

# TileLang-Ascend “Developer模式”

## 开启高效新范式

作者：傅后榆

时间：2026.1.26



# Tile编程语言： TileLang

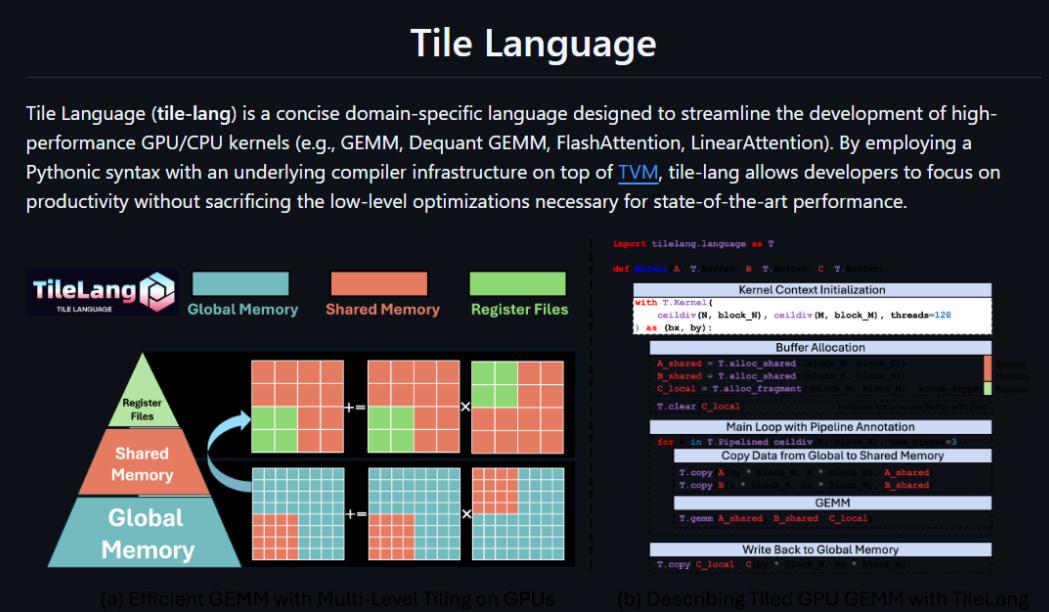
- 面对 Tile 硬件趋势的兴起，核心技术团队来自北京大学的 Tile-AI 社区推出 Tile 编程语言 TileLang
- Tile-level 的类 Python 的 AI 编程语言 (DSL)
  - 更简单的开发模式
  - 提供了更好的性能
  - 可以更加细粒度地控制计算调度
- 2025-01-20 开源，现已获得超过 4.8K stars  
<https://github.com/tile-ai/tilelang>
- 支持多硬件后端  
GPU、NPU、TPU、CPUs



The image shows the TileLang logo, which consists of the word "TileLang" in a large, stylized font with a pink-to-blue gradient, followed by a 3D hexagonal icon. Below the logo is the text "TILE LANGUAGE".

**Tile Language**

Tile Language (tile-lang) is a concise domain-specific language designed to streamline the development of high-performance GPU/CPU kernels (e.g., GEMM, Dequant GEMM, FlashAttention, LinearAttention). By employing a Pythonic syntax with an underlying compiler infrastructure on top of [TVM](#), tile-lang allows developers to focus on productivity without sacrificing the low-level optimizations necessary for state-of-the-art performance.



