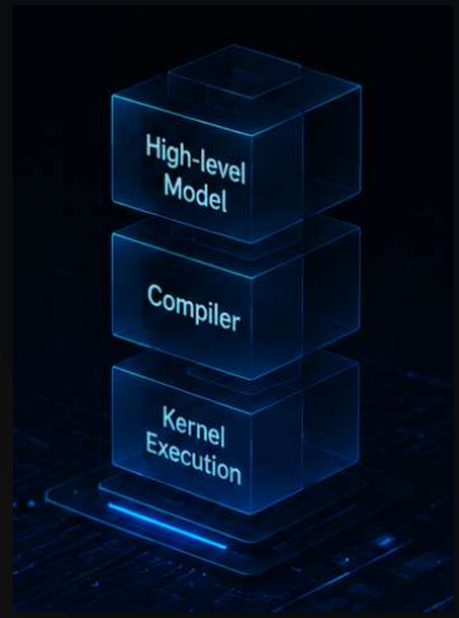
TVM

TileLang

Mooncake



AI编译器与TVM精讲

从自动调优到TileLang大模型落地实战

实验环境：RTX 5090 | CUDA 12.8 | Python 3.12

实验驱动课程

实验环境：RTX 5090 | CUDA 12.8 | Python 3.12

COURSE MAP

课程地图与学习目标

01

宏观破局

为什么需要编译器?

痛点分析 + 融合实验

WHY

02

TVM核心基石

Relax、TensorIR、BYOC

架构底座的建立

WHAT

03

性能黑魔法

自动调优三代演进

搜索空间与Cost Model

HOW

04

破局者TileLang

Tile级抽象与真实性能

155 TFlops

WEAPON

05

Prefill极限压榨

PD分离架构下的终极战场

Mooncake数据支撑

BATTLEFIELD

痛点(Why) → 架构(What) → 调优(How) → 工具(Weapon) → 实战(Battlefield) · 首尾呼应，逻辑闭环

痛点：Eager模式的"死亡循环"

Launch Overhead

小算子的下发时间 > 执行时间，Python→C++→CUDA Driver→GPU 层层链条

Global Memory Round-trip

每个算子都要读写DRAM，4096×4096矩阵一次写回就是33MB

无全局视野

框架看不到"这段可以合并"，无法做跨算子优化

Eager 模式下的"死亡循环"

```
for i in range(100):
x = torch.matmul(x, W1) # Kernel 1: GEMM
x = x + b1 # Kernel 2: BiasAdd
x = torch.relu(x) # Kernel 3: ReLU
x = torch.matmul(x, W2) # Kernel 4: GEMM
x = x + b2 # Kernel 5: BiasAdd

# 5次Kernel launch !
# 4次中间结果写回显存 !
```

Global Memory → Compute → Global Memory → Compute ...（来回折返）

一个简单的前向传播，在 Eager 模式下变成了 **5 个独立 Kernel**，每次算子结束结果必须写回全局显存 —— 对 4096×4096 矩阵，一次写回就是 **33 MB**

编译器的解药：双层抽象

层级	名称	代表优化
上层	图 IR (Graph IR)	算子融合、常量折叠、死代码消除、布局变换
下层	算子 IR (Operator IR)	循环分块、线程绑定、向量化、软件流水线

上层 · Graph IR

负责"**看到全局**"，把能融合的算子合并

宏观策略：算子融合、布局变换、内存规划

↓

下层 · Operator IR

负责"**抠细节**"，决定循环怎么切、线程怎么绑

微观执行：循环分块、线程绑定、软件流水线

核心目标

让数据尽可能留在片上

寄存器 / Shared Memory

减少对 DRAM 的写回次数

两层配合，才能把硬件榨干

Global Memory 写回次数 ↓↓↓ · 编译器核心思想：既然每次写回显存是浪费，那就**别让数据回去**

实验1：算子融合的真实收益

1.17×

融合加速比

4096×4096×4096 FP16

RTX 5090 实测

0.862 ms

Unfused (分离执行)

→

0.737 ms

Fused (TileLang融合)

33.55 MB

中间张量写回 → 0 MB

TileLang 融合 Kernel：matmul + relu 在一个 Kernel 内完成

```
for ko in T.Pipelined(T.ceildiv(K, block_K), num_stages=3):
T.copy(A[...], A_shared)
T.copy(B[...], B_shared)
T.gemm(A_shared, B_shared, C_local) # 片上 GEMM
for i, j in T.Parallel(block_M, block_N):
C_local[i, j] = T.max(C_local[i, j], 0) # 片上 ReLU
T.copy(C_local, C[...]) # 唯一一次写回
```

