从汇编到 PyTorch：基于 MI300X 的 DeepSeek 算子全栈设计和调优

RadeonFlow Team

🏆 Grand Prize Winner of AMD Developer Contest 2025

2025/8/2

About Us

王延葵

上海交通大学，并行与分布式系统研究所，硕士生二年级

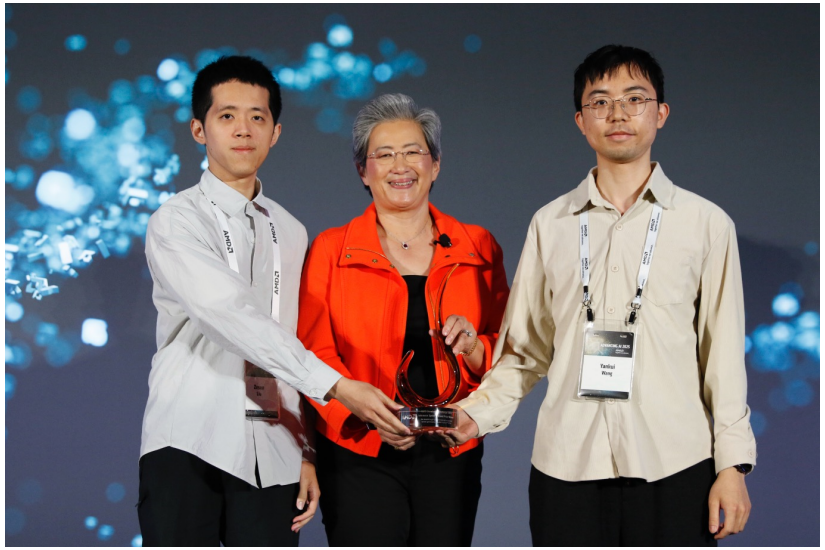
研究方向是大模型系统。

刘泽森

南京大学，硕士生二年级，研究方向聚焦系统性能观测与优化 (AI for OS)。

郝英屹

上海交通大学，并行与分布式研究所，博士生一年级，研究方向是大模型推理系统。



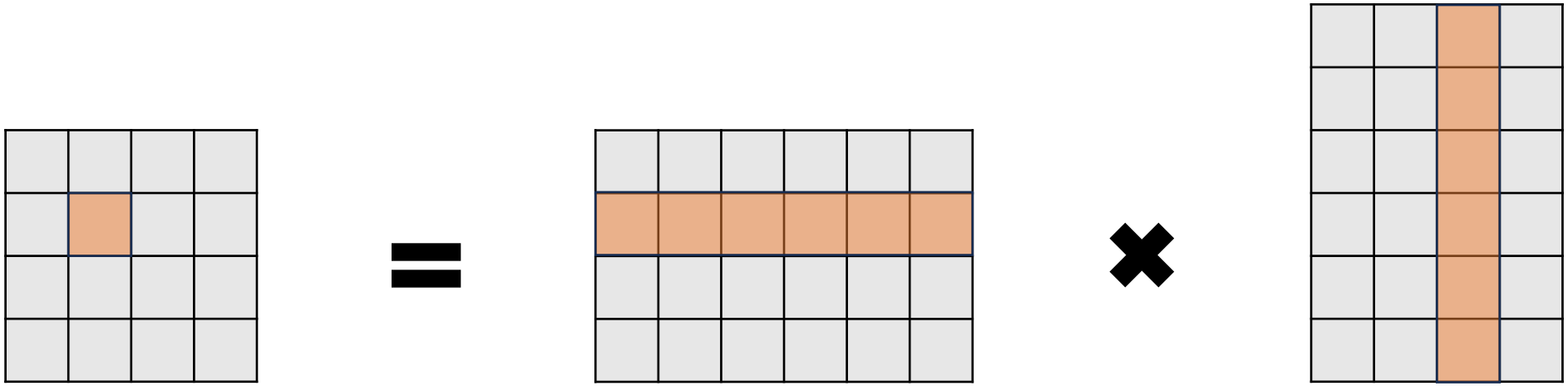
CEO Lisa Su 于 AMD Advancing AI 2025 发布会为我们队伍颁奖

Outline

- **Part 1**: 基于 HIP 的 GEMM-FP8 算子实现
- **Part 2**: 基于 HIP 和 rocBLAS 的 MoE 算子实现
- **Part 3**: 优化 PyTorch Eager MLA 算子

Part 1: 基于 HIP 的 GEMM-FP8 算子实现

GEMM (General Matrix Multiplication) - 矩阵乘法



C A B

- **矩阵乘法:** $C = A \times B$

$$C(i,j) = \sum A(i,k) \times B(k,j)$$

- **FP8矩阵乘法:** $C = \alpha \cdot A \times B \cdot \beta$ a

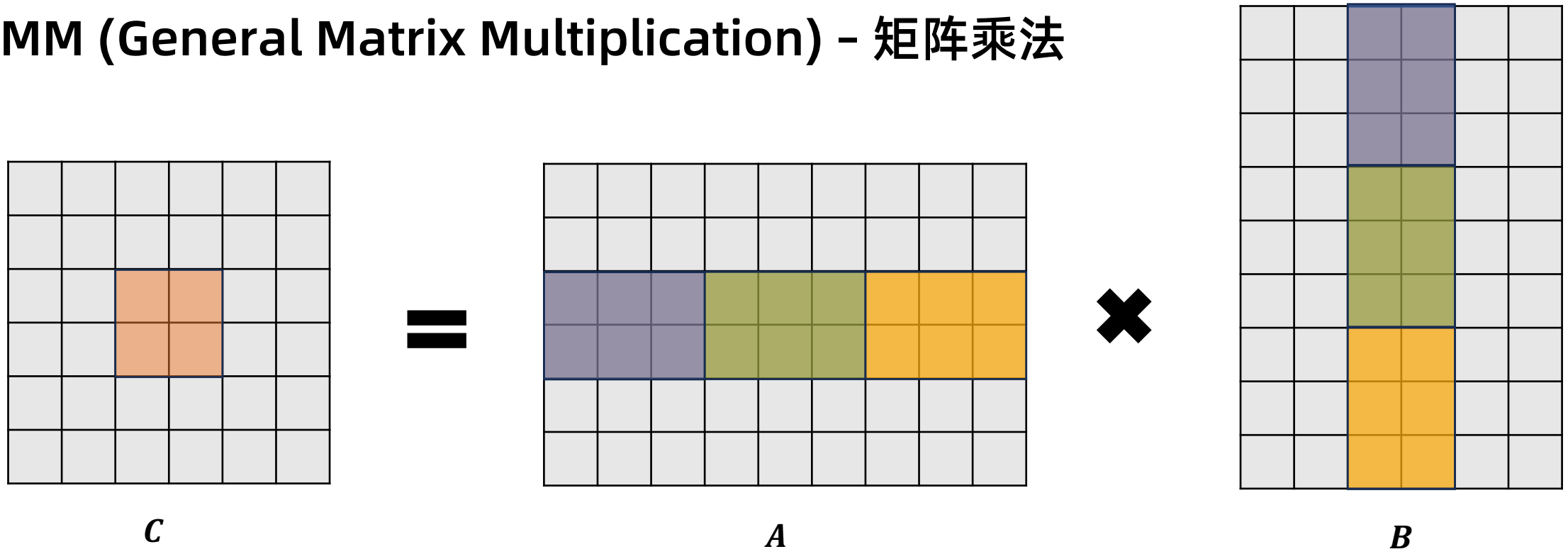
其中$A \in \mathbb{FP}_8^{M \times N}$, $B \in \mathbb{FP}_8^{M \times K}$, $C \in \mathbb{BP}_{16}^{M \times K}$

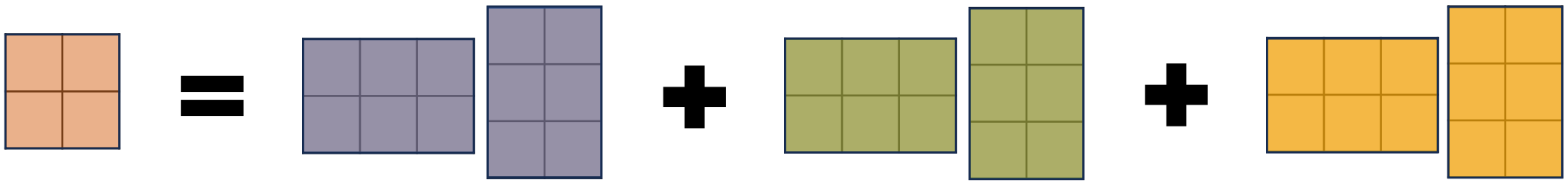
$$\alpha \in \mathbb{FP}_{32}^{M \times (K/128)}$$

$$\beta \in \mathbb{FP}_{32}^{(N/128) \times (K/128)}$$

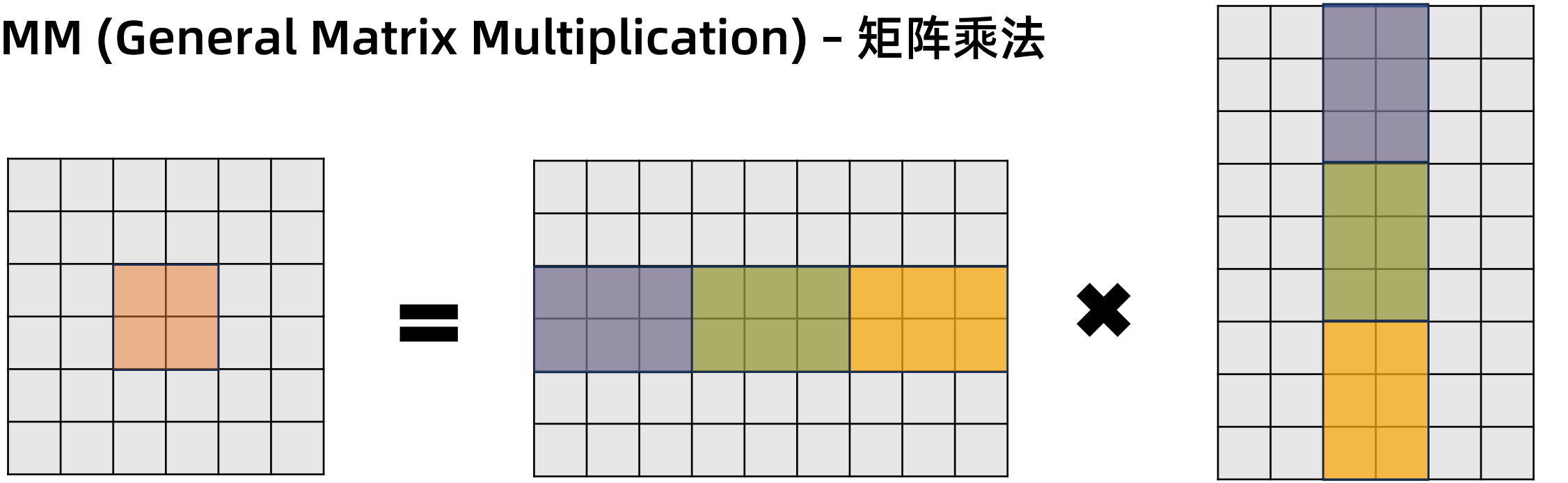
AI PEAK THEORETICAL PERFORMANCE	
TF32 (TFLOPs)	653.7
FP16 (TFLOPs)	1307.4
BFLOAT16 (TFLOPs)	1307.4
INT8 (TOPS)	2614.9
FP8 (TFLOPS)	2614.9

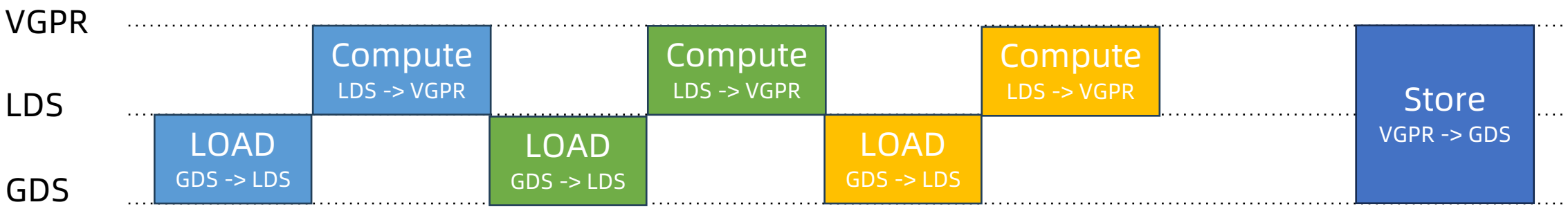
GEMM (General Matrix Multiplication) - 矩阵乘法



