高性能PE算子编程建议

说明：本文档主要基于之前在Perf model上的评测分析结果整理了一些提高PE kernel性能的软件编程建议。

1. 背景说明

以下内容参考自：

1. General-Purpose Graphics Processor Architecture.
2. Computer Architecture: A Quantitative Approach.

1.1 Execution model

下图通过Basic Linear Algebra Software (BLAS) library 中的SAXPY程序在GPU和CPU上的实现来说明两者编程方式和执行过程的不同。

GPU的硬件实现是multi-threaded SIMD，暴露给编程人员的是MIMD，编程人员可以一次launch大量thread，每个thread对array的一个element进行操作。

CPU 通过for循环的每个iteration来完成对array的每个element的操作。因此，CPU如果要达到较高的并行度，需要通过软件和硬件一起尽可能提高**loop-level parallelism**（主要通过loop unroll和SIMD）。

```
void saxpy_serial(int n, float a, float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
main() {
  float *x, *y;
  int n;
  // omitted: allocate CPU memory for x and y and initialize contents
  saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
  // omitted: use y on CPU, free memory pointed to by x and y
}
```

Figure 2.1: Traditional CPU code (based on Harris [2012]).

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if(i<n)
     y[i] = a*x[i] + y[i];
}
int main() {
  float *h_x, *h_y;
  int n;
  // omitted: allocate CPU memory for h_x and h_y and initialize contents
  float *d_x, *d_y;
  int nblocks = (n + 255) / 256;
  cudaMalloc( &d_x, n * sizeof(float) );
  cudaMalloc( &d_y, n * sizeof(float) );
  cudaMemcpy( d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice );
  cudaMemcpy( d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice );
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  cudaMemcpy( h_x, d_x, n * sizeof(float), cudaMemcpyDeviceToHost );
  // omitted: use h_y on CPU, free memory pointed to by h_x, h_y, d_x, and d_y
}
```

Figure 2.2: CUDA code (based on Harris [2012]).

1.2 如何提高Loop level parallelism

本章主要从软件编程（或编译优化）角度讨论如何在Andes Core+TileCore这样的CPU上（Decode宽度1、相同运算部件的指令按序发射、Vector/SIMD运算）尽可能提高program order的loop level parallelism。

总的来说，需要减少以下几种依赖关系导致的CPU流水线执行的bubble。

1.2.1 寄存器访问依赖

寄存器访问依赖主要分为以下三类：

1. WAW（write after write）：两条指令的目的寄存器相同
2. WAR（write after read）：前一条指令源寄存器与后一条指令目的寄存器相同。
3. RAW（read after write）：前一条指令的目的寄存器和后一条指令的源寄存器相同。

其中，WAW和WAR是因为使用相同名字的寄存器导致的，不是真正的数据依赖关系，但是RAW是真实存在的数据流动关系。下面通过一个简单的例子说明这三种依赖关系及其影响。

代码块

```
for
{
    load reg0
    load reg1
    compute reg2, reg0, reg1, reg2 //读reg0,reg1,reg2, 计算结果写回reg2
}
store reg2
```

循环i：

代码块

```
load reg0
load reg1
compute reg2, reg0, reg1, reg2
```

循环 i + 1：

代码块

```
load reg0
load reg1
compute reg2, reg0, reg1, reg2
```

上面的例子中存在以下三种循环间的数据依赖关系（loop-carried dependence）：

1. WAW：循环i和循环i+1的load都需要写reg0/reg1，这两组load之间存在WAW关系。
2. WAR：循环i的compute需要读reg0和reg1，循环i+1的load需要写reg0和reg1，它们之间存在WAR关系。
3. RAW：循环i的compute会将结果写会到reg2，然后再被循环i+1的compute使用，它们之间存在RAW关系。

假设load延迟是3，compute延迟是1。我们希望循环i的load发出后，循环i+1的load也可以顺序发出，从而掩盖访存延迟的影响。即期望的执行情况如下图所示：：

Cycle 0	1	2	3	4	5	6	7	8	9
Load 0									
	Load 1								
		Load 0							
			Load 1						
				Comp					
						Comp			

但是，实际情况下：

由于WAW，循环i+1的load reg0需要等到循环i的load reg0完成才能开始执行。

由于WAR，循环i+1的load reg0需要等到循环i的compute将reg0的数据读走才能开始执行。

由于RAW，循环i+1的compute需要等到循环i的compute完成才能开始执行。

因此，实际的执行情况相对于期望的执行情况多了3个bubble。

Cycle 0	1	2	3	4	5	6	7	8	9
Load 0									
	Load 1								
				Comp					
					Load 0				
						Load 1			
									Comp

如果在软件编程（或编译）过程中进行循环展开，可以将存在寄存器访问依赖的指令间隔开，通过将可并行性执行的指令填入，消除流水线执行的bubble。例如，当循环展开2次时，上述bubble即可全部被消除。

代码块

```
for
```

```
2  {
3      load reg0
4      load reg1
5      compute reg2, reg0, reg1, reg2
6      load reg3
7      load reg4
8      compute reg5, reg3, reg4, reg5
9  }
10 store reg2
11 store reg5
```

1.2.2 访存依赖

上面提到的寄存器访问之间的RAW/WAR/WAW依赖关系，同样也存在于memory access之间。需要说明的是，由于当前TileCore流水线无法直接检测读和写之间的访存依赖关系，因此需要通过插入twait来保证读访问和写访问之间的顺序关系。但是，插入twait会导致loop的执行完全被暂停，即后续的loop iteration在twait等待的访存指令完成之前完全无法展开（具体可参考：Memory Ordering性能评测和分析 中TWAIT的影响）。因此，插入twait在CPU中的影响往往会比在GPU中的影响大很多，建议软件编程时（或者编译插入时）尽可能减少这一类访存依赖关系，且仅在必要时插入twait。

1.2.3 结构依赖

结构依赖是两条指令需要使用相同的运算部件而导致的依赖关系。在TileCore设计中，TMAC/TALU/TSFU等多类运算部件均只有1个，采用多周期指令执行Vector/SIMD运算，因此指令往往会因为等待运算单元ready而被流水线stall住。如果运算单元利用率较高，中间没有bubble，则这一类依赖关系导致的stall可以被忽略掉。需要注意的是因为结构依赖而导致循环无法展开的情况。下面通过一个简单的例子说明该情况。

代码块

```
1  for
2  {
3      TADD
4      TEXP
5      TADD
6  }
```

假设TADD（在TALU中运算）的延迟是10，TEXP（在TSFU中运算）的延迟是30，TSFU运算部件将成为主要瓶颈，如果TSFU在使用中没有bubble即可认为是理想情况。我们期望的流水线执行情况如下图1所示，即第i轮循环和第i+1轮循环的TADD可以连续发出，从而保证TEXP可以连续的在TSFU运算部件上执行。

图1：期望的流水线执行情况

Cycle 0	10	20	30	40	50	60	70	80	90
TADD(i)									
	TEXP(i)								
	TADD(i+1)								
				TADD(i)					
				TEXP(i+1)					
							TADD(i+1)		

但是，由于循环首尾都存在TADD，需使用相同的TALU运算部件，第i+1轮的第一个TADD需要等第i轮最后一个TADD发出后才能发出，从而引入20个cycle的bubble（如下图2所示）。这一类问题可以通过循环展开，或者调整计算部件（比如将TADD指令换成vfadd指令，采用RVV来执行循环最后的加法）来解决。

图2：实际流水线执行情况

Cycle 0	10	20	30	40	50	60	70	80	90
TADD(i)									
	TEXP(i)								
				TADD(i)					
					TADD(i+1)				
						TEXP(i+1)			
									TADD(i+1)

1.2.4 控制依赖

TileCore指令均为Andes Core的ACE扩展指令。当前设计中ACE指令的执行都是非推测式执行的。

对于软件编程而言，这一部分主要需要考虑的是减少不必要的条件控制?

1.3 其他常见优化方法

除了提高loop level parallelism外，其他常见的软件性能优化手段也需要考虑。

1.3.1 访存性能优化

1. 提高访存带宽

   如何能够保证大量数据并行访问时的带宽是提高DLP设计性能的关键因素之一。目前设计中Port interleaving和DRAM row/col/bank的地址映射等机制，将连续访问的地址比较均匀的分布到多个独立的Port或者DRAM bank/col上。因此，在软件编程时需尽可能使用连续访问的地址pattern，从而提升并行访问时的带宽。

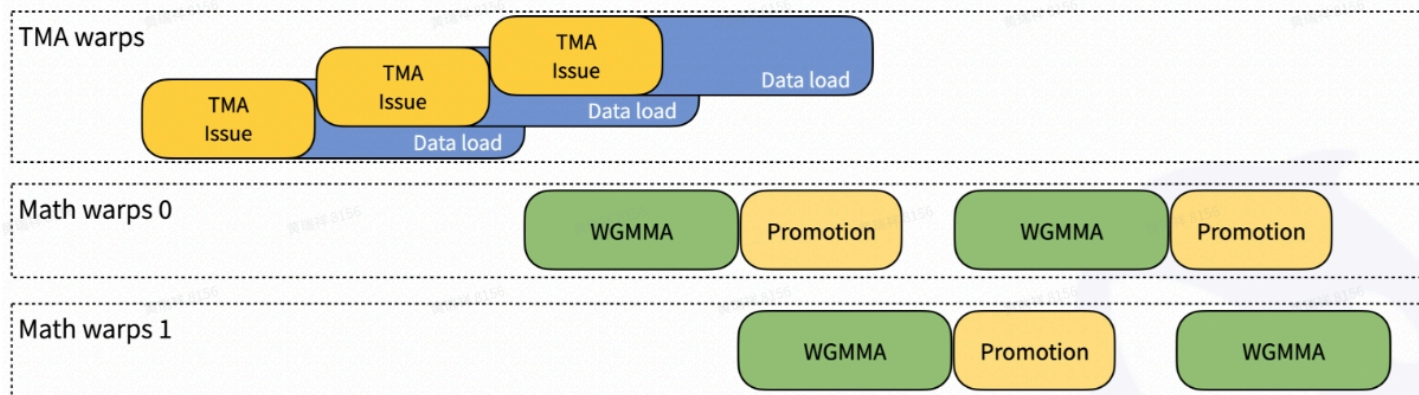
2. 降低访存延迟

   对于GPU这类可并行执行大量线程的设计，访存延迟的影响是相对比较小的。但是，对于CPU流水线设计而言，访存延迟太长可能会直接导致流水线执行中出现的大量的bubble。因此，软件编程中尽可能将重复访问的数据放在延迟较短的local memory。否则，为了掩盖访存延迟软件需要循环展开的次数较高，导致icache miss增加，而且还可能导致寄存器不够用等问题。

   （这也说明了当前设计中使用3D-DRAM设计保证PEC level的低访存延迟对于提升基于RV Core的AI核性能是很重要的）

1.3.2 Dataflow pipelining

利用两个RVCore的SMT2能力，并行进行多个不同的任务，将TileCore中TMAC和其他计算/通信资源同时用起来，从而将其他计算和通信的延迟掩藏在GEMM的计算延迟下。类似于DeepSeek GEMM中这个图例，将数据搬运、其他计算的延迟都尽可能隐藏在GEMM运算下。



2. GEMM编程示例

2.1.1 Simple GEMM kernel

我们首先看一下如果直接从memory（包括local和global memory）读数，计算矩阵乘，再写回memory的情况。

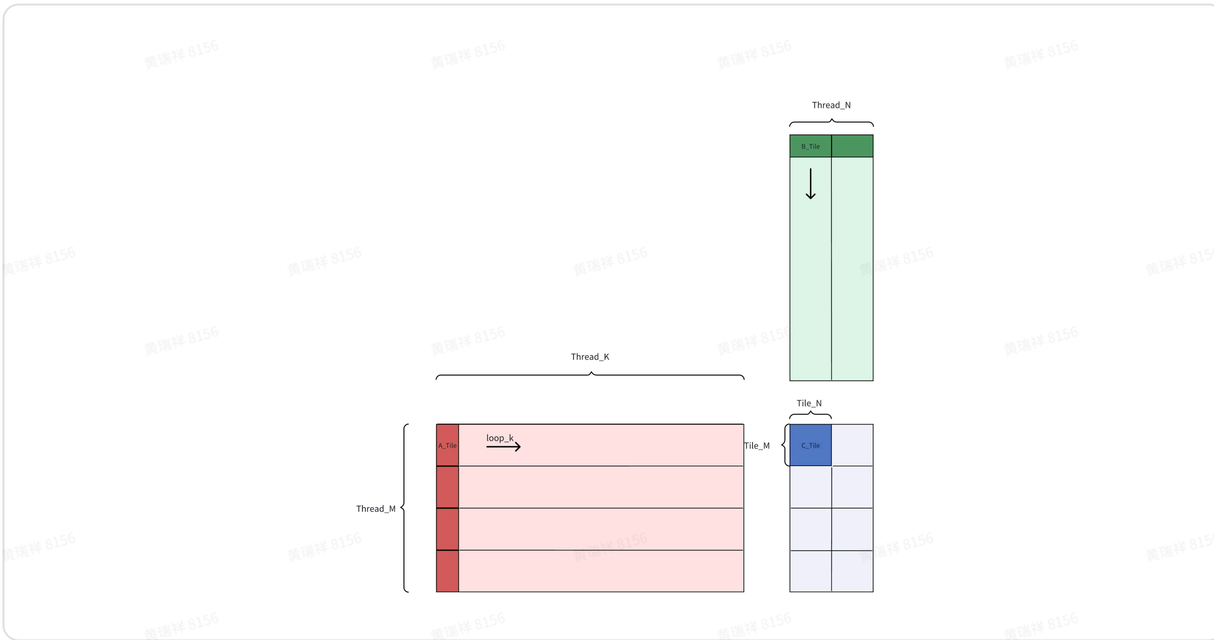
2.1.1.1 K维度累加

2.1.1.1.1 算法说明

（以下内容引自：GEMM simple性能分析）

通过K维度累加进行矩阵运算是一种最常见的写法。基本思想如下面的伪码和图例所示：

