中国科学院大学计算机学院专业选修课

GPU架构与编程

第十课：PTX编程（四）

赵地
中科院计算所
2025年秋季学期

讲授内容

- ➢特殊寄存器
  - ✓概述
  - ✓线程定位
  - ✓设备控制（Warp、CTA、grid控制）
  - ✓资源查询
  - ✓性能分析
- ✓提示符（Directives）
- ✓PTX编程总结
- ✓大作业一加分题（PTX编程）讲解

特殊寄存器（Special Registers）

- ✓概述
  - ✓特殊寄存器：预定义的只读寄存器，用于在并行线程执行过程中获取硬件状态和线程信息。
    - ✓线程定位：%tid + %ctaid + %ntid + %nctaid → 计算全局线程ID。
    - ✓设备控制（Warp、CTA、grid控制）。
    - ✓资源查询：%total_smem_size、%nsmid → 避免资源超限。
    - ✓性能分析：时钟与性能计数器 → 定位瓶颈。

特殊寄存器（Special Registers）

- ✓概述
  - ✓线程定位：%tid + %ctaid + %ntid + %nctaid → 计算全局线程ID。
    - ✓%tid
      - ✓功能：存储当前线程在所属线程块（CTA）内的三维局部ID。
      - ✓维度：
        - ✓%tid.x：线程在CTA x 轴的位置（0 到 %ntid.x-1）。
        - ✓%tid.y：y 轴位置（0 到 %ntid.y-1）。
        - ✓%tid.z：z 轴位置（0 到 %ntid.z-1）。
      - ✓用途：计算线程在数据分块中的索引，例如 index = %tid.x + %tid.y * blockDim.x。

特殊寄存器（Special Registers）

- ✓概述
  - ✓线程定位：%tid + %ctaid + %ntid + %nctaid → 计算全局线程ID。
    - ✓%ntid
      - ✓功能：存储当前线程块的维度大小（线程数量）。
      - ✓维度：
        - ✓%ntid.x：x 轴线程数。
        - ✓%ntid.y：y 轴线程数。
        - ✓%ntid.z：z 轴线程数。
      - ✓与 %tid 关系：%tid 的取值范围由 %ntid 限定。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：Warp 控制。
    - ✓%laneid
      - ✓功能：返回当前线程在 Warp 内的唯一序号（0 到 WARP_SZ-1）。
      - ✓特性：
        - ✓Warp 是硬件调度单位（通常 32 线程）。
        - ✓同一 Warp 内的线程执行相同指令（SIMT）。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：Warp 控制。
    - ✓%warpid
      - ✓功能：返回当前线程所属 Warp 在 CTA 内的序号（0 到 %nwarpid-1）。
      - ✓不同于 %laneid（Warp 内局部ID），%warpid 是 CTA 内 Warp 的全局ID。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：Warp 控制。
    - ✓ %nwarpid
      - ✓功能：存储当前 CTA 包含的 Warp 总数。
      - ✓计算：%nwarpid = ceil(%ntid.x * %ntid.y * %ntid.z / WARP_SZ)。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：Warp 控制。
    - ✓Warp 掩码寄存器，生成 Warp 级的位掩码（32位），用于条件执行

寄存器	功能
%lanemask_eq	等于当前 %laneid 的线程掩码（仅1位为1）。
%lanemask_le	laneid <= 当前线程 的掩码。
%lanemask_lt	laneid < 当前线程 的掩码。
%lanemask_ge	laneid >= 当前线程 的掩码。
%lanemask_gt	laneid > 当前线程 的掩码。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：CTA控制：
    - ✓%ctaid
      - ✓功能：存储当前线程块（CTA）在网格（Grid）中的三维ID。
      - ✓维度：
        - ✓%ctaid.x：CTA 在 Grid x 轴的位置（0 到 %nctaid.x-1）。
        - ✓%ctaid.y：y 轴位置（0 到 %nctaid.y-1）。
        - ✓%ctaid.z：z 轴位置（0 到 %nctaid.z-1）。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：CTA控制：
    - ✓%nctaid
      - ✓功能：存储网格（Grid）的维度（CTA 数量）。
      - ✓维度：
        - ✓%nctaid.x：Grid x 轴 CTA 数量。
        - ✓%nctaid.y：y 轴 CTA 数量。
        - ✓%nctaid.z：z 轴 CTA 数量。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：CTA控制：
    - ✓%nctaid
      - ✓功能：存储网格（Grid）的维度（CTA 数量）。
      - ✓维度：
        - ✓%nctaid.x：Grid x 轴 CTA 数量。
        - ✓%nctaid.y：y 轴 CTA 数量。
        - ✓%nctaid.z：z 轴 CTA 数量。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：CTA控制：
    - ✓%smid
      - ✓功能：返回当前线程所在 流式多处理器（SM）的物理ID。
      - ✓用途：调试或优化 SM 负载均衡。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：CTA控制：
    - ✓%nsmid
      - ✓功能：存储当前 GPU 的 SM 总数。
      - ✓与 %smid 关系：%smid 取值范围为 0 到 %nsmid-1。

特殊寄存器（Special Registers）

- ✓概述
  - ✓设备控制（Warp、CTA、grid控制）：GPU控制：
    - ✓%gridid
      - ✓功能：返回当前网格（Grid）在 应用生命周期内的唯一ID。
      - ✓特性：用于区分多次内核启动。

特殊寄存器（Special Registers）

- ✓概述
  - ✓性能分析：时钟与性能计数器 → 定位瓶颈。
    - ✓时钟寄存器
      - ✓%clock / %clock_hi：
        - ✓返回 SM 核心时钟计数（32位低/高部分）。
      - ✓%clock64：
        - ✓直接返回64位时钟周期数。
      - ✓用途：测量指令耗时（非 wall-time）。

特殊寄存器（Special Registers）

- ✓概述
  - ✓资源查询
    - ✓共享内存信息

寄存器	功能
%total_smem_size	当前 CTA 可用的共享内存总量（字节）。
%dynamic_smem_size	动态分配的共享内存大小（字节）。

特殊寄存器（Special Registers）

- ✓概述
  - ✓性能分析：时钟与性能计数器 → 定位瓶颈。
    - ✓性能计数器
      - ✓%pm0 - %pm7：8个32位硬件性能计数器。
      - ✓%pm0_64 - %pm7_64：8个64位扩展性能计数器。
    - ✓用途：分析缓存命中率、指令吞吐量等。

特殊寄存器（Special Registers）

- ✓概述
  - ✓性能分析：时钟与性能计数器 → 定位瓶颈。
    - ✓%envreg<32>
      - ✓功能：访问 32个环境寄存器（通过索引 <n> 指定，0 ≤ n ≤ 31）。
      - ✓用途：传递驱动或编译器的自定义参数（如调试标志）。

特殊寄存器（Special Registers）

- ✓概述
  - ✓性能分析：时钟与性能计数器 → 定位瓶颈。
    - ✓%globaltimer
      - ✓功能：
        - ✓%globaltimer_lo：全局计时器低32位。
        - ✓%globaltimer_hi：全局计时器高32位。
        - ✓%globaltimer：完整64位计时器（部分架构支持）。
      - ✓与 %clock 区别：跨SM同步，适合多CTA时间测量。

讲授内容

- ➢特殊寄存器
  - ✓概述
  - ✓线程定位
  - ✓设备控制（Warp、CTA、grid控制）
  - ✓资源查询
  - ✓性能分析
- ✓提示符（Directives）
- ✓PTX编程总结
- ✓大作业一加分题（PTX编程）讲解

特殊寄存器：线程定位%tid

- ✓特殊寄存器：线程定位%tid
  - ✓%tid 的功能
    - ✓功能
      - ✓%tid是PTX架构中预定义的 线程标识符寄存器（Thread ID Register），为每个线程提供其在所属线程块（CTA）内的三维空间坐标。
    - ✓关键特性：
      - ✓只读性：硬件自动赋值，软件不可修改
      - ✓三维结构：包含 %tid.x, %tid.y, %tid.z 三个子寄存器
      - ✓作用域：仅在所属 CTA 内有效

特殊寄存器：线程定位%tid

✓特殊寄存器：线程定位%tid

✓坐标分量定义

分量	取值范围	内存占用	定位意义
%tid.x	$0 \le x < \%ntid.x$	32-bit	CTA 内 x 轴线程位置
%tid.y	$0 \le y < \%ntid.y$	32-bit	CTA 内 y 轴线程位置
%tid.z	$0 \le z < \%ntid.z$	32-bit	CTA 内 z 轴线程位置

特殊寄存器：线程定位%tid

✓特殊寄存器：线程定位%tid

✓坐标生成原理

✓当 GPU 启动内核时，硬件调度器自动为每个线程分配唯一三维坐标；

✓此过程在线程实例化阶段完成，无运行时开销。

线程坐标 = (硬件调度单元)
- ├─ 分配 x 轴索引
- ├─ 分配 y 轴索引
- └─ 分配 z 轴索引

特殊寄存器：线程定位%tid

- ✓特殊寄存器：线程定位%tid
  - ✓核心应用场景
    - ✓数据并行索引计算
    - ✓分支优化
    - ✓线程协作

特殊寄存器：线程定位%tid

- ✓特殊寄存器：线程定位%tid
  - ✓硬件实现
    - ✓零开销特性：
      - ✓值在线程启动时固化，无读取延迟
      - ✓不占用通用寄存器空间
    - ✓Warp 级优化：
      - ✓同 Warp 内线程的 %tid.x 连续递增
      - ✓硬件自动生成：%laneid = %tid.x % WARP_SZ

特殊寄存器：线程定位%tid

- ✓特殊寄存器：线程定位%tid
  - ✓硬件实现
    - ✓物理寄存器映射：
      - ✓SM 内部维护 Thread Scheduler Table
      - ✓运行时映射到 SM 的寄存器文件 (Register File Slot)

```
Thread[0] → RF Slot[0] → { %tid.x=0, %tid.y=0, %tid.z=0 }
Thread[1] → RF Slot[1] → { %tid.x=1, %tid.y=0, %tid.z=0 }
...
```

特殊寄存器：线程定位%tid

- ✓特殊寄存器：线程定位%tid
  - ✓性能关键点
    - ✓访存模式优化：
      - ✓优先使用 %tid.x（连续内存访问）
      - ✓避免 %tid.y/%tid.z 主导的跨距访问
    - ✓控制流建议：

```
// 推荐：x 维度分支（同 Warp 内一致）
setp.eq.s32 %p, %tid.z, 0

// 避免：z 维度分支（导致 Warp 发散）
setp.eq.s32 %p, %tid.x, 0  // 同 Warp 内线程行为一致
```

特殊寄存器：线程定位%tid

- ✓特殊寄存器：线程定位%tid
  - ✓性能关键点
    - ✓维度配置黄金法则：
      - ✓理由：最大化内存访问连续性

%ntid.x ≥ %ntid.y ≥ %ntid.z

特殊寄存器：线程定位%tid

- ✓特殊寄存器：线程定位%tid
  - ✓与相关寄存器交互

寄存器	关联公式	协同作用
%ntid	%tid.dim < %ntid.dim	定义坐标边界
%ctaid	global_id = %ctaid * %ntid + %tid	计算全局线程 ID
%laneid	%laneid = %tid.x % 32	Warp 内定位

特殊寄存器：线程定位%tid

✓特殊寄存器：线程定位%tid

- ✓总结
  - ✓%tid 是 PTX 并行编程的基石寄存器，其核心价值在于：
    - ✓提供线程在 CTA 内的空间坐标
    - ✓实现零开销线程定位
    - ✓支撑数据分布与通信原语
    - ✓赋能硬件级执行效率优化
  - ✓正确运用 %tid 可使性能提，是 GPU 高性能编程的关键掌握点。

讲授内容

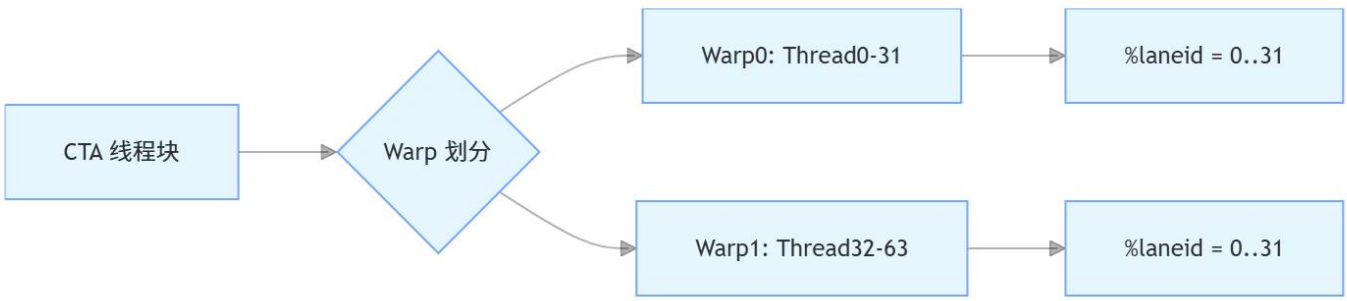
- ➢特殊寄存器
  - ✓概述
  - ✓线程定位
  - ✓设备控制（Warp、CTA、grid控制）
  - ✓资源查询
  - ✓性能分析
- ✓提示符（Directives）
- ✓PTX编程总结
- ✓大作业一加分题（PTX编程）讲解