演进路线：从Eager到Unified Compiler

Eager 模式

PyTorch Eager

TF 1.x

灵活但无法全局优化

Graph Capture

TorchScript

CUDA Graphs

捕获静态子图

消除Python开销

AOT Compiler

XLA, TVM, TensorRT

全局优化、内存规划

动态性弱

Unified Compiler

TVM Relax

Triton / TileLang

动静统一

Pythonic · 多硬件后端

关键跃迁：TVM Unity 时代的 Relax 把"动态Shape"作为一等公民，TileLang 用 Pythonic 语法写底层 Kernel，这才是**大模型时代需要的统一编译器栈**

TVM架构全景：Unity时代的核心设计

v0.25.dev0 Unity

Relax

图级 IR

支持符号化动态 Shape

R.Tensor(("n", 784))

编译一次，运行任意 Batch

TensorIR (s_tir)

算子级 IR

Schedule + DLight + MetaSchedule

循环分块 · 线程绑定

向量化 · 软件流水线

BYOC

外挂厂商高性能库

cuBLAS / cuDNN

编译器排兵布阵

硬仗交给特种兵

编译流程

Frontend (PyTorch/ONNX) → Relax IR → Legalize → DLight / MetaSchedule → TIR → CUDA/LLVM

↘ BYOC → cuBLAS/cuDNN

重大变化：AutoTVM / Ansor 已移除，MetaSchedule + DLight 成为主流 —— TVM 从"模板半自动"走向"全自动+规则互补"

Relax：动态Shape作为一等公民

大模型输入序列长度是**变长**的，静态图 IR 每次变长都要重新编译

Relax 的 **R.Tensor(("n", 784))** 中 "n" 是符号变量

编译一次，运行时传入任意 Batch 大小

Relax 函数定义：动态 Shape + 跨层调用 TIR

```
from tvm.script import relax as R
@R.function
def forward(
data: R.Tensor(("n", 784), dtype="float32"), # n 是动态维度！
w0: R.Tensor((128, 784), dtype="float32"),
b0: R.Tensor((128,), dtype="float32"),
) -> R.Tensor(("n", 128), dtype="float32"):
with R.dataflow():
lv0 = R.matmul(data, R.permute_dims(w0)) + b0
lv1 = R.nn.relu(lv0)
R.output(lv1)
return lv1
```

生死攸关：用户请求的序列长度千差万别，你不能每来一个请求就编译一次。编译一次后，batch=1 或 batch=128 都不需要重新编译

TVMScript：手写硬件级循环的艺术

TVMScript：128×128×128 矩阵乘法 + ReLU

```
from tvm.script import ir as I
from tvm.script import tir as T
@I.ir_module
class MyModule:
@T.prim_func
def mm_relu(A, B, C):
Y = T.alloc_buffer((128, 128))
for i, j, k in T.grid(128, 128, 128):
with T.sblock("Y"):
vi, vj, vk = T.axis.remap("SSR", [i, j, k])
with T.init():
Y[vi, vj] = T.float32(0)
Y[vi, vj] = Y[vi, vj] + A[vi, vk] * B[vk, vj]
```

核心语法糖

@T.prim_func

提取 Python AST，解析成 TensorIR

T.grid(128, 128, 128)

三层循环压缩成一行

T.axis.remap("SSR")

Spatial-Spatial-Reduce 轴映射

T.sblock("Y")

调度的基本单位，MetaSchedule 对这些 block 做变换

Spatial (i) → 输出位置 · **S**patial (j) → 输出位置 · **R**educe (k) → 归约轴

跨层协同：Relax调用TIR

上层变，下层不用改

下层优化，上层无感知

分层解耦

Relax 负责全局策略

TIR 负责极致性能

上层不知道下层的存在

RelaxModuleWithTIR：上层 Relax 图函数 + 底层手写 TIR Kernel

```
@T.prim_func
def relu(x, y): # 底层手写 TIR
n, m = T.int64(), T.int64()
X = T.match_buffer(x, (n, m), "float32")
for i, j in T.grid(n, m):
Y[i, j] = T.max(X[i, j], T.float32(0))
@R.function
def forward(data, w0, b0): # 上层 Relax 图
lv0 = R.matmul(data, R.permute_dims(w0)) + b0
lv1 = R.call_tir(cls.relu, lv0) # ← 跨层调用！
```

