中国科学院大学计算机学院专业选修课

GPU架构与编程

第七课：PTX编程（一）

赵地
中科院计算所
2025年秋季学期

讲授内容

- CUDA编程补充知识：寄存器编程
- PTX编程的主要内容
- PTX 编程模型
- PTX 机器模型
- PTX语言的语法规则（Syntax）
- PTX存储模型和数据类型
- Instruction Operands
- 二进制接口（ABI）

CUDA编程补充知识：寄存器编程

- ✓寄存器编程
  - ✓在NVIDIA V100架构（Volta 架构）下，寄存器编程与优化是高性能CUDA编程的关键，因为寄存器是GPU中最快的存储资源，直接影响线程的执行效率和整体性能。
  - ✓在V100架构下，寄存器优化的Core：减少寄存器使用量、避免溢出、提高占用率和利用架构特性（如Tensor Core和共享内存）。
  - ✓通过简化变量、控制线程块大小、展开循环和使用分析工具，程序员可以显著提升CUDA内核性能。
  - ✓结合Nsight Compute和Occupancy Calculator等工具，可以进一步精细化优化，确保在V100的高并行性下实现最佳性能。

CUDA编程补充知识：寄存器编程

- ✓V100架构与寄存器概述
  - ✓SM结构：V100每个流多处理器（SM）包含64个FP32 Core、32个FP64Core、64个INT32Core和8个Tensor Core，支持更高的并行性和混合精度计算。
  - ✓寄存器文件：每个SM拥有256 KB的寄存器文件，分为 65,536个32位寄存器（每个线程最多可使用255个寄存器）。相比Pascal架构，Volta增加了寄存器文件的容量，减少了寄存器压力。

CUDA编程补充知识：寄存器编程

- ✓V100 架构与寄存器概述
  - ✓线程束（Warp）调度：V100 引入了独立线程调度（Independent Thread Scheduling），允许每个线程在 warp 内有更灵活的执行路径，但可能增加寄存器需求。
  - ✓统一内存和缓存：Volta 架构将 L1 缓存和共享内存统一为 128 KB 可配置存储，优化数据局部性，间接影响寄存器使用。

CUDA编程补充知识：寄存器编程

- ✓V100 架构与寄存器概述
  - ✓寄存器的作用：
    - ✓寄存器用于存储线程私有变量（例如局部变量、临时变量、指针等）。
    - ✓每个线程的寄存器使用量直接影响线程并行度和 SM 占用率（occupancy）。
    - ✓过多的寄存器使用会导致“寄存器溢出”（spill），将数据存储到较慢的本地内存（local memory），显著降低性能。

CUDA编程补充知识：寄存器编程

- ✓寄存器编程
  - ✓寄存器分配
    - ✓CUDA 编译器（NVCC）根据内核代码自动分配寄存器。程序员无法直接指定寄存器，但可以通过代码结构影响分配。
    - ✓每个线程的最大寄存器数由编译选项 --maxrregcount 控制（默认最大 255 个）。
    - ✓使用 nvcc -Xptxas -v 查看内核的寄存器使用量。
    - ✓例如：ptxas info: Used 48 registers, 320 bytes cmem[0] //表示每个线程使用了 48 个寄存器

CUDA编程补充知识：寄存器编程

- ✓寄存器编程
  - ✓寄存器溢出（Spill）
    - ✓当线程使用的寄存器超过分配限制时，编译器会将变量“溢出”到本地内存（local memory），存储在全局内存中，访问延迟高（数百个周期）。
    - ✓检测溢出：使用 --ptxas-options=-v 检查编译器输出中的 spill loads 和 spill stores。
    - ✓减少溢出：
      - ✓减少局部变量和复杂计算。
      - ✓避免过大的数组或结构体作为局部变量。
      - ✓使用共享内存（shared memory）替代溢出的本地内存。

CUDA编程补充知识：寄存器编程

- ✓寄存器编程
  - ✓线程并行度与占用率
    - ✓V100 每个 SM 最多支持 2048 个线程（64 warps）。寄存器使用量决定每个 SM 能容纳的线程块（blocks）和线程数。
    - ✓寄存器使用量高的内核会减少每个 SM 的线程块数，降低占用率，影响性能。
    - ✓占用率公式：

$$\text{Occupancy} = \frac{\text{活跃线程数}}{\text{最大线程数 (2048)}}$$

CUDA编程补充知识：寄存器编程

- ✓寄存器编程
  - ✓Volta 架构的寄存器优化特性
    - ✓独立线程调度：允许线程在 warp 内独立执行，但可能导致寄存器状态保存开销增加。避免不必要的分支发散（divergence）以减少寄存器压力。
    - ✓Tensor Core：Tensor Core操作（如 HMMA）通常需要额外的寄存器来存储中间结果。优化时需平衡Tensor Core指令和常规计算的寄存器分配。
    - ✓增强的统一内存：通过统一内存减少显式数据拷贝，间接减少寄存器用于指针管理的开销。

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓减少寄存器使用量
    - ✓简化变量声明：
      - ✓使用较小的数据类型（如 float 替代 double，int 替代 long），减少寄存器需求。
      - ✓尽量复用变量，避免声明过多临时变量。例如：

```
// 低效：多个临时变量
float a = x * y;
float b = a + z;
float c = b * w;
// 高效：复用变量
float result = x * y;
result = result + z;
result = result * w;
```

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓减少寄存器使用量
    - ✓避免复杂表达式：
      - ✓限制函数调用：
        - ✓内联函数（__inline__ 或 __forceinline__）减少函数调用栈的寄存器开销。
        - ✓避免递归函数，递归会显著增加寄存器使用。

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓减少寄存器使用量
    - ✓避免复杂表达式：
      - ✓将复杂计算拆分为多个简单步骤，允许编译器优化中间结果的寄存器分配。例如：

```
// 复杂表达式，可能占用更多寄存器
float result = (x * y + z) * (w / v + u);
// 拆分
float temp1 = x * y + z;
float temp2 = w / v + u;
float result = temp1 * temp2;
```

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓控制线程块大小
    - ✓选择合适的线程块大小（block size），平衡寄存器使用和占用率。例如，假设每个线程使用 48 个寄存器，线程块大小为 256：寄存器需求 = 256×48 = 12,288个寄存器。每个 SM 有 65,536 个寄存器，可支持约 5 个线程块（65,536 / 12,288 ≈ 5.33）。
    - ✓使用 NVIDIA 的 Occupancy Calculator 工具估算最优线程块大小。