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓%laneid 的本质与硬件原理
    - ✓%laneid 是 SIMT（单指令多线程）架构的核心寄存器，标识线程在 Warp 内的物理位置。
      - ✓关键特性：
        - ✓只读性：由 GPU 硬件在 Warp 分配时固化
        - ✓连续分布：取值范围 [0，WARP_SZ-1]（通常 WARP_SZ=32）
        - ✓硬件映射：直接对应 SM 的 SIMD 通道号
    - ✓Warp 划分按 %tid.x 连续分组，与 y/z 维度无关

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓%laneid 的本质与硬件原理
    - ✓硬件生成逻辑



特殊寄存器：%laneid

✓特殊寄存器：Warp控制%laneid

✓核心功能解析

✓Warp 内线程定位

寄存器	作用域	唯一性	典型用途
%tid	CTA 内	三维唯一	全局数据索引
%laneid	Warp 内	一维唯一	Warp 级协作指令

特殊寄存器：%laneid

✓特殊寄存器：Warp控制%laneid

✓核心功能解析

✓与 Warp 操作指令的协同

```
// 示例：Warp 内数据广播 (shfl.sync)
shfl.sync.down.b32 %dst, %src, 1, 0xFFFFFFFF;
// 语义：将 %src 从 %laneid=0 的线程广播到所有线程
```

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓核心功能解析
    - ✓掩码生成基础

```
// 生成 %laneid > 当前线程的掩码
mov.u32 %mask, %lanemask_gt;
// 等价于：%mask = (0xFFFFFFFF << (%laneid+1))
```

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓硬件实现深度
    - ✓SM 内部结构映射

SM 组件	与 %laneid 的关联
SIMD 单元	直接对应物理 SIMD 通道编号
Warp 调度器	通过 %laneid 映射指令发射槽
寄存器文件	按 %laneid 分区访问

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓硬件实现深度
    - ✓零延迟访问机制
      - ✓固化存储：值存储在 SM 的 Warp State Register File
      - ✓直接通路：执行单元直读，不经过通用寄存器堆
      - ✓并行读取：同 Warp 内 32 线程同时无冲突访问

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓关键应用场景
    - ✓Warp 级规约

```
// 并行求和（Butterfly 模式）
.reduction:
    shfl.sync.xor.b32   %val, %val, mask;  // XOR 交换数据
    add.f32             %sum, %sum, %val;  // 跨通道加法
    setp.ne.u32          %p, mask, 0;      // 判断迭代终止
    @%p                 bra .reduction;
```

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓关键应用场景
    - ✓条件执行优化

```
// 仅高位线程执行写操作
setp.gt.u32 %p_active, %laneid, 16;
@%p_active   st.shared.f32 [addr], %data;
// 避免 Warp 内低效线程执行高开销操作
```

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓关键应用场景
    - ✓数据重排

```
// 矩阵转置 (32x32 分块)
mov.u32    %read_index, %laneid;
shfl.sync.bfly.b32 %write_index, %read_index, 16;
// 奇偶分组交换
```

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓性能优化关键
    - ✓Warp 利用率黄金法则
      - ✓优化目标：避免 %laneid 导致的条件分支发散

Warp 效率 = (活跃线程数) / WARP_SZ

特殊寄存器：%laneid

- ✓特殊寄存器：Warp控制%laneid
  - ✓性能优化关键
    - ✓指令选择建议

操作类型	推荐指令	避坑指南
数据广播	shfl.sync.idx	避免 ld.shared+shfl 组合
掩码生成	%lanemask_xx 系列	优于手动计算掩码
原子操作	atom.warp 级指令	避免跨 Warp 原子操作

特殊寄存器：%laneid

✓特殊寄存器：Warp控制%laneid

- ✓总结
  - ✓%laneid 是 PTX 编程的 Warp 级定位基石。
  - ✓核心价值：
    - ✓物理位置锚点：精准定位线程在 SIMD 单元的位置
    - ✓协作指令枢纽：实现零开销 Warp 内数据交换
    - ✓执行效率开关：直接控制 SIMD 利用率
    - ✓架构兼容核心：跨越 GPU 代际的稳定接口
  - ✓合理利用 %laneid 可提升 Warp 利用率，是高性能 GPU 编程不可替代的核心寄存器。

特殊寄存器：%warpid

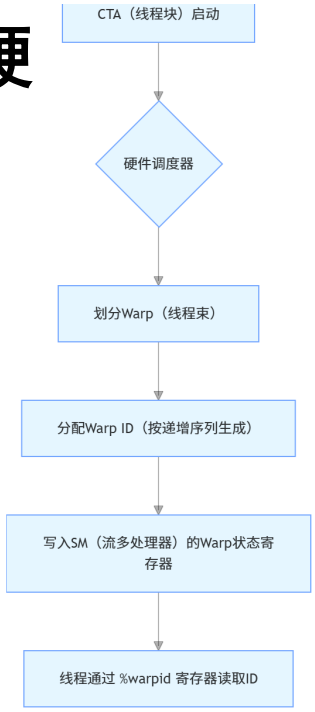
✓特殊寄存器%warpid 的本质与硬件原理

- ✓作用：%warpid 是标识线程在 CTA 内所属 Warp 的逻辑 ID 的关键寄存器
- ✓核心特性：
  - ✓动态性：值在 CTA 生命周期内固定，但不同 CTA 中相同位置线程的 %warpid 不同
  - ✓局部性：作用域限定在所属 CTA 内（0 到 %nwarpid-1）
- ✓计算基础：%warpid = floor(全局线程ID / WARP_SZ) % %nwarpid

特殊寄存器：%warpid

✓特殊寄存器%warpid 的本质与硬件原理

✓硬件生成逻辑



特殊寄存器：%warpid

✓特殊寄存器%warpid 的本质与硬件原理

✓与相关寄存器的关系矩阵

寄存器	数学关系式	协同场景
%tid	全局线程序号 = %tid.x + %ntid.x*(%tid.y + %ntid.y*%tid.z)	CTA 内线程定位
%nwarpid	%nwarpid = ceil((%ntid.x * %ntid.y * %ntid.z) / WARP_SZ)	确定 Warp ID 范围
%ctaid	全局 Warp ID = %ctaid * %nwarpid + %warpid	跨 CTA 的 Warp 调度
%laneid	Warp 内线程ID = %laneid	Warp 内细粒度控制

特殊寄存器：%warpid

- ✓特殊寄存器%warpid 的本质与硬件原理
  - ✓核心应用场景
    - ✓Warp 级资源分区

```
// 为不同 Warp 分配独立的共享内存区域
mul.u32   %warp_offset, %warpid, 256  // 每个 Warp 256 字节
add.u32   %shmem_addr, %shmem_base, %warp_offset
```

特殊寄存器：%warpid

- ✓特殊寄存器%warpid 的本质与硬件原理
  - ✓核心应用场景
    - ✓跨 Warp 同步

```
// 确保所有 Warp 完成阶段工作
bar.sync  0, %nwarpid  // 等待 CTA 内所有 Warp
```

特殊寄存器：%warpid

- ✓特殊寄存器%warpid 的本质与硬件原理
  - ✓核心应用场景
    - ✓动态负载均衡

```
// 将计算任务按 Warp 分组分配
cvt.f32.u32  %task_id, %warpid
mul.f32      %workload, %task_id, %work_per_warp
```

特殊寄存器：%warpid

- ✓特殊寄存器%warpid 的本质与硬件原理
  - ✓硬件实现深度
    - ✓SM 内部结构映射

SM 组件	与 %warpid 的关联
Warp 调度槽	直接对应物理调度槽编号
寄存器文件分区	按 Warp ID 划分存储区域
指令发射单元	依赖 Warp ID 选择发射队列

特殊寄存器：%warpid

- ✓特殊寄存器%warpid 的本质与硬件原理
  - ✓硬件实现深度
    - ✓零开销访问机制
      - ✓专用存储：值存储在 SM 的 Warp Context Register File
      - ✓并行访问：同 Warp 所有线程同时读取相同值
      - ✓硬件优化：Ampere 架构引入 Warp ID 缓存 降低访问延迟

特殊寄存器：%warpid

- ✓特殊寄存器%warpid 的本质与硬件原理
  - ✓性能关键点
    - ✓Warp 调度黄金法则
      - ✓通过 %nwarpid 实时验证是否达标

```
最佳 Warp 数量 = min(
    SM 支持的最大驻留 Warp 数,
    ceil(问题规模 / WARP_SZ)
)
```

特殊寄存器：%warpid

✓特殊寄存器%warpid 的本质与硬件原理

- ✓性能关键点
  - ✓资源冲突规避

```
// 检查寄存器文件容量
cvt.u32.u16   %regs_needed, %reg_per_thread
mul.u32       %total_regs, %regs_needed, WARP_SZ
mul.u32       %total_regs, %total_regs, %nwarpid
cmp.le.u32    %p_ok, %total_regs, SM_REG_CAPACITY
@!%p_ok       trap
```

特殊寄存器：%warpid

✓特殊寄存器%warpid 的本质与硬件原理

- ✓总结
  - ✓%warpid是GPU并行架构的Warp调度锚点寄存器
  - ✓价值：
    - ✓资源分配枢纽：实现 CTA 内 Warp 级别的资源隔离
    - ✓调度控制节点：构建跨 Warp 的同步和协作原语
    - ✓性能诊断窗口：通过 Warp 分布分析负载均衡
    - ✓架构兼容接口：统一 Volta 到 Ampere 的调度模型
  - ✓合理利用 %warpid 可使 SM 利用率提高，是解决 Warp 级负载不均衡 的核心工具。

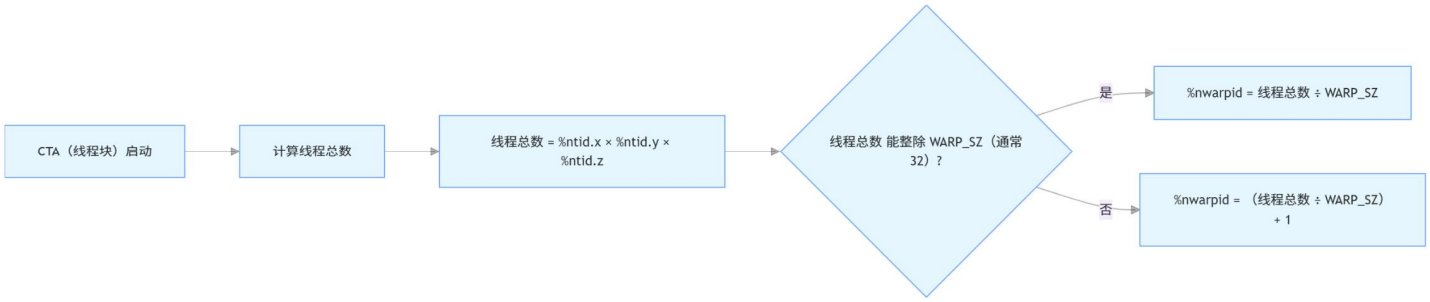
特殊寄存器：%nwarpid

- ✓特殊寄存器%nwarpid 的本质与功能
  - ✓%nwarpid 是预定义的 线程块 Warp 总数寄存器，存储当前线程块（CTA）包含的 Warp 总量。
  - ✓核心特性：
    - ✓只读性：由 GPU 硬件在 CTA 启动时自动计算
    - ✓动态性：值由线程块维度 %ntid 决定
  - ✓数学定义（其中 WARP_SZ 通常是 32）：

$$\%nwarpid = \left\lceil \frac{\%ntid.x \times \%ntid.y \times \%ntid.z}{WARP\_SZ} \right\rceil$$

特殊寄存器：%nwarpid

- ✓硬件实现机制
  - ✓实时计算流程



特殊寄存器：%nwarpid

✓硬件实现机制
  ✓SM 内部存储

存储位置	访问特性	更新时机
SM Context 寄存器	零延迟读取 (1 cycle)	CTA 启动时固化

特殊寄存器：%nwarpid

✓核心应用场景
  ✓Warp 级循环控制

```
// 遍历 CTA 内所有 Warp
mov.u32   %warp_index, 0;
.loop:
    setp.ge.u32  %p_done, %warp_index, %nwarpid;
    @%p_done     bra     EXIT;
    // 处理当前 Warp 的任务...
    add.u32      %warp_index, %warp_index, 1;
    bra          .loop;
```

特殊寄存器：%nwarpid

✓核心应用场景

✓资源预分配验证

```
// 检查共享内存是否足够分配所有 Warp
mul.u32   %warp_mem, %mem_per_warp, %nwarpid;
cmp.le.u32 %p_ok, %warp_mem, %total_smem;
@!%p_ok    trap;  // 资源不足时终止
```

特殊寄存器：%nwarpid

✓核心应用场景

✓原子操作分区

```
// 为每个 Warp 分配独立的原子计数器
mul.u32   %counter_addr, %warpid, 4;  // 每个 Warp 4 字节
atom.global.add.u32 %res, [%global_counter + %counter_addr], 1;
```

特殊寄存器：%nwarpid

✓与架构约束的关系

✓硬件限制矩阵

GPU 架构	最大 %nwarpid	物理限制
Pascal	32	SM 最大驻留 Warp 数
Volta	64	增强型调度器
Ampere	64	MIG 分区动态调整

特殊寄存器：%nwarpid

✓与架构约束的关系

✓超限检测代码

```
// 检测是否超出硬件支持
cmp.gt.u32 %p_err, %nwarpid, 64;
@%p_err     st.global.u32 [%error_flag], 1;
```