BYOC：工业落地的关键拼图

Bring Your Own Codegen

工程最优解：编译器负责"全局排兵布阵"，遇到硬仗交给"特种兵"（cuBLAS/cuDNN）

不追求100%自研，追求全局最优

TVM编译实录与代码结构

构建命令

```
$ cmake .. -DUSE_CUDA=ON \
-DUSE_LLVM=ON \
-G Ninja
$ ninja -j$(nproc)
```

→ libtvm_compiler.so

68 MB

libtvm_compiler.so 体量

大型编译器基础设施的体量

需要 LLVM 15+ 才能编译通过

关键目录结构

目录	功能
python/tvm/relax/	Relax IR 与前端 — 图级动态Shape表示
python/tvm/s_tir/	TensorIR / MetaSchedule / DLight — 统一后的TIR栈
python/tvm/script/	TVMScript 解析器 — Python AST → TensorIR

三代调优演进：AutoTVM→Ansor→MetaSchedule

代际	名称	核心机制	模板依赖	TensorCore	现状	搜索方式
第一代	AutoTVM	专家写模板，搜索参数	必须	弱	已移除	手动模板
第二代	Ansor	自动生成草图+填充	无	较弱	已移除	自动草图
第三代	MetaSchedule	基于TIR的进化搜索	无	强	主流	进化算法
规则式	DLight	专家规则直接生成	无	强	互补	零搜索

→ 越来越自动，越来越统一 →

MetaSchedule 直接基于 TIR 搜索，不需要模板层，和 **DLight** 形成互补：DLight 零搜索做基线，MetaSchedule 精细调热点

最佳实践：开发用 DLight（0秒），部署用 MetaSchedule（5-10分钟）

MetaSchedule架构拆解

设计亮点

CostModel 用 XGBoost

样本量小（几百次测量），XGB 更稳定且可解释

Database 是 JSON 格式

调优结果持久化，跨模型复用

主动学习闭环

预测 → 筛选 Top-K → 测量 → 更新 → 下一轮

MetaSchedule调优流程（代码演示）

Transform-Based API：串进编译管道的调优 Pass

```
from tvm import relax
from tvm.s_tir.meta_schedule.relax_integration import extract_tasks
# 1. 提取任务
tasks = extract_tasks(legalized_mod, target)
for task in tasks:
print(f"{task.task_name} (weight={task.weight})")
# 2. 调优管道：Legalize → Tune → Apply
with target, tempfile.TemporaryDirectory() as tmp_dir:
tuned_mod = tvm.ir.transform.Sequential([
relax.get_pipeline("zero"),
relax.transform.MetaScheduleTuneTIR(
work_dir=tmp_dir,
max_trials_global=300, # 预算：300次真实测量
),
relax.transform.MetaScheduleApplyDatabase(work_dir=tmp_dir),
])(mod)
```

300

max_trials_global 搜索预算

5-10 min

A100 上 300 次测量所需时间

DLight：零搜索时间的规则式调度

0 秒

零搜索时间

基于专家规则的确定性调度

直接应用，无需搜索等待

内置规则

- dl.gpu.Matmul()
- dl.gpu.GEMV()
- dl.gpu.Reduction()

GEMM 怎么分块、Reduction 怎么并行 —— 专家写好的启发式

```
from tvm.s_tir import dlight as dl
with target:
mod = relax.get_pipeline("zero")(mod)
mod = dl.ApplyDefaultSchedule(
dl.gpu.Matmul(),
dl.gpu.GEMV(),
dl.gpu.Reduction(),
)(mod)
```

DLight：0 秒，80% 峰值

开发阶段快速迭代

MetaSchedule：10 分钟，95% 峰值

部署阶段精细调优

自动调优的本质：搜索空间与Cost Model

搜索空间

- Loop Tiling — 循环分块策略
- Thread Binding — 线程绑定方式
- Vectorization — 向量化宽度
- Unrolling — 循环展开因子
- Memory Scope — 内存作用域