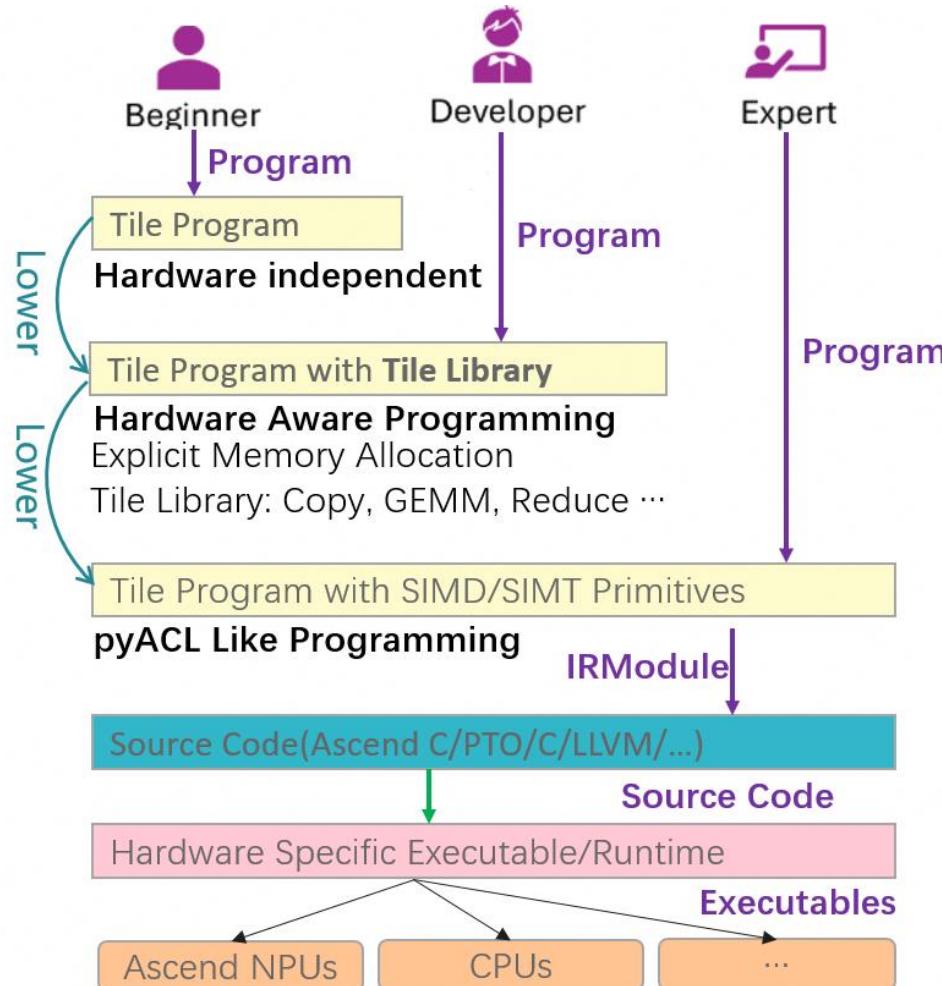
(a) Efficient GEMM with Multi-Level Tiling on GPUs

(b) Describing Tiled GPU GEMM with TileLang

The diagram illustrates a multi-level tiling strategy for a GEMM operation. It shows a pyramid structure with three levels: Global Memory at the base, Shared Memory in the middle, and Register Files at the top. The computation is divided into four main stages: Main Loop with Pipeline Annotation, Copy Data from Global to Shared Memory, GEMM, and Write Back to Global Memory. The code examples show how these stages are implemented using TileLang's DSL.

```
import tilelang.language as T
def Matmul A T.buffer B T.buffer C T.buffer:
    Kernel Context Initialization
    with T.Kernel(
        ceildiv(N, block_N), ceildiv(M, block_M), threads=128
    ) as (tx, by):
        Buffer Allocation
        A_shared = T.alloch_shared(tx * block_N, tx * block_M)
        B_shared = T.alloch_shared(tx * block_N, by * block_M)
        C_local = T.alloch_fragment(tx * block_M, by * block_N, access_dtypes=[float])
        T.clear C_local
        Main Loop with Pipeline Annotation
        for int T.Pipelined ceildiv(tx * block_N, block_N), num_stages=3:
            Copy Data from Global to Shared Memory
            T.copy A[tx * block_N : tx * block_N + block_N] A_shared
            T.copy B[tx * block_N : tx * block_N + block_N] B_shared
            GEMM
            T.gemm A_shared B_shared C_local
        Write Back to Global Memory
        T.copy C_local C[tx * block_M : tx * block_M, by * block_N]
```