特殊寄存器：%nwarpid

✓性能优化关键

✓Warp 数量黄金法则

```
理想 %nwarpid = min(
    SM 最大驻留 Warp 数,
    ceil(问题规模 / WARP_SZ),
    floor(共享内存 / Warp 内存需求)
)
```

特殊寄存器：%nwarpid

✓性能优化关键

✓负载均衡策略

```
// 动态任务分配 (基于 Warp ID)
mul.u32   %task_per_warp, %total_tasks, %warpid;
div.u32   %task_per_warp, %task_per_warp, %nwarpid;
```

特殊寄存器：%nwarpid

✓与相关寄存器的协同

寄存器	协同公式	联合应用场景
%warpid	0 ≤ %warpid < %nwarpid	Warp 遍历循环
%smid	%smid × max_warps + %warpid	全局 Warp ID 计算
%ctaid	(CTA 偏移) × max_warps	跨 CTA 的 Warp 调度

特殊寄存器：%nwarpid

✓总结

- ✓%nwarpid 是 GPU 并行编程的 Warp 规模锚点寄存器
- ✓价值：
  - ✓资源规划基准：确定 CTA 内 Warp 级资源分配上限
  - ✓循环控制枢纽：实现 Warp 粒度的并行任务分发
  - ✓硬件兼容层：屏蔽不同架构的 Warp 数量差异
  - ✓性能诊断镜：通过实际值与理论值差距分析负载均衡
- ✓合理利用 %nwarpid 可使 SM 利用率提升，是解决 Warp 级资源碎片化 的核心工具。

特殊寄存器：%lanemask_eq

- ✓%lanemask_eq 是 PTX 架构中用于 Warp 级线程标识 的关键掩码寄存器
  - ✓核心功能：
    - ✓位掩码生成：返回一个 32 位无符号整数掩码
    - ✓相等标识：仅当前线程对应的 lane 位被置位（1），其余位为 0
- ✓数学表达，其中 %laneid 是当前线程在 Warp 中的位置（0-31）：

$$\%lanemask\_eq = 2^{\%laneid}$$

特殊寄存器：%lanemask_eq

- ✓硬件实现原理
  - ✓位宽：32 位（固定）
  - ✓访问特性：
    - ✓零延迟（Ampere+）
    - ✓单周期完成（Pascal+）
  - ✓存储位置：SM 的 Warp Mask Register File

特殊寄存器：%lanemask_eq

- ✓核心应用场景
  - ✓线程级原子操作
    - ✓原理：掩码确保仅当前线程的原子操作生效

```
// 仅当前线程执行原子操作
atom.global.add.u32 %res, [%addr], %value, %lanemask_eq;
```

特殊寄存器：%lanemask_eq

- ✓核心应用场景
  - ✓条件执行控制

```
// 仅当前线程执行高开销指令
@%lanemask_eq call _expensive_function;
```

特殊寄存器：%lanemask_eq

- ✓性能优化关键
  - ✓掩码使用黄金法则
    - ✓%lanemask_eq 效率：100%（单线程操作）
    - ✓对比：全 Warp 掩码 (0xFFFFFFFF) 效率仅 3.125%

$$\text{掩码效率} = 1 - \frac{\text{未置位位数}}{32}$$

特殊寄存器：%lanemask_eq

- ✓性能优化关键
  - ✓指令选择策略

操作类型	推荐指令	性能收益 vs 全 Warp
原子操作	atom.global + %lanemask_eq	加速
内存访问	ld.global + %lanemask_eq	缓存污染降低
函数调用	条件执行 + %lanemask_eq	分支开销降低

特殊寄存器：%lanemask_eq

✓性能优化关键

✓与相关寄存器协同

寄存器	协同关系	联合应用场景
%laneid	%lanemask_eq = 1 << %laneid	Warp 内线程精确定位
%warpid	Warp 上下文标识	跨 Warp 任务分配
%activemask	当前活跃线程掩码	动态线程调度

特殊寄存器：%lanemask_eq

✓错误处理与验证

```
// 掩码有效性验证
popc.b32   %count, %lanemask_eq;
cmp.ne.u32 %p_err, %count, 1;    // 应仅 1 位置位
@%p_err    trap;
```

特殊寄存器：%lanemask_eq

✓实例

✓线程私有存储

```
// 为当前线程分配独立存储区
mul.u32 %offset, %lanemask_eq, 128;  // 掩码=2^laneid
add.u32 %private_addr, %shared_base, %offset;
st.shared.f32 [%private_addr], %data;
```

特殊寄存器：%lanemask_eq

✓实例

✓精准性能分析

```
// 仅测量当前线程执行时间
mov.u32 %t1, %clock;
@%lanemask_eq {
    // 被测量的代码段
    fma.rn.f32 %res, %a, %b, %c;
}
mov.u32 %t2, %clock;
sub.u32 %cycles, %t2, %t1;
```

特殊寄存器：%lanemask_eq

- ✓总结
  - ✓%lanemask_eq 是 GPU 细粒度并行控制的 原子级掩码工具
  - ✓价值在于：
    - ✓线程级精准控制：实现 Warp 内单线程操作
    - ✓零开销隔离：消除无关线程的资源竞争
    - ✓性能分析探针：提供线程粒度的性能采样
    - ✓虚拟化支持：适配 MIG 分区架构
  - ✓合理使用 %lanemask_eq 可使原子操作吞吐量提升。

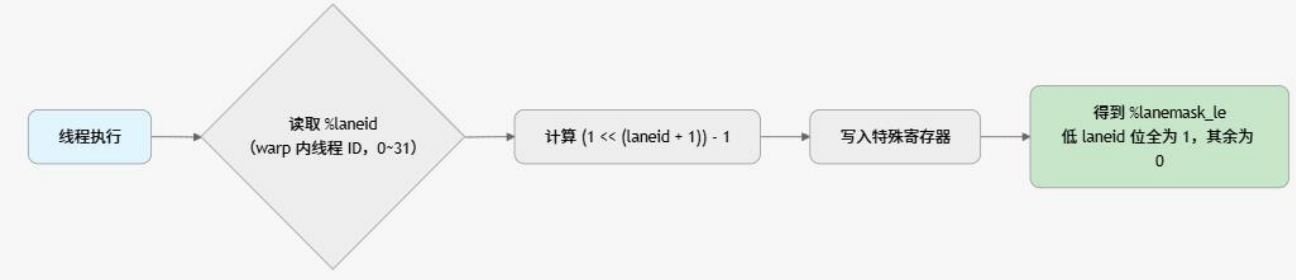
特殊寄存器：%lanemask_le

- ✓本质与功能
  - ✓%lanemask_le 是 PTX 架构中用于 Warp 级线程选择 的关键掩码寄存器，
  - ✓核心功能：
    - ✓位掩码生成：返回一个 32 位无符号整数掩码
    - ✓小于等于标识：所有 lane ID ≤ 当前线程 lane ID 的位被置位（1）
  - ✓数学表达，%laneid 是当前线程在 Warp 中的位置（0-31）：

$$\%lanemask\_le = \left(2^{(\%laneid+1)} - 1\right)$$

特殊寄存器：%lanemask_le

- ✓硬件实现原理
  - ✓位宽：32 位（固定）
  - ✓计算单元：SM 的 Mask Generation Unit (MGU)
  - ✓访问特性：
    - ✓Ampere 架构：0 延迟（硬件直通）
    - ✓Pascal 架构：延迟极低
  - ✓动态更新：值随线程位置实时变化



特殊寄存器：%lanemask_le

- ✓核心应用场景
  - ✓Warp 级前缀和 (Prefix Sum)
    - ✓原理：逐级累加低 ID 线程的值

```
// 并行前缀和计算
shfl.sync.up.b32 %partial_sum, %value, 1, %lanemask_le;
add.u32 %result, %value, %partial_sum;
```

特殊寄存器：%lanemask_le

✓核心应用场景
  ✓数据依赖链

```
// 顺序依赖执行
@%lanemask_le {
    // 仅低 ID 线程先执行
    st.shared.u32 [%shmem + %laneid*4], %data
    bar.sync %lanemask_le  // 同步低 ID 线程
}
```

特殊寄存器：%lanemask_le

✓核心应用场景
  ✓渐进式处理

```
// 按 lane ID 顺序处理数据
mov.u32 %step, 0
.loop:
    setp.le.u32 %p_active, %step, %laneid
    @%p_active process_data(%step)
    add.u32 %step, %step, 1
    setp.lt.u32 %p_continue, %step, 32
    @%p_continue bra .loop
```

特殊寄存器：%lanemask_le

✓性能优化关键
✓掩码效率模型
  ✓最佳情况：%laneid=31 → 高利用率
  ✓最差情况：%laneid=0 → 低利用率

$$\text{活跃线程比例} = \frac{\%laneid + 1}{32}$$

特殊寄存器：%lanemask_le

✓与相关寄存器协同

寄存器	协同关系	联合应用场景
%laneid	掩码计算基础	动态线程选择
%lanemask_lt	排除自身线程	严格前缀操作
%activemask	活跃线程过滤	异常处理
%warpid	跨 Warp 协调	大规模并行算法

特殊寄存器：%lanemask_le

✓实战示例

✓Warp 级归约

```
// 并行最大值归约
mov.u32 %max_val, %my_value;
.reduction:
    shfl.sync.up.b32 %other_val, %max_val, 1, %lanemask_le;
    max.u32 %max_val, %max_val, %other_val;
    setp.gt.u32 %p_continue, %step, 0
    @%p_continue bra .reduction
```

特殊寄存器：%lanemask_le

✓实战示例

✓动态负载均衡

```
// 按线程 ID 顺序处理任务
mov.u32 %tasks_done, 0;
.loop:
   // 仅当有任务且 ID≤当前线程时执行
   setp.le.u32 %p_work, %tasks_done, %laneid
   @%p_work    process_task(%tasks_done)
   add.u32 %tasks_done, %tasks_done, 1
   setp.lt.u32 %p_more, %tasks_done, TOTAL_TASKS
   @%p_more    bra .loop
```

特殊寄存器：%lanemask_le

- ✓总结
  - ✓%lanemask_le 是 GPU 细粒度并行控制的 渐进式掩码工具
  - ✓价值：
    - ✓有序并行引擎：实现 Warp 内线程的渐进式协作
    - ✓数据依赖管理：解决顺序敏感型算法的并行化难题
    - ✓资源效率优化：按需激活线程减少资源争用
    - ✓算法加速器：为前缀操作提供硬件级加速
  - ✓合理使用 %lanemask_le 可使前缀和类算法加速

特殊寄存器：%lanemask_le

- ✓总结
  - ✓%lanemask_le 是 GPU 细粒度并行控制的 渐进式掩码工具
  - ✓价值：
    - ✓有序并行引擎：实现 Warp 内线程的渐进式协作
    - ✓数据依赖管理：解决顺序敏感型算法的并行化难题
    - ✓资源效率优化：按需激活线程减少资源争用
    - ✓算法加速器：为前缀操作提供硬件级加速
  - ✓合理使用 %lanemask_le 可使前缀和类算法加速

特殊寄存器：%lanemask_lt

✓本质与功能

✓%lanemask_lt 是 PTX 架构中用于 Warp 级线程选择 的关键掩码寄存器

✓核心功能：

  ✓位掩码生成：返回一个 32 位无符号整数掩码

  ✓小于标识：所有 lane ID < 当前线程 lane ID 的位被置位（1）

✓数学表达，其中 %laneid 是当前线程在 Warp 中的位置（0-31）：

$$\%lanemask\_lt = (2^{\%laneid} - 1)$$

特殊寄存器：%lanemask_lt

✓硬件实现原理

  ✓位宽：32 位（固定）

  ✓计算单元：SM 的专用掩码生成单元（MGU）

  ✓访问特性：

    ✓Ampere 架构：0 延迟（硬件直通）

    ✓Pascal 架构：延迟极低

  ✓动态更新：值随线程位置实时变化

特殊寄存器：%lanemask_lt

- ✓核心应用场景
  - ✓严格前缀操作
    - ✓原理：仅累加更低 ID 线程的值

```
// 计算严格前缀和（排除当前线程）
shfl.sync.up.b32 %prefix_sum, %value, 1, %lanemask_lt;
```

特殊寄存器：%lanemask_lt

- ✓核心应用场景
  - ✓数据依赖链

```
// 顺序依赖执行（低 ID 线程优先）
@%lanemask_lt {
    // 仅更低 ID 线程执行
    process_data()
    bar.sync %lanemask_lt  // 同步低 ID 线程
}
```

特殊寄存器：%lanemask_lt

✓核心应用场景
  ✓渐进式处理

```
// 按 lane ID 顺序处理数据
mov.u32 %step, 0
.loop:
    setp.lt.u32 %p_active, %step, %laneid
    @%p_active process_data(%step)
    add.u32 %step, %step, 1
    setp.lt.u32 %p_continue, %step, 32
    @%p_continue bra .loop
```

特殊寄存器：%lanemask_lt

✓性能优化关键
  ✓掩码效率模型
    ✓最佳情况：%laneid=31 → 高利用率
    ✓最差情况：%laneid=0 → 低利用率

$$\text{活跃线程比例} = \frac{\%laneid}{32}$$

特殊寄存器：%lanemask_lt

✓性能优化关键
  ✓指令选择策略

算法类型	推荐模式	性能收益 vs 全 Warp
前缀操作	shfl.up + %lanemask_lt	加速
顺序依赖	条件执行 + 掩码同步	分支开销降低
数据广播	shfl.idx + 掩码	带宽利用率提升

