Modern DSLs Compile Workflow

Xinhao Luo

2025.11.28

Prerequisites

IR (Intermediate Representation)

- Def: Source Code —> IR —> Target Code

- Types:
  - High Level IR: Analysis and Transformation based on Source Code
  - Middle IR: Language- and Hardware-Independent Optimization
  - Low IR: Hardware-Specific Semantics and Optimizations

- Example: AST, ONNX Graph, Triton GPU IR

Prerequisites

Pass

- Def: Traversal over the IR to perform analysis or transformation.
- Types:
  - Analysis Pass: Read IR and collect information.
  - Transformation Pass: Modify IR to perform optimization or lowering.
- Example: Loop analysis, loop unroll

```
// 变换前：未展开的循环
scf.for %i = %c0 to %c3 step %c1 {
  %val = arith.muli %i, %i : i32
  memref.store %val, %buffer[%i] : memref<4xi32>
}
```

loop unroll →

```
// 变换后：完全展开
%v0 = arith.muli %c0, %c0 : i32
memref.store %v0, %buffer[%c0] : memref<4xi32>
%v1 = arith.muli %c1, %c1 : i32
memref.store %v1, %buffer[%c1] : memref<4xi32>
%v2 = arith.muli %c2, %c2 : i32
memref.store %v2, %buffer[%c2] : memref<4xi32>
```

Prerequisites

Pattern Rewrite

- Def: Declarative mechanism to transform IR through local match + rewrite.
- Strategy: Patterns are applied greedily until the IR becomes stable.
- Example: Canonicalize, Operator Fusion

```
// 模式 1: x + 0 → x
%sum = arith.addi %x, %c0 : i32 // 变成 %x

// 模式 2: x * 1 → x
%prod = arith.muli %x, %c1 : i32 // 变成 %x

// 模式 3: x * 0 → 0
%prod = arith.muli %x, %c0 : i32 // 变成 %c0
```

```
// 初始 IR
%c0 = arith.constant 0 : i32
%c1 = arith.constant 1 : i32
%t1 = arith.addi %arg, %c0 : i32  // 可以简化
%t2 = arith.muli %t1, %c1 : i32   // 简化后又可以简化
func.return %t2 : i32
```

```
// 第一轮：应用 x + 0 → x
%c1 = arith.constant 1 : i32
%t2 = arith.muli %arg, %c1 : i32  // 现在 %t1 变成了 %arg
func.return %t2 : i32
```

```
// 第二轮：应用 x * 1 → x
func.return %arg : i32
```

Outline

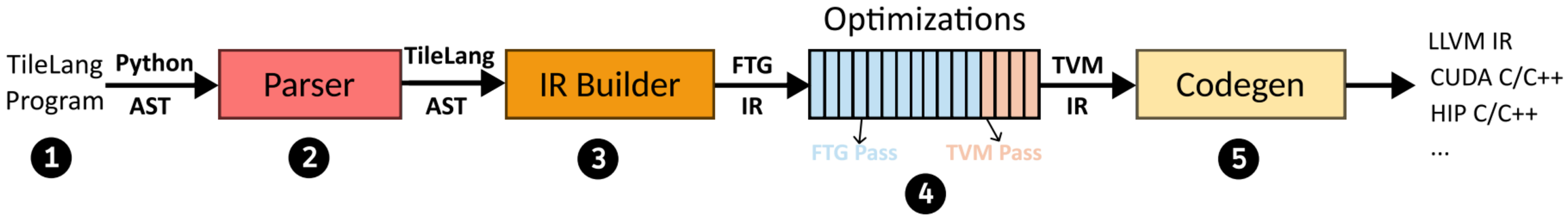
- Modern DSLs Compile Workflow
  - TileLang
  - CuTeDSL, TLX, Triton
- Benchmark
- Conclusion