```
for（Thread_M/Tile_M)
    for (Thread_N/Tile_N)
        for (Thread_K/Tile_K)   // get C_subtile[m,n]，计算矩阵C的第[m,n]个tile
            从L2B 加载 Tile_M * Tile_K的A和Tile_K * Tile_N的B到tile reg
            计算Tile_M * Tile_N * Tile_K并累加到C的tile reg
        K循环完成后将C的tile reg存储到L2B中
```



2.1.1.1.2 优化方式说明

为了保证上述写法在PE上的性能，需要做以下优化：

1. **循环展开**

   K维度累加过程中，每轮循环之间的load指令存在WAW依赖、上一轮循环的MMA指令和下一轮的load指令之间存在WAR依赖都会造成的流水线stall，导致访存指令无法流水发出，从而无法掩盖访存延迟的影响。因此，需要对K维度累加进行循环展开，展开次数需要保证访存延迟能够被掩盖。

   展开次数的计算公式：

   展开次数 * 两条MMA指令的计算间隔 > 源寄存器数据被TMAC读走到新的Load数据写回该寄存器的时间

   后者总延迟包括：源寄存器读取 + Load源寄存器指令OPC + 访存延迟（X） + Load数据WriteBack + 数据发送到TMAC

   例如，对于MXF6 shape32，假设数据放在L2B（延迟是18个cycle），B寄存器被读走到新的数据写回B寄存器的间隔为：A寄存器被TMAC读走（3），Load A和Load B OPC（7），Load数据写回

到再发送到TMAC（5），MMA指令的间隔为16（具体可参考TMAC MAS中性能参数列表），因此至少需要展开2次才能完全掩盖L2B的延迟。具体参考：GEMM/GEMV MXFP6性能分析 中关于展开次数计算公式的解释。

增加展开次数可能无法提升性能的情况：

在某些场景下即使当前展开次数还无法掩盖访存延迟时，继续提高展开次数不能带来性能提升，原因是当前load tag资源已用尽。根据 Memory Ordering性能评测和分析 中的分析，在访问global memory时，Normal和MXF6模式下循环展开到以下次数后性能不再提升。其中，Normal模式下shape32/16时，循环展开到16次已经可以完全掩盖global memory延迟（160个cycle），其他情况均无法完全掩盖global memory的延迟。

	Shape32	Shape16	Shape8
Normal	16	16	4
MXF6	8	4	1

2. **使用Double buffer**

   由于ReuseD使用规则的修改，在K维度累加过程中，下一轮循环的第一条MMA指令需要等前一轮循环的最后一条MMA指令完成输出才可以开始执行，从而导致由于RAW依赖造成的bubble，大小与TMAC的性能参数相关：计算延迟 - 计算间隔 + 输出延迟 + 1。根据TMAC的相关性能参数，对B矩阵中2列进行累加，可以掩盖这段bubble造成的影响。（具体参考： ReuseD修改对性能的影响）。

3. **对Load指令进行重排**

   在使用循环展开和Double buffer以后，每轮循环需要从矩阵A和B中分别读取多个数据，如果数据量非常大（比如MXF6），需要尽可能保证读数的顺序和计算使用的顺序一致，从而避免MMA指令因为RAW stall而被推迟执行（即MMA指令无法完全流水起来）。

   Load重排也可能导致性能下降：

   如果对Load指令进行重排，比如先读一个A矩阵的Tile，再读一个B矩阵的Tile，也可能打乱本来连续的访存pattern，导致访存性能下降。因此，建议仅在发现Load指令顺序对MMA流水化执行有影响时再进行调整。

4. **尽可能使用较大的thread_k**

   K维度累加结束后，需要用store指令将存放在寄存器中的累加结果写回memory，这条store指令与下一轮的第一条MMA指令之间存在WAR依赖。在矩阵切分到各个PE计算时，可以考虑尽可能不切K维度，从而减少这一类WAR空泡出现的频次。

2.1.1.1.3 主要限制

1. **重复读取导致访存带宽占用较多**

   K维度累加的写法的一个主要问题是，对于A矩阵的每一行，都需要读取B矩阵的每一列，因此存在重复读取数据的问题。以下是Normal FP16和MXF6模式下L2B访存带宽占用比例。（从这一点上

看，如果采用K维度累加，应该先将数据搬到L2B）。

	Shape32	Shape16	Shape8
Normal FP16	17.72%	45.50%	
MXF6	32.90%	46.60%	78.20%

2.1.1.1.4 代码示例

下面这段代码通过K维度累加实现输入为fp16的GEMM。在最内层循环展开了4次（这里是访问L2B，如果是Global memory至少展开16次才能掩盖160个cycle的延迟的影响），同时采用double buffer填充RAW stall。这里没有对最内层循环的Load指令进行重排。

代码块

```
//Use 32*32*32B
#define MAC_M 32
#define MAC_N 32
#define MAC_A_K 16 // 32B
#define MAC_B_K 16 // 32B

#define SIZE_TILE_REG 1024 // 1024B

typedef _Float16 float16_t;
typedef float    float32_t;
#define INPUT_DATA_TYPE float16_t
#define OUTPUT_DATA_TYPE float32_t

void gemm_simple_kernel(uint32_t thread_m,
                        uint32_t thread_n,
                        uint32_t thread_k,
                        uint64_t addr_A,
                        uint64_t addr_B,
                        uint64_t addr_C) {
     uint32_t m = 0;
     uint32_t n = 0;
     for(m = 0; m < thread_m/MAC_M; m++) {
         for(n = 0; n < thread_n/MAC_N; n=n+2) {
             // Compute the [m, n] and [m, n+1] tile in C
             GetMatrixC(m, n, thread_m, thread_n, thread_k, addr_A, addr_B,
addr_C);
         }
     }
}