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓利用共享内存
    - ✓将频繁访问的局部数组或变量移到共享内存，减少寄存器和本地内存的使用。
    - ✓权衡寄存器与共享内存：V100的统一内存允许灵活配置共享内存和L1缓存。增加共享内存可能减少寄存器溢出，但会降低L1缓存命中率。

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓优化分支和循环
    - ✓减少分支发散：V100 的独立线程调度减轻了分支发散的性能惩罚，但发散仍可能增加寄存器使用（保存线程状态）。
    - ✓循环展开：使用 #pragma unroll 展开小循环，减少循环控制变量的寄存器开销。例如：

```
#pragma unroll
for (int i = 0; i < 4; i++)
{
    sum += data[i];
}
```

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓使用编译器选项
    - ✓限制寄存器使用：使用 --maxrregcount N 强制限制每个线程的寄存器数（例如，-maxrregcount 32）。
    - ✓优化级别：使用 -O3 启用最高优化级别，编译器会尝试最小化寄存器使用。
    - ✓分析工具：使用 NVIDIA Nsight Compute 或 nvprof 分析寄存器使用和溢出情况。
      - ✓例如：nvprof --metrics l1_local_load_hit_rate,l1_local_store_hit_rate ./my_kernel

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓利用 Tensor Core
    - ✓Tensor Core适用于矩阵运算（如深度学习）。优化时，使用 CUDA 的 WMMA（Warp Matrix Multiply-Accumulate）API，减少手动矩阵操作的寄存器开销。WMMA 指令将矩阵操作卸载到 Tensor Core，减少寄存器需求。

CUDA编程补充知识：寄存器编程

- ✓寄存器优化策略
  - ✓利用 Tensor Core
    - ✓例如：

```
#include <mma.h>
using namespace nvcuda::wmma;
__global__ void wmma_example(half* a, half* b, float* c) {
    fragment<matrix_a, 16, 16, 16, half, row_major> a_frag;
    fragment<matrix_b, 16, 16, 16, half, col_major> b_frag;
    fragment<accumulator, 16, 16, 16, float> c_frag;
    load_matrix_sync(a_frag, a, 16);
    load_matrix_sync(b_frag, b, 16);
    fill_fragment(c_frag, 0.0f);
    mma_sync(c_frag, a_frag, b_frag, c_frag);
    store_matrix_sync(c, c_frag, 16, mem_row_major);
}
```

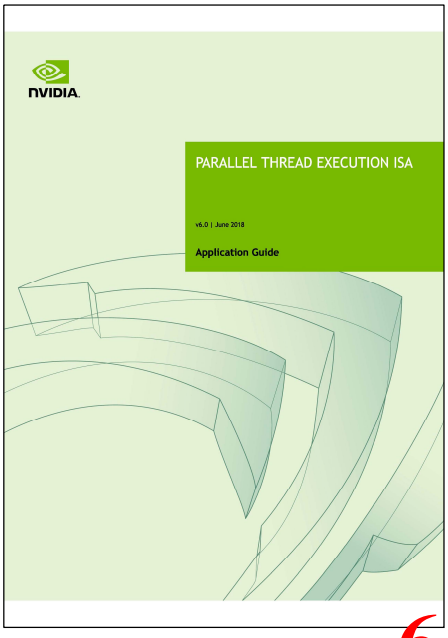
CUDA编程补充知识：寄存器编程

- ✓性能分析与工具
  - ✓Nsight Compute：
    - ✓分析寄存器使用、溢出、本地内存访问。
    - ✓检查 SM 占用率和 warp 调度效率。
  - ✓nvprof（CUDA 11.2 及以下）：
    - ✓收集指标如 spill_transactions 和 local_load_transactions。
  - ✓NVIDIA Occupancy Calculator：
    - ✓输入线程块大小、寄存器使用量和共享内存需求，计算理论占用率。

讲授内容

- CUDA编程补充知识：寄存器编程
- **PTX编程的主要内容**
- PTX 编程模型
- PTX 机器模型
- PTX语言的语法规则（Syntax）
- PTX存储模型和数据类型
- Instruction Operands
- 二进制接口（ABI）

参考书：PTX编程



Parallel Thread Execution ISA Version 6.0, 2018

核心概念：状态空间（State Spaces）

- ✓状态空间（State Spaces）：状态空间定义了数据存储的位置和访问规则，是PTX内存模型的Core
  - ✓.const：只读常量空间，用于存储编译时常量。
  - ✓.global：全局内存空间，所有线程可访问，但延迟较高。
  - ✓.local：线程本地内存，每个线程私有。
  - ✓.shared：线程块共享内存，用于线程间高效通信。
  - ✓.param：参数传递空间，用于内核函数参数。
  - ✓.tex：纹理内存空间（已弃用，被.texref替代）。
- ✓状态空间支持泛型地址（generic addressing），允许动态选择内存空间。

核心概念：类型系统（Types）

- ✓PTX支持多种数据类型，确保与GPU硬件对齐：
  - ✓基本类型：包括位类型（.b8、.b16、.b32、.b64）、整数类型（.u16、.s32等）、浮点类型（.f16、.f32、.f64）和谓词类型（.pred）。
  - ✓向量类型：如.v2.f32（2个float向量），用于SIMD操作。
  - ✓数组和结构体：支持静态数组和扁平化结构，通过字节数组处理复杂数据类型。
  - ✓类型转换：cvt指令支持不同类型间的转换，包括舍入模式（如.rn舍入到最近偶数）。

核心概念：类型系统（Types）

- ✓PTX支持多种数据类型，确保与GPU硬件对齐：
  - ✓基本类型：包括位类型（.b8、.b16、.b32、.b64）、整数类型（.u16、.s32等）、浮点类型（.f16、.f32、.f64）和谓词类型（.pred）。
  - ✓向量类型：如.v2.f32（2个float向量），用于SIMD操作。
  - ✓数组和结构体：支持静态数组和扁平化结构，通过字节数组处理复杂数据类型。
  - ✓类型转换：cvt指令支持不同类型间的转换，包括舍入模式（如.rn舍入到最近偶数）。

