Weiming HPC Training Camp
×
LCPU AI Infra Seminars

北京大学未名超算队

北京大学学生 Linux 俱乐部

Workshop 01.
Parallel Programming with TileLang

孔昊然

2026. 08. 01

Weiming Supercomputing Team
Linux Club of Peking University

Linux俱乐部项目

Recall：Slides from Session 1

CUDA

```
__global__ void matmul(half* A, half* B, half* C,
                       int M, int N, int K) {
    int tx = threadIdx.x, ty = threadIdx.y;
    int row = blockIdx.y * BM + ty;
    int col = blockIdx.x * BN + tx;
    __shared__ half As[BM][BK], Bs[BK][BN];
    float acc = 0.0f;
    for (int k0 = 0; k0 < K; k0 += BK) {
        if (row < M && k0 + tx < K)
            As[ty][tx] = A[row * K + k0 + tx];
        if (k0 + ty < K && col < N)
            Bs[ty][tx] = B[(k0 + ty) * N + col];
        __syncthreads();
        #pragma unroll
        for (int k = 0; k < BK; ++k)
            acc += As[ty][k] * Bs[k][tx];
        __syncthreads();
    }
    if (row < M && col < N)
        C[row * N + col] = acc;
}
```

TileLang

```
with T.Kernel(...) as (bx, by):
    As = T.alloc_shared((BM, BK), "float16")
    Bs = T.alloc_shared((BK, BN), "float16")
    C_local = T.alloc_fragment((BM, BN), "float32")
    T.clear(C_local)
    for ko in T.Pipelined(T.ceildiv(K, BK), num_stages=3):
        T.copy(A[...], As)
        T.copy(B[...], Bs)
        T.gemm(As, Bs, C_local)
    T.copy(C_local, C[...])
```

CUDA 展开线程级细节
TileLang 保留 Tile 级数据流与计算结构

Recall：TileLang Primitives

定义与启动

- `@tilelang.jit`：生成并编译可调用 Kernel
- `@T.prim_func`：定义 TileLang/TIR 函数
- `T.Kernel(...)`：创建 Grid，一个实例对应一个 CTA

内存层次

- 函数参数 `T.Tensor`：Global Memory
- `T.alloc_shared`：CTA 共享的输入 Tile
- `T.alloc_fragment`：寄存器中的计算结果

数据流与计算

```
Global Memory ---> Shared Memory ---> Register Fragment ---> Global Memory
              T.copy                 T.gemm                  T.copy
```

- `T.Pipelined`：沿 K 维分块，并重叠搬运与计算
- `T.copy`：表达不同内存层次之间的数据搬运
- `T.gemm`：表达 Tile 级矩阵乘加

Recall：TileLang MatMal

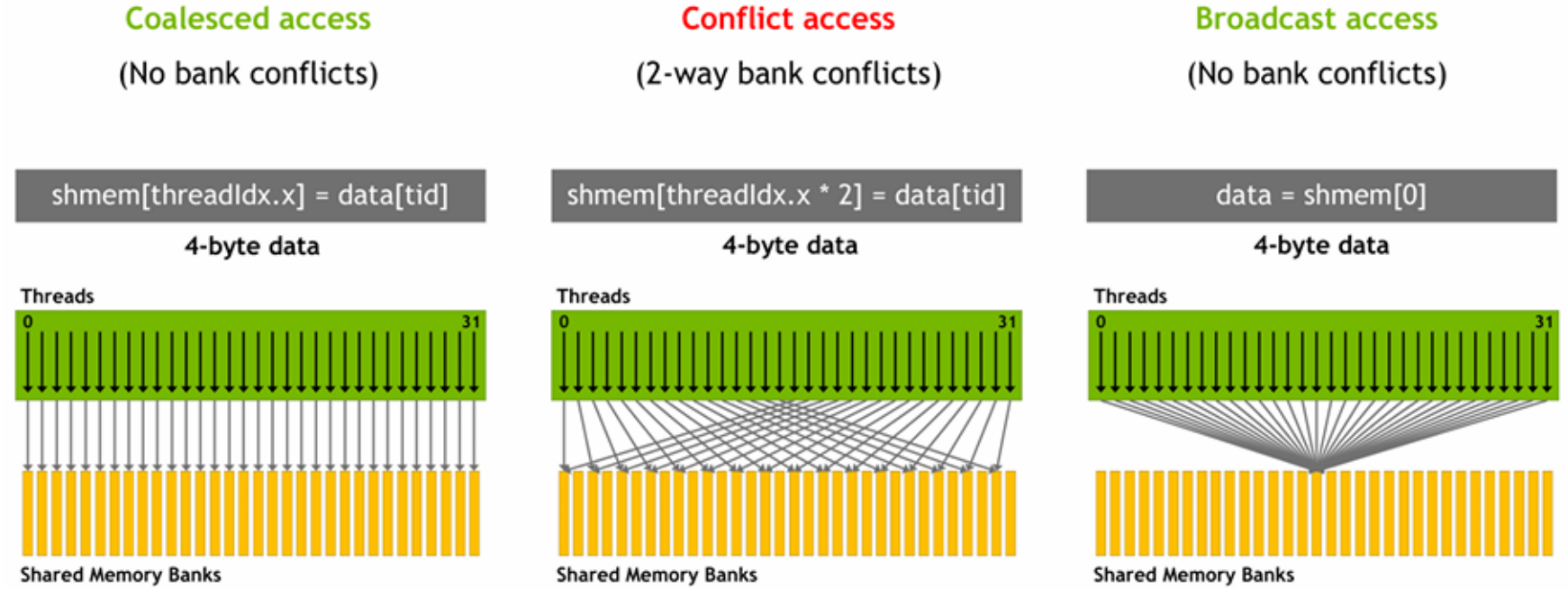
TileLang 的核心结构：

- 显式分配 shared memory tile
  A_shared、B_shared 存储输入数据块
- 分配 register fragment 作为累加器
  C_local 累积中间结果
- K 维分块循环，支持 pipeline
  每轮从 global 搬运数据到 shared
  调用 gemm 更新 C_local
- 循环结束后写回 global memory

强调数据搬运和层次化计算

```python
@tilelang.jit
def matmul(
    M: int,
    N: int,
    K: int,
    BLOCK_M: int,
    BLOCK_N: int,
    BLOCK_K: int,
    dtype: str = "float16",
    accum_dtype: str = "float32",
):
    @T.prim_func
    def main(
        A: T.Buffer((M, K), dtype),
        B: T.Buffer((K, N), dtype),
        C: T.Buffer((M, N), accum_dtype),
    ):
        with T.Kernel(
            T.ceildiv(N, BLOCK_N),
            T.ceildiv(M, BLOCK_M),
            threads=128,
        ) as (bx, by):
            A_shared = T.alloc_shared((BLOCK_M, BLOCK_K), dtype)
            B_shared = T.alloc_shared((BLOCK_K, BLOCK_N), dtype)
            C_local = T.alloc_fragment((BLOCK_M, BLOCK_N), accum_dtype)

            T.clear(C_local)

            for k in T.Pipelined(T.ceildiv(K, BLOCK_K), num_stages=3):
                T.copy(A[by * BLOCK_M, k * BLOCK_K], A_shared)
                T.copy(B[k * BLOCK_K, bx * BLOCK_N], B_shared)
                T.gemm(A_shared, B_shared, C_local)

            T.copy(C_local, C[by * BLOCK_M, bx * BLOCK_N])

    return main
```

Recall：Bank Conflict



Shared Memory 被划分为 32 个 Bank，每个 Bank 可以看作一个独立的小型 SRAM。由于 32 个 Bank 可以并行工作，一个 Warp 中的 32 个线程可以同时访问不同 Bank，但单个 Bank 的硬件访问能力无法在同一周期同时响应多个不同地址的请求。因此，当多个线程访问同一个 Bank 的不同地址时，会发生资源竞争导致访问被串行化。

Multicast 行为解释：同一 warp 中，如16 个线程读 shared[0]，另 16 个线程读 shared[1]。每组内部都是相同地址访问，因此各自形成 broadcast，两个 broadcast 被合并为一次 multicast。整个请求不产生 bank conflict。

2025.11.7 NV OpenDay 杭州的CuTeDSL Slides

TileLang

TileLang as a native code generator

- Tilelang provides hardware independent TIR
- Tilelang can explicit control:
  - Memory allocation
  - Memory movement
- Tilelang provides **indexing** in tiles

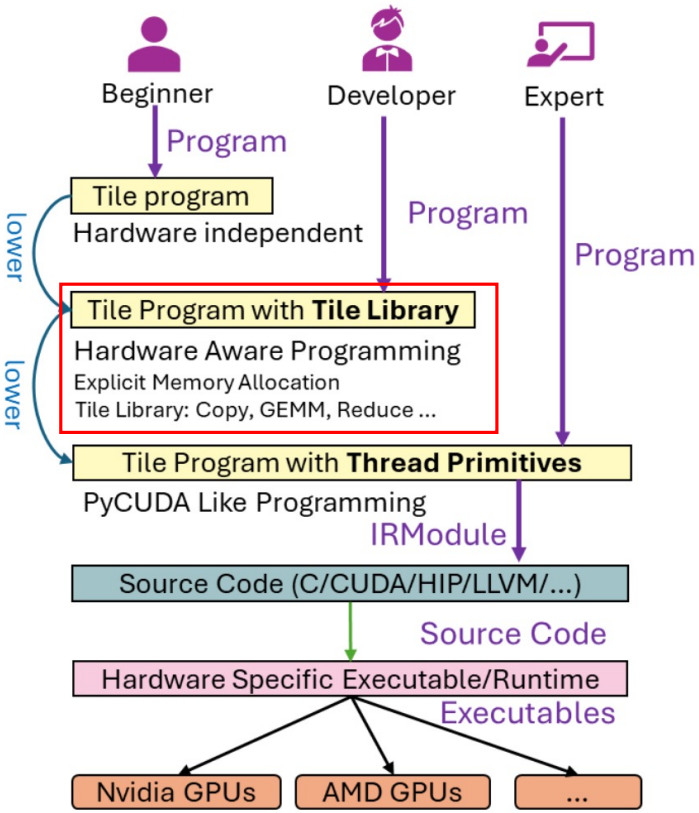
```
A_shared = T.alloc_shared((M,K),dtype)

T.copy(A_shared, A_local)
```

```
for i in T.serial(128):
    for j in T.serial(128):
        B[i, j] = A[i, j]
```

- Tilelang supports **code injection**
  - C++
  - PTX

```
T.ptx(
    "mma.m16n8k32.row.col.s32.s8.s8.s32",
    T.address_of(A), A_offset,
    T.address_of(B), B_offset,
    T.address_of(C), C_offset,
)
```

- Tilelang outputs native codes
  - CUDA C++ on Nvidia GPUs: Volta, Ampere, Hopper, ...
  - HIP C++ for AMD GPUs: MI300, MI250, ...
  - C++/llvm for CPUs
  - Native codes for other Accelerators.



2025.11.7 NV OpenDay 杭州的CuTeDSL Slides

TileLang

TileLang Lowing to TIR

- Tilelang code

```python
@T.prim_func
def main(
        A: T.Tensor((M, K), in_dtype),
        B: T.Tensor((N, K), in_dtype),
        C: T.Tensor((M, N), out_dtype),
):
    with T.Kernel(T.ceildiv(N, bN), T.ceildiv(M, bM), threads=128) as (bx, by):
        A_shared = T.alloc_shared((bM, bK), in_dtype)
        B_shared = T.alloc_shared((bN, bK), in_dtype)
        C_local = T.alloc_fragment((bM, bN), accum_dtype)

        T.clear(C_local)
        for k in T.Pipelined(T.ceildiv(K, bK), num_stages=3):
            T.copy(A[by * bM, k * bK], A_shared)
            T.copy(B[bx * bN, k * bK], B_shared)
            T.gemm(A_shared, B_shared, C_local, transpose_B=True)

        T.copy(C_local, C[by * bM, bx * bN])

return main
```

- Tensor IR (TIR) code

```python
@T.prim_func
def main_kernel(A: T.handle("float8_e4m3", "global"), B: T.handle("float8_e4m3", "global"), C: T.handle
("float32", "global")):
    T.func_attr({"calling_conv": 2, "dyn_shared_memory_buf": 49152, "target": T.target({"arch": "sm_80",
    "keys": ["cutedsl", "cuda", "gpu"], "kind": "cuda", "max_num_threads": 1024, "tag": "",
    "thread_warp_size": 32}), "thread_extent": {"blockIdx.x": 8, "blockIdx.y": 8, "threadIdx.x": 128,
    "threadIdx.y": 1, "threadIdx.z": 1}, "tir.is_global_func": T.bool(True), "tir.kernel_launch_params":
    ["blockIdx.x", "blockIdx.y", "threadIdx.x", "threadIdx.y", "threadIdx.z", "tir.use_dyn_shared_memory"],
    "tir.noalias": True})
    C_1 = T.decl_buffer((1048576,), data=C)
    B_1 = T.decl_buffer((1048576,), "float8_e4m3", data=B)
    buf_dyn_shmem = T.handle("uint8", "shared.dyn")
    B_shared = T.decl_buffer((24576,), "float8_e4m3", data=buf_dyn_shmem, scope="shared.dyn")
    A_1 = T.decl_buffer((1048576,), "float8_e4m3", data=A)
    A_shared = T.decl_buffer((24576,), "float8_e4m3", data=buf_dyn_shmem, scope="shared.dyn")
    C_local = T.handle("float32", "local")
    C_local_1 = T.decl_buffer((128,), data=C_local, scope="local")
    bx = T.launch_thread("blockIdx.x", 8)
    buf_dyn_shmem = T.allocate([49152], "uint8", "shared.dyn")
    C_local = T.allocate([128], "float32", "local")
    by = T.launch_thread("blockIdx.y", 8)
    tx = T.launch_thread("threadIdx.x", 128)
    ty = T.launch_thread("threadIdx.y", 1)
    tz = T.launch_thread("threadIdx.z", 1)
```

Tilelang High level Primitives

T.alloc_shared
T.alloc_fragment
T.clear
T.Pipelined
T.copy
T.gemm
...

tilelang.compile →

TIR

T.Vectorized
T.unroll
T.broadcast
T.call_extern
T.sync_thread
T.async_wait
T.address_of
T.atomic_add
...