Modern DSLs

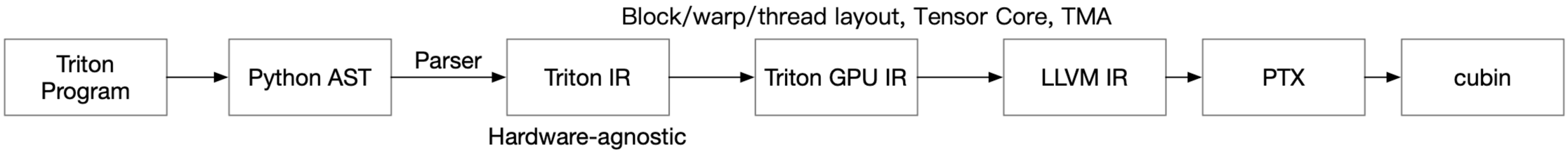
Overview

Workflow

- TileLang



- Triton



Key Points

Workflow

- Structure
- Pipeline
- **Pass: 作用以及Example**
- Optimization

Structure
Frontend

- Language
  - 显式 Grid/Block Level Tile Dataflow
  - 显式 Shared Memory/Local Register
  - TileOp：高层算子抽象（T.copy、T.gemm）
  - Software Pipeline

```
import tilelang.language as T

# 控制流与并行
with T.Kernel(T.ceildiv(M, 128), threads=256) as bx:
    with T.Parallel(128) as ty:
        with T.serial(16) as tx:
            ...

# 高层算子（TileOp）
T.copy(src, dst)            # 数据搬移
T.gemm(A, B, C)             # 矩阵乘法
T.reduce_absmax(tensor)     # 归约操作

# 内存分配
buf = T.alloc_shared([128, 64], "float16")
acc = T.alloc_fragment([8, 8], "float32")

# 原子操作
T.atomic_add(ptr, val)
```

Structure
Engine

- Phase
  - **两阶段 Pass Pipeline**
  - Phase-1: LowerAndLegalize, 把高层 DSL 变成可优化的 IR (TVM中的**TensorIR**)
  - Phase-2: OptimizeForTarget, 根据不同的目标硬件，做不同的优化

- Lower:
  - Host/Device Codegen拆分与调度
    - Host IR → C++ wrapper
    - Device IR → CUDA/PTX

```python
def matmul(M, N, K, block_M, block_N, block_K, dtype="float16",
accum_dtype="float"):

    @T.prim_func
    def gemm(
            A: T.Tensor((M, K), dtype),
            B: T.Tensor((K, N), dtype),
            C: T.Tensor((M, N), 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_

            T.clear(C_local)
            for k in T.Pipelined(T.ceildiv(K, block_K), num_stage
                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 gemm
```

Pipeline

Phase-1: LowerAndLegalize

- BindTarget
  - 将 Target (包含硬件能力信息) 绑定到 IR 中
  - 后续 Pass 需要查询硬件能力才能做决策

```
def matmul(M, N, K, block_M, block_N, block_K, dtype="float16",
accum_dtype="float"):

    @T.prim_func
    def gemm(
            A: T.Tensor((M, K), dtype),
            B: T.Tensor((K, N), dtype),
            C: T.Tensor((M, N), 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 gemm
```

```
# 后续 Pass 会查询
if have_tma(target):    # Hopper 有 TMA 吗?
  use_tma_path()

if target.max_shared_memory_per_block >=163840: # 共享内存够大吗?
  use_large_tile()
```

```
@I.ir_module
class Module:
    @T.prim_func
    def gemm(A_handle: T.handle, B_handle: T.handle, C_handle: T.handle):
        T.func_attr({"target": T.target({"arch": "sm_90", "host": {"keys": ["cpu"], "kind": "c", "tag": ""}, "keys":
        ["cuda", "gpu"], "kind": "cuda", "max_num_threads": 1024, "tag": "", "thread_warp_size": 32})})
```

Pipeline

Phase-1: LowerAndLegalize

- LetInline
  - 内联临时变量，减少嵌套深度
  - 让后续的 Pattern Rewrite 更容易

```
# 嵌套的Let表达式难以分析
Let x = a + b in
  (Let y = x * 2 in
    (y + x))

# LetInline处理后
(a + b) * 2 + (a + b)
```

- Simplify
  - 常量折叠、代数化简、死代码消除

```
# 简化前
if(True): a = 1
x = 0+ y * 1

# 简化后
a = 1
x = y
```

Pipeline

Phase-1: LowerAndLegalize

- InjectAssumes
  - 注入__assume()内置函数，告诉编译器 ”这个条件永远成立”
  - 消除冗余的边界检查， 使用对齐的向量化加载 (ldg.128)

```
// 假设线程索引在合法范围内
__assume(threadIdx.x < blockDim.x);

// 假设指针满足特定对齐要求
__assume((global_ptr & 15) == 0);  // 16字节对齐

// 假设循环次数满足特定条件
__assume(n % 8 == 0);  // 向量化友好
```

```
# 没有对齐假设 → 保守方案
float4 value
value.x = ptr[0]
value.y = ptr[1]
value.z = ptr[2]
value.w = ptr[3]
# 生成: 4条独立的32位加载指令

# 有对齐假设 → 激进优化
__assume((ptr & 15) == 0)  # 保证16字节对齐
float4 value = *(float4*)ptr
# 生成: 1条128位向量加载指令 ldg.128
```