指令集架构（Instruction Set Architecture）：类型系统（Types）：整数算术指令

- ✓基本运算：add、sub、mul、mad（乘加），支持饱和（.sat）和进位（.cc）。
- ✓位操作：and、or、xor、not、shl（左移）、shr（右移）。
- ✓高级操作：popc（人口计数）、clz（前导零计数）、bfind（最高位查找）。
- ✓扩展精度：addc、subc等支持多字运算。

指令集架构：浮点算术指令

- ✓基本运算：add、sub、mul、fma（融合乘加），支持舍入模式（.rn、.rz、.rm、.rp）和刷新子规范数（.ftz）。
- ✓特殊函数：rcp（倒数）、sqrt（平方根）、sin/cos（三角函数）、ex2/lg2（指数对数）。
- ✓半精度支持：.f16和.f16x2类型，用于高效存储和计算。

指令集架构：数据移动和转换指令

- ✓内存访问：ld（加载）、st（存储）、prefetch（预取），支持缓存操作（如.cg全局缓存）。
- ✓类型转换：cvt指令处理类型和大小转换。
- ✓地址操作：cvta指令在泛型地址和具体状态空间地址间转换。

指令集架构：同步和通信指令

- ✓线程同步：bar（屏障同步）、bar.warp.sync（线程束内同步）。
- ✓原子操作：atom（原子运算，如add、min）、red（归约操作），支持内存顺序（.relaxed、.acquire）。
- ✓投票和洗牌：vote（线程束内投票）、shfl（寄存器洗牌），用于数据交换。

指令集架构：矩阵操作指令（新特性）

- ✓wmma指令：用于 warp 级矩阵乘加（D = A * B + C），支持混合精度（如.f16输入、.f32累加），优化深度学习工作负载。

核心概念：特殊寄存器（Special Registers）

- ✓特殊寄存器提供运行时环境信息，只读且线程特定：
  - ✓线程标识：%tid（线程ID）、%ctaid（线程块ID）、%ntid（线程块大小）。
  - ✓线程束信息：%laneid（线程束内通道ID）、%warpid（线程束ID）。
  - ✓性能计数：%clock、%clock64（周期计数器）、%pm0-%pm7（性能监控寄存器）。
  - ✓内存信息：%total_smem_size（共享内存大小）、%dynamic_smem_size（动态分配大小）。
  - ✓掩码寄存器：如%lanemask_eq（通道相等掩码），用于条件执行。

核心概念：内存一致性模型（Memory Consistency Model）

- ✓确保多线程程序的正确性：
  - ✓操作类型：区分宽松（relaxed）、获取-释放（acquire-release）和顺序一致（sequential consistency）操作。
  - ✓因果关系：通过程序顺序、通信顺序和同步操作（如fence）建立线程间可见性。
  - ✓作用域：支持.cta（线程块）、.gpu（GPU内）和.sys（系统级）同步。
  - ✓关键保证：禁止“无中生有”（no thin air）值，确保每个地址的顺序一致性。

讲授内容

- CUDA编程补充知识：寄存器编程
- PTX编程的主要内容
- PTX 编程模型
- PTX 机器模型
- PTX语言的语法规则（Syntax）
- PTX存储模型和数据类型
- Instruction Operands
- 二进制接口（ABI）

讲授内容：PTX 编程模型

① SIMT（Single-Instruction, Multiple-Thread）架构

② 线程层次结构

③ 内存层次结构

④ 编程模型的Core设计原则

⑤ 实际应用示例

SIMT（Single-Instruction, Multiple-Thread）架构

- ✓执行机制
  - ✓主机将数据并行内核（Kernel）函数编译为PTX指令，加载到GPU执行。
  - ✓内核函数独立处理不同数据元素（如像素、网格点），通过线程并发隐藏内存延迟。
- ✓SIMT（Single-Instruction, Multiple-Thread）架构
  - ✓线程以Warp（32 线程）为单位执行相同指令，分支时序列化执行（Divergence）。
  - ✓优势：低开销线程调度（零调度延迟），支持细粒度并行（如单线程处理单数据点）。
- ✓适用场景
  - ✓高算术强度（Arithmetic Intensity）任务（计算/内存访问比 >10），例如矩阵乘法、物理仿真。

讲授内容：PTX 编程模型

① SIMT（Single-Instruction, Multiple-Thread）架构

② 线程层次结构

③ 内存层次结构

④ 编程模型的Core设计原则

⑤ 实际应用示例

线程层次结构

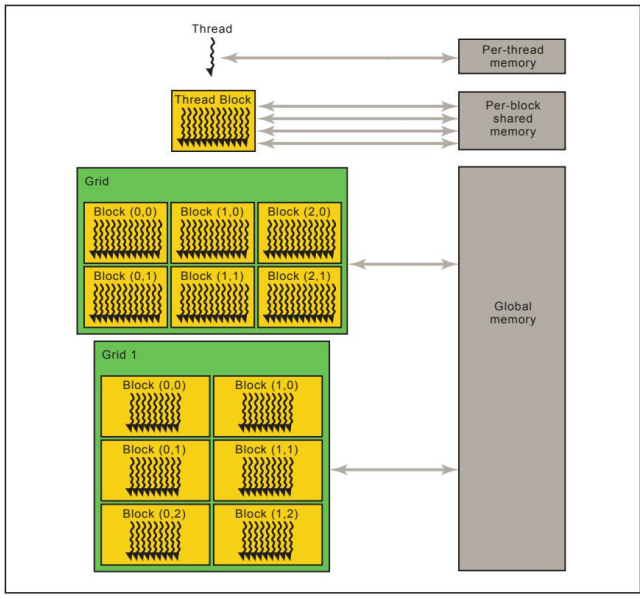
- ✓线程组织为两层结构：网格（Grid） 和协作线程数组（CTA）
- ✓协作线程数组（CTA）：CTA 是线程执行的逻辑单元，等效于CUDA的线程块（Thread Block）
  - ✓线程标识与同步：
    - ✓每个线程有唯一3D ID（%tid.x, %tid.y, %tid.z），范围由CTA维度（%ntid）定义。
    - ✓线程间通过barrier.sync显式同步（如共享内存访问前）。
  - ✓Warp 执行特性：
    - ✓Warp是硬件调度单位（固定32线程），线程在Warp内锁步执行（Lockstep）。