2025.11.7 NV OpenDay 杭州的CuTeDSL Slides

TileLang

TileLang Lowing to TIR

- Tilelang code

```python
@T.prim_func
def main(
        A: T.Tensor((M, K), in_dtype),
        B: T.Tensor((N, K), in_dtype),
        C: T.Tensor((M, N), out_dtype),
):
    with T.Kernel(T.ceildiv(N, bN), T.ceildiv(M, bM), threads=128) as (bx, by):
        A_shared = T.alloc_shared((bM, bK), in_dtype)
        B_shared = T.alloc_shared((bN, bK), in_dtype)
        C_local = T.alloc_fragment((bM, bN), accum_dtype)

        T.clear(C_local)
        for k in T.Pipelined(T.ceildiv(K, bK), num_stages=3):
            T.copy(A[by * bM, k * bK], A_shared)
            T.copy(B[bx * bN, k * bK], B_shared)
            T.gemm(A_shared, B_shared, C_local, transpose_B=True)

        T.copy(C_local, C[by * bM, bx * bN])

return main
```

Automatically select appropriate
Pipeline
CopyAtom
MmaAtom
for different Arch

- TIR code for sm_80

```
for i in T.unroll(64):
    C_local_1[i * 2:i * 2 + 2] = T.Broadcast(T.float32(0.0), 2)
for i in T.unroll(4):
    T.ptx_cp_async("float8_e4m3", buf_dyn_shmem, i * 2048 + T.shift_right(tx, 2) * 64 + T.bitwise_and(T.sh
with T.attr(0, "async_commit_queue_scope", 0):
    for i in T.unroll(4):
        T.ptx_cp_async("float8_e4m3", buf_dyn_shmem, i * 2048 + T.shift_right(tx, 2) * 64 + T.bitwise_and(
for i in T.unroll(4):
    T.ptx_cp_async("float8_e4m3", buf_dyn_shmem, i * 2048 + T.shift_right(tx, 2) * 64 + T.bitwise_and(T.sh
with T.attr(0, "async_commit_queue_scope", 0):
    for i in T.unroll(4):
        T.ptx_cp_async("float8_e4m3", buf_dyn_shmem, i * 2048 + T.shift_right(tx, 2) * 64 + T.bitwise_and(
for k in range(14):
    T.tvm_storage_sync("shared.dyn")
    for i in T.unroll(4):
        T.ptx_cp_async("float8_e4m3", buf_dyn_shmem, T.truncmod(k + 2, 3) * 8192 + i * 2048 + T.shift_righ
    with T.attr(0, "async_commit_queue_scope", 0):
        for i in T.unroll(4):
            T.ptx_cp_async("float8_e4m3", buf_dyn_shmem, T.truncmod(k + 2, 3) * 8192 + i * 2048 + T.shift_
    T.attr(0, "async_wait_queue_scope", 0)
    T.attr(0, "async_wait_inflight_count", 2)
    T.tvm_storage_sync("shared")
    buf_dyn_shmem_1 = T.Buffer((k % 3 * 8192 + 1,), "float8_e4m3", data=buf_dyn_shmem, scope="shared.dyn")
    buf_dyn_shmem_2 = T.Buffer((24576 + k % 3 * 8192 + 1,), "float8_e4m3", data=buf_dyn_shmem, scope="shar
    C_local_2 = T.Buffer((1,), data=C_local, scope="local")
    T.tl_gemm("tl::gemm_ss<128, 128, 64, 2, 2, 0, 1, 0, 64, 64, 0, 0>", T.address_of(buf_dyn_shmem_1[T.tru
with T.attr(0, "async_wait_queue_scope", 0):
    T.attr(0, "async_wait_inflight_count", 1)
    T.tvm_storage_sync("shared")
    buf_dyn_shmem_1 = T.Buffer((16385,), "float8_e4m3", data=buf_dyn_shmem, scope="shared.dyn")
    buf_dyn_shmem_2 = T.Buffer((40961,), "float8_e4m3", data=buf_dyn_shmem, scope="shared.dyn")
    C_local_2 = T.Buffer((1,), data=C_local, scope="local")
    T.tl_gemm("tl::gemm_ss<128, 128, 64, 2, 2, 0, 1, 0, 64, 64, 0, 0>", T.address_of(buf_dyn_shmem_1[16384
with T.attr(0, "async_wait_queue_scope", 0):
    T.attr(0, "async_wait_inflight_count", 0)
    T.tvm_storage_sync("shared")
    buf_dyn_shmem_1 = T.Buffer((1,), "float8_e4m3", data=buf_dyn_shmem, scope="shared.dyn")
    buf_dyn_shmem_2 = T.Buffer((24577,), "float8_e4m3", data=buf_dyn_shmem, scope="shared.dyn")
    C_local_2 = T.Buffer((1,), data=C_local, scope="local")
    T.tl_gemm("tl::gemm_ss<128, 128, 64, 2, 2, 0, 1, 0, 64, 64, 0, 0>", T.address_of(buf_dyn_shmem_1[0]),
for i in T.unroll(64):
    C_1[by * 131072 + T.shift_right(T.bitwise_and(i, 7), 1) * 32768 + T.shift_right(T.bitwise_and(tx, 63)
```

2025.11.7 NV OpenDay 杭州的CuTeDSL Slides

TileLang

TileLang Lowing to TIR

- Tilelang code

```
@T.prim_func
def main(
        A: T.Tensor((M, K), in_dtype),
        B: T.Tensor((N, K), in_dtype),
        C: T.Tensor((M, N), out_dtype),
):
    with T.Kernel(T.ceildiv(N, bN), T.ceildiv(M, bM), threads=128) as (bx, by):
        A_shared = T.alloc_shared((bM, bK), in_dtype)
        B_shared = T.alloc_shared((bN, bK), in_dtype)
        C_local = T.alloc_fragment((bM, bN), accum_dtype)

        T.clear(C_local)
        for k in T.Pipelined(T.ceildiv(K, bK), num_stages=3):
            T.copy(A[by * bM, k * bK], A_shared)
            T.copy(B[bx * bN, k * bK], B_shared)
            T.gemm(A_shared, B_shared, C_local, transpose_B=True)

        T.copy(C_local, C[by * bM, bx * bN])

return main
```

Automatically select appropriate
Pipeline
CopyAtom
MmaAtom
for different Arch

- TIR code for sm_90

```
if T.tl_shuffle_elect(0):
    T.call_extern("handle", "tl::prefetch_tma_descriptor", A_desc)
    T.call_extern("handle", "tl::prefetch_tma_descriptor", B_desc)
    T.ptx_init_barrier_thread_count(T.get_mbarrier(0), 128)
    T.ptx_init_barrier_thread_count(T.get_mbarrier(1), 128)
    T.ptx_init_barrier_thread_count(T.get_mbarrier(2), 128)
    T.ptx_init_barrier_thread_count(T.get_mbarrier(3), 128)
    T.ptx_init_barrier_thread_count(T.get_mbarrier(4), 128)
    T.ptx_init_barrier_thread_count(T.get_mbarrier(5), 128)
T.tvm_storage_sync("shared")
ty = T.launch_thread("threadIdx.y", 1)
tz = T.launch_thread("threadIdx.z", 1)
T.attr([128, 128], "kWarpSpecializationScope", 0)
if 128 <= tx:
    T.set_max_nreg(24, 0)
    for k in range(16):
        T.mbarrier_wait_parity(T.get_mbarrier(T.truncmod(k, 3) + 3), T.bitwise_xor(T.Div(T.truncmod(k, 6),
        if T.tl_shuffle_elect(128):
            T.mbarrier_expect_tx(T.get_mbarrier(T.truncmod(k, 3)), 8192)
            T.fence_proxy_async()
            buf_dyn_shmem_1 = T.Buffer((k % 3 * 8192 + 1,), "float8_e4m3", data=buf_dyn_shmem, scope="share
            T.tma_load(A_desc, T.get_mbarrier(T.truncmod(k, 3)), T.address_of(buf_dyn_shmem_1[T.truncmod(k,
            T.mbarrier_expect_tx(T.get_mbarrier(T.truncmod(k, 3)), 8192)
            T.fence_proxy_async()
            buf_dyn_shmem_2 = T.Buffer((24576 + k % 3 * 8192 + 1,), "float8_e4m3", data=buf_dyn_shmem, scop
            T.tma_load(B_desc, T.get_mbarrier(T.truncmod(k, 3)), T.address_of(buf_dyn_shmem_2[T.truncmod(k,
        T.ptx_arrive_barrier(T.get_mbarrier(T.truncmod(k, 3)))
else:
    T.set_max_nreg(240, 1)
    for i in T.unroll(64):
        C_local_1[i * 2:i * 2 + 2] = T.Broadcast(T.float32(0.0), 2)
    for k in range(16):
        T.mbarrier_wait_parity(T.get_mbarrier(T.truncmod(k, 3)), T.Div(T.truncmod(k, 6), 3))
        T.fence_proxy_async()
        buf_dyn_shmem_1 = T.Buffer((k % 3 * 8192 + 1,), "float8_e4m3", data=buf_dyn_shmem, scope="shared.dy
        buf_dyn_shmem_2 = T.Buffer((24576 + k % 3 * 8192 + 1,), "float8_e4m3", data=buf_dyn_shmem, scope="s
        C_local_2 = T.Buffer((1,), data=C_local, scope="local")
        T.tl_gemm("tl::gemm_ss<128, 128, 64, 4, 1, 0, 1, 0, 64, 64, 0, 0, true>", T.address_of(buf_dyn_shme
        T.ptx_arrive_barrier(T.get_mbarrier(T.truncmod(k, 3) + 3))
    for i in T.unroll(64):
        C_1[by * 131072 + T.shift_right(i, 5) * 65536 + T.shift_right(tx, 5) * 16384 + T.bitwise_and(i, 1)
```

2025.11.7 NV OpenDay 杭州的CuTeDSL Slides

CuTeDSL as a codegen target

Example: Ampere gemm

- TileLang code

```
25  def gemm(
26          A: T.Tensor((M, K), dtype),
27          B: T.Tensor((K, N), dtype),
28          C: T.Tensor((M, N), dtype),
29  ):
30      with T.Kernel(T.ceildiv(N, block_N), T.ceildiv(M, block_M), threads=threads) as (bx, by):
31          A_shared = T.alloc_shared((block_M, block_K), dtype)                    SMEM allocation
32          B_shared = T.alloc_shared((block_K, block_N), dtype)
33          C_local = T.alloc_fragment((block_M, block_N), accum_dtype)             RF allocation
34
35          T.clear(C_local)                                                        RF LD/ST
36          for k in T.Pipelined(T.ceildiv(K, block_K), num_stages=3):
37              T.copy(A[by * block_M, k * block_K], A_shared)                      cp_async G2S
38              T.copy(B[k * block_K, bx * block_N], B_shared)
39              T.gemm(A_shared, B_shared, C_local)                                 mma GEMM
40
41          T.copy(C_local, C[by * block_M, bx * block_N])                          STG
42
43  return gemm
```

Ampere style Pipeline

1. **Prologue**: Issue async memory load of first #stages-1 tiles
2. **Mainloop**: k-Loop from 0 to #tiles-(#stages-1)
   1. Issue #k+#stages-1 async memory load
   2. Wait for #k async memory load ready
   3. Consume #k tile
3. **Epilogue**: Wait and consume for last #stages-1 tiles

- Generated CuTeDSL code

```
1   import cutlass
2   import cutlass.cute as cute
3   import tilelang.contrib.cutedsl as tl
4
5   @cute.kernel
6   def gemm_kernel(A, B, C) :
7     threadIdx = tl.ThreadIdx()                    Dedicated for CuTeDSL Backend
8     blockIdx = tl.BlockIdx()
9   buf_dyn_shmem = tl.get_dyn_smem(cutlass.Uint8, 128)
10  C_local = cute.make_fragment((128),cutlass.Float32)
11  for i in cutlass.range_constexpr(0, 64):
12    tl.make_tensor_offset(C_local.iterator, (i * 2), (2), div_by = 2).store(tl.make_filled_tensor((2), cutlass.Float32(0)).load())
13  for i_1 in cutlass.range_constexpr(0, 4):
14    tl.cp_async_gs(16, buf_dyn_shmem, ((((i_1 * 2048) + ((threadIdx.x >> 2) * 64)) + (((((threadIdx.x & 31) >> 4) + ((threadIdx.x &
15  for i_2 in cutlass.range_constexpr(0, 4):
16    tl.cp_async_gs(16, buf_dyn_shmem, ((((((((threadIdx.x & 15) >> 3) * 4096) + (i_2 * 1024)) + ((threadIdx.x >> 4) * 128)) + ((((tl
17  tl.cp_async_commit()
18  for i_3 in cutlass.range_constexpr(0, 4):
19    tl.cp_async_gs(16, buf_dyn_shmem, ((((i_3 * 2048) + ((threadIdx.x >> 2) * 64)) + (((((threadIdx.x & 31) >> 4) + ((threadIdx.x &
20  for i_4 in cutlass.range_constexpr(0, 4):
21    tl.cp_async_gs(16, buf_dyn_shmem, ((((((((threadIdx.x & 15) >> 3) * 4096) + (i_4 * 1024)) + ((threadIdx.x >> 4) * 128)) + ((((tl
22  tl.cp_async_commit()
23  for k in range(0, 30):
24    tl.sync_threads()
25    for i_5 in cutlass.range_constexpr(0, 4):
26      tl.cp_async_gs(16, buf_dyn_shmem, (((((((k + 2) % 3) * 8192) + (i_5 * 2048)) + ((threadIdx.x >> 2) * 64)) + (((((threadIdx.x &
27    for i_6 in cutlass.range_constexpr(0, 4):
28      tl.cp_async_gs(16, buf_dyn_shmem, ((((((((((k + 2) % 3) * 8192) + (((threadIdx.x & 15) >> 3) * 4096)) + (i_6 * 1024)) + ((thre
29    tl.cp_async_commit()
30    tl.cp_async_wait(2)
31    tl.sync_threads()
32    tl.gemm_ss(128, 128, 32, 2, 2, 0, 0, 0, 32, 128, 0, 0, (cutlass.Float16, (cute.recast_ptr(buf_dyn_shmem, dtype = cutlass.Float16
33  tl.cp_async_wait(1)
34  tl.sync_threads()
35  tl.gemm_ss(128, 128, 32, 2, 2, 0, 0, 0, 32, 128, 0, 0, (cutlass.Float16, (cute.recast_ptr(buf_dyn_shmem, dtype = cutlass.Float16)
36  tl.cp_async_wait(0)
37  tl.sync_threads()
38  tl.gemm_ss(128, 128, 32, 2, 2, 0, 0, 0, 32, 128, 0, 0, (cutlass.Float16, (cute.recast_ptr(buf_dyn_shmem, dtype = cutlass.Float16)
39  for i_7 in cutlass.range_constexpr(0, 64):
40    __1 = cute.make_fragment((2,), cutlass.Float16)
41    v_ = tl.make_tensor_offset(C_local.iterator, (i_7 * 2), (2), div_by = 2)
42    __1.store(v_.load().to(cutlass.Float16))
43    tl.make_tensor_offset(C.iterator, (((((((((blockIdx.y * 131072) + (((i_7 & 7) >> 1) * 32768)) + (((threadIdx.x & 63) >> 5) * 163
```

2025.11.7 NV OpenDay 杭州的CuTeDSL Slides

CuTeDSL as a codegen target

Example: Hopper fp8 gemm

- TileLang code

```
14 def matmul_NT(M, N, K, bM, bN, bK, in_dtype, out_dtype, accum_dtype):
15
16     @T.prim_func
17     def main(
18             A: T.Tensor((M, K), in_dtype),
19             B: T.Tensor((N, K), in_dtype),
20             C: T.Tensor((M, N), out_dtype),
21     ):
22         with T.Kernel(T.ceildiv(N, bN), T.ceildiv(M, bM), threads=128) as (bx, by):
23             A_shared = T.alloc_shared((bM, bK), in_dtype)
24             B_shared = T.alloc_shared((bN, bK), in_dtype)
25             C_local = T.alloc_fragment((bM, bN), accum_dtype)
26
27             T.clear(C_local)
28             for k in T.Pipelined(T.ceildiv(K, bK), num_stages=3):
29                 T.copy(A[by * bM, k * bK], A_shared)
30                 T.copy(B[bx * bN, k * bK], B_shared)
31                 T.gemm(A_shared, B_shared, C_local, transpose_B=True)
32
33             T.copy(C_local, C[by * bM, bx * bN])
34
35     return main
```

SMEM allocation

RF allocation

RF LD/ST

TMA copy G2S

wgmma GEMM

STG

Hopper style Warp-specialized Pipeline:

1. TMA warp: Loop on #stages buffers
   1. Wait for **empty** mbarrier ready
   2. Issue **TMA load**
   3. Arrive on **full** mbarrier
2. GEMM warp: Loop on #stages buffers
   1. Wait for **full** mbarrier ready
   2. Issue **wgmma GEMM**
   3. Arrive on **empty** mbarrier

- Generated CuTeDSL code

Dedicated for CuTeDSL Backend

```
1  import cutlass
2  import cutlass.cute as cute
3  import tilelang.contrib.cutedsl as tl
4
5  @cute.kernel
6  def gemm_kernel(A_desc, B_desc, C, tensormaps) :
7    threadIdx = tl.ThreadIdx()
8    blockIdx = tl.BlockIdx()
9    A_desc_ptr, B_desc_ptr = tl.get_tensormap_ptrs(1, tensormaps, (A_desc, B_desc))
10   buf_dyn_shmem = tl.get_dyn_smem(cutlass.Uint8, alignment = 1024)
11   C_local = cute.make_fragment((128),cutlass.Float32)
12   mbarrier = tl.alloc_smem(cutlass.Uint64, size_in_elems = 6, alignment = 64)
13   if tl.shuffle_elect(0):
14     with cute.arch.elect_one():
15       tl.prefetch_tma_descriptor(A_desc)
16       tl.prefetch_tma_descriptor(B_desc)
17       tl.mbarrier_init((mbarrier+0), 128)
18       tl.mbarrier_init((mbarrier+1), 128)
19       tl.mbarrier_init((mbarrier+2), 128)
20       tl.mbarrier_init((mbarrier+3), 128)
21       tl.mbarrier_init((mbarrier+4), 128)
22       tl.mbarrier_init((mbarrier+5), 128)
23   tl.sync_threads()
24   if 128 <= threadIdx.x:
25     tl.warpgroup_reg_dealloc(24)
26     for k in range(0, 16):
27       tl.mbarrier_wait((mbarrier+((k % 3) + 3)), (((k % 6) // 3) ^ 1))
28       if tl.shuffle_elect(128):
29         with cute.arch.elect_one():
30           tl.mbarrier_expect_tx((mbarrier+(k % 3)) ,8192)
31           tl.fence_proxy_async()
32         tl.tma_load(A_desc_ptr, (mbarrier+(k % 3)), ((cute.recast_ptr(buf_dyn_shmem, dtype = cutlass.Float8E4M3FN) + ((k % 3) * 8192))), ((k * 6
33           tl.mbarrier_expect_tx((mbarrier+(k % 3)) ,8192)
34           tl.fence_proxy_async()
35         tl.tma_load(B_desc_ptr, (mbarrier+(k % 3)), ((cute.recast_ptr(buf_dyn_shmem, dtype = cutlass.Float8E4M3FN) + (((k % 3) * 8192) + 24576))
36       tl.mbarrier_arrive((mbarrier+(k % 3)))
37   else:
38     tl.warpgroup_reg_alloc(240)
39     for i in cutlass.range_constexpr(0, 64):
40       tl.make_tensor_at_offset(C_local.iterator, (i * 2), (2), div_by = 2).store(tl.make_filled_tensor((2), cutlass.Float32(0)).load())
41     for k_1 in range(0, 16):
42       tl.mbarrier_wait((mbarrier+(k_1 % 3)), ((k_1 % 6) // 3))
43       tl.fence_proxy_async()
44       tl.gemm_ss(128, 128, 64, 4, 1, 0, 1, 0, 64, 64, 0, 0, True, A_ptr = ((cute.recast_ptr(buf_dyn_shmem, dtype = cutlass.Float8E4M3FN) + ((k_1 %
45       tl.mbarrier_arrive((mbarrier+((k_1 % 3) + 3)))
46     for i_1 in cutlass.range_constexpr(0, 64):
47       tl.make_tensor_at_offset(C.iterator, (((((((blockIdx.y * 131072) + ((i_1 >> 5) * 65536)) + ((threadIdx.x >> 5) * 16384)) + ((i_1 & 1) * 819
```

简介 OVERVIEW

核心问题：在写算子这个任务中，TileLang 隐藏了什么？要求我们决定什么？(5 min)

主线：

1、Language Basics 、JIT 、TIR / Lowering Trace、TVM-FFI （20 min）

2、T.Kernel、 Fragment、 T.Parallel、Layout Inference (20 min)

3、Tile Library Primitives、Layout Annotation、Reducer、Replication (20 min)

4、T.Pipelined、copy lowering、Autotuning / Carver (15 min)

5、Persistent scheduling 与 MegaKernel-style control (10 min)

6、Runtime Profiling、IKET 与 SASS evidence (10 min)

核心：写kernel需要处理的四个问题

1. 一个 CTA、warp、thread 负责哪些逻辑元素?
2. 数据位于 global、shared 还是 fragment/register?
3. load、compute、store 如何排序和重叠?
4. tile、threads、stages、worker CTA 数量多大?

TileLang隐藏了什么？要求我们决定什么?

1. 可以大量自动推导线程 ownership，但 CTA workload 仍由用户决定。
2. 用户选择 memory scope，编译器推导许多具体地址映射和搬运机制。
3. 规则流水可由 T.Pipelined 自动展开；复杂依赖仍需用户表达结构。
4. 主要是搜索空间，不存在一个接口自动给出普遍最优答案。

1.1 最小 kernel

Python factory：描述 specialization 参数并构造 kernel。

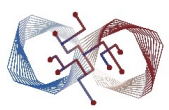
T.prim_func：声明设备端计算与 buffer contract。

T.Kernel：声明 CTA grid 和每个 CTA 的线程数。

T.Parallel：给出逻辑并行域，由 layout inference 映射到线程与 thread-local iterations。

```python
@tilelang.jit(out_idx=[-1])
def make_add(n, block_n=256):
    @T.prim_func
    def kernel(
        x: T.Tensor((n,), T.float32),
        y: T.Tensor((n,), T.float32),
    ):
        with T.Kernel(T.ceildiv(n, block_n), threads=128) as bx:
            for j in T.Parallel(block_n):
                idx = bx * block_n + j
                if idx < n:
                    y[idx] = x[idx] + 1.0

    return kernel
```

1.2 JIT (Just-in-time compilation)



TileLang 0.1.12 的 @tilelang.jit 支持两种主要使用模式：Eager JIT 和 Lazy JIT。在 Lazy JIT 中，用户首先定义参数化 kernel；当给定静态参数（specialization）后，编译器在首次调用时完成 Lowering、优化和代码生成，并缓存生成的 kernel artifact。后续具有相同静态配置的调用可以直接复用已有 artifact，而无需重复编译；动态参数则在运行时传入，用于控制具体 workload 规模。生成的 artifact 除了可以直接执行外，还支持导出 CUDA 源码、获取 profiler 信息、比较不同 schedule specialization 的性能，以及查看和保存 host/device 端代码。

	Eager JIT	Lazy JIT
Python 形式	tensor 作为函数参数，函数体构造并调用 kernel	factory 返回 `PrimFunc`
用户调用	接近普通 PyTorch op	先获得 `JITKernel`，再传 tensor
输出	常用 `T.empty()` 后 return	显式输出参数或 `out_idx`

1.2 JIT (Just-in-time compilation)

JIT 核心是在编译期 specialization 能力与运行时灵活性之间进行权衡：静态参数决定 kernel 的实现方式，使编译器能够针对特定硬件、数据布局和资源约束进行深度优化；动态参数决定 kernel 的执行规模，提高 artifact 复用率并降低编译开销。

经验原则：影响 kernel 实现结构的参数，例如数据布局（layout）、tile 配置、资源分配（shared memory、register usage）以及控制流结构应进行静态特化，以释放代码生成优化空间；仅影响执行规模的参数，例如 token 数、batch size、problem size 或 loop trip count应保持动态，以避免编译缓存膨胀并提升 kernel artifact 的复用能力。

此外，Lazy JIT 在运行时触发编译的具体流程涉及缓存管理、编译调度以及运行时交互等实现细节，不在此展开。

1.3 编译链条

========== User Source ==========

Python TileLang DSL

factory / eager function / @tilelang.jit / @T.prim_func →

========== Frontend Elaboration ==========

执行 factory、静态参数 specialization、解析 DSL →

Apache TVM **TIRx** PrimFunc / IRModule

+ TileLang Tile Library primitives

+ control-flow / layout / pipeline / reducer annotations →

========== Compiler Middle-End ==========

Target normalization / BindTarget →

architecture: sm_80 / sm_90 / sm_100 、 **backend** key: plain CUDA / CuTeDSL

1.3 编译链条

Canonicalization / LayoutReducer

PipelinePlanning / InjectSoftwarePipeline

Warp-specialization planning（目标相关） →

LayoutInference

推导 fragment ownership、parallel-loop layout、shared layout →

LowerTileOp

生成 physical indices、thread-local loops、目标相关 intrinsic →

Target-specific lowered TIRx

legalization / safe access 、 vectorization / unrolling 、 storage planning与reuse 、 synchronization / fence insertion →

SplitHostDevice + ABI / launch lowering →

1.3 编译链条

========== Backend Build ==========

Host side:

host TIRx / launcher metadata / host stub

Device side:

├── CUDA C++ → NVCC/NVRTC → cubin/SASS

└── CuTeDSL Python → nvidia-cutlass-dsl → cubin/SASS →

========== JIT / Runtime ==========

JIT cache / module loading / argument-output organization →

TVM-FFI backend（plain CUDA默认）或 Cython / NVRTC / CuTeDSL adapter → CUDA Driver API → kernel launch →

1.3 编译链条

========== Evaluation ==========

correctness → benchmark → runtime profile → tune

特别推荐：https://tilelang.com/tools/lower_trace.html

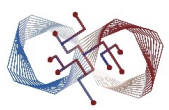
可以通过IR Lower Trace 展示 TileLang 的 TIR 随着编译器 Pass 执行过程中的变化。

1.4 TVM-FFI

有关TVM-FFI的更多细节可以参考：https://tilelang.com/blog/2025/12/11/ffi.html，简单来说， TVM-FFI（Foreign Function Interface） 是 TVM 独立出的跨语言运行时接口，提供了一套统一、稳定的 ABI，使 C++、Python、Rust、Java 等不同语言能够无缝调用同一套运行时和编译产物。

TileLang 基于 TVM-FFI 构建 Host Runtime，将编译生成的 Host Stub 与 Device Module 封装为可直接调用的 Python 对象。它能够直接接收 PyTorch-compatible Tensor，自动完成 Tensor 元数据（Shape、Stride、Dtype、Device）的获取与校验，并组织 Kernel 所需的参数、管理输出 Tensor、执行资源初始化（如 TMA Descriptor）、完成 Kernel Launch，而无需用户在 Python 端手动编写 Shape Check 或组织 Launch 参数，使 GPU Kernel 能够像普通 Python 函数一样调用。

1.4 TVM-FFI



TVM-FFI 的核心优势 在于极低的接口调用开销。相比传统基于 PyBind 的绑定方式，TVM-FFI 具有更低的编译期开销和运行期开销，并采用开放、稳定的 ABI，避免接口绑定依赖于特定框架（如 PyTorch）。

对于执行时间仅为几微秒的 GPU Kernel，Python 调用和 Kernel Launch 已经成为不可忽略的系统开销，而 TVM-FFI 能够将这些 Host 侧操作下沉到编译生成的高性能 C++ Host Stub 中，减少 Python Runtime 参与，从而显著降低 Host Overhead，提高整体执行效率。

Tech Report From DS-V4

Reducing Invocation Overhead with Host Codegen. As accelerators continue to grow in performance, CPU-side orchestration overhead becomes increasingly prominent. For small, highly optimized kernels, such fixed host overhead can easily cap utilization and throughput. A common source of this overhead is that host-side logic, such as runtime contract checks, is typically written in Python for flexibility and thus incurs a fixed per-invocation cost.

We mitigate this overhead with *Host Codegen*, which moves most host-side logic into generated host code. Specifically, we first co-generate the device kernel and a lightweight host launcher at the IR (Intermediate Representation) level, embedding the necessary metadata—such as data types, rank/shape constraints, and stride/layout assumptions—parsed from the language frontend. The launcher is then lowered to the host source code built on top of the TVM-FFI (Chen et al., 2018) framework, whose compact calling convention and zero-copy tensor interop together minimize host-side overhead. At runtime, this generated host code performs validation and argument marshaling, shifting all per-invocation checks out of the Python execution path. Our measurements show that CPU-side validation overhead drops from tens or hundreds of microseconds to less than one microsecond per invocation.

1.5 TIRx

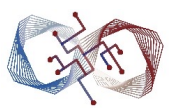
https://tvm.apache.org/2026/06/22/tirx TIRx是用于编写 GPU kernel 的 Python DSL。程序可以直接使用 threads、SMEM、TMEM、barriers 和 Tensor Core 等硬件概念，同时保留结构化的中间表示。

TIRx 中的 tile 操作主要由三项信息决定：哪些 threads 执行操作（scope）、数据采用什么布局（layout），以及操作通过哪条硬件路径实现（dispatch）。

https://mlc.ai/modern-gpu-programming-for-mlsys/ 此处有详细的教程。

TileLang 前端最终构造的是 tvm.tirx.PrimFunc，再包装成 IRModule。这里的 TIR/TIRX 可以反复分析和改写的结构化命令式 tensor IR，并progressive lowering 一步步生成到最后的artifact。整个过程依赖 target、scope、shape、stride、alignment、layout 与 pipeline context 的全局分析和实现选择。

2.1 T.Kernel



定义 CTA workload

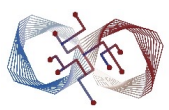
```
with T.Kernel(grid_x, grid_y, threads=256) as (bx, by):
    ...
```

需要做出的决定：一共有多少逻辑 CTA，一个 CTA 负责哪个 tile，一个 CTA 有多少线程可用于 tile 内并行?

