TileLang Ascend开发入门

CANN

目录

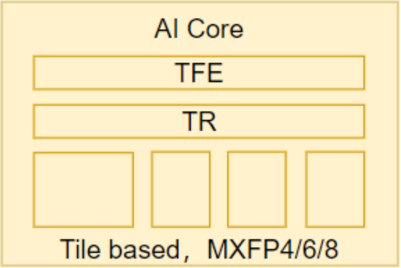
趋势与机遇：基于Tile的软硬件协同

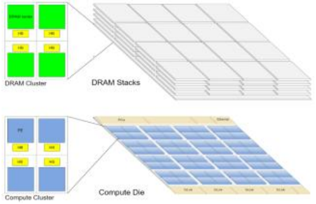
AI加速器发展的新趋势使得基于Tile编程语言变得越来越重要。

Tile-AI加速硬件

- 更高的数据传输和计算效率
- 本地大容量SRAM，再结合3D DRAM，片上互联，提供超高计算带宽
- 要求以更大的Tile粒度来进行指令集设计和体系结构设计

新一代Tile-based AI加速





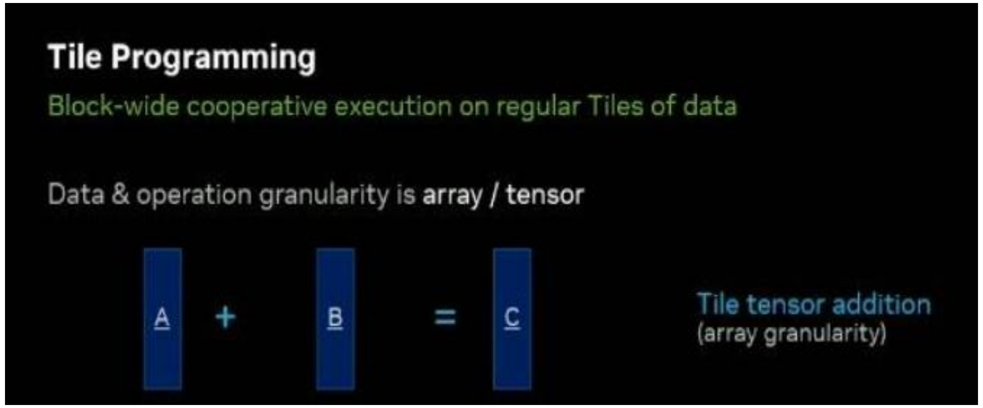
3D先进封装存储

片上网络

推动

Tile-编程语言

- 在更大的Tile粒度上进行编程和编译，可以更方便和高效地优化模型算子
- 实现Tile编程计算抽象和Tile硬件抽象一致，高效协同

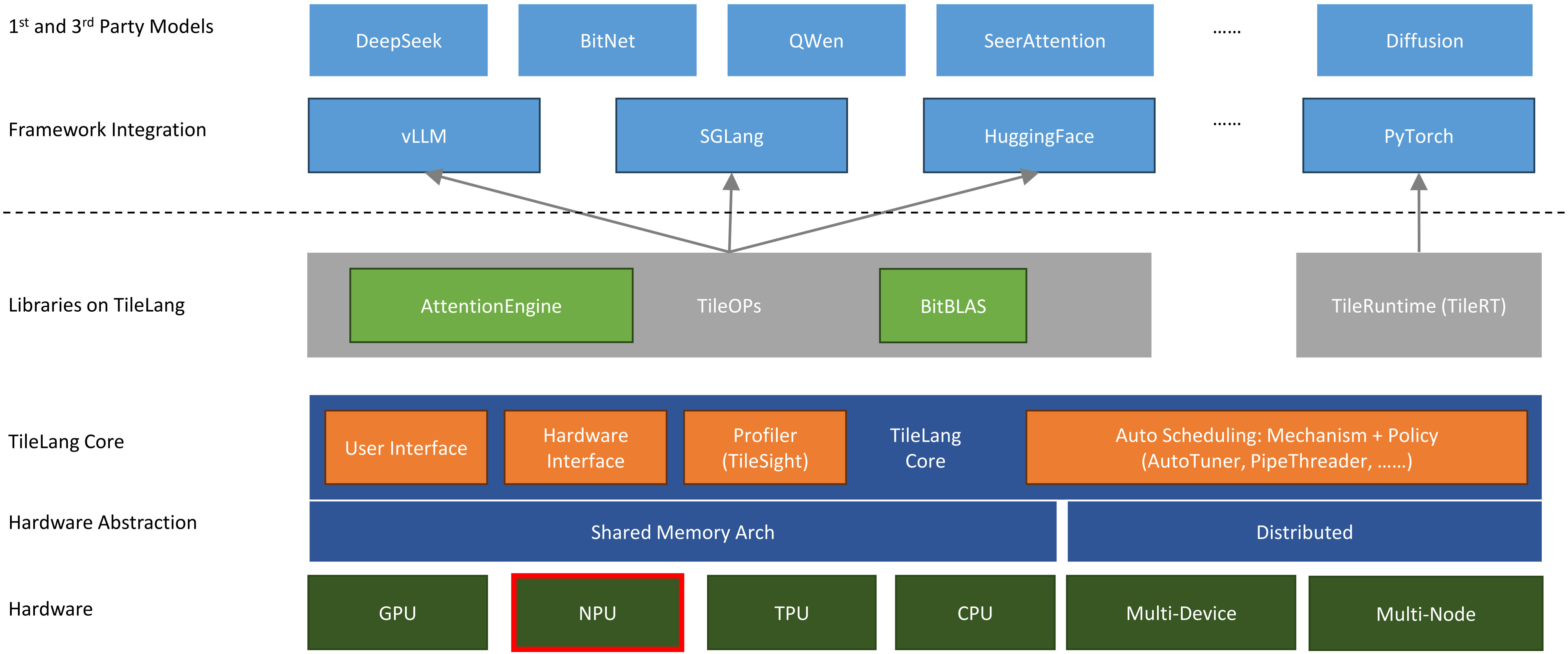


CANN

Tile编程语言：TileLang

- Tile-level的类Python的AI编程语言（DSL）
  - 开发更简单
  - 提供了更好的性能
  - 可以更加细粒度地控制计算调度
- 面对 Tile 硬件趋势的兴起，推出Tile编程语言TileLang，主要核心技术团队来自北京大学
- 2025-01-20开源，现已获得超过3.7K stars
  https://github.com/tile-ai/tilelang
- 支持多硬件后端
  GPU、NPU、TPU、CPUs