Pipeline

Phase-1: LowerAndLegalize

- LayoutInference
  - 推断最终布局与线程映射，固化索引计算
  - 决定每个 Buffer 用什么布局，每个线程访问哪些元素，索引的线性化
  - 索引被改写为线性/规范形态

```
# 推断前（抽象访问）
A_shared[ty, tx] = A_global[bx*128 + ty, tx]

# 推断后（具体索引）
# 假设推断出 function 布局和特定线程映射
linear_idx = function(ty, tx, stride=128)
A_shared[linear_idx] = A_global[bx*128*K + ty*K + tx]
```

```
B_shared = T.Buffer((32, 128), "float16", scope="shared.dyn")
T.block_attr({"layout_map": {C_local: metadata["tl.Fragment"][0], A_shared: metadata["tl.Layout"][0], B_shared: metadata["tl.Layout"]
[1]}})
bx = T.launch_thread("blockIdx.x", 8)
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)
with T.block("tilelang_root"):
    T.reads(A[by * 128, 0:993], B[0:993, bx * 128], C[by * 128, bx * 128])
    T.writes()
    T.block_attr({"layout_map": {C_local: metadata["tl.Fragment"][0], A_shared: metadata["tl.Layout"][0], B_shared: metadata["tl.
    Layout"][1]}})
    A_shared = T.alloc_buffer((128, 32), "float16", data=A_shared.data, scope="shared.dyn")
    B_shared = T.alloc_buffer((32, 128), "float16", data=B_shared.data, scope="shared.dyn")
    C_local = T.alloc_buffer((128, 128), data=C_local.data, scope="local.fragment")
    T.fill(T.tvm_access_ptr(T.type_annotation("float32"), C_local.data, 0, 16384, 2), 0)
```

Pipeline

Phase-1: LowerAndLegalize

- LowerTileOp
  - 把 TileOp (T.copy、T.gemm) 降级为底层 IR (循环、BufferLoad/Store)

```
# DSL
T.copy(A_global[0:128, 0:64], A_shared)

# 降级后（伪代码）
for i in T.parallel(128):
    for j in T.vectorized(64):
        if (i < bound_i) and (j < bound_j):  # 边界保护
            A_shared[i, j] = A_global[i, j]

# DSL
T.gemm(A, B, C)

# 降级后（伪代码）
for i in range(M):
    for j in range(N):
        for k in range(K):
            C[i, j] += A[i, k] * B[k, j]
```

```
with T.block("tilelang_root"):
    T.reads(A[by * 128, 0:993], B[0:993, bx * 128], C[by * 128, bx * 128])
    T.writes()
    T.block_attr({"layout_map": {C_local: metadata["tl.Fragment"][0]}})
    A_shared = T.alloc_buffer((1, 16, 256), "float16", scope="shared.dyn")
    B_shared = T.alloc_buffer((2, 4, 512), "float16", scope="shared.dyn")
    C_local = T.alloc_buffer((128,), data=C_local.data, scope="local")
    if tx < 128 and tx >= 0:
        for i in T.unroll(64, annotations={"pragma_unroll_explicit": T.bool(False)}):
            for vec in T.vectorized(2):
                C_local[((i * 2 + vec) // 64 * 64 + tx // 32 * 16 + (i * 2 + vec) % 4 // 2 *
                vec) % 64 // 4 * 8 + tx % 4 * 2 + (i * 2 + vec) % 2) // 8 * 4 + ((i * 2 + vec
                vec) % 4 // 2 * 8 + tx % 32 // 4) % 16 // 8 * 2 + ((i * 2 + vec) % 64 // 4 *
                = T.Cast("float32", 0)
    for k in T.serial(32, annotations={"num_stages": 3}):
        if tx == 0:
            T.tma_load(T.create_tma_descriptor(6, 2, A.data, 1024, 1024, T.int64(2), T.int64(
            tvm_access_ptr(T.type_annotation("float16"), A_shared.data, 0, 4096, 2), k * 32,
        if tx == 0:
            for i in T.unroll(2):
                T.tma_load(T.create_tma_descriptor(6, 2, B.data, 1024, 1024, T.int64(2), T.in
                T.tvm_access_ptr(T.type_annotation("float16"), B_shared.data, 2048 * i, 2048,
        T.tl_gemm("tl::gemm_ss<128, 128, 32, 4, 1, 0, 0, 0, 32, 128, 0, 0, true>", T.tvm_acce
        A_shared.data, 0, 4096, 1), T.tvm_access_ptr(T.type_annotation("float16"), B_shared.d
        type_annotation("float32"), C_local.data, 0, 16384, 3))
    if tx < 128 and tx >= 0:
        for i in T.unroll(64, annotations={"pragma_unroll_explicit": T.bool(False)}):
            for vec in T.vectorized(2):
                C[by * 128 + (i * 2 + vec) // 64 * 64 + tx // 32 * 16 + (i * 2 + vec) % 4 //
                + vec) % 64 // 4 * 8 + tx % 4 * 2 + (i * 2 + vec) % 2] = T.Cast("float16", C_
                32 * 16 + (i * 2 + vec) % 4 // 2 * 8 + tx % 32 // 4) // 64 * 64 + ((i * 2 + v
                + vec) % 2) // 8 * 4 + ((i * 2 + vec) // 64 * 64 + tx // 32 * 16 + (i * 2 + v
                8 * 2 + ((i * 2 + vec) % 64 // 4 * 8 + tx % 4 * 2 + (i * 2 + vec) % 2) % 2])
```