TileLang 不替用户选择数学分块。若一个 CTA 的 tile 太小，reuse 和 ILP 不足；太大则 register/shared-memory 压力上升。

编译器自动化的是 tile 内映射，不是取消 workload decomposition。

2.2 Fragment



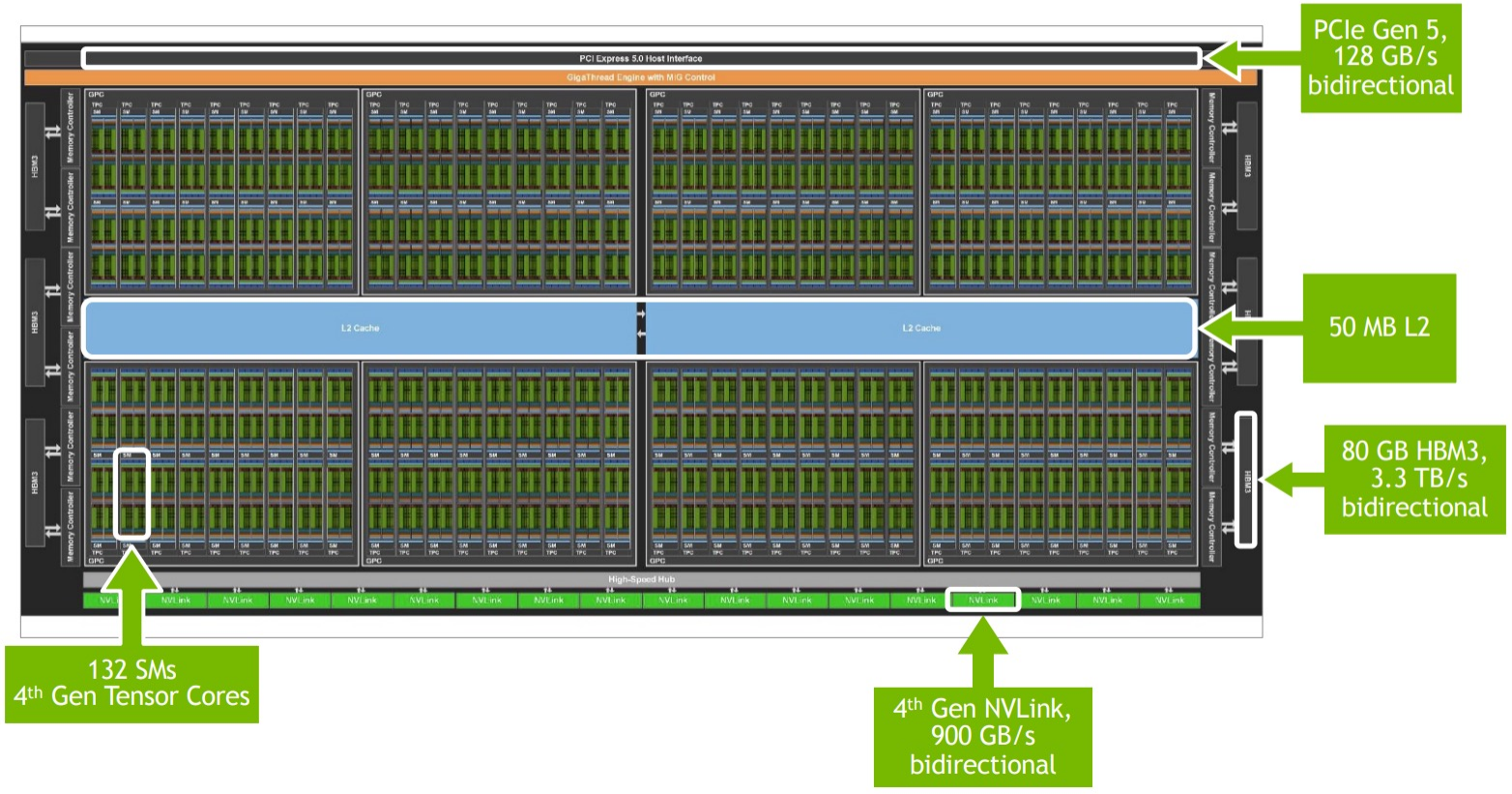
Scope	TileLang 对象	主要用途
Global	`T.Tensor` / `T.StridedTensor`	完整输入输出，跨 CTA 可见
Shared	`T.alloc_shared`	CTA staging、重排、跨线程复用
Fragment/register	`T.alloc_fragment`	逻辑 tile 的 thread-distributed register state

T.alloc_fragment((M, N)) 表示 CTA 共同持有一个逻辑 M×N fragment（不是每个线程都有完整 M×N 私有副本）。

物理映射可写成：

Fragment layout: (logical indices) → (thread_id, thread-local slot)

Recall: Memory Hierarchy



Recall: Memory Hierarchy

Streaming Multiprocessor (SM)

Hopper architecture

SM has 4 sub-partitions

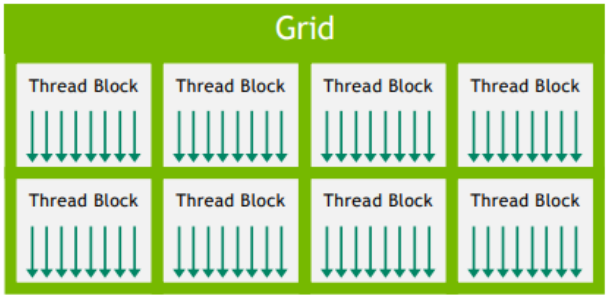
- 128 FP32 units
- 64 FP64 units
- 64 INT32 units
- 4 mixed-precision Tensor Cores
- 16 special function units (transcendentals)
- 4 warp schedulers

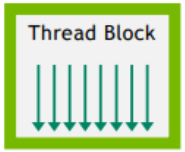
- 32 LD/ST units
- 64K 32-bit registers
- 256 KiB unified L1 data cache and shared memory
- Tensor Memory Accelerator (TMA)

Recall: Memory Hierarchy

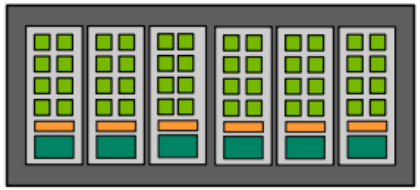
Thread Hierarchy: Grid & Blocks

CUDA/Software

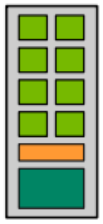




Hardware



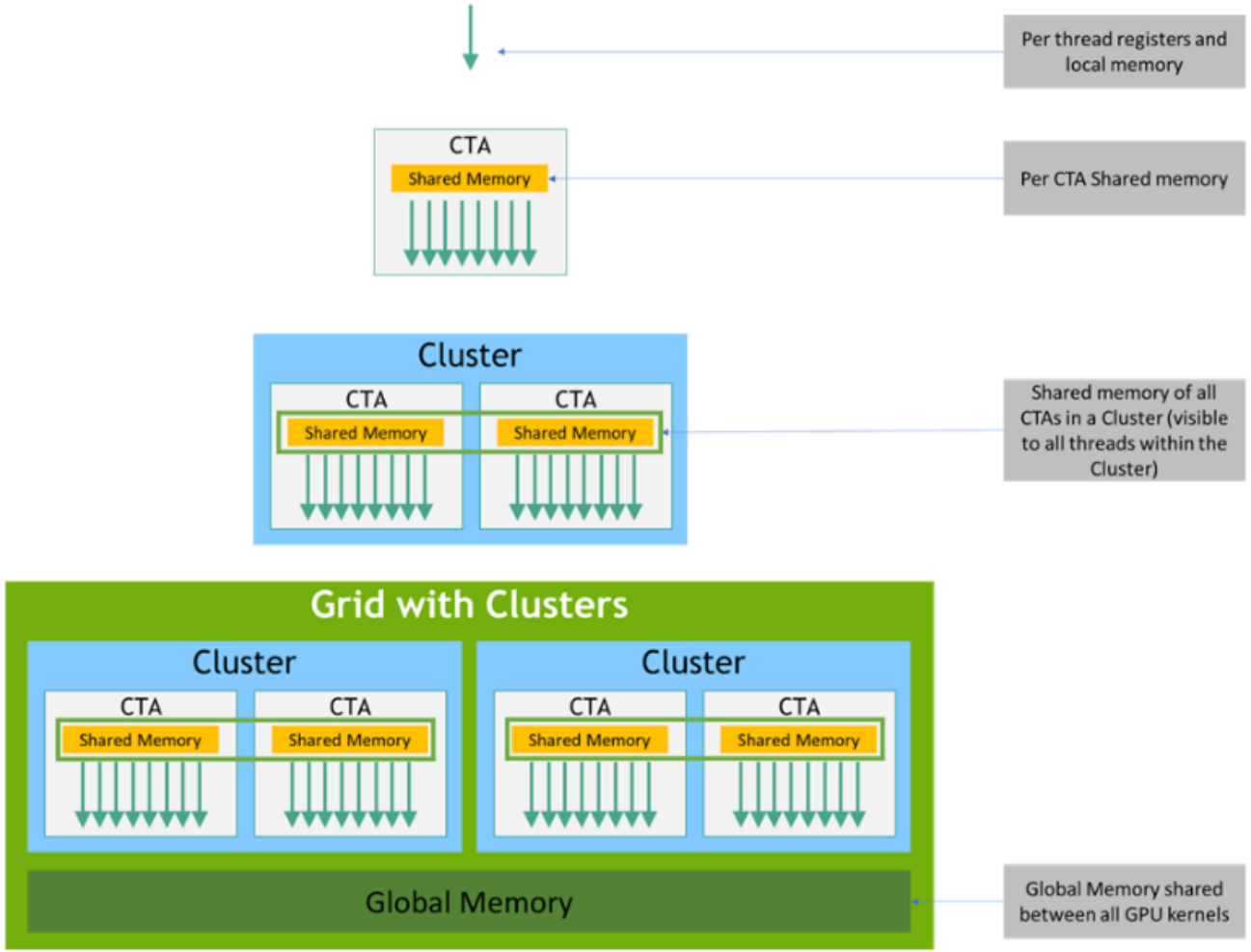
Device



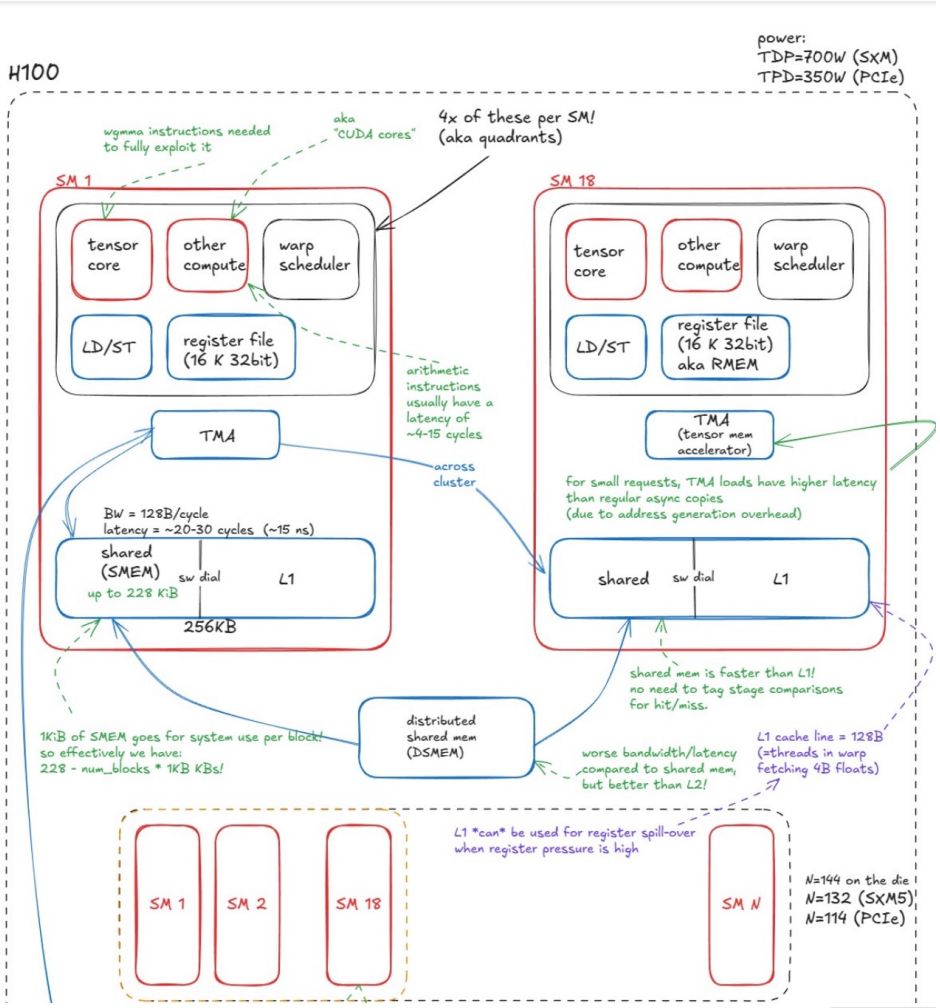
SM

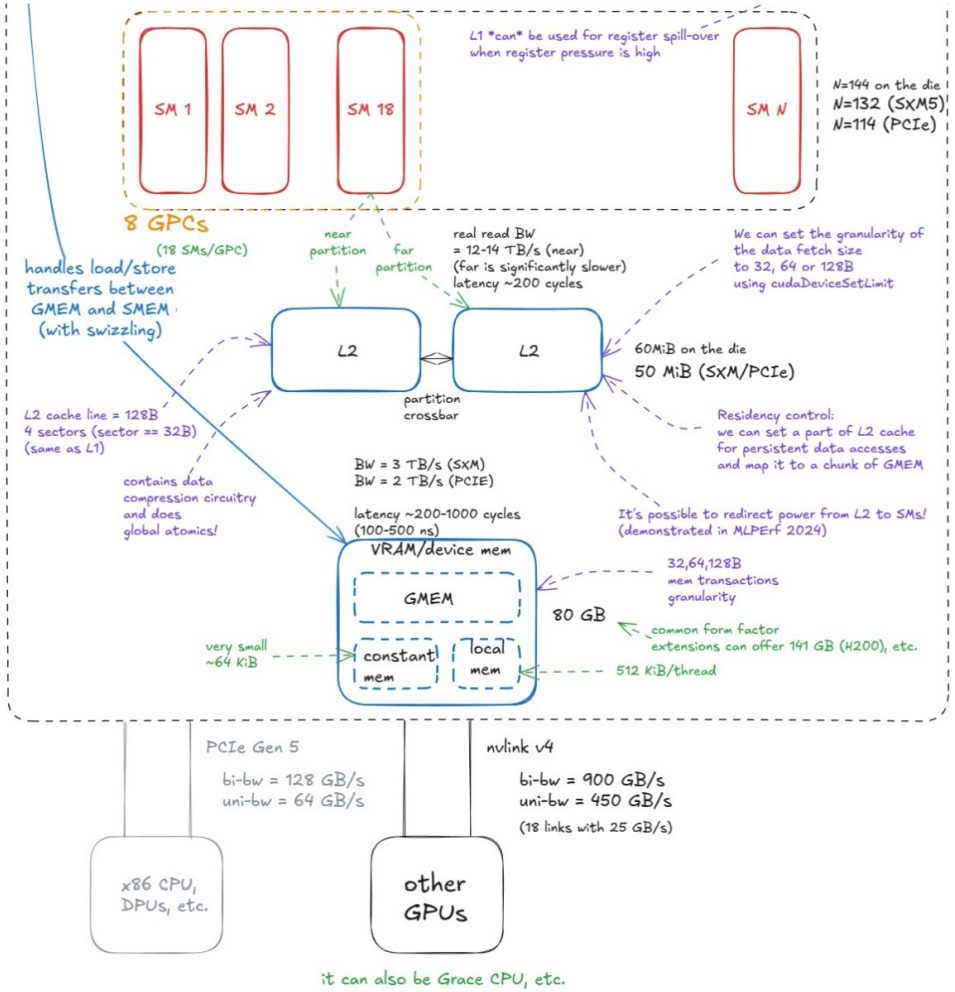
- A CUDA kernel is a launched grid of thread blocks, which are completely independent.
- Thread blocks are executed on SMs.
  - Several blocks can reside on an SM concurrently.
  - Blocks do not migrate.
  - Each block can be scheduled on any of the available SMs, in any order, concurrently or in series.

