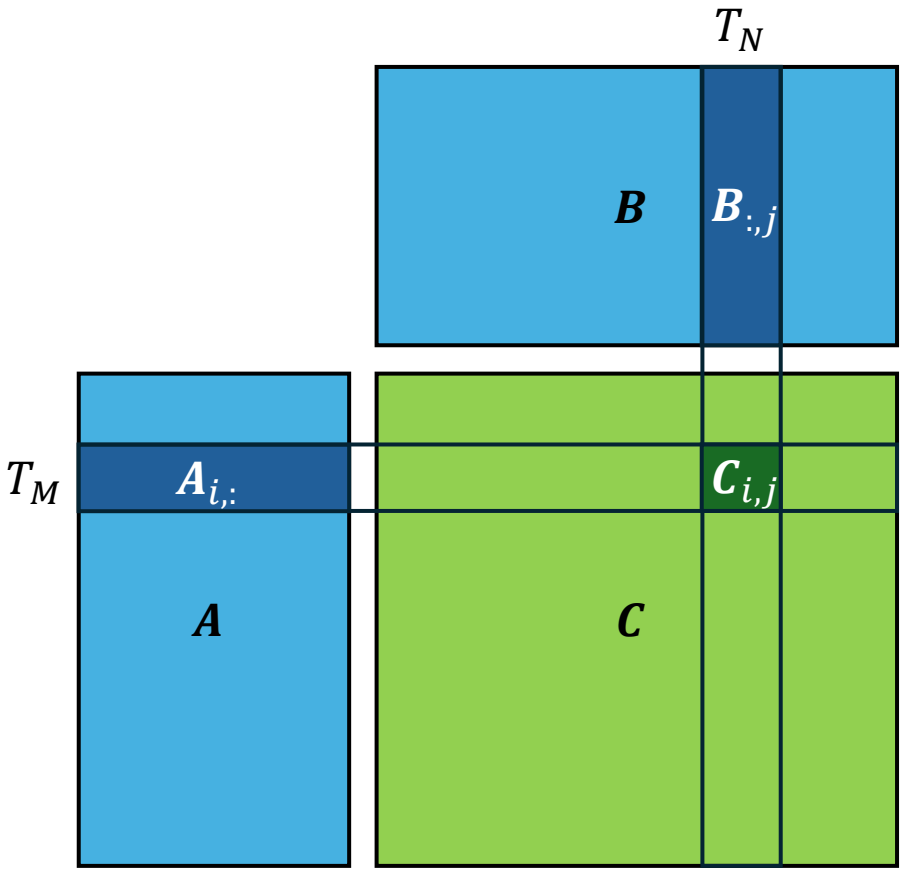
MSRA SH Triton Study Group
3. Triton GeMM Kernel