Pipeline

Phase-1: LowerAndLegalize

- LegalizeVectorizedLoop
  - 修正向量化中的非法模式

```
# 非法：跨步向量化
for i in T.vectorized(8):  # 步长=2
    A[i*2] = ...

# 修正后：回退到标量
for i in T.serial(8):
    A[i*2] = ...
```

- LegalizeSafeMemoryAccess
  - 为可能越界的访存注入保护

```
// 插入前
shared[idx] = val

// 插入后
if (idx < bound) {
    shared[idx] = val
} else {
    shared[idx] = 0
}
```

Pipeline

Phase-1: LowerAndLegalize

- Conclusion: 经过14个pass
  - 输入DSL —> 规范 TIR(循环、BufferLoad/Store，layout推断，线性化索引，向量化，硬件信息绑定)

```
with T.block("root"):
    T.reads()
    T.writes()
    C_local = T.Buffer((128,), scope="local")
    T.block_attr({"layout_map": {C_local: metadata["tl.Fragment"][0]}})
    bx = T.launch_thread("blockIdx.x", 8)
    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)
    with T.block("tilelang_root"):
        T.reads(A[by * 128, 0:993], B[0:993, bx * 128], C[by * 128, bx * 128])
        T.writes()
        T.block_attr({"layout_map": {C_local: metadata["tl.Fragment"][0]}})
        A_shared = T.alloc_buffer((1, 16, 256), "float16", scope="shared.dyn")
        B_shared = T.alloc_buffer((2, 4, 512), "float16", scope="shared.dyn")
        C_local = T.alloc_buffer((128,), data=C_local.data, scope="local")
        for i in T.unroll(64, annotations={"pragma_unroll_explicit": T.bool(False)}):
            for vec in T.vectorized(2):
                C_local[i * 2 + vec] = T.float32(0.0)
        for k in T.serial(32, annotations={"num_stages": 3}):
            if tx == 0:
                T.tma_load(T.create_tma_descriptor(6, 2, A.data, 1024, 1024, T.int64(2), T.int64(2048), 32, 128, 1, 1, 0, 2
                tvm_access_ptr(T.type_annotation("float16"), A_shared.data, 0, 4096, 2), k * 32, by * 128, 0)
            if tx == 0:
                for i in T.unroll(2):
                    T.tma_load(T.create_tma_descriptor(6, 2, B.data, 1024, 1024, T.int64(2), T.int64(2048), 64, 32, 1, 1, 0
                    T.tvm_access_ptr(T.type_annotation("float16"), B_shared.data, i * 2048, 2048, 2), bx * 128 + i * 64, k
            T.tl_gemm("tl::gemm_ss<128, 128, 32, 4, 1, 0, 0, 0, 32, 128, 0, 0, true>", T.tvm_access_ptr(T.type_annotation("
            A_shared.data, 0, 4096, 1), T.tvm_access_ptr(T.type_annotation("float16"), B_shared.data, 0, 4096, 1), T.tvm_ac
            type_annotation("float32"), C_local.data, 0, 16384, 3))
        for i in T.unroll(64, annotations={"pragma_unroll_explicit": T.bool(False)}):
            for vec in T.vectorized(2):
                C[by * 128 + i // 32 * 64 + tx // 32 * 16 + i % 2 * 8 + tx % 32 // 4, bx * 128 + i % 32 // 2 * 8 + tx % 4 *
                Cast("float16", C_local[i * 2 + vec])
```