网格（Grid）

- ✓网格（Grid）：网格是 CTA 的集合，管理大规模并行任务
- ✓结构定义：
  - ✓网格包含多个 CTA，每个 CTA 有唯一 3D ID（%ctaid），网格尺寸由 %nctaid 定义。
  - ✓网格 ID（%gridid）标识唯一内核调用，支持多内核并发。
- ✓动态扩展性：
  - ✓CTA 间 无同步/通信，依赖全局内存交换数据（避免死锁）。

网格（Grid）

✓ 网格（Grid）包含多个CTA，每个CTA包含多个线程。主机启动内核时，分配网格到GPU多处理器（SM）。



讲授内容：PTX 编程模型

① SIMT（Single-Instruction, Multiple-Thread）架构

② 线程层次结构

③ 内存层次结构

④ 编程模型的Core设计原则

⑤ 实际应用示例

内存层次结构：内存空间与特性

✓ GPU 内存采用分级设计，优化访问延迟和带宽：

内存类型	作用域	生命周期	访问权限	使用场景
寄存器（.reg）	线程私有	线程执行期	读/写	存储局部变量（最快访问）
共享内存（.shared）	CTA 内共享	CTA 执行期	读/写	线程间通信（低延迟缓存）
本地内存（.local）	线程私有	线程执行期	读/写	溢出寄存器（慢于寄存器）
全局内存（.global）	全局可见	应用生命周期	读/写	跨 CTA 数据交换（高延迟）
常量内存（.const）	全局可见	内核启动期	只读	存储常量数据（缓存加速）
纹理内存（.tex）	全局可见	应用生命周期	只读	图像处理（硬件滤波）

内存层次结构：关键机制

- ✓一致性模型：
  - ✓全局内存无硬件缓存一致性，CTA 间通信需显式同步（如原子操作 atom.global）。
  - ✓共享内存仅在 CTA 内一致，通过 barrier.sync 确保可见性。
- ✓访问优化：
  - ✓合并访问（Coalesced Access）：相邻线程访问连续地址，触发单次内存事务。
  - ✓常量/纹理内存：只读缓存，适合广播数据（如查找表）。

讲授内容：PTX 编程模型

① SIMT（Single-Instruction, Multiple-Thread）架构

② 线程层次结构

③ 内存层次结构

④ 编程模型的Core设计原则

⑤ 实际应用示例

编程模型的Core设计原则

- ✓数据并行优先：
  - ✓任务分解为独立数据单元（如单像素），映射到线程。
  - ✓避免线程间依赖（如 CTA 内共享内存优化，网格级全局内存通信）。
- ✓资源感知设计：
  - ✓线程数量受限于 SM 资源（寄存器/共享内存）。示例：CTA 尺寸（%ntid）需适配硬件（如 SM 支持 1024 线程/CTA）。
- ✓显式同步控制：
  - ✓使用 barrier 协调内存访问，避免竞态（Race Condition）。
  - ✓原子操作（atom）用于全局内存互斥访问。

讲授内容：PTX 编程模型

① SIMT（Single-Instruction, Multiple-Thread）架构

② 线程层次结构

③ 内存层次结构

④ 编程模型的Core设计原则

⑤ 实际应用示例

编程模型的Core设计原则

✓图像滤波：
- ✓网格划分图像区块，CTA 内线程处理滤波窗口，使用纹理内存加速采样。
- ✓通过此模型，开发者可高效利用 GPU 硬件资源，实现高性能并行计算。

编程模型的Core设计原则

✓矩阵乘法优化：
  ✓CTA 分配子矩阵计算，共享内存缓存输入数据块，减少全局内存访问。
  ✓代码片段：

```
.shared .f32 blockA[16][16];  // CTA 共享内存
ld.global.f32 data, [globalPtr];
st.shared.f32 blockA[threadIdx.y][threadIdx.x] = data;
barrier.sync;  // 同步确保数据就绪
```

讲授内容

- CUDA编程补充知识：寄存器编程
- PTX编程的主要内容
- PTX 编程模型
- **PTX 机器模型**
- PTX语言的语法规则（Syntax）
- PTX存储模型和数据类型
- Instruction Operands
- 二进制接口（ABI）

SIMT多处理器架构

- ✓Core硬件单元：流式多处理器（SM）
  - ✓标量处理器Core（SP Cores）
    - ✓数量随架构演进（Maxwell: 128, Ampere: 128），执行基础算术逻辑运算
    - ✓支持并行执行整数（INT）、浮点（FP）、张量（Tensor）指令
  - ✓硬件线程调度器
    - ✓零开销切换Warp（32 线程组），通过指令流水线隐藏延迟
    - ✓Volta 架构引入独立线程调度（Independent Thread Scheduling），打破Warp锁步限制

SIMT多处理器架构

- ✓关键执行机制：SIMT（Single-Instruction, Multiple-Thread）
  - ✓Warp执行原理：
    - ✓32线程共享指令指针，执行相同指令流
    - ✓分支分歧（Branch Divergence）时串行执行所有路径，性能损失 = 分支路径数
  - ✓动态资源分配：
    - ✓每个SM并发管理多个Warp（Ampere: 64 Warps/SM
    - ✓硬件自动平衡计算/内存等待：当Warp阻塞（如内存访问），立即切换就绪Warp

SIMT多处理器架构

✓片上内存层次

内存类型	位置	访问延迟	带宽	特性
寄存器文件（Register）	SM 片上	1 周期	10 TB/s	线程私有，零访问开销
共享内存（Shared）	SM 片上	20~30 周期	1.5 TB/s	CTA 内共享，bank 冲突敏感
L1 缓存/常量缓存	SM 片上	30~50 周期	400 GB/s	自动缓存全局/常量数据
纹理缓存	SM 片上	50~80 周期	300 GB/s	优化空间局部性访问

内存子系统详解

✓寄存器文件设计
- ✓精细分区：
  - ✓每个线程独占寄存器组（Max: 255 寄存器/线程，Ampere）
  - ✓分区策略：静态分配至 Warp，避免动态竞争
- ✓性能约束：
  - ✓寄存器用量影响 SM 并发 Warp 数：
    - ✓Max Warps/SM = (Register File Size) / (Registers per Thread × 32)
  - ✓溢出处理：寄存器不足时数据移入本地内存（+200 周期延迟）