# TileLang Ascend 开发模式

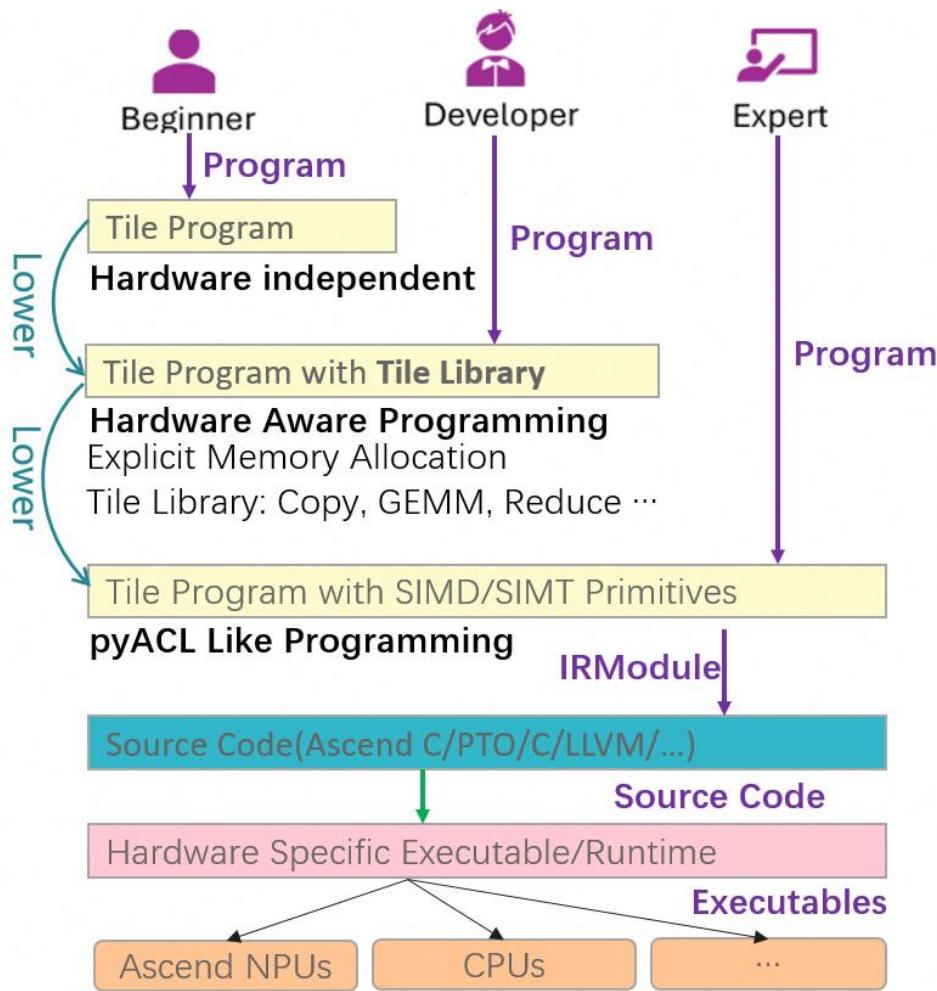


## Expert

```
with T.Kernel(m_num * n_num, is_npu=True) as (cid, vid):
    bx = cid // n_num
    by = cid % n_num
    a_ub = T.alloc_ub(block_M // VEC_NUM, block_N, dtype)
    b_ub = T.alloc_ub(block_M // VEC_NUM, block_N, dtype)
    c_ub = T.alloc_ub(block_M // VEC_NUM, block_N, dtype)
    with T.Scope("V"):
        T.copy(A[bx * block_M + vid * block_M // VEC_NUM, by * block_N], a_ub)
        T.copy(B[bx * block_M + vid * block_M // VEC_NUM, by * block_N], b_ub)
        T.set_flag("mte2", "v", 0)
        T.wait_flag("mte2", "v", 0)
        T.tile.add(c_ub, a_ub, b_ub)
        T.set_flag("v", "mte3", 1)
        T.wait_flag("v", "mte3", 1)
        T.copy(c_ub, C[bx * block_M + vid * block_M // VEC_NUM, by * block_N])
```

细粒度控制硬件底层执行细节

# TileLang Ascend 开发模式



$C[i, j] = A[i, k] + B[k, j]$

## Simple Tensor Expression

```
with T.Kernel(m_num * n_num, is_npu=True) as (cid, vid):
    bx = cid // n_num
    by = cid % n_num
    a_ub = T.alloc_ub(block_M // VEC_NUM, block_N, dtype)
    b_ub = T.alloc_ub(block_M // VEC_NUM, block_N, dtype)
    c_ub = T.alloc_ub(block_M // VEC_NUM, block_N, dtype)
    T.copy(A[bx * block_M + vid * block_M // VEC_NUM, by * block_N], a_ub)
    T.copy(B[bx * block_M + vid * block_M // VEC_NUM, by * block_N], b_ub)
    T.tile.add(c_ub, a_ub, b_ub)
    T.copy(c_ub, C[bx * block_M + vid * block_M // VEC_NUM, by * block_N])
```