主动学习闭环

① Cost Model 预测数百万种调度的性能

↓

② 筛选 Top-K 最有希望的候选

↓

③ 真实硬件测量验证

↓

④ 测量结果反馈训练 Cost Model

Cost Model：XGBoost

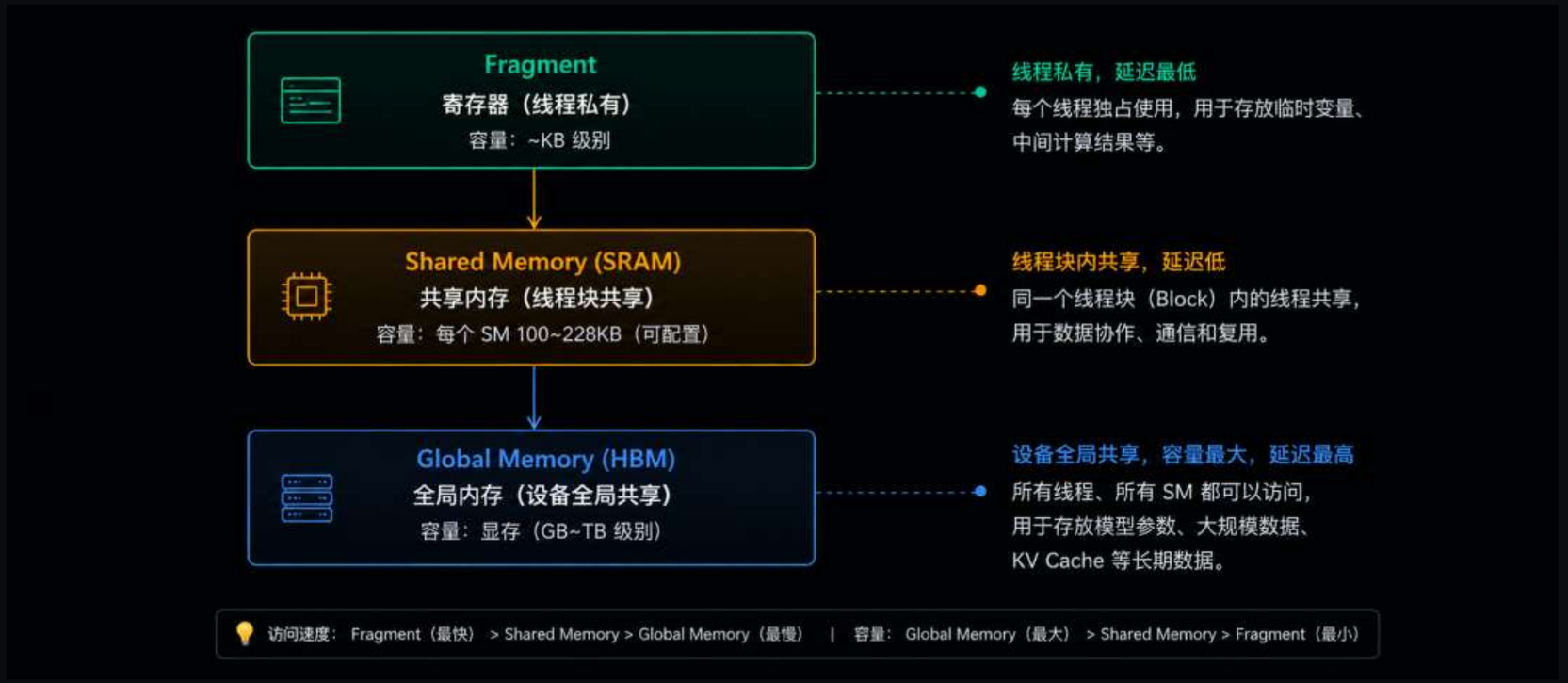
从 TIR 提取特征：循环深度 · 访存模式 · 计算强度 → 预测 latency

为什么不用深度学习？样本量小（几百次测量），**XGBoost 在中小数据集上更稳定，而且可解释**

本质：用机器学习预测代替昂贵的硬件测量，用最少次数逼近最优解

核心洞察：模型和测量互相促进，**主动学习**——用最少次数逼近最优解

核心抽象：Tile级数据流



五大核心 API

T.Tensor(...) → Global Memory (HBM)

T.alloc_shared(...) → Shared Memory (SRAM)

T.alloc_fragment(...) → 寄存器 / Fragment

T.gemm(...) → 自动调用 Tensor Core MMA

T.Pipelined(..., num_stages=N)

