上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion

黄子昱

2025年02月28日

饮 水 思 源 • 爱 国 荣 校

一、motivation

TABLE I

THE COMPUTE/MEMORY BREAKDOWN OF ML MODELS AND THE PERFORMANCE OF DIFFERENT ACCELERATORS.

ML Model Breakdown			
Name	**%MI**	**%CI**	**%BMM**
Transformer	19.45%	40.51%	40.04%
Bert-Base	30.56%	42.79%	26.65%
ViT-Huge	15.63%	50.85%	33.52%
Compute and Memory Characteristics of Accelerators			
Device	**Xeon Gold**	**A100**	**Ascend 910**
Dedicated Unit	AVX-512	Tensor Core	Cube Unit
Peak Perf.	12 TFlops	312 TFlops	320 TFlops
Memory BW.	131 GB/s	1555 GB/s	1200 GB/s
Peak Perf/BW	92 Flop/byte	200 Flop/byte	267 Flop/byte

%MI 指的是所有内存密集型运算符占执行时间的比例；

%CI 表示除关注层中批量 GEMM 外的计算密集型操作符的比例；

%BMM 表示批处理 GEMM（其实现受内存约束）所占的比例。

从表中可以看出，受内存限制的批处理 GEMM 占据了很大的执行时间比例（26.65% ~ 40.04%），超过了其他内存密集型操作。

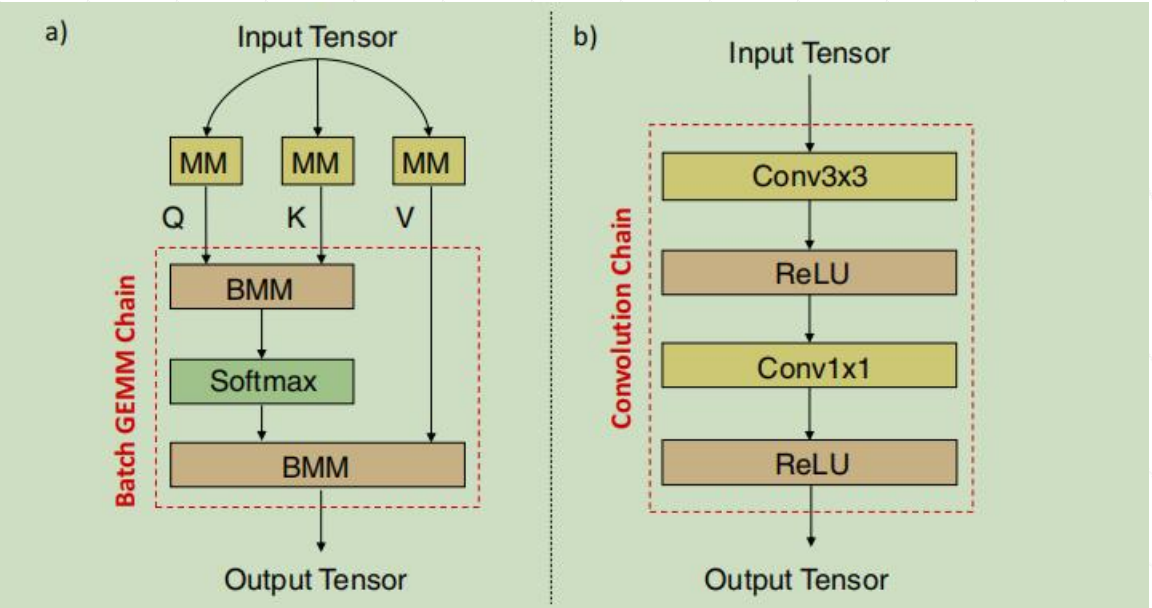
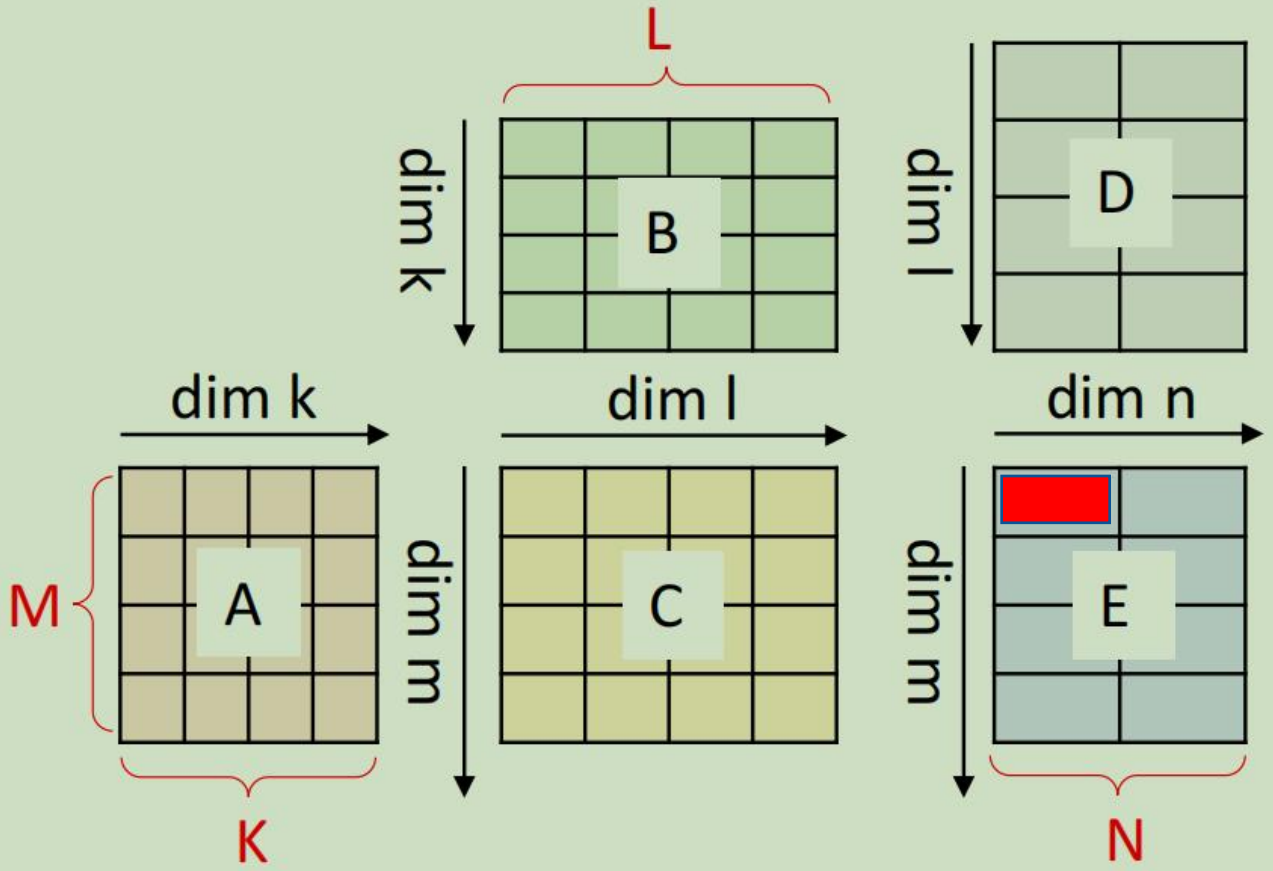


Fig. 1. Typical tensor operators in machine learning. a) batch GEMM chains from Transformers. b) convolution chains from CNNs.

在本文中，我们提出了 Chimera，这是一个机器学习模型的优化框架，可以为计算密集型算子链生成高性能的融合内核。Chimera 将计算密集型算子链分解为计算块，其优化分为两个方面：块间优化 和 块内优化。

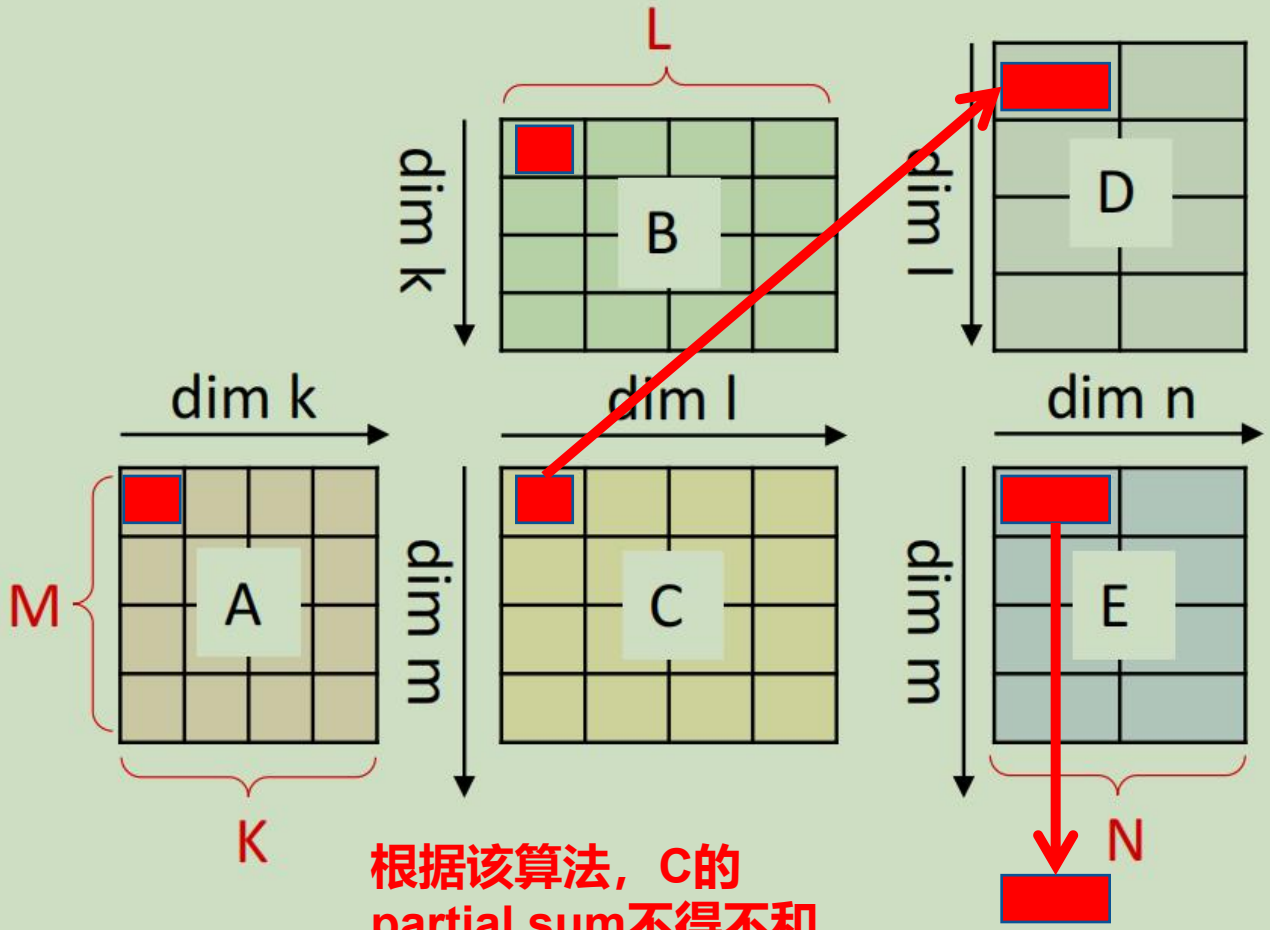
对于块间优化，Chimera 通过最小化数据移动量（最大化数据局部性）来优化块执行顺序。具体来说，Chimera 枚举了不同的块执行顺序，并分析估计了块之间的输入/输出数据移动量。之后，Chimera 选择给出最小数据移动量的执行顺序，从而实现最佳的数据位置。

二、design



		Reuse Dimension			
No.	order	A	B	D	E
1	mnkl	l	-	k	k,l
2	mnlk	-	-	k	k,l
3	mknl	n,l	-	k	k,l
4	mkln	l,n	n	k	k
5	mlnk	-	-	k	k
6	mlkn	n	n	k	k
⋮	⋮	⋮		⋮	⋮
24	lknm	-	n,m	m,k	m,k

Fig. 2. Different block execution orders result in different data reuse (use GEMM chain as an example).