## Block-level Tile Programming

# Developer 特性总览

特性	用法示例	价值场景
硬件流水自动同步	配置JIT编译选项: tilelang.PassConfigKey.TL_ASCEND_AUTO_SYN_C	在Ascend NPU上，计算和搬运等单元是异步并行工作的。此特性 <b>自动插入同步指令</b> ，确保数据依赖正确，同时最大化硬件流水线的利用。
内存规划与复用	配置JIT编译选项： TL_ASCEND_MEMORY_PLANNING	<b>自动管理内存分配，尽可能复用缓冲区</b> ，减少内存占用，提高缓存利用率。
自动拆分CV指令	配置JIT编译选项： TL_ASCEND_AUTO_CV_COMBINE	Tilelang语句 <b>自动拆分Cube和Vector指令</b> ，分别在CV计算核心上执行，用户无需关注CV分离架构。
T.Pipelined 原语	for i_i in T.Pipelined(NI, num_stages=2):	用于显式声明一个流水线循环，编译器会自动进行 <b>流水线优化</b> ，包括数据搬运和计算的并行。
T.Persistent 原语	for bx, by in T.Persistent([T.ceildiv(M, block_M), T.ceildiv(N, block_N)], core_num, cid):	用于创建持久化调度。在持久化调度中，每个NPU核心会连续处理多个工作项，从而 <b>减少任务启动和调度的开销</b> 。
T.Parallel 原语	for i in T.Parallel(v_block):	将 <b>循环迭代空间并行化</b> ，转换为向量操作，利用向量核心并行执行。

# 硬件流水自动同步

## 为什么需要同步?

昇腾 NPU 芯片集成了多种硬件单元（如 Cube 计算单元、Vector 计算单元、数据搬运单元等），这些单元是异步并行工作的，因此需要同步指令解决数据依赖的时序问题

## 传统手动同步的挑战

- 容易遗漏
- 过度同步
- 死锁风险
- 代码冗长

## 自动插入同步指令

将用户从繁琐易错的同步管理中解放出来，让 NPU 内核开发更加接近传统的串行编程体验，同时保持了对硬件特性的充分利用。

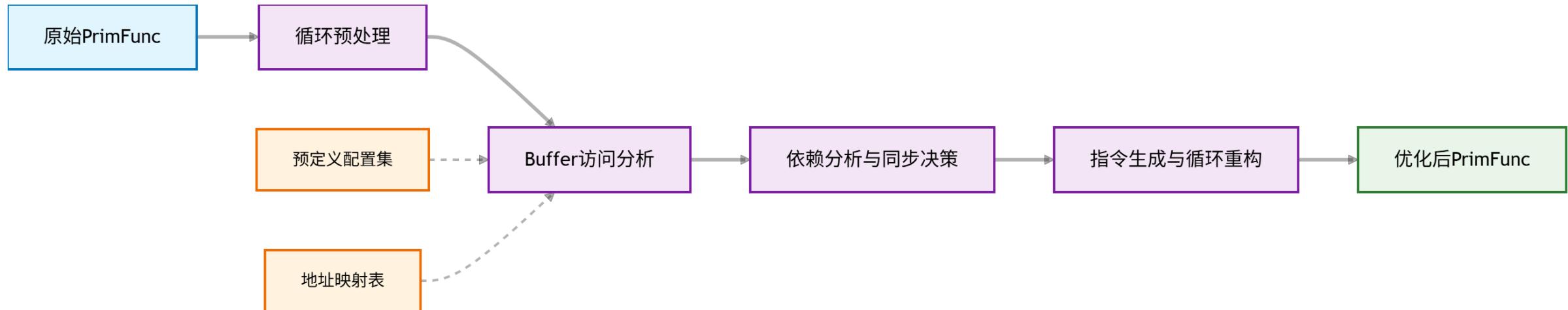
```
pass_configs = {
    tilelang.PassConfigKey.TL_ASCEND_AUTO_SYNC: True,
}
@tilelang.jit(out_idx=[-1], pass_configs=pass_configs)
def vec_add(M, N, block_M, block_N, dtype="float"):
    m_num = M // block_M
    n_num = N // block_N
    VEC_NUM = 2
    @T.prim_func
    def main(
        A: T.Tensor((M, N), dtype),
        B: T.Tensor((M, N), dtype),
        C: T.Tensor((M, N), dtype),
    ):
        with T.Kernel(m_num * n_num, is_npu=True) as (cid, vid):
            bx = cid // n_num
            by = cid % n_num
            a_ub = T.alloc_ub((block_M // VEC_NUM, block_N), dtype)
            b_ub = T.alloc_ub((block_M // VEC_NUM, block_N), dtype)
            c_ub = T.alloc_ub((block_M // VEC_NUM, block_N), dtype)
            with T.Scope("V"):
                T.copy(A[bx * block_M + vid * block_M // VEC_NUM, by * block_N], a_ub)
                T.copy(B[bx * block_M + vid * block_M // VEC_NUM, by * block_N], b_ub)
                # T.set_flag("mte2", "v", 0)
                # T.wait_flag("mte2", "v", 0)
                T.tile.add(c_ub, a_ub, b_ub)
                # T.set_flag("v", "mte3", 1)
                # T.wait_flag("v", "mte3", 1)
                T.copy(c_ub, C[bx * block_M + vid * block_M // VEC_NUM, by * block_N])
return main
```

