

# PTO ISA Compiler

**Programmable Tensor Operations (PTO)** 是一种面向张量计算的领域特定语言（DSL），提供统一的编程模型来描述深度学习算子，并支持编译到多种硬件后端。

## 目录

- 1. PTO ISA 定义
- 2. 编程接口 - Function Builder
- 3. 文件接口 - .pto Assembly
- 4. Runtime Task 数据结构
- 5. Task Dump 与可视化
- 6. 动态 Shape 支持

## 1. PTO ISA 定义

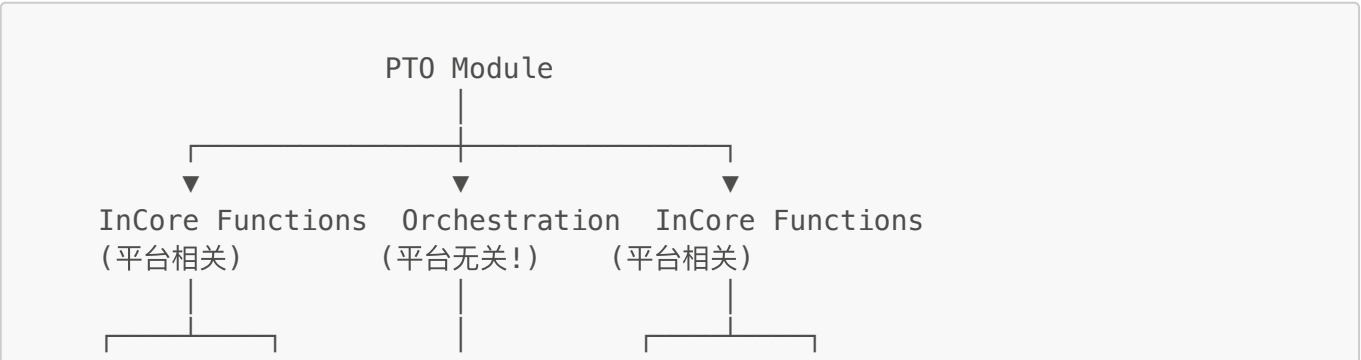
### 1.1 指令集概述

PTO ISA 定义了一套平台无关的张量操作指令集，包括：

类别	指令	描述
数据移动	TLOAD, TSTORE	Tile 加载/存储
算术运算	TADD, TSUB, TMUL, TDIV	逐元素算术
矩阵运算	TMATMUL, TMATMUL_ACC	矩阵乘法
归约运算	TROWSUM, TROWMAX, TCOLSUM	行/列归约
激活函数	TEXP, TLOG, TSQRT, TSILU	非线性变换
广播运算	TROWEXPANDSUB, TROWEXPANDDIV	行广播
标量运算	SADD, SMUL, SLI, SCMP	标量计算
控制流	FOR, IF, CALL, RETURN	程序控制

### 1.2 多后端代码生成

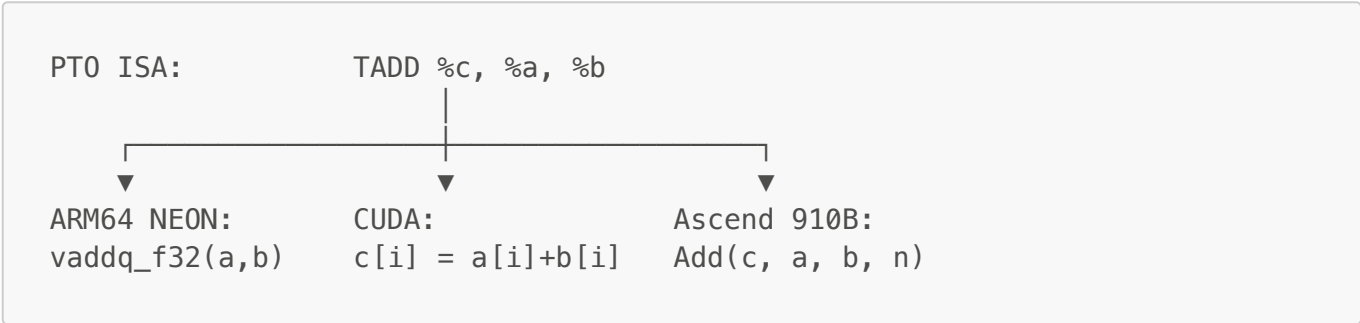
PTO 编译器支持将统一的 PTO ISA 编译到多种物理 ISA：





- 关键设计原则：
- **InCore Functions:** 平台相关，编译到 ARM64/CUDA/Ascend 物理 ISA
  - **Orchestration Functions:** 平台无关，只调用 PT0 Runtime 的 task 接口
  - 同一份 Orchestration 代码用于所有后端