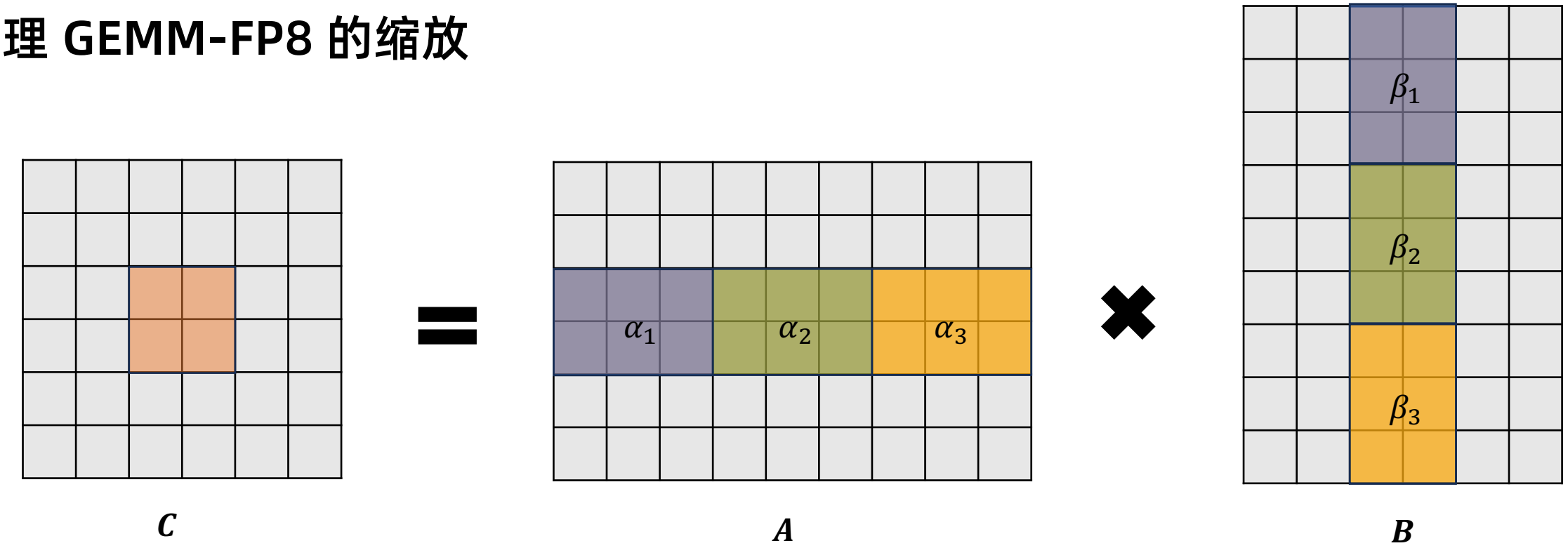


GEMM (General Matrix Multiplication) - 矩阵乘法

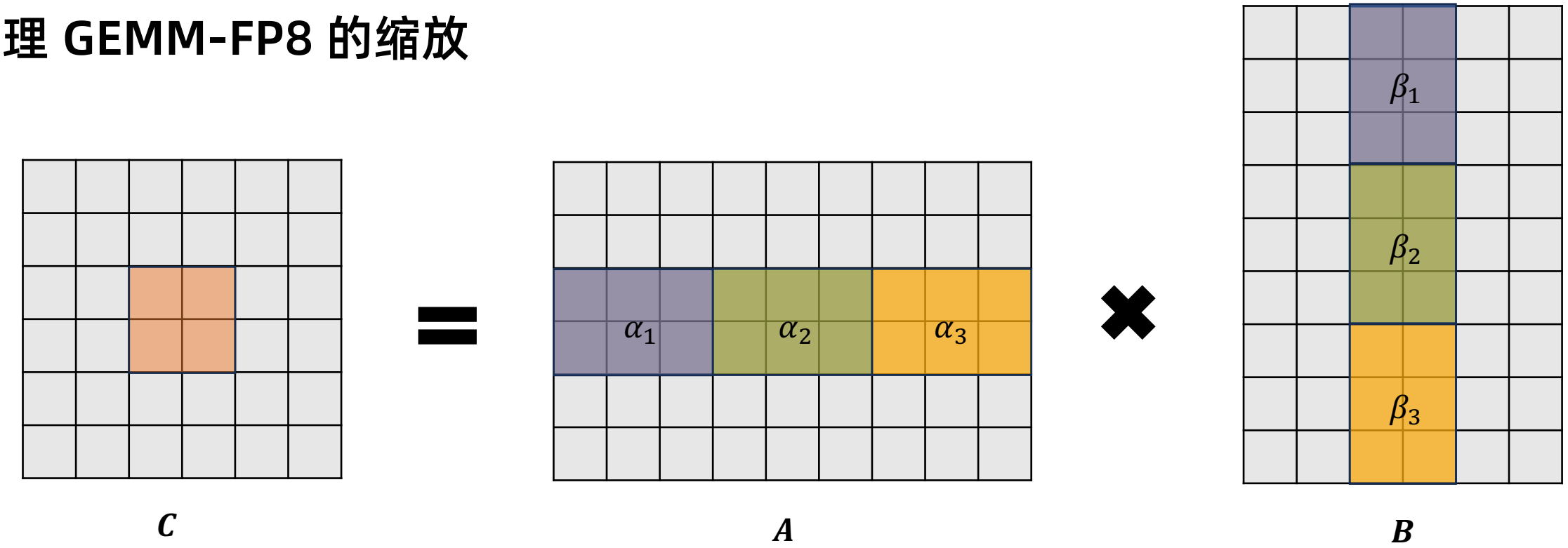


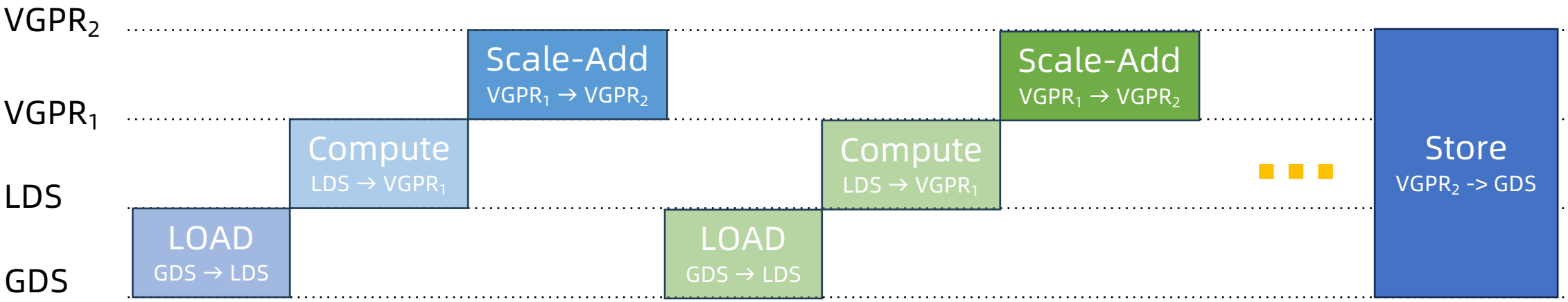


处理 GEMM-FP8 的缩放



处理 GEMM-FP8 的缩放

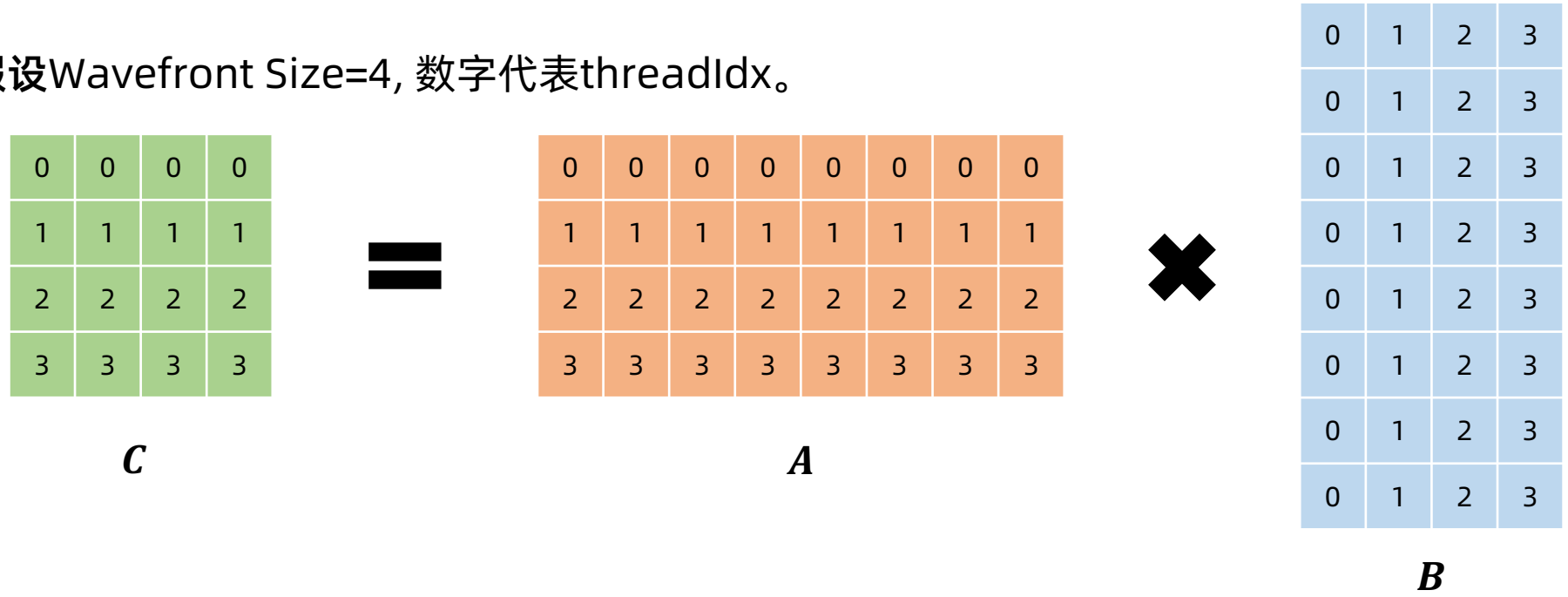




优化数据读取效率 - 预转置

- MatrixCore指令要求数据的特定寄存器布局，以指令*V_MFMA_F32_32X32X16_FP8_FP8*为例：

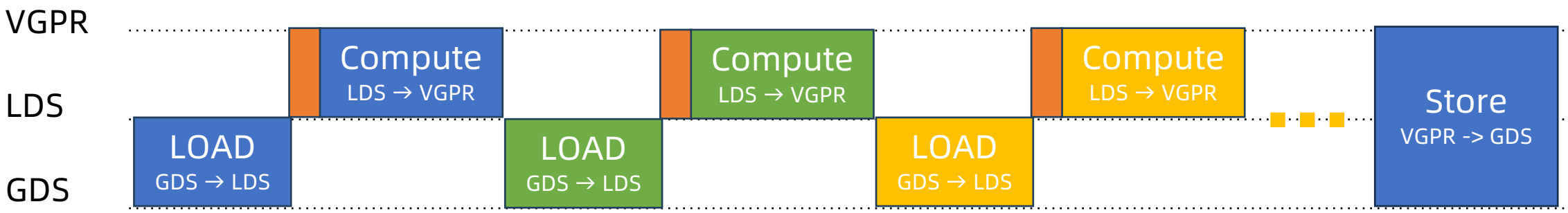
假设Wavefront Size=4, 数字代表threadIdx。



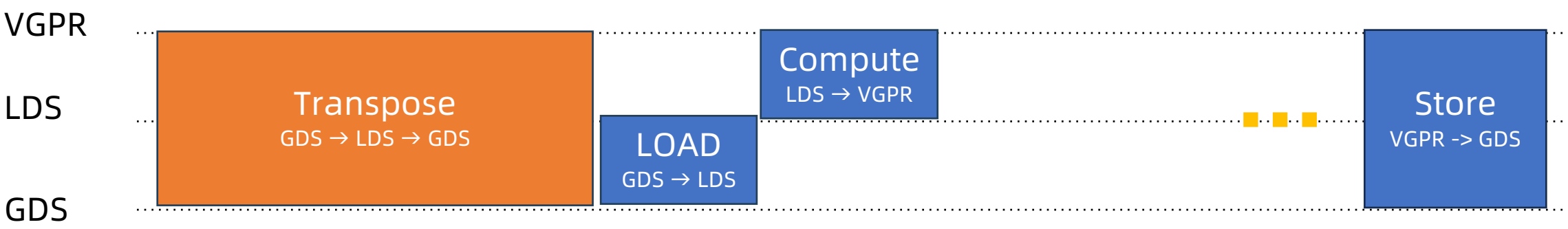
- 因此，矩阵A的最佳内存布局是Row-Major，矩阵B的最佳布局的最佳内存布局是Col-Major。
  然而，题目中的矩阵A的内存布局是Col-Major，矩阵B的内存布局是Row-Major。
- 我们实现了一个矩阵转置的高效算子。 吞吐： ~ 3TB/s