→ 自动异步双缓冲流水线

代码Walkthrough：Quickstart深度解读

TileLang Quickstart：80行Python → 异步流水线GEMM+ReLU

```
@tilelang.jit # ① JIT 编译为 CUDA
def matmul(M, N, K, block_M, block_N, block_K):
@T.prim_func
def kernel(A: T.Tensor((M, K), T.float16),
B: T.Tensor((K, N), T.float16)):
with T.Kernel(T.ceildiv(N, block_N), ..., threads=128) as (bx, by): # ② 隐式线程绑定
A_shared = T.alloc_shared((block_M, block_K), T.float16) # ③ Shared Memory
B_shared = T.alloc_shared((block_K, block_N), T.float16)
C_local = T.alloc_fragment((block_M, block_N), T.float32) # ③ 寄存器
T.clear(C_local)
for ko in T.Pipelined(T.ceildiv(K, block_K), num_stages=3): # ④ 异步流水线
T.copy(A[by*block_M, ko*block_K], A_shared) # ⑤ 自动Swizzling
T.copy(B[ko*block_K, bx*block_N], B_shared)
T.gemm(A_shared, B_shared, C_local) # ⑥ Tensor Core
for i, j in T.Parallel(block_M, block_N):
C_local[i, j] = T.max(C_local[i, j], 0) # ⑦ 片上ReLU
T.copy(C_local, C[by*block_M, bx*block_N]) # 唯一写回
return kernel
```

声明式编程：你说做什么，编译器说怎么做。T.Kernel 自动算 Grid/Block，num_stages=3 自动生成 cp.async 和同步屏障，T.gemm 自动选择 mma/wgmma 指令

实验2：GEMM尺寸扫描（188 TFlops）

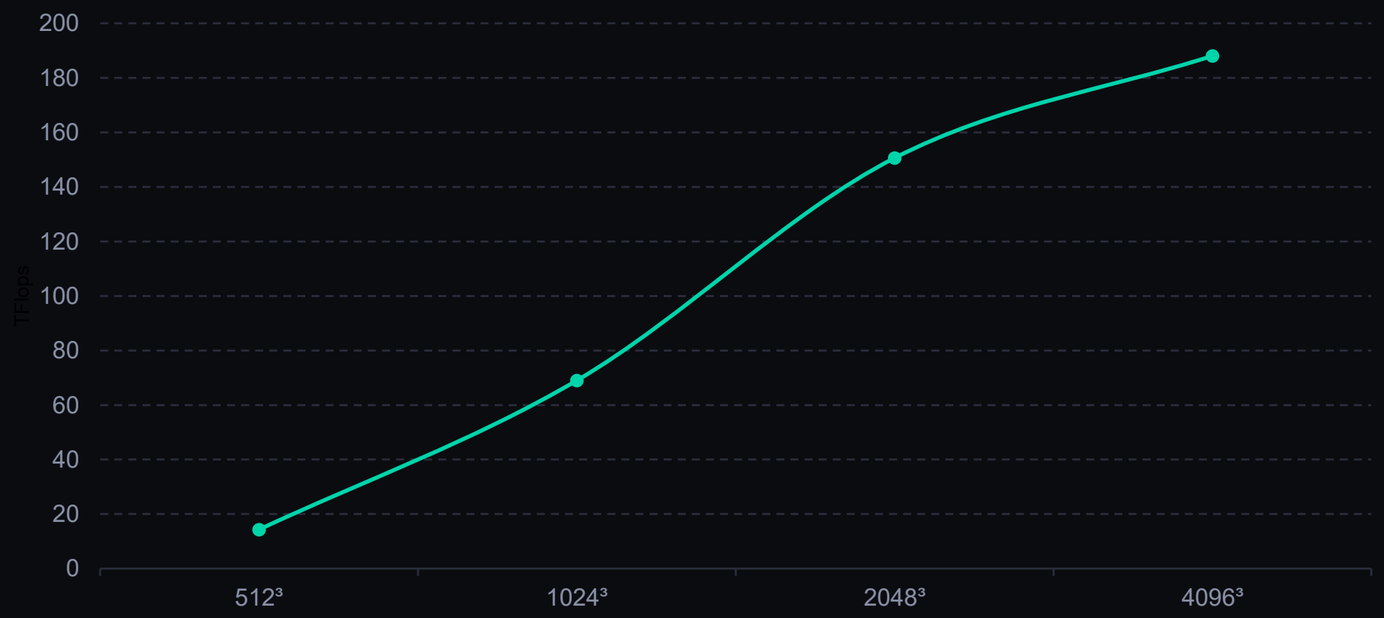
188.06

TFlops

RTX 5090 · FP16 · TileLang GEMM

block_M=128, block_N=128, block_K=32

num_stages=3



尺寸	延迟	TFlops
512^3	0.019 ms	14.32
1024^3	0.031 ms	69.01
2048^3	0.114 ms	150.62
4096^3	0.731 ms	188.06