代码生成示例：



1.3 编译优化

- **Loop Fusion:** 合并相邻循环，减少循环开销
- **Buffer Reuse:** 基于 Tile 生命周期分析复用缓冲区
- **Inline Expansion:** InCore 函数调用内联展开

2. 编程接口 - Function Builder

2.1 概述

PT0FunctionBuilder 提供 Fluent API 来构建 PT0 程序：

```
from pto_compile import PT0FunctionBuilder, PT0Module

# 创建模块
module = PT0Module("my_module")

# 构造函数
builder = PT0FunctionBuilder("my_function", module=module)
program = (builder
    .in_core() # 声明为 InCore 函数
    .memref("input", MemorySpace.GLOBAL, ElementType.F32)
    .memref("output", MemorySpace.GLOBAL, ElementType.F32)
    .tile("x", 8, 8, ElementType.F32)
    .load("x", "input") # TLOAD
    .exp("y", "x") # TEXP)
```

```
.store("output", "y") # TSTORE
.build())
```

2.2 函数类型：InCore vs Orchestration

PTO 定义了两种函数类型，具有不同的语义和执行模式：

特性	InCore Function	Orchestration Function
声明方式	<code>.in_core()</code>	<code>.not_in_core()</code>
执行位置	计算核心 (GPU/NPU/CPU SIMD)	主机 CPU
物理 ISA	CUDA / Ascend C / ARM64 NEON	ARM64 C
主要内容	Tile 级张量操作	控制流 + CALL InCore 函数
嵌套调用	被内联展开	保留 CALL，生成任务调度代码
内存模型	核内 Tile Buffer	全局内存指针

InCore Function 示例

```
def create_rowmax():
    """InCore: 计算行最大值，在计算核心执行"""
    return (PTOFunctionBuilder("rowmax", module=module)
            .in_core() # ← InCore 函数
            .memref("input", MemorySpace.GLOBAL, ElementType.F32)
            .memref("output", MemorySpace.GLOBAL, ElementType.F32)
            .tile("x", 8, 8, ElementType.F32)
            .tile("result", 8, 1, ElementType.F32) # 归约输出
            .load("x", "input")
            .rowmax("result", "x") # TROWMAX 指令
            .store("output", "result")
            .build())
```

生成的 CUDA 代码：

```
__global__ void rowmax(float* input, float* output) {
    float x[8][8], result[8][1];
    // TLOAD
    for (int _row = 0; _row < 8; _row++)
        for (int _col = 0; _col < 8; _col++)
            x[_row][_col] = input[_row * 8 + _col];
    // TROWMAX
    for (int _row = 0; _row < 8; _row++) {
        float _max = x[_row][0];
        for (int _col = 1; _col < 8; _col++)
            if (x[_row][_col] > _max) _max = x[_row][_col];
        result[_row][0] = _max;
    }
}
```

```

    }
    // TSTORE
    ...
}

```

## Orchestration Function 示例

```

def create_dynamic_softmax():
    """Orchestration: 动态分块, 在主机 CPU 执行"""
    return (PTOFunctionBuilder("dynamic_softmax", module=module)
            .not_in_core() # ← Orchestration 函数
            .memref("input", MemorySpace.GLOBAL, ElementType.F32)
            .memref("output", MemorySpace.GLOBAL, ElementType.F32)
            .scalar("num_tiles", ElementType.I32) # 动态参数

            # 动态循环 - 根据输入大小迭代
            .for_loop("tile_idx", 0, "num_tiles", 1)
                # CALL InCore 函数 - 使用偏移参数区分不同 tile
                # 格式: ("tensor_name", "loop_var", col_offset)
                .call("rowmax", {
                    "input": ("input", "tile_idx", 0), # input[tile_idx]
                    "output": ("temp_max", "tile_idx", 0)
                })
                .call("rowexpandsub", {
                    "input_x": ("input", "tile_idx", 0),
                    "input_row": ("temp_max", "tile_idx", 0),
                    "output": ("temp_shifted", "tile_idx", 0)
                })
                # ... 其他 CALL
            .end_for()
            .build())

```

## 生成的 ARM64 C 代码（主机端）：

```

void dynamic_softmax(PTORuntime* rt, float* input, float* output,
                    int num_tiles) {
    for (int tile_idx = 0; tile_idx < num_tiles; tile_idx++) {
        // CALL 指令转换为任务调度代码
        int32_t t0 = pto_task_alloc(rt, "rowmax", NULL, 288, 288);
        pto_task_add_input(rt, t0, input, tile_idx, 0, 8, 8);
        pto_task_add_output(rt, t0, temp_max, tile_idx, 0, 8, 1);
        pto_task_submit(rt, t0);

        int32_t t1 = pto_task_alloc(rt, "rowexpandsub", ...);
        // ... 依赖自动建立
    }
}