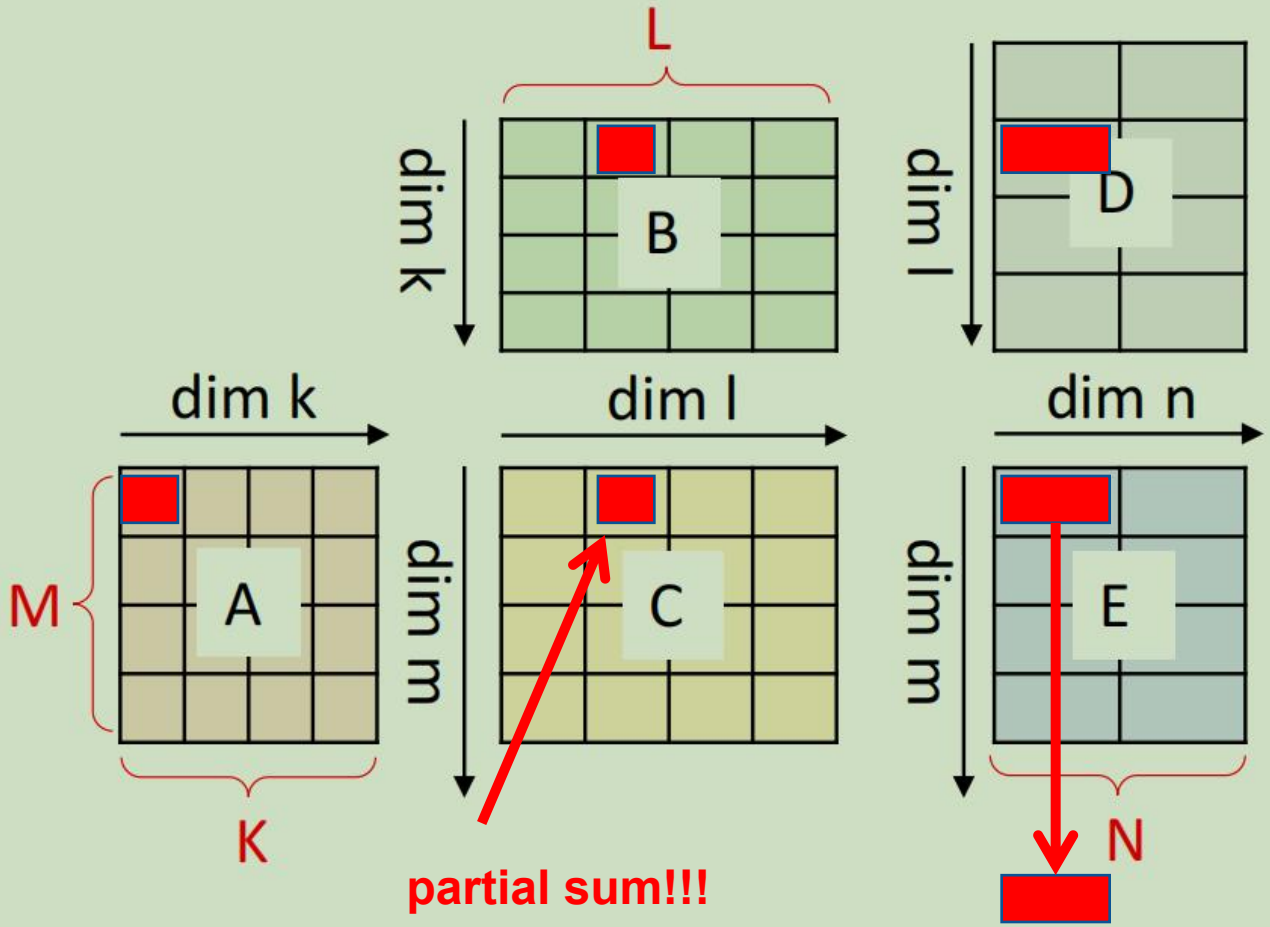
根据该算法，C的
partial sum不得不和
D矩阵进行计算

因为是partial sum还不完整，
也必须存到smem留着

No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(k){
        for(n){
            for(l)
        }
    }
}
```

Fig. 2. Different block execution orders result in
GEMM chain as an example).

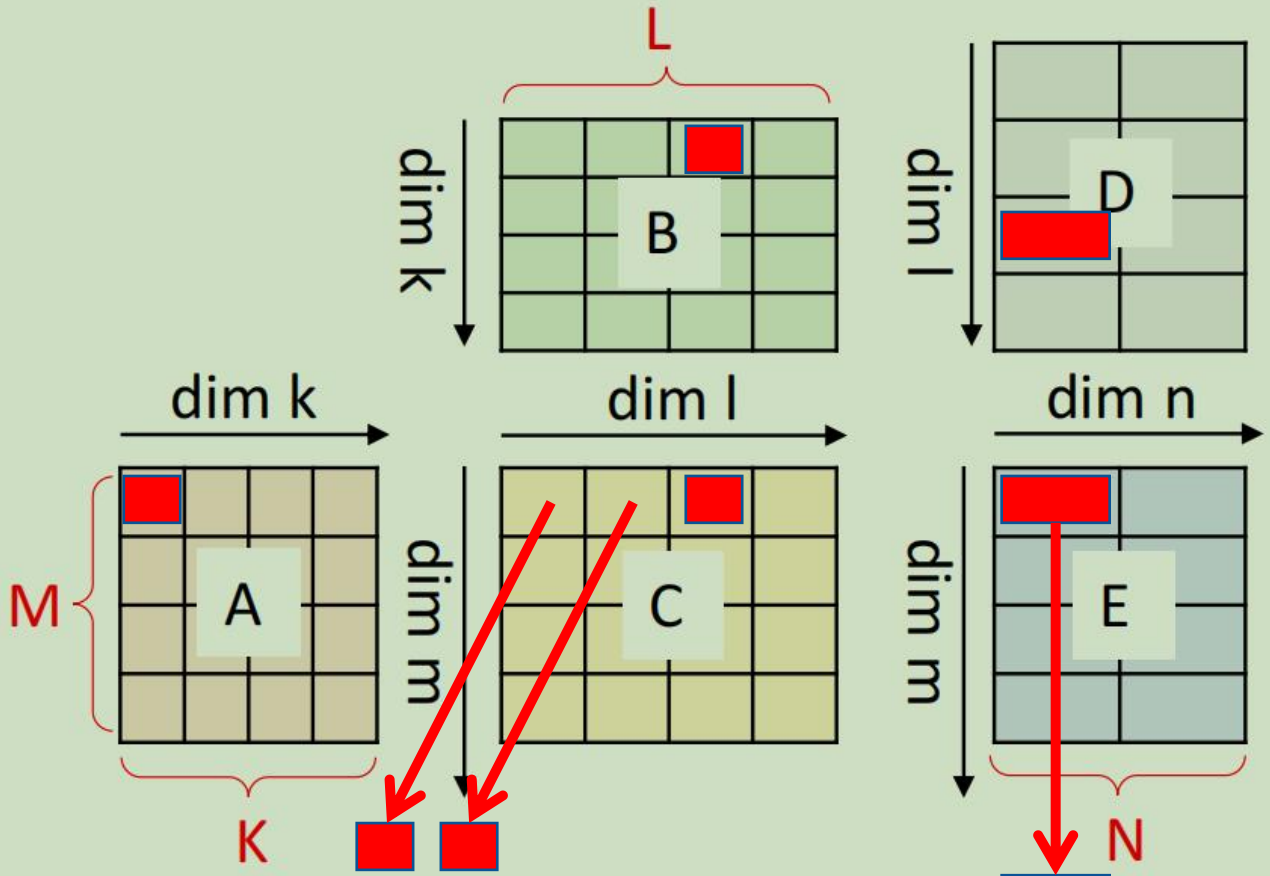


No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(k){
        for(n){
            for(l)
        }
    }
}
```

因为是partial sum还不完整，
也必须存到smem留着

Fig. 2. Different block execution orders result in
GEMM chain as an example).



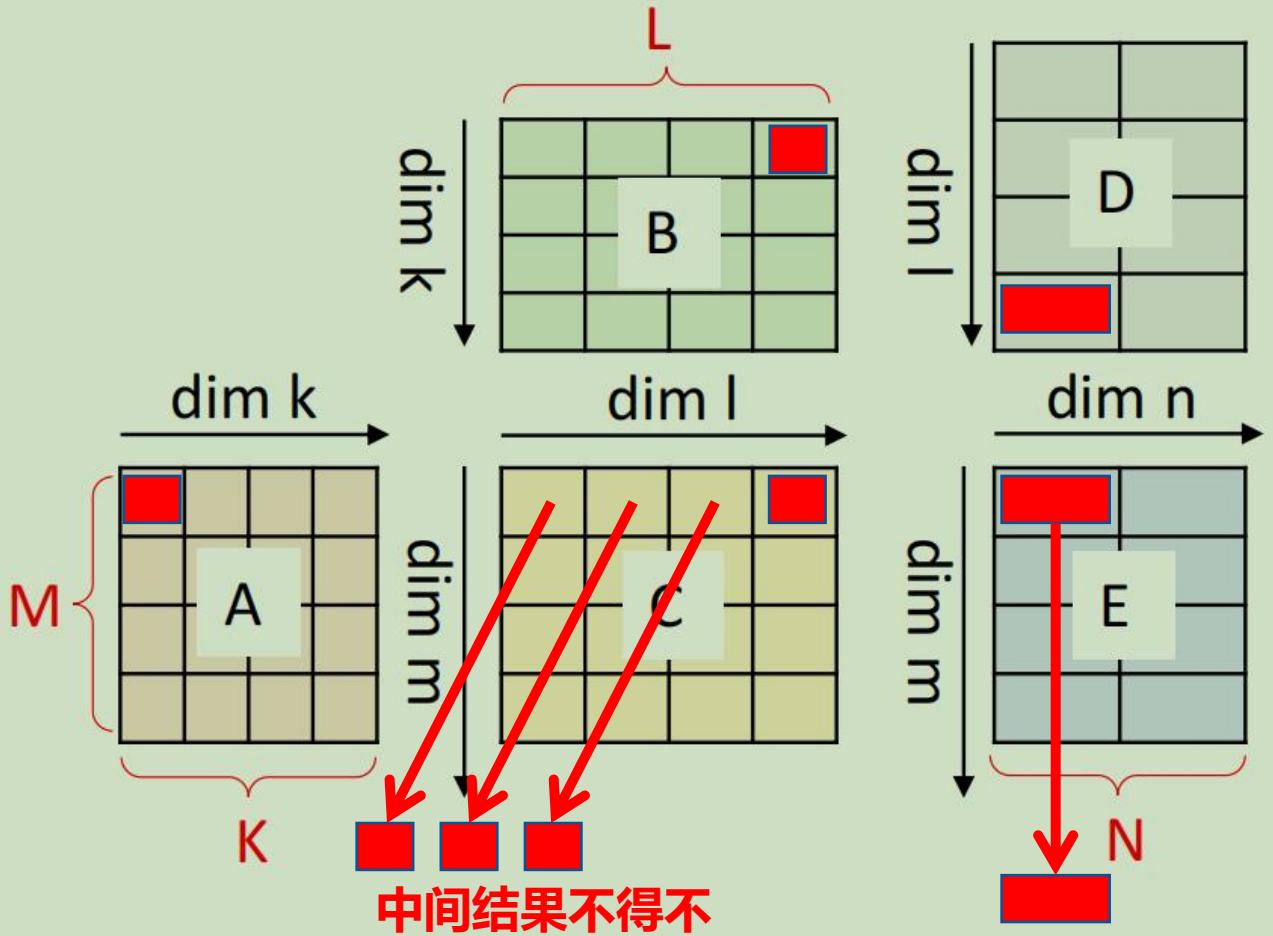
No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(k){
        for(n){
            for(l)
        }
    }
}
```

中间结果不得不全都存在smem

因为是partial sum还不完整，
也必须存到smem留着

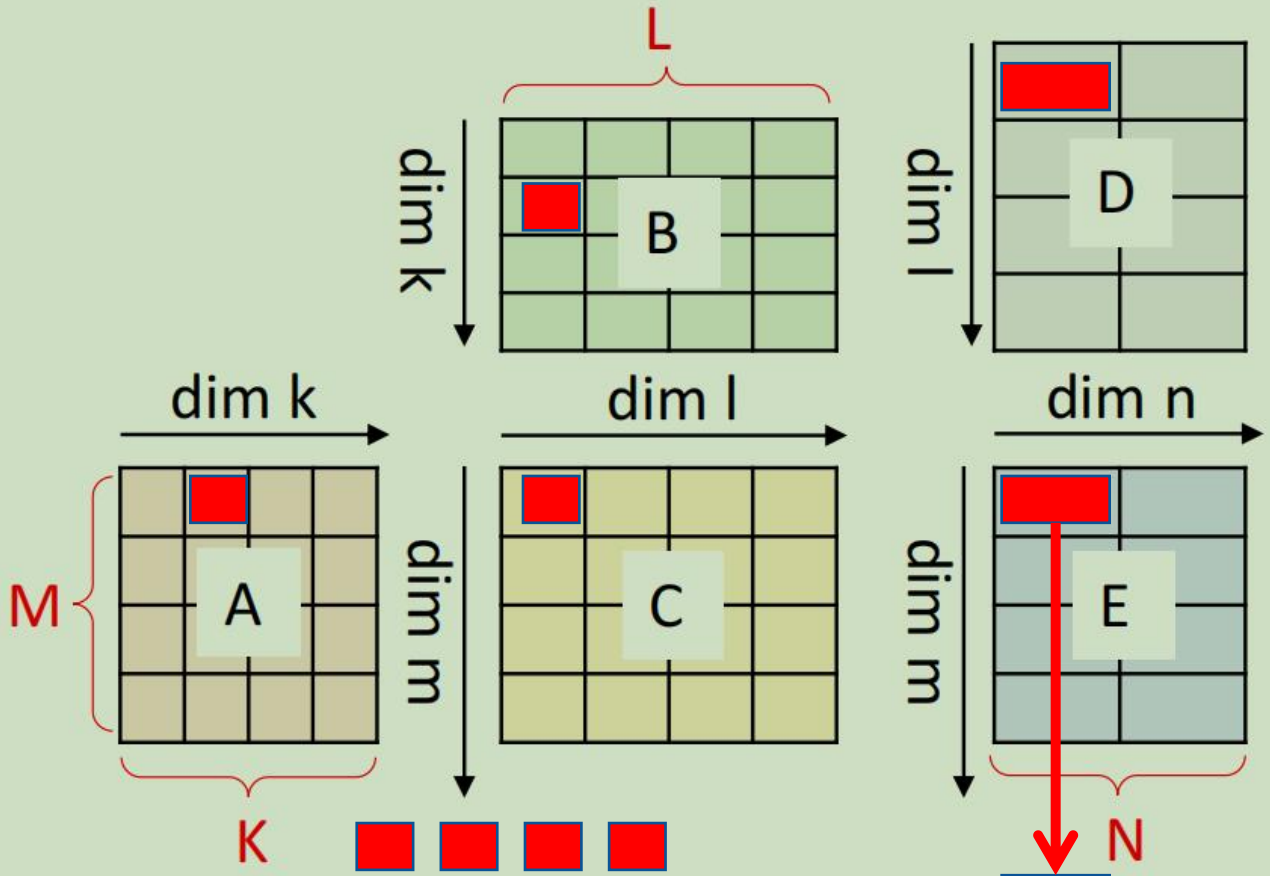
Fig. 2. Different block execution orders result in
GEMM chain as an example).



No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(k){
        for(n){
            for(l)
        }
    }
}
```

Fig. 2. Different block execution orders result in
GEMM chain as an example).