优化数据读取效率 - 预转置

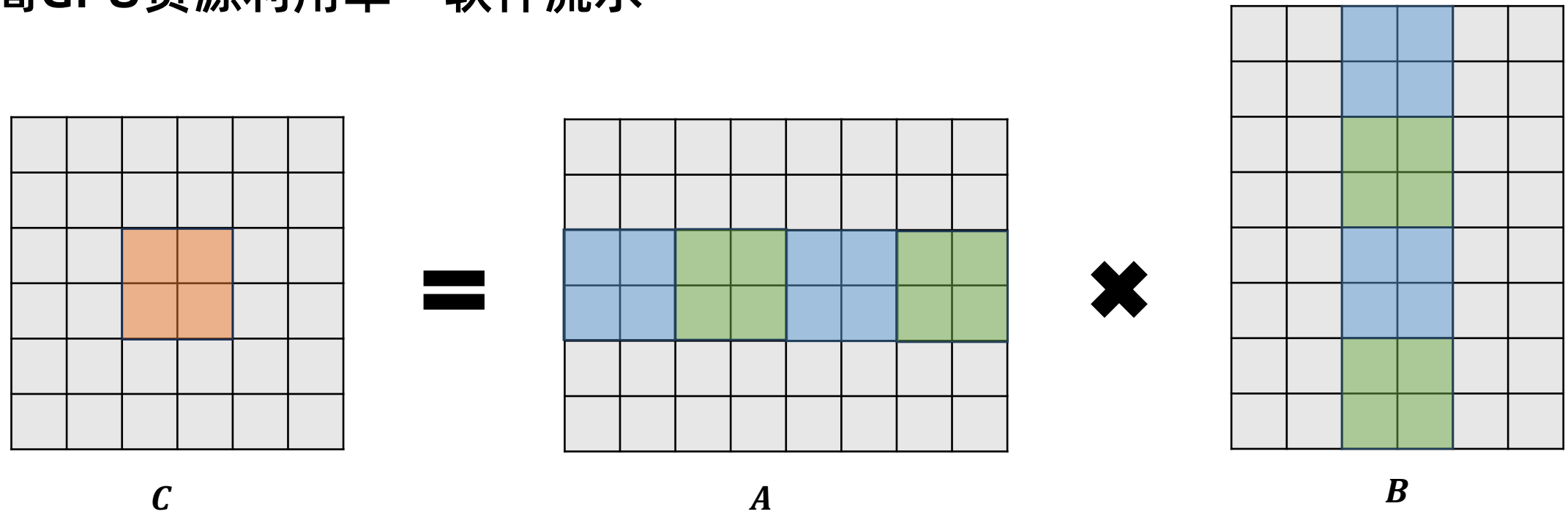
- **方案一：加载时不转置，计算时转换寄存器布局，总耗时 ~120μs**



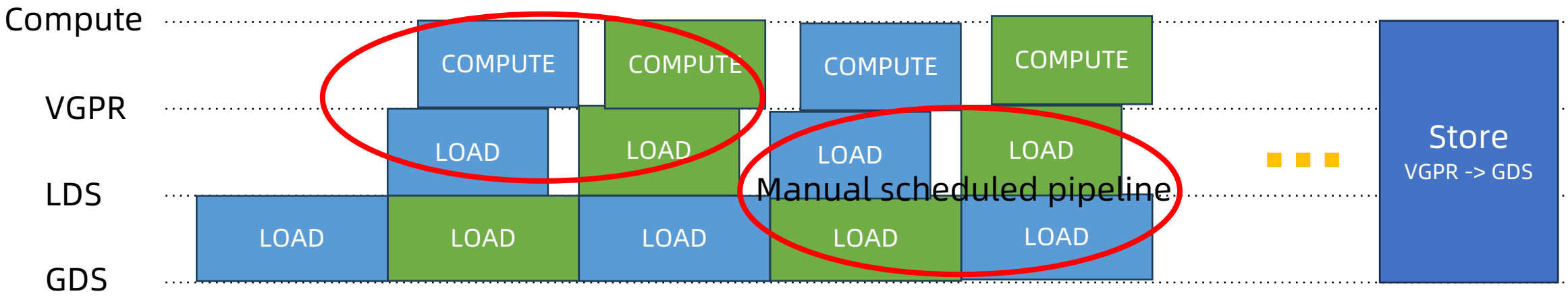
- **方案二：计算前先将矩阵A和矩阵B转置，总耗时 ~100μs，转置耗时 ~10μs**



