

Modern Programming Model for Writing Kernels on GPUs

罗新浩

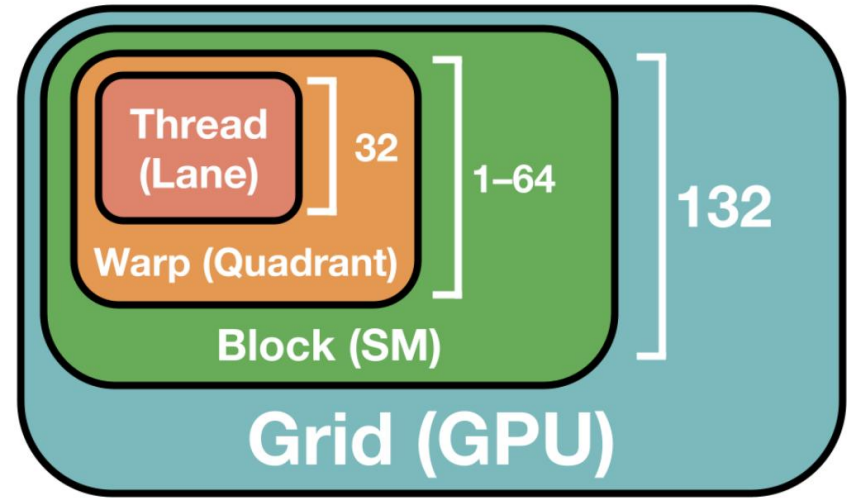
2025.5.30

Outline

- Fundamentals
- Motivation
- Prior work
- Tile-based programming model
 - TileLang(Compiler)
 - ThunderKittens(C++ embedded library, ICLR 2025)
- Discussion: Task-based programming model
 - Hidet(Compiler, ASPLOS 23)
 - Cypress(Compiler, PLDI 25)

Fundamentals

- GPU software hierarchy:
 - Thread.
 - Warp: consist of groups of 32 nearby threads.
 - Block: groups of warps which together execute on a core.
- Each thread -> per-thread register file
- All threads within a block -> shared memory
- All blocks -> global memory






Motivation

- Writing high-performance kernels on modern GPUs is very complex!
 - Thread binding: allocation of tasks across blocks, warps, and threads.
 - Memory layout: organization of data in global/shared/register memory.
 - Pipeline: overlap data movement with computation.
- Hardware-centric optimization
 - Hopper: TMA, WGMMA
 - Blackwell: Tensor memory, tcgen05(no WGMMA)

[flashinfer](#) / [include](#) / [flashinfer](#) / [attention](#) / 

 yzh119 bugfix: fix fp8 attention kernels aot compilation issue ([#1087](#)) 

Name	Last commit message
 ..	
 blackwell	bugfix: follow us
 hopper	bugfix: fix fp8 at