现在再回到起点，计算K=2的C，再和之前算出的E的partial sum加上去

因为是partial sum还不完整，也必须存到smem留着

No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(k){
        for(n){
            for(l)
        }
    }
}
```

Fig. 2. Different block execution orders result in
GEMM chain as an example).

三、algorithm

目标：global到smem的data traffic最小

约束：smem的空间有限

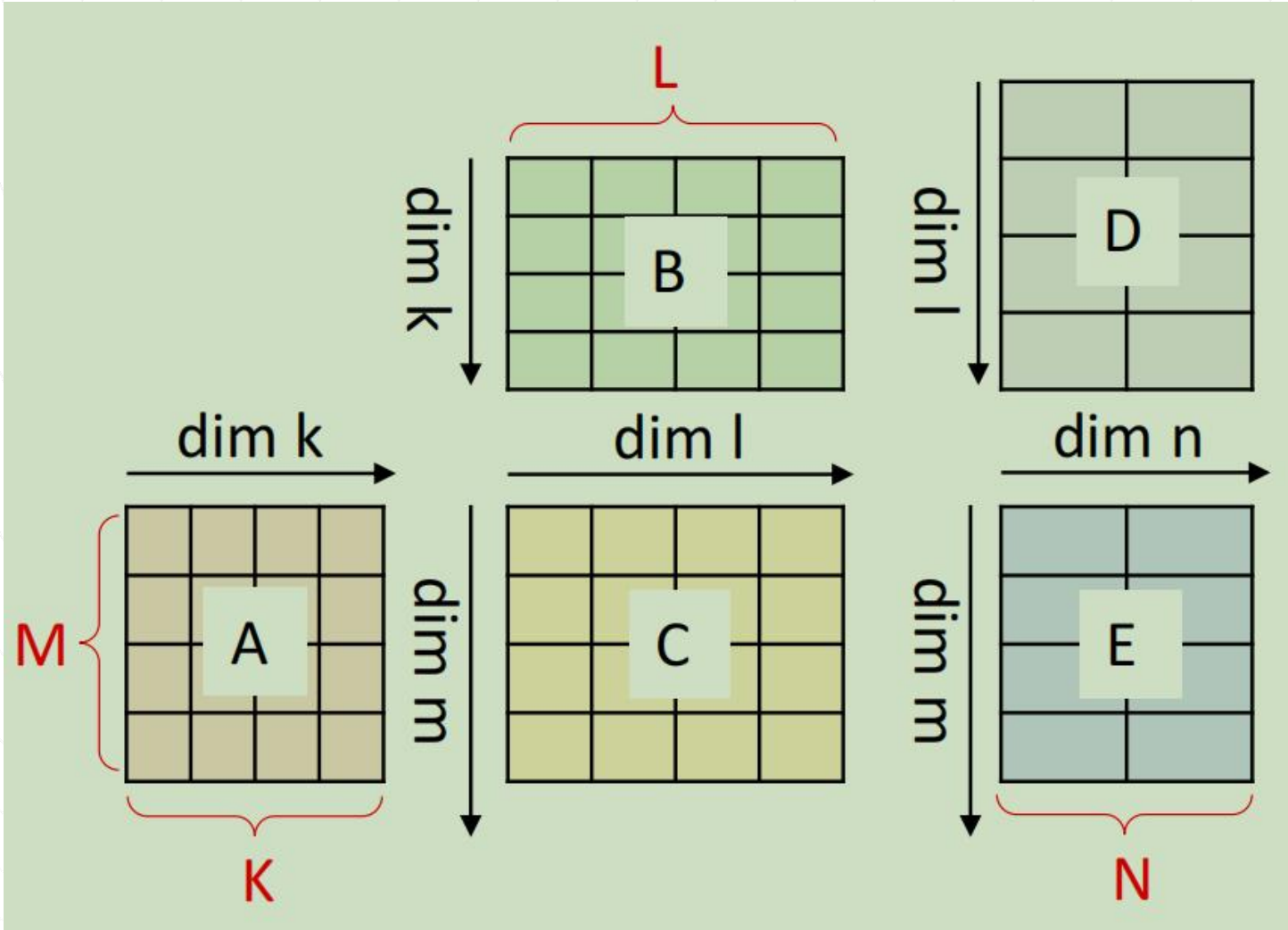
首先，一些循环不会引起任何数据移动，因为它们的迭代变量和它们内部循环的迭代变量都没有在张量访问索引中使用。

其次，一旦一个循环引起数据移动，周围所有的外循环都会引起数据移动。

第三，只在生产者运算符中出现（私有）的循环不会在消费者运算符中引起数据移动。

我们使用图 2 中的 GEMM 链示例 来解释观察结果：
1.在 **mknl** 顺序下，循环 n 和 l 不会引起矩阵 A 的数据移动，因为它们的循环变量不用于访问矩阵 A（观察值 1）。

（变换n l 对A矩阵不会造成新的A块被访问）



目标：global到smem的data traffic最小

约束：smem的空间有限

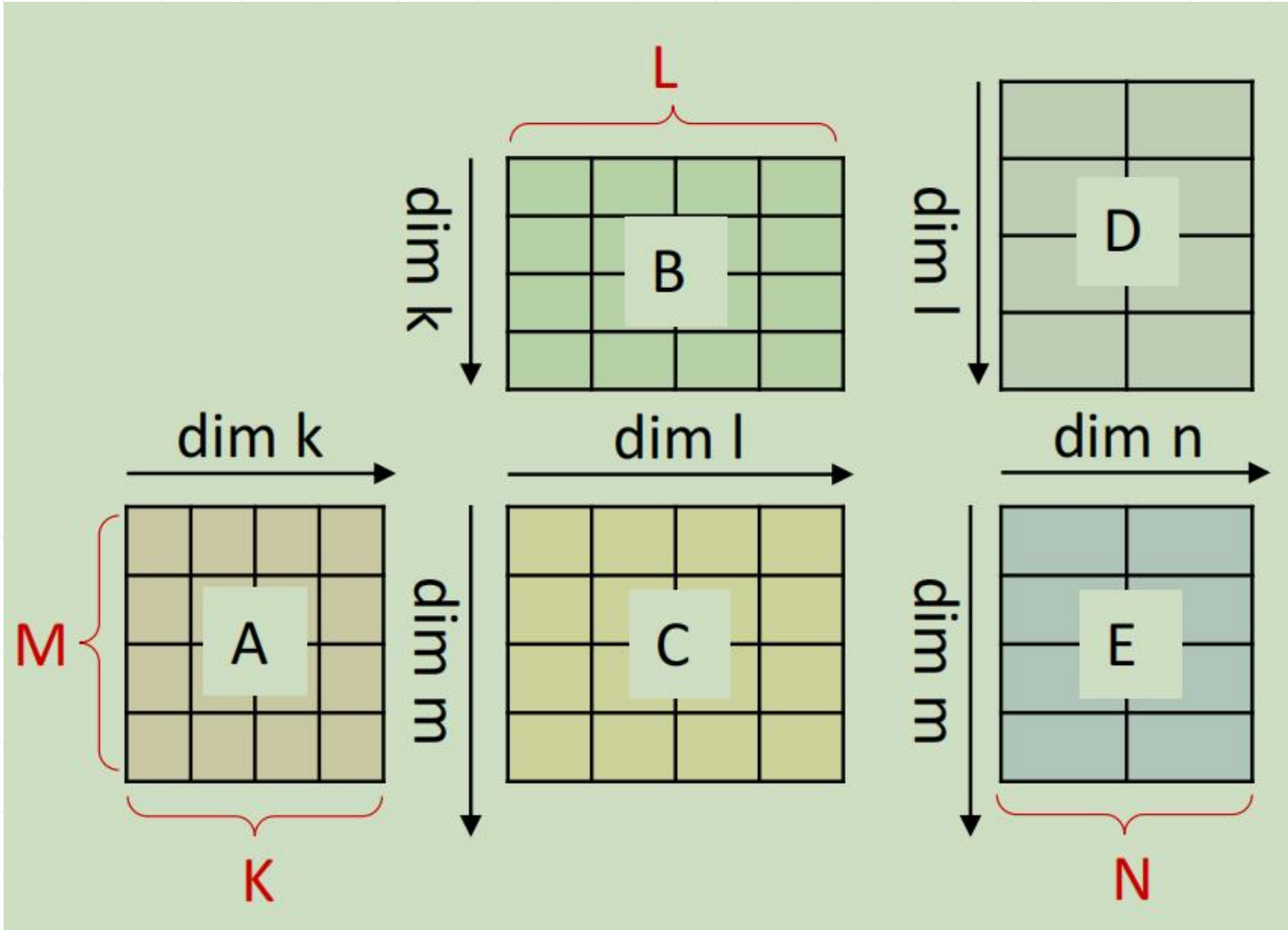
首先，一些循环不会引起任何数据移动，因为它们的迭代变量和它们内部循环的迭代变量都没有在张量访问索引中使用。

其次，一旦一个循环引起数据移动，周围所有的外循环都会引起数据移动。

第三，只在生产者运算符中出现（私有）的循环不会在消费者运算符中引起数据移动。

2。**mnlk** 顺序下，循环 n 和 l 将导致矩阵 A 的数据移动，因为内部循环 k 已经导致了数据移动（观察值 2）。

（先访问K的所有值，然后去换下一个L的值，这一步就会再次访问所有的A的数据）



目标：global到smem的data traffic最小

约束：smem的空间有限

首先，一些循环不会引起任何数据移动，因为它们的迭代变量和它们内部循环的迭代变量都没有在张量访问索引中使用。

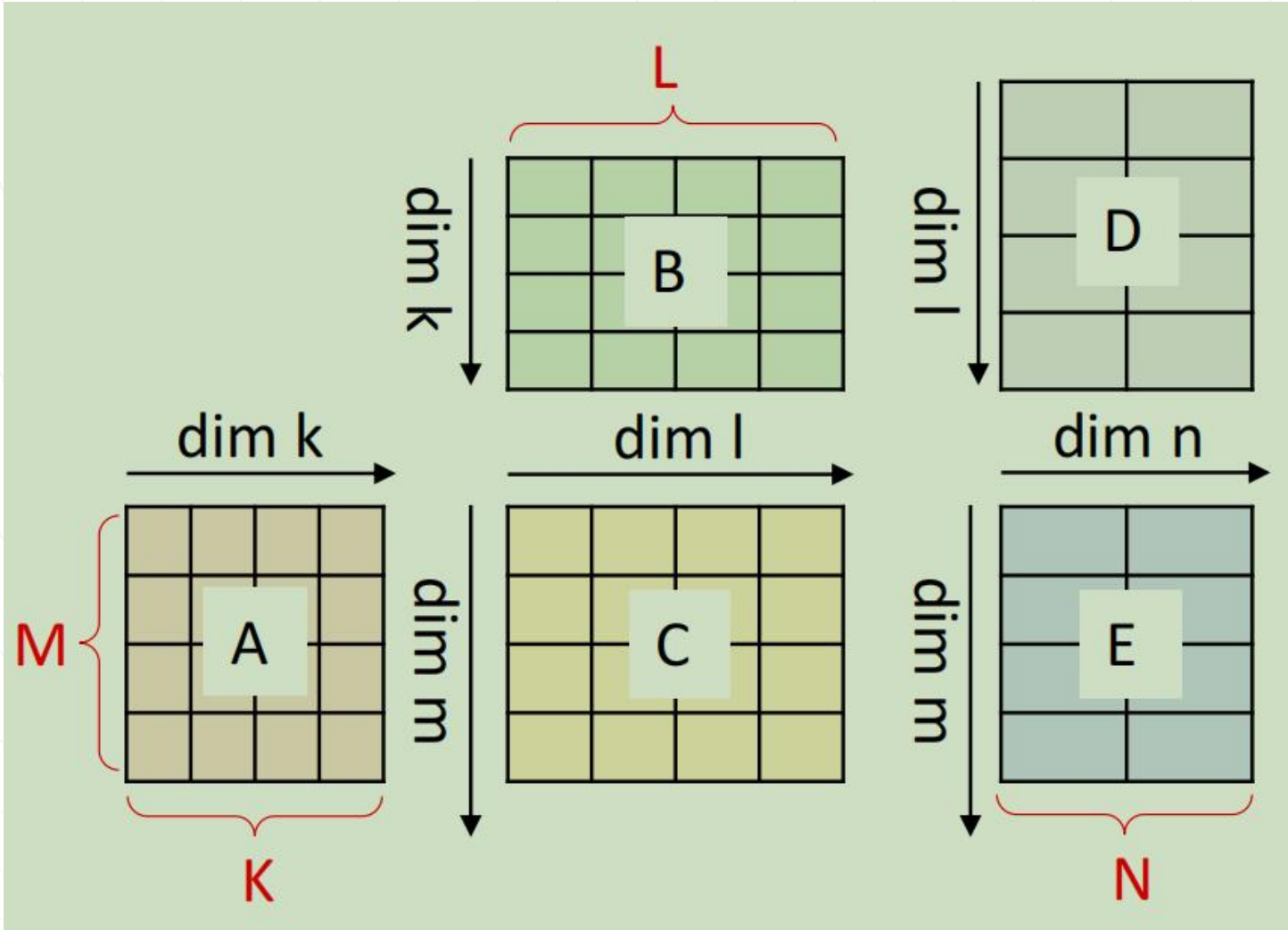
其次，一旦一个循环引起数据移动，周围所有的外循环都会引起数据移动。

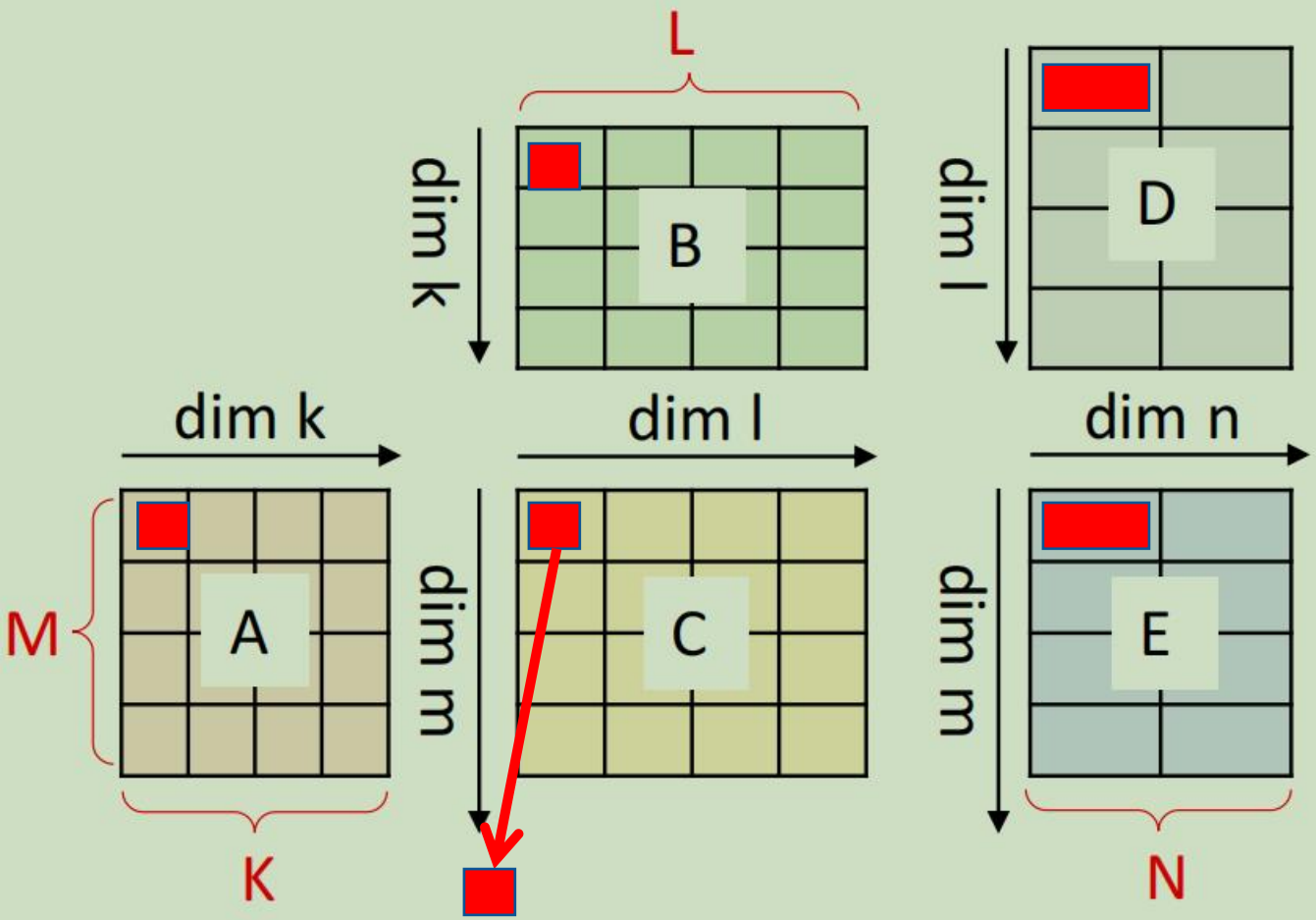
第三，只在生产者运算符中出现（私有）的循环不会在消费者运算符中引起数据移动。

3. 在任何块顺序下，循环 k 都不会导致数据移动到矩阵 D 和 E，因为 k 是第一个 GEMM 的约简循环，对第二个 GEMM 没有影响（观察值 3）。

（因为k是私有变量）

我们使用观察值 1 和 2 来计算一个操作符内块之间的数据移动，并使用观察值 3 来检测操作符之间的数据重用。

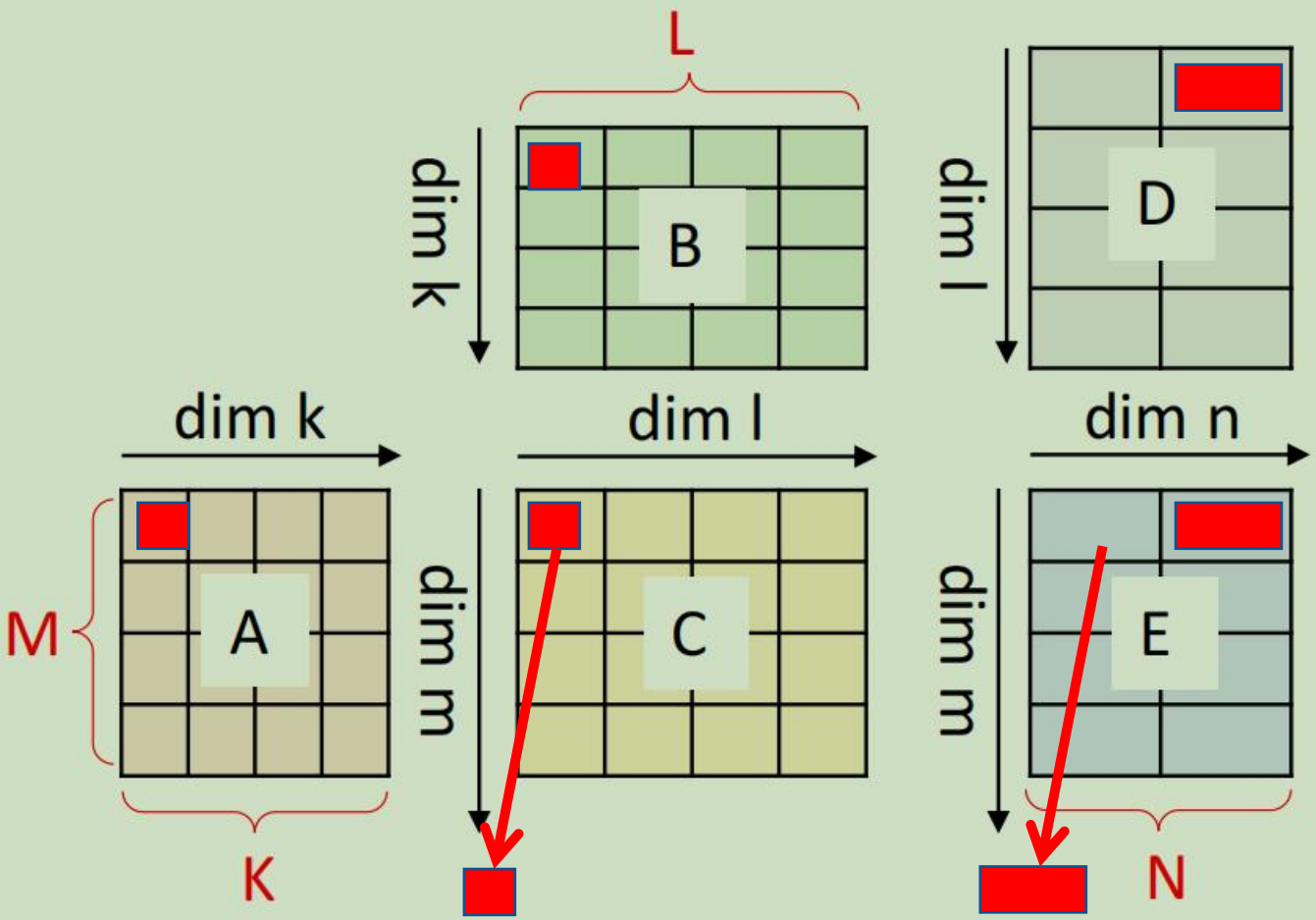




C是中间结果，当然要存在smem里

No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(l){
        for(k){
            for(n)
        }
    }
}
```