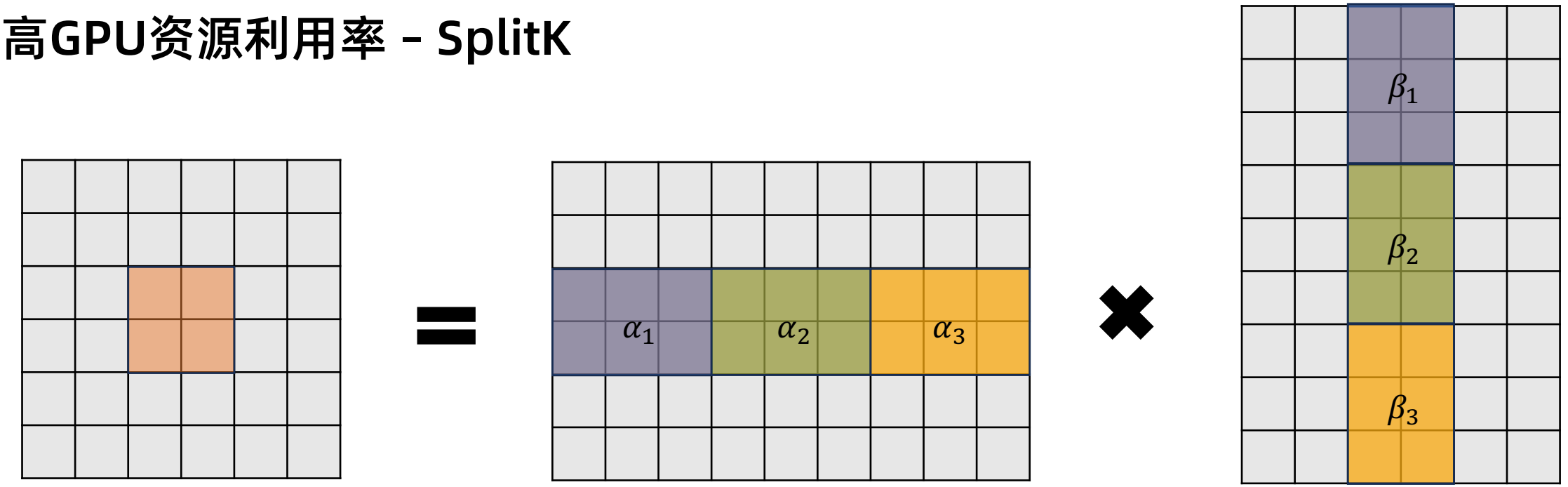
提高GPU资源利用率 – 软件流水

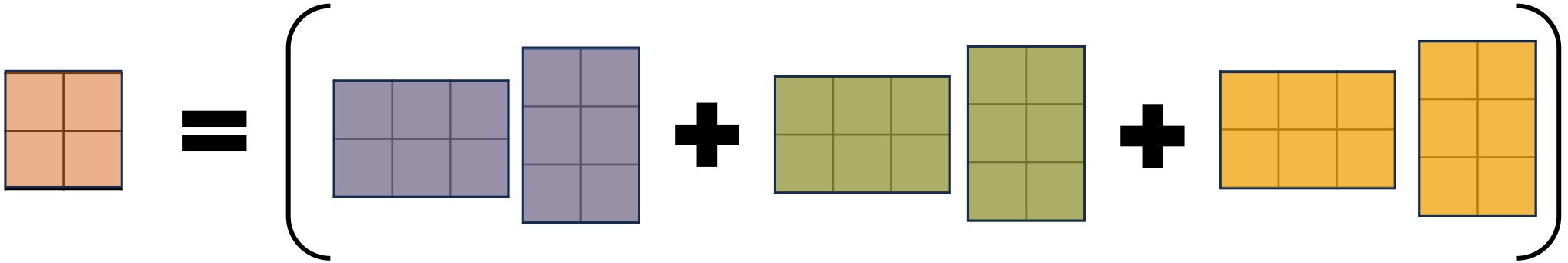


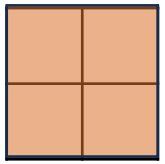
Compiler scheduled pipeline



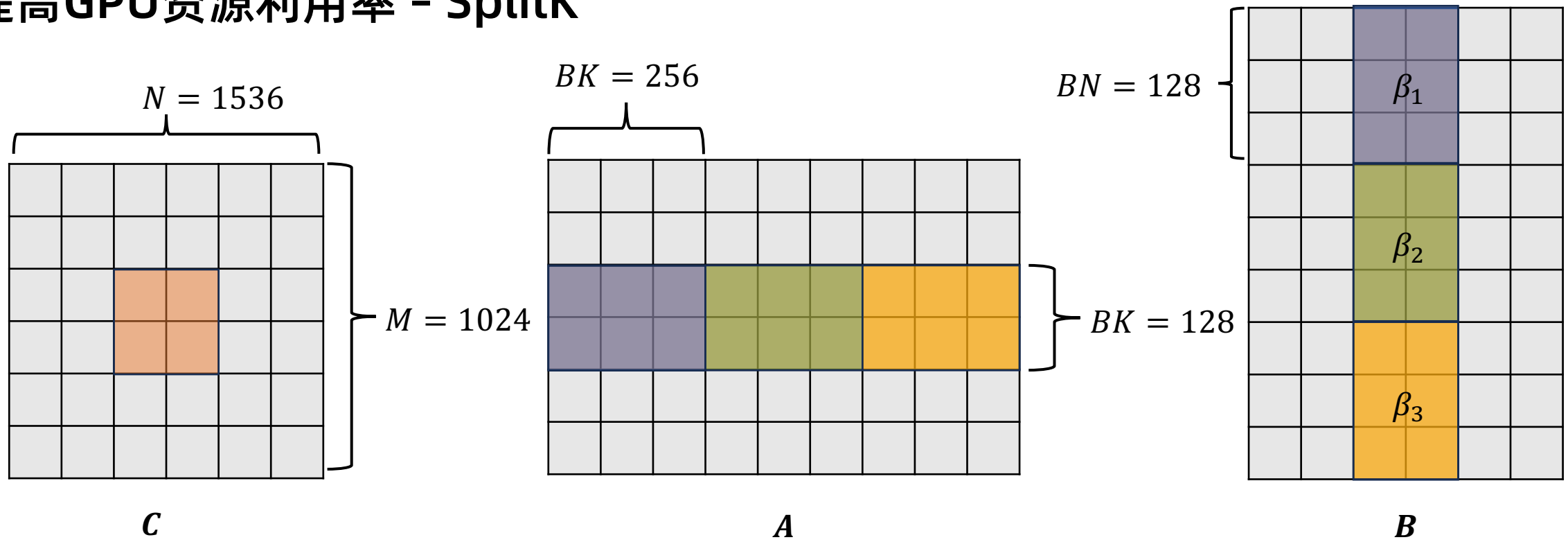
提高GPU资源利用率 - SplitK





每个  由1个Block计算，总共需要9个Block完成整个计算过程

提高GPU资源利用率 - SplitK

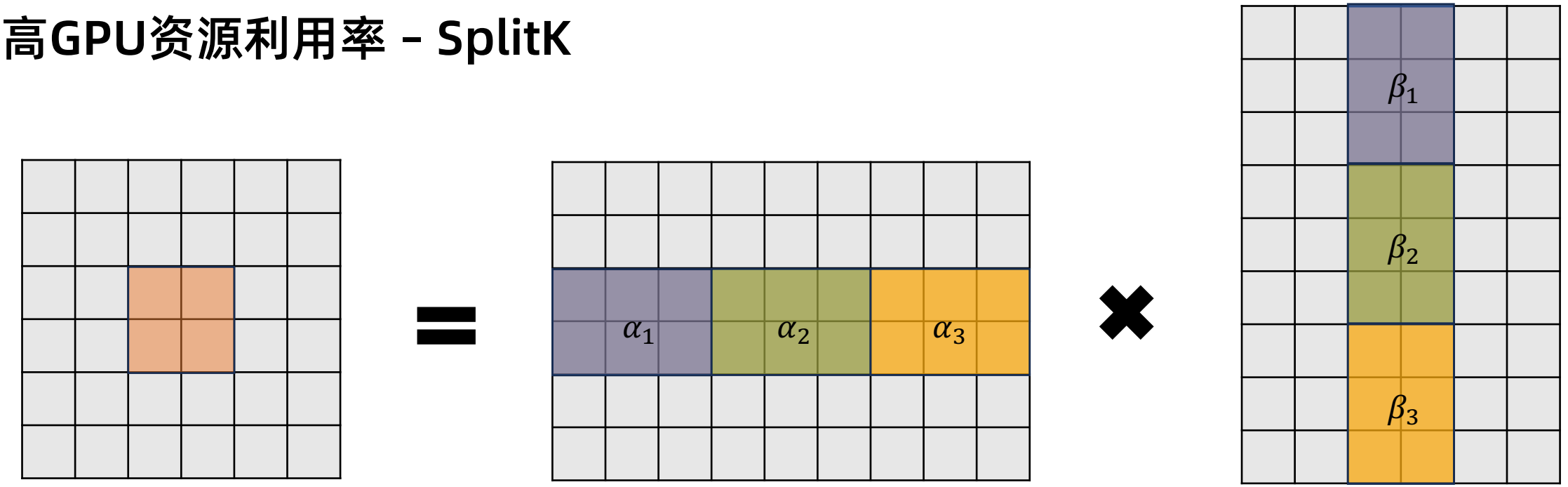


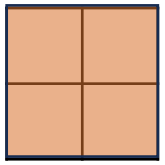
一个实际的例子：

$$\frac{N}{BN} \times \frac{M}{BM} = 4 \times 12 = 48$$

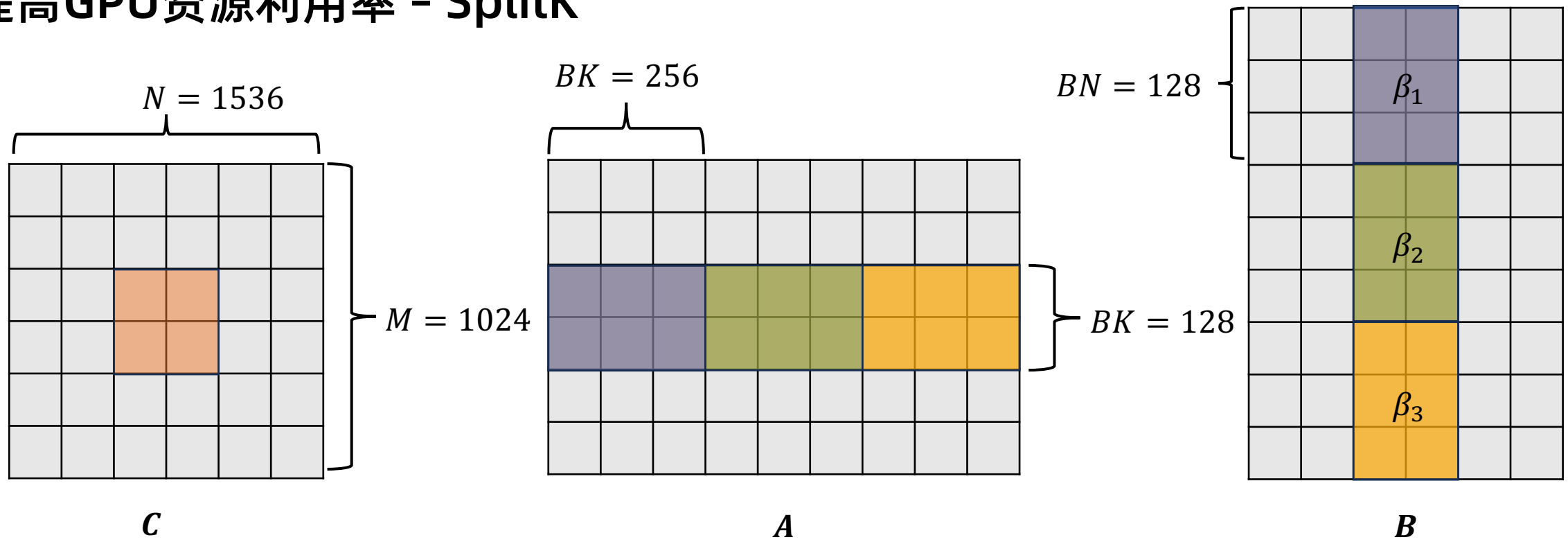
- 但是MI300X有304个CU，GPU峰值利用率：$48/304 = 15.7\%$

提高GPU资源利用率 – SplitK



每个  由3个Block计算，总共需要27个Block完成整个计算过程

提高GPU资源利用率 - SplitK



一个实际的例子：

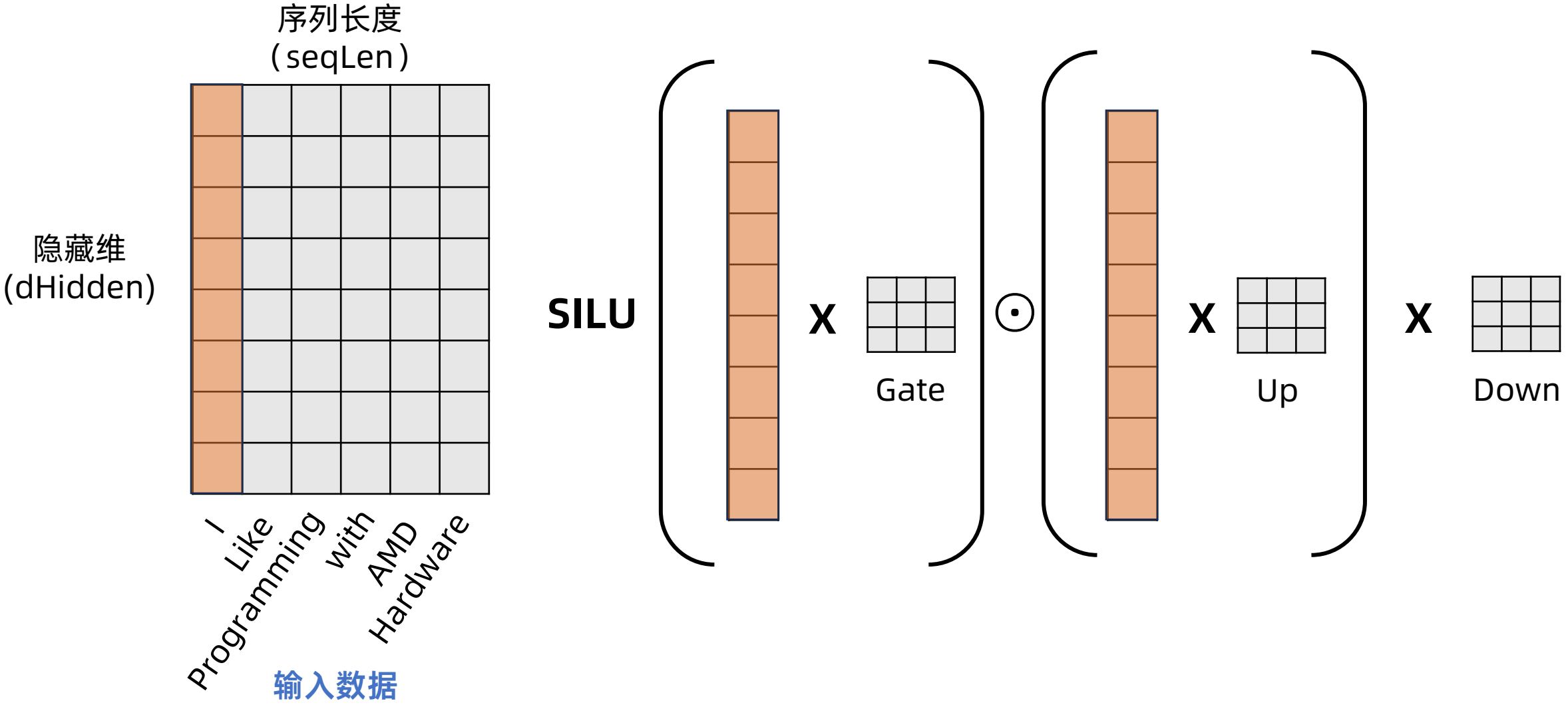
$$\frac{N}{BN} \times \frac{M}{BM} = 4 \times 12 = 48$$

- 但是MI300X有304个CU，GPU峰值利用率：100%
- SplitK需要进行额外在计算完成后进行一次额外的Reduce操作，以及额外的显存用于存放中间结果。

Part 2: 基于 HIP 和 rocBLAS 的 MoE 算子实现

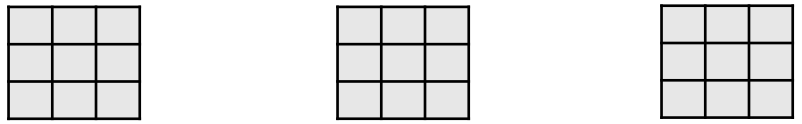
From FFN to MoE (Mixture of Experts)

FFN 模型通过 Gate, Up 和 Down 三个 projection transform 单个 token



From FFN to MoE (Mixture of Experts)