2025/07/18

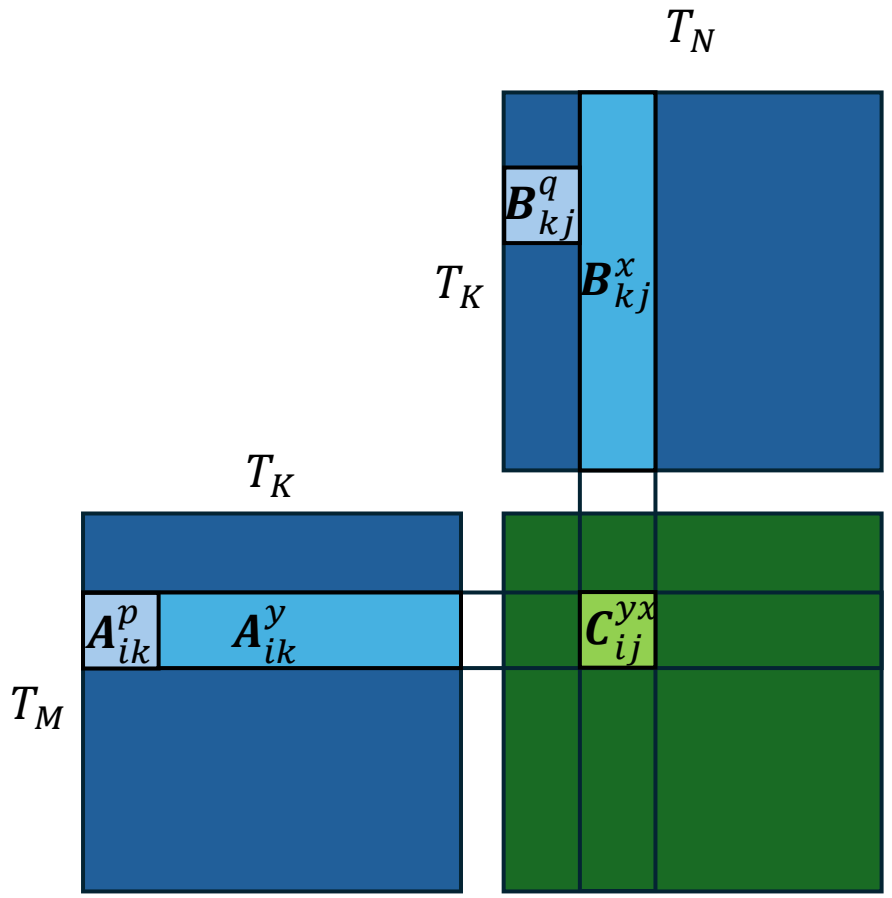
General Matrix Multiplication on GPU

- $\boldsymbol{A} \in \mathbb{R}_{\text{float16}}^{M \times K}; \boldsymbol{B} \in \mathbb{R}_{\text{float16}}^{K \times N}$
- Block size T_M, T_N, T_K
- Tile output $\boldsymbol{C}$ into $\frac{M}{T_M} \frac{N}{T_N}$ blocks $\boldsymbol{C}_{i,j}$
- $\boldsymbol{C}_{i,j} = \boldsymbol{A}_{i,:}\boldsymbol{B}_{:,j} = \sum_{k=1}^{K} \boldsymbol{A}_{i,k}\boldsymbol{B}_{k,j}$
  - Initialize $\boldsymbol{C}_{i,j} \leftarrow 0_{\text{float32}}^{T_M \times T_N}$
  - For $k \leftarrow 0$ to $\frac{K}{T_K}$:
    - Load $\boldsymbol{A}_{ik}$ and $\boldsymbol{B}_{kj}$
    - Calculate $\boldsymbol{C}_{i,j} \leftarrow \boldsymbol{C}_{i,j} + \boldsymbol{A}_{i,k}\boldsymbol{B}_{k,j}$
  - Store $\boldsymbol{C}_{i,j}$