特殊寄存器：%lanemask_lt

✓性能优化关键
  ✓与相关寄存器协同

寄存器	协同关系	联合应用场景
%laneid	掩码计算基础	动态线程选择
%lanemask_le	包含自身线程	包含当前线程的前缀操作
%activemask	活跃线程过滤	异常处理
%warpid	跨 Warp 协调	大规模并行算法

特殊寄存器：%lanemask_lt

✓错误处理与验证

```
// 掩码有效性验证
popc.b32   %pop_count, %lanemask_lt;
cmp.ne.u32 %p_err, %pop_count, %laneid;  // 应等于当前 laneid
@%p_err    trap;  // 触发硬件异常
```

特殊寄存器：%lanemask_lt

✓实战示例

✓Warp 级归约（排除自身）

```
// 并行最小值归约（排除当前线程）
mov.u32 %min_val, MAX_INT;
.reduction:
    shfl.sync.up.b32 %other_val, %min_val, 1, %lanemask_lt;
    min.u32 %min_val, %min_val, %other_val;
    setp.gt.u32 %p_continue, %step, 0
    @%p_continue bra .reduction
```

特殊寄存器：%lanemask_lt

✓实战示例

✓动态负载均衡

```
// 按线程 ID 顺序处理任务（排除自身）
mov.u32 %tasks_done, 0;
.loop:
    // 仅当有任务且 ID<当前线程时执行
    setp.lt.u32 %p_work, %tasks_done, %laneid
    @%p_work    process_task(%tasks_done)
    add.u32 %tasks_done, %tasks_done, 1
    setp.lt.u32 %p_more, %tasks_done, TOTAL_TASKS
    @%p_more    bra .loop
```

特殊寄存器：%lanemask_lt

✓总结

✓%lanemask_lt 是 GPU 细粒度并行控制的 严格前缀掩码工具

✓价值：

✓精确依赖控制：实现 Warp 内线程的严格顺序协作

✓数据隔离引擎：解决需要排除当前线程的并行算法

✓资源效率优化：按需激活线程减少资源争用

✓算法加速器：为严格前缀操作提供硬件级加速

✓合理使用 %lanemask_lt 可使严格前缀类算法加速

特殊寄存器：%lanemask_ge

- ✓本质与功能
  - ✓%lanemask_ge 是 PTX 架构中用于 Warp 级线程选择 的关键掩码寄存器
    - ✓核心功能：
      - ✓位掩码生成：返回一个 32 位无符号整数掩码
      - ✓大于等于标识：所有 lane ID ≥ 当前线程 lane ID 的位被置位（1）
  - ✓数学表达：

$$\%lanemask\_ge = (0xFFFFFFFF \gg (31 - \%laneid))$$

特殊寄存器：%lanemask_ge

- ✓硬件实现原理
  - ✓位宽：32 位（固定）
  - ✓计算单元：SM 的专用掩码生成单元（MGU）
  - ✓访问特性：
    - ✓Ampere 架构：0 延迟（硬件直通）
    - ✓Pascal 架构：延迟极低
  - ✓动态更新：值随线程位置实时变化

特殊寄存器：%lanemask_ge

- ✓应用场景
  - ✓后缀操作 (Suffix Operations)
    - ✓原理：累加更高 ID 线程的值

```
// 计算后缀最大值（包含当前线程）
shfl.sync.down.b32 %suffix_max, %value, 1, %lanemask_ge;
max.u32 %result, %value, %suffix_max;
```

特殊寄存器：%lanemask_ge

- ✓应用场景
  - ✓数据反向依赖链

```
// 逆序依赖执行（高 ID 线程优先）
@%lanemask_ge {
    // 仅当前及更高 ID 线程执行
    process_data()
    bar.sync %lanemask_ge  // 同步高 ID 线程
}
```

特殊寄存器：%lanemask_ge

✓应用场景
  ✓渐进式反向处理

```
// 按 lane ID 逆序处理数据
mov.u32 %step, 31
.loop:
    setp.ge.u32 %p_active, %step, %laneid
    @%p_active process_data(%step)
    sub.u32 %step, %step, 1
    setp.ge.u32 %p_continue, %step, 0
    @%p_continue bra .loop
```

特殊寄存器：%lanemask_ge

✓性能优化关键
  ✓掩码效率模型
    ✓最佳情况：%laneid=0 → 高利用率
    ✓最差情况：%laneid=31 → 低利用率

$$\text{活跃线程比例} = \frac{32 - \%laneid}{32}$$

特殊寄存器：%lanemask_ge

✓性能优化关键
  ✓指令选择策略

算法类型	推荐模式	性能收益 vs 全 Warp
后缀操作	shfl.down + %lanemask_ge	加速
逆序依赖	条件执行 + 掩码同步	分支开销降低
数据广播	shfl.idx + 掩码	带宽利用率提升

特殊寄存器：%lanemask_ge

✓与相关寄存器协同

寄存器	协同关系	联合应用场景
%laneid	掩码计算基础	动态线程选择
%lanemask_gt	排除自身线程	严格后缀操作
%activemask	活跃线程过滤	异常处理
%warpid	跨 Warp 协调	大规模并行算法

特殊寄存器：%lanemask_ge

✓错误处理与验证

```
// 掩码有效性验证
popc.b32   %pop_count, %lanemask_ge;
add.u32    %expected, 32, -%laneid;  // 32 - laneid
cmp.ne.u32 %p_err, %pop_count, %expected;
@%p_err    trap;  // 触发硬件异常
```

特殊寄存器：%lanemask_ge

✓示例

✓Warp 级后缀归约

```
// 并行最大值后缀归约（包含当前线程）
.reduction:
    shfl.sync.down.b32 %other_val, %max_val, 1,
%lanemask_ge;
    max.u32 %max_val, %max_val, %other_val;
    setp.gt.u32 %p_continue, %step, 0
    @%p_continue bra .reduction
```

特殊寄存器：%lanemask_ge

✓示例

✓动态负载均衡（逆序）

```
// 按线程 ID 逆序处理任务
mov.u32 %tasks_done, 31;
.loop:
    // 仅当有任务且 ID≥当前线程时执行
    setp.ge.u32 %p_work, %tasks_done, %laneid
    @%p_work    process_task(%tasks_done)
    sub.u32 %tasks_done, %tasks_done, 1
    setp.ge.u32 %p_more, %tasks_done, 0
    @%p_more    bra .loop
```

特殊寄存器：%lanemask_ge

✓总结

- ✓%lanemask_ge 是 GPU 细粒度并行控制的 后缀掩码工具
- ✓价值：
  - ✓逆序并行引擎：实现 Warp 内线程的逆序协作
  - ✓后缀操作加速：优化最大值/最小值后缀查找等算法
  - ✓资源效率优化：按需激活高 ID 线程减少资源争用
  - ✓算法加速器：为后缀操作提供硬件级加速
- ✓合理使用 %lanemask_ge 可使后缀操作类算法加速

特殊寄存器：%lanemask_gt

- ✓功能
  - ✓%lanemask_gt 是 PTX 架构中用于 Warp 级线程选择 的高级掩码寄存器
  - ✓核心功能是：
    - ✓位掩码生成：返回一个 32 位无符号整数掩码
    - ✓大于标识：所有 lane ID > 当前线程 lane ID 的位被置位（1）
  - ✓数学表达，其中 %laneid 是当前线程在 Warp 中的位置（0-31）：

$$%lanemask\_gt = (0xFFFFFFFF \gg (32 - %laneid)) \gg 1$$

特殊寄存器：%lanemask_gt

- ✓硬件实现原理
  - ✓位宽：32 位（固定）
  - ✓计算单元：SM 的专用掩码生成单元（MGU）
  - ✓访问特性：
    - ✓Ampere 架构：0 延迟（硬件直通）
    - ✓Pascal 架构：延迟极低
  - ✓动态更新：值随线程位置实时变化

特殊寄存器：%lanemask_gt

- ✓核心应用场景
  - ✓严格后缀操作
    - ✓原理：仅累加更高 ID 线程的值

```
// 计算严格后缀最大值（排除当前线程）
shfl.sync.down.b32 %strict_suffix, %value, 1, %lanemask_gt;
max.u32 %result, %value, %strict_suffix;
```

特殊寄存器：%lanemask_gt

- ✓核心应用场景
  - ✓数据反向依赖链

```
// 逆序依赖执行（仅更高 ID 线程）
@%lanemask_gt {
    // 仅更高 ID 线程执行
    process_data()
    bar.sync %lanemask_gt  // 同步高 ID 线程
}
```

特殊寄存器：%lanemask_gt

✓核心应用场景

✓渐进式反向处理

```
// 按 lane ID 逆序处理数据（排除自身）
mov.u32 %step, %laneid + 1
.loop:
    setp.le.u32 %p_active, %step, 31
    @%p_active process_data(%step)
    add.u32 %step, %step, 1
    setp.le.u32 %p_continue, %step, 31
    @%p_continue bra .loop
```

特殊寄存器：%lanemask_gt

✓性能优化关键

✓掩码效率模型

✓最佳情况：%laneid=0 → 高利用率

✓最差情况：%laneid=31 → 低利用率

$$\text{活跃线程比例} = \frac{31 - \%laneid}{32}$$

特殊寄存器：%lanemask_gt

✓性能优化关键
  ✓指令选择策略

算法类型	推荐模式	性能收益 vs 全 Warp
严格后缀操作	shfl.down + %lanemask_gt	加速
逆序依赖	条件执行 + 掩码同步	分支开销降低
数据广播	shfl.idx + 掩码	带宽利用率提升

特殊寄存器：%lanemask_gt

✓错误处理与验证

```
// 掩码有效性验证
popc.b32   %pop_count, %lanemask_gt;
add.u32    %expected, 31, -%laneid;  // 31 - laneid
cmp.ne.u32 %p_err, %pop_count, %expected;
@%p_err    trap;  // 触发硬件异常
```

特殊寄存器：%lanemask_gt

✓实战示例

✓Warp 级严格后缀归约

```
// 并行最小值严格后缀归约（排除当前线程）
mov.u32 %min_val, MAX_INT;
.reduction:
    shfl.sync.down.b32 %other_val, %min_val, 1, %lanemask_gt;
    min.u32 %min_val, %min_val, %other_val;
    setp.gt.u32 %p_continue, %step, 0
    @%p_continue bra .reduction
```

特殊寄存器：%lanemask_gt

✓实战示例

✓动态负载均衡（严格逆序）

```
// 按线程 ID 严格逆序处理任务（排除自身）
mov.u32 %tasks_done, %laneid + 1;
.loop:
    setp.le.u32 %p_work, %tasks_done, 31
    @%p_work    process_task(%tasks_done)
    add.u32 %tasks_done, %tasks_done, 1
    setp.le.u32 %p_more, %tasks_done, 31
    @%p_more    bra .loop
```

特殊寄存器：%lanemask_gt

- ✓总结
  - ✓%lanemask_gt 是 GPU 细粒度并行控制的 严格后缀掩码工具
  - ✓价值：
    - ✓精确逆序控制：实现 Warp 内线程的严格逆序协作
    - ✓后缀操作加速：优化需要排除当前线程的后缀算法
    - ✓资源效率优化：按需激活高 ID 线程减少资源争用
    - ✓算法加速器：为严格后缀操作提供硬件级加速
  - ✓合理使用 %lanemask_gt 可使严格后缀操作类算法加速

特殊寄存器：%clock, %clock_hi

- ✓本质与功能
  - ✓%clock 和 %clock_hi 是 PTX 中用于高精度计时的协同寄存器组，共同组成 64 位时钟计数器：
    - ✓%clock：存储低 32 位时钟周期计数
    - ✓%clock_hi：存储高 32 位时钟周期计数
  - ✓时钟源：SM 核心时钟（非全局墙钟），频率为 GPU Boost 频率
  - ✓数学关系：

$$\text{完整时钟周期} = (\%clock\_hi \ll 32) + \%clock$$

讲授内容

- ➢特殊寄存器
  - ✓概述
  - ✓线程定位
  - ✓设备控制（Warp、CTA、grid控制）
  - ✓资源查询
  - ✓性能分析
- ✓提示符（Directives）
- ✓PTX编程总结
- ✓大作业一加分题（PTX编程）讲解

特殊寄存器：%total_smem_size

- ✓本质与功能
  - ✓%total_smem_size 是 PTX ISA 中的关键特殊寄存器，提供共享内存（Shared Memory）的总容量信息：
    - ✓只读性：硬件自动注入，软件不可修改
    - ✓动态性：值随 GPU 架构和配置变化
  - ✓核心功能：
    - ✓查询当前 SM 的共享内存物理容量
    - ✓指导共享内存分配策略
    - ✓实现架构无关的内存管理
  - ✓数学定义：

$$\%total\_smem\_size = \text{物理共享内存字节数}$$

特殊寄存器：%total_smem_size

- ✓硬件实现机制
  - ✓存储位置：SM 的 Configuration Register File
  - ✓访问路径：
    - ✓共享内存控制器直连
    - ✓访问延迟：极低（Ampere 架构）
  - ✓动态更新：GPU 启动时固化，运行时不变

特殊寄存器：%total_smem_size

- ✓核心应用场景
  - ✓共享内存预分配验证

```
// 检查共享内存需求是否超标
mov.u32 %req_size, 8192;  // 8KB 需求
cmp.gt.u32 %p_overflow, %req_size, %total_smem_size;
@%p_overflow exit_program;  // 超出硬件容量
```