配置[自动插入核内同步指令]编译选项

无需手动插入同步指令，降低开发心智

# 硬件流水自动同步

解决昇腾 NPU 算子开发中手动插入同步指令的痛点，通过自动化分析实现同步指令的精准生成，兼顾算子正确性与运行性能。



- **循环预处理**: 通过将for循环递归展开两份，准确分析嵌套循环中的依赖关系
- **Buffer 访问分析**: 结合预定义配置集确定操作所属的硬件流水线；解析对Buffer 的读写操作
- **依赖分析与同步决策**: 识别 数据依赖并根据硬件特性决策同步类型，通过同步图剔除冗余指令。
- **指令生成与循环重构**: 将展开后的循环重构为原始嵌套结构，

# 内存规划与复用

## 传统手动规划内存与复用的挑战

- 开发效率极低：需人工计算各个层级下每个 Buffer 的 Offset，维度 / dtype 变更需全量重算，迭代成本高
- 易出错且难调试：人工计算易出现地址重叠、32 字节对齐错误，触发 NPU 内存超限 / 执行异常，定位问题需逐行核对 Offset

## 解决的关键痛点

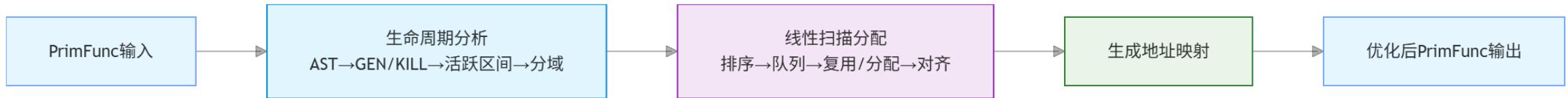
- **提效**：完全替代人工 Offset 计算，维度 / 类型变更无需手动适配，最大化利用昇腾 NPU 有限的共享内存资源
- **降险**：自动规避地址重叠、对齐错误，消除内存超限 / 执行异常的人为因素

```
pass_configs = {
    tilelang.PassConfigKey.TL_ASCEND_MEMORY_PLANNING: True,      配置[内存规划与复用]编译选项
}
@tilelang.jit(out_idx=[3], workspace_idx=[4,5,6], pass_configs=pass_configs)
def flash_attention_fwd(
    # ...other code
):
    # ...other code
@T.prim_func
def main(
    Q: T.Tensor(shape, dtype), # type: ignore
    # ...other code
):
    with T.Kernel(block_num, is_npu=True) as (cid, vid):
        # ...other code
        # T.annotate_address({
        #     # L1 address
        #     q_l1: 0,
        #     k_l1: block_M * dim * DataType(dtype).bits // 8,
        #     acc_s_l1: block_M * dim * DataType(dtype).bits // 8,
        #     v_l1: block_M * (block_N + dim) * DataType(dtype).bits // 8,
        #     # L0C address
        #     acc_s_l0c: 0,
        #     acc_o_l0c: 0,
        #     ## ub address
        #     acc_o: 0,
        #     sumexp: 65536,
        #     m_i: 65664,
        #     acc_s_ub: 66048,
        #     m_i_prev: 74240,
        #     acc_s_ub_: 74368,
        #     tmp_ub: 74368,
        #     sumexp_i_ub: 98944,
        #     acc_s_half: 98944,
        #     acc_o_ub: 98944,
        #     acc_o_half: 98944
        # })
```

无需手动规划各个内存层级上的buffer排布与内存复用

# 内存规划与复用

通过精准的缓冲区生命周期分析与高效的线性扫描内存分配算法，替代人工内存 Offset 配置，提高昇腾 NPU 内存利用率，适配昇腾 NPU 硬件约束，避免内存超限，提升算子开发效率