Recall: Memory Hierarchy



Recall: Memory Hierarchy





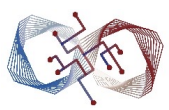
2.3 T.Parallel

```
for i, j in T.Parallel(block_m, block_n):
    y_frag[i, j] = activation(x_frag[i, j])
```

程序员描述完整逻辑迭代域。

Layout inference 结合输入、输出、parallel-loop layout 与 tile-operator 约束，决定：(i, j) 由哪个 thread 执行，该 thread 的哪个 local iteration，访问哪个 register slot，是否形成 vectorized load/store等等。

2.4 Layout Inference



LayoutInference 以显式 layout、目标 ISA TileOp contract 和线程范围为约束锚点，通过 Gemm、Copy、Parallel/elementwise、Reduce 等算子的 InferLayout 规则，在 buffer-use graph 上执行 strict constraint extraction、fixed-point propagation 和 free-layout candidate selection，最终得到 fragment ownership/register packing、parallel-loop thread mapping，以及参与相关 TileOp 的 shared-memory physical/swizzle layout。（Global tensor 的 shape/stride 属于地址布局层，不等同于 fragment ownership，但会在 Copy、TMA 和向量化的合法性判断中与其发生关联。）

Layout	映射
Fragment layout	logical index → thread/local slot
Parallel loop layout	logical iteration → thread
Shared layout	logical coordinate → shared address/swizzle

2.5 Case Study: Transpose

CTA workload:

每个 CTA 使用 256 个线程，处理一个输入 tile，并写出其转置 tile：

input: block_x × block_y

output: block_y × block_x

block_x 和 block_y 分别根据对应维度选择 64 或 128，因此代码实际支持：

生产阶段中，每个线程沿输入 tensor 的最后一个连续维度执行 vec4 合并读取。对于 BF16，每次读取 4 个元素，即 8 bytes。

```python
def _kernel(shape_x_mod_128: int, shape_y_mod_128: int, dtype: T.dtype):
    batches = T.dynamic("batches")
    shape_x = T.dynamic("shape_x")
    shape_y = T.dynamic("shape_y")
    stride_x = T.dynamic("stride_x")
    threads = 256
    block_x = 128 if shape_x_mod_128 == 0 else 64
    block_y = 128 if shape_y_mod_128 == 0 else 64
    block_k = 4
    threads_per_row = block_y // block_k
    layout = T.Fragment((block_y, block_x), forward_fn=_loop_layout(block_x, threads))

    @T.prim_func
    def impl(
        x: T.StridedTensor[(batches, shape_x, shape_y), (shape_x * stride_x, stride_x, 1), dtype],
        out: T.Tensor[(batches, shape_y, shape_x), dtype],
    ):
        with T.Kernel(shape_y // block_y, shape_x // block_x, batches, threads=threads) as (
            by,
            bx,
            batch,
        ):
            smem = T.alloc_shared((block_y, block_x + block_k), dtype)
            tid = T.get_thread_binding()
            row, col = tid // threads_per_row, tid % threads_per_row

            T.assume(shape_x % block_x == 0)
            T.assume(shape_y % block_y == 0)
            T.assume(stride_x % block_k == 0)

            tmp = T.alloc_local((block_k, block_k), dtype)
            row_tmp = T.alloc_local((block_k,), dtype)
            for ii in T.unroll(block_x // block_k // (threads // threads_per_row)):
                i = ii * (threads // threads_per_row) + row
                for j in T.unroll(block_k):
                    for kk in T.vectorized(block_k):
                        row_tmp[kk] = x[batch, bx * block_x + i * block_k + j, by * block_y + col * block_k + kk]
                    for kk in T.unroll(block_k):
                        tmp[kk, j] = row_tmp[kk]
                for j in T.unroll(block_k):
                    sj = (j + tid // (8 // dtype.bytes)) % block_k
                    for kk in T.vectorized(block_k):
                        smem[col * block_k + sj, i * block_k + kk] = tmp[sj, kk]
            T.sync_threads()
            for i, j in T.Parallel(block_y, block_x, loop_layout=layout):
                out[batch, by * block_y + i, bx * block_x + j] = smem[i, j]

    return impl
```

2.5 Case Study: Transpose

Shared memory 两个作用：

1、数据转置与线程间交换

global input →连续 vec4 read → thread-local 4×4 tile → register micro-transpose → shared memory → 按输出顺序连续 vec4 read → global output

线程先读取一个局部 4×4 tile，通过 tmp[kk, j] 在线程寄存器内完成 micro-transpose，然后按转置后的 (y, x) 坐标写入 shared memory。使输入和输出两个阶段都能够沿各自的连续维度执行合并、向量化访问。

```python
def _kernel(shape_x_mod_128: int, shape_y_mod_128: int, dtype: T.dtype):
    batches = T.dynamic("batches")
    shape_x = T.dynamic("shape_x")
    shape_y = T.dynamic("shape_y")
    stride_x = T.dynamic("stride_x")
    threads = 256
    block_x = 128 if shape_x_mod_128 == 0 else 64
    block_y = 128 if shape_y_mod_128 == 0 else 64
    block_k = 4
    threads_per_row = block_y // block_k
    layout = T.Fragment((block_y, block_x), forward_fn=_loop_layout(block_x, threads))

    @T.prim_func
    def impl(
        x: T.StridedTensor[(batches, shape_x, shape_y), (shape_x * stride_x, stride_x, 1), dtype],
        out: T.Tensor[(batches, shape_y, shape_x), dtype],
    ):
        with T.Kernel(shape_y // block_y, shape_x // block_x, batches, threads=threads) as (
            by,
            bx,
            batch,
        ):
            smem = T.alloc_shared((block_y, block_x + block_k), dtype)
            tid = T.get_thread_binding()
            row, col = tid // threads_per_row, tid % threads_per_row

            T.assume(shape_x % block_x == 0)
            T.assume(shape_y % block_y == 0)
            T.assume(stride_x % block_k == 0)

            tmp = T.alloc_local((block_k, block_k), dtype)
            row_tmp = T.alloc_local((block_k,), dtype)
            for ii in T.unroll(block_x // block_k // (threads // threads_per_row)):
                i = ii * (threads // threads_per_row) + row
                for j in T.unroll(block_k):
                    for kk in T.vectorized(block_k):
                        row_tmp[kk] = x[batch, bx * block_x + i * block_k + j, by * block_y + col * block_k + kk]
                    for kk in T.unroll(block_k):
                        tmp[kk, j] = row_tmp[kk]
                for j in T.unroll(block_k):
                    sj = (j + tid // (8 // dtype.bytes)) % block_k
                    for kk in T.vectorized(block_k):
                        smem[col * block_k + sj, i * block_k + kk] = tmp[sj, kk]
            T.sync_threads()
            for i, j in T.Parallel(block_y, block_x, loop_layout=layout):
                out[batch, by * block_y + i, bx * block_x + j] = smem[i, j]

    return impl
```

2.5 Case Study: Transpose

Shared memory 两个作用：

2. 改善 shared-memory bank mapping

第二维额外增加 block_k=4：

logical width = block_x

physical shared stride = block_x + 4

Padding 打破相邻 shared-memory 行之间不利的 bank 周期对齐，同时保持 vec4 访问所需的对齐。

此外：sj 对每个线程的四个转置行进行循环置换，使同一条 shared vector-store 指令中的不同 lane group 访问不同的行偏移；配合 padding 提供的逐行 bank 偏移，将访问分散到更多 smem banks。

```python
def _kernel(shape_x_mod_128: int, shape_y_mod_128: int, dtype: T.dtype):
    batches = T.dynamic("batches")
    shape_x = T.dynamic("shape_x")
    shape_y = T.dynamic("shape_y")
    stride_x = T.dynamic("stride_x")
    threads = 256
    block_x = 128 if shape_x_mod_128 == 0 else 64
    block_y = 128 if shape_y_mod_128 == 0 else 64
    block_k = 4
    threads_per_row = block_y // block_k
    layout = T.Fragment((block_y, block_x), forward_fn=_loop_layout(block_x, threads))

    @T.prim_func
    def impl(
        x: T.StridedTensor[(batches, shape_x, shape_y), (shape_x * stride_x, stride_x, 1), dtype],
        out: T.Tensor[(batches, shape_y, shape_x), dtype],
    ):
        with T.Kernel(shape_y // block_y, shape_x // block_x, batches, threads=threads) as (
            by,
            bx,
            batch,
        ):
            smem = T.alloc_shared((block_y, block_x + block_k), dtype)
            tid = T.get_thread_binding()
            row, col = tid // threads_per_row, tid % threads_per_row

            T.assume(shape_x % block_x == 0)
            T.assume(shape_y % block_y == 0)
            T.assume(stride_x % block_k == 0)

            tmp = T.alloc_local((block_k, block_k), dtype)
            row_tmp = T.alloc_local((block_k,), dtype)
            for ii in T.unroll(block_x // block_k // (threads // threads_per_row)):
                i = ii * (threads // threads_per_row) + row
                for j in T.unroll(block_k):
                    for kk in T.vectorized(block_k):
                        row_tmp[kk] = x[batch, bx * block_x + i * block_k + j, by * block_y + col * block_k + kk]
                    for kk in T.unroll(block_k):
                        tmp[kk, j] = row_tmp[kk]
                for j in T.unroll(block_k):
                    sj = (j + tid // (8 // dtype.bytes)) % block_k
                    for kk in T.vectorized(block_k):
                        smem[col * block_k + sj, i * block_k + kk] = tmp[sj, kk]
            T.sync_threads()
            for i, j in T.Parallel(block_y, block_x, loop_layout=layout):
                out[batch, by * block_y + i, bx * block_x + j] = smem[i, j]

    return impl
```

2.5 Case Study: Transpose

显式 loop layout：layout = T.Fragment((block_y, block_x), forward_fn=_loop_layout(block_x, threads)) Producer 阶段完成 shared-memory transpose 后，consumer 阶段使用显式 loop_layout 将输出 tile 的逻辑坐标 (i, j) 映射为：(thread_id, thread-local slot)

映射首先按照 row-major 顺序展平输出坐标： elem = i * block_x + j

然后将每四个连续输出元素组成一个 vec4 chunk，因此：每个 vec4 连续片段由同一个线程处理；vec4 chunk 在 256 个线程之间轮转；每个线程可能拥有多个彼此分离的 vec4 片段；同一 warp 每轮覆盖一段连续输出区域，便于合并写回。

loop_layout 描述的是输出 iteration ownership。LayoutInference 负责验证和记录映射，后续 lowering 再生成 thread-local loops、地址表达式与可能的向量化访问。

```
def _loop_layout(block_x: int, threads: int = 256):
    def fn(i, j):
        elem = i * block_x + j
        return (elem // 4) % threads, elem % 4 + elem // (threads * 4) * 4

    return fn
```

```
for i, j in T.Parallel(block_y, block_x, loop_layout=layout):
    out[batch, by * block_y + i, bx * block_x + j] = smem[i, j]
```

2.5 Case Study: Transpose

同步边界：global input → thread-local tmp → shared memory → T.sync_threads() → output loop → global output

写入某个 shared element 的线程，通常不是随后读取该元素并写回输出的线程。因此，shared 写入和读取之间存在 CTA 内的跨线程、跨 warp producer-consumer 依赖。

```
# Producer phase   smem[...] = tmp[...] T.sync_threads()
# Consumer phase  out[...] = smem[...]
```

- 这里的 T.sync_threads() 同时提供：CTA-wide 阶段汇合：所有线程完成 producer phase 后才能进入 consumer phase；shared-memory ordering：barrier 前的 shared writes 对 barrier 后的 CTA 内读取可见。

由于 shared-memory load/store 是手工编写的，barrier 必须显式表达。loop_layout 只规定 iteration ownership，不提供同步语义。

3.1 Tile Library Primitives

Case Study : RMSNorm

$s = \Sigma_j x_j^2$

$rstd = rsqrt(s / N + eps)$

$y_j = x_j \cdot rstd \cdot weight_j$

数据流包含：

global → fragment/shared copy；

elementwise square；

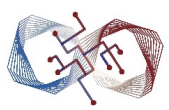
hidden 维 reduction；

一个 row scalar 被同一行所有元素复用；

elementwise scale 与 writeback。

```python
@tilelang.jit(pass_configs={"tl.disable_tma_lower": True})
def rms_norm(A, blk_m):
    M, N = T.const("M, N")
    dtype = T.float

    A: T.Tensor((M, N), dtype)
    B = T.empty((M, N), dtype)

    with T.Kernel(T.ceildiv(M, blk_m), threads=128) as bx:
        A_local = T.alloc_fragment((blk_m, N), dtype)
        A_pow_local = T.alloc_fragment((blk_m, N), dtype)
        A_powsum = T.alloc_fragment((blk_m,), dtype)

        T.copy(A[bx * blk_m : (bx + 1) * blk_m, :], A_local)
        for i, j in T.Parallel(blk_m, N):
            A_pow_local[i, j] = A_local[i, j] * A_local[i, j]
        T.reduce_sum(A_pow_local, A_powsum, dim=1)
        for i in T.Parallel(blk_m):
            A_powsum[i] = T.rsqrt(A_powsum[i] / N + 1e-12)
        for i, j in T.Parallel(blk_m, N):
            A_local[i, j] *= A_powsum[i]
        T.copy(A_local, B[bx * blk_m : (bx + 1) * blk_m, :])

    return B
```

3.1 Tile Library Primitives

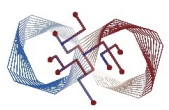


Tile Library primitives 的核心价值：让数据移动、归约和矩阵计算在 IR 中继续保留可分析的 tile/region 语义，而不是过早展开为逐线程循环和地址计算。

它们与逻辑并行循环共同构成 layout inference 的约束图。

- T.copy 保留 source/destination buffer、memory scope、访问 region 和读写方向，使编译器能够协调两端 layout，并选择global/shared/fragment 之间目标相关搬运实现。
- T.reduce_sum 保留 source/destination region、reduction dimension 和类型，使编译器能够根据 source fragment ownership 推导并验证 reducer/result layout。
- T.gemm 保留矩阵 tile、M/N/K、转置方式和目标 ISA 契约，是 MMA/WGMMA/tcgen05 fragment 与 shared-memory layout 的重要约束锚点。
- T.Parallel 描述逻辑并行迭代域，可以携带显式 loop_layout；否则编译器可根据循环中的 fragment 读写关系推导 thread ownership。

3.1 Tile Library Primitives



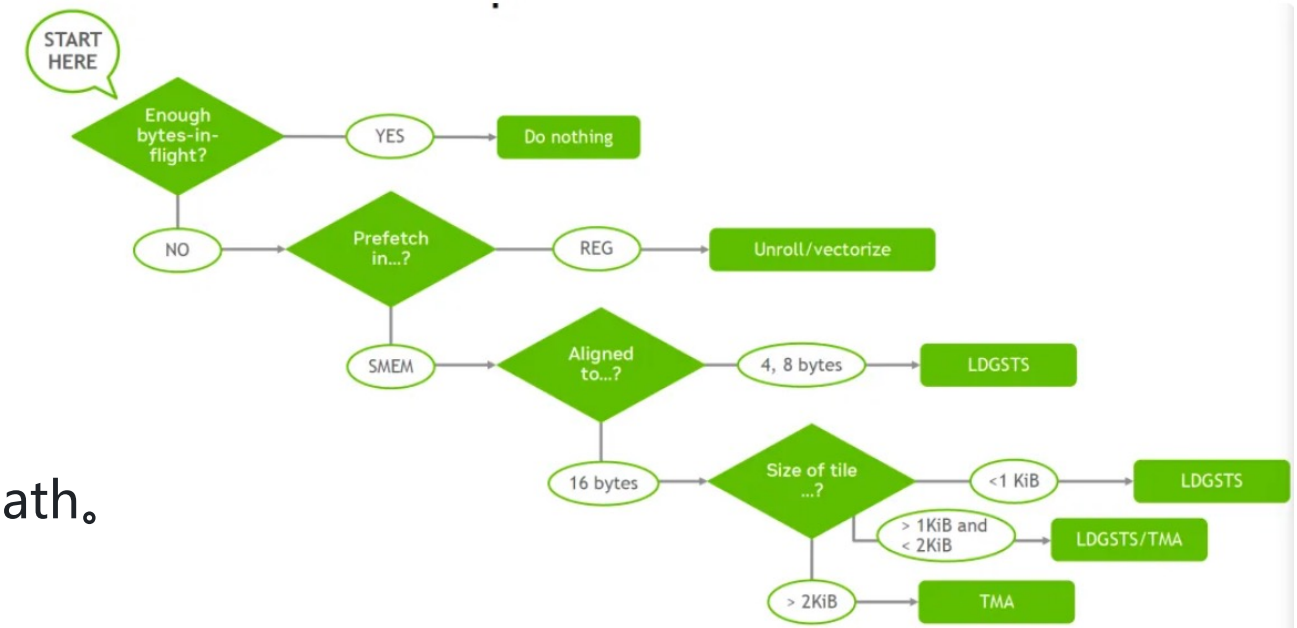
Layout Inference：将显式 layout annotation、目标 ISA 的 TileOp contract、parallel-loop mapping，以及通过 buffer use-def/access graph 相连的算子进行迭代传播、补全和冲突检查，最终得到兼容的 fragment ownership、parallel-loop thread mapping，以及相关 shared-memory layout。