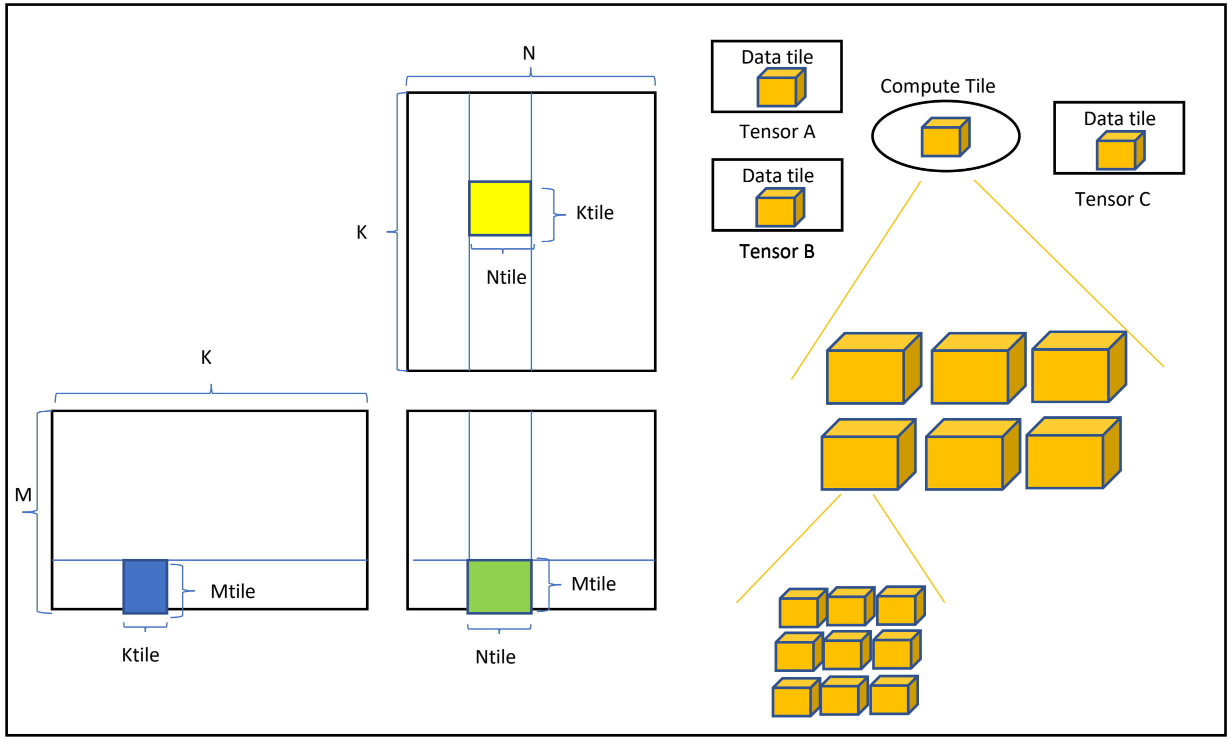
TileLang系统和生态



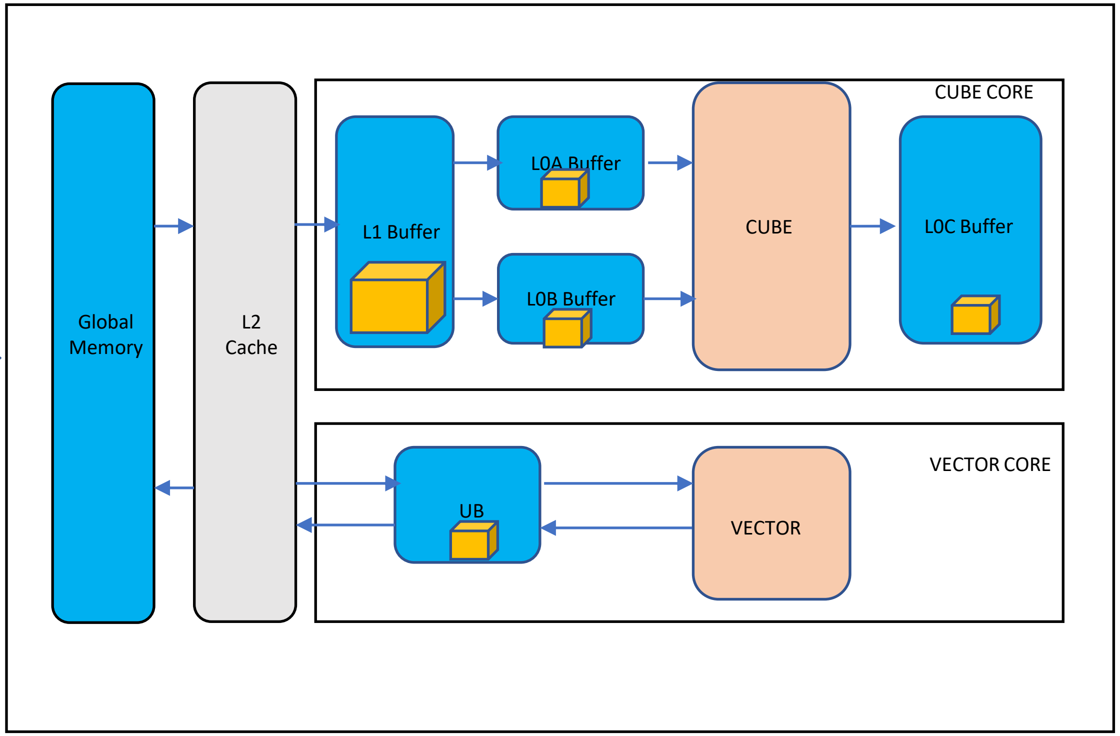
TileLang：核心设计理念

TileLang = Tile-VM + API原语

Tile-VM = Tile Task 模型 + Tile 硬件抽象



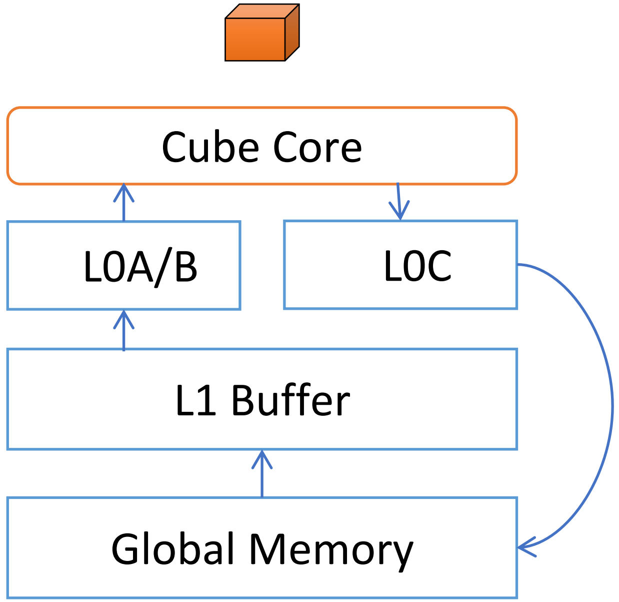
层次化 Tile Task模型



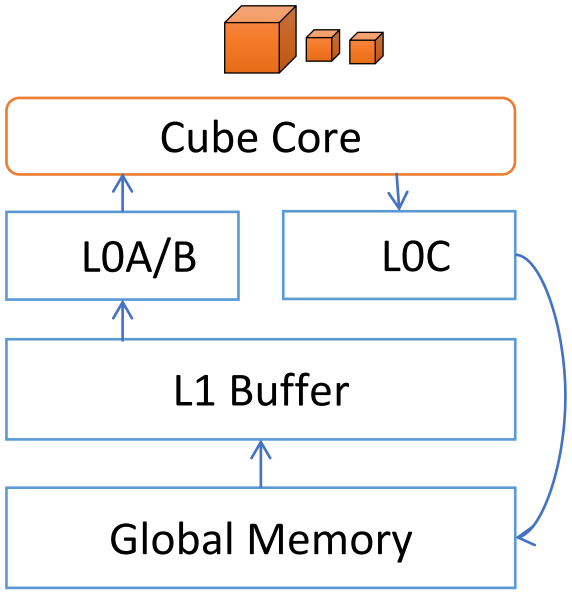
Tile 硬件抽象

TileLang API原语：Tile Allocation