FFN 模型通过 Gate, Up 和 Down 三个 projection transform 单个 token

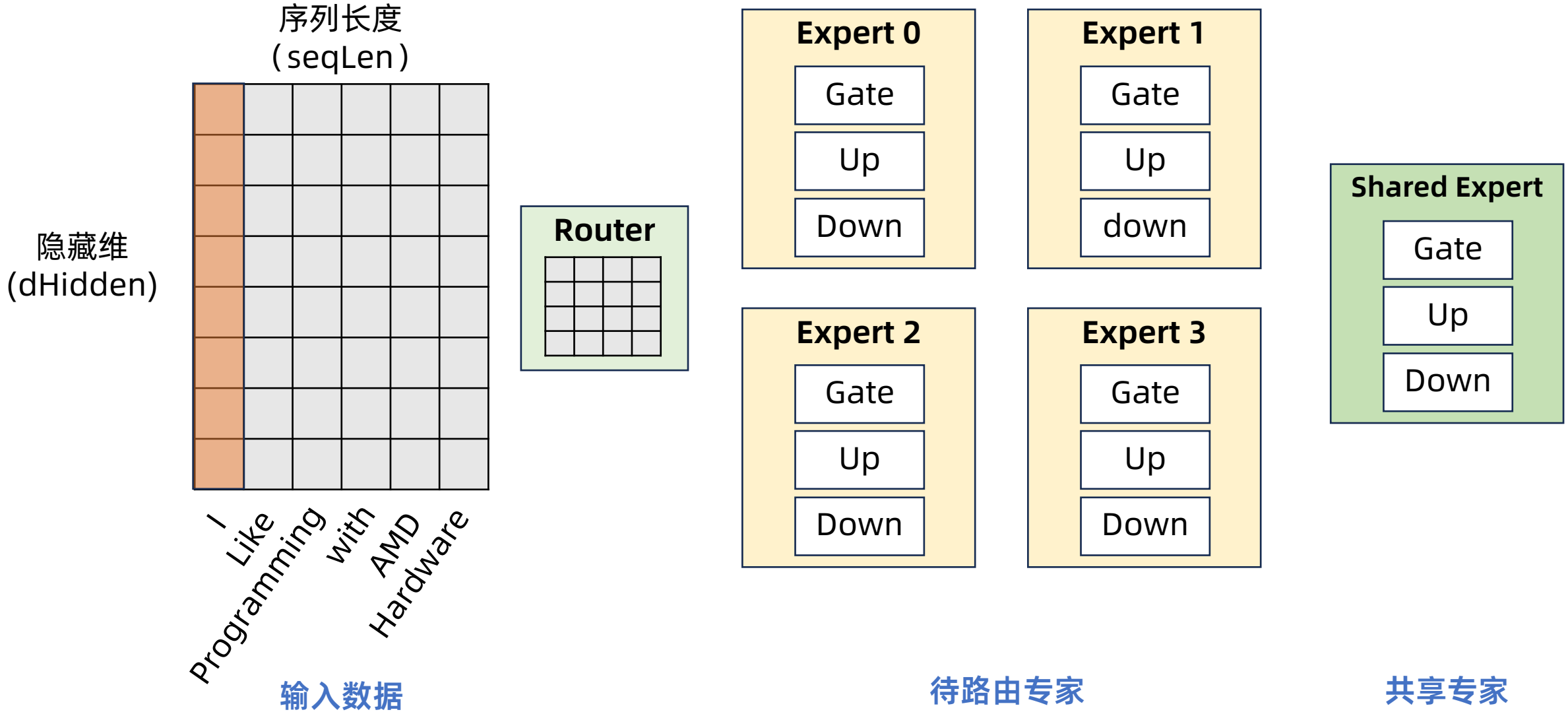


然而，传统 Dense FFN 架构在模型规模上存在瓶颈：

- 推理时大部分参数冗余
- 模型训练需要大量算力资源

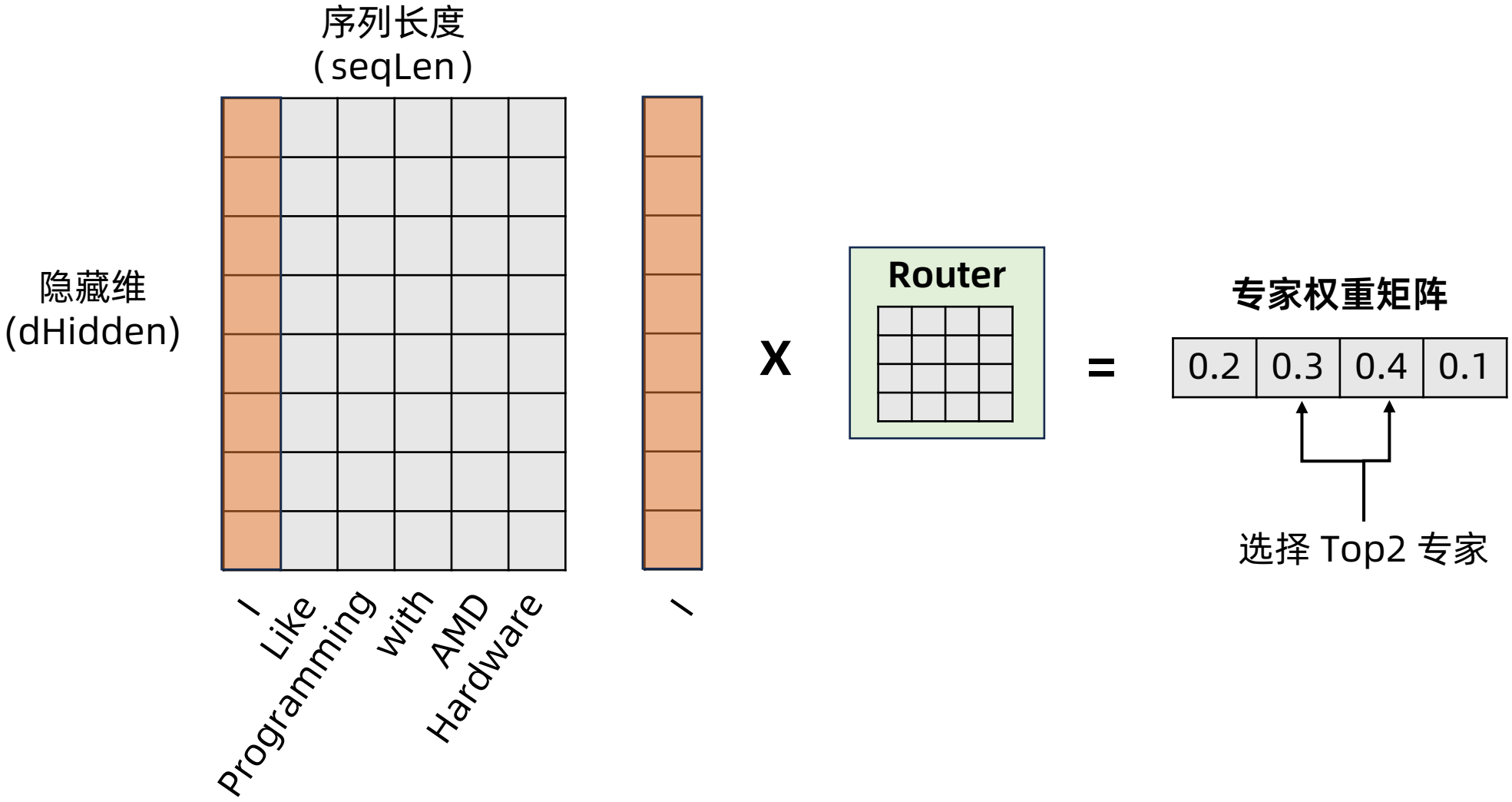
MoE - 专家路由

MoE 使用门控网络动态选择 Top-K 专家，在一次计算中仅激活少量参数



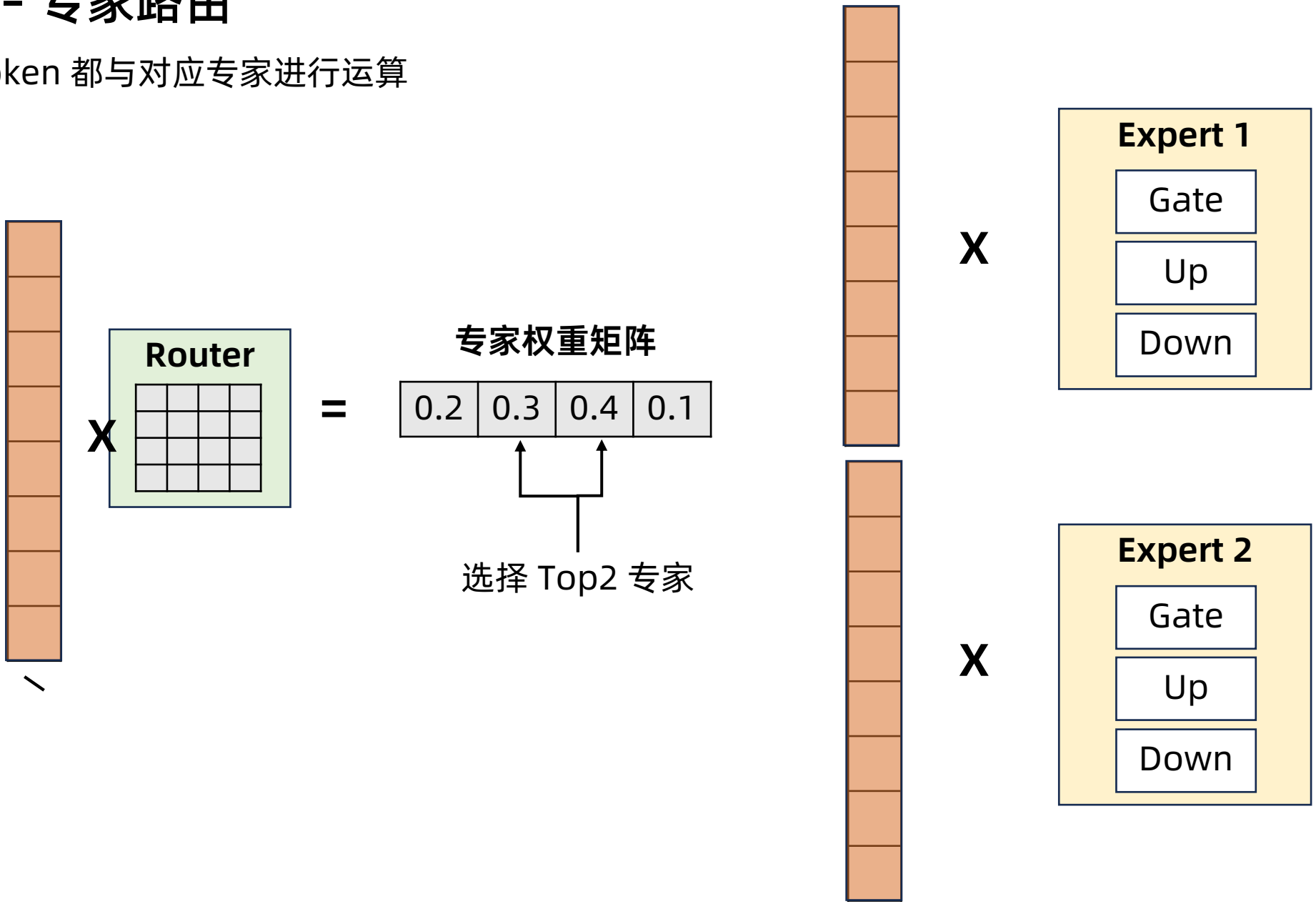
MoE – 专家路由

每个 token 都会被路由到对应专家（此处展示配置路由 2 专家的情形）



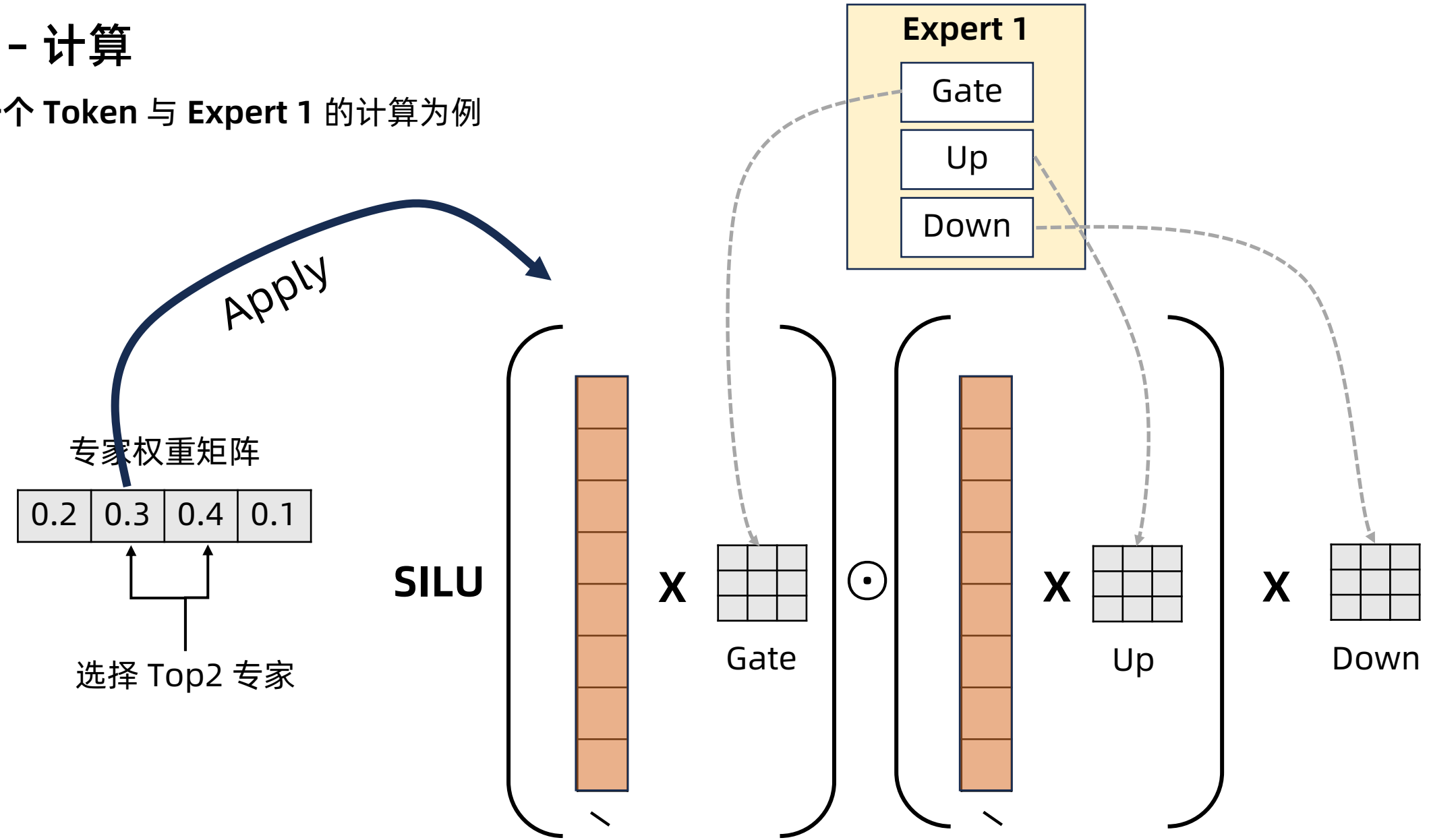