趋势：矩阵越大 TFlops 越高

512 时 Kernel launch 开销占主导

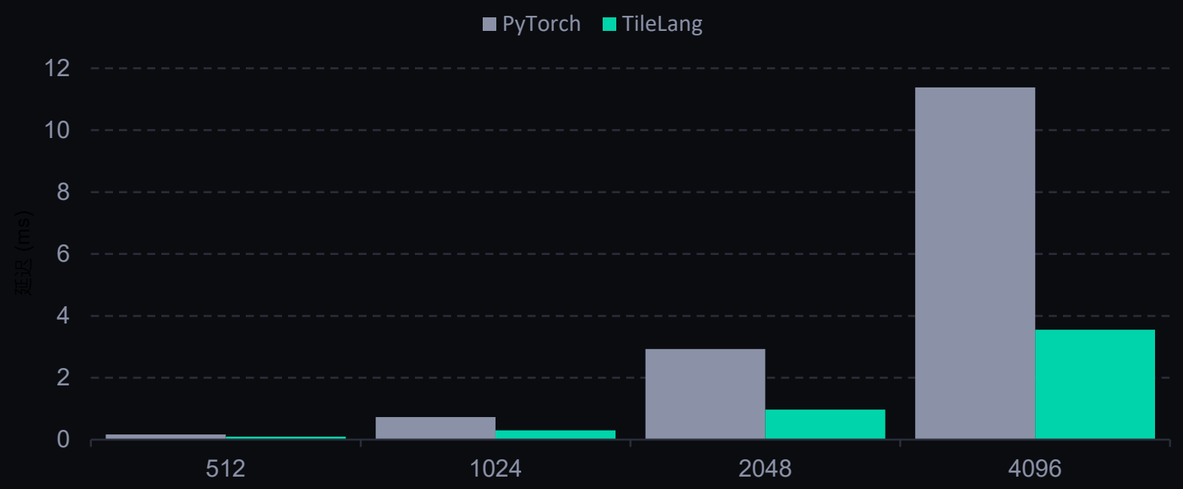
4096 时 Tensor Core 接近满载

注意：这是完全 naive 的 Tiling 参数，没有针对 5090 专门调优。如果上 autotuner，数字还能往上走

实验3：FlashAttention Benchmark（155 TFlops）

batch=4, heads=16, dim=128, causal=False, RTX 5090

序列长度	PyTorch Ref	TileLang	加速比	TileLang TFlops
512	0.17 ms	0.10 ms	1.7×	89.98
1024	0.73 ms	0.30 ms	2.4×	116.06
2048	2.93 ms	0.97 ms	3.0×	141.99
4096	11.38 ms	3.55 ms	3.2×	154.88