FlashInfer: Attention kernel library

Triton

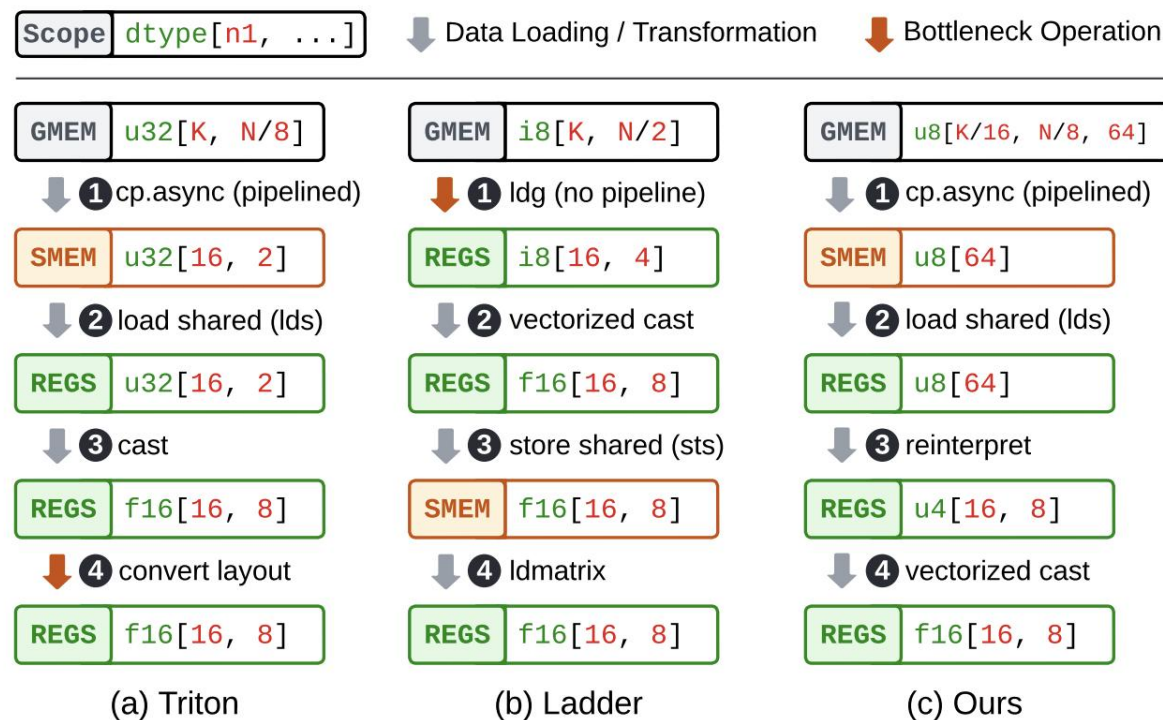
- Kernels are defined as decorated Python functions.
- Triton abstracts away all the issues related to thread blocks (memory coalescing, shared memory synchronization/conflicts).

```
@triton.jit
def matmul(A, B, C, M, N, K, stride_am, stride_ak,
          stride_bk, stride_bn, stride_cm, stride_cn,
          **META):
    # extract metaparameters
    BLOCK_M, GROUP_M = META['BLOCK_M'], META['GROUP_M']
    BLOCK_N = META['BLOCK_N']
    BLOCK_K = META['BLOCK_K']
    # programs are grouped together to improve L2 hit rate
    _pid_m = tl.program_id(0)
    _pid_n = tl.program_id(1)
    pid_m = _pid_m // GROUP_M
    pid_n = (_pid_n * GROUP_M) + (_pid_m % GROUP_M)
    # rm (resp. rn) denotes a range of indices
    # for rows (resp. col) of C
    rm = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    rn = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    # rk denotes a range of indices for columns
    # (resp. rows) of A (resp. B)
    rk = tl.arange(0, BLOCK_K)
    # the memory addresses of elements in the first
    block of
```

```
    # A and B can be computed using numpy-
    style broadcasting
    A = A + (rm[:, None] * stride_am + rk[None, :]
    * stride_ak)
    B = B + (rk[:, None] * stride_bk + rn[None, :]
    * stride_bn)
    # initialize and iteratively update accumulator
    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
    for k in range(K, 0, -BLOCK_K):
        a = tl.load(A)
        b = tl.load(B)
        # block level matrix multiplication
        acc += tl.dot(a, b)
        # increment pointers so that the next blocks of A
    and B
        # are loaded during the next iteration
        A += BLOCK_K * stride_ak
        B += BLOCK_K * stride_bk
```