void GetMatrixC(uint32_t m,
                uint32_t n,
```

```
32                  uint32_t thread_m,
33                  uint32_t thread_n,
34                  uint32_t thread_k,
35                  uint64_t addr_A,
36                  uint64_t addr_B,
37                  uint64_t addr_C) {
38       // Total number of tiles in K dimention
39       uint32_t loop_k = thread_k / MAC_A_K ;
40       // Start add for [m, 0] tile in A
41       uint64_t addr_a = addr_A + m*thread_k/MAC_A_K*SIZE_TILE_REG;
42       // Start addr for [0, n] tile in B and the one in the col next it(double
   buffer)
43       uint64_t addr_b = addr_B + n*thread_k/MAC_B_K*SIZE_TILE_REG;
44       uint64_t addr_b_1 = addr_b + thread_k/MAC_B_K*SIZE_TILE_REG;
45       // Start addr for [m, n] and [m, n+1] tile in C
46       uint64_t addr_c = addr_C + (m*(thread_n/MAC_N)+n)*
   (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
47       uint64_t addr_c_1 = addr_c + (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
48
49      // Load 4 tiles from 1 A row and 1 B col
50      Tile_SMEM_UnitStride(TMEM_LD_B32, r0, addr_a);
51      Tile_SMEM_UnitStride(TMEM_LD_B32, r1, addr_a+SIZE_TILE_REG);
52      Tile_SMEM_UnitStride(TMEM_LD_B32, r2, addr_a+2*SIZE_TILE_REG);
53      Tile_SMEM_UnitStride(TMEM_LD_B32, r3, addr_a+3*SIZE_TILE_REG);
54      Tile_SMEM_UnitStride(TMEM_LD_B32, r4, addr_b);
55      Tile_SMEM_UnitStride(TMEM_LD_B32, r5, addr_b+SIZE_TILE_REG);
56      Tile_SMEM_UnitStride(TMEM_LD_B32, r6, addr_b+2*SIZE_TILE_REG);
57      Tile_SMEM_UnitStride(TMEM_LD_B32, r7, addr_b+3*SIZE_TILE_REG);
58      // Load another 4 tiles from the next col in B
59      Tile_SMEM_UnitStride(TMEM_LD_B32, r8, addr_b1+(i+0)*SIZE_TILE_REG);
60      Tile_SMEM_UnitStride(TMEM_LD_B32, r9, addr_b1+(i+1)*SIZE_TILE_REG);
61      Tile_SMEM_UnitStride(TMEM_LD_B32, r10, addr_b1+(i+2)*SIZE_TILE_REG);
62      Tile_SMEM_UnitStride(TMEM_LD_B32, r11, addr_b1+(i+3)*SIZE_TILE_REG);
63
64      //Do MMA, accumulated to r40
65      Tile_MMA_ts3dis_reused(r40, r0, r4);
66      Tile_MMA_ts3en_reused(r40, r1, r5);
67      Tile_MMA_ts3en_reused(r40, r2, r6);
68      // For the new rule, change to ReuseD=0 for the last MMA in the loop body
69      Tile_MMA_ts3en_noreused(r40, r3, r7);
70
71      //Do MMA, accumulated to r44
72      Tile_MMA_ts3dis_reused(r44, r0, r8);
73      Tile_MMA_ts3en_reused(r44, r1, r9);
74      Tile_MMA_ts3en_reused(r44, r2, r10);
75      // For the new rule, change to ReuseD=0 for the last MMA in the loop body
76      Tile_MMA_ts3en_noreused(r44, r3, r11);
```

```
77
78      //K dimension acc for 1 row in matrix A and 1 column in matrix B
79      for (i = 4; i < loop_k; i+=4) {
80          //Load 4 tiles from 1 A row and 1 B col
81          Tile_SMEM_UnitStride(TMEM_LD_B32, r0, addr_a+(i+0)*SIZE_TILE_REG);
82          Tile_SMEM_UnitStride(TMEM_LD_B32, r1, addr_a+(i+1)*SIZE_TILE_REG);
83          Tile_SMEM_UnitStride(TMEM_LD_B32, r2, addr_a+(i+2)*SIZE_TILE_REG);
84          Tile_SMEM_UnitStride(TMEM_LD_B32, r3, addr_a+(i+3)*SIZE_TILE_REG);
85          Tile_SMEM_UnitStride(TMEM_LD_B32, r4, addr_b+(i+0)*SIZE_TILE_REG);
86          Tile_SMEM_UnitStride(TMEM_LD_B32, r5, addr_b+(i+1)*SIZE_TILE_REG);
87          Tile_SMEM_UnitStride(TMEM_LD_B32, r6, addr_b+(i+2)*SIZE_TILE_REG);
88          Tile_SMEM_UnitStride(TMEM_LD_B32, r7, addr_b+(i+3)*SIZE_TILE_REG);
89          //Read another 4 tiles from the next col in B
90          Tile_SMEM_UnitStride(TMEM_LD_B32, r8, addr_b1+(i+0)*SIZE_TILE_REG);
91          Tile_SMEM_UnitStride(TMEM_LD_B32, r9, addr_b1+(i+1)*SIZE_TILE_REG);
92          Tile_SMEM_UnitStride(TMEM_LD_B32, r10, addr_b1+(i+2)*SIZE_TILE_REG);
93          Tile_SMEM_UnitStride(TMEM_LD_B32, r11, addr_b1+(i+3)*SIZE_TILE_REG);
94
95          //Do MMA, accumulated to r40
96          Tile_MMA_ts3en_reused(r40, r0, r4);
97          Tile_MMA_ts3en_reused(r40, r1, r5);
98          Tile_MMA_ts3en_reused(r40, r2, r6);
99          // For the new rule, change to ReuseD=0 for the last MMA in the loop
    body
100         Tile_MMA_ts3en_noreused(r40, r3, r7);
101
102         //Do MMA, accumulated to r44
103         Tile_MMA_ts3en_reused(r44, r0, r8);
104         Tile_MMA_ts3en_reused(r44, r1, r9);
105         Tile_MMA_ts3en_reused(r44, r2, r10);
106         // For the new rule, change to ReuseD=0 for the last MMA in the loop
    body
107         Tile_MMA_ts3en_noreused(r44, r3, r11);
108     }
109
110     //After K dim acc, store result back
111     Tile_SMEM_UnitSride(TMEM_ST_B32, r40, addr_c, tilesize_4);
112     Tile_SMEM_UnitSride(TMEM_ST_B32, r44, addr_c_1, tilesize_4);
113     }
114
```

2.1.1.2 A列*B行

2.1.1.2.1 算法说明

（以下内容引自：GEMM/GEMV simple 2level性能分析）

for（Thread_M/Chunk_M)

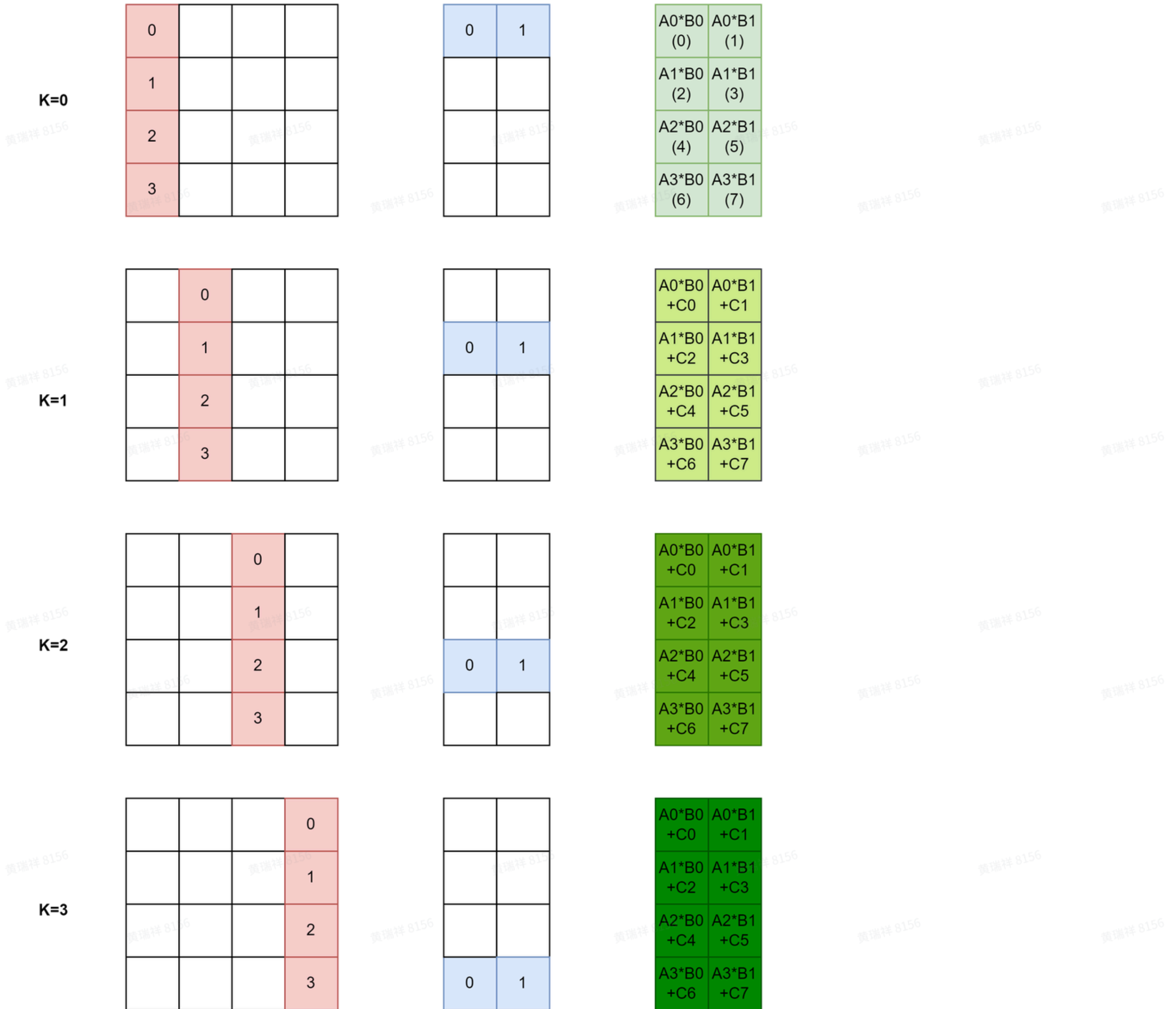
for (Thread_N/Chunk_N)

for (Thread_K/MAC_K)

从L2B搬运Chunk_M * MAC_K的A和MAC_K * Chunk_N的B到tile reg

计算Chunk_M * Chunk_N * MAC_K，并累加到C的tilereg

所有计算都完成后将Chunk_M * Chunk_N的C写回memory。



2.1.1.2.2 优化方式说明

1. **循环展开**

这一类写法相当于将double buffer进一步扩展，即每一次循环顺序计算所有的Chunk_M * Chunk_N的C tile输出，因此在最内层循环中，相邻的load、MMA指令之间，均不存在WAW依赖关系。两轮循环之间的指令仍然存在依赖关系，但是已经被间隔开，如果不足以隐藏访存延迟，则可以考虑进一步做循环展开的优化（比如使用更大的Chunk_M和Chunk_N，但是这一样依赖寄存器占用量会大规模增加，如下所示）。

2.1.1.2.3 主要限制

1. **寄存器数量使用较多问题**

   当前算法需要将Chunk_M * Chunk_N的所有的C tile输出都存储在寄存器中，直到这一部分C tile全部计算完成才能写回到memory，因此算法会使用比较多的寄存器。例如，在Normal mode和MXF6下，假设Chunk_M和Chunk_N分别为128和64时（即总共存放8个C tile输出），寄存器使用数量如下表所示：

	Shape32	Shape16	Shape8
Normal	38	44	56
MXF6	50	68	104

2.1.1.2.4 代码示例

下面这段代码通过A列*B行实现输入为fp16的GEMM。其中Chunk_M和Chunk_N分别为128和64，即总共存放8个C tile输出。

代码块