共享内存优化机制

- ✓32-bank 组织：
  - ✓Bank宽度4字节（32-bit），连续地址映射至循环bank
  - ✓冲突场景：同一Warp多线程访问同一bank → 序列化访问
- ✓访问模式优化：

```
// 避免 bank 冲突：线程 i 访问 bank[i % 32]
.shared .f32 data[128];
ld.shared.f32 r0, [data + tid.x*4];  // stride=4 字节，无冲突
```

全局内存访问

- ✓合并访问（Coalesced Access）：
  - ✓理想情况：Warp 内 32 线程访问连续128字节（缓存行）→ 单次事务
  - ✓非合并惩罚：随机访问触发32次独立事务，带宽下降32倍
- ✓缓存策略：

缓存操作符	目标缓存	使用场景
.ca	L1+L2	高频复用数据
.cg	L2	只读全局数据
.cv	无缓存	流式写入数据

资源分配与限制

✓SM 资源容量模型
  ✓关键约束公式：

```
有效并发 Warp = min(
  物理 Warp 槽数量,
  ⌊ 寄存器总量 / (每线程寄存器 × 32)⌋,
  ⌊ 共享内存总量 / 每 CTA 共享内存⌋
)
```

资源分配与限制

✓实例计算（Ampere GA102）：
  ✓物理上限：64 Warps/SM
  ✓寄存器约束：128K 寄存器 ÷ (64 reg/thread × 32) = 62.5 Warps
  ✓共享内存约束：164 KB ÷ 48 KB/CTA = 3.4 CTA → 3 CTA × 16 Warps/CTA = 48 Warps

线程块（CTA）调度策略

- ✓贪婪调度（Greedy Scheduling）：
  - ✓硬件分发器动态分配 CTA 至空闲 SM
  - ✓单个 SM 运行多个 CTA（Ampere: 16 CTA/SM）
- ✓负载均衡：
  - ✓自动填充所有 SM，无空闲周期（Zero Idle）
  - ✓支持 CTA 间无依赖并行

与编程模型的协同

- ✓SIMT与线程层次的映射
  - ✓物理执行流程：

Grid → 分配到多个 SM
→ CTA 被 SM 接收
→ Warp 由调度器发射
→ 线程在 SP Core执行

与编程模型的协同

- ✓关键优化点：
  - ✓Warp占用率：保持≥32活跃Warp/SM以隐藏内存延迟
  - ✓分支效率：控制Warp内分支分歧率（< 10%）

与编程模型的协同

- ✓内存模型实现
- ✓一致性协议：
  - ✓共享内存：CTA内通过barrier.sync保证一致性
  - ✓全局内存：依赖 .acquire/.release 语义跨 CTA 同步
- ✓原子操作支持：

```
atom.global.add.u32 d, [addr], 1;  // 全局内存原子加
atom.shared.and.b32 d, [saddr], mask; // 共享内存原子与
```

架构演进对比

架构特性	Pascal (2016)	Ampere (2020)	变革点
SM 数量/GPU	56	108	+93%
每 SM 寄存器容量	64K	128K	+100%
共享内存容量/SM	96 KB	164 KB	+71%
Tensor Core	不支持	支持	AI 计算革命

架构演进对比

- ✓设计哲学：
  - ✓PTX 机器模型通过抽象硬件细节（如 SM 数量、缓存大小），使同一份 PTX 代码可跨代执行（Maxwell → Hopper），同时暴露关键优化开关（如缓存操作符、同步指令）实现峰值性能。

讲授内容

- CUDA编程补充知识：寄存器编程
- PTX编程的主要内容
- PTX 编程模型
- PTX 机器模型
- PTX语言的语法规则（Syntax）
- PTX存储模型和数据类型
- Instruction Operands
- 二进制接口（ABI）

源文件格式

✓行分隔：使用换行符（\n）分隔行。
✓空白符处理：所有空白字符（空格、制表符等）等效，仅用于分隔token，编译时忽略。
✓预处理器支持：支持C预处理器（如#include, #define），以#开头的行是预处理指令。
✓大小写敏感：PTX关键字小写（如.reg, .global），标识符大小写敏感模块起始要求：每个PTX模块必须以.version指令指定PTX语言版本，后跟.target指令指定目标架构（如sm_70）。

```
.version 6.0
.target sm_70
```

源文件格式

- ✓关键约束：
  - ✓标识符长度：建议实现支持至少1024字符。
  - ✓文件编码：纯ASCII，不支持Unicode。

注释

- ✓多行注释：用/*和*/包围，可跨行。
- ✓单行注释：以//开始，延伸至行尾。
- ✓规则：
  - ✓注释不能嵌套（如/* /* 嵌套 */ */ 非法）。
  - ✓注释不能出现在字符串或字符常量内。
  - ✓注释被视为空白符，不影响语法解析。

语句

- ✓PTX语句分为两类：指令语句（Instruction Statements）和提示语句（Directive Statements）。
- ✓所有语句以分号（;）结束。

语句

- ✓提示语句（Directive Statements）
  - ✓指令以.开头，控制汇编过程，不生成机器码。常见指令包括：
    - ✓状态空间声明：如.global, .shared。
    - ✓类型定义：如.b32（32位位类型）。
    - ✓对齐与初始化：如.align 8（8字节对齐）。

```
.global .u32 g_var; // 声明全局变量
.shared .align 8 .b8 s_buffer[128]; // 共享内存，8字节对齐
```

语句

✓指令语句（Instruction Statements）

✓指令执行计算或内存操作，格式为：

```
[@p] opcode [d], [a], [b], [c]; // 可选谓词@P，操作码+操作数
```

✓其中：

✓操作码（Opcode）：如add, ld, st（表2列出保留关键字）。

✓操作数：目标操作数d在前，源操作数a, b, c在后。

✓谓词保护：可选@p或@!p控制条件执行。

```
@p add.u32 %r1, %r2, %r3; // 若谓词p为真，执行加法
```

语句

✓指令语句（Instruction Statements）

✓保留指令关键字：

指令类别	示例指令	描述
算术运算	add, mul	加/乘法
内存访问	ld, st	加载/存储
控制流	bra, call	分支/函数调用
纹理操作	tex, tld4	纹理采样