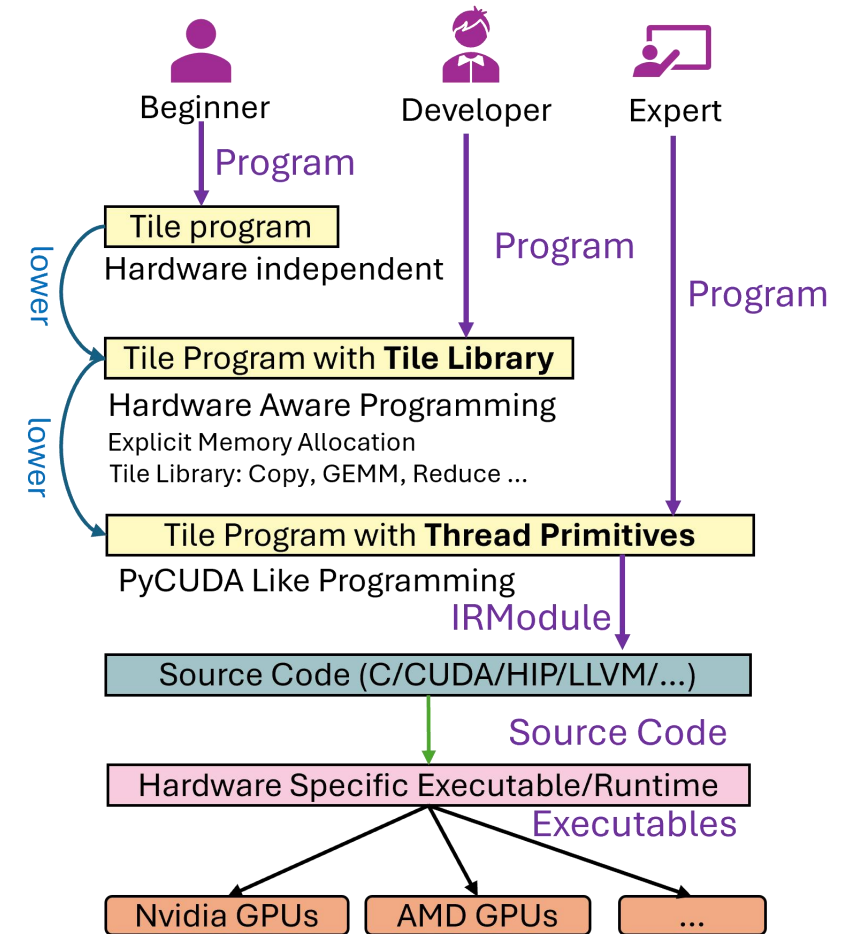
Triton: Problems

- Ease programming, yet it does not expose the GPU memory hierarchy.
 - Limit programmers' custom pipeline.
 - Lack inline assembly to perform vectorized datatype conversion.



TileLang

- Three levels of abstraction:
 - Level 1: A user writes pure compute logic without concern for hardware details (Not implemented).
 - Level 2: A user is aware of GPU architecture—such as shared memory, tiling, and thread blocks (comparable to Triton) .
 - Level 3: A user takes full control of thread-level primitives and can write code that is almost as explicit as CUDA (Not fully implemented).



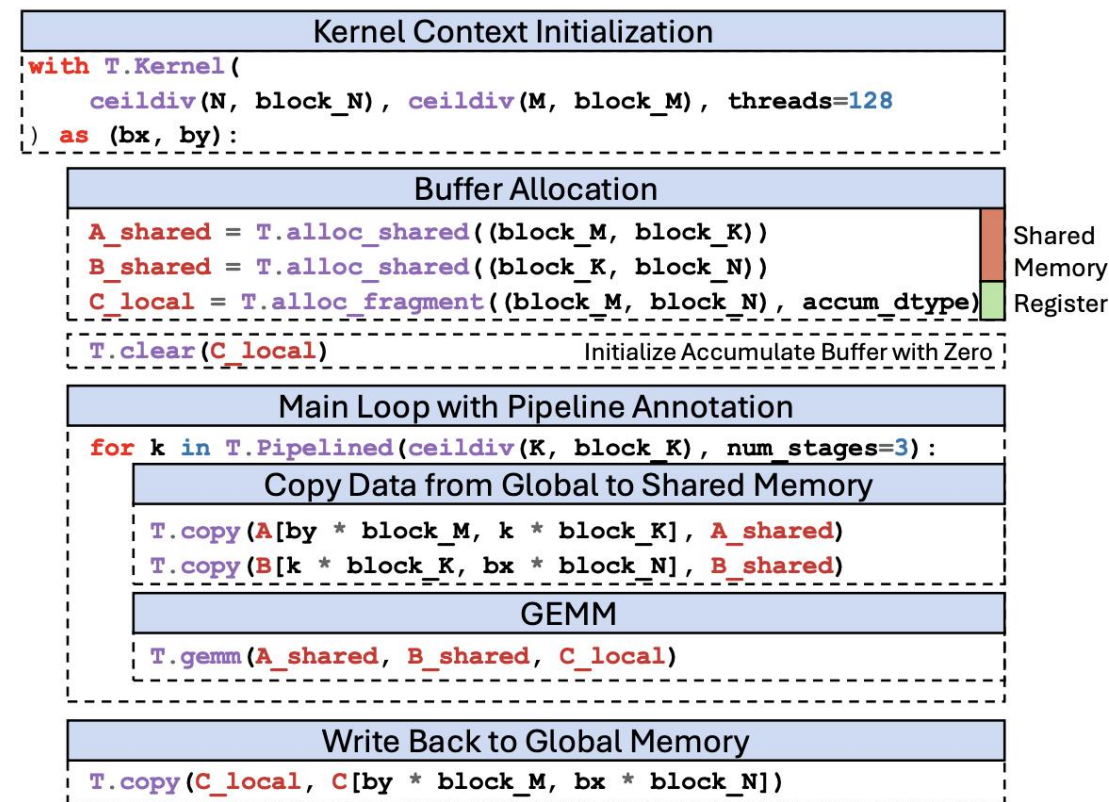
	Flexibility	Simplicity	Performance	Frontend
Triton	low	high	low	Python
TileLang	medium	medium	high	Python
ThunderKittens	high	low	high	C++

Tile-based Programming Model

1. Kernel context initialization.
 - Include thread block index and the number of threads.
2. Explicit memory allocation.
 - TileLang exposes user-facing primitives that map directly to physical memory spaces .
3. Main loop with automatic pipeline.
 - T.pipelined(..., num_stages=xx)
 - PipeThreader(OSDI 25)

```
import tilelang.language as T
```

```
def Matmul(A: T.Buffer, B: T.Buffer, C: T.Buffer):
```



Dataflow Centric Tile Operators

- These operators abstract low-level hardware memory access and computation
- Developers only focus on high-level algorithm design from dataflow perspective.