```
//Use 32*32*32B
#define MAC_M 32
#define MAC_N 32
#define MAC_A_K 16 // 32B
#define MAC_B_K 16 // 32B

#define SIZE_TILE_REG 1024 // 1024B

typedef _Float16 float16_t;
typedef float    float32_t;
#define INPUT_DATA_TYPE float16_t
#define OUTPUT_DATA_TYPE float32_t

void gemm_simple_kernel(uint32_t thread_m,
                        uint32_t thread_n,
                        uint32_t thread_k,
                        uint64_t addr_A,
                        uint64_t addr_B,
                        uint64_t addr_C) {
     uint32_t m = 0;
     uint32_t n = 0;
     for(m = 0; m < thread_m/MAC_M; m=m+4) {
         for(n = 0; n < thread_n/MAC_N; n=n+2) {
             for(k = 0; k < thread_k/MAC_A_K; k++) {
                 // Compute 8 tiles in C
                 GetMatrixC(k, m, n, thread_k, addr_A, addr_B, addr_C);
             }
```

```
28                  // Write out 8 C tiles after computation is done
29                  uint64_t addr_c = Addr_C + (m*(THREAD_N/MAC_N)+n)*
    (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
30                  uint64_t addr_c_1 = addr_c +
    (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
31                  uint64_t addr_c_2 = addr_c + (THREAD_N/MAC_N)*
    (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
32                  uint64_t addr_c_3 = addr_c_2 +
    (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
33                  uint64_t addr_c_4 = addr_c + 2*(THREAD_N/MAC_N)*
    (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
34                  uint64_t addr_c_5 = addr_c_4 +
    (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
35                  uint64_t addr_c_6 = addr_c + 3*(THREAD_N/MAC_N)*
    (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
36                  uint64_t addr_c_7 = addr_c_6 +
    (MAC_M*MAC_N)*sizeof(OUTPUT_DATA_TYPE);
37                  Tile_SMEM_UnitStride(TMEM_ST_B32,  addr_c, TILE_REG_IDX(6),
    TILE_REG_SIZE_4);
38                  Tile_SMEM_UnitStride(TMEM_ST_B32,  addr_c_1, TILE_REG_IDX(10),
    TILE_REG_SIZE_4);
39                  Tile_SMEM_UnitStride(TMEM_ST_B32,  addr_c_2, TILE_REG_IDX(14),
    TILE_REG_SIZE_4);
40                  Tile_SMEM_UnitStride(TMEM_ST_B32,  addr_c_3, TILE_REG_IDX(18),
    TILE_REG_SIZE_4);
41                  Tile_SMEM_UnitStride(TMEM_ST_B32,  addr_c_4, TILE_REG_IDX(22),
    TILE_REG_SIZE_4);
42                  Tile_SMEM_UnitStride(TMEM_ST_B32,  addr_c_5, TILE_REG_IDX(26),
    TILE_REG_SIZE_4);
43                  Tile_SMEM_UnitStride(TMEM_ST_B32,  addr_c_6, TILE_REG_IDX(30),
    TILE_REG_SIZE_4);
44                  Tile_SMEM_UnitStride(TMEM_ST_B32,  addr_c_7, TILE_REG_IDX(34),
    TILE_REG_SIZE_4);
45              }
46          }
47  }
48
49  void GetMatrixC(uint32_t k,
50                     uint32_t m,
51                     uint32_t n,
52                     uint32_t thread_k,
53                     uint64_t addr_A,
54                     uint64_t addr_B,
55                     uint64_t addr_C) {
56      // Start add for [m, i] tile in A
57      uint64_t addr_a = addr_A + m*(thread_k/MAC_A_K)*SIZE_TILE_REG +
    k*SIZE_TILE_REG;
```

```
58        uint64_t addr_a_1 = addr_a + (thread_k/MAC_A_K)*SIZE_TILE_REG;
59        uint64_t addr_a_2 = addr_a + 2*(thread_k/MAC_A_K)*SIZE_TILE_REG;
60        uint64_t addr_a_3 = addr_a + 3*(thread_k/MAC_A_K)*SIZE_TILE_REG;
61        // Start add for [i, n] tile in B
62        uint64_t addr_b = addr_B + n*(thread_k/MAC_A_K)*SIZE_TILE_REG +
    k*SIZE_TILE_REG;
63        uint64_t addr_b_1 = addr_b + (thread_k/MAC_B_K)*SIZE_TILE_REG;
64
65        // Load 4 tiles in 1 A col
66        Tile_SMEM_UnitStride(TMEM_LD_B32, addr_a, TILE_REG_IDX(0),
    TILE_REG_SIZE_1);
67        Tile_SMEM_UnitStride(TMEM_LD_B32, addr_a_1, TILE_REG_IDX(1),
    TILE_REG_SIZE_1);
68        Tile_SMEM_UnitStride(TMEM_LD_B32, addr_a_2, TILE_REG_IDX(2),
    TILE_REG_SIZE_1);
69        Tile_SMEM_UnitStride(TMEM_LD_B32, addr_a_3, TILE_REG_IDX(3),
    TILE_REG_SIZE_1);
70        // Load 2 tiles in 1 B row
71        Tile_SMEM_UnitStride(TMEM_LD_B32, addr_b, TILE_REG_IDX(4),
    TILE_REG_SIZE_1);
72        Tile_SMEM_UnitStride(TMEM_LD_B32, addr_b_1, TILE_REG_IDX(5),
    TILE_REG_SIZE_1);
73
74        //Do MMA for 8 C tiles
75        if (k == 0)
76        {
77            Tile_MMA_ts3dis_noreused(r6, r0, r4);
78            Tile_MMA_ts3dis_noreused(r10, r0, r5);
79            Tile_MMA_ts3dis_noreused(r14, r1, r4);
80            Tile_MMA_ts3dis_noreused(r18, r1, r5);
81            Tile_MMA_ts3dis_noreused(r22, r2, r4);
82            Tile_MMA_ts3dis_noreused(r26, r2, r5);
83            Tile_MMA_ts3dis_noreused(r30, r3, r4);
84            Tile_MMA_ts3dis_noreused(r34, r3, r5);
85        }
86        else
87        {
88            Tile_MMA_ts3en_noreused(r6, r0, r4);
89            Tile_MMA_ts3en_noreused(r10, r0, r5);
90            Tile_MMA_ts3en_noreused(r14, r1, r4);
91            Tile_MMA_ts3en_noreused(r18, r1, r5);
92            Tile_MMA_ts3en_noreused(r22, r2, r4);
93            Tile_MMA_ts3en_noreused(r26, r2, r5);
94            Tile_MMA_ts3en_noreused(r30, r3, r4);
95            Tile_MMA_ts3en_noreused(r34, r3, r5);
96        }
97    }
```

2.1.2 DTE边搬边算

如果直接从Global memory读数计算，由于访存延迟较长，需要循环展开多次才能掩盖访存延迟的影响。但是受限于寄存器数量和LSTag等资源的限制，实际可展开的次数可能无法掩盖访存延迟的影响。因此，建议使用DTE指令异步的将global memory的数据搬运到L2B，将数据搬运的延迟隐藏在矩阵乘法计算中。L2B的高带宽和低延迟使得MMA指令更容易流水起来，提高TMAC的利用率。

说明：以下内容引自 GEMM w/ DTE性能分析，考虑使用一个线程边搬运边计算的情况。

2.1.2.1 算法说明

base case的kernel，采取A和B均用DTE从DRAM搬到L2B，内部从L2B中读数计算，如下所示：

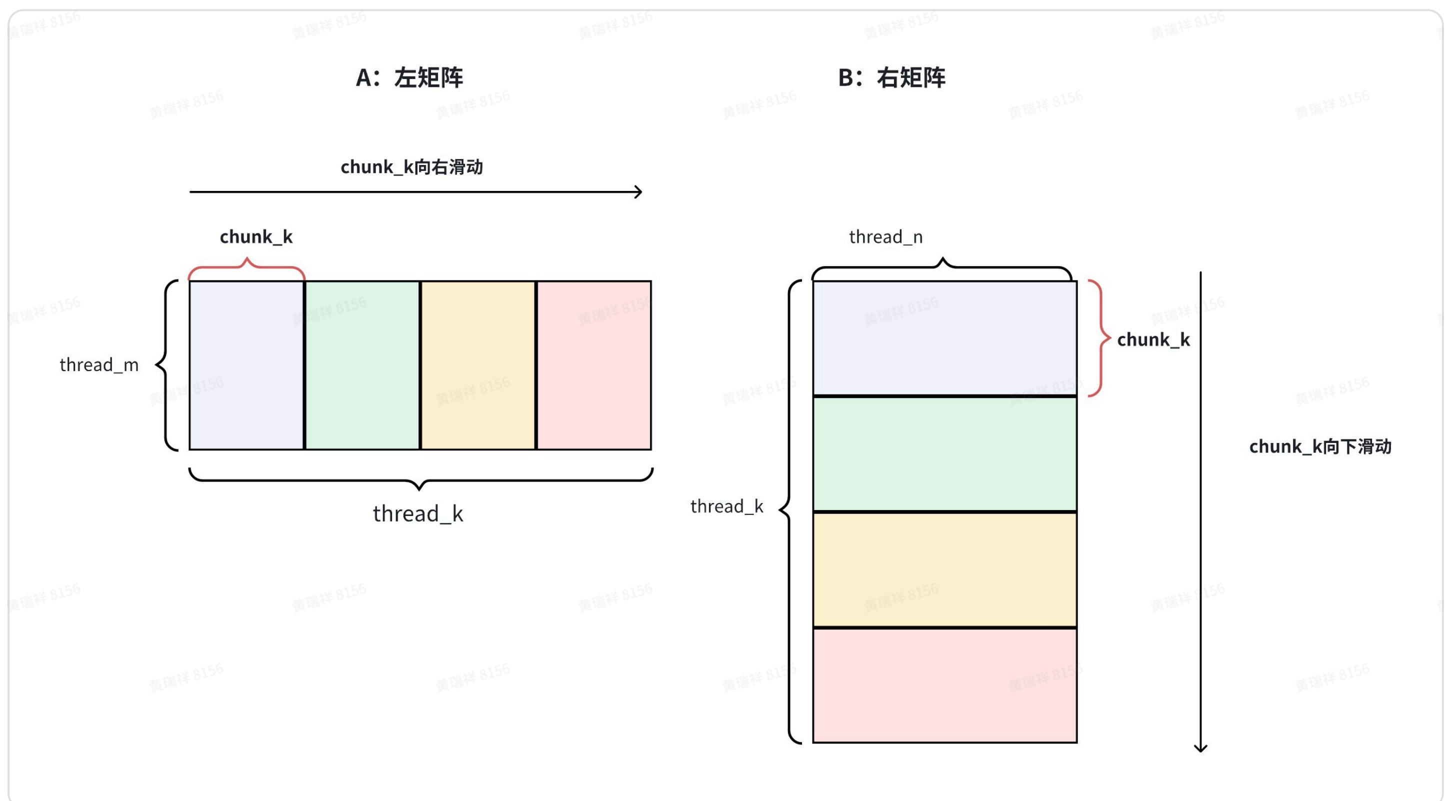
代码块