Pipeline

Phase-2: OptimizeForTarget

- 根据目标硬件能力，走不同的特定优化路径
- TMA（Tensor Memory Accelerator）：bulk 异步复制
- Warp Specialization：Producer/Consumer 角色拆分
- WGMMA：4-warp 组矩阵乘加
- mbarrier：可计数字节屏障

Pipeline

Phase-2: OptimizeForTarget

- LowerSharedBarrier
  - 在共享内存中创建 mbarrier 并初始化

```
// 初始化
mbarrier_init(&barrier, num_threads);

// Producer 发起异步传输前
mbarrier_expect_tx(&barrier, bytes);  // 告诉 barrier 期待 N 字节

// TMA 自动递减计数
tma_load_async(...);

// Producer 到达
mbarrier_arrive(&barrier);

// Consumer 等待
mbarrier_wait_parity(&barrier, parity);  // parity ∈ {0, 1} 交替
```

```
A_shared = T.alloc_buffer((3, 1, 16, 256), "float16", scope="shared.dyn")
B_shared = T.alloc_buffer((3, 2, 4, 512), "float16", scope="shared.dyn")
C_local = T.alloc_buffer((128,), data=C_local.data, scope="local")
T.create_list_of_mbarrier(1, 1, 1, 128, 128, 128)
T.attr([128, 128], "kWarpSpecializationScope", 0)
if tx >= 128:
    for k in range(32):
        with T.block("", no_realize=True):
            T.reads()
            T.writes()
            T.block_attr({"stmt_group": 1})
            T.mbarrier_wait_parity(T.get_mbarrier(k % 3 + 3), T.bitwise_xor(k // 3 % 2, 1))
            if T.tl_shuffle_elect(128):
                T.tma_load(T.create_tma_descriptor(6, 2, A.data, 1024, 1024, T.int64(2), T.i
                0), T.get_mbarrier(k % 3), T.tvm_access_ptr(T.type_annotation("float16"), A_
                k * 32, by * 128, 0)
        with T.block("", no_realize=True):
            T.reads()
            T.writes()
            T.block_attr({"stmt_group": 1})
            if T.tl_shuffle_elect(128):
                for i in T.unroll(2):
                    T.tma_load(T.create_tma_descriptor(6, 2, B.data, 1024, 1024, T.int64(2),
                    2, 0), T.get_mbarrier(k % 3), T.tvm_access_ptr(T.type_annotation("float1
                    * 4096, 2048, 2), bx * 128 + i * 64, k * 32, 0)
            T.ptx_arrive_barrier(T.get_mbarrier(k % 3))
```

Pipeline

Phase-2: OptimizeForTarget

- MultiVersionBuffer
  - 构造环形缓冲，支持生产/消费重叠

```
// 单版本（无流水）
__shared__ float A[128][64]
for (int i = 0; i < N; i++) {
    copy(global, A);       // 阶段1: copy
    __syncthreads();
    compute(A);            // 阶段2: compute
    __syncthreads();
}

// 多版本（3-阶段流水）
__shared__ float A[3][128][64];  // A[stage][...]
for (int i = 0; i < N; i++) {
    int stage = i % 3;
    copy(global, A[stage]);       // copy(i) 与 compute(i-1) 并发
    compute(A[(stage-1+3)%3]);
}
```

```
T.block_attr({"layout_map": {C_local: metadata["tl.Fragment"][0]}})
A_shared = T.alloc_buffer((1, 16, 256), "float16", scope="shared.dyn")
B_shared = T.alloc_buffer((2, 4, 512), "float16", scope="shared.dyn")
C_local = T.alloc_buffer((128,), data=C_local.data, scope="local")
```

```
A_shared = T.alloc_buffer((3, 1, 16, 256), "float16", scope="shared.dyn")
B_shared = T.alloc_buffer((3, 2, 4, 512), "float16", scope="shared.dyn")
C_local = T.alloc_buffer((128,), data=C_local.data, scope="local")
```