Dataflow Centric Tile Operators	
T.copy	A specialized memory copy operator that abstracts parallel data movement among registers, shared memory, and global memory.
T.gemm	Automatically selects implementations (cute/cuda/hip) for high-performance matrix multiplication on different GPUs.
T.reduce	A flexible reduction operator (e.g., sum, min, max) exploiting warp- and block-level parallelism.
T.atomic	Provides atomic operations (e.g., add, min, max) to ensure thread-safe updates in shared or global memory.

Scheduling Primitives

- Dataflow form the foundation of kernel.
- Scheduling primitives enable developers to precisely tune performance.

```
with T.Kernel(threads=32):
```

(a) Block Auto Copy

```
A_shared = T.alloc_shared((8, 32))  
T.copy(A, A_shared)
```

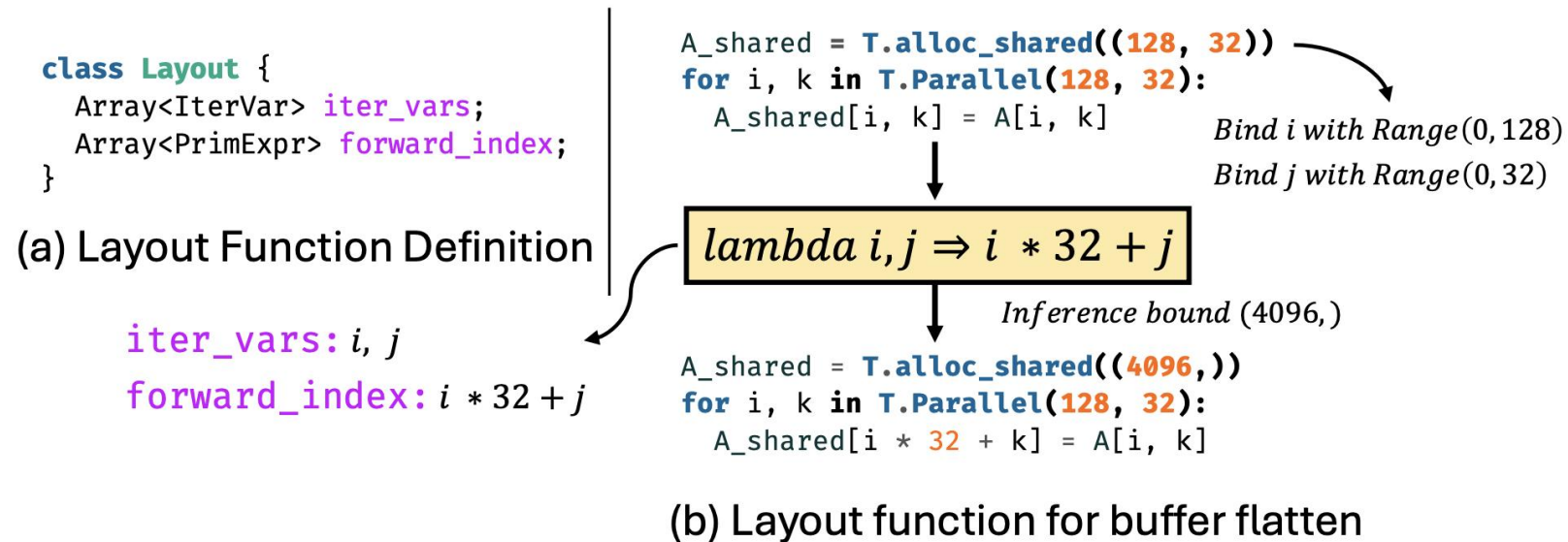
(b) Buffer Store with Primitive Parallel

```
for i, k in T.Parallel(8, 32):  
    A_shared[i, k] = A[by * BM + i, ko * BK + k]
```

Scheduling Primitives	
T.Parallel	Automates parallelization of loop iterations, mapping them to hardware threads, can also enable vectorization for additional performance gains.
T.Pipelined	Enables loop-level pipelining to overlap data transfers with computation and supports hardware-specific instructions such as async copy and TMA.
T.annotate_layout	Allows the definition of custom memory layouts to minimize bank conflicts and optimize thread binding.
T.use_swizzle	Improves L2 cache locality via swizzle thread blocks.

Memory Layout Composition

- Describe how data is organized and mapped in memory.
- Solution: IterVar+ForwardIndex



Thread Binding

- Determine how to map these layouts onto threads and how to infer register layout among individual threads.
- Solution: hierarchical priority system for tile operator layouts, where higher priority indicate stricter layout requirements and impact.

```
with T.Kernel(num_threads=4):  
    for i, j in T.Parallel(4, 4):  
        C[i, j] = C[i, j] + D[j]
```

Tile Program

T0	T1	T0	T1
0	0	1	1
T2	T3	T2	T3
0	0	1	1
T0	T1	T0	T1
2	2	3	3
T2	T3	T2	T3
2	2	3	3

C

+

?
?
?
?