```

## 2.3 CALL 指令与偏移参数

在 Orchestration 函数中，`.call()` 支持两种参数格式：

```
# 格式 1: 简单参数 (无偏移)
.call("func_name", {"param": "tensor_name"})

# 格式 2: 带偏移参数 (用于动态 tiling)
.call("func_name", {"param": ("tensor_name", "row_offset", col_offset)})
```

偏移参数说明：

- `tensor_name`: 全局内存中的张量名
- `row_offset`: 行偏移表达式，可以是：
  - 循环变量名 (如 `"tile_i"`, `"q_tile"`)
  - 标量变量名
  - 整数常量 (如 `0`)
- `col_offset`: 列偏移，通常为 `0`

示例：LLaMA Flash Attention 的三阶段依赖

```
# Phase 1: Pre-Attention (所有 tile 并行)
.for_loop("tile_i", 0, "num_tiles", 1)
    .call("rmsnorm_tile", {
        "input": ("input", "tile_i", 0),      # input[tile_i]
        "output": ("temp_norm", "tile_i", 0)
    })
    .call("tile_matmul", {
        "input_a": ("temp_norm", "tile_i", 0),
        "input_b": "wq",                      # 共享权重，无偏移
        "output": ("all_q_tiles", "tile_i", 0)
    })
.end_for()

# Phase 2: Flash Attention (交叉 tile 依赖)
.for_loop("q_tile", 0, "num_tiles", 1)
    .for_loop("kv_tile", 0, "num_tiles", 1)
        .call("flash_attn_score_block", {
            "input_q": ("all_q_rope", "q_tile", 0),  # Q[q_tile]
            "input_k": ("all_k_rope", "kv_tile", 0),  # K[kv_tile] ← 交叉依
            "output_s": ("temp_scores", "q_tile", 0)
        })
    .end_for()
.end_for()
```

偏移的作用：

- 不同 tile 使用不同偏移，使得任务之间没有虚假依赖

- Runtime 可以正确识别并行机会
- Phase 1 和 Phase 3: 所有 tile 完全并行
- Phase 2: Q[i] 依赖所有 K[j], V[j], 形成 N×N 交叉依赖

2.4 编程语法对比

操作	InCore Function	Orchestration Function
Tile 操作	<code>.add()</code> , <code>.mul()</code> , <code>.exp()</code>	❌ 不支持
标量运算	<code>.scalar_add()</code>	<code>.scalar_add()</code>
控制流	<code>.for_loop()</code> , <code>.if_then()</code>	<code>.for_loop()</code> , <code>.if_then()</code>
函数调用	<code>.call()</code> → 内联展开	<code>.call()</code> → 生成任务调度代码
偏移参数	❌ 不需要	✅ 支持 ("tensor", "offset", 0)
内存访问	<code>.load()</code> , <code>.store()</code>	❌ 不支持
二进制展开	❌ 不支持	✅ <code>max_range</code> , <code>min_range</code> 参数

2.5 动态循环的二进制展开 (Binary Expansion)

对于动态上限的循环，可以指定 `max_range` 和 `min_range` 参数启用二进制展开优化：

```
# 动态循环，上限 n 最大为 4096，最小量化块为 256
.for_loop("i", 0, "n", 1, max_range=4096, min_range=256)
    .call("process_tile", {...})
.end_for()
```

参数说明：

参数	说明
<code>max_range</code>	动态上限的最大可能值（必须是 2 的幂次）
<code>min_range</code>	最小量化块大小（小于此值的剩余迭代由单独的残差循环处理）

**原理：** 将动态循环上限转换为二进制表示，每个二进制位控制一个固定大小的循环块。`min_range` 设置最小块大小，低于此阈值的迭代由一个残差循环统一处理。

**示例：** `max_range=4096`, `min_range=256`, 实际 `n=300`

- 量化部分： `300 & ~255 = 256` → 一个 256-块
- 残差部分： `300 & 255 = 44` → 残差循环 44 次

生成的代码：

```
// Binary-expanded loop: n quantized into [4096,2048,1024,512,256] + residual
```

```
{
    int _i_base = 0;
    int _i_limit = n;
    int _i_residual = _i_limit & 255;      // 残差迭代 < 256
    int _i_quantized = _i_limit - _i_residual; // 量化到 256 边界

    // 2的幂次块: [4096, 2048, 1024, 512, 256]
    if (_i_quantized & 4096) {
        for (int i = _i_base; i < _i_base + 4096; i += 1) { BODY }
        _i_base += 4096;
    }
    if (_i_quantized & 2048) { /* ... */ }
    if (_i_quantized & 1024) { /* ... */ }
    if (_i_quantized & 512)  { /* ... */ }
    if (_i_quantized & 256)  { /* ... */ }

    // 残差循环: 处理 < min_range 的剩余迭代
    if (_i_residual > 0) {
        for (int i = _i_base; i < _i_base + _i_residual; i += 1) { BODY }
    }
}
```