Pipeline

Phase-2: OptimizeForTarget

- WarpSpecialized
  - Producer 在加载 tile(i+1) 时，Consumer 在计算 tile(i)
  - 寄存器压力降低：Producer不需要accumulator, Consumer 不需要地址计算

```
__global__ void kernel() {
    for (int i = 0; i < N; i++) {
        // 所有 warp 都执行相同的序列
        copy_phase();       //  copy
        __syncthreads();
        compute_phase();    //  compute
        __syncthreads();
    }
}
```

```
__global__ void kernel() {
    int warp_id = threadIdx.x / 32;

    if (warp_id < 4) {
        // Producer warpgroup (warp 0-3)
        for (int i = 0; i < N; i++) {
            mbarrier_expect_tx(&barrier[i%3], bytes);
            tma_load_async(global, shared[i%3]);  // 只负责 copy
            mbarrier_arrive(&barrier[i%3]);
        }
    } else {
        // Consumer warpgroup (warp 4-7)
        for (int i = 0; i < N; i++) {
            mbarrier_wait_parity(&barrier[i%3], i%2);
            wgmma(shared[i%3], acc);  // 只负责 compute
        }
    }
}
```

```
if tx >= 128:
    for k in range(32):
        with T.block("", no_realize=True):
            T.reads()
            T.writes()
            T.block_attr({"stmt_group": 1})
            T.mbarrier_wait_parity(T.get_mbarrier(k % 3 + 3), T.bitwise_xor(k // 3 % 2, 1))
            if T.tl_shuffle_elect(128):
                T.tma_load(T.create_tma_descriptor(6, 2, A.data, 1024, 1024, T.int64(2), T.i
                0), T.get_mbarrier(k % 3), T.tvm_access_ptr(T.type_annotation("float16"), A_
                k * 32, by * 128, 0)
        with T.block("", no_realize=True):
            T.reads()
            T.writes()
            T.block_attr({"stmt_group": 1})
            if T.tl_shuffle_elect(128):
                for i in T.unroll(2):
                    T.tma_load(T.create_tma_descriptor(6, 2, B.data, 1024, 1024, T.int64(2),
                    2, 0), T.get_mbarrier(k % 3), T.tvm_access_ptr(T.type_annotation("float1
                    * 4096, 2048, 2), bx * 128 + i * 64, k * 32, 0)
            T.ptx_arrive_barrier(T.get_mbarrier(k % 3))
else:
    for i in T.unroll(64, annotations={"pragma_unroll_explicit": T.bool(False)}):
        for vec in T.vectorized(2):
            C_local[i * 2 + vec] = T.float32(0.0)
    for k in range(32):
        with T.block("", no_realize=True):
            T.reads()
            T.writes()
            T.block_attr({"stmt_group": 1})
            T.mbarrier_wait_parity(T.get_mbarrier(k % 3), k // 3 % 2)
            T.tl_gemm("tl::gemm_ss<128, 128, 32, 4, 1, 0, 0, 0, 32, 128, 0, 0, true>", T.tvm
            ("float16"), A_shared.data, k % 3 * 4096, 4096, 1), T.tvm_access_ptr(T.type_anno
```

Pipeline

Phase-2: OptimizeForTarget

- 输入Phase-1 的规范 IR：

```
// 伪 IR
for tile in range(N):
    for thread in parallel:
        if in_copy_region:
            shared[idx] = global[idx]  // copy
    __syncthreads()

    for thread in parallel:
        if in_compute_region:
            acc += shared[idx] * weight[idx]  // compute
    __syncthreads()
```

- 输出：

```
// 伪 IR
int warp_role = get_warp_role();  // Producer/Consumer/Both

if (warp_role == Producer) {
    for tile in range(N):
        mbarrier_expect_tx(&barrier[tile%stages], bytes);
        tma_load_async(global, shared[tile%stages]);
        mbarrier_arrive(&barrier[tile%stages]);
}

if (warp_role == Consumer) {
    for tile in range(N):
        mbarrier_wait_parity(&barrier[tile%stages], tile%2);
        wgmma(shared[tile%stages], acc);
}

if (warp_role == Both) {
    // epilogue
    ...
}
```

- Pass 的改写步骤：
  - 1. 找到 copy 和 compute 的分界
  - 2. 为循环/分支打标签:
    - Producer: copy
    - Consumer: compute
    - Both: epilogue
  - 3. 插入 mbarrier
    - Producer: expect_tx → tma → arrive
    - Consumer: wait → compute
  - 4. 分裂控制流：用 if (warp_role) 条件化执行

Pipeline

Phase-2: OptimizeForTarget

- AnnotateWarpGroupRegAlloc
  - 为 WS 的不同分支注入寄存器预算设置

```
if 128 <= tx:
    T.set_max_nreg(24, 0)
    for k in range(32):
        T.mbarrier_wait_parity(T.ge
        if T.tl_shuffle_elect(128):
            T.mbarrier_expect_tx(T.
            T.fence_proxy_async()
            T.tma_load(A_desc, T.ge
            4096, 2), k * 32, by *
            T.mbarrier_expect_tx(T.
            T.fence_proxy_async()
            T.tma_load(B_desc, T.ge
            2), bx * 128, k * 32, 0
            T.tma_load(B_desc, T.ge
            2048, 2), bx * 128 + 64
        T.ptx_arrive_barrier(T.get_
else:
    T.set_max_nreg(240, 1)
```