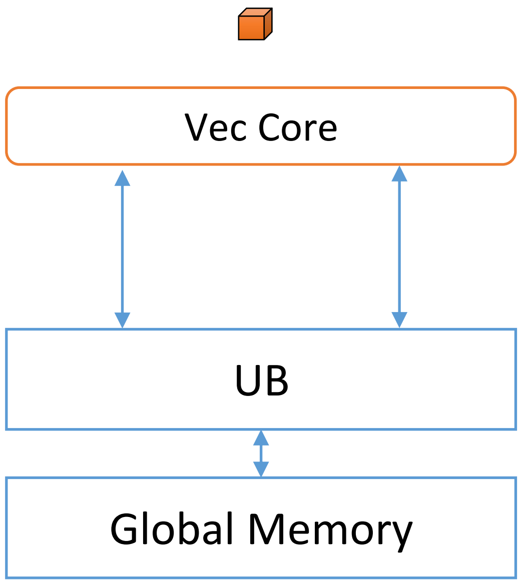
- T.alloc_L1



- T.alloc_l0a/b/c

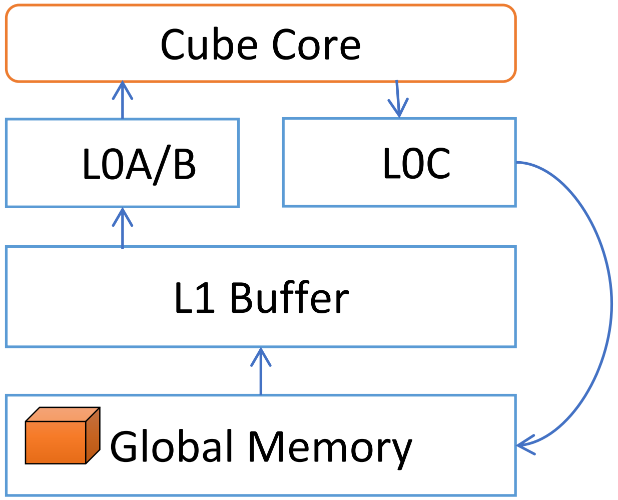


- T.alloc_UB

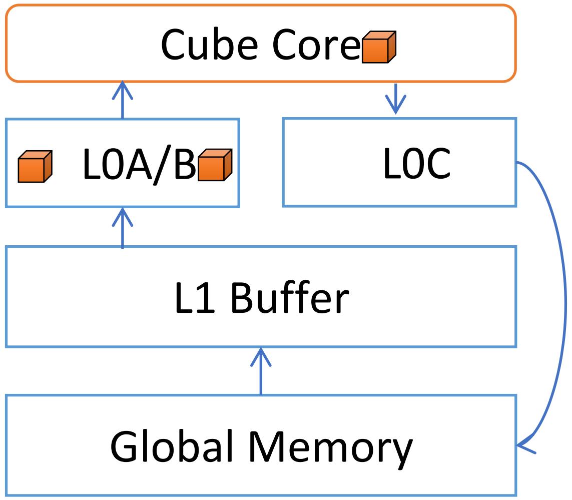


TileLang API原语： Tile Operation

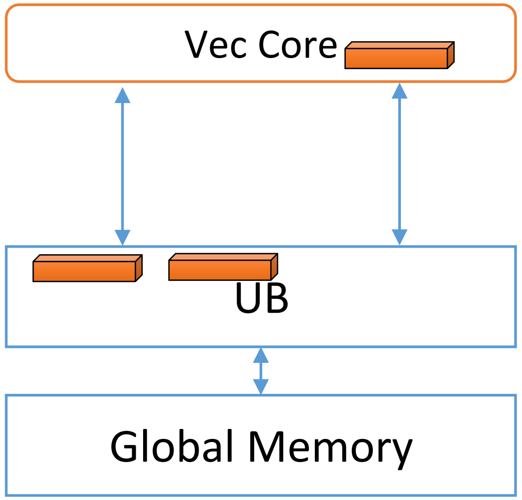
- T.copy()



- T.gemm()



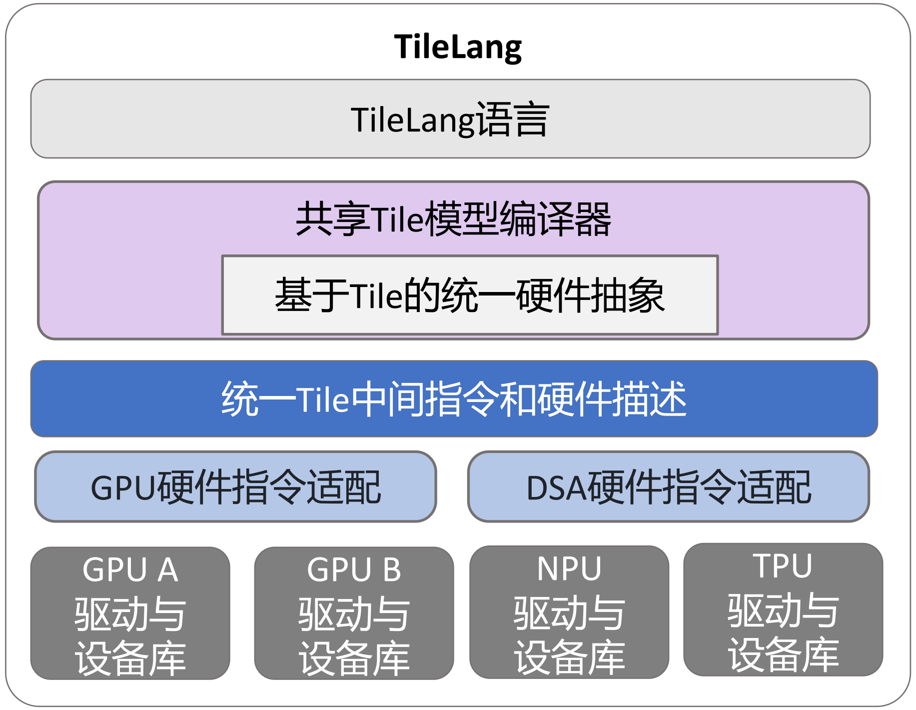
- T.reduce()