优势:

特性	说明
固定大小块	每个 2^k 块迭代次数固定，便于硬件流水线优化
分支预测	每个 if 只检查一位，预测准确率高
循环展开	固定大小循环更容易展开优化
减少代码膨胀	min_range 限制生成的块数量，避免过多 if 分支
残差处理	小于 min_range 的迭代由单循环处理，简化控制流

2.6 自适应 Tile 大小 (Adaptive Tile Sizing)

二进制展开可以结合自适应 Tile 大小，通过 tile\_levels 参数为不同块大小选择不同的 Tile 尺寸:

```
# 定义每个块大小对应的 Tile 行数
# Key 0 是特殊标记, 表示 residual iterations (< min_range) 的 tile 大小
TILE_ROWS_BY_LEVEL = {
    4096: 64,    # 大块: 64 行 Tile
    2048: 64,    # 大块: 64 行 Tile
    1024: 64,    # 中块: 64 行 Tile
    512:  64,    # 小块: 64 行 Tile
    256:  64,    # 最小量化块: 64 行 Tile
    0:     32,    # Residual (< min_range): 32 行 Tile (更细粒度)
}

.for_loop("tile_i", 0, "num_tiles", 1,
```

```
        max_range=4096, min_range=256,
        tile_levels=TILE_ROWS_BY_LEVEL) # 自适应 Tile 大小
    .call("rmsnorm_tile", {...})
.end_for()
```

工作原理：

1. 迭代次数缩减: 当使用更大的 Tile（如 64 行 vs 32 行），每次迭代处理更多数据，所以迭代次数减少。
- `scale = tile_rows / base_tile_rows`

◦ `actual_iters = block_size / scale`

◦ 例：4096 块使用 64 行 Tile → `4096 / 2 = 2048` 次迭代（减少 50%！）
2. 函数变体选择: 编译器自动将函数名替换为对应的变体（如 `rmsnorm_tile_64`）
3. 偏移缩放: 生成 `_tile_i_row_offset = _tile_i_base_rows + (tile_i * scale)` 确保正确的内存访问

递归嵌套循环支持：

二进制展开会递归应用到循环体内的嵌套循环，实现  $N^2 \rightarrow (N/2)^2$  的优化！

```
# Flash Attention 嵌套循环
.for_loop("q_tile", 0, "num_tiles", 1,
        max_range=4096, min_range=256, tile_levels=TILE_ROWS_BY_LEVEL)
    .for_loop("kv_tile", 0, "num_tiles", 1, # 内层循环也会被展开！
        max_range=4096, min_range=256,
        tile_levels=TILE_ROWS_BY_LEVEL)
        .call("flash_attn_score_block", {...})
    .end_for()
.end_for()
```

LLaMA 7B 优化效果（256 tiles = 8K 序列长度）：

配置	外层迭代	内层迭代	Flash Attention 任务	总任务	缩减
无优化	256	256	$256^2 \times 3 = 196,608$	~200K	-
有优化	128	128	$128^2 \times 3 = 49,152$	51,200	75%

PTO Assembly 格式：

```
FOR %i:idx, 0:idx, %n:idx, 1:idx max_range=4096 min_range=256 tile_levels=
{4096:64,2048:64,1024:64,512:64,256:64}
```

LLaMA 示例：序列长度上限 128K (4096 tiles)，最小块 256 tiles (8K 序列)



```
MAX_NUM_TILES = 4096    # 128K / 32 = 4096 tiles
MIN_NUM_TILES = 256     # 最小量化块

.for_loop("tile_i", 0, "num_tiles", 1,
          max_range=MAX_NUM_TILES, min_range=MIN_NUM_TILES,
          tile_levels=TILE_ROWS_BY_LEVEL) # 自适应 Tile 大小
```