D

Inference →

T0	T1	T0	T1
0	0	1	1
T2	T3	T2	T3
0	0	1	1
T0	T1	T0	T1
2	2	3	3
T2	T3	T2	T3
2	2	3	3

C

+

T0
T2
T0
T2

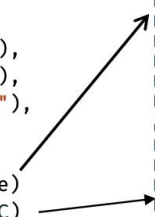
D

High-Performance Hardware Instructions

- Modern hardware often support multiple instructions for the same computational operation.
- Solution: C++ source injection, T.ptx primitive that allows direct emission of inline PTX instructions, Tile Libraries(cute).

```
1 def dp4a_example(  
2     A: T.Tensor((4,), dtype="int8"),  
3     B: T.Tensor((4,), dtype="int8"),  
4     C: T.Tensor((1,), dtype="int32"),  
5 ):  
6     with T.Kernel(num_threads=1):  
7         T.import_source(dp4a_template)  
8         T.call_extern("DP4A", A, B, C)
```

```
template <typename In, typename Out>  
__device__ void DP4A(  
    In *a, In *b,  
    Out *c) {  
    *c = __dp4a(*a, *b, *c);  
}  
  
__global__ void dp4a_example(  
    int8_t *a, int8_t *b,  
    int32_t *c) {  
    DP4A(a, b, c);  
}
```



(a) Utilize Instruction via C Source injection

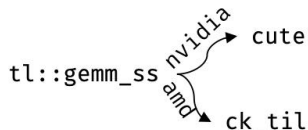
```
1 with T.Kernel(num_threads=32):  
2     ...  
3     T.ptx(  
4         "mma.m16n8k32.row.col.s32.s8.s8.s32",  
5         T.address_of(A), A_offset,  
6         T.address_of(B), B_offset,  
7         T.address_of(C), C_offset,  
8     )  
9     ...
```

↓

```
--asm__volatile__(  
    "...": "r"(): "r"()...  
)
```

(b) Leverage Instruction via T.ptx

```
1 with T.Kernel(num_threads=32):  
2     A_shared = ...  
3     B_shared = ...  
4     C_local = ...  
5     // tl::gemm_ss warps cute/ck  
6     T.call_extern("tl::gemm_ss",  
7         A_shared, B_shared, C_local)  
8     ...
```



(c) Leverage Instruction via Tile Library

ThunderKittens

PyTorch attention:

```
1  # imports
2  import torch
3  import torch.nn.functional as F
4
5
6  # compute Q@K.T
7  att = torch.matmul(
8      q, k.transpose(2, 3))
9
10 # compute softmax
11 att = F.softmax(
12     att, dim=-1,
13     dtype=torch.float32)
14
15 # convert back to bf16
16 att = att.to(q.dtype)
17
18 # mma att@V
19 attn_output = torch.matmul(att, v)
```

THUNDERKITTENS attention:

```
1  // imports
2  using namespace kittens;
3  rt_bf<16, 64> k_reg, v_reg;
4  // load k from shared memory to register
5  load(k_reg, k_smem[subtile]);
6  // compute Q@K.T
7  zero(att);
8  mma_ABt(att, q_reg, k_reg, att);
9  // compute softmax
10 sub_row(att, att, max_vec);
11 exp(att, att);
12 div_row(att, att, norm_vec);
13 // convert to bf16 for mma_AB
14 copy(att_mma, att);
15 // load v from shared memory to register
16 load(v_reg, v_smem[subtile]);
17 auto &v_reg_col = swap_layout_inplace(v_reg);
18 // mma att@V onto o_reg
19 mma_AB(o_reg, att_mma, v_reg_col, o_reg);
```

Programming Abstractions

- The operations are executed by a warp or warpgroup (4 warps) that collaboratively operate on a tile data.
- Register tiles, which are templated by type, shape, and layout.
- Shared tiles, which are templated by type and shape.

```
rt_bf<16, 64> k_reg, v_reg;  
// load k from shared memory to register  
load(k_reg, k_smem[subtile]);
```

- Global layout descriptors: HBM loads and stores are indexed into 4D tensors (similar to {batch, head, seqlen, and embed} in PyTorch).

LCSF: Generalized Asynchronous Template

- General pattern of a kernel is to load tiles from HBM to SRAM, perform computation in fast memory, store the result for the tile back to HBM.
- 1. Load function: Specify the data that load workers should load from HBM to shared memory, and when to signal to compute workers.
- 2. Compute function: Specify the kernel that compute workers should execute, using the tile data and operation primitives.
- 3. Store function: Specify what data needs to be stored to HBM.
- 4. Finish function: Workers store the final result and exit.