特殊寄存器：%total_smem_size

✓核心应用场景
  ✓动态内存布局

```
// 基于容量优化数据结构
mov.u32 %smem_size, %total_smem_size;
cvt.f32.u32 %util_factor, %smem_size / MAX_SMEM;
adjust_data_layout(%util_factor);  // 动态调整
```

特殊寄存器：%total_smem_size

✓核心应用场景
  ✓多架构兼容

```
// 跨架构共享内存分配
#if __CUDA_ARCH__ >= 800
    %block_size = 128;  // Ampere优化块大小
#else
    %block_size = %total_smem_size / 32;  // 通用策略
#endif
```

特殊寄存器：%total_smem_size

✓性能优化策略
  ✓容量感知分配

```
// 最大化利用共享内存
mov.u32 %total, %total_smem_size;
%usable_size = %total - RESERVED_FOR_SYSTEM;
cudaFuncSetSharedMemSize(kernel, %usable_size);
```

特殊寄存器：%total_smem_size

✓性能优化策略
  ✓Bank 冲突规避

```
// 基于容量优化访问模式
%banks = %total_smem_size / BANK_SIZE;  // Bank 数量
adjust_access_stride(%banks);  // 避免 Bank 冲突
```

特殊寄存器：%total_smem_size

✓错误处理

```
// 容量有效性验证
setp.eq.u32 %p_invalid, %total_smem_size, 0;
@%p_invalid trap;  // 零容量异常

setp.gt.u32 %p_valid, %total_smem_size, MIN_SMEM_SIZE;
@!%p_valid exit_program;  // 容量不足
```

特殊寄存器：%total_smem_size

✓与相关寄存器协同

寄存器	协同关系	联合应用场景
%dynamic_smem_size	动态分配容量	内存池管理
%warpid	Warp 级分配	细粒度共享内存分区
%ctaid	线程块定位	跨块内存协调

特殊寄存器：%total_smem_size

✓实例
  ✓矩阵乘法优化

```
// 基于共享内存容量的分块策略
mov.u32 %smem_capacity, %total_smem_size;
%tile_size = sqrt(%smem_capacity / (2 * sizeof(float)));
adjust_tile_size(%tile_size);  // 优化分块
```

特殊寄存器：%total_smem_size

✓实例
  ✓动态内存池

```
// 创建共享内存池
extern __shared__ char pool[];
mov.u32 %pool_size, %total_smem_size - SYSTEM_RESERVE;

// 子分配器实现
void* allocate(size_t size) {
    atomicAdd(&pool_offset, size);
    return pool + (pool_offset - size);
}
```

特殊寄存器：%total_smem_size

- ✓总结
  - ✓%total_smem_size 是 GPU 并行编程的 共享内存资源锚点
  - ✓价值：
    - ✓硬件资源探针：精确反映物理内存容量
    - ✓性能优化器：指导共享内存分配策略
    - ✓架构兼容层：统一不同代际 GPU 的编程模型
    - ✓并行效率提升：最大化内存带宽利用率

特殊寄存器：%dynamic_smem_size

- ✓功能
  - ✓%dynamic_smem_size 是 PTX ISA 中的关键特殊寄存器，提供动态分配的共享内存大小信息：
    - ✓只读性：由 GPU 运行时自动注入，软件不可修改
    - ✓动态性：值随内核启动配置变化
  - ✓核心功能：
    - ✓查询当前线程块（CTA）动态分配的共享内存字节数
    - ✓指导共享内存的动态布局
    - ✓实现运行时内存分配验证
  - ✓数学关系：

$$\%dynamic\_smem\_size = \text{动态共享内存字节数}$$

特殊寄存器：%dynamic_smem_size

- ✓硬件实现机制
  - ✓存储位置：CTA 上下文寄存器（每个线程块独立）
  - ✓访问路径：
    - ✓SM 的 Shared Memory Controller
    - ✓访问延迟：极低（Ampere 架构）
  - ✓更新时机：CTA 启动时固化，生命周期内不变

特殊寄存器：%dynamic_smem_size

- ✓核心应用场景
  - ✓动态内存布局

```
// 基于动态大小创建数据结构
.extern .shared .align 4 .b8 dynamic_smem[];
mov.u32 %dyn_size, %dynamic_smem_size;
%float_array = dynamic_smem;
%num_elements = %dyn_size / 4;  // float类型大小
```

特殊寄存器：%dynamic_smem_size

✓核心应用场景
  ✓内存安全访问

```
// 防止动态内存越界
mov.u32 %dyn_size, %dynamic_smem_size;
setp.le.u32 %p_safe, %index, %dyn_size;
@%p_safe ld.shared.f32 %data, [%addr + %index];
@!%p_safe trap;  // 越界处理
```

特殊寄存器：%dynamic_smem_size

✓核心应用场景
  ✓混合内存管理

```
// 结合静态和动态共享内存
.shared .f32 static_buffer[512];  // 静态分配
.extern .shared .f32 dynamic_buffer[];  // 动态部分

// 计算动态部分起始地址
mov.u32 %dyn_start, sizeof(static_buffer);
add.u32 %dyn_addr, %dyn_start, %index;
```

特殊寄存器：%dynamic_smem_size

✓与相关寄存器关系

寄存器	协同关系	联合应用场景
%total_smem_size	物理总容量	验证分配有效性
%ctaid	线程块标识	跨CTA内存协调
%warpid	Warp标识	细粒度内存分区

特殊寄存器：%dynamic_smem_size

✓关键使用约束

✓分配上限

```
// 验证动态分配未超限
mov.u32 %total, %total_smem_size;
setp.le.u32 %p_valid, %dynamic_smem_size, %total;
@!%p_valid trap;  // 分配无效
```

特殊寄存器：%dynamic_smem_size

✓关键使用约束
  ✓内核启动依赖

```
// 主机端配置动态共享内存
kernel<<<grid, block, dynamic_smem_size>>>(...);
```

特殊寄存器：%dynamic_smem_size

✓性能优化策略
  ✓内存对齐优化

```
// 确保动态内存对齐
mov.u32 %dyn_size, %dynamic_smem_size;
and.b32 %misalign, %dyn_size, 0x1F;  // 32字节对齐检查
@%misalign != 0 add.u32 %dyn_size, %dyn_size, 32 - %misalign;
```

特殊寄存器：%dynamic_smem_size

✓性能优化策略

✓子分配器设计

```
// 动态内存池管理
.shared .b8 pool[];
mov.u32 %pool_size, %dynamic_smem_size;
%offset = 0;

allocate:
    atomic.add.u32 %new_offset, %offset, %req_size;
    setp.le.u32 %p_ok, %new_offset, %pool_size;
    @%p_ok ret pool + %offset;
    @!%p_ok ret 0;  // 分配失败
```

特殊寄存器：%dynamic_smem_size

✓错误处理

```
// 动态内存有效性检查
setp.eq.u32 %p_uninit, %dynamic_smem_size, 0;
@%p_uninit trap;  // 未初始化异常

setp.gt.u32 %p_valid, %dynamic_smem_size, MIN_DYN_SMEM;
@!%p_valid exit_program;  // 容量不足
```

特殊寄存器：%dynamic_smem_size

✓实例
  ✓动态数据结构

```
// 创建动态大小的矩阵
mov.u32 %dyn_size, %dynamic_smem_size;
%cols = 128;
%rows = %dyn_size / (cols * sizeof(float));
// 使用动态矩阵
```

特殊寄存器：%dynamic_smem_size

✓实例
  ✓自适应算法

```
// 根据内存大小选择算法
mov.u32 %dyn_size, %dynamic_smem_size;
if (%dyn_size > 8192) {
    use_memory_intensive_algorithm();
} else {
    use_memory_efficient_algorithm();
}
```

特殊寄存器：%dynamic_smem_size

✓与静态内存协同

```
// 混合内存布局示例
.shared .f32 static_part[1024];  // 静态分配1KB
.extern .shared .f32 dynamic_part[];  // 动态部分

// 计算动态部分偏移
%static_end = sizeof(static_part);
%dynamic_addr = dynamic_part + %static_end;
```

特殊寄存器：%dynamic_smem_size

✓总结

- ✓%dynamic_smem_size 是 GPU 编程的 动态内存管理核心
- ✓价值：
  - ✓运行时灵活性：支持内核启动时动态配置内存
  - ✓资源优化器：最大化共享内存利用率
  - ✓安全卫士：防止内存越界访问
  - ✓架构统一层：跨代际 GPU 的兼容接口
  - ✓合理使用动态共享内存可使内核性能提升
  - ✓减少共享内存浪费

讲授内容

- ➢特殊寄存器
  - ✓概述
  - ✓线程定位
  - ✓设备控制（Warp、CTA、grid控制）
  - ✓资源查询
  - ✓性能分析
- ✓提示符（Directives）
- ✓PTX编程总结
- ✓大作业一加分题（PTX编程）讲解

特殊寄存器：%clock, %clock_hi

- ✓硬件实现机制
  - ✓计数器架构
    - ✓更新频率：每 SM 时钟周期递增 1
    - ✓存储位置：SM 的 Core Clock Register File
    - ✓原子性：读取时自动锁定计数器，确保 %clock/%clock_hi 值一致

特殊寄存器：%clock, %clock_hi

✓核心应用场景
  ✓微基准测试（指令级计时）

```
// 测量单指令延迟
mov.u64    %t0, %clock64;    // 读取完整时钟
add.s32    %dummy, %a, %b;   // 目标指令
mov.u64    %t1, %clock64;
sub.u64    %latency, %t1, %t0;
```

特殊寄存器：%clock, %clock_hi

✓核心应用场景
  ✓性能热点分析

```
// 定位计算瓶颈
%start = %clock64;
@%p_hot_region {
    // 热点代码
    heavy_computation();
}
%end = %clock64;
%cycles = %end - %start;
```

特殊寄存器：%clock, %clock_hi

- ✓核心应用场景
  - ✓动态负载均衡

```
// 基于执行时间调整任务量
%t0 = %clock64;
process_chunk();
%t1 = %clock64;
%chunk_time = %t1 - %t0;
adjust_work_size(%chunk_time); // 动态优化
```

特殊寄存器：%clock, %clock_hi

- ✓关键使用约束
  - ✓线程局部性
    - ✓SM 绑定：各 SM 独立计数器（不可跨 SM 比较）
    - ✓Warp 同步：同 Warp 内线程读取值相同（原子广播）
  - ✓功耗影响
    - ✓时钟门控：未使用时不耗电
    - ✓读取代价：每次访问消极低

特殊寄存器：%clock, %clock_hi

✓性能优化策略

✓64 位读取最佳实践

```
// 原子读取 64 位时钟（防撕裂）
.macro read_clock64(dst)
    mov.u32    %hi1, %clock_hi;
    mov.u32    %lo,  %clock;
    mov.u32    %hi2, %clock_hi;
    setp.ne.u32 %p, %hi1, %hi2;  // 检查高字撕裂
    @%p        bra.retry;        // 撕裂则重试
    shl.u64    %dst, %hi1, 32;
    add.u64    %dst, %dst, %lo;
.endm
```

特殊寄存器：%clock, %clock_hi

✓性能优化策略

✓避免测量干扰

✓隔离机制：禁用测量期间的线程调度

```
// 暂停线程调度
.param .b32 __tmp;
asm volatile ("pause;") :: "r"(__tmp));
```

特殊寄存器：%clock, %clock_hi

✓与相关指令协同

指令/寄存器	协同机制	应用场景
%globaltimer	全局同步时钟	跨 SM 时间关联
membar	内存屏障	确保计时与内存操作顺序
nanosleep	精确延时	控制循环频率

特殊寄存器：%clock, %clock_hi

✓错误处理

```
// 检测时钟倒退（硬件异常）
mov.u64    %t0, %clock64;
mov.u64    %t1, %clock64;
setp.lt.u64 %p_err, %t1, %t0;  // 时间不应倒退
@%p_err     trap;
```

特殊寄存器：%clock, %clock_hi

- ✓总结
  - ✓%clock 和 %clock_hi 是 GPU 微架构级性能分析的 原子计时基元。
  - ✓功能：
    - ✓精度锚点：提ns 级别分辨率
    - ✓并行追踪：支持 多线程并发计时
    - ✓优化镜：量化指令/内存延迟
    - ✓跨代兼容：统一 Volta 到 Ampere 的计时模型
  - ✓合理利用时钟寄存器可使性能分析效率提升

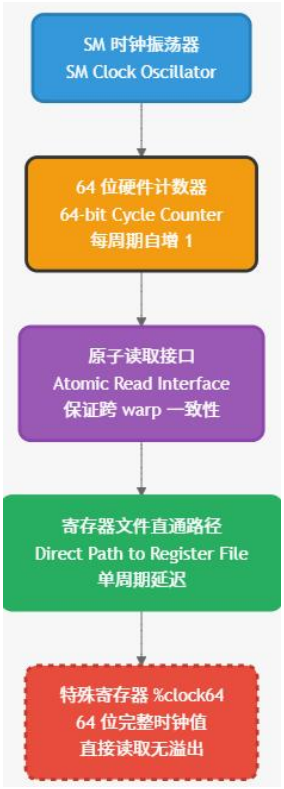
特殊寄存器：%clock64

- ✓本质与功能
  - ✓%clock64 是 PTX ISA 中的 64 位时钟周期计数器，提供高精度计时能力：
    - ✓原子访问：单指令读取完整 64 位值（无需 %clock + %clock_hi 组合）
    - ✓时钟源：SM 核心时钟（非全局墙钟），频率 = GPU Boost 频率
    - ✓精度：ns级别，比 32 位 %clock 精度高
  - ✓数学定义：

$$\%clock64 = \text{自 SM 启动以来的时钟周期数}$$

特殊寄存器：%clock64

- ✓硬件实现机制
  - ✓计数器架构：
    - ✓真 64 位硬件计数器（Ampere+）
    - ✓48 位模拟 64 位（Pascal/Volta）
  - ✓访问路径：
    - ✓SM 专用时钟总线
    - ✓零延迟（Ampere 架构）
  - ✓原子性保障：硬件锁存机制确保读取完整性