核心洞察

趋势1：加速比随序列长度增加

从 1.7× → 3.2×

序列越长，PyTorch 的 $O(S^2)$ 写回越致命

趋势2：TFlops 从 90 → 155

长序列更能喂饱 Tensor Core

关键：$O(S^2)$ 中间结果 → 0 MB 显存写回

实验4：算子融合与MLA前沿

融合实验回顾

4096 矩阵：**1.17× 加速**，**33.55 MB** 显存节省

matmul + relu 在一个 Kernel 内完成

80 行 Python

≈ FlashMLA 性能

TileLang 示例生态

deepseek_mla/

Multi-Layer Attention

2025 最火的高效注意力

~80 行 ≈ FlashMLA

deepseek_nsa/

Native Sparse Attention

原生稀疏注意力实现

前沿研究快速验证

flash_decoding/

Split-KV 解码优化

长序列解码加速

生产级性能

核心洞察：开发速度

TileLang 的价值不仅是性能，更是**开发速度**——快速魔改 Attention Mask 和循环逻辑

DeepSeek 的 MLA 是 2025 年最火的高效注意力机制之一，TileLang 官方示例里 MLA Decode 只用了约 80 行 Python

你不用管 Warp 怎么分，但可以轻松改 Attention 的 mask 逻辑和循环边界 —— 对研究新算法的人是降维打击

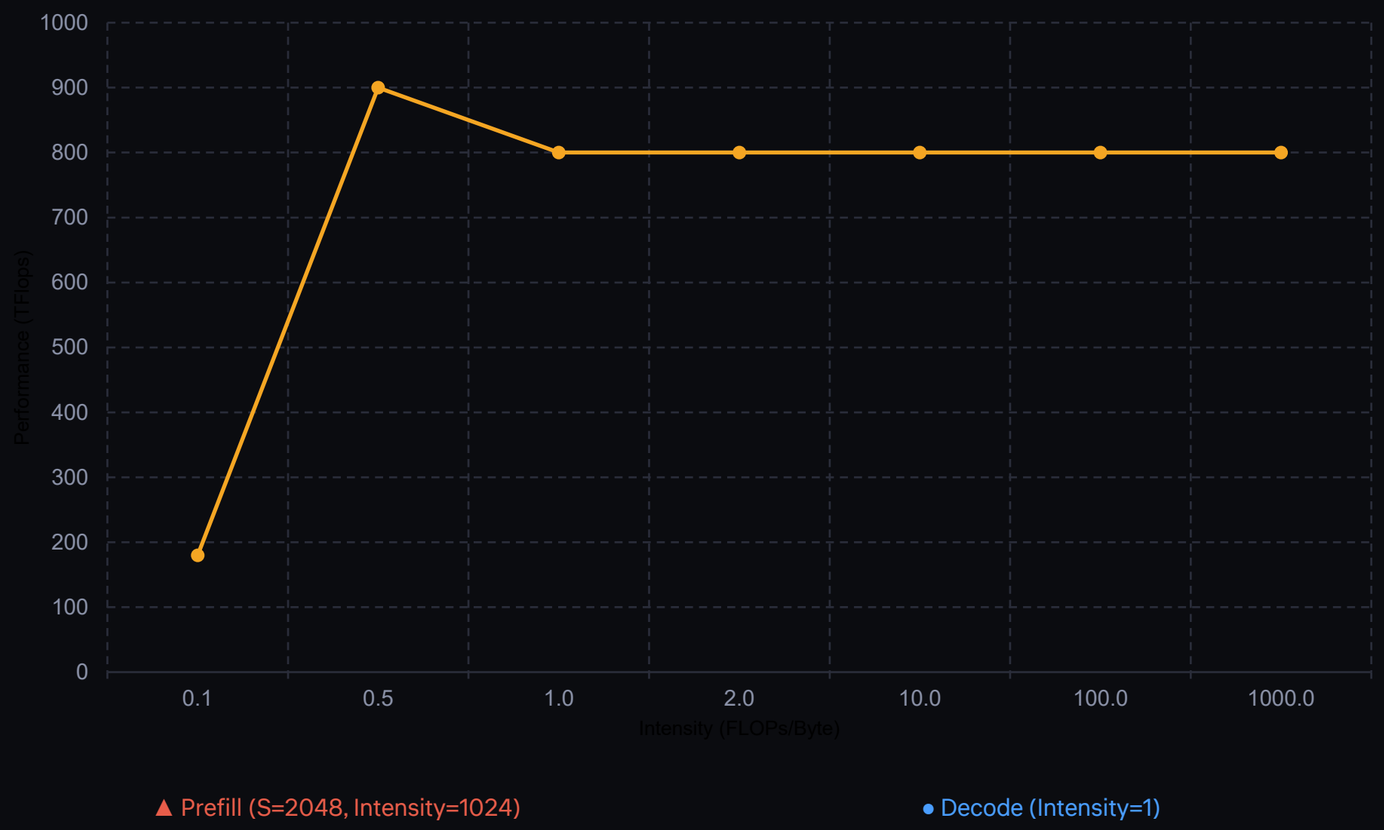
PD分离架构：Mooncake的工程实践

FAST 2025 Best Paper

核心洞察：Prefill 和 Decode 的硬件特性完全相反，放在一起只会互相拖累。干脆物理隔离，中间用 RDMA 传 KV Cache

Roofline模型： Prefill vs Decode大分流

Perf = min(Peak Compute, Intensity × Peak BW) · RTX 5090: Peak Compute ≈ **800 TFlops** · Peak BW ≈ **1800 GB/s**



Prefill (S=2048)

Intensity = **1024**

→ **Compute-bound**

可达 **800 TFlops**

堆算力有用！

Decode (任意S)