LCSF: Generalized Asynchronous Template

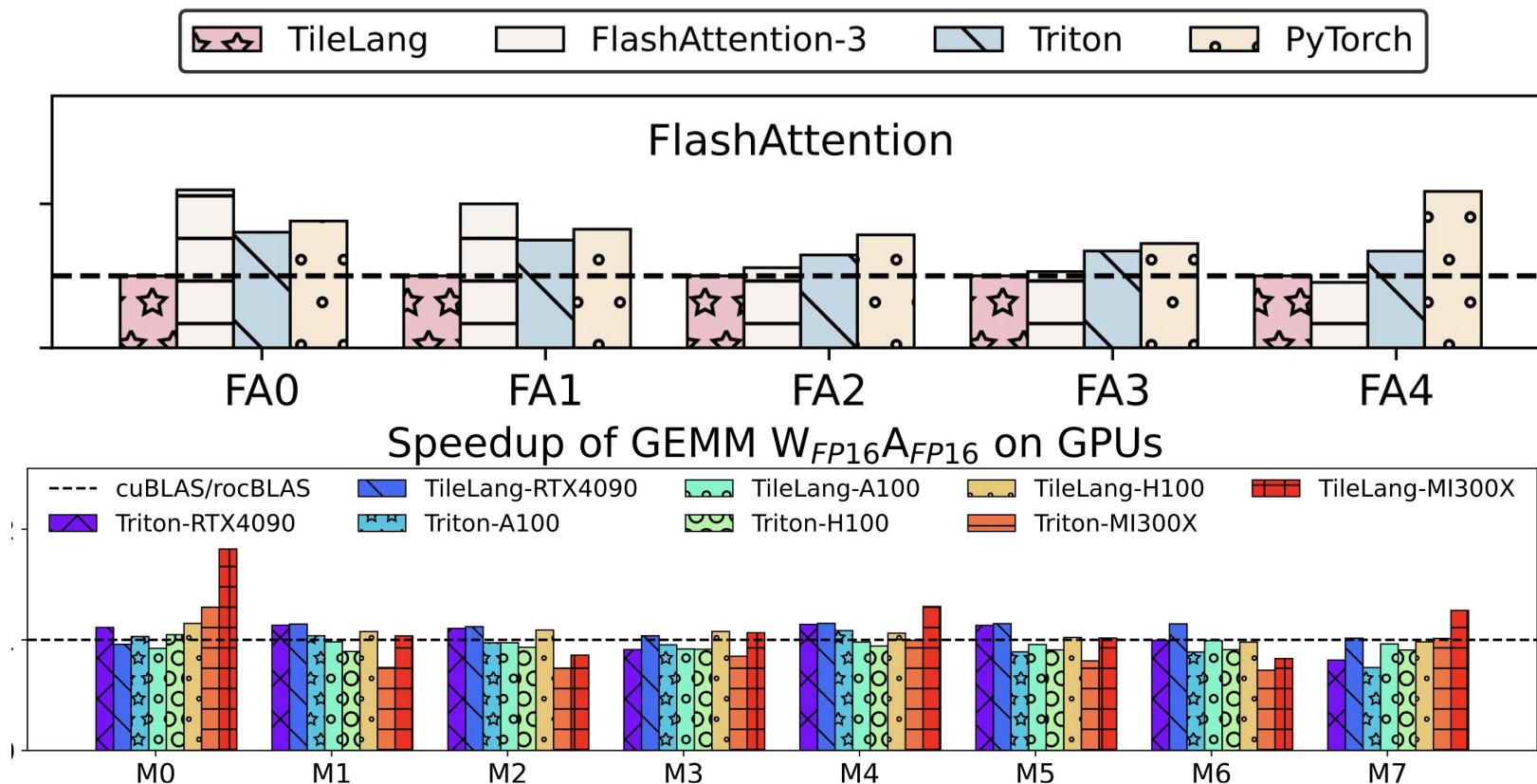
```
struct TestOp {
    static constexpr int opcode = 1;
    struct controller {
        static __device__ int init_semaphores(const globals &g, state &s) {
            return 0;
        }
        static __device__ int release_lid(const globals &g, typename {{PROJECT_NAME_LOWER}}
            return query;
        }
    };
    struct loader {
        static __device__ void run(const globals &g, state &s) {
            if(laneid() == 0) { printf("Hello, world from {{PROJECT_NAME_LOWER}}!\n"); }
        }
    };
    struct launcher {
        static __device__ void run(const globals &g, state &s) {
```

```
        // Wait and release pages
        if(laneid() < {{PROJECT_NAME_LOWER}}_config::NUM_PAGES) {
            s.wait_page_ready(laneid());
            s.finish_page(laneid(), {{PROJECT_NAME_LOWER}}_config::NUM_CONSUMER_WARPS);
        }
#ifdef KITTENS_BLACKWELL
        else if(laneid() == {{PROJECT_NAME_LOWER}}_config::NUM_PAGES) {
            s.wait_tensor_ready();
            arrive(s.tensor_finished, {{PROJECT_NAME_LOWER}}_config::NUM_CONSUMER_WARPS);
        }
#endif
    };
    struct consumer {
        static __device__ void run(const globals &g, state &s) {}
    };
    struct storer {
        static __device__ void run(const globals &g, state &s) {}
    };
};
```