## 3. 文件接口 - .pto Assembly

### 3.1 格式说明

.pto 文件是 PTO 程序的文本表示形式，与 Function Builder 构建的程序等价：

```
// 模块声明
// PTO Module: softmax_module
// Entry: @dynamic_softmax

// InCore 函数定义
// Function Type: InCore
func @rowmax(%input: !pto.memref<gm,...,f32>, %output:
!pto.memref<gm,...,f32>) {
    // Tile 声明
    %x = alloc_tile : !pto.tile<8x8xf32>
    %result = alloc_tile : !pto.tile<8x1xf32>

    // 指令
    %x = tload %input[0, 0] : (!pto.memref) -> !pto.tile<8x8xf32>
    %result = trowmax %x : !pto.tile<8x8xf32> -> !pto.tile<8x1xf32>
    tstore %result, %output[0, 0]

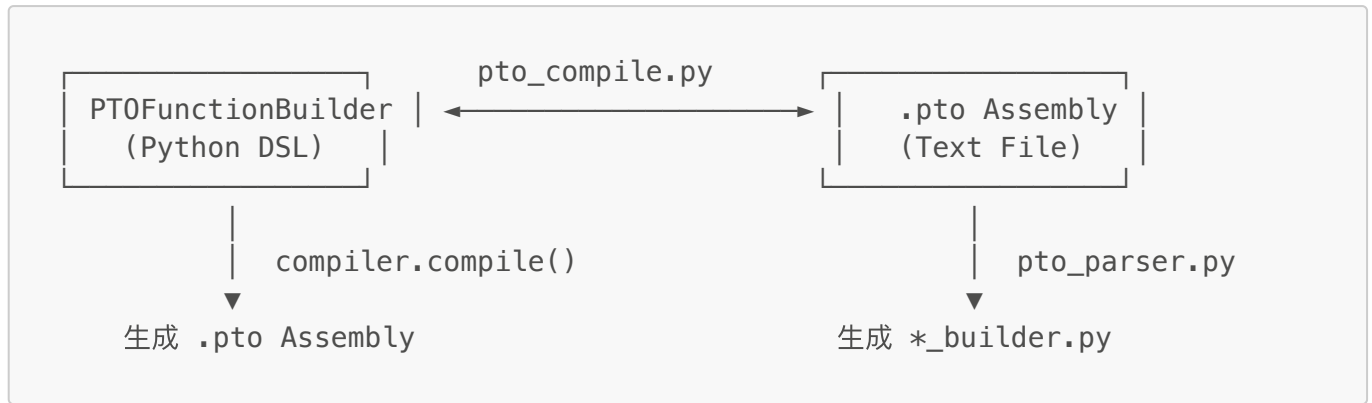
    return
}

// Orchestration 函数定义
// Function Type: Orchestration
func @dynamic_softmax(%input: !pto.memref, %output: !pto.memref) {
    // 标量声明
    %num_tiles = alloc_scalar : i32

    // 动态循环
    FOR %tile_idx:idx, 0:idx, %num_tiles:idx, 1:idx
        CALL @rowmax(%input -> %input, %output -> %temp_max)
        CALL @rowexpandsub(...)
    ENDFOR

    return
}
```

### 3.2 双向转换



双向转换已实现：

- ☒ **Function Builder → .pto Assembly:** `PTOCompiler.compile()`
- ☒ **.pto Assembly → Function Builder:** `pto_parser.py` 或 `pto_compile.py parse`

### 3.3 从 .pto 生成 Python 代码

使用命令行工具可以从 `.pto` 文件生成等价的 `PTOfuctionBuilder` Python 代码：

```
# 方式 1: 使用 pto_compile.py
python pto_compile.py parse examples/output_pto/llama7b/

# 方式 2: 使用 pto_parser.py
python pto_parser.py examples/output_pto/llama7b/

# 指定输出目录
python pto_compile.py parse examples/output_pto/llama7b/ -o ./generated/
```

输出示例：

```
Found 1 .pto file(s) in examples/output_pto/llama7b/

Processing: llama7b_layer.pto
Module: llama7b_flash
Functions: 26
  InCore: 25
  Orchestration: 1
Generated: examples/output_pto/llama7b/llama7b_layer_builder.py
```

生成的 **Python** 文件可以直接运行，重建相同的 **PTOModule**：

```
# 运行生成的代码，验证正确性
python examples/output_pto/llama7b/llama7b_layer_builder.py
```

用途：

- 将现有 .pto 程序转换为 Python 代码，便于修改和扩展
- 实现程序变换和优化工具
- 编辑器支持和代码生成

## 4. Runtime Task 数据结构

### 4.1 概述

当执行 Orchestration Function 时，编译器生成的代码会构建 **Task Graph** 数据结构：

```
// 任务定义
typedef struct {
    char          func_name[64];           // InCore 函数名
    void*         func_ptr;                // 函数指针
    int32_t       fanin;                   // 依赖计数
    int32_t       fanout_count;            // 下游任务数
    int32_t       fanout[512];             // 下游任务 ID
    TaskArg       args[16];                // 输入/输出参数
    int32_t       buffer_size;             // InCore 缓冲区大小
} PendingTask;

// 运行时状态
typedef struct {
    PendingTask   pend_task[65536];        // 待执行任务表
    TensorMap     tensor_map;               // 张量 → 生产者任务映射
    int32_t       ready_queue[1024];        // 就绪队列
    int64_t       total_tasks;              // 总任务数
} PTORuntime;
```