MoE - 专家路由

每个 Token 都与对应专家进行运算



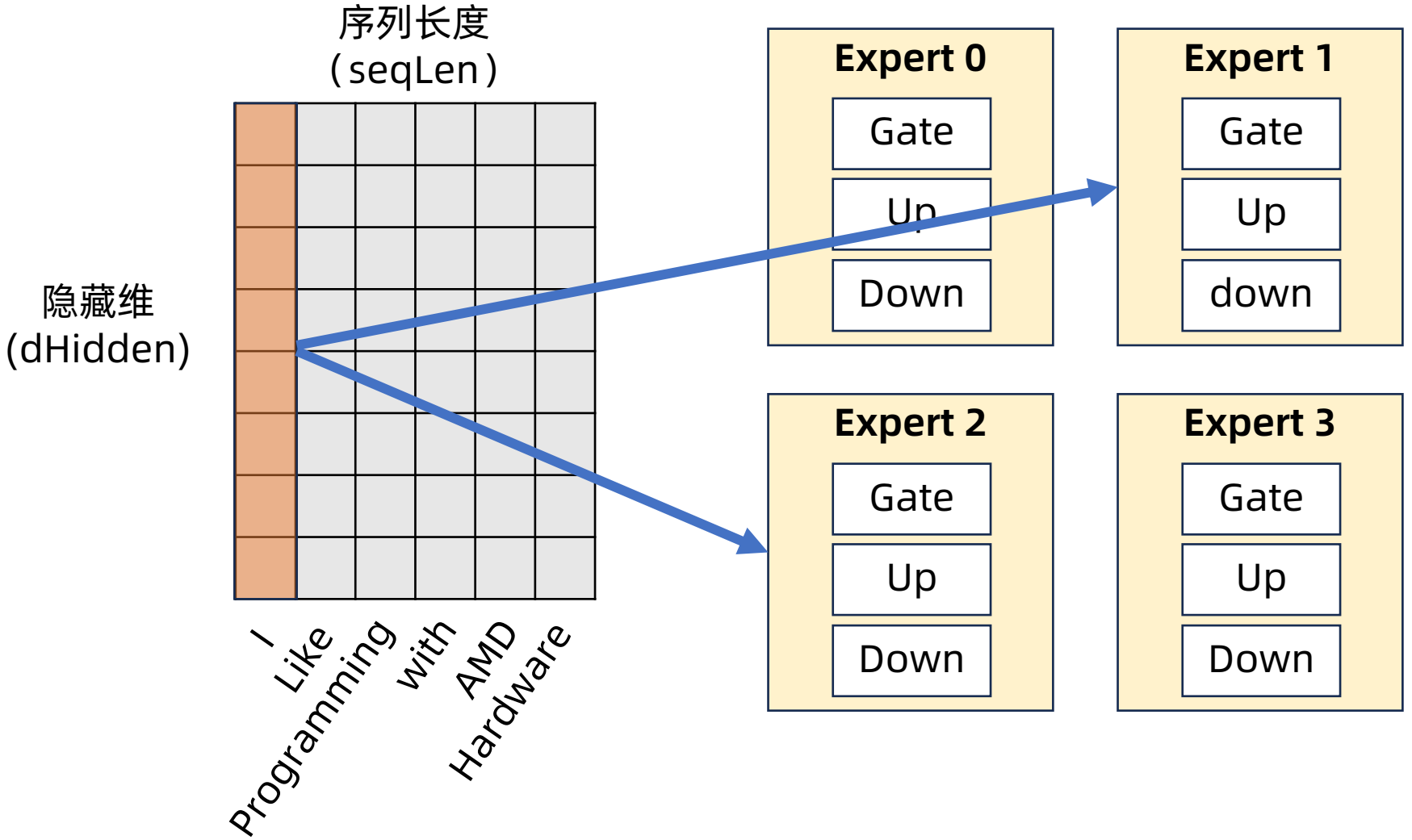
MoE - 计算

以 **第一个 Token** 与 **Expert 1** 的计算为例



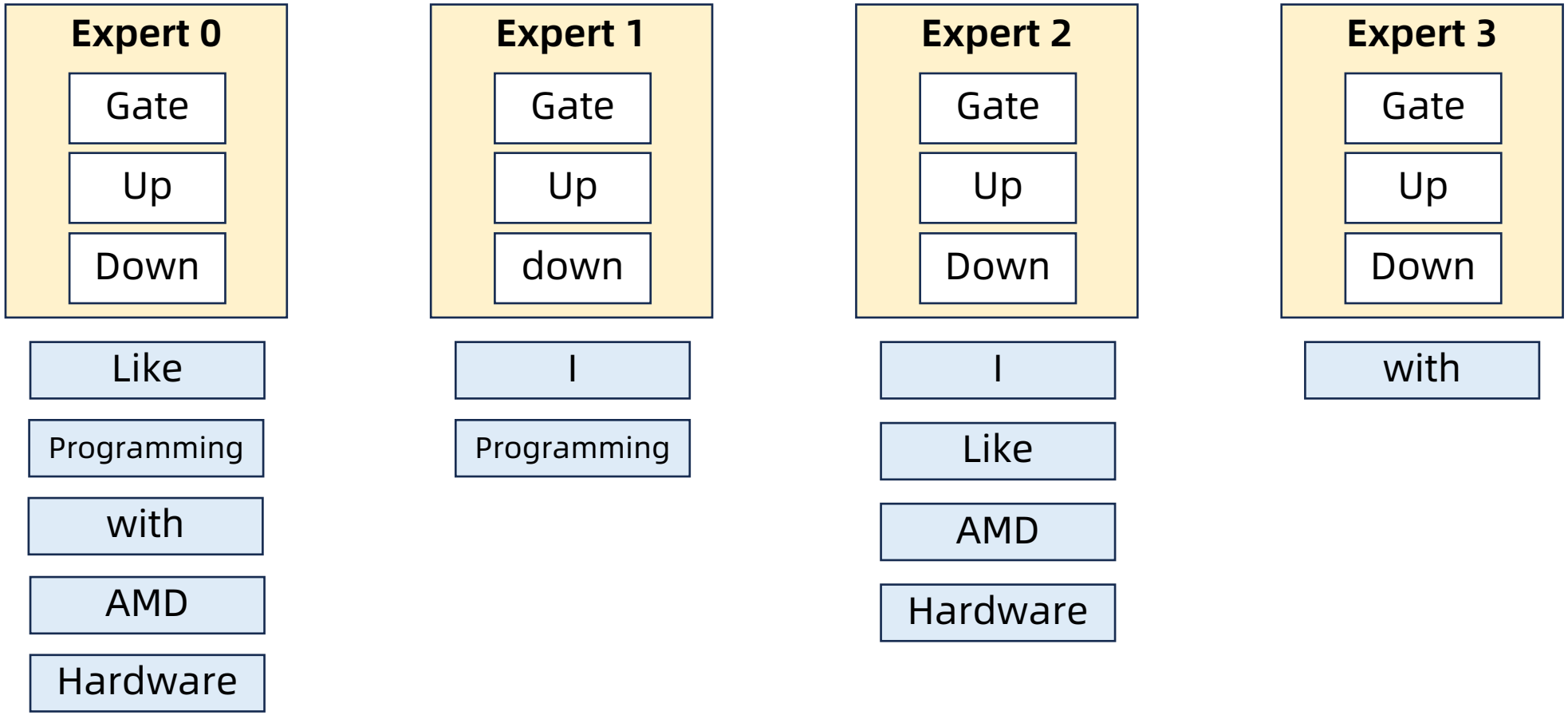
MoE - 计算

Token 和专家之间存在“一对多”关系



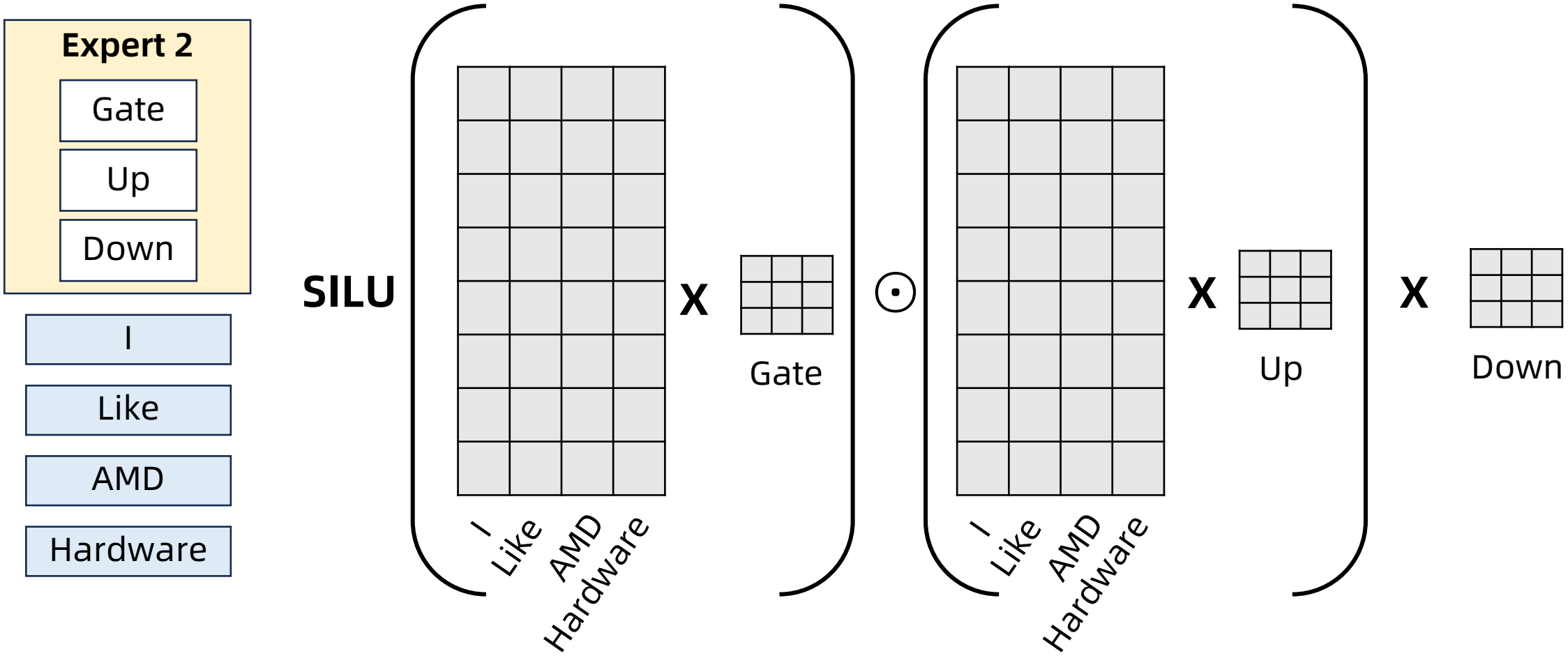
Expert Centric!

- 每个 Token 需要2个专家服务
- 每个专家服务多个 Token，且专家之间的计算没有依赖关系



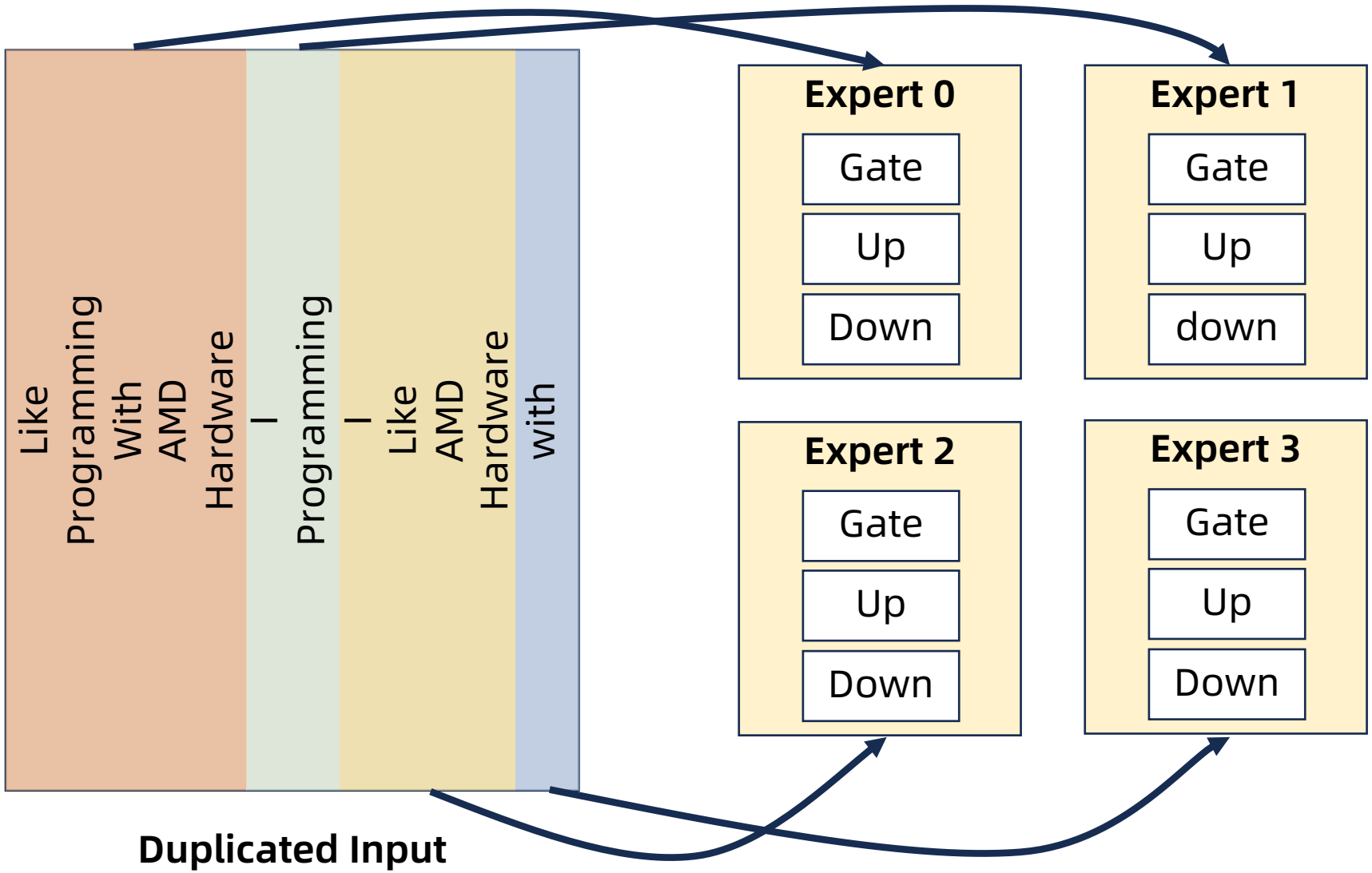
Expert Centric!