- Programming remains complex.
- Not much different from Cute/Cutlass.

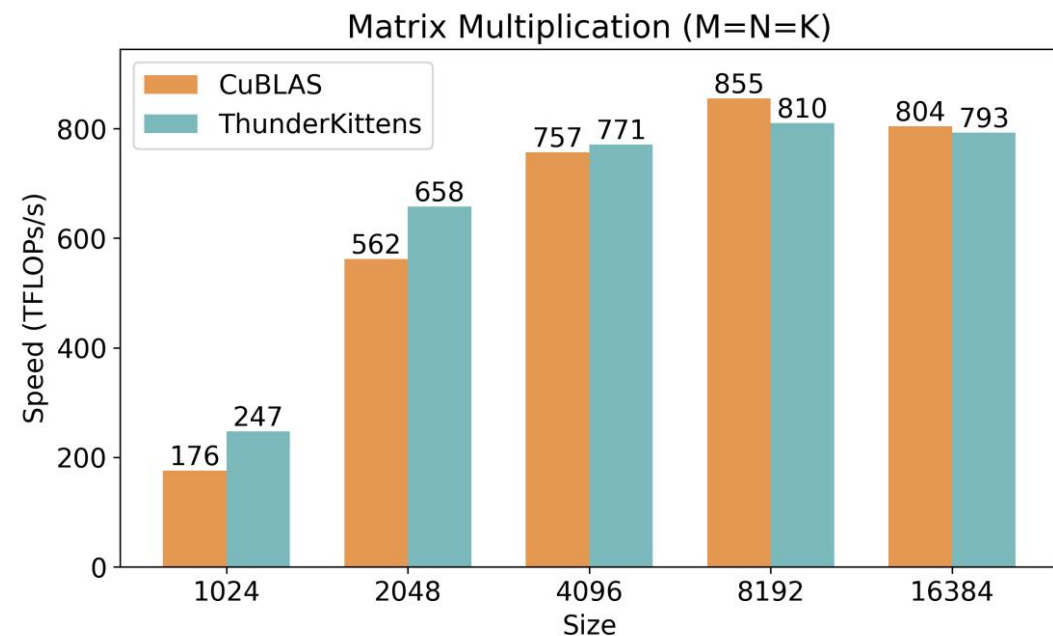
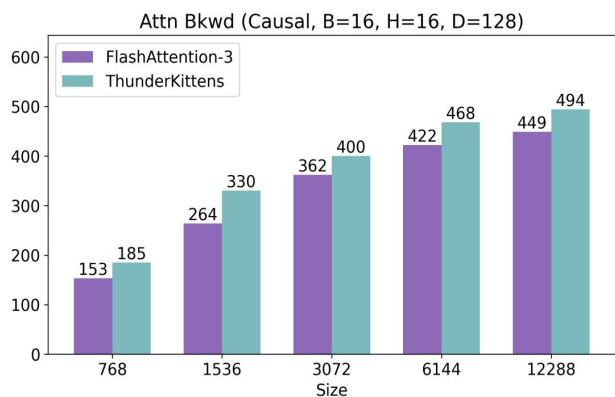
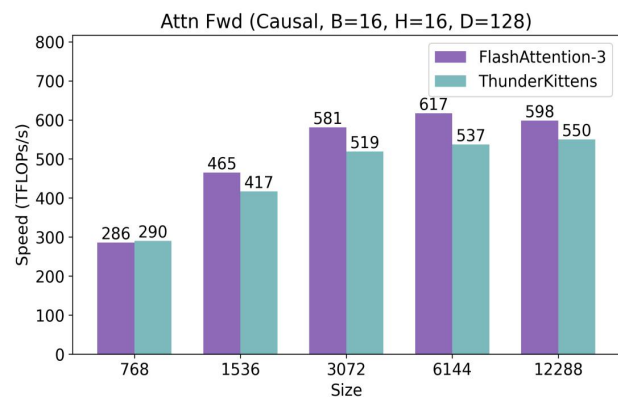
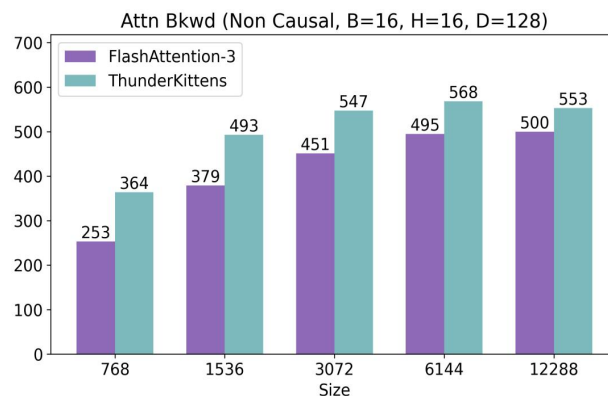
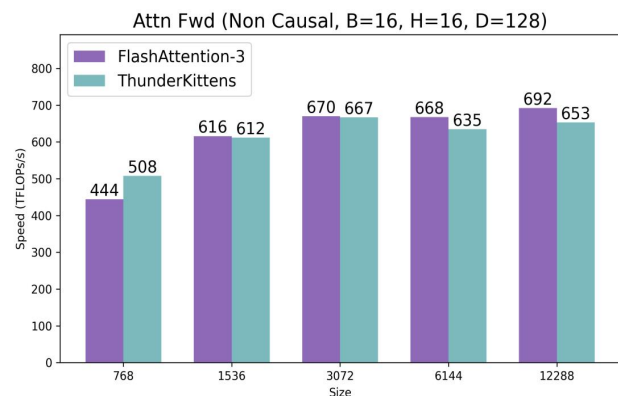
Performance