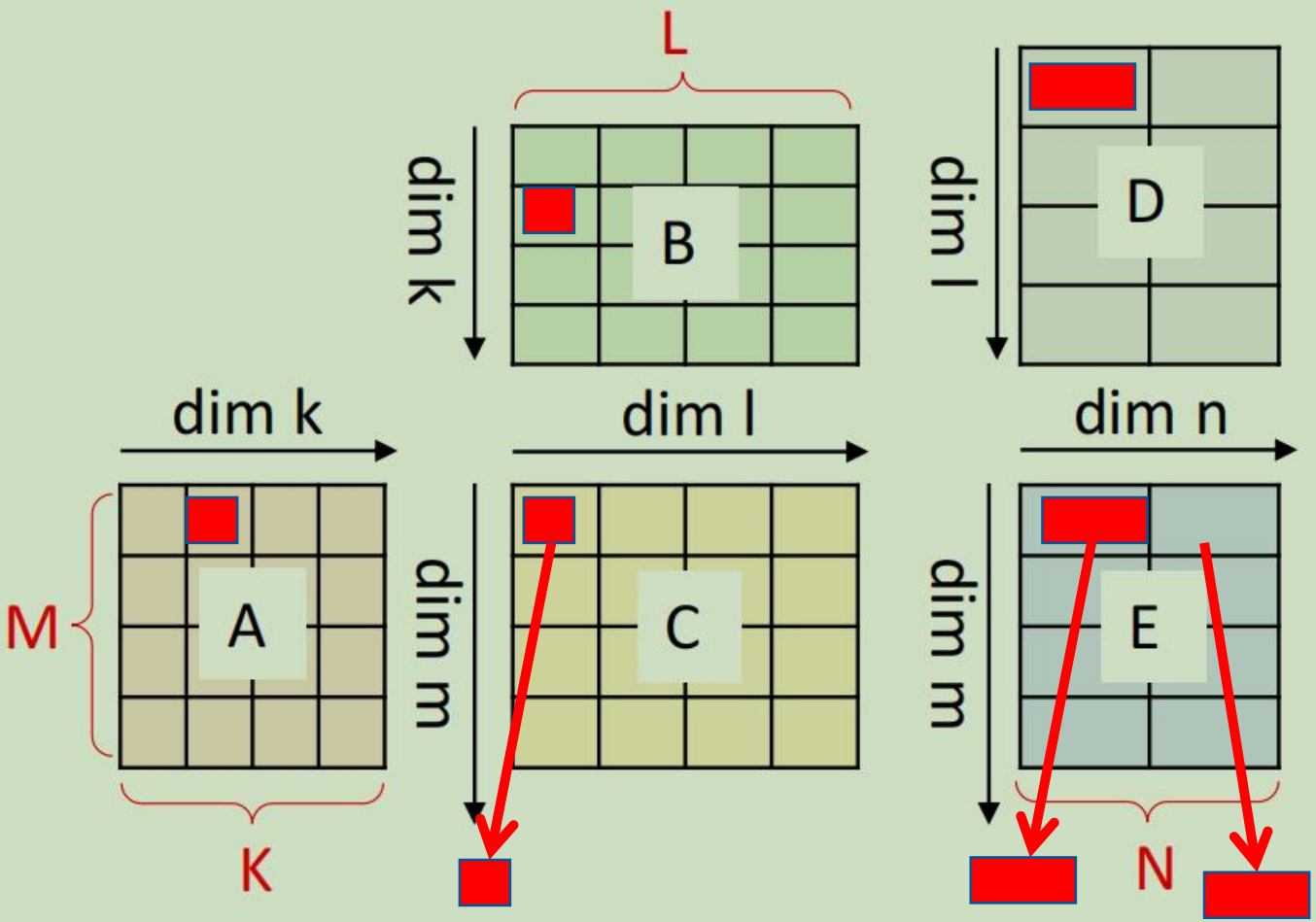


C是中间结果，当然
要存在smem里

No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

因为C是中间结果，所以这里
也是中间结果，要存到smem

```
for(m){
    for(l){
        for(k){
            for(n)
        }
    }
}
```