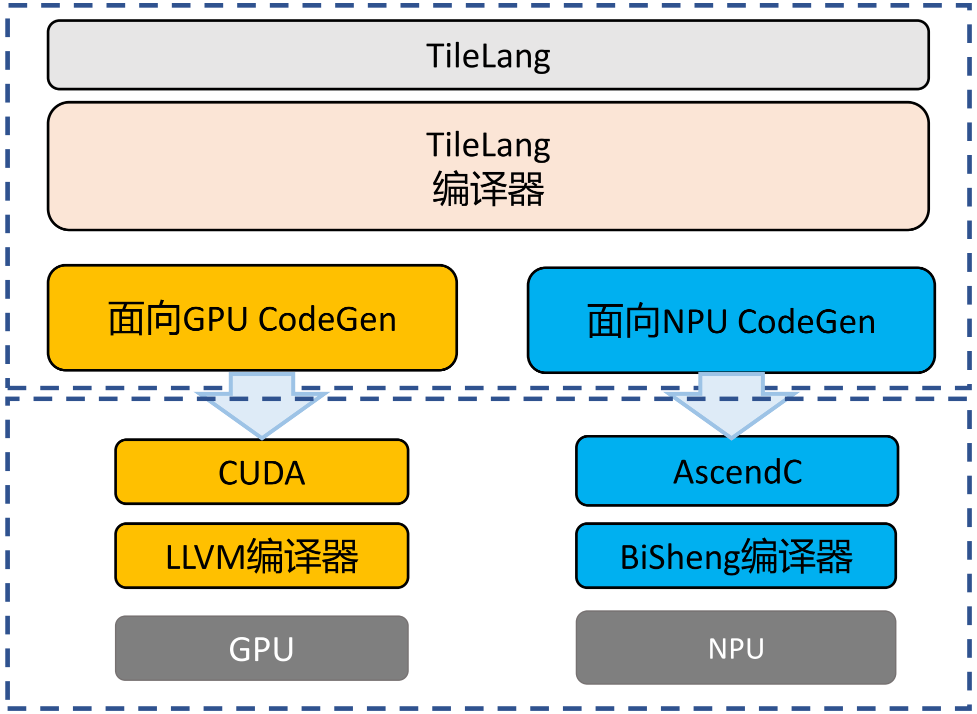


目录

TileLang Ascend适配

Ascend适配TileLang：基于Tile的统一硬件表达的抽象，提供轻量级的底层装换，并提供扩展接口提供Ascend专属能力。





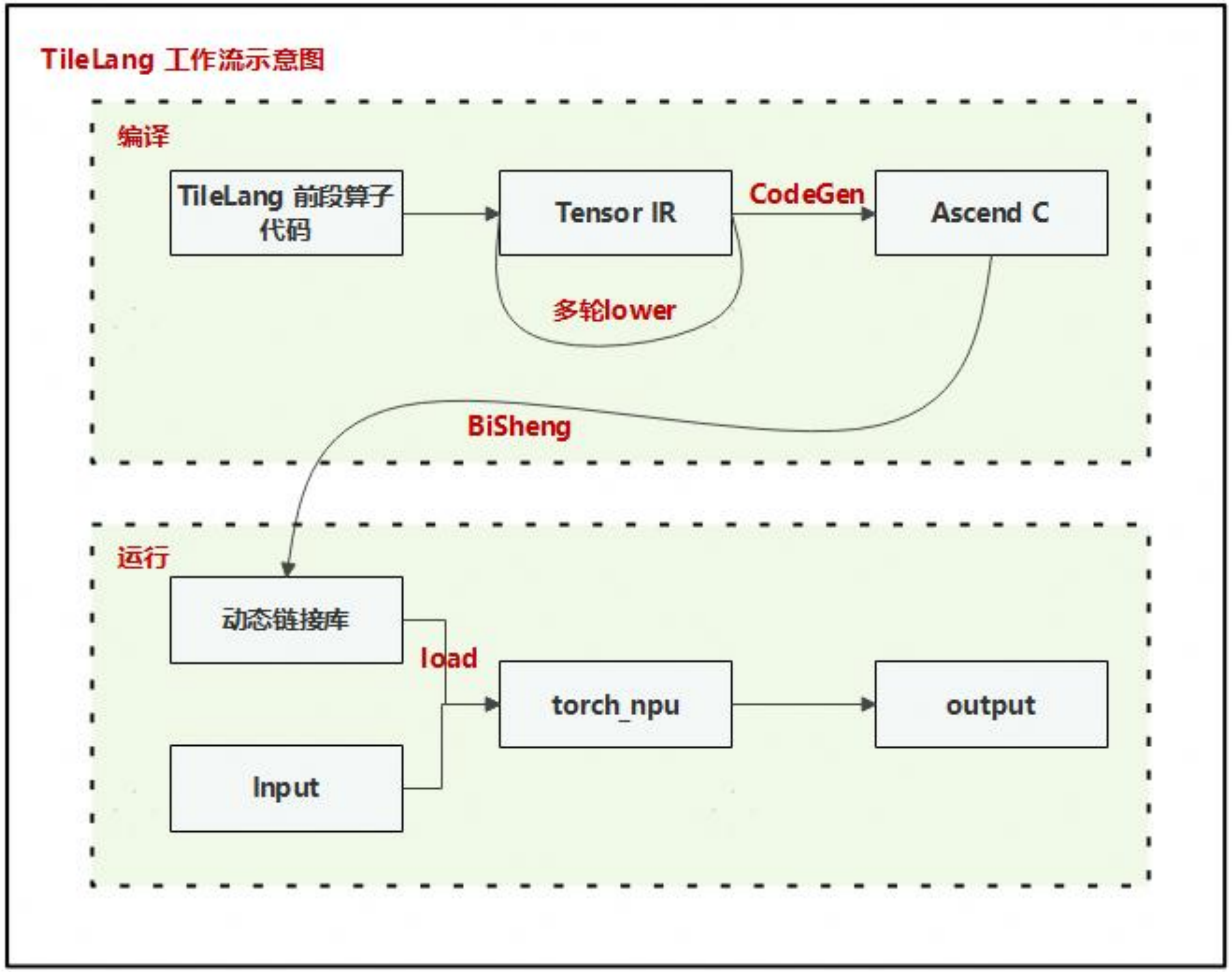
TileLang Ascend工作流程

编译阶段

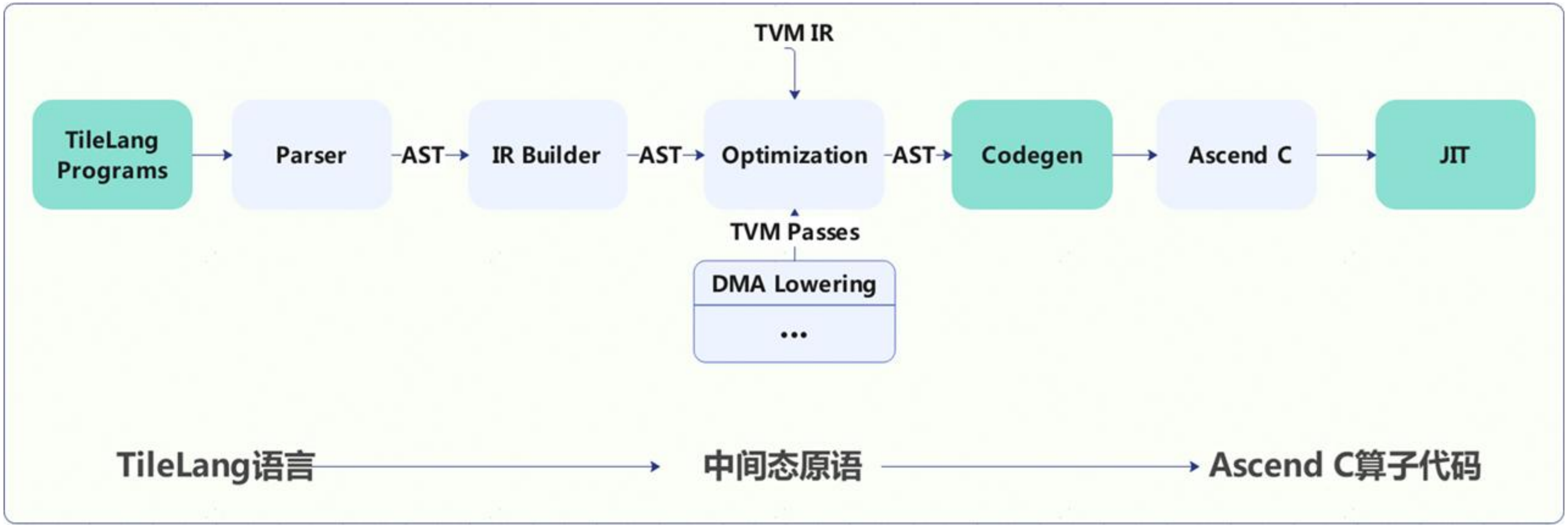
- 多级Lowering转换：TileLang算子根据NPU硬件特性进行多级降级，生成针对昇腾硬件优化的TensorIR表示。
- AscendC代码生成：基于TensorIR，使用专门的Ascend CodeGen模块生成对应的AscendC代码。
- 动态库编译：通过毕昇编译器（BiSheng）将AscendC代码编译成动态链接库（.so文件）。

运行阶段

- 库文件加载：通过torch_npu运行时库将.so文件加载到Python环境中
- 函数封装：将算子封装为可调用的Python函数对象
- 执行调用：用户以普通Python函数方式调用，提供输入张量即可在昇腾NPU上执行计算并获得结果



TileLang Ascend编译运行流程



- CodeGen

  通过预定义的AscendC指令模板，封装NPU底层硬件操作（如AI Core计算指令、DMAC数据传输指令）的语法细节与参数约束，CodeGen可自动解析TileLang语法树，根据语义逻辑匹配对应的指令模板，并通过指令映射规则，实现TileLang与AscendC NPU后端的无缝适配。

- JIT

  通过JIT调用CodeGen生成AscendC代码，并对整个过程进行动态调控，解决静态编译的局限性，确保生成的AscendC代码动态适配NPU特性，同时最大化提升算子执行效率。

TileLang Ascend原语介绍

原语类型	原语名称	用法示例	功能介绍
内核原语	kernel	T.kernel(block_num, is_npu=True) as (cid, vid)	该原语对应到Ascend C的kernel调用处，其中block_num对应于<<< >>>中的第一个参数，表示这个kernel开启多少个子任务数量。cid的范围为 [0,block_num)[0, block_num)[0,*block_num*), vid的范围为0或1。因为A2的cv核默认配比为1:2, 可以通过vid指定当前vector的索引。
内存分配原语	alloc_L1/L0A/L0B/L0C/UB	T.alloc_L1/L0A/L0B/L0C/UB(shape, dtype)	用于分配位于片上内存的buffer；通过指定shape和数据类型标记buffer的信息。
数据搬运原语	copy	T.copy(src, dst)	将src上的buffer拷贝到dst上，注意buffer可以通过BufferLoad或者BufferRegion指定一小块区域。