- 生命周期分析：

标记每个缓冲区的 GEN (生成) /KILL (销毁) 事件，精准界定其活跃区间（首次使用→最后一次使用的执行阶段）；

按昇腾硬件内存域分组缓冲区，适配不同分区的内存上限约束。

- 线性扫描分配算法：

按活跃区间起始位置排序缓冲区，构建线性执行序列；

维护活跃队列与空闲内存块池，处理每个缓冲区的分配需求；

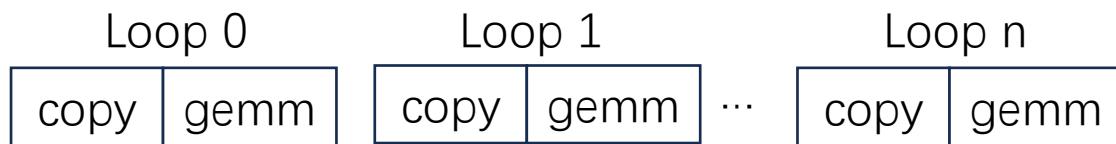
分配策略：优先复用已释放的空闲内存块，无可用块时分配新内存，

# T.Pipelined 流水并行

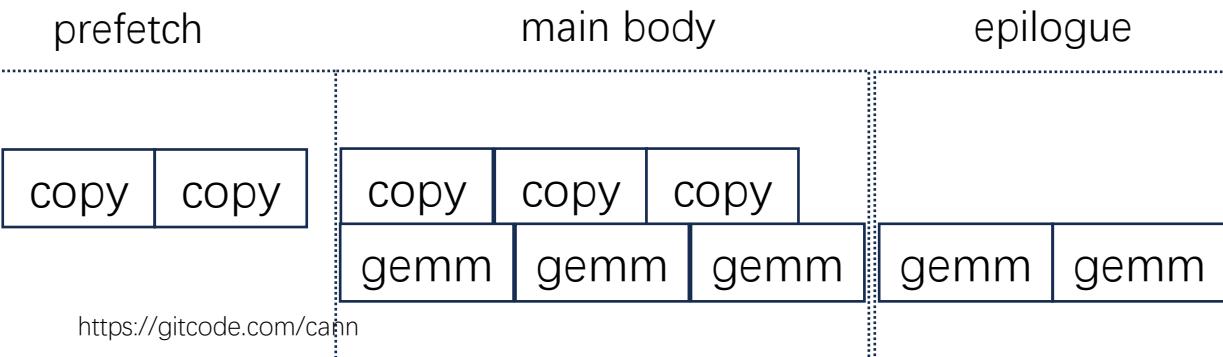
## 核心目标

通过编译时静态分析与代码变换，实现计算与内存访问的重叠执行，最大化硬件利用率，显著提升计算密集型任务的性能。

原始执行顺序：



使用pipeline后执行顺序：



```
@tilelang.jit(out_idx=[-1])
def matmul_add(M, N, K, block_M, block_N, block_K, 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, vid):
            bx = cid // n_num
            by = cid % n_num
            A_L1 = T.alloc_L1((block_M, block_K), dtype)
            B_L1 = T.alloc_L1((block_K, block_N), dtype)
            C_L0 = T.alloc_L0C((block_M, block_N), accum_dtype)

        with T.Scope("C"):

            loop_k = T.ceildiv(K, block_K)
            for k in T.Pipelined(loop_k, num_stages=2):
                T.copy(A[bx * block_M, k * block_K], A_L1)
                T.copy(B[k * block_K, by * block_N], B_L1)

                T.barrier_all()
                if k == 0:
                    T.gemm_v0(A_L1, B_L1, C_L0, init=True)
                else:
                    T.gemm_v0(A_L1, B_L1, C_L0)

                T.barrier_all()

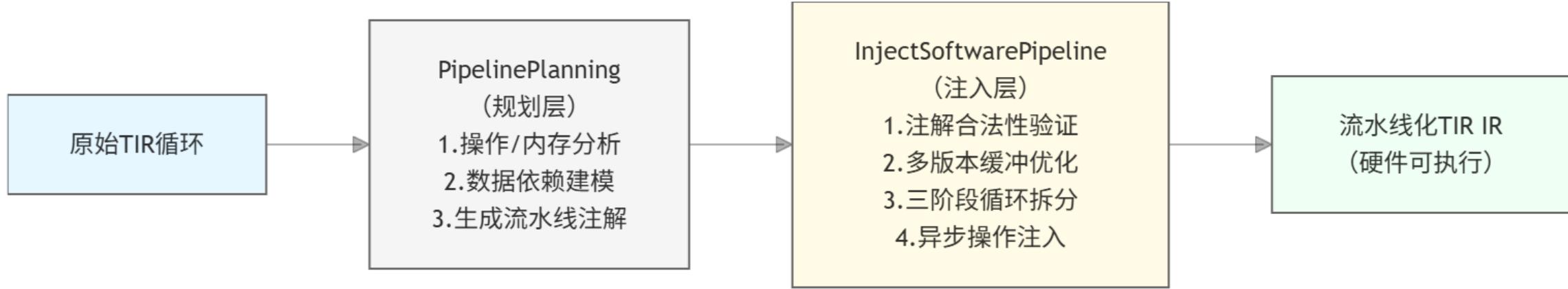
                T.copy(C_L0, C[bx * block_M, by * block_N])

    return main
```