标识符（Identifiers）

- ✓标识符命名变量、标签或函数，规则如下：
  - ✓格式：以字母开头，后跟字母、数字、下划线（_）或美元符（$）；或以_, $, %开头。
    - ✓有效：foo, _data, %r1
    - ✓无效：123var（数字开头）。
  - ✓预定义标识符（表3）：如%tid（线程ID）、%ctaid（CTA ID），用于访问硬件寄存器。

```
mov.u32 %r1, %tid.x; // 读取线程ID的x分量
```

  - ✓作用域：标识符在定义的作用域内唯一（如函数内或模块全局）。

常量（Constants）

- ✓常量是编译时确定的值，支持整数、浮点数和谓词类型。
  - ✓整数常量（Integer Constants）
    - ✓表示法：
      - ✓十进制：42
      - ✓十六进制：0x1F
      - ✓八进制：0123
      - ✓二进制：0b1010
    - ✓类型推断：默认带符号（.s64），若值超范围或加U后缀则为无符号（.u64）。

```
0xFFFF0000U // 无符号64位常量
```

    - ✓ 特殊常量：WARP_SZ 定义Warp大小（通常为32）。

常量（Constants）

- ✓常量是编译时确定的值，支持整数、浮点数和谓词类型。
  - ✓浮点常量（Floating-Point Constants）
    - ✓表示法：
      - ✓标准：3.14, -1.0e-5
      - ✓十六进制精确值：0f3f800000（单精度1.0）。
    - ✓默认类型：双精度（.f64），无单精度直接后缀。

常量（Constants）

- ✓常量是编译时确定的值，支持整数、浮点数和谓词类型。
  - ✓谓词常量（Predicate Constants）
    - ✓整数常量在谓词上下文中解释：非零为真（True），零为假（False）。

```
mov.pred %p, 1; // %p = True
```

常量表达式（Constant Expressions）

- ✓常量表达式在编译时求值，支持运算符和类型转换。
- ✓表达式组成
  - ✓操作数：常量字面量、预定义标识符（如WARP_SZ）。
  - ✓运算符：支持算术（+, *）、比较（<, ==）、位操作（&, <<）。优先级：

```
.const .u32 SIZE = 1024 + WARP_SZ; // 编译时计算
```

常量表达式（Constant Expressions）

- ✓评估规则
  - ✓整数表达式：
    - ✓类型提升：操作数较小类型提升至.s64或.u64。
    - ✓除法/取余：除零行为未定义，结果平台相关。
  - ✓浮点表达式：以双精度（.f64）计算。
  - ✓类型转换：显式转换用(.s64)或(.u64)。

```
.const .u32 OFFSET = (.u32)(SIZE * 2); // 显式转换
```

常量表达式（Constant Expressions）

✓评估规则：

操作	规则	示例
加法（整数）	结果类型同操作数	1 + 2 → .s64
移位	第二操作数解释为.u32	a << b（b为.u32）
浮点比较	操作数必须同为.f64	1.0 < 2.0 → True

常量表达式（Constant Expressions）

✓常量表达式限制：

✓不支持函数调用或运行时变量。

✓数组大小、内存对齐值必须为常量表达式。

PTX语言的语法规则（Syntax）总结

- ✓语法一致性：PTX语法设计确保与CUDA模型兼容，例如线程标识符（%tid）直接映射硬件。
- ✓性能影响：常量表达式优化（如编译时计算）减少运行时开销。
- ✓错误处理：语法错误在编译时捕获（如非法标识符或未闭合注释）。
- ✓工具链集成：语法规则支持无缝编译至目标ISA（如通过NVCC）。

常量表达式（Constant Expressions）

- ✓常量表达式限制：
  - ✓不支持函数调用或运行时变量。
  - ✓数组大小、内存对齐值必须为常量表达式。

讲授内容

- CUDA编程补充知识：寄存器编程
- PTX编程的主要内容
- PTX 编程模型
- PTX 机器模型
- PTX语言的语法规则（Syntax）
- PTX存储模型和数据类型
- Instruction Operands
- 二进制接口（ABI）

讲授内容：PTX存储模型和数据类型

① State Spaces（状态空间）线程层次结构

② Types（类型系统）

③ Texture/Surface类型

④ Variables（变量声明）

State Spaces（状态空间）线程层次结构

✓状态空间定义了变量的存储位置和访问特性。
✓PTX中所有变量必须声明在特定状态空间中，其属性和用途总结如下：

状态空间	描述	可寻址	共享范围	典型用途
.reg	线程私有寄存器（最快访问）	×	单线程	临时计算/指令操作数
.sreg	只读特殊寄存器（如%tid.x, %ctaid）	×	CTA内共享	线程/块ID获取
.const	常量内存（主机初始化，只读）	√	全部线程	查表/固定参数
.global	全局内存（跨网格持久化，可读写）	√	全部线程+主机	输入/输出大数据

State Spaces（状态空间）线程层次结构

状态空间	描述	可寻址	共享范围	典型用途
.local	线程私有内存（栈空间）	√	单线程	大型临时变量/递归
.param	参数空间（传递内核参数或函数参数）	√	网格或线程	内核参数/函数传值
.shared	CTA共享内存（低延迟，块内共享）	×	CTA内线程	线程协作/数据复用
.tex	纹理内存（已废弃，改用.texref）	×	全部线程	废弃API

State Spaces（状态空间）线程层次结构

- ✓关键特性：
  - ✓地址对齐：访问全局/共享内存时需自然对齐（如.f32需4字节对齐），否则行为未定义。
  - ✓作用域：.param空间的双重性：内核参数为网格级只读，函数参数为线程级可读写。
  - ✓生命周期：.global和.const在网格间持久化；.shared和.reg仅在CTA内有效。

State Spaces（状态空间）线程层次结构

- ✓向量类型（Vectors）
- ✓语法：.v2/.v4 + 基础类型（如 .v4.f32）。
- ✓限制：向量总长≤128位（如 .v4.f64 非法）。
- ✓访问：

```
.reg .v4 .f32 V;
mov.f32 %r1, V.x;   // 取第一个元素
mov.b64  %r2, V;    // 整体打包到64位寄存器
```