计算原语	gemm, add, mul, reduce_max...	T.reduce_max(dst, src, tmp, dim)	其中dim指定为对应规约的维度，目前只支持二维的规约。
同步原语	set_flag, wait_flag, set_cross_flag, wait_cross_flag, pipe_barrier	T.set_cross_flag(pipe: str, eventId: int)	其中pipe为需要同步的流水线，eventId为同步事件编号。

目录

TileLang基础语法结构

```python
@tilelang.jit(out_idx=[-1])
def matmul(M, N, K, block_M, block_N, K_L1, dtype="float16", accum_dtype="float"):
  m_num = M // block_M
  n_num = N // block_N

  @T.prim_func
  def main(
      A: T.Tensor((M, K), dtype),
      B: T.Tensor((K, N), dtype),
      C: T.Tensor((M, N), dtype),
  ):
    with T.Kernel(m_num * n_num, is_npu=True) as (cid, _):
      bx = cid // n_num
      by = cid % n_num

      A_L1 = T.alloc_L1((block_M, K_L1), dtype)
      B_L1 = T.alloc_L1((K_L1, block_N), dtype)

      C_L0 = T.alloc_L0C((block_M, block_N), accum_dtype)

      with T.Scope("C"):
        loop_k = T.ceildiv(K, K_L1)
        for k in T.serial(loop_k):
          T.copy(A[bx * block_M, k * K_L1], A_L1)
          T.copy(B[k * K_L1, by * block_N], B_L1)

          T.barrier_all()
          T.gemm_v0(A_L1, B_L1, C_L0, init=(k == 0))

          T.barrier_all()

        T.copy(C_L0, C[bx * block_M, by * block_N])

  return main
```

```python
func = matmul(M, N, K, 128, 256, 64)
a = torch.randn(M, K).half().npu()
b = torch.randn(K, N).half().npu()
c = torch.empty(M, N).half().npu()
c = func(a, b)
```