使能2 stage的pipeline并行

CANN

# T.Pipelined 流水并行



**PipelinePlanning (规划层)**：深度解析循环内操作类型（拷贝 / 计算）与数据依赖关系，基于依赖合法性原则为每个操作分配流水线阶段与执行顺序，最终生成标准化注解，为后续提供清晰的调度依据。

**InjectSoftwarePipeline (注入层)**：先验证注解的合法性以规避数据冲突风险；再通过多版本buffer解耦数据，三阶段拆分（Prologue、Body、Epilogue），最终将抽象注解转化为硬件可执行的流水线化 IR。

# T.Persistent

在NPU上实现持久化内核调度Persistent，对齐主社区原语

- **负载均衡保证：**适应任意任务规模，任务数可大于、等于或小于核心数
- **数据局部性改善：**相邻核心处理相邻数据块，提高缓存命中率

```
with T.Kernel(m_num * n_num, is_npu=True) as (cid, _):
    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"):  
        ↑ 使能持久化调度
        for bx, by in T.Persistent([T.ceildiv(M, block_M), T.ceildiv(N, block_N)],  
            core_num, cid):
            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])
```

## T.Parallel

在 TileLang 的编程模型中，T.Parallel是用于表达 tile 内元素向量化计算的核心原语。它在 IR 层以“并行循环”的形式描述数据并行，而不直接暴露底层硬件指令细节，极大的提升用户的算子编程体验。

### 已支持的运算场景

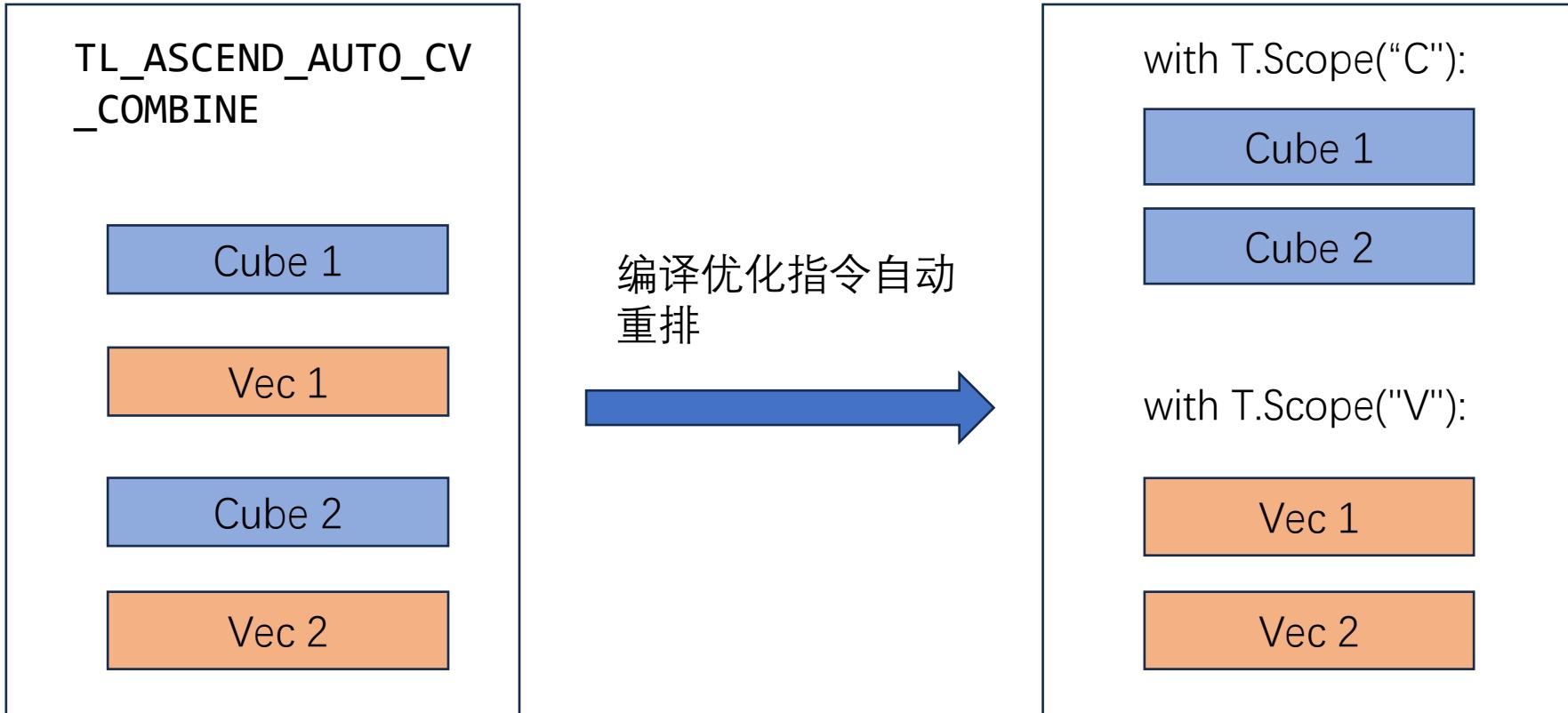
- 支持的双目运算符
- 支持的单目运算符
- 多运算场景
- 1D / 2D 场景
- 双目“向量 + 标量”场景
- 行切分场景
- Buffer + 标量广播运算
- 拷贝场景

```
for i in T.Parallel(v_block):
    m_i[i] = T.max(m_i[i], m_i_prev[i])

for (i, j) in T.Parallel(v_block, BI):
    acc_s_ub[i, j] = acc_s_ub[i, j] + acc_s_ub_[i, j]
```