- PipelinePlanning
  - 计算流水线的阶段数、调度顺序、依赖关系

```
# 注解示例（在 IR 上）
loop.attr["software_pipeline_stage"] = [0, 1, 2]       # 阶段 ID
loop.attr["software_pipeline_order"] = [0, 2, 1]      # 执行顺序
loop.attr["software_pipeline_async_stages"] = 2       # 异步阶段数
```

- InjectSoftwarePipeline
  - 根据规划，展开为实际的 prologue/mainloop/epilogue。

```
// 原始循环
for (int i = 0; i < N; i++) {
    stage0: mbarrier_expect_tx + tma_load
    stage1: mbarrier_wait + compute
}

// 展开后（3-阶段）
// === Prologue ===
stage0(tile=0);  // 预加载
stage0(tile=1);

// === Main Loop ===
for (int i = 0; i < N-2; i++) {
    stage1(tile=i);     // compute tile i
    stage0(tile=i+2);   // load tile i+2
}

// === Epilogue ===
stage1(tile=N-2);
stage1(tile=N-1);
```

Pipeline

Phase-2: OptimizeForTarget

- RewriteWgmmaSync
  - Warp-Group Matrix-Multiply-Accumulate，Hopper 的 Tensor Core 指令。
  - 特点：
    - 1. 以 4 个 warp为单位执行异步，发射后立即返回
    - 2. 需要显式同步：warpgroup_wait<N>

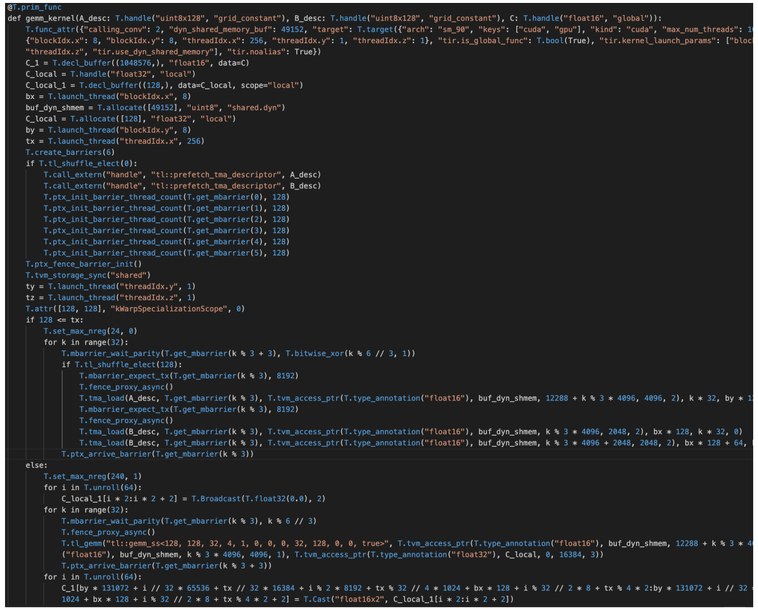
```
// 原始（无同步）
wgmma.mma_async(...);  // 发射 WGMMA 1
wgmma.mma_async(...);  // 发射 WGMMA 2
// ... 立即使用 accumulator

// 改写（插入组同步）
wgmma.mma_async(...);          // 发射 WGMMA 1
wgmma.mma_async(...);          // 发射 WGMMA 2
warpgroup_wait<0>;             // 等待所有 WGMMA 完成
// ... 安全使用 accumulator
```

- warpgroup_wait<N> 语义：
  - N=0：等待所有未完成的 WGMMA； N=1：等待除最近 1 条外的所有 WGMMA
- 收集同一作用域内的所有 WGMMA ，在最后一个之前插入 warpgroup_wait<N>，合并多余的等待语句