@tilelang.jit 装饰器标识函数会被jit动态编译，out_idx表示主计算函数返回参数索引。

@T.prim_func 装饰器在模块内定义主计算函数。

T.Tensor声明数据类型Tensor数据缓冲区，并指定其形状和数据类型。

T.Kernel原语触发算子kernel调用，对应AscendC的kernel调用。

T.Scope标识代码的计算单元，” C”表示在cube核运行， “V”表示在vector核上运行。

算子具体计算逻辑。

返回主计算函数对象供调用。

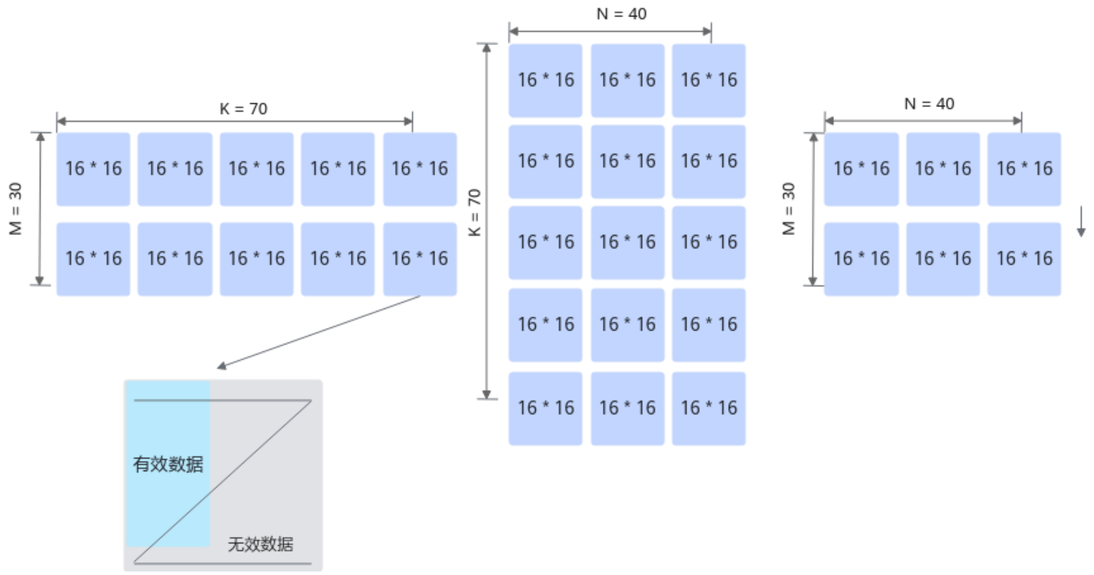
在主程序中调用TileLang算子，触发jit编译。调用主计算函数。

CANN

TileLang样例：GEMM_Basic

```
@tilelang.jit(out_idx=[-1])
def matmul(M, N, K, block_M, block_N, K_L1, dtype="float16", accum_dtype="float"):
  m_num = M // block_M
  n_num = N // block_N

  @T.prim_func
  def main(
      A: T.Tensor((M, K), dtype),
      B: T.Tensor((K, N), dtype),
      C: T.Tensor((M, N), dtype),
  ):
    with T.Kernel(m_num * n_num, is_npu=True) as (cid, _):
      bx = cid // n_num # 计算BlockM偏移量
      by = cid % n_num # 计算BlockN偏移量

      A_L1 = T.alloc_L1((block_M, K_L1), dtype) # 申请L1空间
      B_L1 = T.alloc_L1((K_L1, block_N), dtype)

      C_L0 = T.alloc_L0C((block_M, block_N), accum_dtype) # 申请L0C空间

      with T.Scope("C"):
        loop_k = T.ceildiv(K, K_L1) # 计算k轴上按基本块大小K_L1切分的次数
        for k in T.serial(loop_k):
          T.copy(A[bx * block_M, k * K_L1], A_L1)
          T.copy(B[k * K_L1, by * block_N], B_L1)

          T.barrier_all()
          T.gemm_v0(A_L1, B_L1, C_L0, init=(k == 0)) # 当k为0时表示第一块基本块计算，将
L0C清0
          T.barrier_all()

        T.copy(C_L0, C[bx * block_M, by * block_N])

  return main
```

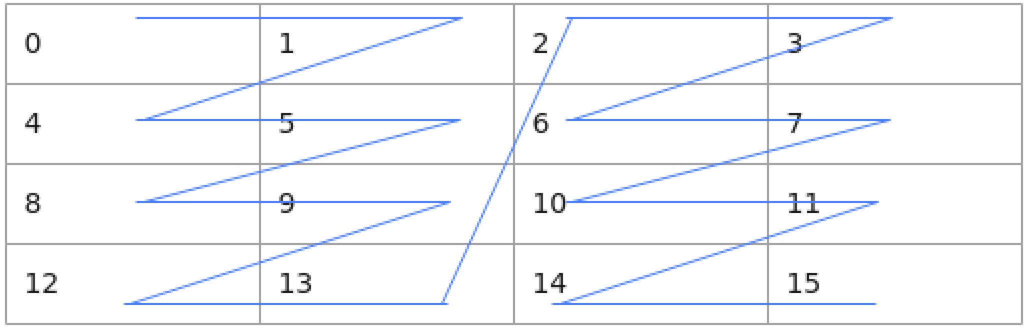
左侧为tilelang实现的matmul算子的基础样例。
主要功能：
实现A和B的分块矩阵乘法：C = A @ B。