讲授内容：PTX存储模型和数据类型

① State Spaces（状态空间）线程层次结构

② Types（类型系统）

③ Texture/Surface类型

④ Variables（变量声明）

Types（类型系统）

✓PTX的类型系统支持标量、向量和特殊类型，强调操作兼容性和硬件优化。

✓基础类型（Fundamental Types）

类型类别	类型标识符	尺寸（位）	说明
有符号整数	.s8, .s16, .s32, .s64	8/16/32/64	用于算术/逻辑运算
无符号整数	.u8, .u16, .u32, .u64	8/16/32/64	地址计算/位操作
浮点数	.f16, .f32, .f64	16/32/64	.f16需特殊指令支持
位类型	.b8, .b16, .b32, .b64	8/16/32/64	无类型数据，兼容所有操作
谓词类型	.pred	1	条件执行（如 @p 守卫）

Types（类型系统）

- ✓关键规则：
- ✓类型兼容性：
  - ✓同尺寸整数类型互通（如.s32与.u32可互操作）。
  - ✓位类型（.b*）兼容所有同尺寸类型。
- ✓子字限制：
  - ✓.u8/.s8仅用于ld/st/cvt；.f16需显式转换到.f32。

讲授内容：PTX存储模型和数据类型

① State Spaces（状态空间）线程层次结构

② Types（类型系统）

③ Texture/Surface类型

④ Variables（变量声明）

Variables（变量声明）

✓核心语法：

```
.space .type [.align N] [= init] name [dim];
```

✓示例：

```
.const .align 8 .b8  lookup[256] = {0}; // 常量数组，8字节对齐
.shared .v4 .f32 buf[32];              // CTA共享内存，128位向量
.reg .pred p, q;                       // 谓词寄存器
```

Variables（变量声明）

✓关键特性：

- ✓对齐（Alignment）：
  - ✓.align N 强制对齐（N为2的幂），缺省时按类型尺寸对齐（如.f32→4字节）。
  - ✓向量变量默认按总尺寸对齐（如.v4.f32→16字节）。
- ✓初始化（Initialization）：
  - ✓.const/.global支持静态初始化（缺省为0），数组可缺省第一维（由初始化式推断）。
  - ✓其他空间（如.shared）需运行时初始化。

Variables（变量声明）

- ✓关键特性：
  - ✓地址操作：
    - ✓获取变量地址：mov.u64 %addr, generic(var);
    - ✓数组索引：ld.global.f32 %val, [array + %index*4];
  - ✓变量属性（PTX 4.0+）：

```
.global .attribute(.managed) .s32 g; // UVM统一内存管理
```

Variables（变量声明）

- ✓设计意图与硬件映射：
  - ✓性能导向：
    - ✓.reg和.shared映射到SM片上资源（低延迟），.global映射到显存（高延迟）。
    - ✓向量类型（.v4.f32）优化内存吞吐（合并访问）。
  - ✓兼容性：
    - ✓类型转换需显式指令（如 cvt.f32.b32），避免隐式行为。
  - ✓扩展性：
    - ✓纹理/表面类型抽象硬件单元（如Texture Cache），支持高级采样和过滤。

讲授内容

- CUDA编程补充知识：寄存器编程
- PTX编程的主要内容
- PTX 编程模型
- PTX 机器模型
- PTX语言的语法规则（Syntax）
- PTX存储模型和数据类型
- Instruction Operands
- 二进制接口（ABI）

讲授内容：Instruction Operands

① 总体介绍

② 类型兼容性规则（Operand Type Information）

③ 源操作数（Source Operands）

④ 目标操作数（Destination Operands）

⑤ 地址格式，数组访问，向量支持指令

⑥ 数据类型转换

⑦ 操作数开销（Operand Costs）

总体介绍

- ✓操作数类型兼容性规则：
  - ✓操作数类型必须与指令模板和指令类型兼容。
  - ✓bit-size 类型（如 .b32）与同尺寸的任意类型兼容。
  - ✓整数类型（.s32/.u32）在相同尺寸下相互兼容，但需按指令类型静默转换（如无符号数用于有符号指令时视为有符号数）。
  - ✓浮点类型需完全匹配（如 .f32 指令不可用 .f64 操作数）。

总体介绍

- ✓源操作数（a, b, c）
  - ✓绝大多数 ALU 指令要求源操作数为 .reg 寄存器空间的变量。
  - ✓例外指令：
    - ✓ld/st/mov/cvt：支持跨状态空间（如全局内存）的数据搬运。
    - ✓cvt：允许不同类型/尺寸的转换
  - ✓谓词操作数（如 p, q）：
    - ✓可选条件执行（@p 或 @!p），控制指令是否生效。

总体介绍

- ✓目标操作数（d）：
  - ✓存储指令结果的标量或向量寄存器（.reg 空间）。
  - ✓单结果指令（如 add）写入 d；双结果指令（如 setp）用 | 分隔（例：setp.lt.s32 p|q, a, b）。
  - ✓位桶操作：用 _ 忽略结果（如 st.param.b32 [ptr], _）。

总体介绍

- ✓Using Addresses, Arrays, and Vectors：
  - ✓Addresses as Operands
    - ✓地址格式（内存指令如 ld/st）
    - ✓对齐要求：地址需按访问尺寸对齐（未对齐行为未定义）
    - ✓通用寻址（Generic Addressing）
  - ✓Arrays as Operands
    - ✓数组访问
  - ✓Vectors as Operands
    - ✓向量支持指令：mov、ld、st、tex
    - ✓元素访问
    - ✓内存宽访问
  - ✓Labels and Function Names as Operands
    - ✓标签/函数名
    - ✓mov 取函数地址（例：mov.u32 %ptr, myFunc）

总体介绍

- ✓Type Conversion：
  - ✓Scalar Conversions
    - ✓显式转换指令：cvt
      - ✓整数间转换：零扩展（无符号）或符号扩展（有符号）。
      - ✓浮点转整数：需指定舍入模式（如 .rni 就近取偶）。
      - ✓特殊值处理：超出范围的浮点数转为最大浮点值（如 Inf）。
  - ✓Rounding Modifiers
    - ✓浮点舍入：.rn（就近取偶）、.rz（向零舍入）、.rm（向负无穷）、.rp（向正无穷）
    - ✓整数舍入：.rni（同 .rn）、.rpi（向上取整）

总体介绍

- ✓Operand Costs
  - ✓状态空间访问开销
    - ✓零开销：.reg、.sreg、.const（首次访问需缓存）。
    - ✓高延迟：.local（>100 周期）、.global（>100 周期）。
  - ✓延迟隐藏技巧：
    - ✓多线程切换
    - ✓提前发起 load 指令
    - ✓store 指令的寄存器可快速释放