考虑 Expert 2 的计算，可以使用一次矩阵乘法完成其负责的四个 token 的计算

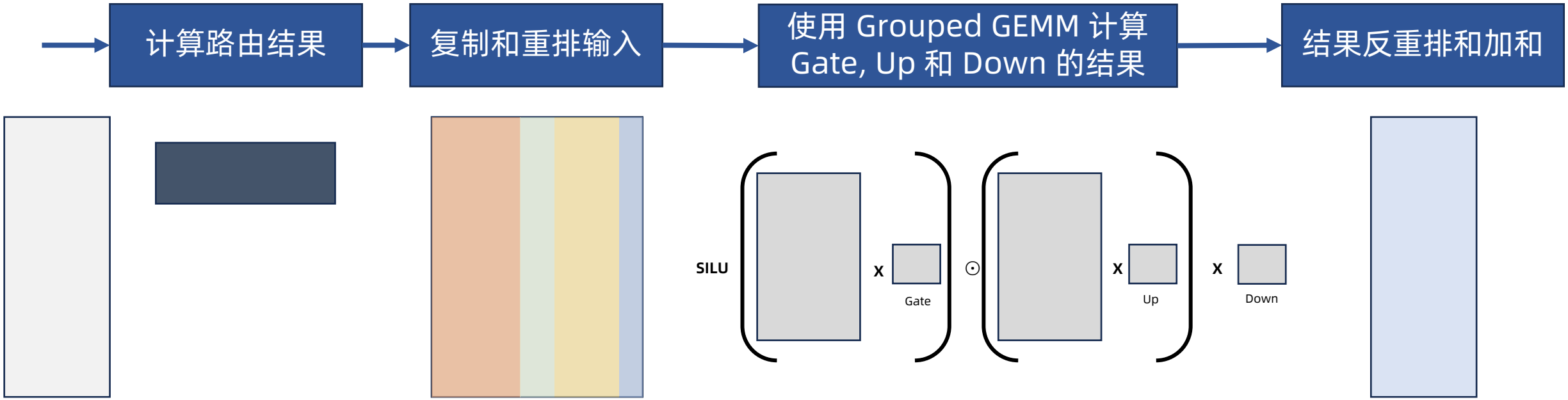


将多专家推理转化成 Grouped GEMM

Grouped GEMM: 将一组独立的 GEMM 计算使用单一 GPU kernel 完成，从而降低 kernel 启动开销



MoE 的高效计算流程



使用的 Kernel

FP16 GEMM Kernel	计算路由权重
TopK Kernel	专家选择和复制重排
FP16 Grouped GEMM Kernel	计算专家输出
Reduce Kernel	计算结果

MoE 的 Kernel 实现

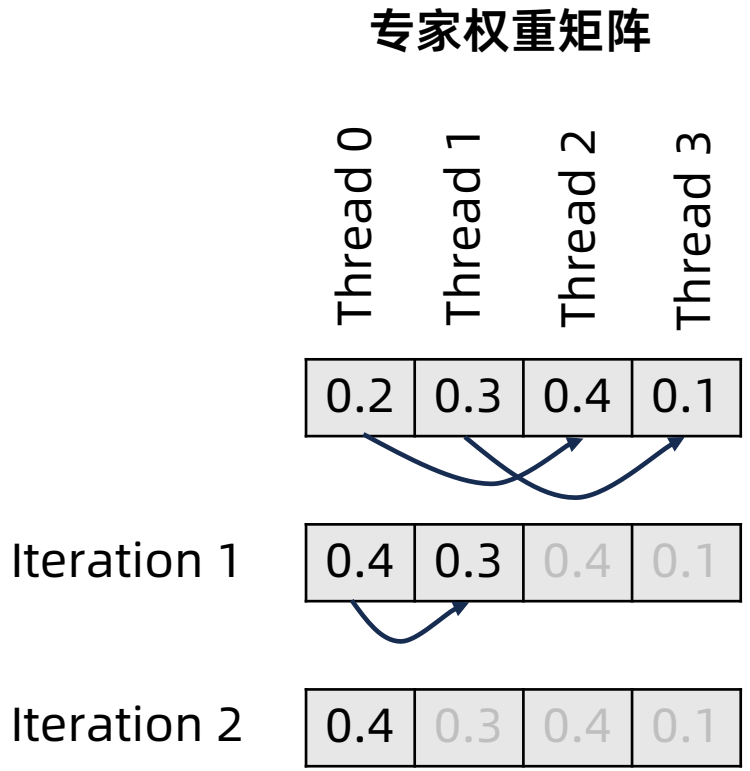
TopK Kernel	专家选择和复制重排

Q: 如何最快找出 Top-K ?

A: 通过蝶形规约多轮迭代和比较，最终得到最大值

使用 HIP 提供的 **__shfl_xor_sync()** 函数和对面线程通信，
获取两个 slots 中的最大值并写入自己的 slot

时间复杂度 O(logn)



MoE 的 Kernel 实现

FP16 Grouped GEMM Kernel | 计算专家输出

rocBLAS 提供 Grouped GEMM 接口，能够通过一次 GPU Kernel Launch 完成多个不同形状的 GEMM 计算

```
for (int i = 0; i < n_expert; i++) {
    m[i] = h_IDX[i + 1] - h_IDX[i];
    inputs[i].setB(d_A + h_IDX[i] * K);
    if constexpr (CONT_LAYOUT) {
        inputs[i].setA(d_B + i * N * K);
    } else {
        inputs[i].setA(expert_weights.ptr[i]);
    }
    inputs[i].setC(d_C + h_IDX[i] * N);
    inputs[i].setD(d_C + h_IDX[i] * N);
    inputs[i].setAlpha(gemm_thirdparty.d_alpha);
    inputs[i].setBeta(gemm_thirdparty.d_beta);
    epilogue[i].setMode(HIPBLASLT_EPILOGUE_DEFAULT);
}
```

设置每组 GEMM 的输入尺寸和矩阵数据指针

设置输出的矩阵数据指针

MoE 的 Kernel 实现

FP16 GEMM Kernel	计算路由权重

- 使用 PyTorch (实际调用 rocBLAS) 完成计算
- 可以和 Shared Expert 的计算并行

Reduce Kernel	计算结果

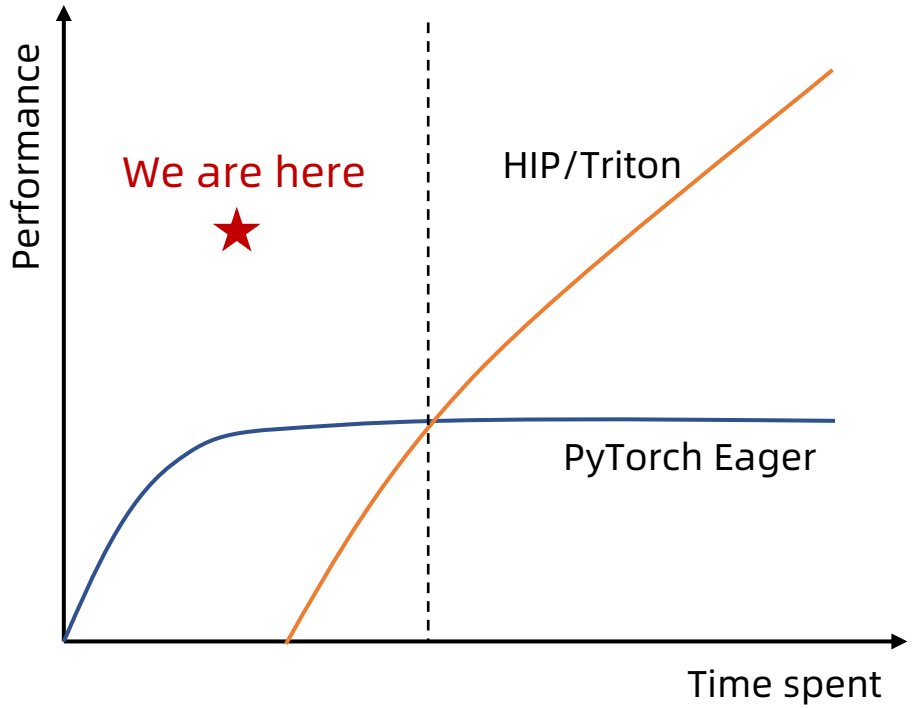
- 通过之前调用 hipCUB 库得到的 index 和第一步的路由权重结果计算输出
- 一组 thread 负责单个 token 的结果累加

更多细节请参考我们的代码和技术报告（末尾将放出二维码）

Part 3: 优化 PyTorch Eager MLA 算子

Why PyTorch Eager Instead of HIP/Triton?

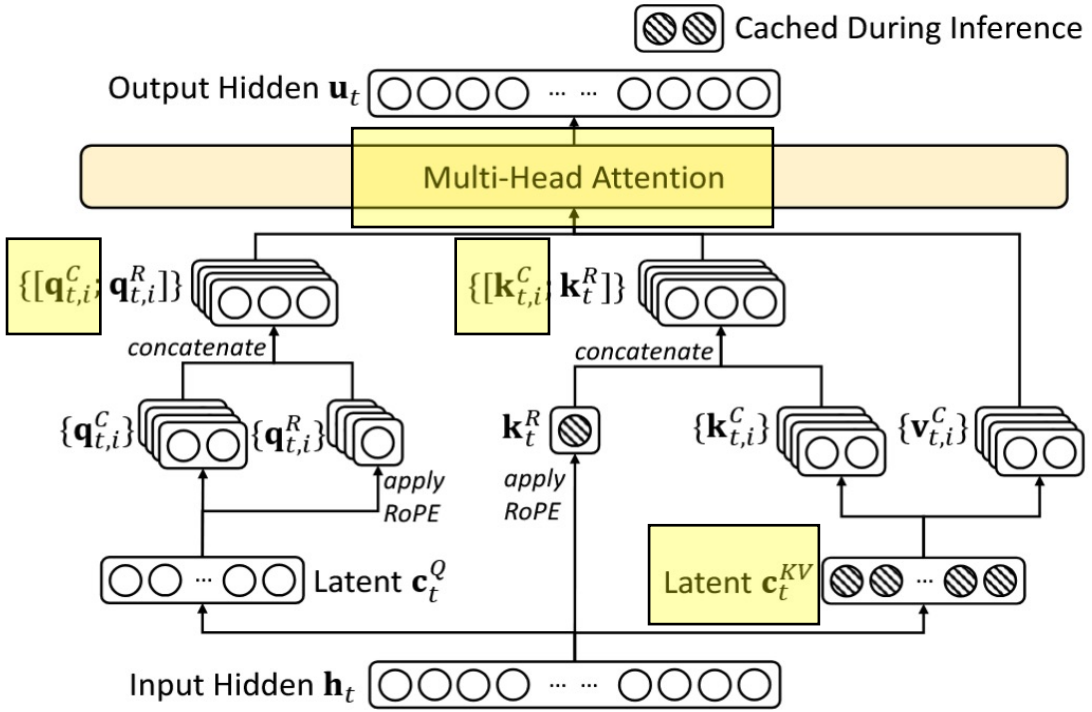
- **节省时间，快速开发** ~~**（DDL快到了）**~~
  - **基于 rocBLAS 的 PyTorch 的 GEMM 性能很高（汇编仙人）**
  - **实现 FlashAttention 需要先优化 GEMM 性能（接近 rocBLAS）**