这种语义保留适用于 Tile Library primitive；普通手写的 shared load/store 虽然仍可参与访问分析，但不会自动获得 primitive 所封装的搬运或同步协议。

3.1 Tile Library Primitives

同一个 T.copy(src, dst) 根据 scope、shape、alignment、target 和 schedule，可能 lowering 成：

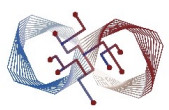
- scalar/vector load-store；
- CTA cooperative copy；
- cp.async 风格异步搬运；
- TMA；
- 对应 target 的其他 copy path。



（图片来自GTC25的Maximum Memory Bandwidth：Optimization Guidelines）

因此 T.copy 描述数据运动；target-specific lowering 选择实现。

3.2 Layout Annotation



E5M6 quant 将每个值编码为 12 bits，因此每 8 个连续的待量化值需要打包成 3 个 uint32。float_to_e5m6 使用线程私有的 T.LocalBuffer 完成位级打包，所以在执行 packing 时，这 8 个值必须能够由同一线程以thread-local 方式访问。

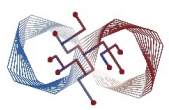
T.annotate_layout 显式规定 out_fragment 的 ownership：每个连续 8 元素组归同一线程所有，并映射到连续的 local slots。LayoutInference 再将这一约束传播到产生 out_fragment 的 normalization loop 和消费它的 packing loop，使 packing 可以直接成为线程局部计算，而不需要额外的数据重排或跨线程交换。

（LayoutInference 可能自行推导出同样的布局，但不再有显式的 8-wide ownership 保证；若得到不兼容布局，则需要重新布局、跨线程交换，或者产生 layout conflict。）

```
T.annotate_layout({
    out_fragment: T.Fragment(
        (block_m, block_k),
        forward_fn=out_forward_fn,
    ),
})
```

```
def out_forward_fn(i: int, j: int):
    elems = i * block_k + j
    return elems // 8 % num_threads, elems % 8 + elems // (8 * num_threads) * 8
```

3.3 Reducer

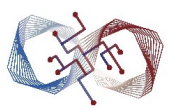


核心原则：

Reduction API 的选择取决于两个 ownership 问题：1. 一个逻辑输出的 partial values 分布在哪些线程；2. 最终结果需要由哪些线程持有和消费。

- partial 全在线程内：使用 serial/unroll 本地累计。
- partial 分布在完整 warp：使用 T.warp_reduce_*。所有32个 lane 必须收敛执行，无效 lane 提供归约单位元。
- fragment/tile 归约：使用 T.reduce_*。编译器根据 source fragment layout 和归约维，推导本地归约以及必要的跨线程通信。
- 在 T.Parallel 中声明式累计：使用 T.alloc_reducer 和 T.finalize_reducer。
- partial 跨多个 CTA：需要 atomic 或多阶段归约。

3.3 Reducer、Replication



其中着重分析一下声明式reducer。

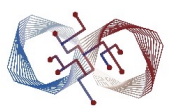
T.alloc_reducer 声明带 reduction operation 和 replication 元数据的 fragment reducer，而不是直接声明 CTA collective。编译过程中：LayoutReducer 根据 replication 建立 reducer layout，T.Parallel layout 决定每个 partial contribution 写入哪个线程副本；FinalizeReducerOp::InferLayout 仅保留该 layout， Lower 根据其 ReplicateExtent 和目标后端物化通信。

- replication="none"：每个逻辑 reducer 元素只有一个确定的线程 owner，ReplicateExtent=1。所有 contribution 必须已由该owner 本地累计，finalize_reducer lower 为 no-op，不会合并其他线程上的 partial values。

如果用声明式 reducer 表达：

```
sq_sum = T.alloc_reducer(
    (block_m,),
    T.float32,
    op="sum",
    replication="all",
)

T.fill(sq_sum, 0.0)
for i, j in T.Parallel(block_m, hidden):
    sq_sum[i] += x_frag[i, j] * x_frag[i, j]
T.finalize_reducer(sq_sum)
```

3.3 Reducer、Replication



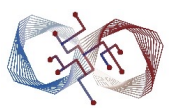
- replication="all"：current thread range 中每个线程持有覆盖完整 logical shape 的本地 partial 副本。T.fill 将其初始化为归约单位元，T.Parallel mapping 决定各线程产生哪些 contribution；finalize_reducer 对每个逻辑元素跨整个 current thread range 执行 AllReduce，完成后各线程才获得完整归约结果。

因此：ReplicateExtent 直接决定 finalize_reducer 是否通信及其通信宽度，而 T.Parallel mapping 决定 partial contribution的线程分布。当前 lowering 仅支持：ReplicateExtent = 1 或 current thread range extent 。代表 finalize 为 no-op 或执行全 range AllReduce。

如果用声明式 reducer 表达：

```
sq_sum = T.alloc_reducer(
    (block_m,),
    T.float32,
    op="sum",
    replication="all",
)

T.fill(sq_sum, 0.0)
for i, j in T.Parallel(block_m, hidden):
    sq_sum[i] += x_frag[i, j] * x_frag[i, j]
T.finalize_reducer(sq_sum)
```

3.3 Reducer、Replication



结论：声明式reducer不会根据实际贡献线程自动缩小为任意 subgroup collective。因此，alloc_reducer 并非 CTA reducer 的同义词；但若 current thread range 包含完整的128个 CTA 线程，replication="all" 当前就会生成128-thread AllReduce。

如果用声明式 reducer 表达：

```
sq_sum = T.alloc_reducer(
    (block_m,),
    T.float32,
    op="sum",
    replication="all",
)

T.fill(sq_sum, 0.0)
for i, j in T.Parallel(block_m, hidden):
    sq_sum[i] += x_frag[i, j] * x_frag[i, j]
T.finalize_reducer(sq_sum)
```

下面以 T.alloc_reducer + T.finalize_reducer 为例，按照编译流水线梳理其从 Python DSL 到 CUDA 代码的转换过程，包括Python 前端构造、LayoutReducer、LayoutInference、LowerTileOp/FinalizeReducer lowering，以及最终生成的 CUDA 代码。

3.3 Lowering一览

1. Python 前端

alloc_reducer 做两件事。1. 分配一个 local.fragment buffer； 2. 在当前 SBlock 上附加 reducer 元数据。

所以 sq_sum在前端 IR 中表示为：一个 local.fragment buffer + reducer_info { op = "sum", rep = "all" }

```python
def alloc_reducer(shape: ShapeType, dtype: DType, op: ReducerOp = "sum", replication=None) -> Buffer:
    """
    Allocate a reducer buffer.

    Modifications needs to conform with `op`,
    such as `op="sum"` requires `reducer[...] += ...` and
    `op="max"` requires `reducer[...] = T.max(reducer[...], ...)`.

    Only after T.fill with proper initializer the reduction may begin;
    only after T.finalize_reducer the partial results will be available.

    For `op="sum"`, filled value must be 0; for min and max, the filled initializer will become max or min clamper correspondingly.
    You may want to use `T.max_value` for min and `T.min_value` for max.

    Args:
        shape (tuple): The shape of the buffer to allocate
        dtype (str): The data type of the buffer (e.g., 'float32', 'int32')
        op (str): The reduce operation corresponded with the reducer
        replication (str | None): Replication strategy, can be "all" or "none". Defaults to not specified, and the compiler will do whatever it want.

    Returns:
        T.Buffer: A TVM buffer object allocated in thread-private storage, available to reduce values in T.Parallel loops.
    """

    assert op in ["sum", "max", "min"]
    # TODO: support automatic layout
    if replication is None:
        replication = "none"
    assert replication in ["all", "none"]

    reducer = T.sblock_alloc_buffer(shape, dtype, scope="local.fragment")
    sblock_attr({"reducer_info": {reducer.data: {"rep": replication, "op": op}}})

    return reducer
```

3.3 Lowering一览

1. Python 前端

finalize_reducer 只生成高层 TileOp intrinsic。 因此前端初始 IR 近似为：

T.finalize_reducer(sq_sum_region)

```python
def finalize_reducer(reducer: tirx.Buffer, batch: int = 1) -> tirx.PrimExpr:
    """
    Finalize a reducer buffer by emitting the `tl.tileop.finalize_reducer` intrinsic.

    This returns a TVM `tirx.Call` handle that finalizes the given reducer using its writable pointer.
    The call does not modify Python objects directly; it produces the low-level intrinsic call used by the IR.

    Parameters:
        reducer (tirx.Buffer): Reducer buffer whose writable pointer will be finalized.
        batch (int): Batch size for the AllReduce call (default 1 = scalar path,
            matching the T.reduce default).  When batch > 1, the compiler emits a
            single batched AllReduce call covering `batch` output elements at a
            time, reducing barrier count by batch×.  batch must evenly divide the
            total number of per-thread output elements.

    Returns:
        tirx.Call: Handle to the finalize reducer intrinsic call.
    """
    if batch < 1:
        raise ValueError(f"finalize_reducer: batch must be >= 1, got {batch}")
    annotations = {}
    if batch > 1:
        annotations["batch"] = batch
    return tirx.call_intrin(
        "handle",
        tirx.op.Op.get("tl.tileop.finalize_reducer"),
        to_tile_region(reducer, access_type="W"),
        annotations=annotations if annotations else None,
    )
```

3.3 Lowering一览

2. LayoutReducer： 界定归约区域并建立 reducer layout。layout_reducer.cc 会：1. 从 SBlock 的 reducer_info annotation 读取 reducer 的 op 和 replication。 2. 遇到作用于 reducer 的 tl.tileop.fill 时，将其标记为进入 reduction region。T.clear 在前端已展开为 fill(..., 0)。3. 为该 region 内最外层的 T.Parallel loop nest 添加 reducer_info annotation，并依据 replication 建立 reducer 的 Fragment layout，写入 layout_map。4. 遇到匹配的 T.finalize_reducer 时结束 region，并将 reduction operation 的整数编码追加到 intrinsic 参数中。

```
Stmt VisitStmt_(const SBlockNode *op) final {
  // Record annotations
  if (op->annotations.count(attr::kReducerInfo)) {
    auto map = op->annotations.Get(attr::kReducerInfo)
                   ->as<Map<Var, Map<String, String>>>();
    ICHECK(map) << "reducer_replication map is not defined";
    for (auto &&[var, rep] : map.value()) {
      reducer_info_map_.Set(
          var, ReducerInfo{rep.Get("op").value(), rep.Get("rep").value()});
    }
  }
  for (auto &&buffer : op->alloc_buffers) {
    var_to_buffer_.Set(buffer->data, buffer);
  }
  auto result = IRMutatorWithAnalyzer::VisitStmt_(op).as<SBlock>().value();
  // After iterating over the body, set all layout_map to block
  auto p_result = result.CopyOnWrite();
  Map<Var, Layout> layout_map;
  if (auto layout_map_ref = p_result->annotations.Get(attr::kLayoutMap)) {
    if (auto maybe_layout_map =
            layout_map_ref.value().as<Map<Var, Layout>>()) {
      layout_map = maybe_layout_map.value();
    }
  }
  for (auto &&[k, v] : new_layout_map_)
    layout_map.Set(k, v);
  if (!layout_map.empty())
    p_result->annotations.Set(attr::kLayoutMap, layout_map);
  new_layout_map_.clear();
  return result;
}
```

3.3 Lowering一览

2. LayoutReducer：当前实现会改写为 T.finalize_reducer(row_sum, 0)，其中 0 是当前 ReducerOpType::SUM 的编码。对于 replication=“all”，LayoutReducer 构造：

Fragment::FullyReplicated(buffer->shape, thread_extent)

如果 row_sum.shape=[128]、当前线程范围为 256，则其布局可表示为：

logical shape = [128]

forward_thread = replica_id

forward_index = row

ReplicateExtent = 256

(row, replica_id)→ (thread_id=replica_id, local_slot=row)

```
Stmt VisitStmt_(const SBlockNode *op) final {
  // Record annotations
  if (op->annotations.count(attr::kReducerInfo)) {
    auto map = op->annotations.Get(attr::kReducerInfo)
                   ->as<Map<Var, Map<String, String>>>();
    ICHECK(map) << "reducer_replication map is not defined";
    for (auto &&[var, rep] : map.value()) {
      reducer_info_map_.Set(
          var, ReducerInfo{rep.Get("op").value(), rep.Get("rep").value()});
    }
  }
  for (auto &&buffer : op->alloc_buffers) {
    var_to_buffer_.Set(buffer->data, buffer);
  }
  auto result = IRMutatorWithAnalyzer::VisitStmt_(op).as<SBlock>().value();
  // After iterating over the body, set all layout_map to block
  auto p_result = result.CopyOnWrite();
  Map<Var, Layout> layout_map;
  if (auto layout_map_ref = p_result->annotations.Get(attr::kLayoutMap)) {
    if (auto maybe_layout_map =
            layout_map_ref.value().as<Map<Var, Layout>>()) {
      layout_map = maybe_layout_map.value();
    }
  }
  for (auto &&[k, v] : new_layout_map_)
    layout_map.Set(k, v);
  if (!layout_map.empty())
    p_result->annotations.Set(attr::kLayoutMap, layout_map);
  new_layout_map_.clear();
  return result;
}
```

3.3 Lowering一览

即每个线程都有覆盖全部 128 个逻辑元素的本地 reducer 副本。T.fill 后副本保存单位元，归约循环后保存各线程的 partial values；只有执行 finalize_reducer 后，各线程副本才成为完整归约结果。

```
Stmt VisitStmt_(const SBlockNode *op) final {
  // Record annotations
  if (op->annotations.count(attr::kReducerInfo)) {
    auto map = op->annotations.Get(attr::kReducerInfo)
                   ->as<Map<Var, Map<String, String>>>();
    ICHECK(map) << "reducer_replication map is not defined";
    for (auto &&[var, rep] : map.value()) {
      reducer_info_map_.Set(
          var, ReducerInfo{rep.Get("op").value(), rep.Get("rep").value()});
    }
  }
  for (auto &&buffer : op->alloc_buffers) {
    var_to_buffer_.Set(buffer->data, buffer);
  }
  auto result = IRMutatorWithAnalyzer::VisitStmt_(op).as<SBlock>().value();
  // After iterating over the body, set all layout_map to block
  auto p_result = result.CopyOnWrite();
  Map<Var, Layout> layout_map;
  if (auto layout_map_ref = p_result->annotations.Get(attr::kLayoutMap)) {
    if (auto maybe_layout_map =
            layout_map_ref.value().as<Map<Var, Layout>>()) {
      layout_map = maybe_layout_map.value();
    }
  }
  for (auto &&[k, v] : new_layout_map_)
    layout_map.Set(k, v);
  if (!layout_map.empty())
    p_result->annotations.Set(attr::kLayoutMap, layout_map);
  new_layout_map_.clear();
  return result;
}
```