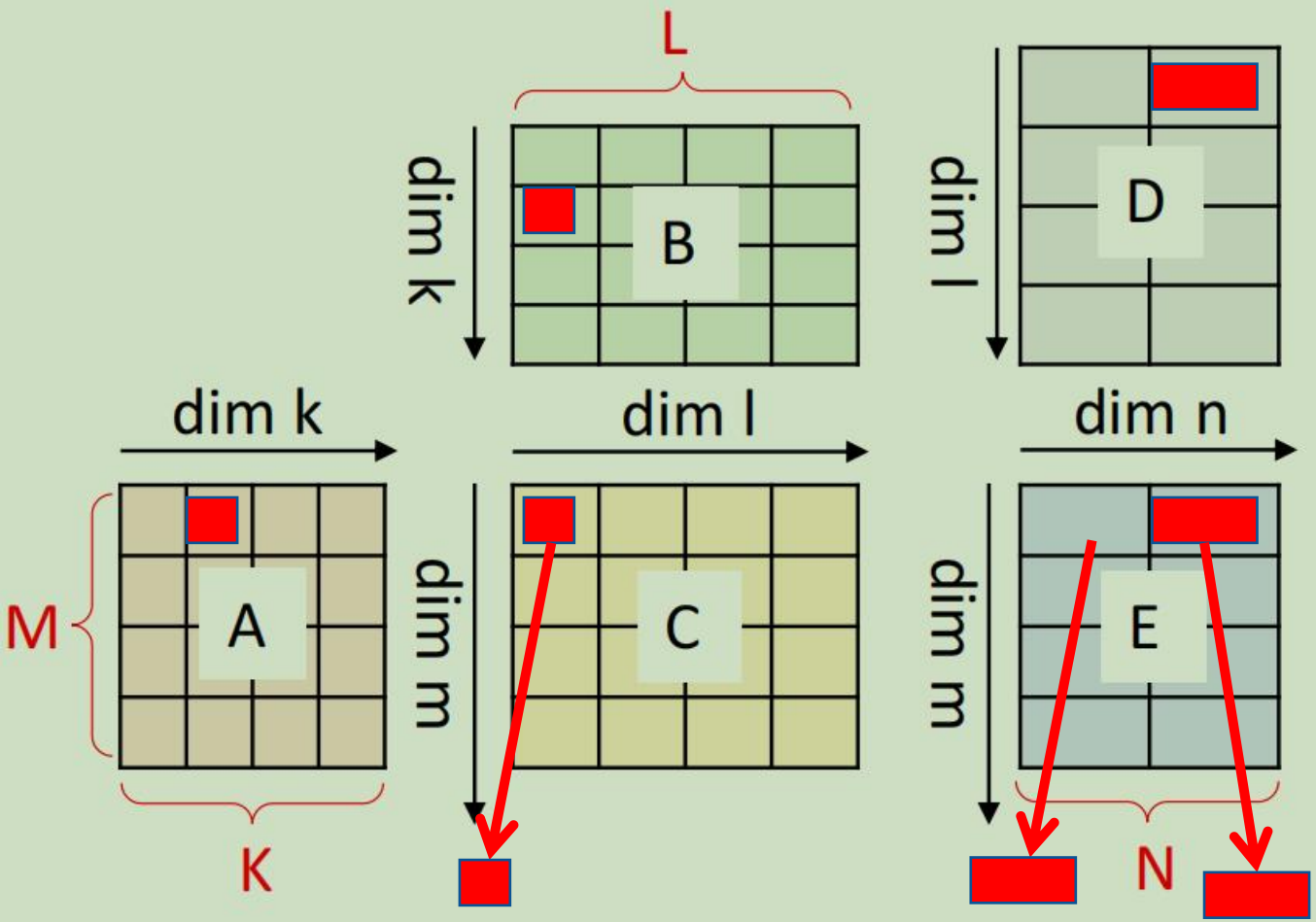


No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(l){
        for(k){
            for(n)
        }
    }
}
```

C是中间结果，当然要存在smem里

因为C是中间结果，所以这里也是中间结果，要存到smem

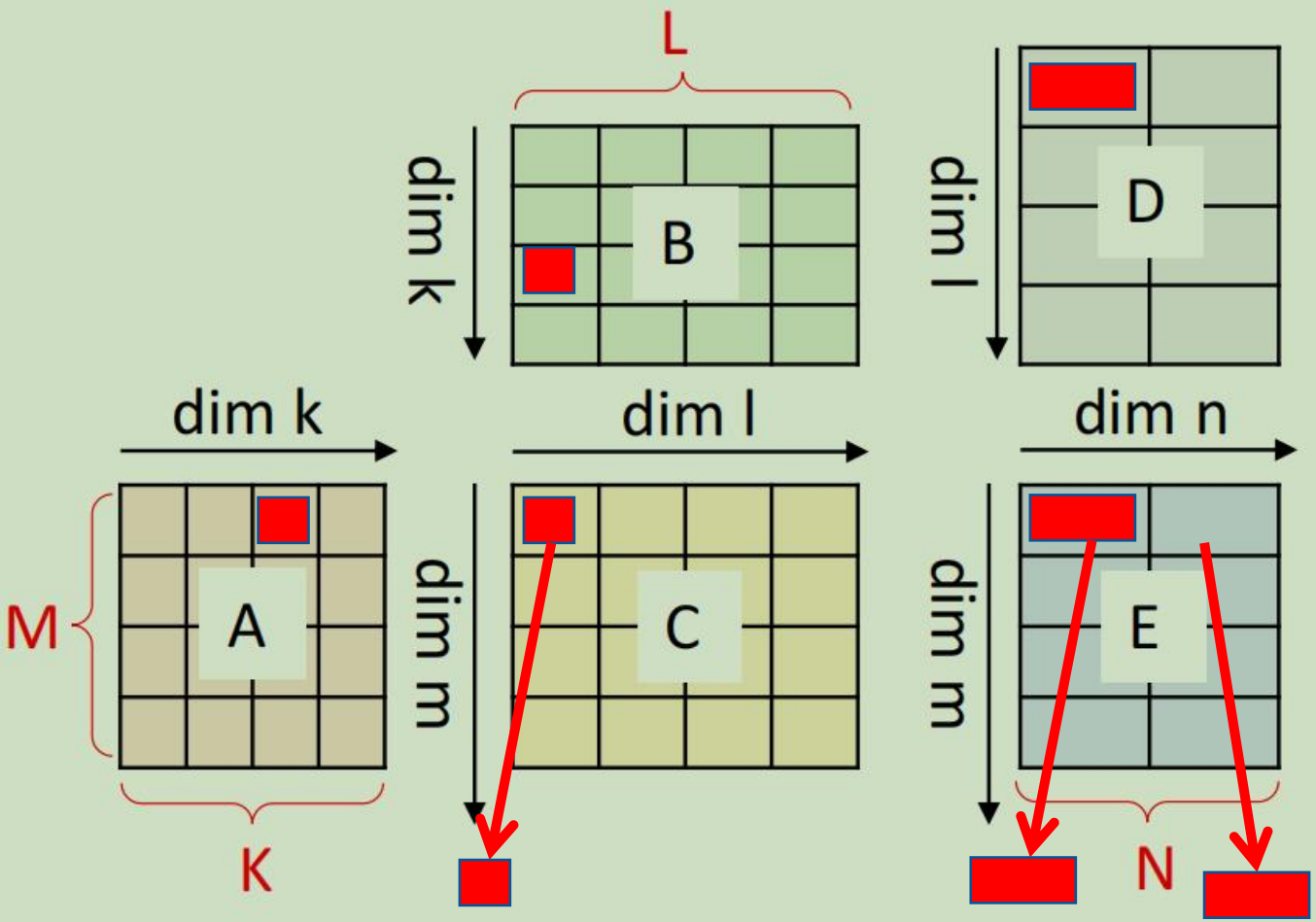


No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(l){
        for(k){
            for(n)
        }
    }
}
```

C是中间结果，当然要存在smem里

因为C是中间结果，所以这里也是中间结果，要存到smem

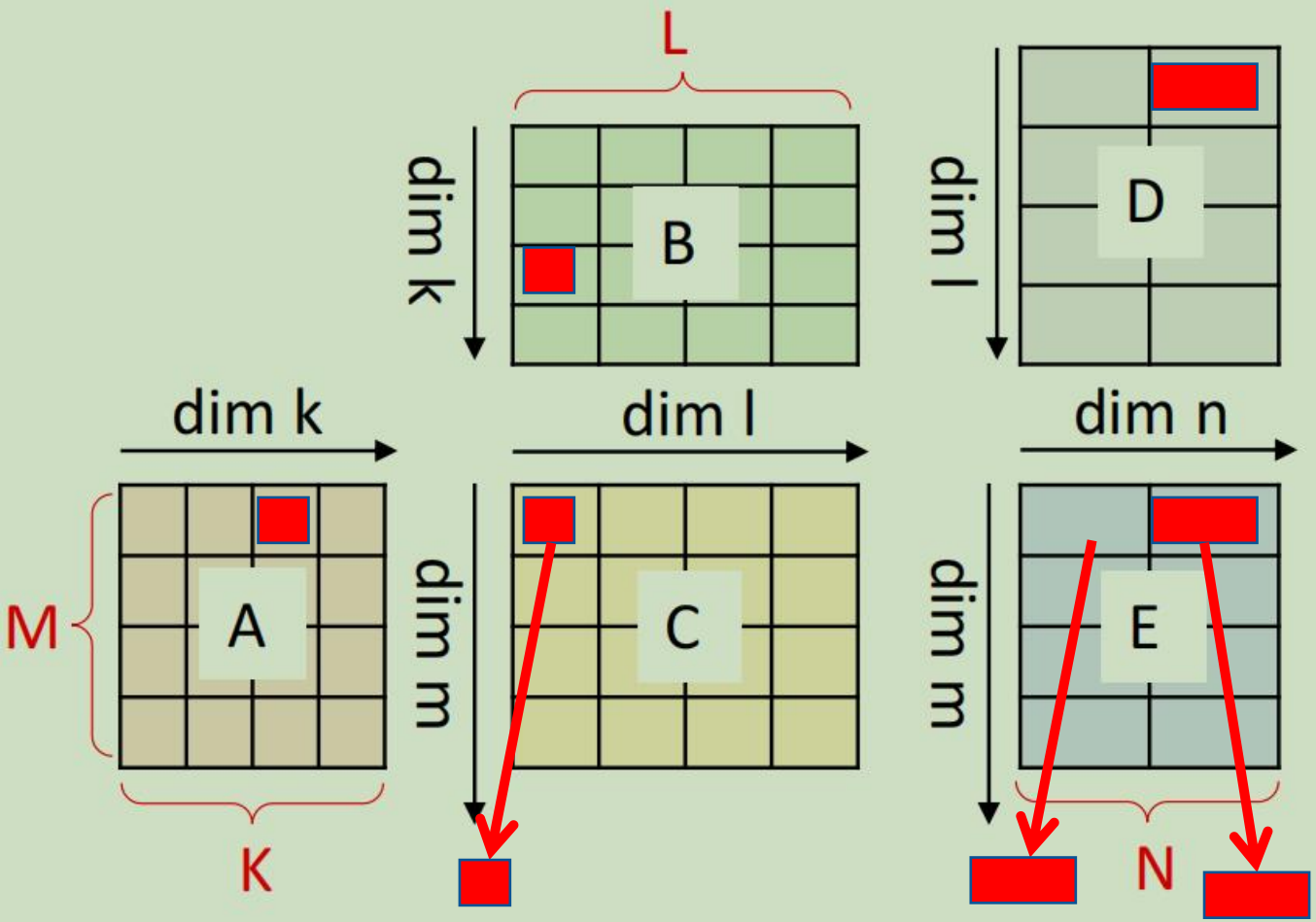


No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(l){
        for(k){
            for(n)
        }
    }
}
```

C是中间结果，当然要存在smem里

因为C是中间结果，所以这里也是中间结果，要存到smem



No.	order
1	mnkl
2	mnlk
3	mknl
4	mkln
5	mlnk
6	mlkn
⋮	⋮
24	lknm

```
for(m){
    for(l){
        for(k){
            for(n)
        }
    }
}
```

C是中间结果，当然要存在smem里

因为C是中间结果，所以这里也是中间结果，要存到smem

目标：global到smem的data traffic最小

约束：smem的空间有限

可见，这里的mlkn策略虽然也算不上多聪明，不过对于smem的占用已经比之前所述的mknl要少很多了。

我们逐步过一下这个算法。

算法1计算给定排列选择下的数据移动量。具体来说，对于目标算子链Ops，在给定排列(lp1, lp2, ..., lpI)的情况下，算法根据拓扑顺序遍历算子链（从生产者到消费者，见第2行）。

只考虑整个算子链的输入/输出张量（由函数IOTensors返回，见第7行）

Algorithm 1: Data Movement Volume Calculation and Memory Usage Algorithm for Operator Chains

input : Operator chain Ops
input : Permutation $Perm = (l_{p_1}, l_{p_2}, ..., l_{p_I})$
input : Decomposition parameters $\vec{S} = (s_1, s_2, ..., s_I)$
output : data movement volume DV
output : memory usage MU

```
1  DV = 0; MU = 0;
2  for op ∈ Ops do
3      total_DF = 0;
4      for tensor T ∈ op.allTensors() do
5          DF = getFootprint(T, S⃗);
6          total_DF += DF;
7          if T ∈ Ops.IOTensors() then
8              DM = DF;
9              keep_reuse = true;
10             for loop l_{p_i} ∈ reversed(Perm) do
11                 if l_{p_i} ∈ op.allLoops() then
12                     if l_{p_i} accesses tensor T then
13                         keep_reuse = false;
14                     if not keep_reuse then
15                         DM *= ⌈L_{p_i} / s_{p_i}⌉
16             DV += DM;
17     for loop l_{p_i} ∈ Perm do
18         if l_{p_i} is private to op then
19             Perm.erase(l_{p_i});
20     MU = max(MU, total_DF);
21 return DV, MU;
```

目标：global到smem的data traffic最小

约束：smem的空间有限

我们使用getFootprint来计算数据移动量。函数根据分解参数s计算每个张量（第5行）的数据块足迹（DF）。

为了计算数据移动量，我们需要计算数据块在执行过程中被替换的次数。注意，只有访问张量的循环才会导致数据块的替换（观察1）。

我们使用一个标志keep_reuse来检查当前循环lpi是否会导致替换（第12-14行）。

如果是，我们通过乘以循环行程计数来增加当前张量T的数据移动量（第15行）。

该标志对于所有其他外部循环保持为真，并将它们的行程计数乘以数据移动量（观察2）。

在我们转移到消费者操作符之前，我们需要排除生产者操作符的私有循环的影响（见第17-19行），因为这样的私有循环不会迭代消费者操作符的张量（观察3）。

算法还返回最大内存使用量MU，作为问题约束。

Algorithm 1: Data Movement Volume Calculation and Memory Usage Algorithm for Operator Chains