Pipeline

Phase-2: OptimizeForTarget

- Conclusion: 经过13个pass
  - 创建了 mbarrier
  - 构造了缓冲（环形阶段）
  - 拆分了 Producer/Consumer（Warp Specialization）
  - 注入了 TMA（expect_tx/arrive/wait_parity）
  - 规划并展开了流水线（prologue/mainloop/epilogue）
  - 改写了 WGMMA 同步（warpgroup_wait）



```
@T.prim_func
def gemm_kernel(A_desc: T.handle("uint8x128", "grid_constant"), B_desc: T.handle("uint8x128", "grid_constant"), C: T.handle("float16", "global")):
    T.func_attr({"calling_conv": 2, "dyn_shared_memory_buf": 49152, "target": T.target({"arch": "sm_90", "keys": ["cuda", "gpu"], "kind": "cuda", "max_num_threads": 1
    {"blockIdx.x": 8, "blockIdx.y": 8, "threadIdx.x": 256, "threadIdx.y": 1, "threadIdx.z": 1}, "tir.is_global_func": T.bool(True), "tir.kernel_launch_params": ["bloc
    "threadIdx.x", "tir.use_dyn_shared_memory"], "tir.noalias": True})
    C_1 = T.decl_buffer((1048576,), "float16", data=C)
    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", 256)
    T.create_barriers(6)
    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.ptx_fence_barrier_init()
    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(32):
            T.mbarrier_wait_parity(T.get_mbarrier(k % 3 + 3), T.bitwise_xor(k % 6 // 3, 1))
            if T.tl_shuffle_elect(128):
                T.mbarrier_expect_tx(T.get_mbarrier(k % 3), 8192)
                T.fence_proxy_async()
                T.tma_load(A_desc, T.get_mbarrier(k % 3), T.tvm_access_ptr(T.type_annotation("float16"), buf_dyn_shmem, 12288 + k % 3 * 4096, 4096, 2), k * 32, by *
                T.mbarrier_expect_tx(T.get_mbarrier(k % 3), 8192)
                T.fence_proxy_async()
                T.tma_load(B_desc, T.get_mbarrier(k % 3), T.tvm_access_ptr(T.type_annotation("float16"), buf_dyn_shmem, k % 3 * 4096, 2048, 2), bx * 128, k * 32, 0)
                T.tma_load(B_desc, T.get_mbarrier(k % 3), T.tvm_access_ptr(T.type_annotation("float16"), buf_dyn_shmem, k % 3 * 4096 + 2048, 2048, 2), bx * 128 + 64,
            T.ptx_arrive_barrier(T.get_mbarrier(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(32):
            T.mbarrier_wait_parity(T.get_mbarrier(k % 3), k % 6 // 3)
            T.fence_proxy_async()
            T.tl_gemm("tl::gemm_ss<128, 128, 32, 4, 1, 0, 0, 0, 32, 128, 0, 0, true>", T.tvm_access_ptr(T.type_annotation("float16"), buf_dyn_shmem, 12288 + k % 3 *
            ("float16"), buf_dyn_shmem, k % 3 * 4096, 4096, 1), T.tvm_access_ptr(T.type_annotation("float32"), C_local, 0, 16384, 3))
            T.ptx_arrive_barrier(T.get_mbarrier(k % 3 + 3))
        for i in T.unroll(64):
            C_1[by * 131072 + i // 32 * 65536 + tx // 32 * 16384 + i % 2 * 8192 + tx % 32 // 4 * 1024 + bx * 128 + i % 32 // 2 * 8 + tx % 4 * 2:by * 131072 + i // 32
            1024 + bx * 128 + i % 32 // 2 * 8 + tx % 4 * 2 + 2] = T.Cast("float16x2", C_local_1[i * 2:i * 2 + 2])
```

Benchmarks

Benchmark Results

Dense GEMM & Attention

- ***H100 SXM5 80GB, FP16***
- ***GEMM: MNK = 8192***
- ***Attention: Seqlen = 4096, Heads = 32, Dim = 128***

- 参考性能，来源是开源仓库中的***example***

	GEMM	Attention
CUTLASS	830	602
CuTeDSL	806	650
TileLang	743	606
TLX	665	594
Triton	635	515

Conclusion

Conclusion

Just use DSLs to write kernels

维度	**CuTeDSL**	**TileLang**	**Triton**	**TLX**
易用性	较低，需要深入理解 CUTLASS概念	中等	较高	中等 基于Triton改进
性能上限	最高 匹配手写CUDA性能	较高	中等 通用性好但峰值受限	中等偏上 针对性优化
优化表达空间	最丰富 可控制底层硬件细节	丰富 Pass配置+硬件特定优化	适中 高级抽象易用但限制多	较丰富 扩展Triton优化能力
硬件适配性	专门为NVIDIA优化	良好 多后端支持	良好 支持多硬件平台	专注NVIDIA特性的扩展
生态成熟度	NVIDIA官方维护	中等 快速发展中	最高 社区活跃用户多	较低

Thanks