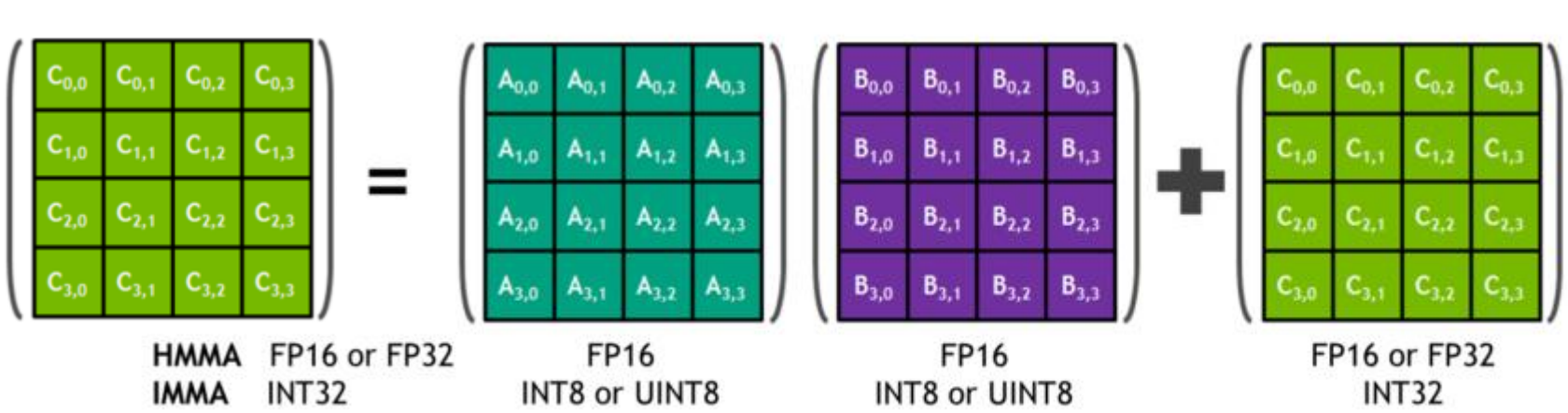


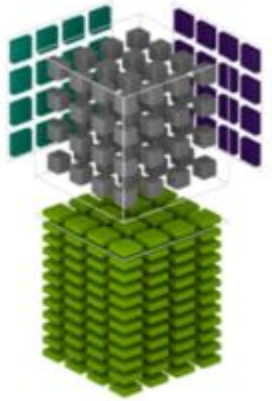
General Matrix Multiplication on GPU

- For each thread t :
  - $y, x \leftarrow \text{partition\_C}(T_M, T_N, 128, t)$
  - $p \leftarrow \text{partition\_A}(T_M T_K, 128, t)$
  - $q \leftarrow \text{partition\_B}(T_K T_N, 128, t)$
  - Initialize $\boldsymbol{C}_{ij}^{yx} \leftarrow 0$ in **registers**
  - For $k \leftarrow 0$ to $\frac{K}{T_K}$:
    - Load $\boldsymbol{A}_{ik}^{p}$ and $\boldsymbol{B}_{kj}^{q}$ from **global** to **shared**
    - Load $\boldsymbol{A}_{ik}^{y}$ and $\boldsymbol{B}_{kj}^{x}$ from **shared** to **registers**
    - Calculate $\boldsymbol{C}_{ij}^{y,x} \leftarrow \boldsymbol{C}_{ij}^{y,x} + \boldsymbol{A}_{ik}^{y}\boldsymbol{B}_{kj}^{x}$
  - Store $\boldsymbol{C}_{ij}^{yx}$ to **global**



Tensor Core





- Warp level parallel: $\boldsymbol{C}_{ij}^{y,x} \leftarrow \boldsymbol{A}_{ik}^{y}\boldsymbol{B}_{kj}^{x} + \boldsymbol{C}_{ij}^{y,x}$
- All threads execute `wmma` instruction simultaneously

Triton GeMM: Kernel Signature

```
@triton.jit
def matmul_kernel(
    a_ptr, b_ptr, c_ptr,  # Pointers to input and output matrices
    M, N, K,  # Matrix dimensions
    stride_am, stride_ak,  # Strides for A matrix
    stride_bk, stride_bn,  # Strides for B matrix
    stride_cm, stride_cn,  # Strides for C matrix
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,  # Block sizes
):
```

Triton GeMM: Offsets

```
# Thread block index
pid_n = tl.program_id(axis=0)
pid_m = tl.program_id(axis=1)
start_m = pid_m * BLOCK_SIZE_M
start_n = pid_n * BLOCK_SIZE_N

# Create pointers for the first blocks of A and B
offs_m = start_m + tl.arange(0, BLOCK_SIZE_M)
offs_n = start_n + tl.arange(0, BLOCK_SIZE_N)
offs_k = tl.arange(0, BLOCK_SIZE_K)
a_ptrs = a_ptr + offs_m[:, None] * stride_am + offs_k[None, :] * stride_ak
b_ptrs = b_ptr + offs_k[:, None] * stride_bk + offs_n[None, :] * stride_bn
```