```
   input  : Operator chain Ops
   input  : Permutation Perm = (l_{p1}, l_{p2}, ..., l_{pI})
   input  : Decomposition parameters S = (s_1, s_2, ..., s_I)
   output : data movement volume DV
   output : memory usage MU
1  DV = 0; MU = 0;
2  for op ∈ Ops do
3      total_DF = 0;
4      for tensor T ∈ op.allTensors() do
5          DF = getFootprint(T, S);
6          total_DF += DF;
7          if T ∈ Ops.IOTensors() then
8              DM = DF;
9              keep_reuse = true;
10             for loop l_{pi} ∈ reversed(Perm) do
11                 if l_{pi} ∈ op.allLoops() then
12                     if l_{pi} accesses tensor T then
13                         keep_reuse = false;
14                     if not keep_reuse then
15                         DM *= ⌈L_{pi} / s_{pi}⌉
16             DV += DM;
17     for loop l_{pi} ∈ Perm do
18         if l_{pi} is private to op then
19             Perm.erase(l_{pi});
20     MU = max(MU, total_DF);
21 return DV, MU;
```

这里的单个数据移动量（DM）仍然是按照tile_m和tile_k来计算。然而，由于存在循环pi，例如对于矩阵A，数据移动量会逐步累加，直到覆盖整个M维度。

目标：global到smem的data traffic最小

约束：smem的空间有限

顺序是reverse（perm）这很重要，因为访问顺序是从最后开始看的

循环是按照每个block来看的。我们依此得到DF，为TMTK这样。

对A矩阵的一个块，我们需要访问完整的M*K，而对于（取上限）L/T_L个块，我们对A矩阵的总数据访问量就是M*K*（取上限）L/T_L。

这里是单独从算子角度，一个一个来看各自对所有矩阵造成的数据访问量的影响的。比如先看GEMM1，对ABCDE的影响。然后再看GEMM0。

TABLE III

DATA MOVEMENT VOLUME AND MEMORY USAGE FOR GEMM CHAIN UNDER THE ORDER OF $mlkn$.

	A	B	C	D	E
DM	$MK\lceil\frac{L}{T_L}\rceil$	$KL\lceil\frac{M}{T_M}\rceil$	0	$NL\lceil\frac{M}{T_M}\rceil$	$MN\lceil\frac{L}{T_L}\rceil$
DF	$T_M T_K$	$T_K T_L$	$T_M T_L$	$T_L T_N$	$T_M T_N$

Algorithm 1: Data Movement Volume Calculation and Memory Usage Algorithm for Operator Chains

input : Operator chain Ops
input : Permutation $Perm = (l_{p_1}, l_{p_2}, ..., l_{p_I})$
input : Decomposition parameters $\vec{S} = (s_1, s_2, ..., s_I)$
output : data movement volume DV
output : memory usage MU