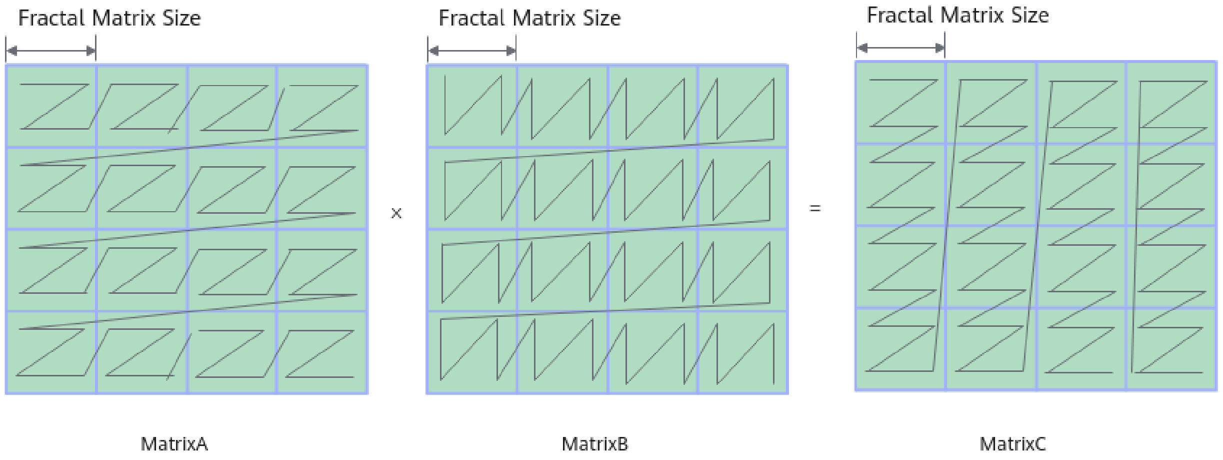


TileLang样例： GEMM_Highperf

```python
@tilelang.jit(out_idx=[-1]) # 此装饰器会被jit动态编译，out_idx表示kernel返回值索引
def matmul(M, N, K, block_M, block_N, block_K, K_L1, S1, S2, dtype="float16", accum_dtype="float"):
    m_num = M // block_M # M和N方向切分基本块数目
    n_num = N // block_N

    core_num = 20

    @T.macro
    def init_flag(): # 同步相关
        T.set_flag("mte1", "mte2", 0)
        T.set_flag("mte1", "mte2", 1)
        T.set_flag("m", "mte1", 0)
        T.set_flag("m", "mte1", 1)
        T.set_flag("fix", "m", 0)

    @T.macro
    def clear_flag():
        T.wait_flag("mte1", "mte2", 0)
        T.wait_flag("mte1", "mte2", 1)
        T.wait_flag("m", "mte1", 0)
        T.wait_flag("m", "mte1", 1)
        T.wait_flag("fix", "m", 0)

    @T.prim_func
    def main(
            A: T.Tensor((M, K), dtype),
            B: T.Tensor((K, N), dtype),
            C: T.Tensor((M, N), dtype),
    ):
        with T.Kernel(core_num, is_npu=True) as (cid, _):
            A_L1 = T.alloc_L1((S1, block_M, K_L1), dtype) # L1空间申请，S1表示L1上DB数目
            B_L1 = T.alloc_L1((S1, K_L1, block_N), dtype)

            T.annotate_layout({# 指定L1上的内存排布，后续将tilelang前端的偏移量自动转化为实际内存的偏移量
                A_L1: make_zn_layout(A_L1),
                B_L1: make_zn_layout(B_L1),
            })

            A_L0 = T.alloc_L0A((S2, block_M, block_K), dtype) # L0空间申请，S2表示L2上DB数目
            B_L0 = T.alloc_L0B((S2, block_K, block_N), dtype)
            C_L0 = T.alloc_L0C((block_M, block_N), accum_dtype)
```

Nd->Nz示意:



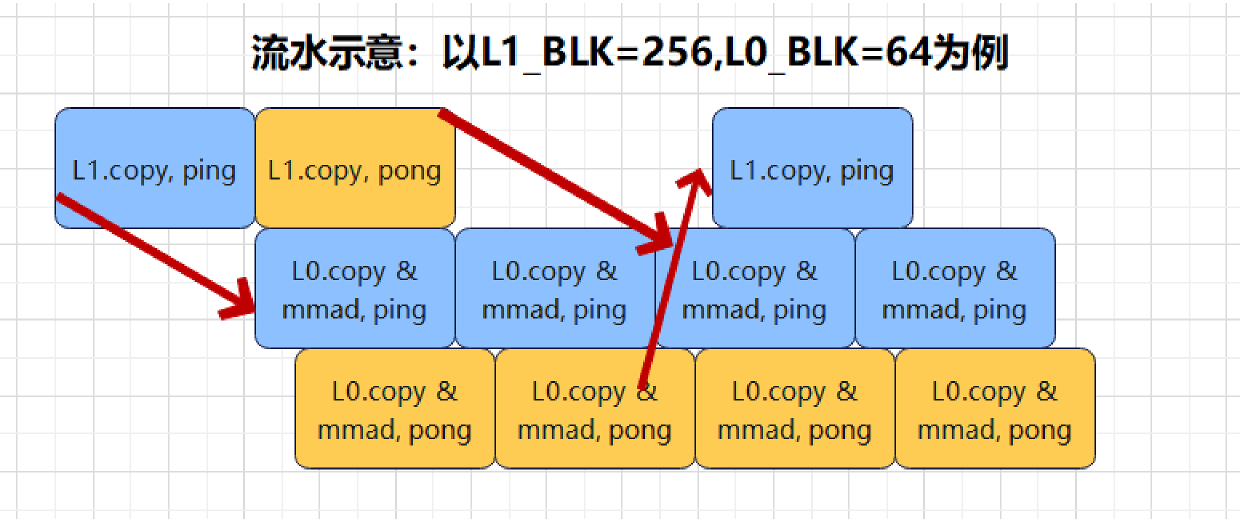


CANN

TileLang样例： GEMM_Highperf

```
with T.Scope("C"):
  init_flag() # 发射初始同步
  for i in T.serial(T.ceildiv(m_num * n_num, core_num)):
    T.use_swizzle(i * core_num + cid, M, N, K, block_M, block_N, off=3, in_loop=True) # 增加L2cache命中率
    bx = cid // n_num  # 计算BlockM偏移量
    by = cid % n_num  # 计算BlockN偏移量
    loop_k = T.ceildiv(K, K_L1) # 计算k方向在l1上的循环次数
    T.wait_flag("mte1", "mte2", 0)
    T.copy(A[bx * block_M, 0], A_L1[0, :, :]) # 提前搬运矩阵基本块，先使用第一块L1buffer
    T.copy(B[0, by * block_N], B_L1[0, :, :])
    T.set_flag("mte2", "mte1", 0)
    T.wait_flag("fix", "m", 0)
    for k in T.serial(loop_k): # 计算k方向在l1上的循环次数
      if k < loop_k - 1: # 由于提前搬运了一次，不执行完毕
        T.wait_flag("mte1", "mte2", (k + 1) % S1)
        T.copy(A[bx * block_M, (k + 1) * K_L1], A_L1[(k + 1) % S1, :, :]) # 按照 pingpong进行搬运
        T.copy(B[(k + 1) * K_L1, by * block_N], B_L1[(k + 1) % S1, :, :])
        T.set_flag(“mte2”, “mte1”, (k + 1) % S1)
      loop_kk = T.ceildiv(K_L1, block_K) # 计算L0上循环次数
      for kk in T.serial(loop_kk):
        if kk == 0: # 该分支用来等待提前发射的同步
          T.wait_flag("mte2", "mte1", k % S1)
        T.wait_flag("m", "mte1", kk % S2)
        T.copy(A_L1[k % S1, 0, kk * block_K], A_L0[kk % S2, :, :]) # 计算L0上循环次数
        T.copy(B_L1[k % S1, kk * block_K, 0], B_L0[kk % S2, :, :])
        if kk == 3: # L0全部搬运完毕，通知L1继续搬运
          T.set_flag("mte1", "mte2", k % S1)
        T.set_flag("mte1", "m", kk % S2)
        T.wait_flag(“mte1”, “m”, kk % S2)
        if k == 0 and kk == 0: # 此分支表示L0C清0，else分支表示L0C累加
          T.mma(A_L0[kk % S2, :, :], B_L0[kk % S2, :, :], C_L0, init=True)
        else:
          T.mma(A_L0[kk % S2, :, :], B_L0[kk % S2, :, :], C_L0)
        T.set_flag("m", "mte1", kk % S2)

    T.set_flag("m", "fix", 0)
    T.wait_flag("m", "fix", 0)
    T.copy(C_L0, C[bx * block_M, by * block_N]) # 计算完毕，根据偏移量搬运到GM中对应位置
    T.set_flag(“fix”, “m”, 0)
  clear_flag()
  T.barrier_all()
```

下方为左侧代码流水示意图：



CANN

TileLang样例：GEMM_函数调用

```
func = matmul(M, N, K, 128, 256, 64, 256, 2, 2)
torch.manual_seed(0)

a = torch.randn(M, K).half().npu()
b = torch.randn(K, N).half().npu()

c = func(a, b)
ref_c = a @ b

torch.npu.synchronize()

torch.testing.assert_close(c, ref_c, rtol=1e-2, atol=1e-2)
print("Kernel Output Match!")

tilelang_time = do_bench(lambda: func(a, b))
torch_time = do_bench(lambda: a @ b)

print(f"tilelang time: {tilelang_time} ms")
print(f"torch time: {torch_time} ms")
```

与torch_npu高性能算子的耗时比较：
已达到torch_npu高性能算子的70%左右的性能。

```
Kernel Output Match!
tilelang time: 0.8647611737251282 ms
torch time: 0.583781361579895 ms
```

如何调试TileLang Ascend算子

当前tilelang调试还需依赖AscendC调试工具来进行

步骤一：通过kernel.get_kernel_source()获取codegen后的AscendC源码

```
func = matmul(M, N, K, 128, 256, 64, 256, 2, 2)

print(func.get_kernel_source())
```

步骤二：编写静态调用脚本，在kernel入参中加入调试所需的workspace

```
dump_workspace = torch.empty(75 * 1024 * 1024 // 4).float().npu() # 初始化workspace空间

stream = torch.npu.current_stream()._as_parameter_


def tl_gemm():
   return lib.call(
      ctypes.c_void_p(a.data_ptr()),
      ctypes.c_void_p(b.data_ptr()),
      ctypes.c_void_p(c.data_ptr()),
      ctypes.c_void_p(dump_workspace.data_ptr()), # 加入到kernel入参中
      stream)

tl_gemm()
torch.npu.synchronize()
dump_workspace.detach().cpu().numpy().tofile("dump_workspace.bin") # 将结果作为.bin文件保存
```


CANN

如何调试TileLang Ascend算子

步骤三：将翻译后的AscendC代码加入相关初始化代码

```
extern "C" void call(uint8_t* A_handle, uint8_t* B_handle, uint8_t* C_handle, GM_ADDR dbgspace, aclrtStream stream) {

main_kernel<<<256, nullptr, stream>>>(A_handle, B_handle, C_handle, dbgspace, fftsAddr);


#kenrel 内部
void main_kernel( GM_ADDR A_handle,  GM_ADDR B_handle,  GM_ADDR C_handle, GM_ADDR dbgspace, uint64_t fftsAddr) {
  AscendC::InitDump(true, dump_workspace, 1024 * 1024);
  AscendC::PRINTF("cid = %d\n", cid);
```

步骤四：编译->执行->查看debug结果

```
bash build.sh example_gemm.cpp
python test_example_gemm.py
show_kernel_debug_data dump_workspace.bin ./
```

```
================ block.0 begin ==============
[Meta Info] block num: 256, core type: AIC, isMix: True
cid = 206
================ block.0 end ================
================ block.1 begin ==============
[Meta Info] block num: 256, core type: AIC, isMix: True
cid = 207
================ block.1 end ================
================ block.2 begin ==============
[Meta Info] block num: 256, core type: AIC, isMix: True
cid = 208
```

参考用例：
https://github.com/tile-ai/tilelang-ascend/tree/ascendc_pto/examples/gemm_aot

目录

TileLang Ascend后续计划

Framework Enhancement

- [ ] Automatic separation of cube and vector code, and elimination of explicit allocation of workspace on global memory
- [ ] Automatic pipeline injection and insertion of neccessary synchronization between different hardware resources
- [ ] Automatic buffer reuse
- [x] Dynamic Shape Support, including automatic TilingData extraction
- [x] More complete tilelang primitives support
- [ ] PTO instruction support

Kernel Support

- [ ] Implementation of TileOPs using tilelang-ascend

Performance Optimization

- [ ] Optimization of instruction templates
- [ ] Complete integration into the existing network infrastructure
- [ ] Target comparable performance of corresponding AscendC programs

Tool development

- [ ] Profiling Tools for performance analysis

欢迎大家关注CANN开源社区和TileLang-Ascend开源社区！

https://gitcode.com/cann/cann-recipes-infer/tree/master/ops/tilelang

TileLang-Ascend

https://github.com/tile-ai/tilelang-ascend

Thank you.

社区愿景：打造开放易用、技术领先的AI算力新生态

社区使命：使能开发者基于CANN社区自主研究创新，构筑根深叶茂、跨产业协同共享共赢的CANN生态

Vision: Building an Open, Easy-to-Use, and Technology-leading AI Computing Ecosystem

Mission: Enable developers to independently research and innovate based on the CANN community and build a win-win CANN ecosystem with deep roots and cross-industry collaboration and sharing.

上CANN社区获取干货

关注CANN公众号获取资讯