3.3 Lowering一览

3. LayoutInference：推导 contribution ownership

LayoutInference 对推导队列中的 TileOp 调用 InferLayout，在不同算子之间传播并补全 Fragment layout。 FinalizeReducerOpNode::InferLayout 本身不推导新的布局，只返回 reducer 已有的 layout：

layout_map.Set(reducer, layout_args.layout_map.Get(reducer).value());

因此，reducer 的 fully replicated layout 已由 LayoutReducer 确定。LayoutInference 在这里真正需要确定的是 x_frag 的 Fragment layout 以及 reduction region 对应的 T.Parallel loop layout，即每个 (row, col) iteration 由哪个线程执行

```
// Run InferLayout
auto updates = next->InferLayout(LayoutInferArgs{target_,
                                                  thread_bounds,
                                                  layout_map,
                                                  cur_analyzer,
                                                  buffer_oob,
                                                  {},
                                                  bind_var_to_expr_,
                                                  false},
                                 level);
```

```
/**
 * @brief Infer and return the layout mapping for the reducer buffer.
 *
 * Copies the existing layout for the reducer from the provided LayoutInferArgs
 * into a new LayoutMap and returns it. The inference does not modify the
 * layout; it preserves the reducer's current layout.
 *
 * @param layout_args Provides the input layout map from which the reducer's
 * layout is copied.
 * @param level Unused by this operator; present for API compatibility.
 * @return LayoutMap A map that contains the reducer buffer mapped to its
 * original layout.
 */
LayoutMap FinalizeReducerOpNode::InferLayout(const LayoutInferArgs &layout_args,
                                             InferLevel level) const {
  LayoutMap layout_map;
  layout_map.Set(reducer, layout_args.layout_map.Get(reducer).value());
  return layout_map;
}
```

3.3 Lowering一览

3. LayoutInference：推导 contribution ownership

对于 replication="all"，ParallelOp::InferLayout 还会明确跳过 reducer，不使用它作为 loop layout 的推导来源；loop layout 通常从 x_frag 等其他 Fragment access 推导，或者由自由布局规划器生成。

```
// Step 1: try to infer loop's partition from a source fragment
Buffer source_buffer, read_source_buffer;
bool source_buffer_is_write = false;
Buffer replicated_write_buffer; // Backup: fully replicated write buffer

for (const auto &buffer : access_order_) {
  const auto &access = GetAccessInfo(buffer);
  if (layout_args.layout_map.count(buffer)) {
    // skip reducers with rep=ALL
    if (auto info = reducer_info_map_.Get(buffer->data);
        info && info.value()->rep == ReducerRepType::ALL)
      continue;
```

```
// Try to infer loop layout from buffers in order of preference only if we
// don't already have a layout (e.g., from annotations):
// 1. Annotated loop layout
// 2. Non-replicated write buffer (most reliable)
// 3. Non-replicated read buffer
// 4. Fully replicated write buffer (backup, may cause issues)
// 5. Free inference mode (no source buffer)
if (!loop_layout_.defined() && annotated_layout_unbound_.defined()) {
  loop_layout_ = annotated_layout_unbound_.value()->BindThreadRange(
      layout_args.thread_bounds);
  loop_layout_requires_padding_guard_ = annotated_requires_padding_guard_;
  if (annotated_predicate_.defined()) {
    predicate_ = annotated_predicate_.value();
  }
} else if (!loop_layout_.defined() && source_buffer.defined() &&
           (allow_layout_propgate || source_buffer_is_write)) {
  loop_layout_ = ComputeLoopLayoutFromBuffer(source_buffer, layout_args);
} else if (!loop_layout_.defined() && level == InferLevel::kFree) {
  // For free layout inference
  // In free inference, try two mechanisms and prefer the one that
  // minimizes replication while remaining compatible:
  // 1) compute_loop_layout_from_buffer (always correct but may
  // over-replicate) 2) PlanLoopPartition (often smaller replication)
  Fragment candidate_from_buffer;
  Fragment candidate_from_plan;
  bool selected_plan_candidate = false;

  if (read_source_buffer.defined() && allow_layout_propgate) {
    candidate_from_buffer =
        ComputeLoopLayoutFromBuffer(read_source_buffer, layout_args);
  }

  // try to infer loop layout with two mechanisms and choose the best one
  {
    candidate_from_plan = ComputePlanCandidate(layout_args);
  }

  // Choose the best candidate:
  if (candidate_from_buffer.defined() && candidate_from_plan.defined()) {
    loop_layout_ = ChooseBestCandidate(candidate_from_buffer,
                                       candidate_from_plan, layout_args);
  } else if (candidate_from_plan.defined()) {
    loop_layout_ = candidate_from_plan;
    selected_plan_candidate = true;
    DLOG(INFO) << "[FreeInfer] only PlanLoopPartition available, choose it.";
  } else if (candidate_from_buffer.defined()) {
    loop_layout_ = candidate_from_buffer;
    DLOG(INFO)
        << "[FreeInfer] only compute_from_buffer available, choose it.";
  }
  loop_layout_requires_padding_guard_ =
```

3.3 Lowering一览

3. LayoutInference：推导 contribution ownership

本例推导出的 iteration mapping 为：

thread_id = (row % 16) * 16 + col // 4

local_id = (row // 16) * 4 + col % 4

这组映射描述 x_frag[row, col] 的线程 ownership：每行的 64 个元素分布在 16 个线程上；每个线程负责该行连续的 4 个元素；每个线程共持有 8 × 4 = 32 个 x_frag 元素；对于给定 row，只有对应的 16 个线程更新各自的 sq_sum[row]；其余 240 个线程的该槽位未被更新，因先前的 T.fill(..., 0) 而保持求和单位元。需要注意，sq_sum 自身仍采用 fully replicated layout： (row, replica_id) → (thread_id=replica_id, local_slot=row) 因此，每个线程都有完整的128个 sq_sum 本地槽位，但在 finalize 前，这些槽位只是该线程的 partial value 或单位元。后续 finalize_reducer 才会对每个 row 跨 256 个副本执行 AllReduce。

3.3 Lowering一览

4. Lowering：生成后端 AllReduce 调用。

LowerTileOp 遇到Evaluate(tl.tileop.finalize_reducer(...))

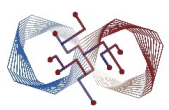
首先通过 ParseOperator 恢复 FinalizeReducerOp，然后构造 LowerArgs。

其中 buffer_remap 将逻辑 fragment buffer 映射为按照 layout->OutputShape() 分配的线程本地 buffer；add_workspace 用于申请 shared.dyn workspace。

```
LowerArgs lower_args;
lower_args.target = target_;
lower_args.thread_bounds = thread_bounds;
lower_args.thread_var = thread_var_->var;
lower_args.layout_map = layout_map_;
lower_args.buffer_remap = buffer_remap_;
lower_args.bind_var_to_expr = bind_var_to_expr;
lower_args.mbar_phase_expr = loop_mbar_phase_stack_.empty()
                             ? PrimExpr(IntImm(DataType::Int(32), 0))
                             : loop_mbar_phase_stack_.back();
lower_args.mbarrier_buffer = &mbarrier_buffer_;
lower_args.cluster_size = cluster_size_;
lower_args.add_workspace = add_workspace_callback;
lower_args.alloc_mbarrier = mbarrier_callback;
lower_args.update_barrier_arrive = barrier_arrive_callback;
lower_args.require_smem_alignment = require_smem_alignment_callback;

auto lowered = tile_op->Lower(lower_args, analyzer_);

return IRMutatorWithAnalyzer::VisitStmt(lowered);
```

```
static Buffer makeBufferWithLayout(const Buffer &buffer, const Layout &layout,
                                   Map<Var, Var> &var_remap) {
  const auto *ptr_type =
      TVM_TYPE_AS(buffer->data->type_annotation, PointerTypeNode);
  Type new_type;
  // convert fragments to normal local buffer
  if (IsFragmentBuffer(buffer)) {
    new_type = PointerType(ptr_type->element_type, "local");
  } else {
    new_type = buffer->data->type_annotation;
  }
  Var new_var;
  if (IsGlobalBuffer(buffer)) {
    new_var = buffer->data;
  } else {
    if (var_remap.count(buffer->data)) {
      new_var = var_remap[buffer->data];
    } else {
      new_var = Var(buffer->data->name_hint, new_type);
      var_remap.Set(buffer->data, new_var);
    }
  }
  Array<PrimExpr> layout_shape = layout->OutputShape();
  Array<PrimExpr> output_shape = layout_shape;
```

3.3 Lowering一览



4. Lowering：生成后端 AllReduce 调用。

FinalizeReducerOpNode::Lower 根据 target 从后端注册表中选择实现。CUDA target 使用公共的 FinalizeReducerLowerer。

公共 lowerer 首先读取 reducer 的 Fragment layout并检查：extent == 1 || extent == current_thread_range_extent

随后分两种情况：ReplicateExtent = 1→ return Evaluate(0)，finalize 为 no-op

ReplicateExtent = current thread range extent → 生成全 range AllReduce

```
template <typename Impl> struct FinalizeReducerLowerer {
  static Stmt Lower(const FinalizeReducerOpNode &op,
                    const LowerArgs &lower_args, arith::Analyzer *) {
    auto buffer = lower_args.buffer_remap[op.reducer];
    auto opt_layout = lower_args.layout_map.Get(op.reducer);
    ICHECK(opt_layout);
    ICHECK(opt_layout->as<Fragment>());
    auto layout = opt_layout->as<Fragment>().value();
    Array<PrimExpr> indices_0;
    indices_0.reserve(layout->OutputDim());
    for (int i = 0; i < layout->OutputDim(); ++i) {
      indices_0.push_back(Var("__finred_" + std::to_string(i)));
    }

    const int64_t *p_extent = as_const_int(layout->ReplicateExtent());
    ICHECK(p_extent);
    int extent = *p_extent;
    ICHECK(extent == 1 ||
           extent == *as_const_int(lower_args.thread_bounds->extent))
        << "Illegal finalize_reducer: extent=" << extent
        << "; T.thread_bounds=" << lower_args.thread_bounds;

    if (extent == 1) {
      return Evaluate(0);
    }
```

3.3 Lowering一览

4. Lowering：生成后端 AllReduce 调用。

默认 batch=1 时，lowerer 构造对 layout-OutputShape() 的 parallel loop，并为每个线程本地输出槽位生成一次 call_extern(AllReduceName, buffer[index], ...)。

```
std::string allreduce = Impl::MakeScalarAllReduce(
    op_str, reducing_threads, 1, thread_offset,
    lower_args.thread_bounds->extent, lower_args.target);
Array<PrimExpr> thread_reduce_args = {StringImm(allreduce),
                                      BufferLoad(buffer, indices_0)};
if (reducing_threads > Impl::WarpSize(lower_args.target)) {
  PrimExpr workspace = lower_args.add_workspace(
      *as_const_int(lower_args.thread_bounds->extent), buffer->dtype);
  thread_reduce_args.push_back(workspace);
}
auto call = Call(buffer->dtype, builtin::call_extern(), thread_reduce_args);
Stmt body = BufferStore(buffer, call, indices_0);

for (int i = layout->OutputDim() - 1; i >= 0; i--) {
  body = For(indices_0[i].as<Var>().value(), 0, layout->OutputShape()[i],
             ForKind::kParallel, body);
}

return body;
```

当 ReplicateExtent 超过目标 warp size 时，通过 LowerTileOp 提供的callback 申请包含 current_thread_range_extent 个 reducer-dtype 元素的 shared.dyn workspace，并将其作为 AllReduce 参数。

```
AddWorkspaceCallback add_workspace_callback = [this](int num_elem,
                                                     DataType dtype) {
  auto workspace =
      decl_buffer({PrimExpr(num_elem)}, dtype, "workspace", "shared.dyn");
  // Record workspace under the innermost block scope so its lifetime
  // covers the statements that requested it and does not sink into
  // subsequently created inner blocks (e.g., GEMM macro blocks).
  if (!workspace_stack_.empty()) {
    workspace_stack_.back().push_back(workspace);
  } else {
    // Fallback: create a temporary frame (should be rare)
    workspace_stack_.emplace_back(Array<Buffer>{workspace});
  }
  return workspace.access_ptr(2); // write
};
```

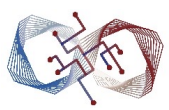
3.3 Lowering一览

4. Lowering：生成后端 AllReduce 调用。

对于本例，logical shape=[128]、OutputShape=[128]、ReplicateExtent=256。LowerTileOp 申请 256 个 float32 元素的 shared.dyn workspace，并为每个本地输出槽位生成：

```
for row in T.Parallel(128):
    row_sum[row] = T.call_extern(
        "float32",
        "tl::AllReduce<tl::SumOp, 256, 1, 0, "
        "tl::NamedBarrier<256>>::run",
        row_sum[row],
        workspace,
    )
```

3.3 Lowering一览



LayoutReducer 根据 replication=“all” 建立 fully replicated reducer layout；LayoutInference 推导 reduction region 中 partial contribution 的线程映射。而 FinalizeReducerOp::InferLayout 仅保留既有 reducer layout；LowerTileOp 随后读取其 ReplicateExtent，为每个本地输出槽位生成具体的 CUDA tl::AllReduce<...>::run 调用。最后，CUDA backend 根据 reduction operation、参与线程范围和 target 构造 AllReduce 模板名称。

3.3 Lowering一览

5. CUDA target 生成模板调用

本例参数为：

op = tl::SumOp

reducing_threads = 256

scale = 1

thread_offset = 0

all_threads = 256

target = SM90

```
  static std::string MakeScalarAllReduce(std::string reducer,
                                         int reducing_threads, int scale,
                                         PrimExpr thread_offset,
                                         PrimExpr all_threads, Target target) {
    std::stringstream ss;
    ss << "tl::AllReduce<" << reducer << ", " << reducing_threads << ", "
       << scale << ", " << thread_offset;
    if (TargetHasSMVersionGE(target, 90)) {
      ss << ", tl::NamedBarrier<" << all_threads << ">";
    }
    ss << ">::run";
    return ss.str();
  }
};
```

SM90+ 会选择 NamedBarrier<all_threads> 因此生成：

tl::AllReduce<tl::SumOp, 256, 1, 0, tl::NamedBarrier<256>>::run

3.3 Lowering一览

5. CUDA target 生成模板调用

编译后生成 CUDA 为：

```
for (int row = 0; row < 128; ++row) {row_sum[row] = tl::AllReduce< tl::SumOp, 256, 1, 0,
tl::NamedBarrier<256>>::run(row_sum[row], &workspace[0]); }
```

tl::SumOp	归约操作
256	归约线程跨度；scale=1 时参与线程数为256
1	相邻逻辑参与者的 thread-id 步长
0	参与线程范围的起始偏移
NamedBarrier<256>	SM90+ 使用的256线程同步策略