```
void gemm_with_dte_kernel(int thread_m, int thread_n, int thread_k)
{
    // 选取合适的chunk_k
    int chunk_k;
    // 提前申请好一次 thread_m * thread_n * chunk_k所需的td寄存器
    tfloat32m4_t td[];
    memset td;

    int ping = 0, pong = 1;

    // 搬ping
    TACP(a_l2b[ping], a_dram_addr + ..., thread_m * chunk_k); // ping
    TACP(b_l2b[ping], b_dram_addr + ..., thread_n * chunk_k); // ping
    tacp.commit_group;

    int kLoop = thread_k / chunk_k;
    for (int kIdx = 0; kIdx < kLoop - 1; kIdx++)
    {
        // 等inner_block_gemm函数中所有的smem.load全都commit
        twait.ls(0)
        // 搬pong
        TACP(a_l2b[pong], a_dram_addr + ..., thread_m * chunk_k); // pong
        TACP(b_l2b[pong], b_dram_addr + ..., thread_n * chunk_k); // pong
        tacp.commit_group;

        // ping的gemm计算
        twait.tacp_cg(1);
```

```
28            inner_block_gemm(thread_m, thread_n, chunk_k, td, a_l2b[ping],
    b_l2b[ping]); // ping
29
30            // ping pong switch
31            ping = (ping + 1) % 2;
32            pong = (pong + 1) % 2;
33        }
34
35        // last kLoop
36          twait.tacp_cg(0);
37          inner_block_gemm(thread_m, thread_n, chunk_k, td, a_l2b[ping],
    b_l2b[ping]);
38    }
```

1. 切分搬运块（Chunk）

为了尽量避免重复从DRAM读取数据，会尽可能从K方向切分搬运块，如果因为寄存器个数限制放不下再考虑从M或N方向切分。如果能够不对M和N切分，则A和B均只要从DRAM中读取一次。

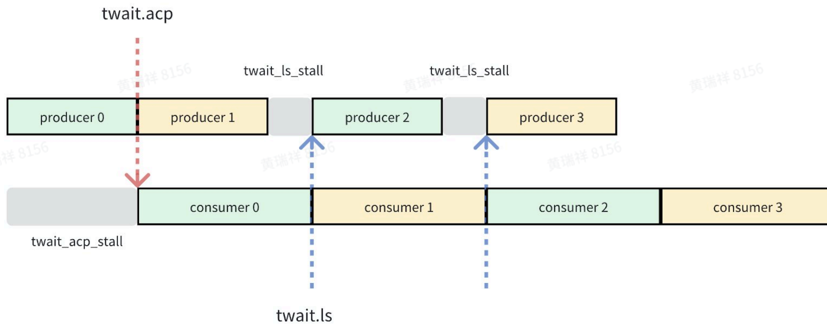


2. 使用double buffer

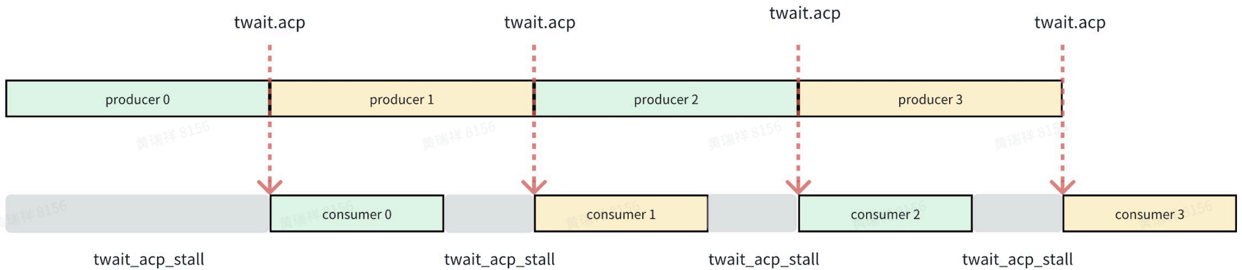
为了使得数据计算和搬运能并行，在L2B中申请double buffer来分别保存ping chunk和pong chunk的数据，以达到如下的并行效果。

每次循环前有twait.acp，后有twait.ls，下图只画出起了阻挡作用的twait指令

情况一：acp latency可以被gemm隐藏



情况二：acp latency不能被gemm隐藏



3. 块内计算

   块内计算的方式参考Simple GEMM kernel即可。

2.1.2.2 主要限制

这里会有两个资源限制，一个是L2B的容量限制，一个是寄存器个数限制。

1. L2B的容量限制

   L2B需要能够放下A Chunk + B Chunk的double buffer。

2. 寄存器个数限制

如果在K维度切了chunk，会沿着K方向滑动，依次计算 chunk_m * chunk_k * chunk_n 的矩阵乘，并将本次的结果累加到下一轮chunk_k之上。因此，我们需要将每轮的结果保存起来，以便下轮使用：

   a. 可以保存在寄存器中，下一轮mma指令直接使用
   b. 可以写出到L2B，下一轮使用TLSU从L2B读到寄存器中

   这里选择方案a，原因是：

   - 方案a可以直接利用寄存器的raw依赖来保证指令执行的正确性，即使发生stall，也是寄存器粒度的

- 方案b需要软件插入twait.store来处理，stall的粒度特别粗（除非你精心计算每个twait的count），直接从tile core 流水线源头砍断指令接收

因此，在切chunk时，还需要考虑切完的chunk_m * chunk_n 所代表的临时结果，是否能被tile core内的寄存器全都装下，并且在块内计算时还要考虑A和B所使用的寄存器。

2.1.2.3 如何设置Buffer的大小

在使用double buffer时，我们希望DTE搬运的时间与GEMM计算的时间相当或者少一些，从而保证搬运的延迟能够被隐藏，尽可能提高TMAC利用率。下面的方式可以简单进行理论时间的计算：

a. DTE搬运一个Buffer的数据所需要的时间（from intra-cluster dram to local l2b)

$$\begin{cases} \text{ost} > \text{dram_latency}: T = \text{dram_latency} + \text{request_count} \\ \text{ost} < \text{dram_latency}: T = \text{dram_latency} \times \left\lceil \frac{\text{request_count}}{\text{ost}} \right\rceil + \text{ost}) \end{cases}$$

> 1. request_count = copy_size(byte) / 256B
> 2. 查看dte的remote_outstanding(后简称为ost)和dram_latency(包括ICI和dram)的大小关系
>   a. 目前ost配置为128
>   b. intra-cluster的dram_latency目前不太确定，可能在100ns和130ns附近?
> 3. if ost > dram_latency
>
>   cycle = dram_latency + request_count
> 4. if ost < dram_latency
>   a. if request_count % ost == 0
>
>   cycle = dram_latency x ceil(request_count / ost) + ost
>   b. if request_count % ost != 0
>
>   cycle = dram_latency x ceil(request_count / ost) + (request_count % ost)
> 5. 上述计算公式将DRAM当作Simple memory，考虑到实际DRAM访问中各种行/列/Bank切换等时间，会比以上计算结果更长。

b. TMAC计算一个Buffer的GEMM所需要的时间：

(总计算量/单个Tile计算量) * 计算间隔

> 1. 总计算量：Chunk_M * Chunk_N * Chunk_K
> 2. 单个Tile计算量：Tile_M * Tile_N * Tile_K
> 3. 计算间隔可以参考：TMAC MAS中性能参数列表

> 4. 上述公式只cover了核心计算部分，没有考虑整个流水线的结束，可能小于gemm的实际运行时间

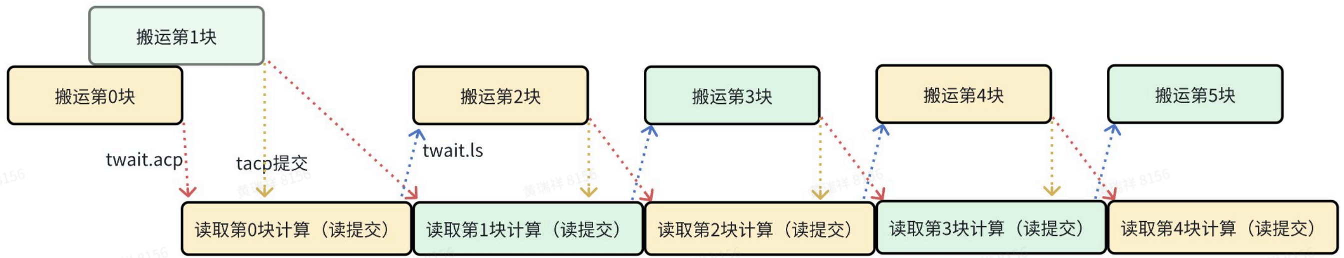
根据a和b的计算公式，我们可以得出不同DRAM延迟下DTE每次需要搬运多大的Chunk才能将搬运延迟掩盖在计算延迟下。需要注意的是，由于当前仅使用一个线程边搬边算，ACP指令与GEMM指令按序提交，当计算时间与搬运时间基本相同时，会引入【循环内指令数】个因为按序提交而引入的Bubble。因此，实际测试下DTE每次需要搬运的Chunk大小可能会比理论计算的结果大一些。例如，在Thread_M/N/K=128/64/8192（假设只切分K维度）上，计算需搬运的Chunk_K的大小和实测的大小如下表所示。

DRAM延迟	**Chunk_K(理论)**	**搬运时间（理论）**	**计算时间（理论）**	**Chunk_K（实测）**
150 cycle（100ns）	32	198 cycle	256 cycle	32
200 cycle（133ns）	32	248 cycle	256 cycle	64
300 cycle（200ns）	64	396 cycle	512 cycle	64

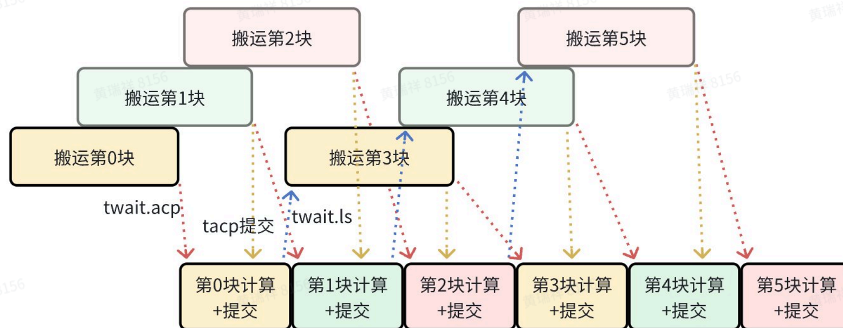
2.1.2.4 使用Multi-buffer

如上一章所示，在使用double buffer时，我们希望DTE搬运的时间与GEMM计算的时间相当或者少一些，从而保证搬运的延迟能够被隐藏。但是，如果DRAM的延迟比较长，每次搬运的Chunk比较大，会造成较高的overhead（如上表所示）。此外，DRAM的延迟往往不是一个固定的数值，可能是在某一个范围内变化，导致难以设置合适的Chunk大小。为了解决这个问题，我们可以使用multi-buffer。如下图3-buffer的例子所示，与double buffer不同，在第0块开始计算时，第2块可以立刻开始搬运。假设计算时间比搬运时间的1/2更长一些，则在第0块计算完成时，第2块已经搬运了>1/2的大小，继而在第1块计算完成时，第2块已完成搬运，可以开始计算。因此，在3-buffer时，我们只需要计算时间比1/2的搬运时间更长一些，即可掩盖搬运延迟。如果推广至multi-buffer，我们可以根据DRAM的延迟范围以及一个最小Chunk（例如只是A的一列和B的一行）的搬运和计算时间（详见上一章公式）设定multi-buffer的buffer个数，从而保证将搬运延迟掩盖在计算延迟内，并尽可能减少overhead。

Double Buffer：由于计算只能与单个buffer的搬运重叠，需要计算比搬运时间更长一点，从而掩盖搬运延迟



Multi-Buffer(3 buffer)：由于计算可以与2个buffer的搬运重叠，计算只需要比1/2的搬运时间更长一点，即可掩盖搬运延迟



2.1.2.5 代码示例：边搬边算，double buffer

备注：

1. 由于没有使用后端更新的编译器，tile寄存器全都是手动指令编号
2. 没有使用driver API，tensormap手动构造
3. kernel入口在*Gemm_Kernel_DTE()*
4. 使用KernelParam中的3个参数分别代表本kernel的m/n/k，为简化示例代码
   a. 将chunk_m和chunk_n的取值和thread_m和thread_n保持一致
   b. 固定chunk_m = 32, chunk_n = 128

- 文件1：tensormap构造，示例中为手动构造

代码块