Flash Attention performance on H100



- Comparable to FA3/CuBLAS.
- Better than Triton/PyTorch.

Performance



- Comparable to FA3 in FWD, better than FA3 in BWD.
- Comparable to CuBLAS.

Discussion: Task-based programming model

```

1  [
2  TaskMapping(
3      instance="gemm_host",
4      variant="gemm_host",
5      proc=HOST,
6      mems=[GLOBAL, GLOBAL, GLOBAL],
7      tunables={"U": 256,
8                "V": 256},
9      entrypoint=True,
10     calls=["gemm_block"]
11 ),
12 TaskMapping(
13     instance="gemm_block",
14     variant="gemm_block",
15     proc=BLOCK,
16     mems=[GLOBAL, GLOBAL, GLOBAL],
17     tunables={"W": 64},
18     calls=["clear_tile",
19            "gemm_tile",
20            "copy_tile"],
21     warpspecialize=True,
22     pipeline=3
23 ),
24 TaskMapping(
25     instance="gemm_tile",
26     variant="gemm_tile",
27     proc=BLOCK,
28     mems=[NONE, SHARED, SHARED],
29     tunables={"WGS": 2},
30     calls=["gemm_warpgroup"]
31 )
32 TaskMapping(
33     instance="gemm_warpgroup",
34     variant="gemm_inner",
35     proc=WARPGROUP,
36     mems=[NONE, SHARED, SHARED],
37     tunables={"PIECES": 4,
38               "PROC": WARP}
39     calls=["gemm_warpgroup"]
40 ),
41 TaskMapping(
42     instance="gemm_warpgroup",
43     variant="gemm_inner",
44     proc=WARP,
45     mems=[NONE, SHARED, SHARED],
46     tunables={"PIECES": 32,
47               "PROC": THREAD}
48     calls=["gemm_thread"]
49 ),
50 TaskMapping(
51     instance="gemm_thread",
52     variant="gemm_thread",
53     proc=THREAD,
54     mems=[REGISTER, SHARED, SHARED],
55 ),
56 # Mappings for `fill` and
57 # `copy` task trees elided.
58 ]

```

Diagram annotations:

- (1) Grouping lines 2-11 (TaskMapping for host).
- (2) Grouping lines 6-8 (tunables for host).
- (3) Grouping lines 9-10 (calls for host).

```

1  def block_mma(SmemA: fp32[64, 8], SmemB: fp32[8, 64],
2                RegsC: fp32[4, 4, 4]):
3      RegsA, RegsB = register fp32[4], fp32[4]
4      task_map = spatial(2, 2) * repeat(2, 2)
5      worker_id = threadIdx.x / 32 # warp index
6      for i, j in task_map(worker_id):
7          wmma_load_a(&SmemA[i * 16, 0], RegsA)
8          wmma_load_b(&SmemB[0, j * 16], RegsB)
9          wmma_mma(RegsA, RegsB, RegsC[i, j])

```

- It's been published, but hasn't been widely used.
- Orthogonal to the tile-based programming model?