### 4.2 Task Graph 构建流程



### 4.3 动态参数与 Task Graph 重建

**关键特性：** 每次使用不同的动态参数调用 Orchestration Function，都会重新执行并生成新的 Task Graph：

```
// 示例：不同 sequence length 生成不同 Task Graph
void run_llama_layer(int seq_len) {
    PTORuntime* rt = malloc(sizeof(PTORuntime));
    pto_runtime_init(rt);

    int num_tiles = seq_len / TILE_ROWS;

    // Orchestration function 是纯 C 代码，直接在主机 CPU 执行
    // 每次调用都重新构建 Task Graph
    llama_layer_dynamic(rt, input, output, num_tiles);

    // 此时 rt 包含完整的 Task Graph
    // 运行时可以：
    //   1. 并行调度 ready_queue 中的任务
    //   2. 完成任务后更新 fanin，解除依赖
    //   3. 新就绪的任务加入 ready_queue

    pto_runtime_shutdown(rt);
    free(rt);
}
```

### 4.4 LLaMA 7B 性能统计

以下是 LLaMA 7B Layer（含 Flash Attention + 自适应 Tile 优化）的 Task Graph 构建性能：

SeqLen	Tiles	实际迭代	Tasks	无优化	Build Time	Tasks/ms	Memory	节省
1K	32	16	1,024	3,584	0.144 ms	7,111	2.81 MB	71%
2K	64	32	3,584	13,312	0.444 ms	8,072	9.81 MB	73%
4K	128	64	13,312	51,200	1.696 ms	7,849	36.4 MB	74%
8K	256	128	51,200	200,704	8.503 ms	6,021	140 MB	74.5%
16K	512	256	200,704	794,624	34.2 ms	5,868	548 MB	75%

**自适应 Tile 优化 (64-row tiles)：**

- 无优化公式:  $Tasks = 16N + 3N^2$  ( $N = seq\_len / 32$ )
- 有优化公式:  $Tasks = 7N + 3N^2/4$  (迭代次数减半)
- 节省:  $N^2 \rightarrow (N/2)^2 = N^2/4 \rightarrow 75\%$  任务数缩减

**128K 序列长度推断：**

指标	16K	128K (推断)	比例
----	-----	-----------	----

指标	16K	128K (推断)	比例
Tiles	512	4,096	8x
Tasks	200,704	12,611,584	63x
Build Time	34 ms	~2.1 秒	61x
Memory	548 MB	~33.6 GB	63x

SRAM 限制 Tile 大小:

- 64-row Flash Attn:  $Q(32KB) + K(32KB) + V(32KB) + S(16KB) + O(32KB) = 144\text{ KB} < 256\text{ KB} \checkmark$
- 128-row:  $S = 128 \times 128 = 64KB \rightarrow \text{Total} = 321\text{ KB} > 256\text{ KB} \times$
- Score 矩阵  $S = \text{tile\_rows}^2$  是瓶颈!

数据结构大小:

- `sizeof(PendingTask)` = 2,864 bytes
- `sizeof(TensorMapEntry)` = 56 bytes
- `PTO_MAX_TASKS` = 262,144 (可容纳 `seq_len`  $\leq 16K$ )

## 5. Task Dump 与可视化

### 5.1 Dump 函数

```
// 输出到文件
pto_runtime_dump(rt, "task_graph.txt");

// 输出到 stdout
pto_runtime_dump_stdout(rt);
```

输出格式:

```
=====
=====
PTO RUNTIME DUMP
=====
=====

SUMMARY
  Total tasks scheduled:  40
  Ready queue size:      16

TASK TABLE
  Task 0: rowmax          [READY] fanin=0 buf=288.0KB fanout=[]
  Task 1: rowexpandsub    [READY] fanin=0 buf=544.0KB fanout=[2]
  Task 2: elem_exp        [WAIT]  fanin=1 buf=512.0KB fanout=[3,4]
  Task 3: rowsum          [WAIT]  fanin=1 buf=288.0KB fanout=[]
```

```
Task 4: rowexpanddiv [WAIT] fanin=1 buf=544.0KB fanout=[]
...

DEPENDENCY GRAPH
Task 1 -> Task 2
Task 2 -> Task 3
Task 2 -> Task 4
...
```

5.2 可视化工具

使用 `visualize_taskgraph.py` 将 Task Dump 转换为 PDF:

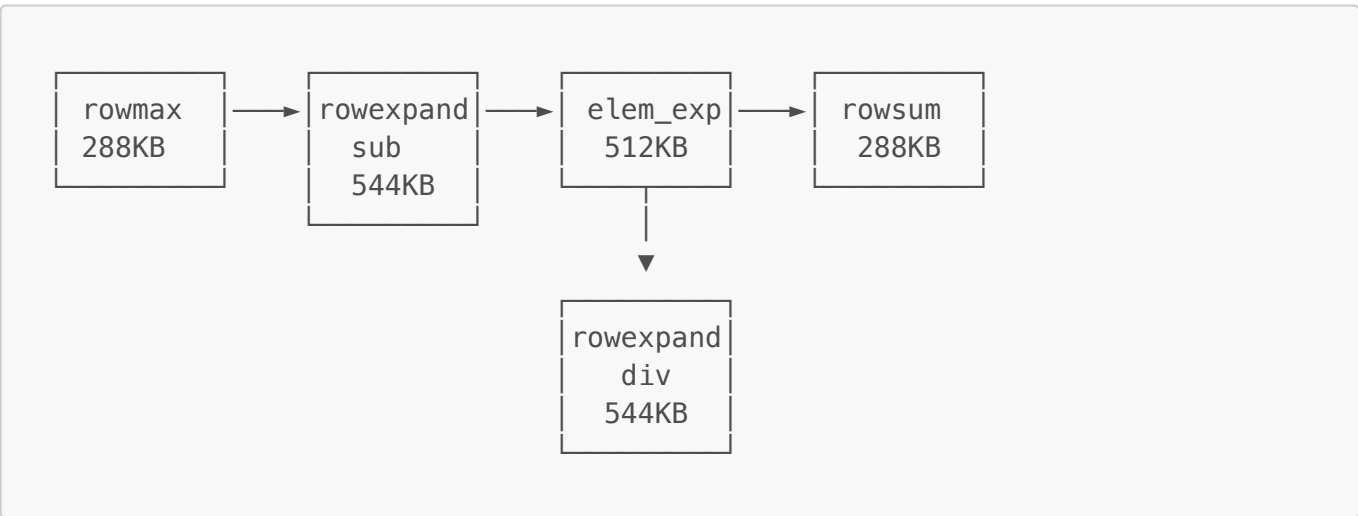
```
# 使用方式
python visualize_taskgraph.py <dump_file> <output_pdf>

# 示例
python visualize_taskgraph.py \
  examples/output_arm64/llama7b/llama_layer_dynamic_task_graph.txt \
  examples/output_arm64/llama7b/llama_layer_dynamic_task_graph.pdf
```

可视化特性:

- 左到右布局 (rankdir=LR)
- 同一执行层级的任务在同一列 (rank=same)
- 显示函数名和缓冲区大小
- 箭头表示依赖关系

示例输出:



6. 动态 Shape 支持

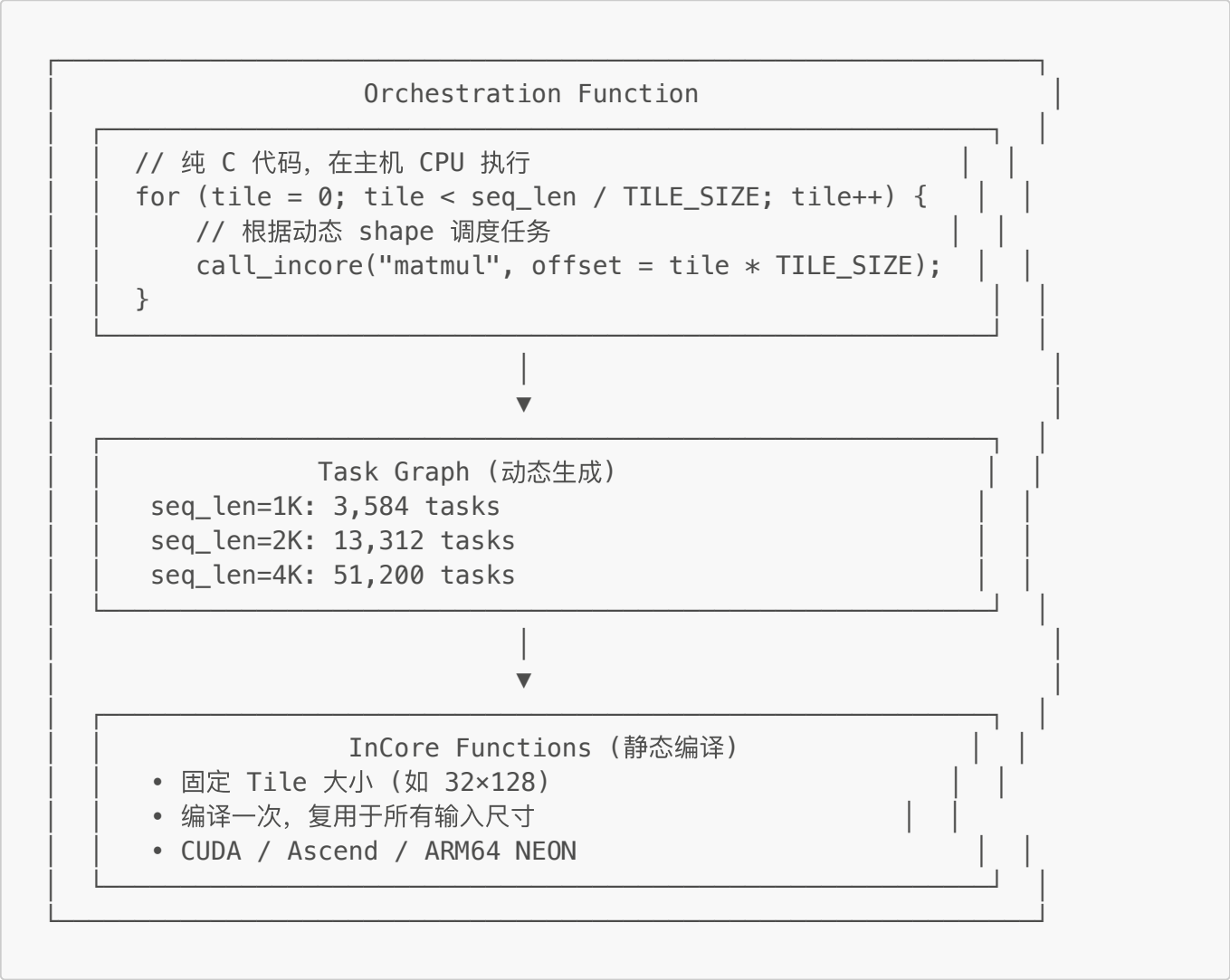
6.1 问题背景

深度学习推理中，输入尺寸（如 sequence length）通常是动态的。传统静态编译方法需要为每种输入尺寸编译不同的 kernel，导致：

- 编译时间长
- 二进制体积大
- 无法处理未见过的尺寸

6.2 PTO 解决方案

PTO 通过 **Orchestration Function** 完全解决动态 shape 问题：



6.3 性能分析

**关键问题：** Orchestration Function 是否会成为性能瓶颈？

根据 LLaMA 7B 的测试数据：

Sequence Length	Task Graph 构建时间	推理时间 (估计)	占比
1K	0.8 ms	~50 ms	1.6%
2K	2.6 ms	~100 ms	2.6%
4K	9.3 ms	~200 ms	4.7%

**结论：** 只要 Task Graph 构建时间远小于 InCore 函数的执行时间，Orchestration Function 就不会成为瓶颈。  
在实际场景中：

- **构建时间  $O(N^2)$ :** 主要来自 Flash Attention 的  $N \times N$  交叉依赖
- **执行时间  $O(N^2)$ :** Flash Attention 本身就是  $O(N^2)$  复杂度
- **比例稳定:** 构建时间占比基本保持在 2-5%

6.4 优势总结

特性	传统方法	PTO Orchestration
动态 shape	需要多个编译版本	✅ 原生支持
编译时间	每种 shape 都要编译	✅ InCore 只编译一次
运行时开销	无	~2-5% (可接受)
灵活性	低	✅ 任意控制流
依赖管理	手动	✅ 自动建立

快速开始

安装依赖

```
pip install -r requirements.txt
```

运行示例

```
# Fused Softmax 示例
python examples/pto_fused_softmax.py

# LLaMA 7B Layer 示例
python examples/pto_llama7B_dynamic.py

# 性能测试
cd examples && gcc -O2 -I.. -o test_llama_performance
test_llama_performance.c
./test_llama_performance
```

生成的文件结构

```
examples/
├── output_pto/                # PTO Assembly
│   └── llama7b/
│       └── llama7b_layer.pto
├── output_arm64/             # ARM64 NEON 代码
```



```
├── llama7b/
│   ├── tile_matmul.c
│   ├── rmsnorm_tile.c
│   └── llama_layer_dynamic_orchestration.c
├── output_cuda/          # CUDA 代码
│   ├── llama7b/
│   │   └── *.cu
├── output_ascend910b/    # Ascend 910B 代码
│   ├── llama7b/
│   │   ├── *.cpp
│   │   ├── llama_layer_dynamic_task_graph.txt
│   │   └── llama_layer_dynamic_task_graph.pdf
```

---

## License

MIT License