```
1  // Generate coordinates in DTE transfer
2  static inline vuint32m1_t GenCoord(uint32_t coord0, uint32_t coord1)
3  {
4      size_t vl = __riscv_vsetvl_e32m1(32);
5      vuint32m1_t v0 = __riscv_vmv_v_x_u32m1(0, vl);
6
7      vuint32m1_t va = __riscv_vslide1up_vx_u32m1(v0, coord1, vl);
```

```
    va = __riscv_vslide1up_vx_u32m1(va, coord0, vl);
    return va;
}

// Generate tensormap for DTE transfer
static inline vuint32m1_t GenTensormap(uint64_t addr, uint32_t globalDim0, uint32_t globalDim1, uint32_t boxDim0, uint32_t boxDim1)
{
    uint32_t data[32] = {0};

    union TensorMapFields* tensormap = (union TensorMapFields*)data;
    tensormap->mixedFields.tensorDataType = 5; //TMAP_DTYPE_FP16
    tensormap->mixedFields.tensorRank = 2;
    tensormap->mixedFields.tensorTiled = 1;

    tensormap->mixedFields.tileDim[0] = 32;
    tensormap->mixedFields.tileDim[1] = 16;

    tensormap->mixedFields.tileHeaderSize = 0;
    tensormap->mixedFields.tileDataSize = 1024;

    tensormap->mixedFields.superTileDim[0] = 1;
    tensormap->mixedFields.superTileDim[1] = 1;

    tensormap->mixedFields.globalAddress = addr;

    tensormap->mixedFields.globalDim[0] = globalDim0;
    tensormap->mixedFields.globalDim[1] = globalDim1;
    tensormap->mixedFields.globalDim[2] = 1;
    tensormap->mixedFields.globalDim[3] = 1;
    tensormap->mixedFields.globalDim[4] = 1;

    tensormap->mixedFields.globalStrides[0] = globalDim0;
    tensormap->mixedFields.globalStrides[1] = globalDim1 * globalDim0;
    tensormap->mixedFields.globalStrides[2] = 1;
    tensormap->mixedFields.globalStrides[3] = 1;
    tensormap->mixedFields.globalStrides[4] = 1;

    tensormap->mixedFields.boxDim[0] = boxDim0;
    tensormap->mixedFields.boxDim[1] = boxDim1;
    tensormap->mixedFields.boxDim[2] = 1;
    tensormap->mixedFields.boxDim[3] = 1;
    tensormap->mixedFields.boxDim[4] = 1;

    tensormap->mixedFields.oobFillValue = 0;

    size_t vl = __riscv_vsetvl_e32m1(32);
```

```
54      vuint32m1_t src_tensormap = __riscv_vle32_v_u32m1(tensormap, vl);
55
56      return src_tensormap;
57  }
58
```

- 文件2：GEMM kernel

代码块

```
1   #include "ace_inst.h"
2   #include "ace_user.h"
3   #include <stdio.h>
4   #include "cJSON.h"
5   #include <assert.h>
6   #include "kernel_comm_api.h"
7
8   typedef _Float16 float16_t;
9   typedef float    float32_t;
10  #define INPUT_DATA_TYPE float16_t
11  #define OUTPUT_DATA_TYPE float32_t
12
13  #define TILE_BYTE (1024)
14  #define UNIT_STORE
15
16  #define LOAD_MULTI
17
18  typedef struct KernelParam
19  {
20      int batchSize;
21      int hiddenDim;
22      int headDim;
23  } KernelParam;
24
25  KernelParam g_kernelParam;
26
27
28  // Generate result address to store back
29  static inline uint32_t GetRsltAddrOffset(int mTileIdx, int nTileIdx, int
    dTileIdx)
30  {
31      // uint32_t linearOffset = g_kernelParam.rsltOffsetInKBuffer + nTileIdx *
    4 * 32 + dTileIdx * 32;
32      uint32_t tileIdx = nTileIdx * 4 + dTileIdx;
33      if (g_kernelParam.vectorFormat == true) {
34          return g_kernelParam.rsltOffsetInKBuffer + tileIdx * 32;
```

```
35        } else {
36            return g_kernelParam.rsltOffsetInKBuffer + tileIdx * TILE_BYTE;
37        }
38    }
39
40    // -----------------Inner GEMM compute functions start --------------------
41    // 这里为了简单示例，只将td寄存器传入函数中，其余寄存器在函数内分配即可，该代码暂时不能编译
42    void TileInnerProduct(uint64_t a_l2bAddr, uint64_t b_l2bAddr, bool is_firstK, tfloat32m4_t td)
43    {
44        Tile_SMEM_UnitStride(TMEM_LD_B32, a_l2bAddr, TILE_REG_IDX(0), TILE_SIZE_4);
45            Tile_SMEM_UnitStride(TMEM_LD_B32, b_l2bAddr, TILE_REG_IDX(100), TILE_SIZE_4);
46
47        if (is_firstK) {
48            Tile_MMASync_Resued_Ts3Dis(TMMA_f32_f16f16,
49                TILE_REG_IDX(0), TILE_REG_IDX(100), td, //ts1Idx, ts2Idx, tdIdx
50                TILE_SIZE_1,
51                TMMA_shape32);
52        } else {
53            Tile_MMASync_Resued_Ts3en(TMMA_f32_f16f16,
54                TILE_REG_IDX(0), TILE_REG_IDX(100), td, //ts1Idx, ts2Idx, tdIdx
55                TILE_SIZE_1,
56                TMMA_shape32);
57        }
58
59
60        Tile_MMASync_Resued_Ts3en(TMMA_f32_f16f16,
61            TILE_REG_IDX(1), TILE_REG_IDX(101), TILE_REG_IDX(200), //ts1Idx, ts2Idx, tdIdx
62            TILE_SIZE_1,
63            TMMA_shape32);
64
65        Tile_MMASync_Resued_Ts3en(TMMA_f32_f16f16,
66            TILE_REG_IDX(2), TILE_REG_IDX(102), td, //ts1Idx, ts2Idx, tdIdx
67            TILE_SIZE_1,
68            TMMA_shape32);
69
70        Tile_MMASync_NoResue_Ts3en(TMMA_f32_f16f16,
71            TILE_REG_IDX(3), TILE_REG_IDX(103), td, //ts1Idx, ts2Idx, tdIdx
72            TILE_SIZE_1,
73            TMMA_shape32);
74    }
75
76    //-----------Inner GEMM compute functions end----------------
77
```