特殊寄存器：%clock64

- ✓核心应用场景
  - ✓微基准测试（纳秒级精度）

```
// 测量内存加载延迟
mov.u64    %t0, %clock64;
ld.global.f32 %data, [%addr];
mov.u64    %t1, %clock64;
sub.u64    %latency, %t1, %t0;  // 单位：时钟周期
```

特殊寄存器：%clock64

✓核心应用场景

✓性能热点分析

```
// 精确定位计算瓶颈
%start = %clock64;
@%p_hot_region {
    // 热点代码段
    compute_intensive_kernel();
}
%end = %clock64;
%cycles = %end - %start;  // 精确周期计数
```

特殊寄存器：%clock64

✓核心应用场景

✓实时系统控制

```
// 硬实时控制循环
.loop:
    %t0 = %clock64;
    process_frame();
    %t1 = %clock64;
    %elapsed = %t1 - %t0;
    %sleep_cycles = TARGET_CYCLES - %elapsed;
    nanosleep(%sleep_cycles);  // 精确休眠
    bra .loop;
```

特殊寄存器：%clock64

- ✓关键使用约束
  - ✓线程局部性
    - ✓SM 绑定：各 SM 独立计数器（不可跨 SM 比较）
    - ✓Warp 一致性：同 Warp 内线程读取值相同（硬件广播）

特殊寄存器：%clock64

- ✓性能优化策略
  - ✓精确延时控制

```
// 纳秒级精确等待
mov.u64    %target, %clock64 + DELAY_CYCLES;
.spin:
    %current = %clock64;
    setp.lt.u64 %p, %current, %target;
    @%p bra .spin;
```

特殊寄存器：%clock64

✓性能优化策略
  ✓避免测量干扰

```
// 隔离测量环境
asm volatile ("prefetch.L1 [%0];" :: "r"(&dummy));  // 填充流水线
asm volatile ("bar.sync 0;");  // 同步所有线程
%t0 = %clock64;
// 目标代码
%t1 = %clock64;
```

特殊寄存器：%clock64

✓与相关指令协同

指令/寄存器	协同机制	应用场景
%globaltimer	全局同步时钟	跨设备时间关联
membar.gl	全局内存屏障	确保计时与内存操作顺序
nanosleep	周期级休眠	硬实时控制

特殊寄存器：%clock64

✓错误处理

```
// 检测时钟异常（硬件级验证）
mov.u64    %t0, %clock64;
mov.u64    %t1, %clock64;
setp.lt.u64 %p_err, %t1, %t0;  // 时间不应倒流
@%p_err     trap;  // 触发硬件异常
```

特殊寄存器：%clock64

✓总结

- ✓%clock64 是 GPU 性能分析的 原子计时基石，
- ✓价值：
  - ✓纳秒级精度：提供 ns 分辨率
  - ✓免撕裂访问：64 位原子读取消除组合风险
  - ✓实时控制核：赋能硬实时系统（自动驾驶/工业控制）
  - ✓跨代兼容：统一 Volta 到 Ampere 的计时模型
- ✓使用 %clock64 可使性能分析效率提升

特殊寄存器：%pm0..%pm7

- ✓本质与功能
  - ✓%pm0 到 %pm7 是 PTX 架构中的 性能监控寄存器（Performance Monitoring Registers），提供硬件级性能数据采集能力：
    - ✓8 个专用寄存器：%pm0 到 %pm7 独立计数不同性能事件
    - ✓只读性：由 GPU 硬件自动更新，软件不可修改
    - ✓事件驱动：每个寄存器绑定特定硬件事件（如缓存命中/分支预测）
  - ✓数学定义：

$$\%pmX = \sum_{i=0}^{N} \delta(event_X) \quad (\text{事件X的发生次数})$$

特殊寄存器：%pm0..%pm7

- ✓硬件实现机制
  - ✓计数器架构：
    - ✓32 位硬件计数器（Pascal+）
    - ✓专用事件总线直连
  - ✓访问路径：
    - ✓SM 的 Performance Monitor Unit (PMU)
    - ✓访问延迟：极低
  - ✓原子性：读取时暂停计数确保值一致性

特殊寄存器：%pm0..%pm7

✓核心应用场景
  ✓微架构性能分析

```
// 测量 L2 缓存命中率
mov.u32    %start, %pm0;  // L2 访问计数
ld.global.f32 %data, [%addr];
mov.u32    %end, %pm0;
sub.u32    %accesses, %end, %start;
mov.u32    %hits, %pm1;   // L2 命中计数
cvt.f32.u32 %hit_rate, %hits / %accesses;
```

特殊寄存器：%pm0..%pm7

✓核心应用场景
  ✓瓶颈诊断

```
// 检测指令发射停顿
mov.u32    %stall_start, %pm5;  // 发射停顿周期
compute_kernel();
mov.u32    %stall_end, %pm5;
sub.u32    %stall_cycles, %stall_end, %stall_start;
```

特殊寄存器：%pm0..%pm7

✓核心应用场景

✓动态优化

```
// 基于分支误预测调整策略
mov.u32    %miss_start, %pm3;  // 分支误预测
@%p_cond   bra TARGET;
mov.u32    %miss_end, %pm3;
%miss_count = %miss_end - %miss_start;
@%miss_count > THRESHOLD
adjust_branch_strategy();
```

特殊寄存器：%pm0..%pm7

✓性能事件映射

寄存器	典型事件 (Ampere)	测量意义
%pm0	L2 cache accesses	二级缓存访问量
%pm1	L2 cache hits	缓存命中率基础
%pm2	FP32 operations	单精度计算吞吐
%pm3	Branch mispredictions	分支预测效率
%pm4	DRAM reads	显存带宽压力
%pm5	Instruction issue stalls	指令调度效率
%pm6	Shared memory conflicts	存储体冲突
%pm7	Atomic operations	原子操作竞争强度

特殊寄存器：%pm0..%pm7

- ✓关键使用约束
  - ✓特权级别要求
    - ✓用户模式：仅部分事件可访问（需驱动授权）
    - ✓内核模式：完整访问权限（需 CUDA 特权 API）

特殊寄存器：%pm0..%pm7

- ✓关键使用约束
  - ✓计数器复用

```
// 事件重配置（需特权）
call cudaConfigPMEvent(EVENT_L2_ACCESS, %pm0);  // 绑定新事件
```

特殊寄存器：%pm0..%pm7

✓性能优化策略
  ✓多寄存器协同分析

```
// 计算指令效率
%fp_ops = %pm2;      // FP32 操作数
%stalls = %pm5;      // 停顿周期
%active_cycles = %clock64 - %stalls;
%ips = %fp_ops / %active_cycles;  // 指令/周期
```

特殊寄存器：%pm0..%pm7

✓性能优化策略
  ✓实时反馈控制

```
// 动态调整计算策略
%conflicts = %pm6;  // 共享内存冲突
if (%conflicts > THRESHOLD) {
    adjust_access_pattern();  // 优化存储体访问
}
```

特殊寄存器：%pm0..%pm7

✓与相关寄存器协同

寄存器	协同机制	联合应用场景
%clock64	时间基准	计算事件发生率
%smid	SM 定位	跨 SM 性能对比
%warpid	Warp 级分析	细粒度性能诊断

特殊寄存器：%pm0..%pm7

✓错误处理

```
// 检测计数器异常
mov.u32    %t0, %pm0;
mov.u32    %t1, %pm0;
setp.gt.u32 %p_err, %t1, %t0 + MAX_DELTA;  // 合理增量
@%p_err     trap;  // 触发硬件异常
```

特殊寄存器：%pm0..%pm7

✓实例

✓缓存优化

```
// 优化 L1 缓存访问模式
%base_access = %pm0;  // L1 访问
optimize_data_layout();
%new_access = %pm0;
%reduction = 1.0 - (%new_access / %base_access);
```

特殊寄存器：%pm0..%pm7

✓实例

✓分支预测优化

```
// 减少分支误预测
%start_miss = %pm3;
// 重构分支逻辑
unroll_loops();
vectorize_branches();
%end_miss = %pm3;
%improvement = %start_miss - %end_miss;
```

特殊寄存器：%pm0..%pm7

- ✓总结
  - ✓%pm0 到 %pm7 是 GPU 微架构性能分析的 硬件级探针
  - ✓价值：
    - ✓硅级洞察：提供晶体管级性能数据
    - ✓精准诊断：定位纳秒级性能瓶颈
    - ✓动态优化：赋能实时自适应算法
    - ✓架构镜：反映硬件实际行为
  - ✓合理利用性能监控寄存器可使优化效率提升

讲授内容

- ➢特殊寄存器
  - ✓概述
  - ✓线程定位
  - ✓设备控制（Warp、CTA、grid控制）
  - ✓资源查询
  - ✓性能分析
- ✓提示符（Directives）
- ✓PTX编程总结
- ✓大作业一加分题（PTX编程）讲解

提示符（Directives）

- ✓核心功能
  - ✓提示符（Directives） 是 PTX 汇编语言中的元编程元素，用于指导编译器和汇编器的行为：
    - ✓非执行性：不生成机器指令，只影响编译过程
    - ✓预处理性：在编译前期处理
  - ✓核心功能：
    - ✓控制代码生成
    - ✓定义符号属性
    - ✓管理内存布局
    - ✓优化指令选择

提示符（Directives）

✓分类与功能矩阵

指令类型	关键提示符（Directives）	功能描述
模块控制	.version, .target	定义 PTX ISA 版本和目标架构
符号声明	.global, .extern	管理符号可见性
状态空间	.const, .shared	定义内存区域属性
变量属性	.align, .attribute	控制内存对齐和变量特性
调试信息	.file, .loc	嵌入源代码级调试信息
性能调优	.maxnreg, .minnctapersm	优化寄存器/线程块使用

提示符（Directives）

- ✓核心指令详解
  - ✓模块控制指令
    - ✓.version：启用特定 ISA 特性（如 sm_80 支持 tensor core）
    - ✓.target：控制指令选择和优化策略
    - ✓.address_size：影响指针大小和内存操作

```
.version 7.0        // 指定 PTX ISA 版本
.target sm_80       // 目标架构 Ampere
.address_size 64    // 使用 64 位地址空间
```

提示符（Directives）

- ✓核心指令详解
  - ✓状态空间指令
    - ✓对齐优化：.align N 确保 N 字节对齐（N=2^k）
    - ✓类型规范：.b8, .u32 等定义数据格式

```
.const .align 16 .b8 global_data[1024]; // 常量内存
.shared .u32 smem_buffer[256];          // 共享内存
.local .f32 lmem_array[128];            // 本地内存
```

提示符（Directives）

- ✓核心指令详解
  - ✓变量属性指令
    - ✓.managed：启用 CUDA 统一内存
    - ✓.weak：允许符号未定义链接

```
.global .attribute(.managed) .f32 gpu_array[]; // 统一内存
.weak .func device_function;                   // 弱符号
```

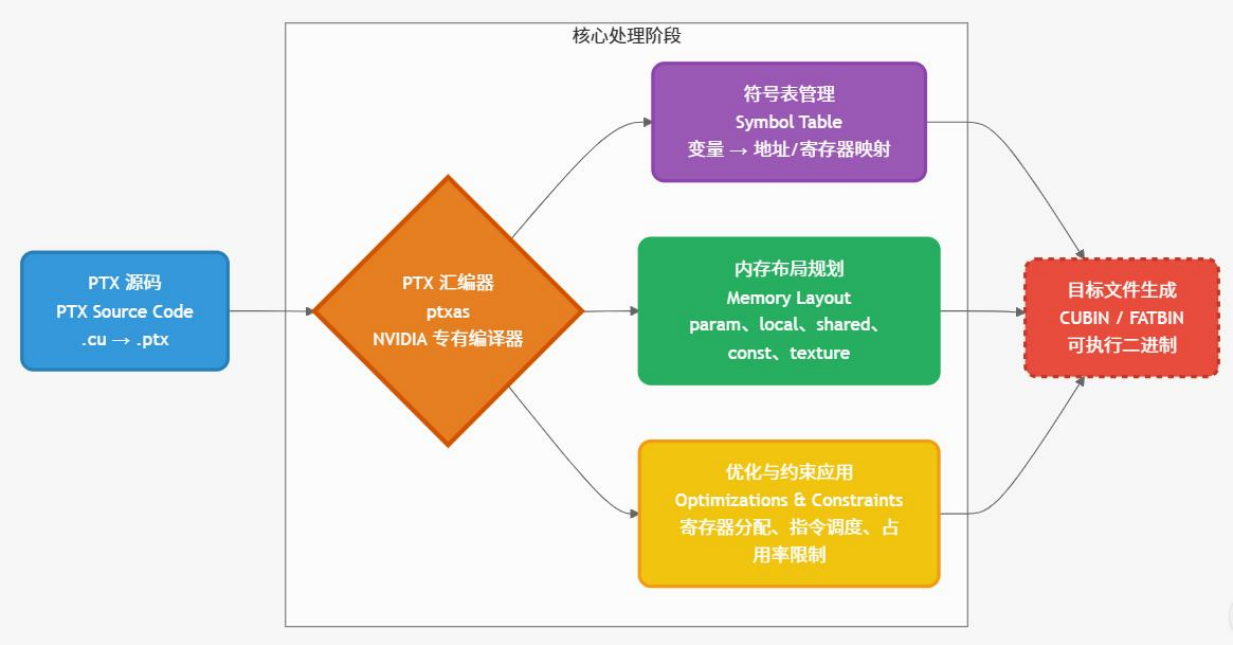
提示符（Directives）

- ✓核心指令详解
  - ✓性能调优指令
    - ✓.maxnreg：限制寄存器使用（减少溢出）
    - ✓.minnctapersm：确保最小线程块/SM（提高占用率）

```
.entry .maxnreg 32 .minnctapersm 4 kernel(...)
```