Triton GeMM: Main Loop

```
# Iterate to compute a block of the C matrix
accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
    a = tl.load(a_ptrs)  # Load a block of A to shared memory
    b = tl.load(b_ptrs)  # Load a block of B to shared memory
    accumulator = tl.dot(a, b, accumulator)  # Call tensor core
    a_ptrs += BLOCK_SIZE_K * stride_ak  # Move to the next block of A
    b_ptrs += BLOCK_SIZE_K * stride_bk  # Move to the next block of B
c = accumulator.to(c_ptr.type.element_ty)  # Convert to the output dtype

# Write back the block of the output matrix C
c_ptrs = c_ptr + offs_m[:, None] * stride_cm + offs_n[None, :] * stride_cn
tl.store(c_ptrs, c)
```

Triton GeMM: Kernel Launch

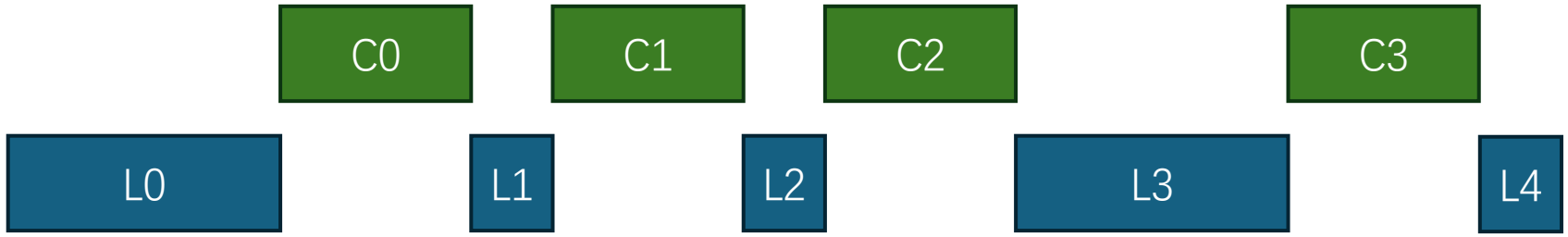
```python
def triton_matmul(a: torch.Tensor, b: torch.Tensor) -> torch.Tensor:
    M, K = a.shape
    K2, N = b.shape
    assert K == K2, "Inner dimensions must match for matrix multiplication"
    c = torch.empty((M, N), dtype=a.dtype, device=a.device)
    grid = lambda META: (triton.cdiv(N, META['BLOCK_SIZE_N']), triton.cdiv(M, META['BLOCK_SIZE_M']), )
    matmul_kernel[grid](
        a, b, c, M, N, K,
        a.stride(0), a.stride(1),
        b.stride(0), b.stride(1),
        c.stride(0), c.stride(1),
    )
    return c
```

Triton Auto-tune

```
@triton.autotune(
    configs=[
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 256, 'BLOCK_SIZE_K': 64}, num_stages=3, num_warps=8),
        triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 256, 'BLOCK_SIZE_K': 32}, num_stages=4, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 32}, num_stages=4, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 64, 'BLOCK_SIZE_K': 32}, num_stages=4, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 32}, num_stages=4, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 32, 'BLOCK_SIZE_K': 32}, num_stages=4, num_warps=4),
    ],
    key=['M', 'N', 'K'],
)
```

Async-load and Multi-stage

Synchronized



Asynchronized

Multi-Stage

Triton GeMM: Benchmark

- Not so fast

```
Problem size: (4096, 4096, 4096)
Torch GeMM Latency: 0.606 ms
Triton GeMM Latency: 0.771 ms
```

- Why? Will be discussed in the next class

Homework

- Play with TritonStudyGroup/3_Triton_GeMM at main · Starmys/TritonStudyGroup

- Questions:
  - Implement a Triton linear kernel $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}^{\boldsymbol{T}}$
  - How to estimate `num_stages` under a given problem shape?

Reading Materials

- Matrix Multiplication — Triton documentation
- Warp Matrix Functions— CUDA C++ Programming Guide