEMM 与转置分离, GEMM 与转置融合) 进行性能测试, 结果如下。可以看出融合了转置的 GEMM (绿色) 相比于不做转置的 GEMM (灰色) 性能下降很小, 只有不到 2%, 且扩展性很好, 随着计算核数的增加, 平均单核性能没有显著的降低。反观计算与转置分离的 GEMM (蓝色), 单核执行的性能与融合算法相近 (若矩阵规模减小则性能不如融合算法), 但随着计算核数的增加, 平均单核性能下降明显, 这是因为转置算子是访存受限的, 性能无法随计算核数的增加而提高。