提示符（Directives）

✓编译处理流程



提示符（Directives）

✓关键应用场景

✓跨架构兼容

```
#if __CUDA_ARCH__ >= 800
    .target sm_80
    .pragma "tensorcore_enabled", 1
#else
    .target sm_70
#endif
```

提示符（Directives）

- ✓关键应用场景
  - ✓安全内存布局

```
.const .align 8 .b8 sensitive_data[256] = { ... };
.attribute(.protection) "checksum=0xA5F3";
```

提示符（Directives）

- ✓关键应用场景
  - ✓混合精度优化

```
.func .attribute(.fp16_enabled) half_precision(...)
```

提示符（Directives）

✓错误处理

```
// 指令冲突检测
.if .target != sm_80
    .error "Requires Ampere architecture"
.endif

// 属性兼容性验证
.if .attribute(.managed) && .space != .global
    .warning "Managed only valid in global space"
.endif
```

提示符（Directives）

✓总结

- ✓PTX 指令系统是 GPU 编程的元控制层，其战略价值在于：
  - ✓架构抽象层：统一不同代际 GPU 的编程接口
  - ✓性能调节阀：精细控制编译优化策略
  - ✓内存设计器：规划物理内存布局
  - ✓跨平台桥梁：实现 CPU-GPU 无缝协作
- ✓最佳实践指南：
  - ✓合理使用指令可使性能提升
  - ✓减少 编译警告
  - ✓提升 代码可移植性

讲授内容

- ➢特殊寄存器
  - ✓概述
  - ✓线程定位
  - ✓设备控制（Warp、CTA、grid控制）
  - ✓资源查询
  - ✓性能分析
- ✓提示符（Directives）
- ✓PTX编程总结
- ✓大作业一加分题（PTX编程）讲解

编程模型（Programming Model）

- ✓核心架构
  - ✓GPU 作为 CPU（主机）的协处理器，承接数据并行、计算密集型任务，通过内核函数（kernel）执行，内核被映射为大量并行线程。

编程模型（Programming Model）

- ✓线程层次结构
  - ✓线程束（Warp）：CTA 内的最小执行单元，默认 32 个线程，按 SIMT（单指令多线程）模式执行，同一线程束执行相同指令，分支时串行执行各路径。
  - ✓协作线程数组（CTA）：即 CUDA 线程块，包含多个线程束，线程间可通过共享内存通信与同步，每个线程有唯一标识 %tid（三维向量：tid.x/tid.y/tid.z）。
  - ✓网格（Grid）：由多个 CTA 组成，CTA 间无直接通信与同步，每个 CTA 有唯一标识 %ctaid，网格有唯一标识 %gridid。

编程模型（Programming Model）

- ✓内存层次结构
  - ✓线程私有内存（Local Memory）：每个线程独立访问，用于存储私有数据。
  - ✓块共享内存（Shared Memory）：CTA 内线程共享，生命周期与 CTA 一致，支持低延迟通信。
  - ✓全局内存（Global Memory）：所有线程可访问，生命周期跨内核调用，是设备内存的主要部分。
  - ✓常量内存（Constant Memory）：只读，64KB静态常量区+10个64KB独立区域，经缓存优化，适合频繁访问的数据。
  - ✓纹理/表面内存（Texture/Surface Memory）：只读（纹理）或可读写（表面），带缓存与特殊寻址模式（如归一化坐标、滤波），不与全局内存写操作保持一致性。

PTX 机器模型（PTX Machine Model）

- ✓核心硬件抽象
  - ✓基于 SIMT 多处理器（SM）阵列：每个 SM 包含多个标量处理器（SP）、线程指令单元、片上共享内存。
  - ✓SM 工作机制：接收并拆分 CTA 为线程束，通过零开销线程调度管理数百个并发线程，支持屏障同步。
  - ✓内存缓存：SM 包含共享内存（与块共享内存对应）、常量缓存、纹理缓存，全局/本地内存无缓存。

PTX 机器模型（PTX Machine Model）

- ✓关键特性
  - ✓线程束（warp）调度：线程束（warp）按就绪状态被调度执行，当一个线程束（warp）等待内存操作时，SM切换至其他就绪线程束（warp），隐藏内存延迟。
  - ✓资源分配：SM的寄存器与共享内存按CTA需求拆分，若内核所需寄存器/共享内存超出SM容量，内核无法启动。

语法（Syntax）

- ✓源格式与预处理
  - ✓源文件为 ASCII 文本，支持 C 预处理器指令（#include/#define 等），区分大小写，关键字小写。
  - ✓必须以 .version（指定 PTX 版本）和 .target（指定目标架构）指令开头。
- ✓语句与注释
  - ✓语句类型：指令语句（操作码 + 操作数）和指令语句（以 . 开头，如 .reg 声明寄存器），均以分号结束。
  - ✓注释：支持 /* … */（多行）和 //（单行），被视为空白字符。

语法（Syntax）

- ✓标识符与常量
  - ✓标识符规则：字母开头（后跟字母/数字/下划线//%开头（后跟字母/数字/下划线/$），预定义标识符以 % 开头（如 %tid）。
  - ✓常量类型：整数常量（64 位，.s64/.u64）、浮点常量（64 位双精度，支持 32 位单精度十六进制表示）、谓词常量（0 为假，非 0 为真）。
  - ✓常量表达式：支持算术、逻辑、比较运算，遵循 C 运算符优先级，无整数与浮点间的隐式转换。

状态空间、类型与变量（State Spaces, Types, and Variables）

✓状态空间（State Spaces）

状态空间	可寻址性	初始化	访问权限	共享范围	关键特性
.reg	否	否	读写	线程内	快速寄存器，溢出时 spilling 到内存
.sreg	否	否	只读	CTA 内	特殊寄存器（如线程/CTA 标识）
.const	是	是（默认 0）	只读	网格内	分静态区与动态缓冲区，64KB/缓冲区
.global	是	是（默认 0）	读写	上下文内	全局可见，生命周期跨内核

状态空间、类型与变量（State Spaces, Types, and Variables）

✓状态空间（State Spaces）

状态空间	可寻址性	初始化	访问权限	共享范围	关键特性
.local	是	否	读写	线程内	线程私有内存，ABI 模式下分配在栈上
.param	是/受限	否	读/读写	网格内/线程内	内核参数（只读）、函数参数（读写）
.shared	是	否	读写	CTA 内	块共享，支持广播与顺序访问优化
.tex	否	是（驱动初始化）	只读	上下文内	已废弃，建议用 .texref 类型

语法（Syntax）

- ✓数据类型
  - ✓基础类型：有符号整数（.s8/.s16/.s32/.s64）、无符号整数（.u8/.u16/.u32/.u64）、浮点（.f16/.f16x2/.f32/.f64）、位类型（.b8/.b16/.b32/.b64）、谓词（.pred）。
  - ✓聚合类型：向量（.v2/.v4 前缀，如 .v4.f32）、数组（固定大小，支持初始化）。
  - ✓不透明类型：.texref（纹理引用）、.samplerref（采样器引用）、.surfref（表面引用），用于纹理/表面操作。

语法（Syntax）

- ✓变量声明与属性
  - ✓声明格式：状态空间 类型 变量名[数组大小] [= 初始化值]，如 .global .u32 arr[10] = {0,1,2};。
  - ✓对齐：通过 .align 指定，默认按类型大小对齐，向量按整体大小对齐。
  - ✓属性：支持 .managed（统一虚拟内存分配）等属性。

指令操作数（Instruction Operands）

- ✓操作数类型与兼容性
  - ✓源操作数：寄存器、常量表达式、地址（变量地址/寄存器地址+偏移/立即地址）、标签/函数名。
  - ✓目标操作数：寄存器（标量/向量），仅寄存器状态空间。
  - ✓兼容性规则：位类型与同大小任意类型兼容；同大小有符号/无符号整数兼容；浮点类型需完全匹配大小。

指令操作数（Instruction Operands）

- ✓地址与数组/向量操作
  - ✓地址格式：[变量]/[寄存器]/[寄存器+立即偏移]/[变量+立即偏移]/[立即地址]，需按访问大小自然对齐。
  - ✓数组访问：支持索引语法（如 arr[i]），复杂索引需提前计算地址。
  - ✓向量操作：支持元素提取（.x/.y/.z/.w 或 .r/.g/.b/.a）与打包/解包。

指令操作数（Instruction Operands）

- ✓类型转换
  - ✓标量转换：通过 cvt 指令，支持整数/浮点/位类型间转换，支持舍入修饰符（.rn/.rz/.rm/.rp）和饱和修饰符（.sat）。
  - ✓向量转换：按元素逐个转换，支持宽寄存器存储窄值时的零扩展/符号扩展。

ABI 抽象（Abstracting the ABI）

- ✓函数声明与定义
  - ✓声明格式：用 .func 指令，支持输入参数、返回参数，参数可位于 .reg（标量/向量）或 .param（结构体/大对象）状态空间。
  - ✓调用规则：参数类型/大小需匹配，.param 空间参数需通过 ld.param/st.param 访问，调用前需用 st.param 传递参数，调用后用 ld.param 获取返回值。

ABI 抽象（Abstracting the ABI）

- ✓变长函数与栈分配
  - ✓支持无大小数组参数（如 .param .b8 arr[]），用于实现变长参数函数。
  - ✓栈分配：通过 %alloca 函数在本地内存分配栈空间，默认 4 字节对齐，需手动调整对齐。

内存连续性模型（Memory Consistency Model）

- ✓适用范围
  - ✓适用于 PTX 任意版本，运行在 sm_70 及以上架构，不涵盖纹理/表面访问。

内存连续性模型（Memory Consistency Model）

- ✓核心概念
  - ✓内存操作：读（ld）、写（st）、原子读写修改（atom/red），向量操作被视为多个标量操作的集合。
  - ✓作用域（Scope）：.cta（CTA 内线程）、.gpu（同一 GPU 内线程）、.sys（主机+所有 GPU 线程）。
  - ✓操作类型：强操作（带 .relaxed/.acquire/.release/.acq_rel/.volatile 修饰）、弱操作（带 .weak 修饰）、栅栏（fence.sc/membar）。

内存连续性模型（Memory Consistency Model）

- ✓关键规则
  - ✓因果顺序（Causality Order）：通过同步操作（栅栏、屏障、释放-获取模式）建立跨线程的操作顺序。
  - ✓相干顺序（Coherence Order）：重叠写操作的可见顺序，避免数据竞争。
  - ✓原子性：强操作的单拷贝原子性，混合大小数据竞争的限制。
  - ✓无凭空值（No Thin Air）：禁止投机执行产生自满足的无效值。

指令集（Instruction Set）

- ✓整数算术指令
  - ✓基础运算：add（加）、sub（减）、mul（乘）、mad（乘加）、div（除）、rem（取余）。
  - ✓扩展运算：mul24/mad24（24 位乘/乘加）、popc（位计数）、clz（前导零计数）、bfind（最高置位bit查找）、fns（第n个置位bit查找）。
  - ✓位操作：bfe（位域提取）、bfi（位域插入）、brev（位反转）、dp4a/dp2a（点积累加）。

指令集（Instruction Set）

- ✓扩展精度整数指令
  - ✓带进位/借位运算：add.cc/addc（带进位加）、sub.cc/subc（带借位减）、mad.cc/madc（带进位乘加），依赖条件码寄存器（[CC.CF](CC.CF)）。

指令集（Instruction Set）

- ✓浮点指令
  - ✓基础运算：add/sub/mul/div/mad/fma（融合乘加），支持舍入（.rn/.rz/.rm/.rp）和刷新次正规数（.ftz）修饰。
  - ✓特殊运算：rcp（倒数）、sqrt（平方根）、rsqrt（平方根倒数）、sin/cos（正弦/余弦）、lg2/ex2（对数/指数）。
  - ✓半精度运算：支持 .f16/.f16x2 类型的 add/sub/mul/fma。

指令集（Instruction Set）

- ✓比较与选择指令
  - ✓比较：set（比较结果存整数）、setp（比较结果存谓词），支持有符号/无符号/浮点（含 NaN 处理）比较。
  - ✓选择：selp（按谓词选择）、slct（按符号选择）。

指令集（Instruction Set）

- ✓逻辑与移位指令
  - ✓逻辑运算：and/or/xor/not/cnot（逻辑非）、lop3（三输入任意逻辑）。
  - ✓移位运算：shl（左移）、shr（右移）、shf（漏斗移位）。