# 自动拆分CV指令

基于昇腾NPU分离架构（AIC核和AIV核分离）开发融合算子时，算子逻辑中同时涉及AIV核和AIC核的处理逻辑，指令自动重排编译优化允许用户以更自然的方式编写Tilelang代码，无需手动分离CV逻辑

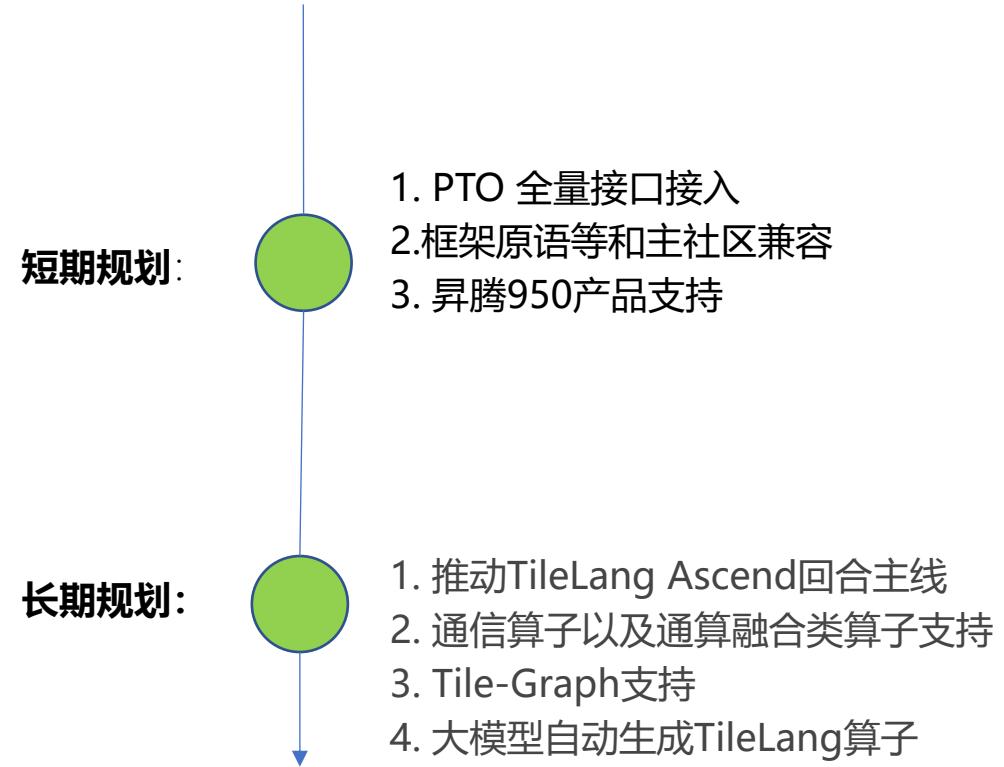


# 内存层级抽象

将底层的硬件内存层级抽象为开发者可直接调用的高层接口 —— 对齐社区公共接口，屏蔽昇腾NPU底层内存层级复杂度

内存分配原语	抽象NPU内存层级	价值场景
T.alloc_shared	T.alloc_l1/ub	在快速的片上存储空间中分配内存，对应NPU上的Unified Buffer统一缓冲区，以及通用内部存储L1缓冲区，可暂存Cube计算单元需要反复使用的一些数据从而减少从总线读写的次数。
T.alloc_fragment	T.alloc_l0A/l0B/l0C	Cube指令的输入、输出

# TileLang-Ascend未来规划



# Thank you.

更多TileLang-Ascend案例尽在：[tile-ai/tilelang-ascend: Ascend TileLang adapter](https://tile-ai/tilelang-ascend: Ascend TileLang adapter)

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