```
DV = 0; MU = 0;
for op ∈ Ops do
    total_DF = 0;
    for tensor T ∈ op.allTensors() do
        DF = getFootprint(T, S⃗);
        total_DF += DF;
        if T ∈ Ops.IOTensors() then
            DM = DF;
            keep_reuse = true;
            for loop l_pi ∈ reversed(Perm) do
                if l_pi ∈ op.allLoops() then
                    if l_pi accesses tensor T then
                        keep_reuse = false;
                    if not keep_reuse then
                        DM *= ⌈L_pi / s_pi⌉
            DV += DM;
    for loop l_pi ∈ Perm do
        if l_pi is private to op then
            Perm.erase(l_pi);
    MU = max(MU, total_DF);
return DV, MU;
```

这里的单个数据移动量（DM）仍然是按照tile_m和tile_k来计算。然而，由于存在循环pi，例如对于矩阵A，数据移动量会逐步累加，直到覆盖整个M维度。

$$min_{\vec{S}} \ \text{DV}, \quad \text{s.t. MU} \leq MemoryCapacity \tag{1}$$

目标：global到smem的data traffic最小

约束：smem的空间有限

$$\text{DV}_{\text{GEMM Chain}} = DM_A + DM_B + DM_C + DM_D + DM_E$$
$$= MK\lceil\frac{L}{T_L}\rceil + KL\lceil\frac{M}{T_M}\rceil + NL\lceil\frac{M}{T_M}\rceil + MN\lceil\frac{L}{T_L}\rceil$$

The peak memory usage MU of all the tensors is

$$\text{MU} = max\{\text{GEMM1}_{MU}, \text{GEMM2}_{MU}\}$$
$$\text{GEMM1}_{MU} = DF_A + DF_B + DF_C = T_M T_K + T_K T_L + T_M T_L$$
$$\text{GEMM2}_{MU} = DF_C + DF_D + DF_E = T_M T_L + T_L T_N + T_M T_N$$

顺序是reverse（perm）这很重要，因为访问顺序是从最后开始看的

循环是按照每个block来看的。我们依此得到DF，为TMTK这样。

对A矩阵的一个块，我们需要访问完整的M*K，而对于（取上限）L/T_L个块，我们对A矩阵的总数据访问量就是M*K*（取上限）L/T_L。

这里是单独从算子角度，一个一个来看各自对所有矩阵造成的数据访问量的影响的。比如先看GEMM1，对ABCDE的影响。然后再看GEMM0。

TABLE III
DATA MOVEMENT VOLUME AND MEMORY USAGE FOR GEMM CHAIN UNDER THE ORDER OF $mlkn$.

	A	B	C	D	E
DM	$MK\lceil\frac{L}{T_L}\rceil$	$KL\lceil\frac{M}{T_M}\rceil$	0	$NL\lceil\frac{M}{T_M}\rceil$	$MN\lceil\frac{L}{T_L}\rceil$
DF	$T_M T_K$	$T_K T_L$	$T_M T_L$	$T_L T_N$	$T_M T_N$

$$min_{\vec{S}} \ \text{DV}, \quad \text{s.t. MU} \leq MemoryCapacity \tag{1}$$

TABLE III
DATA MOVEMENT VOLUME AND MEMORY USAGE FOR GEMM CHAIN UNDER THE ORDER OF $mlkn$.

	A	B	C	D	E
DM	$MK\lceil\frac{L}{T_L}\rceil$	$KL\lceil\frac{M}{T_M}\rceil$	0	$NL\lceil\frac{M}{T_M}\rceil$	$MN\lceil\frac{L}{T_L}\rceil$
DF	T_MT_K	T_KT_L	T_MT_L	T_LT_N	T_MT_N

$$\begin{aligned}\text{DV}_{\text{GEMM Chain}} &= DM_A + DM_B + DM_C + DM_D + DM_E \\ &= MK\lceil\frac{L}{T_L}\rceil + KL\lceil\frac{M}{T_M}\rceil + NL\lceil\frac{M}{T_M}\rceil + MN\lceil\frac{L}{T_L}\rceil\end{aligned}$$

The peak memory usage MU of all the tensors is

$$\begin{aligned}\text{MU} &= max\{\text{GEMM1}_{MU}, \text{GEMM2}_{MU}\} \\ \text{GEMM1}_{MU} &= DF_A + DF_B + DF_C = T_MT_K + T_KT_L + T_MT_L \\ \text{GEMM2}_{MU} &= DF_C + DF_D + DF_E = T_MT_L + T_LT_N + T_MT_N\end{aligned}$$

$$DV^* = \frac{2ML(K+N)}{T_M^*}, \quad T_M^* = T_L^* = -\alpha + \sqrt{\alpha^2 + MC}, T_N^* = \alpha$$

2.1 用拉格朗日乘子法做一个示例推导

为演示，先做一个**最简化**：

- 假设 T_N, T_K 已经定为 α，不再是自由变量；
- 假设约束就是 $T_M T_L = \text{MC}$（把它当成等式），忽略"max{}"那部分复杂性。

此时目标函数（忽略系数）为：

$$DV(T_M, T_L) \ = \ M\,L\,(K+N)\left(\tfrac{1}{T_M} + \tfrac{1}{T_L}\right),$$

约束为

$$T_M\,T_L \ = \ \text{MC}.$$

我们可以写出简化的拉格朗日函数（不带 max）：

$$\mathcal{L}(T_M, T_L, \lambda) \ = \ M\,L\,(K+N)\left(\tfrac{1}{T_M} + \tfrac{1}{T_L}\right) \ + \ \lambda\left(T_M\,T_L - \text{MC}\right).$$

对 T_M、T_L 分别求偏导并令其为 0。

1. **对 T_M 的偏导**

$$\frac{\partial \mathcal{L}}{\partial T_M} = M\,L\,(K+N)\left(-\tfrac{1}{T_M^2}\right) \ + \ \lambda T_L = 0.$$

即

$$-\frac{M\,L\,(K+N)}{T_M^2} \ + \ \lambda T_L = 0 \implies \lambda = \frac{M\,L\,(K+N)}{T_M^2\,T_L}.$$

2. **对 T_L 的偏导**

$$\frac{\partial \mathcal{L}}{\partial T_L} = M\,L\,(K+N)\left(-\tfrac{1}{T_L^2}\right) \ + \ \lambda T_M = 0.$$

即

$$-\frac{M\,L\,(K+N)}{T_L^2} \ + \ \lambda T_M = 0 \implies \lambda = \frac{M\,L\,(K+N)}{T_L^2\,T_M}.$$

$$min_{\vec{S}} \ \text{DV}, \quad \text{s.t. MU} \leq MemoryCapacity \tag{1}$$

TABLE III
DATA MOVEMENT VOLUME AND MEMORY USAGE FOR GEMM CHAIN UNDER THE ORDER OF $mlkn$.

	A	B	C	D	E
DM	$MK\lceil\frac{L}{T_L}\rceil$	$KL\lceil\frac{M}{T_M}\rceil$	0	$NL\lceil\frac{M}{T_M}\rceil$	$MN\lceil\frac{L}{T_L}\rceil$
DF	T_MT_K	T_KT_L	T_MT_L	T_LT_N	T_MT_N

$$\begin{aligned}\text{DV}_{\text{GEMM Chain}} &= DM_A + DM_B + DM_C + DM_D + DM_E \\ &= MK\lceil\frac{L}{T_L}\rceil + KL\lceil\frac{M}{T_M}\rceil + NL\lceil\frac{M}{T_M}\rceil + MN\lceil\frac{L}{T_L}\rceil\end{aligned}$$

The peak memory usage MU of all the tensors is

$$\begin{aligned}\text{MU} &= max\{\text{GEMM1}_{MU}, \text{GEMM2}_{MU}\} \\ \text{GEMM1}_{MU} &= DF_A + DF_B + DF_C = T_MT_K + T_KT_L + T_MT_L \\ \text{GEMM2}_{MU} &= DF_C + DF_D + DF_E = T_MT_L + T_LT_N + T_MT_N\end{aligned}$$

$$DV^* = \frac{2ML(K+N)}{T_M^*}, \quad T_M^* = T_L^* = -\alpha + \sqrt{\alpha^2 + MC}, T_N^* = \alpha$$

把这两式比较起来，得到

$$\frac{M\,L\,(K+N)}{T_M^2\,T_L} = \frac{M\,L\,(K+N)}{T_L^2\,T_M} \implies T_L^2\,T_M = T_M^2\,T_L \implies T_L = T_M.$$

最后再结合**约束** $T_MT_L = \text{MC}$，可得

$$T_M^* = T_L^* = \sqrt{\text{MC}}.$$

如果还要考虑到 α，文中有更一般的表达 $-\alpha + \sqrt{\alpha^2 + \text{MC}}$，那是因为实际的 MU 里还包含 α 相关项，会使得"可分配给 T_MT_L 的那部分内存"为 \text{MC} - \text{(一些跟\(\alpha有关的占用)}})。推导细节大同小异，这里就不赘述了。

---

3. 得到最优解后的最小 DV^*

把 T_M^*, T_L^* 代回到目标函数即可。仍用上面简化场景举例：

$$T_M^* = T_L^* = \sqrt{\text{MC}},$$

$$DV^* = M\,L\,(K+N)\left(\frac{1}{T_M^*} + \frac{1}{T_L^*}\right) = M\,L\,(K+N)\left(\frac{1}{\sqrt{\text{MC}}} + \frac{1}{\sqrt{\text{MC}}}\right) = 2\,\frac{M\,L\,(K+N)}{\sqrt{\text{MC}}}.$$

这就类似你截图里出现的那种形式：

$$DV^* = 2\,M\,L\,\frac{(K+N)}{T_M^*},$$

只不过他们用的符号是 $T_M^* \approx T_L^* \approx \sqrt{\text{MC}}$，或 $-\alpha + \sqrt{\alpha^2 + \text{MC}}$ 等。

$$min_{\vec{S}} \ \text{DV}, \quad \text{s.t. MU} \leq MemoryCapacity \tag{1}$$

TABLE III
DATA MOVEMENT VOLUME AND MEMORY USAGE FOR GEMM CHAIN UNDER THE ORDER OF $mlkn$.

	A	B	C	D	E
DM	$MK\lceil\frac{L}{T_L}\rceil$	$KL\lceil\frac{M}{T_M}\rceil$	0	$NL\lceil\frac{M}{T_M}\rceil$	$MN\lceil\frac{L}{T_L}\rceil$
DF	T_MT_K	T_KT_L	T_MT_L	T_LT_N	T_MT_N

$$\begin{aligned}\text{DV}_{\text{GEMM Chain}} &= DM_A + DM_B + DM_C + DM_D + DM_E \\ &= MK\lceil\frac{L}{T_L}\rceil + KL\lceil\frac{M}{T_M}\rceil + NL\lceil\frac{M}{T_M}\rceil + MN\lceil\frac{L}{T_L}\rceil\end{aligned}$$

The peak memory usage MU of all the tensors is

$$\begin{aligned}\text{MU} &= max\{\text{GEMM1}_{MU}, \text{GEMM2}_{MU}\} \\ \text{GEMM1}_{MU} &= DF_A + DF_B + DF_C = T_MT_K + T_KT_L + T_MT_L \\ \text{GEMM2}_{MU} &= DF_C + DF_D + DF_E = T_MT_L + T_LT_N + T_MT_N\end{aligned}$$

$$DV^* = \frac{2ML(K+N)}{T_M^*}, \quad T_M^* = T_L^* = -\alpha + \sqrt{\alpha^2 + MC}, T_N^* = \alpha$$

$$\frac{DV_{app}}{DV^*} \leq max_{X\in\{M,L\}}\{1 + \frac{T_X^*}{X} + \frac{1}{T_X}\} \leq$$

$$max_{X\in\{M,L\}}\{1 + \frac{\sqrt{MC}}{X} + \frac{1}{\min\{X, \sqrt{MC}\}}\}, \ (MC >> \alpha)$$

$$\frac{DV_{\text{app}}}{DV^*} = \frac{ML(K+N)\left(\frac{1}{T_M} + \frac{1}{T_L}\right)}{ML(K+N)\left(\frac{1}{T_M^*} + \frac{1}{T_L^*}\right)} = \frac{\frac{1}{T_M} + \frac{1}{T_L}}{\frac{1}{T_M^*} + \frac{1}{T_L^*}}.$$

$$\frac{DV_{\text{app}}}{DV^*} \approx \frac{\frac{1}{T_M} + \frac{1}{T_L}}{\frac{1}{T_M^*} + \frac{1}{T_L^*}} \leq \max\left\{\frac{1/T_M}{1/T_M^*}, \frac{1/T_L}{1/T_L^*}\right\} = \max\left\{\frac{T_M^*}{T_M}, \frac{T_L^*}{T_L}\right\}.$$

这里用到了一个常见不等式：$\frac{a+b}{c+d} \leq \max\left(\frac{a}{c}, \frac{b}{d}\right)$ 对 $a, b, c, d \geq 0$ 成立。

根据拉格朗日之前所求，Tx*可以替换为sqrt(MC)

4. 因此实际的 T_M 就是"$\min\{\lfloor T_M^* \rfloor, M\}$"，而又常常 $T_M^* \approx \sqrt{\text{MC}}$。**所以** $T_M \approx \min\{\sqrt{\text{MC}}, M\}$。

当你要做一个简单的"不等式上界"时，就可以说：

$$T_M \geq \min\{\sqrt{\text{MC}}, M\} \implies \frac{1}{T_M} \leq \frac{1}{\min\{\sqrt{\text{MC}}, M\}}.$$

现在就卡在下面这个不等式的证明上了：

$$\frac{T_m^*}{T_m} \leq 1 + \frac{T_m^*}{M} + \frac{1}{T_m}.$$

三、evaluation

我们测试了子图融合性能和全网络性能。我们使用的子图包括来自**Bert、ViT和MLP-Mixer**的批处理GEMM链，以及来自**CNN（如SqueezeNet和Yolo）的卷积链**。对于全网评估，我们使用了**Transformer、Bert和ViT**。

我们使用了三个服务器级加速器：Intel Xeon Gold 6240 AVX-512 **CPU**（1.125MB L1缓存，18MB L2缓存和24.75MB L3缓存）、Nvidia **A100**张量核心GPU（最高164KB/SM共享内存，40.96MB L2缓存）和华为**Ascend 910** NPU（64KB L0A/B缓冲区，256KB L0C缓冲区，1MB L1缓冲区，256KB统一缓冲区）。

TABLE IV
THE CONFIGURATIONS OF BATCH GEMM CHAINS.

Name	batch	M	N	K	L	Network
G1	8	512	64	64	512	Bert-Small
G2	12	512	64	64	512	Bert-Base
G3	16	512	64	64	512	Bert-Large
G4	12	256	64	64	256	ViT-Base/14
G5	16	256	64	64	256	ViT-Large/14
G6	16	256	80	80	256	ViT-Huge/14
G7	12	208	64	64	208	ViT-Base/16
G8	16	208	64	64	208	ViT-Large/16
G9	16	208	80	80	208	ViT-Huge/16
G10	1	512	64	64	256	MLP-Mixer
G11	1	768	64	64	384	MLP-Mixer
G12	1	1024	64	64	512	MLP-Mixer

我们的基线包括手工调优的库和最先进的编译器。对于库，我们比较了**PyTorch**（在CPU上使用MKL和oneDNN，在GPU上使用CuBlas和CuDNN）、**TensorRT**和CANN（NPU库）。对于编译器，我们比较了最先进的机器学习编译器，包括**Relay、Ansor、TASO、TVM+Cutlass**和AKG（NPU编译器）。

TABLE V
THE CONFIGURATIONS OF CONVOLUTION CHAINS.

Name	IC	H	W	OC_1	OC_2	st_1	st_2	k_1	k_2
C1	64	112	112	192	128	2	1	3	1
C2	32	147	147	64	80	2	1	3	1
C3	64	56	56	128	64	1	1	3	1
C4	128	28	28	256	128	1	1	3	1
C5	16	227	227	64	16	4	1	3	1
C6	64	56	56	64	64	1	1	1	3
C7	64	56	56	64	64	1	1	1	1
C8	256	56	56	256	64	1	1	1	1

GPU上 chimera比baseline性能要好

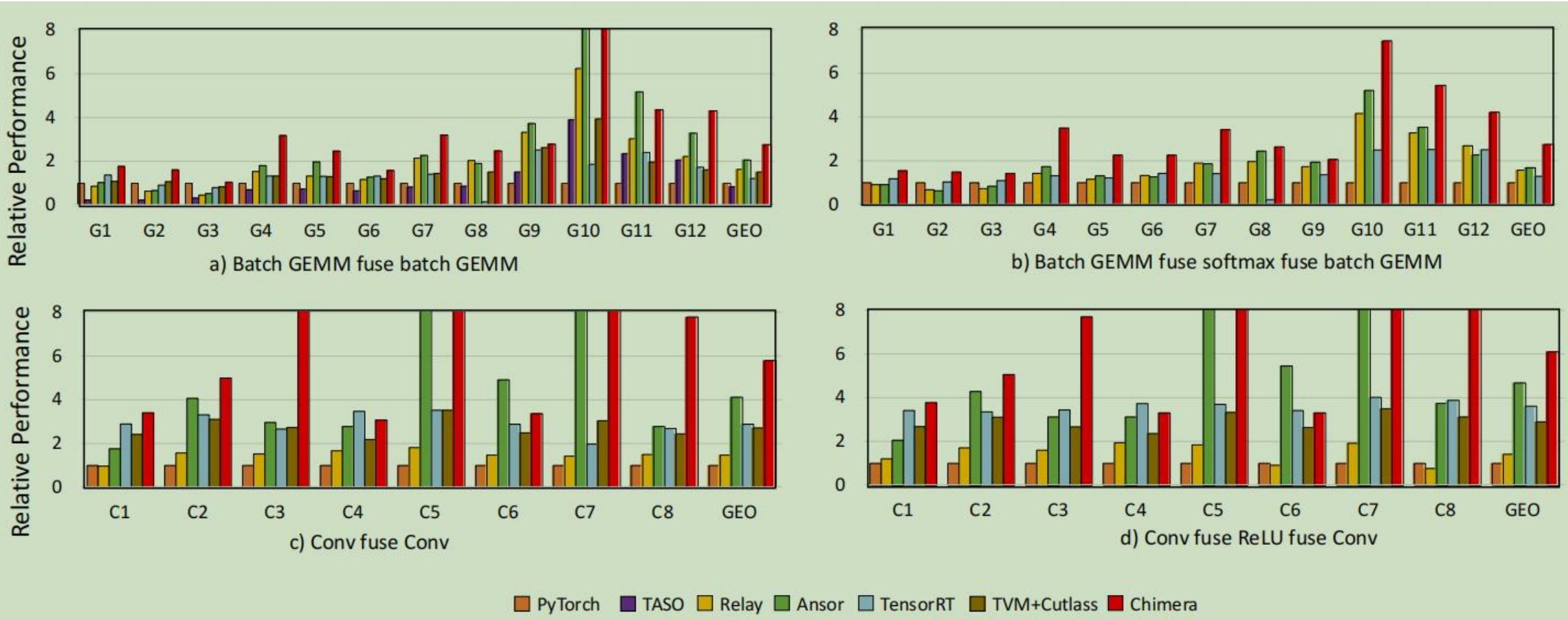


Fig. 6. The performance of fusing batch GEMM chains and fusing convolution chains on GPU.

CPU上 chimera比baseline性能要好

这里和softmax做了结合。额，没说和FA比，我猜基线就不是FA

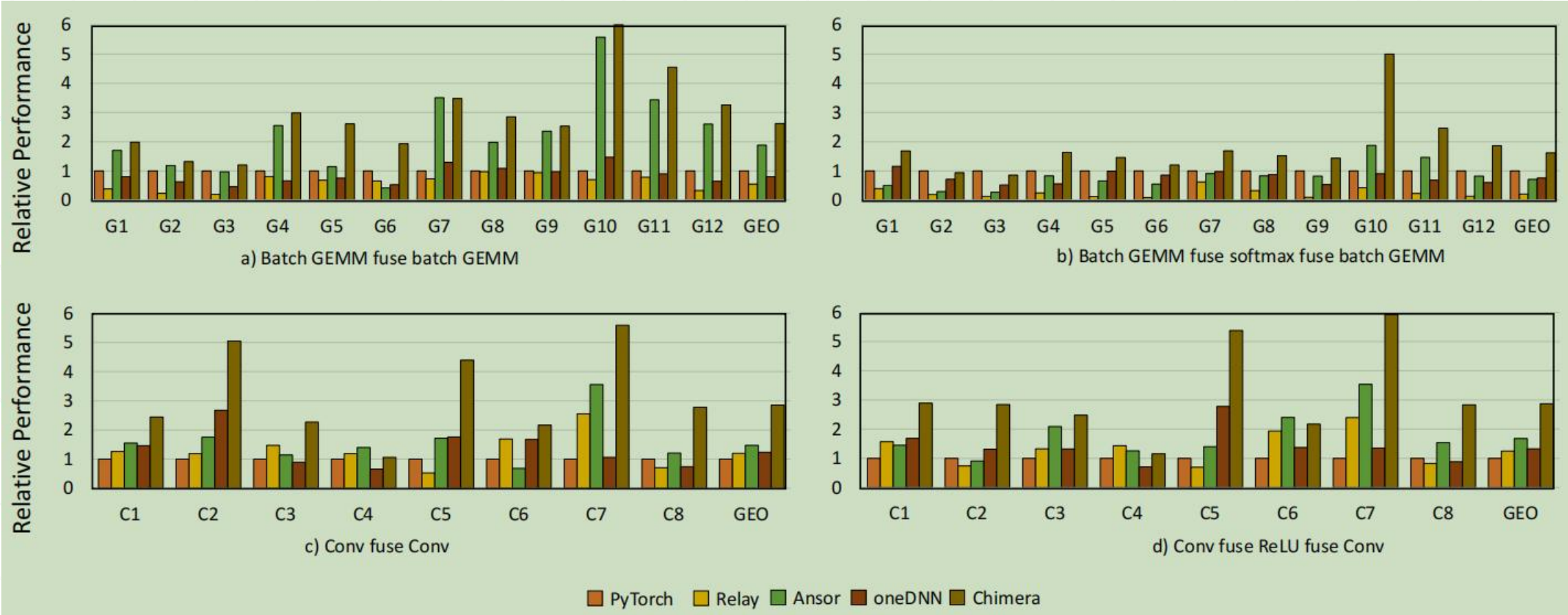


Fig. 5. The performance of fusing batch GEMM chains and fusing convolution chains on CPU.

NPU上 chimera比baseline性能要好

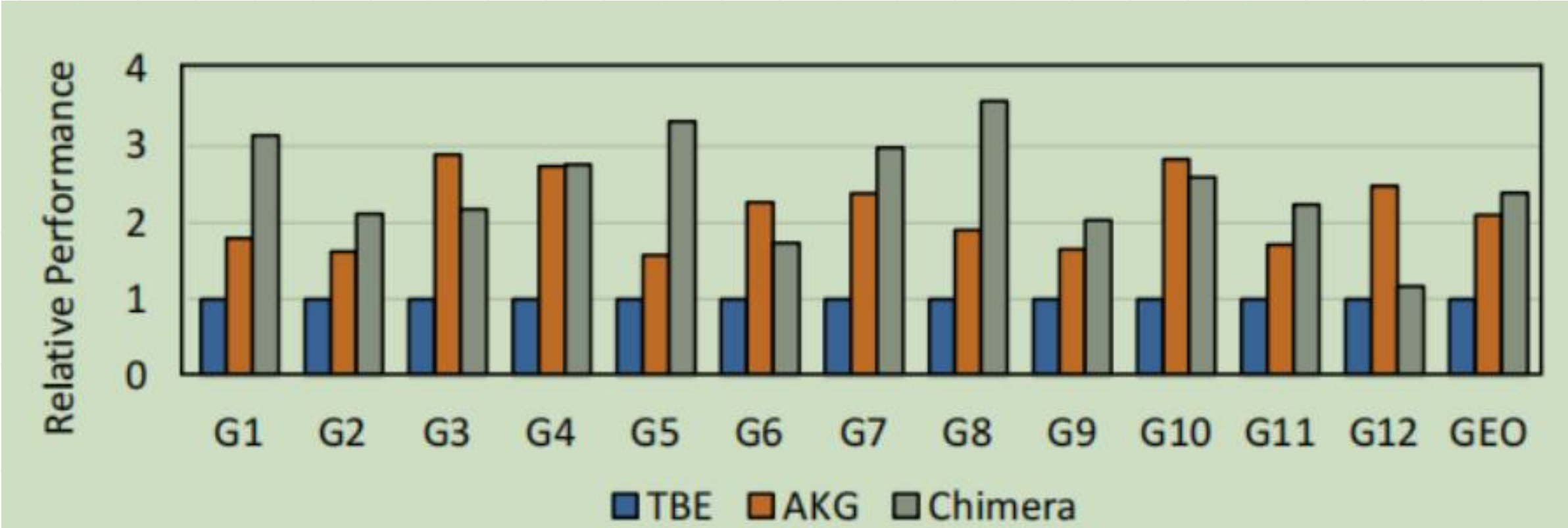


Fig. 7. The performance of fusing GEMM chain on NPU.

CPU上证明，真的能提高cache命中率，以及预期数据访问量和真实非常相符！

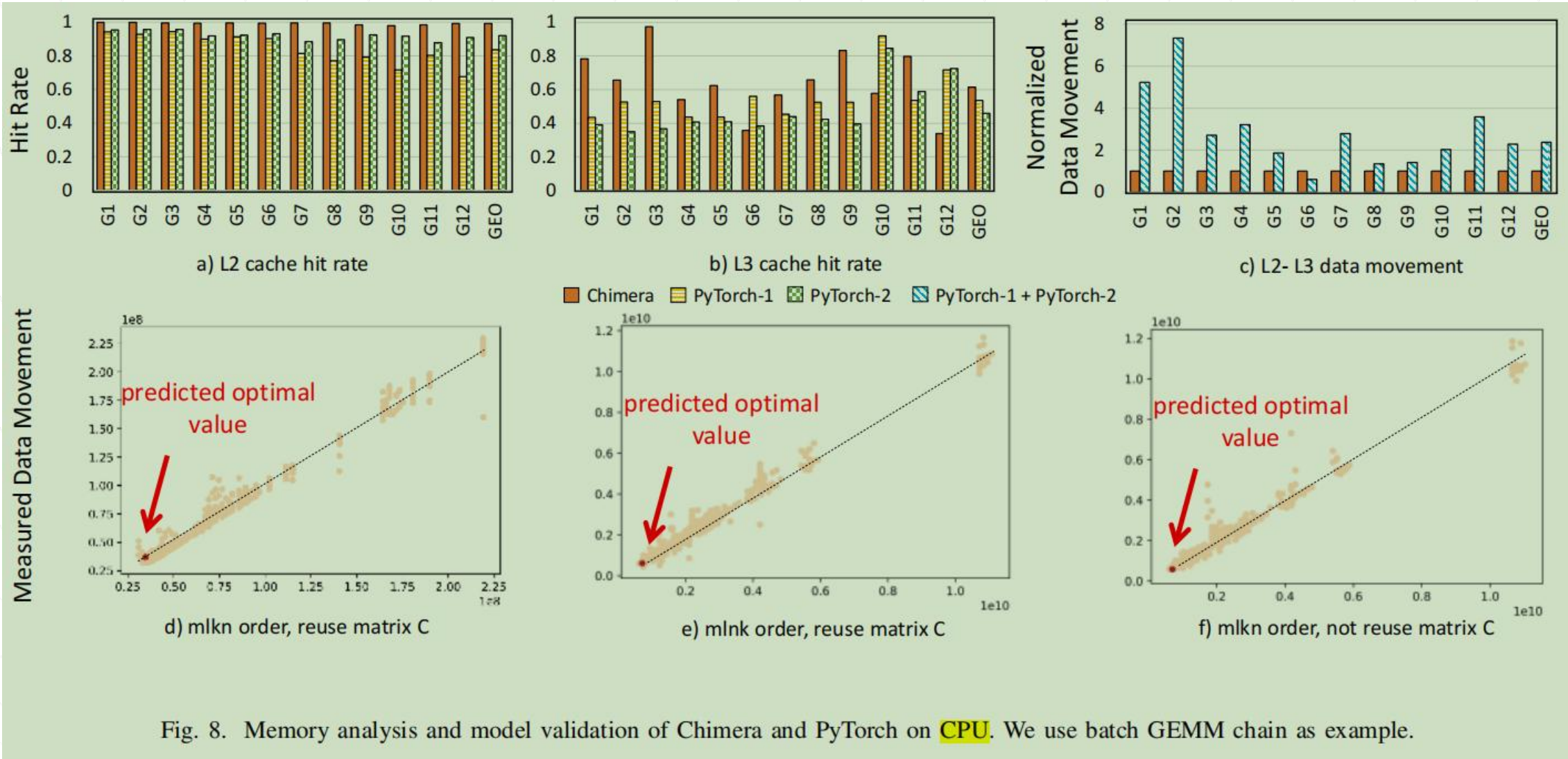


Fig. 8. Memory analysis and model validation of Chimera and PyTorch on CPU. We use batch GEMM chain as example.

五、B200的CTA pair

9.7.16.4.1. CTA Pair

Any 2 CTAs within the cluster whose `%cluster_ctarank` differs by the last bit only is said to form a CTA pair.

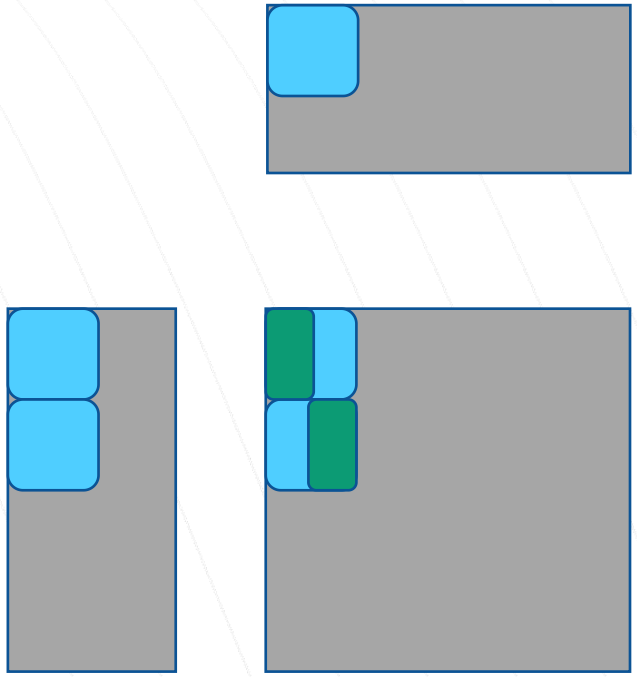
Within a CTA pair, the CTA whose last bit in the `%cluster_ctarank` is:

- 0 is termed the even numbered CTA within the CTA pair.
- 1 is termed as the odd numbered CTA within the CTA pair.

Most of the `tcgen05` operations can either execute at a single CTA level granularity OR at a CTA pair level granularity. When a `tcgen05` operation is performed at CTA pair granularity, the Tensor Memory of both the CTAs within the CTA pair are accessed. The set of threads that need to issue the `tcgen05` operation is listed in the Issue Granularity.

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#tcgen05-cta-pair

在有了2 SM pair之后，类似下面的doc，就有一种猜想，对于右图所示的情况，若在M维度上有2个SM，是不是可以每个SM对于B矩阵，各读一半，然后在需要的时候交换彼此的smem的值？这一点在论文T10里已经提过。（cerebras芯片）



9.7.16.9. TensorCore 5th Generation Matrix Multiply and accumulate Operations

The 5th generation of TensorCore operations of shape *MxNxK* perform matrix multiplication and accumulation of the form:

```
D = A*B+D
```

where:

- the `A` matrix has shape *MxK*, in either Tensor Memory or Shared Memory
- the `B` matrix has shape *KxN*, in Shared Memory of the current CTA and optionally in peer CTA
- the `D` matrix is of the shape *MxN*, in Tensor Memory

```
227 struct SM100_TMA_2SM_LOAD_MULTICAST_1D
228 {
229   CUTE_HOST_DEVICE static void
230   copy(void const* desc_ptr, uint64_t* mbar_ptr, uint16_t multicast_mask, uint64_t
      cache_hint,
231        void      * smem_ptr,
232        int32_t const& crd0)
233   {
234 #if defined(CUTE_ARCH_TMA_SM100_ENABLED)
235     uint64_t gmem_int_desc = reinterpret_cast<uint64_t>(desc_ptr);
236     // Executed by both CTAs. Set peer bit to 0 so that the
237     // transaction bytes will update CTA0's barrier.
238     uint32_t smem_int_mbar = cast_smem_ptr_to_uint(mbar_ptr) & Sm100MmaPeerBitMask;
239     uint32_t smem_int_ptr  = cast_smem_ptr_to_uint(smem_ptr);
240     asm volatile (
241       "cp.async.bulk.tensor.1d.cta_group::2.shared::cluster.global.
      mbarrier::complete_tx::bytes.multicast::cluster.L2::cache_hint"
242       " [%0], [%1, {%4}], [%2], %3, %5;"
243       :
244       : "r"(smem_int_ptr), "l"(gmem_int_desc), "r"(smem_int_mbar), "h"
      (multicast_mask),
245         "r"(crd0), "l"(cache_hint)
246       : "memory");
247 #else
248     CUTE_INVALID_CONTROL_PATH("Trying to use tma without
      CUTE_ARCH_TMA_SM100_ENABLED.");
249 #endif
250   }
251 };
```

include\cute\arch\copy_sm100_tma.hpp

```
346 template <class a_type, class b_type, class c_type,
347           int M, int N, UMMA::Major a_major, UMMA::Major b_major,
348           UMMA::ScaleIn a_neg = UMMA::ScaleIn::One, UMMA::ScaleIn
              b_neg = UMMA::ScaleIn::One>
349 struct SM100_MMA_F16BF16_2x1SM_SS
350 {
351   static_assert(M == 128 || M == 256, "SM100_MMA_F16BF16_2x1SM_SS
      M-mode size should be 128 or 256 for 2 CTA cluster MMA.");
352   static_assert((N % 32 == 0) && (32 <= N) && (N <= 256),
      "SM100_MMA_F16BF16_2x1SM_SS N-mode size should be a multiple of
      32 between 32 and 256.");
353
354   using DRegisters = void;
355   using ARegisters = uint64_t[1];
356   using BRegisters = uint64_t[1];
357   using CRegisters = uint32_t[1];
358
359   CUTE_HOST_DEVICE static void
360   fma(uint64_t const& desc_a,
361       uint64_t const& desc_b,
362       uint32_t const& tmem_c,
363       uint32_t const& scaleC,
364       uint64_t const& idescE)
365   {
```

不过从cutlass里并没有看到这样的写法。当前cutlass的写法类似于TMA-multicast

\include\cute\arch\mma_sm100_umma.hpp

```
366 #if defined(CUTE_ARCH_TCGEN05_F16F32_MMA_ENABLED)
367     if (cute::elect_one_sync()) {
368       uint32_t mask[8] = {0, 0, 0, 0, 0, 0, 0, 0};
369       asm volatile(
370         "{\n\t"
371         ".reg .pred p;\n\t"
372         "setp.ne.b32 p, %4, 0;\n\t"
373         "tcgen05.mma.cta_group::2.kind::f16 [%0], %1, %2, %3,
          {%5, %6, %7, %8, %9, %10, %11, %12}, p; \n\t"
374         "}\n"
375         :
376         : "r"(tmem_c), "l"(desc_a), "l"(desc_b), "r"(uint32_t
          (idescE>>32)), "r"(scaleC),
377           "r"(mask[0]), "r"(mask[1]), "r"(mask[2]), "r"(mask[3]),
378           "r"(mask[4]), "r"(mask[5]), "r"(mask[6]), "r"(mask[7]));
379     }
380 #else
381     CUTE_INVALID_CONTROL_PATH("Attempting to use
      SM100_MMA_F16BF16_2x1SM_SS without
      CUTE_ARCH_TCGEN05_F16F32_MMA_ENABLED");
382 #endif
383   }
384 };
```

```
206  template <class ClusterShapeMNK, class AtomThrId>
207  constexpr auto
208  sm100_cluster_shape_to_tma_atom_A(ClusterShapeMNK
     cluster_shape_mnk, AtomThrId atom_thr_id) {
209    static_assert(cute::rank(cluster_shape_mnk) == 3);
210    constexpr bool IsDynamicCluster = not
       cute::is_static_v<ClusterShapeMNK>;
211
212    if constexpr (cute::size(atom_thr_id) == 2) {
213      if constexpr (!IsDynamicCluster) {
214        static_assert(cute::size<0>(cluster_shape_mnk) % 2 == 0,
         "Cluster shape not divisible by MMA size");
215        if constexpr (cute::size<1>(cluster_shape_mnk) == 1) {
216          return cute::SM100_TMA_2SM_LOAD{};
217        }
218        else {
219          return cute::SM100_TMA_2SM_LOAD_MULTICAST{};
220        }
221      }
222      else {
223        return cute::SM100_TMA_2SM_LOAD_MULTICAST{};
224      }
225    }
```

\include\cutlass\gemm\collective\builders\sm100_common.inl

非常奇怪的是，multicast和2SM并不冲突，而且对A和对B在何时使用2SM的条件并不相同。

```
242  template <class ClusterShapeMNK, class AtomThrId>
243  constexpr auto
244  sm100_cluster_shape_to_tma_atom_B(ClusterShapeMNK
     cluster_shape_mnk, AtomThrId atom_thr_id) {
245    static_assert(cute::rank(cluster_shape_mnk) == 3);
246    constexpr bool IsDynamicCluster = not
       cute::is_static_v<ClusterShapeMNK>;
247
248    if constexpr (cute::size(atom_thr_id) == 2) {
249      if constexpr (!IsDynamicCluster) {
250        static_assert(cute::size<0>(cluster_shape_mnk) % 2 == 0,
         "Cluster shape not divisible by MMA size");
251        if constexpr (cute::size<0>(cluster_shape_mnk) == 2) {
252          return cute::SM100_TMA_2SM_LOAD{};
253        }
254        else {
255          return cute::SM100_TMA_2SM_LOAD_MULTICAST{};
256        }
257      }
258      else {
259        return cute::SM100_TMA_2SM_LOAD_MULTICAST{};
260      }
```

当前的结论是悬而未决。

谢谢大家！