指令集（Instruction Set）

- ✓数据移动与转换指令
  - ✓移动：mov（寄存器/地址移动）、shfl/shfl.sync（线程束内数据洗牌）、prmt（字节置换）。
  - ✓加载/存储：ld（加载）、st（存储）、ldu（只读共享地址加载）、prefetch（预取），支持缓存操作修饰符（.ca/.cg/.cs 等）。
  - ✓转换：cvt（类型转换）、cvta（地址转换，通用地址与状态空间地址互转）。

指令集（Instruction Set）

- ✓纹理与表面指令
  - ✓纹理操作：tex（纹理查找）、tld4（4-texel 双线性采样）、txq（查询纹理/采样器属性）。
  - ✓表面操作：suld（表面加载）、sust（表面存储）、sured（表面归约）、suq（查询表面属性）。

指令集（Instruction Set）

- ✓控制流指令
  - ✓分支：bra（无条件/条件分支）、brx.idx（索引分支）、call（函数调用）、ret（返回）、exit（线程终止）。
  - ✓谓词执行：指令前加 @p（谓词 p 为真时执行）或 @!p（谓词 p 为假时执行）。

指令集（Instruction Set）

- ✓并行同步与通信指令
  - ✓屏障：bar/barrier（CTA 内屏障）、bar.warp.sync（线程束内屏障）。
  - ✓栅栏：membar（内存屏障）、fence（同步栅栏）。
  - ✓原子操作：atom（原子读写修改）、red（归约操作），支持 add/min/max/cas 等操作。
  - ✓投票：vote/vote.sync（线程束内谓词投票）、match.sync（线程束内值广播与比较）。

指令集（Instruction Set）

- ✓矩阵乘加指令（wmma）
  - ✓预览特性，支持 16x16x16 矩阵操作：wmma.load（加载矩阵）、wmma.mma（乘加运算）、wmma.store（存储矩阵），需线程束协作执行。

指令集（Instruction Set）

- ✓视频指令
  - ✓标量视频指令：vadd/vsub/vabsdiff（字节/半字/字运算）、vshl/vshr（移位）、vmad（乘加）。
  - ✓SIMD 视频指令：vadd2/vadd4、vsub2/vsub4 等，支持多元素并行运算。

特殊寄存器（Special Registers）

- ✓线程标识、CTA/网格标、硬件信息、计时/性能、lane 掩、等功能。

提示符（Directives）

- ✓模块提示符（Directives）
  - ✓.version：指定 PTX 版本（如 .version 6.0）。
  - ✓.target：指定目标架构（如 .target sm_70）、纹理模式（texmode_unified/texmode_independent）。
  - ✓.address_size：指定地址大小（32/64 位）。

提示符（Directives）

- ✓内核与函数提示符（Directives）
  - ✓.entry：声明内核入口函数（如 .entry kernel(...) { ... }）。
  - ✓.func：声明设备函数。
  - ✓.param：声明参数（内核/函数参数）。

提示符（Directives）

- ✓控制流与性能提示符（Directives）
  - ✓.branchtargets/.calltargets：声明分支/调用目标。
  - ✓.maxnreg：指定函数最大寄存器使用数。
  - ✓.maxntid/.reqntid：指定 CTA 最大/要求的线程数。
  - ✓.pragma：性能调优杂注（如 #pragma nounroll 禁止循环展开）。

提示符（Directives）

- ✓调试与链接提示符（Directives）
  - ✓调试：@@dwarf（DWARF 调试信息）、.section/.file/.loc（调试相关段/文件/位置信息）。
  - ✓链接：.extern（外部符号）、.visible（可见性）、.weak（弱符号）、.common（公共符号）。

讲授内容

- ➢特殊寄存器
  - ✓概述
  - ✓线程定位
  - ✓设备控制（Warp、CTA、grid控制）
  - ✓资源查询
  - ✓性能分析
- ✓提示符（Directives）
- ✓PTX编程总结
- ✓大作业一加分题（PTX编程）讲解

课程大作业一

✓内容：基于CUDA语言实现和PTX语言优化（加分题），基于LeNet，实现脉冲卷积神经网络（Spiking Convolutional Neural Network，SCNN）的推理部分，并在Fashion MNIST数据集合上进行分类。

✓调优。与英伟达library相比，你的实现达到了几成功力呢？

课程大作业一

- ✓截止日期：作业一、第10周上课前，截止日期之前可多次提交；
- ✓评分：国科大计算中心（八卡英伟达V100）；
- ✓算力：自选
- ✓可以使用豆包AI编程等工具。

课程大作业一

- ✓评分指标：准确率、运行时间。
- ✓课程大作业（一）：30%，后续发布加分题分数。
- ✓评分细节即将发布。
- ✓参考文献：https://spikingjelly.readthedocs.io/zh-cn/latest/activation_based/conv_fashion_mnist.html

课程大作业一讲解

- ✓前置依赖与宏定义
  - ✓宏定义：LeNet-SCNN网络参数（Fashion MNIST匹配）
    - ✓输入层参数：Fashion MNIST为单通道（1）、28×28像素图像
    - ✓卷积层1参数（LeNet经典配置）
    - ✓池化层1参数（Max Pooling，压缩特征图）
    - ✓卷积层2参数（LeNet经典配置）
    - ✓池化层2参数
    - ✓全连接层参数（LeNet经典配置，输出10类对应Fashion MNIST类别）

课程大作业一讲解

- ✓前置依赖与宏定义
  - ✓宏定义：LIF神经元参数（脉冲神经网络核心）
  - ✓宏定义：GPU线程配置（适配V100算力）

课程大作业一讲解

✓工具函数：数据加载与脉冲编码（SCNN输入预处理）

✓Fashion MNIST数据加载（主机端，适配数据集二进制格式）

```
/**
 * @brief 加载Fashion MNIST图像数据集（二进制格式）
 * @param path 数据集文件路径（如"t10k-images-idx3-ubyte"）
 * @param host_images 输出：主机端存储的图像数据（归一化到[0,1]）
 * @param num_samples 输出：加载的样本数量
 * @return 加载成功返回true，失败返回false
 */
bool load_fashion_mnist_images(const char* path, std::vector<float>& host_images, int&
num_samples) {
...
}
```

课程大作业一讲解

✓工具函数：数据加载与脉冲编码（SCNN输入预处理）

✓随机数初始化（设备端，用于泊松脉冲编码）

```
/**
 * @brief 初始化设备端随机数生成器（curand库，用于脉冲编码的泊松分布采样）
 * @param dev_curand_states 输出：设备端存储的随机数状态数组
 * @param num_states 需要初始化的状态数量（=样本数×总像素数×时间步）
 */
__global__ void init_curand_kernel(curandState_t* dev_curand_states, int num_states) {
    // 线程索引（每个线程对应一个随机数状态）
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= num_states) return;  // 避免线程越界

    // 初始化curand状态：seed=1234（固定种子便于复现），序列=idx（每个线程独立序列），偏移=0
    curand_init(1234, idx, 0, &dev_curand_states[idx]);
}
```

课程大作业一讲解

✓工具函数：数据加载与脉冲编码（SCNN输入预处理）

✓泊松脉冲编码（设备端，将静态图像转为脉冲序列）

```
/**
 * @brief 泊松脉冲编码核函数（将归一化图像转为SIM_STEPS步的脉冲序列）
 * @param dev_images 输入：设备端归一化图像数据（[0,1]）
 * @param dev_curand_states 输入：设备端随机数状态数组
 * @param dev_spike_seq 输出：设备端脉冲序列（1=发放脉冲，0=不发放）
 * @param batch_size 输入：批量处理的样本数
 * @param sim_steps 输入：脉冲编码时间步数（=SIM_STEPS）
 */
__global__ void poisson_encoding_kernel(
    const float* dev_images,
    curandState_t* dev_curand_states,
    float* dev_spike_seq,
    int batch_size,
    int sim_steps
) {
...
}
```

课程大作业一讲解

✓PTX核心模块（SCNN计算密集型部分）

✓PTX实现脉冲卷积（Conv1，单时间步，输入通道=1）

✓脉冲卷积的核心逻辑：卷积窗口内“输入脉冲×权重”的加权求和 + 膜电位累加。

✓通过PTX汇编直接操作GPU寄存器，减少CUDA编译器的冗余指令，优化内存访问模式（合并访问），提升并行效率。

课程大作业一讲解

✓PTX核心模块（SCNN计算密集型部分）

✓PTX实现脉冲卷积（Conv1，单时间步，输入通道=1）

```
/**
 * @brief PTX优化的脉冲卷积核（Conv1，处理单时间步的输入脉冲）
 * @param dev_spike_in 输入：单时间步的设备端输入脉冲序列（batch×1×28×28）
 * @param dev_conv1_w 输入：设备端Conv1权重（6×1×5×5，格式：out_c×in_c×k_h×k_w）
 * @param dev_conv1_mem 输入输出：设备端Conv1神经元膜电位（batch×6×24×24）
 * @param batch_size 输入：批量处理的样本数
 * @param t 输入：当前时间步（用于判断是否初始化膜电位）
 */
__global__ void ptx_spike_conv1_kernel(
    const float* dev_spike_in,
    const float* dev_conv1_w,
    float* dev_conv1_mem,
    int batch_size,
    int t
) {
...
}
```

课程大作业一讲解

✓PTX核心模块（SCNN计算密集型部分）

✓PTX实现LIF神经元脉冲发放与重置（Conv1输出）

✓LIF神经元核心逻辑：膜电位≥阈值→发放脉冲+重置膜电位；否则→膜电位保持衰减后的值。

✓通过PTX优化条件判断与内存写入，减少分支延迟，提升并行效率。

课程大作业一讲解

✓PTX核心模块（SCNN计算密集型部分）

✓PTX实现LIF神经元脉冲发放与重置（Conv1输出）

```
/**
 * @brief PTX优化的LIF神经元核（处理Conv1的膜电位，生成脉冲输出）
 * @param dev_conv1_mem 输入输出：设备端Conv1神经元膜电位（batch×6×24×24）
 * @param dev_conv1_spike 输出：设备端Conv1神经元脉冲输出（batch×6×24×24）
 * @param batch_size 输入：批量处理的样本数
 */
__global__ void ptx_lif_conv1_kernel(
    float* dev_conv1_mem,
    float* dev_conv1_spike,
    int batch_size
) {
...
}
```

课程大作业一讲解

✓PTX核心模块（SCNN计算密集型部分）

✓ 脉冲池化（Max Pooling，Pool1）

✓脉冲池化核心逻辑：池化窗口内取脉冲值最大的神经元输出（1若有任一神经元发放脉冲，0否则）。

```
/**
 * @brief 脉冲池化核（Pool1，Max Pooling，处理Conv1的脉冲输出）
 * @param dev_conv1_spike 输入：设备端Conv1神经元脉冲输出（batch×6×24×24）
 * @param dev_pool1_spike 输出：设备端Pool1神经元脉冲输出（batch×6×12×12）
 * @param batch_size 输入：批量处理的样本数
 */
__global__ void spike_pool1_kernel(
    const float* dev_conv1_spike,
    float* dev_pool1_spike,
    int batch_size
) {
...
}
```

课程大作业一讲解

- ✓全连接层与分类（CUDA实现，PTX优化思路）
  - ✓全连接层核心逻辑：脉冲值×权重的加权求和 + LIF神经元更新
  - ✓用CUDA实现基础逻辑，PTX优化思路参考卷积层。

课程大作业一讲解

- ✓全连接层与分类（CUDA实现，PTX优化思路）
  - ✓全连接层1（FC1：256→120）

```
/**
 * @brief 脉冲全连接核（FC1，处理Pool1的脉冲输出，生成膜电位）
 * @param dev_pool1_spike 输入：设备端Pool1脉冲输出（batch×6×12×12=batch×256）
 * @param dev_fc1_w 输入：设备端FC1权重（120×256）
 * @param dev_fc1_mem 输入输出：设备端FC1神经元膜电位（batch×120）
 * @param batch_size 输入：批量处理的样本数
 * @param t 输入：当前时间步（用于膜电位衰减）
 */
__global__ void spike_fc1_kernel(
    const float* dev_pool1_spike,
    const float* dev_fc1_w,
    float* dev_fc1_mem,
    int batch_size,
    int t
) {
...
}
```

课程大作业一讲解

- ✓全连接层与分类（CUDA实现，PTX优化思路）
  - ✓LIF神经元（FC1）与后续全连接层（FC2、FC3）
    - ✓FC2（120→84）、FC3（84→10）的实现逻辑与FC1完全一致，仅需修改宏定义参数（如FC2_IN_DIM、FC2_OUT_DIM）。

课程大作业一讲解

- ✓全连接层与分类（CUDA实现，PTX优化思路）
  - ✓多时间步脉冲计数与分类（主机端）
    - ✓SCNN分类逻辑：统计SIM_STEPS时间步内FC3输出层每个类别的脉冲总数，脉冲数最多的类别即为预测结果。

```
/**
 * @brief 多时间步脉冲计数与分类（基于FC3的脉冲输出）
 * @param dev_fc3_spike 输入：设备端FC3多时间步脉冲输出（batch×10×SIM_STEPS）
 * @param batch_size 输入：批量处理的样本数
 * @param sim_steps 输入：脉冲编码时间步数（=SIM_STEPS）
 * @param host_pred 输出：主机端存储的预测类别（batch_size个）
 */
void classify_by_spike_count(
    const float* dev_fc3_spike,
    int batch_size,
    int sim_steps,
    std::vector<int>& host_pred
) {
...
}
```

课程大作业一讲解

✓主函数：SCNN推理流程整合（适配大作业要求）