```
78  void Gemm_Kernel_Store(uint64_t c_addr)
79  {
80      uint64_t c_tempAddr;
81
82      c_tempAddr = c_addr;
83      Tile_GMEM_UnitStride(TMEM_ST_B32, c_tempAddr, TILE_REG_IDX(200),
    TILE_SIZE_4);
84
85      c_tempAddr += 4 * TILE_BYTE;
86      Tile_GMEM_UnitStride(TMEM_ST_B32, c_tempAddr, TILE_REG_IDX(204),
    TILE_SIZE_4);
87
88      c_tempAddr += 4 * TILE_BYTE;
89      Tile_GMEM_UnitStride(TMEM_ST_B32, c_tempAddr, TILE_REG_IDX(208),
    TILE_SIZE_4);
90
91      c_tempAddr += 4 * TILE_BYTE;
92      Tile_GMEM_UnitStride(TMEM_ST_B32, c_tempAddr, TILE_REG_IDX(212),
    TILE_SIZE_4);
93  }
94
95  // DTE kernel top
96  uint64_t Gemm_Kernel_DTE(int thread_m, int thread_n, int thread_k, uint64_t
    a_dramAddr, uint64_t b_dramAddr, uint64_t c_dramAddr,
97      uint64_t a_l2bAddr, uint64_t b_l2bAddr, uint32_t chunk_k)
98  {
99      int mac_m = 32;
100     int mac_n = 32;
101     int mac_k = 16;
102
103     int mLoop = thread_m / mac_m;
104     int nLoop = thread_n / mac_n;
105
106     uint64_t a_l2b_tempAddr, b_l2b_tempAddr;
107
108     int chunk_k_tile = chunk_k / mac_k;
109     int kLoop = thread_k / chunk_k;
110
111     size_t vl = __riscv_vsetvl_e32m1(32);
112     vuint32m1_t src_tensormap_a = GenTensormap(a_dramAddr, thread_k / mac_k,
    mLoop, chunk_k_tile, mLoop);
113     vuint32m1_t src_tensormap_b = GenTensormap(b_dramAddr, thread_k / mac_k,
    nLoop, chunk_k_tile, nLoop);
114     vuint32m1_t coord_a, coord_b;
115
116     uint64_t a_chunk_size = thread_m * chunk_k * sizeof(INPUT_DATA_TYPE);
117     uint64_t b_chunk_size = thread_n * chunk_k * sizeof(INPUT_DATA_TYPE);
```

```
118
119     uint64_t a_l2bAddr_pp[2] = {a_l2bAddr, a_l2bAddr + a_chunk_size};
120     uint64_t b_l2bAddr_pp[2] = {b_l2bAddr, b_l2bAddr + b_chunk_size};
121
122     int ping_idx = 0, pong_idx = 1;
123
124     // 提前申请好4个td寄存器，其个数和thread_m * thread_n有关，在本case中按照thread_m为32，thread_n为128来写
125     tfloat32m4_t td[4];
126
127     uint64_t a_l2bAddr_ping = a_l2bAddr + ping_idx * a_chunk_size;
128     uint64_t b_l2bAddr_ping = b_l2bAddr + ping_idx * b_chunk_size;
129     uint64_t a_l2bAddr_pong = a_l2bAddr + pong_idx * a_chunk_size;
130     uint64_t b_l2bAddr_pong = b_l2bAddr + pong_idx * b_chunk_size;
131
132     coord_a = GenCoord(0, 0);
133     Tile_DTE_Tensor_TensorChunk_BulkGroup(src_tensormap_a, coord_a, vl, a_l2bAddr_ping, TDTE_LOC_REMOTE, TDTE_LOC_LOCAL, TDTE_Tensor_Tensor, 0);
134     coord_b = GenCoord(0, 0);
135     Tile_DTE_Tensor_TensorChunk_BulkGroup(src_tensormap_b, coord_b, vl, b_l2bAddr_ping, TDTE_LOC_REMOTE, TDTE_LOC_LOCAL, TDTE_Tensor_Tensor, 0);
136
137     Tacp_Commit_Group();
138
139     for (int tempK = 0; tempK < kLoop - 1; tempK++) {
140         ace_twait_ls(0, 1, 0);
141
142         coord_a = GenCoord((tempK + 1) * chunk_k_tile, 0);
143         Tile_DTE_Tensor_TensorChunk_BulkGroup(src_tensormap_a, coord_a, vl, a_l2bAddr_pong, TDTE_LOC_REMOTE, TDTE_LOC_LOCAL, TDTE_Tensor_Tensor, 0);
144         coord_b = GenCoord((tempK + 1) * chunk_k_tile, 0);
145         Tile_DTE_Tensor_TensorChunk_BulkGroup(src_tensormap_b, coord_b, vl, b_l2bAddr_pong, TDTE_LOC_REMOTE, TDTE_LOC_LOCAL, TDTE_Tensor_Tensor, 0);
146
147         Tacp_Commit_Group();
148         ace_twait_tacp_cg(1);
149
150         a_l2b_tempAddr = a_l2bAddr_ping;
151         b_l2b_tempAddr = b_l2bAddr_ping;
152
153         for (int i = 0; i < chunk_k / 64; i++) {
154             a_l2b_tempAddr = a_l2bAddr_ping + i * mac_m * 64 * sizeof(INPUT_DATA_TYPE);
155             b_l2b_tempAddr = b_l2bAddr_ping + i * mac_n * 64 * sizeof(INPUT_DATA_TYPE);
156
```

```
157                 TileInnerProduct(a_l2b_tempAddr, b_l2b_tempAddr, tempK == 0 && i == 0, td[0]);
158
159                 uint64_t b_l2b_tempAddr_1 = b_l2b_tempAddr + mac_n * chunk_k * sizeof(INPUT_DATA_TYPE);
160                 TileInnerProduct(a_l2b_tempAddr, b_l2b_tempAddr_1, tempK == 0 && i == 0, td[1]);
161
162                 uint64_t b_l2b_tempAddr_2 = b_l2b_tempAddr + 2 * mac_n * chunk_k * sizeof(INPUT_DATA_TYPE);
163                 TileInnerProduct(a_l2b_tempAddr, b_l2b_tempAddr_2, tempK == 0 && i == 0, td[2]);
164
165                 uint64_t b_l2b_tempAddr_3 = b_l2b_tempAddr + 3 * mac_n * chunk_k * sizeof(INPUT_DATA_TYPE);
166                 TileInnerProduct(a_l2b_tempAddr, b_l2b_tempAddr_3, tempK == 0 && i == 0, td[3]);
167             }
168
169             ping_idx = (ping_idx + 1) % 2;
170             pong_idx = (pong_idx + 1) % 2;
171
172             a_l2bAddr_ping = a_l2bAddr + ping_idx * a_chunk_size;
173             b_l2bAddr_ping = b_l2bAddr + ping_idx * b_chunk_size;
174             a_l2bAddr_pong = a_l2bAddr + pong_idx * a_chunk_size;
175             b_l2bAddr_pong = b_l2bAddr + pong_idx * b_chunk_size;
176         }
177
178         ace_twait_tacp_cg(0);
179
180         for (int i = 0; i < chunk_k / 64; i++) {
181             a_l2b_tempAddr = a_l2bAddr_ping + i * mac_m * 64 * sizeof(INPUT_DATA_TYPE);
182             b_l2b_tempAddr = b_l2bAddr_ping + i * mac_n * 64 * sizeof(INPUT_DATA_TYPE);
183
184             TileInnerProduct_Func_Table[0](a_l2b_tempAddr, b_l2b_tempAddr, false);
185
186             TileInnerProduct_Func_Table[1](a_l2b_tempAddr, b_l2b_tempAddr + mac_n * chunk_k * sizeof(INPUT_DATA_TYPE), false);
187
188             TileInnerProduct_Func_Table[2](a_l2b_tempAddr, b_l2b_tempAddr + 2 * mac_n * chunk_k * sizeof(INPUT_DATA_TYPE), false);
189
190             TileInnerProduct_Func_Table[3](a_l2b_tempAddr, b_l2b_tempAddr + 3 * mac_n * chunk_k * sizeof(INPUT_DATA_TYPE), false);
191         }
```

```
192
193         Gemm_Kernel_Store(c_dramAddr);
194
195         LS_Wait(1, 1, 0);
196         LS_Wait(1, 0, 0);
197         uint64_t tmask_value = ace_tcsrr_r(0x0, 0);
198         return tmask_value;
199     }
```

2.1.2.6 代码示例：Multi-buffer

代码块

```
1   #define BufferSize 3 // Assume use 3 buffer
2   #define ChunkLoadNum 6 //Assume 6 loads in inner GEMM for 1 Chunk
3
4   // DTE kernel top
5   uint64_t Gemm_Kernel_DTE(int thread_m, int thread_n, int thread_k, uint64_t
    a_dramAddr, uint64_t b_dramAddr, uint64_t c_dramAddr,
6       uint64_t a_l2bAddr, uint64_t b_l2bAddr, uint32_t chunk_k)
7   {
8       int mac_m = 32;
9       int mac_n = 32;
10      int mac_k = 16;
11
12      int mLoop = thread_m / mac_m;
13      int nLoop = thread_n / mac_n;
14
15      uint64_t a_l2b_tempAddr, b_l2b_tempAddr;
16      uint32_t l2b_buffer_id;
17
18      int chunk_k_tile = chunk_k / mac_k;
19      int kLoop = thread_k / chunk_k;
20
21      // Set source tensormap
22      size_t vl = __riscv_vsetvl_e32m1(32);
23      vuint32m1_t src_tensormap_a = GenTensormap(a_dramAddr, thread_k / mac_k,
    mLoop, chunk_k_tile, mLoop);
24      vuint32m1_t src_tensormap_b = GenTensormap(b_dramAddr, thread_k / mac_k,
    nLoop, chunk_k_tile, nLoop);
25      vuint32m1_t coord_a, coord_b;
26
27      uint64_t a_chunk_size = thread_m * chunk_k * sizeof(INPUT_DATA_TYPE);
28      uint64_t b_chunk_size = thread_n * chunk_k * sizeof(INPUT_DATA_TYPE);
29
30      // Transfer 1 chunk of A and B from DRAM to L2B
```