算法优化

- **利用结合律调整** attention score **的计算顺序，降低计算量**
- $head_i$ **的计算量为：**

$$FLOPs(q^C k^C) = q^C \cdot \left(W_i^{UK} \cdot c^{KV}\right) \rightarrow \left(q^C \cdot W_i^{UK}\right) \cdot c^{KV}$$



算法优化

- **利用结合律调整** attention score **的计算顺序，降低计算量**
- $head_i$ **的计算量为：**

$$FLOPs(q^C k^C) = q^C \cdot \left(W_i^{UK} \cdot c^{KV}\right) \rightarrow \left(q^C \cdot W_i^{UK}\right) \cdot c^{KV}$$

- **加速比（**$\mathrm{s} = 6144$**）：**

$$\frac{2bd_c d_h s + 2bd_h s}{2bd_h d_c + 2bd_c s} = \frac{d_c d_h s + d_h s}{d_h d_c + d_c s} = \frac{513\ s}{512 + 4\ s} = 125.6$$

Free Lunch from `torch.compile`

- **利用 JIT 实现 Element-wise 算子融合**
- **利用 HIP Graph 减少 CPU launch kernel 开销**

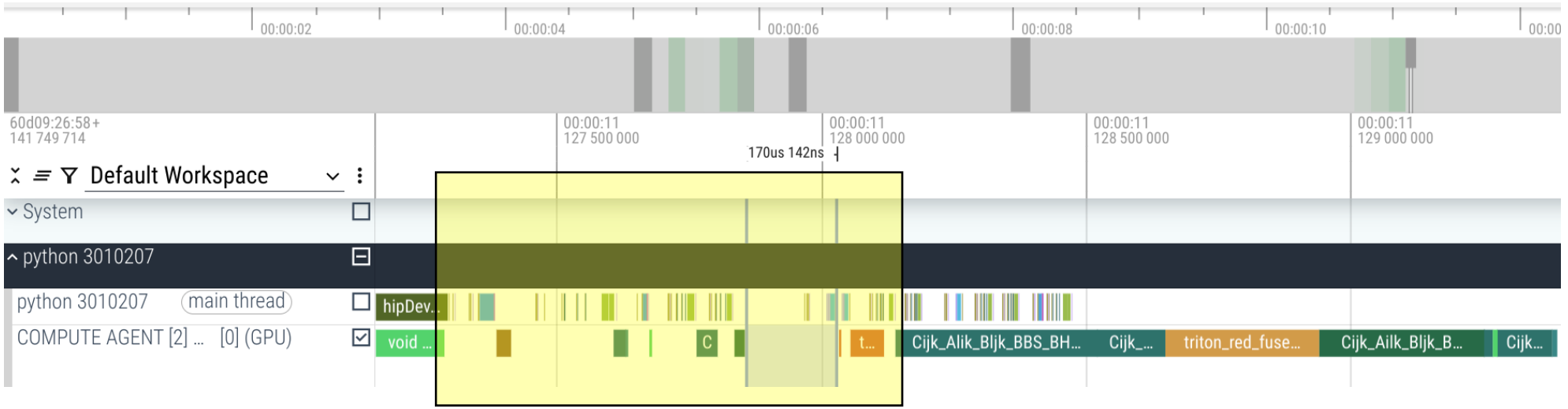
```
11      def rotate_half(x):
12          x1 = x[..., : x.shape[-1] // 2]
13          x2 = x[..., x.shape[-1] // 2 :]
14          return torch.cat((-x2, x1), dim=-1)
15
16      @torch.compile
17      def apply_rotary_pos_emb(q, k, cos, sin, seq_len):
18          q_embed = (q * cos[seq_len - 1]) + (rotate_half(q) * sin[seq_len - 1])
19          k_embed = (k * cos[:seq_len]) + (rotate_half(k) * sin[:seq_len])
20          return q_embed, k_embed
```

Analyze GPU Kernel Trace

- 在计算图的层次上进一步寻找性能优化点

```
rocprofv3  --hip-trace --output-format pftrace -- python mla.py
```

- 分析结果[1]：存在 CPU launch kernel 开销导致的 GPU bubble



[1] https://ui.perfetto.dev/

减少 CPU Launch Kernel 开销

- Solution #1：HIP Graph or `torch.compile(mode='reduce-overhead')`

```
97  def custom_kernel_graph(data: input_t) -> output_t:
98      global output_static, input_static
99      _, _, kv_cache_wrapper = data
100     prefill = kv_cache_wrapper.seq_len
101     if not input_static:
102         with torch.cuda.graph(g):
103             output_static = custom_kernel_impl(data)
104         input_static = data
105     copy_input(input_static, data)
106     g.replay()
```

Initial capture

Replay with new input data

减少 CPU Launch Kernel 开销

- Solution #1：HIP Graph or `torch.compile(mode='reduce-overhead')`

```
97  def custom_kernel_graph(data: input_t) -> output_t:
98      global output_static, input_static
99      _, _, kv_cache_wrapper = data
100     prefill = kv_cache_wrapper.seq_len
101     if not input_static:
102         with torch.cuda.graph(g):
103             output_static = custom_kernel_impl(data)
104         input_static = data
105     copy_input(input_static, data)
106     g.replay()
```

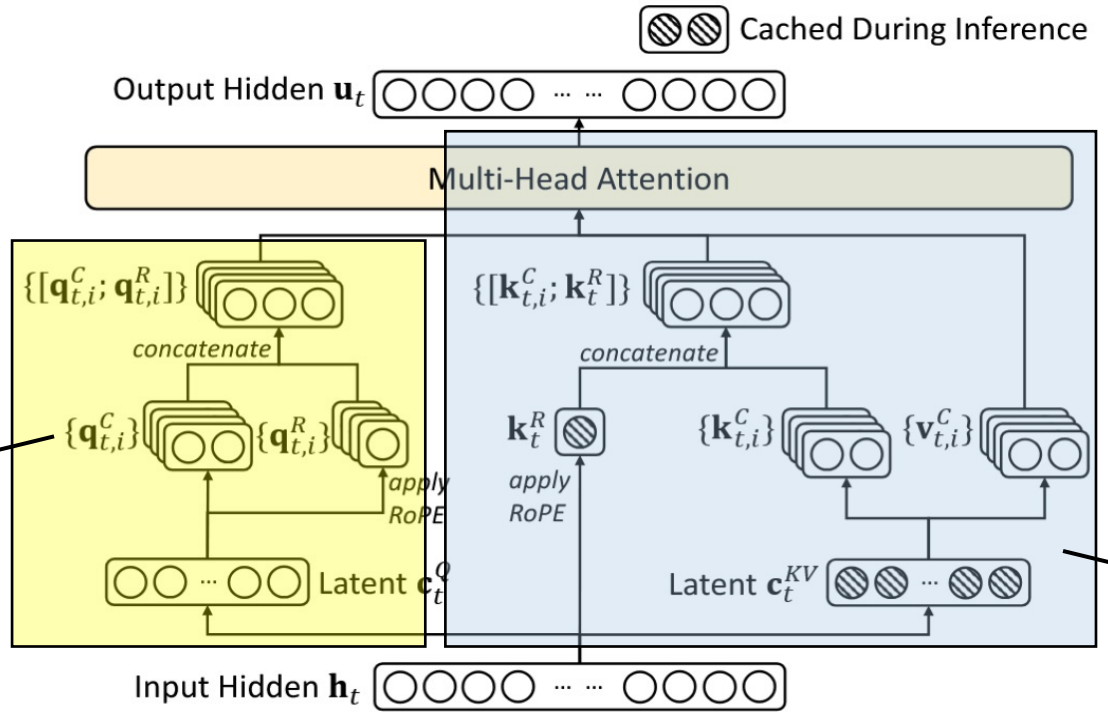
问题：KV $ copy overhead !!

减少 CPU Launch Kernel 开销

- Solution #2：Create Subgraph

Step #1:

- Lightweight kernels
- No KV $ required



Step #2:

- GPU intensive kernels
- Requires KV $

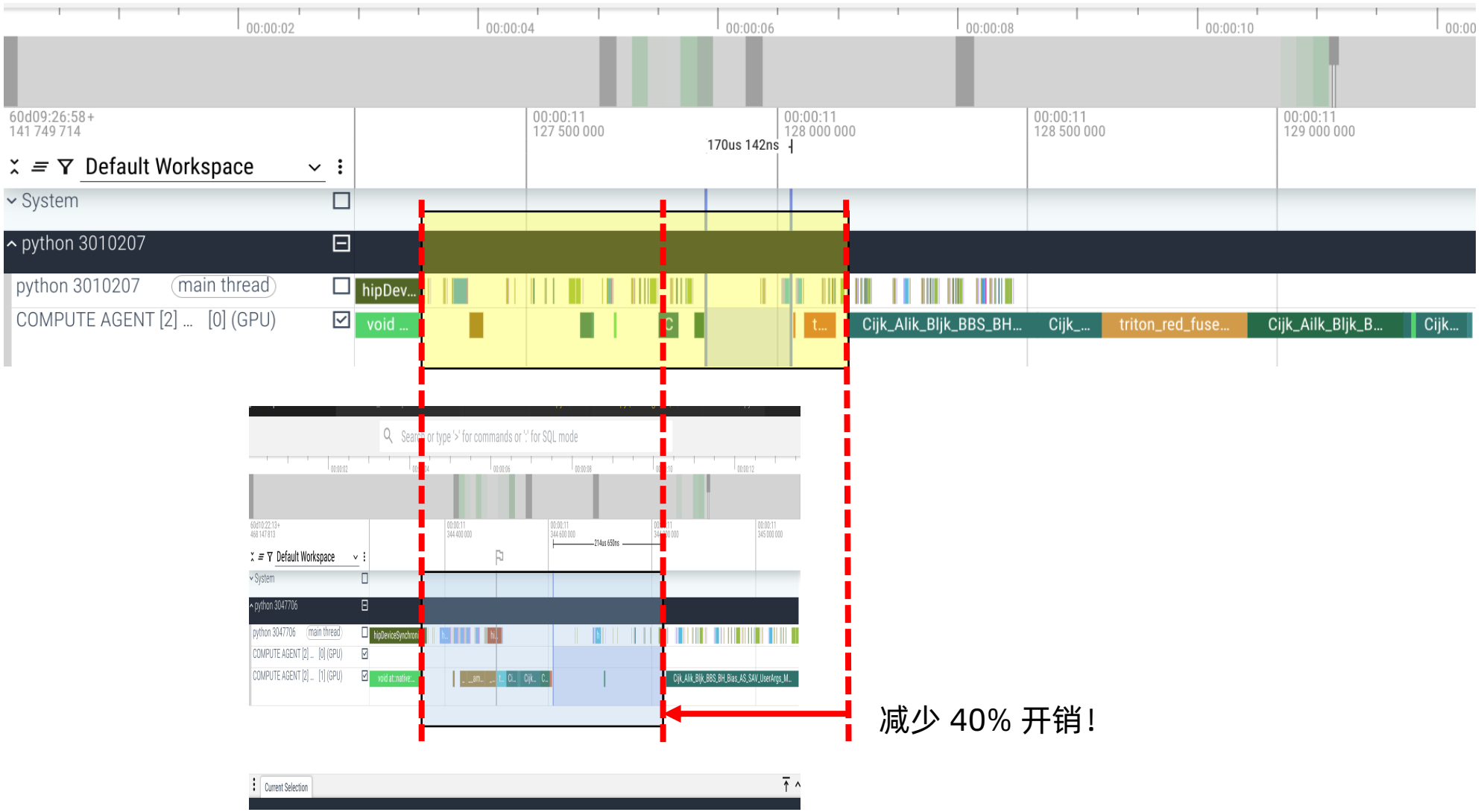
减少 CPU Launch Kernel 开销

- Solution #2：Create Subgraph

```
107  def custom_kernel(data: input_t) -> output_t:
108      global output_static, input_static
109      _, _, kv_cache_wrapper = data
110      prefill = kv_cache_wrapper.seq_len
111      if not input_static:
112          with torch.cuda.graph(g):
113              output_static = custom_kernel_step_1(data)
114          input_static = data
115      copy_input(data)
116      g.replay()
117      return custom_kernel_step_2(data, *output_static)
```

减少 CPU Launch Kernel 开销

- 效果



Thanks!

RadeonFlow Kernels Github

（包含技术报告）

欢迎 Star 🌟

AMD 开发者云

（可提供MI300X GPU实例）

欢迎在上面运行我们的项目