Intensity = **1**

→ **Memory-bound**

上限仅 **1.8 TFlops**

堆算力无用，省带宽才有用！

五大性能密码（上）：大Batch + FlashAttention

01 大 Batch / 长 Token —— 喂饱 GPU

FLOPs ∝ B × S^2 × H × D，Bytes ∝ B × S × H × D

长序列 S 是提升计算密度最廉价的方式（**平方级收益**）

Chunked Prefill：将长序列切成多个 chunk，提升调度灵活性

02 FlashAttention —— 规避 N^2 显存读写

标准 Attention：显存化 $O(S^2)$ 的 Attention Score

FlashAttention：Tiling + Online Softmax，**不显存化中间结果**

TileLang 4096 序列：3.2× 加速 · 155 TFlops

五大性能密码（下）： Fusion + TensorCore + MFU

03 Kernel Fusion —— 把整条链路熔成一个 Kernel

理想融合链：Embedding → QKV GEMM → RoPE → FlashAttention → Proj → Residual + Norm → FFN

TileLang 的 T.Pipelined + T.gemm 使描述这种融合成为可能

04 TensorCore 满载 —— 调用硬件矩阵乘法单元

数据类型对齐：FP16/BF16/FP8 · 矩阵尺寸对齐：Warp-level MMA shape（16×8×16）

T.gemm 自动 dispatch 到正确指令

05 高 MFU —— 逼近理论上限

MFU = 实际 TFlops / 峰值 TFlops

TileLang GEMM 4096	TileLang FlashAttn 4096
188 / 800 ≈ 23.5%	155 / 800 ≈ 19.4%

工业界顶尖水平：15%-20%

Attention 的 Softmax 开销天然低于纯 GEMM

TileLang在Prefill中的降维打击

痛点	手写 CUDA	TileLang
Swizzling / Bank Conflict	手动设计	T.copy 自动处理
异步流水线 (cp.async)	手动插入同步点	T.Pipelined 自动生成
Tensor Core 指令选择	手动选 mma/wgmma	T.gemm 自动 dispatch
Warp Specialization	手动分配 Producer/Consumer	编译器自动分析
Attention Mask 修改	改 C++，编译慢	改 Python，JIT 编译

核心认知：TileLang 不是让你"不用懂底层"，而是让你"把精力放在算法和架构上"

Swizzling、流水线、Tensor Core 指令选择 —— 机器比人更靠谱

人的价值：设计新的 Attention 变体、针对特定模型做定制化融合、在 PD 分离架构下做极致的流水线排布

终极场景：Chunked Prefill与自定义Attention

投机解码 和 **Prefix Caching** 带来非均匀输入：

Prefix（有缓存，1024 tokens）：只需 Extend · Draft（新 token，128 个）：需要完整 Prefill

标准 FlashAttention 库无法直接处理这种混合逻辑

Prefix (Cached) — 1024 tokens — 只需 Extend，不需要完整计算

Draft (New) — 128 tokens

TileLang 中自定义混合 Attention Mask

```
for k in T.Pipelined(loop_range, num_stages=2):
if is_causal:
for i, j in T.Parallel(block_M, block_N):
# 支持 prefix + draft 的混合逻辑
acc_s[i, j] = T.if_then_else(
(bx * block_M + i >= prefix_len) and
(bx * block_M + i >= k * block_N + j),
-T.infinity(acc_s.dtype), 0
)
```

从想法 → JIT Kernel ≈ 10 分钟 · 改几行 Python，调整 mask 条件和循环边界，重新 JIT 编译即可

课程总结与进阶路线

逻辑闭环回顾

WHY
Eager 模式显存墙
33.55 MB 融合节省

→

WHAT
TVM Relax + TensorIR
双层抽象

→

HOW
MetaSchedule + DLight
300 trials 搜索预算

→

WEAPON
TileLang Tile 级抽象
155 TFlops

→

BATTLEFIELD
PD 分离
31.65 ms

进阶路线

1 **动手跑 TileLang 示例**
quickstart.py → flash_attention → deepseek_mla

2 **读 TVM 官方文档**
Relax、TIR、MetaSchedule 教程

3 **读 Mooncake 论文**
FAST 2025，理解 PD 分离的调度策略

4 **尝试自定义 TileLang 算子**
挑一个模型里的瓶颈算子，用 TileLang 重写

最好的学习就是动手