```
31        a_l2b_tempAddr= a_l2bAddr + l2b_buffer_id * a_chunk_size;
32        b_l2b_tempAddr= b_l2bAddr + l2b_buffer_id * b_chunk_size;
33        l2b_buffer_id = (l2b_buffer_id + 1) % BufferSize;
34        coord_a = GenCoord(0, 0);
35        Tile_DTE_Tensor_TensorChunk_BulkGroup(src_tensormap_a, coord_a, vl,
    a_l2b_tempAddr, TDTE_LOC_REMOTE, TDTE_LOC_LOCAL, TDTE_Tensor_Tensor, 0);
36        coord_b = GenCoord(0, 0);
37        Tile_DTE_Tensor_TensorChunk_BulkGroup(src_tensormap_b, coord_b, vl,
    b_l2b_tempAddr, TDTE_LOC_REMOTE, TDTE_LOC_LOCAL, TDTE_Tensor_Tensor, 0);
38
39        Tacp_Commit_Group();
40
41        for (int tempK = 0; tempK < kLoop - 1; tempK++) {
42
43            // Transfer 1 chunk of A and B from DRAM to L2B
44            a_l2b_tempAddr= a_l2bAddr + l2b_buffer_id * a_chunk_size;
45            b_l2b_tempAddr= b_l2bAddr + l2b_buffer_id * b_chunk_size;
46            l2b_buffer_id = (l2b_buffer_id + 1) % BufferSize;
47            coord_a = GenCoord((tempK + 1) * chunk_k_tile, 0);
48            Tile_DTE_Tensor_TensorChunk_BulkGroup(src_tensormap_a, coord_a, vl,
    a_l2bAddr_tempAddr, TDTE_LOC_REMOTE, TDTE_LOC_LOCAL, TDTE_Tensor_Tensor, 0);
49            coord_b = GenCoord((tempK + 1) * chunk_k_tile, 0);
50            Tile_DTE_Tensor_TensorChunk_BulkGroup(src_tensormap_b, coord_b, vl,
    b_l2bAddr_tempAddr, TDTE_LOC_REMOTE, TDTE_LOC_LOCAL, TDTE_Tensor_Tensor, 0);
51            Tacp_Commit_Group();
52
53            // Wait for 1 buffer transfer done
54            ace_twait_tacp_cg(1);
55
56            // Inner GEMM compute for 1 buffer
57            ...
58
59            // Wait for loads in the buffer to be transferred done
60            ace_twait_ls(0, 1, ChunkLoadNum);
61        }
62
63        ace_twait_tacp_cg(0);
64
65        // Inner GEMM compute for last Chunk of A and B
66        ...
67
68        // Write back C tile outputs to DRAM
69        Gemm_Kernel_Store(c_dramAddr);
70
71        LS_Wait(1, 1, 0);
72        LS_Wait(1, 0, 0);
73        uint64_t tmask_value = ace_tcsrr_r(0x0, 0);
```

```
74         return tmask_value;
75     }
```

2.1.3 Index load/store（Batched GEMV in k projection）

2.1.3.1 指令说明

和普通的linear load/store比，index load/store指的是对32个32B数据分别指定地址。此时带来的最直观的问题是，其tlsu的访存带宽直接减半(总线数据位宽为64B)，每个port每拍处理的有效数据从64B降到32B。

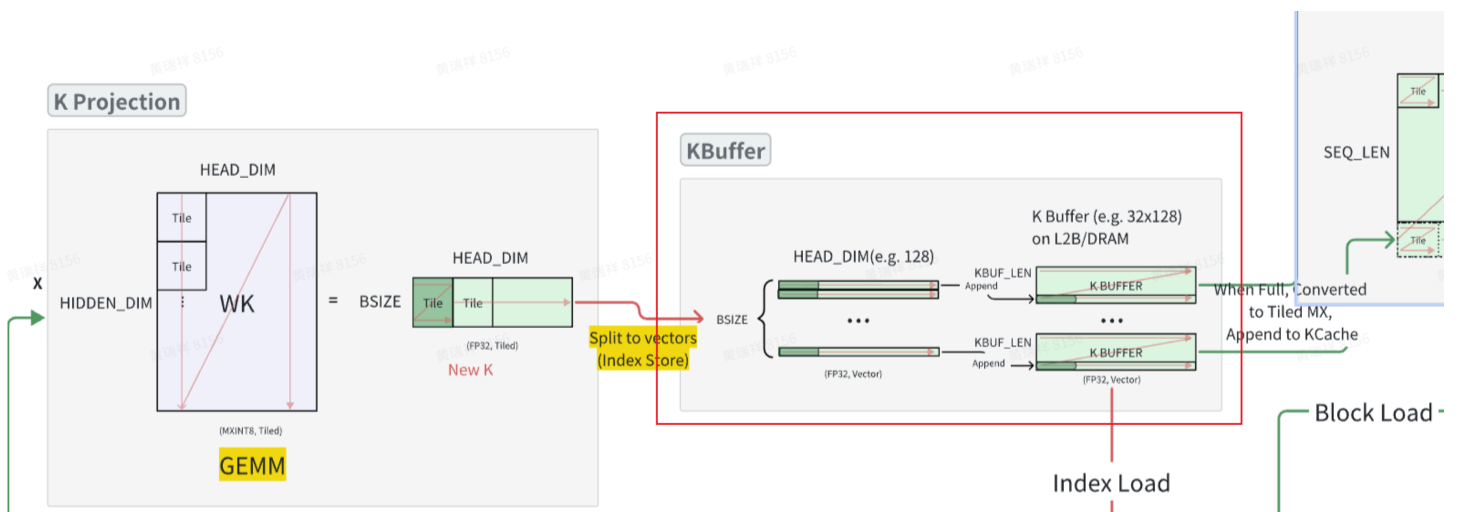
另外一个隐藏的问题是，这种方式会增加TLSU发送请求时的port冲突。TLSU内部，会根据请求的地址，将其交织到对应的port，同Port的请求需要串行执行，每个port的处理是相互独立的：

1. 对于L2B的访问，根据addr_bit[8:6](低6bit为64B的传输带宽)选出share memory的8个port
2. 对于dram的访问，根据addr_bit[7+X: 6+X]选出global memory的4个port
   a. X = log_2(burst_length)
   b. burst_length = 1,2,4,8

最差情况下，这32个地址会交织到同一个port上，使得LSU的带宽降至32B/cycle

2.1.3.2 场景说明

如下图所示，在计算完K之后，需要将数据存储到各自batch的K buffer中，并在下一个步骤中当做GEMM或者GEMV的B矩阵，也即下一步计算需要其为tile format



此时计算结果需要使用index store来存到各个batch各自对应的kbuffer。

下图为batch32时，前8条gmem.store指令的32个32B选到的LSU port，L2B是类似的，只不过是由4个port变为8个port：

vector format

dram port tileIdx	batch索引 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
6	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
7	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

tile format

dram port tileIdx	batch索引 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

需要对上图做如下几个说明：

a. 由于总线数据宽度为64B，因此tileIdx_0和tileIdx_1同batch的两个32B，一定选中的是同一个port

b. tile format全选到0，是因为我现在假设都写的是kbuffer的第一笔数据，且kbuffer的起始地址对齐到256B；实际上每次decode可以轮流去到不同的port

2.1.3.3 bank冲突解决思路

只要改变kbuffer之间的地址间隔，使得不同batch相同位置的数据，去到不同的port，就可以解决该问题。

这里采用的方式是：

> kbuffer_batch1_addr = kbuffer_batch0_addr + kbuffer_size + extra_offset
>
> extra_offset = 64B

此时的port分配情况如下所示：

vector format, extra_addr_offset = 64B

dram port tileIdx	batch索引 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
1	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
2	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0
3	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0
4	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1
5	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1
6	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2
7	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2

tile format, extra_addr_offset = 64B

dram port tileIdx	batch索引 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
1	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
2	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
4	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
5	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
6	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
7	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

可以看到，在增加额外的地址偏移后，就可以使得同一条指令内的所有请求port均匀分布在4个上，此时性能也有大幅度提升，比较接近unit_store。不如unit store的原因是，index_store最好情况下的带宽是unit_store的一半。

2.1.3.4 index构造示例

- 方式一：使用uint32数组来构造

代码块

```
1   static inline vuint32m1_t CalcIndexStride_mode1(size_t vl)
2   {
3       uint32_t stride = g_kernelParam.kBufferExtraOffset +
    g_kernelParam.kBufferSizeBytes;
4       stride = (stride >> 5);
5       uint64_t buffer_offset = g_kernelParam.vectorFormat == true ? 1 :
    (TILE_BYTE >> 5);
6
7       uint32_t index_stride[32];
8       for (int i = 0; i < 8; i++) {
9           uint32_t batch_offset = i * stride;
10          for (int j = 0; j < 4; j++) {
11              index_stride[j * 8 + i] = batch_offset + j * buffer_offset;
12          }
13      }
14
15      vuint32m1_t va = __riscv_vle32_v_u32m1(index_stride, vl);
16      return va;
17  }
18
```

- 方式二：使用rvv的vector move和vector slide指令来构造

代码块

```
1   static inline vuint32m1_t CalcIndexStride_mode2(size_t vl)
2   {
3       uint32_t stride = g_kernelParam.kBufferExtraOffset +
    g_kernelParam.kBufferSizeBytes;
4       stride = (stride >> 5);
5       uint64_t buffer_offset = g_kernelParam.vectorFormat == true ? 1 :
    (TILE_BYTE >> 5);
6
7       vuint32m1_t va;
8
9       uint32_t index_stride = 3 * buffer_offset + 7 * stride;
10      va = __riscv_vmv_s_x_u32m1(index_stride, vl);
```

```
    for (int i = 30; i >= 0; i--) {
        int out_batch = i / 8;
        int batchIdx = i % 8;
        index_stride = out_batch * buffer_offset + batchIdx * stride;
        va = __riscv_vslide1up_vx_u32m1(va, index_stride, vl);
    }
    return va;
}
```

备注：如果index是有符号数，直接改成vint32m1_t即可

3. 讨论

针对当前还未评测的一些场景中的问题进行分析。

3.1 1个PE的2个RVCore如何配合

问题1：1个PE中有两个RVCore可以同时向一个TileCore发射指令。在编程过程中，即便单个RVCore+TileCore在GEMM执行中有很多bubble，两个RVCore一起将TMAC利用率打满是否就可以?

我的理解是，这样无法高效的利用PE核的各种计算和通信资源。

一个TileCore中除了TMAC，还有TALU，TSFU，TDTE等多种计算和通信资源。如果2个RVCore才能将TMAC打满，那么在TMAC执行过程中，其他资源完全是闲置的状态。如果能够通过一个RVCore将TMAC利用率打满，我们就可以充分利用2个RVCore的SMT2的能力，将TileCore中TMAC和其他计算/通信资源同时用起来，从而将其他计算和通信的延迟掩藏在GEMM的计算延迟下（通常情况下TMAC运算是主要瓶颈）。

3.2 4个PE如何配合完成GEMM

问题1：如何将一个大矩阵运算切分到一个PEC上的4个PE进行配合?

如果是在一个PEC内部，矩阵切分是不是始终在M或N维度，相当于每个PE算一部分，最后结果在local的DRAM中拼成完整的结果？如果是切K维度，还需要额外多一步reduce的操作。如果是这样，可以先算一下4个PE配合时的最好和最坏情况，最好情况是每个PE仍然都能拿到256B/cycle的带宽，最坏情况是4个PE在port访问或者DRAM访问时完全冲突，带宽会降到多少（20%？）？这时PE边搬边算还能做到吗？是不是可以在边搬边算的case上先试一下?

问题2：换一种情况，如果是uSiPU定义为4个PEC，应该如何切分更合适?