CTA 中的 256 个线程都会执行该 row 循环；对于每个 row，所有线程分别提交自己的本地 partial value，并共同完成一次 256-thread AllReduce。

3.3 Lowering一览

6. AllReduce 内部执行过程

AllReduce 使用递归 XOR butterfly。 对于 threads=256, scale=1：

offset 128：shared memory + NamedBarrier

offset 64：shared memory + NamedBarrier

offset 32：shared memory + NamedBarrier

offset 16：shfl_xor_sync

offset 8：shfl_xor_sync

offset 4：shfl_xor_sync

offset 2：shfl_xor_sync

offset 1：shfl_xor_sync

完成后，参与 collective 的 256 个线程都得到相同的完整归约结果。

3.3 Lowering一览

6. AllReduce 内部执行过程

```
// AllReduce performs a cross-thread reduction over a group of `threads`
// threads.
//
// Template parameters:
//   Reducer          - binary reduction functor (e.g. SumOp, MaxOp).
//   threads          - number of threads that span the reduce dimension,
//                      equal to extent * scale.
//   scale            - stride of participating threads in the thread index
//                      space. When the thread-to-data mapping is normalized as
//                        threadIdx = source * scale + ...
//                      `scale` is the stride between consecutive logical
//                      participants in the reduce dimension.
//                      The recursion terminates when threads == scale, meaning
//                      each reduce group has been collapsed to a single thread.
//                      Uses a recursive XOR-butterfly pattern: at each level,
//                      offset >= 32 goes through shared memory + barrier,
//                      offset < 32 uses warp shuffle (shfl_xor_sync).
//   thread_offset    - base thread index offset within the block.
//   Barrier          - barrier policy type (SyncThreadsBarrier or
//                      NamedBarrier<N>).
//   batch_size       - number of independent values to reduce in parallel,
//                      sharing synchronization barriers across all values.
//                      Default 1 preserves the original scalar behaviour.
//   workspace_stride - stride between per-channel slices in the shared-memory
//                      workspace (typically total threads in the block).
//                      Only used when batch_size > 1.
```

```
private:
  using Next = AllReduce<Reducer, threads / 2, scale, thread_offset, Barrier,
                         batch_size, workspace_stride>;

  template <typename T>
  static TL_DEVICE T butterfly_reduce_scalar(T x, T *red_buf) {
    constexpr int offset = threads / 2;
    if constexpr (offset >= 32) {
      Barrier::template sync<1>();
      red_buf[threadIdx.x - thread_offset] = x;
      Barrier::template sync<2>();
      x = Reducer()(x, red_buf[(threadIdx.x - thread_offset) ^ offset]);
    } else {
      x = Reducer()(x, tl::shfl_xor_sync(uint32_t(-1), x, offset));
    }
    if constexpr (offset == scale) {
      return x;
    } else {
      return Next::run(x, red_buf);
    }
```

3.4 Replication

T.Fragment(..., replicate=R) 为 Fragment layout 引入复制坐标 r ∈ [0,R):

(logical_indices, r) → (thread_id, thread_local_index)

因此，同一逻辑元素具有 R 个物理实例；每个 (logical_indices, r) 对应一个确定的物理 owner。forward_fn(..., r) 决定各副本具体落在哪个线程和线程本地槽位。

Fragment replication 描述“同一逻辑值有多少个物理实例以及它们放在哪里”；reducer replication 则是基于该机制定义的两种受限策略。(ReplicateExtent=1 或完整 current thread range)

接口	表达什么
`T.Fragment(..., replicate=R)`	通用 fragment layout 的 replica dimension
`T.alloc_reducer(..., replication="all")`	reducer 最终结果在所有线程上 fully replicated

两者都涉及"多份本地可见值"，但一个是 layout 构造能力，一个是 reducer 的声明式结果策略。

Recall : Warp scheduler

H100 的每个 SM 维护多个 resident warps，其执行上下文（PC、寄存器、谓词和状态）始终驻留在片上，因此无需像 CPU 一样进行寄存器保存与恢复。

Warp scheduler 每个周期根据 scoreboard 从 Ready Warps 中选择一条可发射（issue）的指令；当某个 warp 因 memory dependency 或数据未就绪而 stall 时，scheduler 会立即切换到其他 Ready warps 发射指令。

Bottom line: The two best things about GPUs.

1. SIMD is great.

But wait, there's more!

2. Hyperthreading to the extreme.
   - Warps time-slice usage of all processor pipeline resources, to hide each other's latency and keep hardware resources busy.
   - "Context switch" between warps is "free" from SW perspective.
     - Context is always resident on processor.
     - Switch is implemented in hardware.

Recall : Occupancy

这种近乎零开销的硬件调度（hardware warp switching）通过在多个 warp 间交替发射指令来隐藏访存和执行延迟，使计算单元持续保持高利用率。

我们无法控制每个周期发射哪条指令，但可以通过提高 occupancy、增加 ILP、拉大 load-use 距离以及软件流水（software pipelining）增加 Ready work，从而提升 scheduler 隐藏 latency 的能力。

（个人理解：GPU 的 latency hiding 本质上是利用时分复用将多个 warp 的执行交织到同一条硬件流水线上，使一个 warp 的等待时间成为另一个 warp 的执行时间。）

Occupancy（占用率） 描述的是 SM 上能够驻留（resident）的 warp 数量相对于硬件最大支持数量的比例。反映了 warp scheduler 可供选择的潜在工作量（work pool），而不是执行单元的实际利用率。

Recall : Occupancy

Occupancy: What is it? How is it determined?

- There is a maximum number of warps which can be concurrently active on an SM.
  - **Device** (depends on compute capability of the GPU)
  - **Achievable** (depends on kernel implementation + compiler)
  - **Achieved** (depends mostly on the grid size or workload behavior. Such as workload imbalance.)

$$Occupancy = \frac{Achievable\ \#\ active\ warps\ per\ SM}{Device\ \#\ active\ warps\ per\ SM}$$

- Achievable Occupancy of a CUDA kernel will be limited by at least one of several factors.
  - SM resource assignment:
    - For example: Shared memory and Registers must be partitioned among threads.
    - Block size
  - Other hardware factors:
    - For example: Max blocks per SM, Max warps per SM etc.
    - Outlined in Table 23 Technical Specifications per Compute Capability

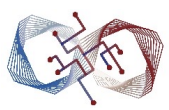
4.1 T.Pipelined

```
for ko in T.Pipelined(num_k_tiles, num_stages=3):
    T.copy(A[ko], A_shared)
    T.copy(B[ko], B_shared)
    T.gemm(A_shared, B_shared, C_fragment)
```

T.Pipelined 是用户可见的软件流水循环构造。用户描述单个逻辑 iteration 中的操作及数据流；自动模式下，编译器分析 producer-consumer 依赖并规划 stage/order，手动模式下则由用户显式提供调度信息。

编译器将循环改写为 prologue → steady state → epilogue，完成逻辑 iteration 重映射；对存活区间重叠的中间 buffer 进行multi-versioning，并通过取模选择循环使用的版本。对于后端支持的异步操作，还会处理 producer grouping、commit/wait 以及必要的 barrier。

4.1 Case Study: GEMM stage sweep



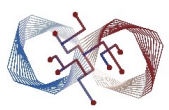
增大 num_stages 通常会扩大数据发起与消费之间的迭代距离，使更多 iteration 同时在途，从而提高延迟隐藏能力；代价可能包括更多 shared-memory versions、更长的 live range 和更高的寄存器压力、更长的 prologue/epilogue，以及由资源占用增加导致的 occupancy 下降。实际 buffer version 数量由存活区间和访问冲突分析决定，不一定等于 num_stages。

Case Study: 4096^3 GEMM、128×128×32 tile、128-thread 配置在 H100 shape/configuration 的结果，最优 stage 数sweep: 1 2 3 4 5 6 7。

检查：latency 最低点、generated code 中 buffer versions 和 prologue/epilogue 、结合 ptxas registers/shared memory 判断资源压力、用 NCU stall/pipe counters 做更细归因。

在这个例子中，stage 5 / 6 性能是相对最优的。

5.1 Persistent



T.Pipelined 描述时间调度：在一个循环内部重叠不同逻辑 iteration 的 producer、数据搬运和 consumer，通过多版本 buffer、异步操作及同步来隐藏延迟。

而 T.Persistent 描述空间任务调度：用有限数量的物理 worker CTA，以 grid-stride 方式在生命周期内依次处理更大的逻辑 tile域。物理 CTA 数等于 SM 数是常见配置，并非 persistent 的定义。

普通 grid：

```
logical tile 0 → CTA 0
logical tile 1 → CTA 1
...
```

Persistent schedule：

```
有限 worker CTA pool
  worker 0 → task 0 → task P → task 2P → ...
  worker 1 → task 1 → task P+1 → ...
```

```python
with T.Kernel(num_workers, threads=threads) as worker:
    for bm, bn in T.Persistent(
        [num_m_tiles, num_n_tiles], num_workers, worker
    ):
        compute_one_tile(bm, bn)
```

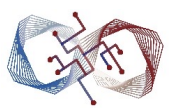
5.1 Persistent 与 MegaKernel

在 MegaKernel-style 多阶段融合中，persistent execution 使用有限数量、长期存活的 worker CTA 循环处理更大的逻辑 task域，使程序员能够更直接地设计 task 到 worker 的映射、worker 内的领取与执行顺序、CTA 内状态的跨 task 生命周期，以及shared/register 资源的复用与重置策略。结合显式任务队列、T.Pipelined、warp specialization 和同步原语，还可以组织 producer/consumer phase，并实现不规则任务的动态负载均衡。

代价是需要谨慎处理状态重置、尾部任务和同步；使用动态队列或 atomic 时，还需考虑队列争用与 determinism。长期存活的状态也可能增加寄存器和 shared-memory 占用，从而降低 occupancy。配置的 worker 数不等于实际同时驻留的 CTA 数，后者仍受硬件资源限制。

Persistent 提供的是更强的调度控制能力，而不是自动的性能提升。

5.1 Case Study: Engram backward



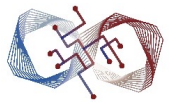
生产实现使用固定 worker CTA 数：

with T.Kernel(num_persistent_blocks, threads=256) as pid_p:

虽然没有直接使用 T.Persistent，但每个 worker CTA 通过显式循环或任务队列连续处理多个逻辑 task，因此具有相同的 persistent 调度语义。

```
@T.prim_func
def engram_gate_bwd_kernel(
    grad_out: T.Tensor[(num_tokens, hc_mult, hidden_size), T.bfloat16],
    hidden_states: T.Tensor[(num_tokens, hc_mult, hidden_size), T.bfloat16],
    k: T.StridedTensor[(num_tokens, hc_mult, hidden_size), (k_stride_s, k_stride_h, 1), T.bfloat16],
    v: T.StridedTensor[(num_tokens, hidden_size), (v_stride_s, 1), T.bfloat16],
    weight_fused: T.Tensor[(hc_mult, hidden_size), T.float],
    dot_in: T.Tensor[(num_tokens, hc_mult), T.float],
    gate_in: T.Tensor[(num_tokens, hc_mult), T.float],
    rstd_x_in: T.Tensor[(num_tokens, hc_mult), T.float],
    rstd_k_in: T.Tensor[(num_tokens, hc_mult), T.float],
    grad_x: T.Tensor[(num_tokens, hc_mult, hidden_size), T.bfloat16],
    grad_k: T.Tensor[(num_tokens, hc_mult, hidden_size), T.bfloat16],
    grad_v: T.Tensor[(num_tokens, hidden_size), T.bfloat16],
    grad_w_partial: T.Tensor[(num_persistent_blocks, hc_mult, hidden_size), T.float],
):
    with T.Kernel(num_persistent_blocks, threads=threads) as pid_p:
        tid = T.get_thread_binding()
        warp_id = T.get_warp_idx()
        lane_id = T.get_lane_idx()
        head_id = warp_id // warps_per_head
        sub_warp_id = warp_id % warps_per_head
```

5.1 Case Study: 长生命周期状态选择persistent



grad_w_local 在 worker CTA 的任务循环外分配，并在该 worker 处理的整个 token 集合期间保持存活。它是分布在 CTA 各线程 register/fragment 中的 worker-local partial state。每个 token 只更新该 partial；worker 完成后再按输出元素写出一次，最后由独立 reduction kernel 沿 worker 维合并。

因此，它避免了逐 token 更新全局 grad_w 的 atomic 竞争。这里 Persistent 的实质价值是建立稳定的 worker identity → long-lived partial state 归属关系。

```
grad_w_local = T.alloc_local((elems_per_warp_pair,), T.float32)
T.clear(grad_w_local)

for token in assigned_tokens:
    ...
    grad_w_local[...] += token_contribution

grad_w_partial[worker, head, ...] = grad_w_local[...]
```

`grad_w_local` 跨该 CTA 负责的整个 token range 存活。这样避免每个 token 都对全局 `grad_w` 做 atomic：

5.1 Persistent 与 Pipeline

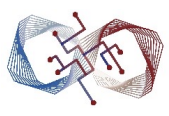
Workload 特征	首选结构
tile 独立、规则、数量充足	普通 grid
单个 CTA 内有规则 iteration-local producer/consumer	`T.Pipelined`
task 多于 worker、需跨 task 保存状态或自定义次序	Persistent
多阶段融合但任务仍规则独立	普通 MegaKernel 也完全可行
多阶段融合且需要跨 task state/queue/phase control	Persistent MegaKernel 更有控制力

选择 primitive 应从 lifetime 和 dependency 出发，而不是从"我想使用某个高级 feature"出发。

普通/Persistent 决定 task 如何映射到 CTA；

T.Pipelined 决定 CTA 内 iteration 如何在时间上重叠。

6.1 Profiling

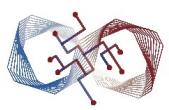


之前的Session，我们介绍了通过 Nsight Compute (NCU) 分析单个 kernel 的性能特征，包括occupancy、memory throughput、stall reason / pipe utilization counters；在更高的系统执行层面，我们还可以通过 Nsight Systems (NSYS) / PyTorch Profiler 从系统执行视角分析 host-side launch overhead、kernel timeline、多 kernel 依赖关系以及 CPU-GPU overlap。

不过现在TileLang集成了一种更先进的实验性预览 IKET （最早在CuteDSL中支持）

最后由于大模型的高速发展，我们还可以直接Dump SASS做分析。SASS 可以证明两个版本是否具有不同 lowering，例如：指令类别与数量不同；async copy / barrier sequence 不同；pipeline prologue/epilogue 不同；registers、shared memory 与代码体积不同。

参考资料 Reference



本文选择 TileKernels 中四个具有代表性的 Memory-Bound Kernel 作为分析对象（https://github.com/deepseek-ai/TileKernels ）以及一个额外的RMSNorm https://github.com/tile-ai/tilelang/blob/main/examples/norm/rms_norm.py 和最普通的GEMM。本次分享的API 与实验测试基于以下实验环境：H100 80GB、TileLang 0.1.12 与 Apache TVM-FFI 0.1.11。

TileLang在许多 memory bound 算子上表现很好，能达到接近人工优化 CUDA C，有效带宽达到理论HBM带宽（3.35TB/s）的75%-90% 。

部分图片来自GTC和NV OpenDay的分享（如果已过时，以最新的行为为准）

Documentation可参考： https://tilelang.com/ （TileLang 0.1.12）

目标：建立一套可以迁移到新算子的编程心智模型。