讲授内容：Instruction Operands

① 总体介绍

② 类型兼容性规则（Operand Type Information）

③ 源操作数（Source Operands）

④ 目标操作数（Destination Operands）

⑤ 地址格式，数组访问，向量支持指令

⑥ 数据类型转换

⑦ 操作数开销（Operand Costs）

类型兼容性规则（Operand Type Information）

- ✓定义：PTX 指令中操作数的类型兼容性规则，确保指令执行时数据类型和尺寸严格匹配。
- ✓Core逻辑：
  - ✓指令类型修饰符（如 .u32/.f64）决定操作数的预期类型。
  - ✓操作数实际类型需通过声明或上下文推断（如寄存器变量 .reg .s32 %r）。

类型兼容性规则（Operand Type Information）

✓基础兼容性矩阵

✓✓：允许隐式转换（如 .b32 操作数用于 .u32 指令时按无符号处理）。

✓✗：禁止使用（如 .f32 指令不可接受 .s32 操作数）。

指令类型	.bX	.sX	.uX	.fX
.bX	✓	✓	✓	✓
.sX	✓	✓	✓	✗
.uX	✓	✓	✓	✗
.fX	✓	✗	✗	✓

类型兼容性规则（Operand Type Information）

✓特殊规则

✓bit-size 类型（.bX）：

✓与同尺寸的任何类型兼容（例：.b32 可替代 .u32/.s32/.f32）。

✓常用于位操作（如 and.b32）或未明确类型的中间数据。

✓整数类型（.sX/.uX）：

✓同尺寸时兼容，但符号语义由指令决定（例：add.u32 将 .s32 操作数视为无符号数）。

✓浮点类型（.fX）：

✓必须完全匹配（.f32≠.f64），不支持与整数混合运算。

类型兼容性规则（Operand Type Information）

- ✓隐式转换行为
  - ✓当操作数类型与指令类型兼容但不完全匹配时，PTX 自动触发以下转换：
    - ✓整数扩展/截断：
      - ✓小尺寸→大尺寸：零扩展（无符号）或符号扩展（有符号）。
      - ✓大尺寸→小尺寸：截断低有效位（如 .u64→.u32 保留低 32 位）。
    - ✓bit-size 转换：
      - ✓直接按位复制，无数值解释（例：.b16 移入 .b32 时高 16 位未定义）。

类型兼容性规则（Operand Type Information）

- ✓例外与限制
  - ✓子字尺寸限制（如 .u8/.s8）：
    - ✓仅允许用于 ld/st/cvt 指令（见表 24）。
    - ✓其他指令需扩展至 32/64 位（如 add.u16 需 16 位寄存器）。
  - ✓浮点精度控制：
    - ✓.f16 仅支持与 .f32/.f64 相互转换或纹理指令。
    - ✓.f16x2 专用于半精度向量运算（如 add.f16x2）。

类型兼容性规则（Operand Type Information）

- ✓类型检查流程示例：
  - ✓add.s32 %r0, %r1, %r2
    - ✓指令类型：.s32 → 要求所有操作数为 32 位有符号整数。
    - ✓操作数检查：
      - ✓若 %r1 声明为 .u32：兼容（静默视为有符号数）。
      - ✓若 %r2 声明为 .b32：兼容（按位解释为 .s32）。
      - ✓若 %r2 声明为 .f32：报错（类型不匹配）。

源操作数（Source Operand）

- ✓定义： PTX 指令的输入数据来源。
- ✓核心规则：PTX 采用 Load-Store 架构，算术逻辑指令（ALU）的源操作数必须来自寄存器空间（.reg）内存数据需先通过 ld/mov 加载至寄存器才能参与运算。

源操作数（Source Operand）

✓源操作数的分类与规则

✓通用源操作数（a, b, c）：

类型	允许的状态空间	示例指令	限制条件
ALU 操作数	.reg	add.s32 d, a, b	尺寸必须一致
内存地址	全局/本地/共享/常量空间	ld.global.f32 d, [a]	地址需对齐访问尺寸
常量立即数	编译时常量	add.u32 d, a, 42	值在指令类型范围内

源操作数（Source Operand）

✓特殊指令的源操作数

✓转换指令（cvt）：

✓支持任意类型/尺寸的源操作数

✓数据搬运指令：

指令	功能	源操作数位置
ld	内存→寄存器	可寻址状态空间
st	寄存器→内存	.reg
mov	寄存器间复制/取地址	.reg 或变量地址

源操作数（Source Operand）

- ✓关键约束与例外
  - ✓尺寸一致性要求：
    - ✓多数指令要求源操作数尺寸匹配（如 min.u32 的 a 和 b 需同为 32 位）。
    - ✓例外：cvt、ld/st 支持尺寸转换
  - ✓子字操作数限制：
    - ✓.u8/.s8/.b8 仅限 ld/st/cvt 使用（其他指令需扩展至 32+ 位）。
    - ✓例：8 位加法需先扩展：

```
ld.u8  %r1, [addr]      // 加载 8 位数据
cvt.u32.u8 %r2, %r1     // 扩展至 32 位
add.u32 %r3, %r2, 1     // 32 位加法
```

源操作数（Source Operand）

- ✓关键约束与例外
  - ✓谓词操作数范围：
    - ✓仅限 .pred 类型寄存器（1 位值）。
    - ✓不可与整数混用（需通过 selp 转换）。

源操作数（Source Operand）

✓典型用例分析

✓例子一：向量点积计算

```
// 步骤 1: 从全局内存加载向量至寄存器
ld.global.v4.f32 {f1,f2,f3,f4}, [vecA_addr]
ld.global.v4.f32 {f5,f6,f7,f8}, [vecB_addr]

// 步骤 2: 寄存器内执行浮点乘法 (源操作数均在 .reg)
mul.f32 f9, f1, f5
mul.f32 f10, f2, f6
mul.f32 f11, f3, f7
mul.f32 f12, f4, f8

// 步骤 3: 寄存器内累加结果
add.f32 f13, f9, f10
add.f32 f14, f11, f12
add.f32 result, f13, f14
```

源操作数（Source Operand）

✓典型用例分析

✓例子二：条件性内存存储

```
@p ld.global.u32  %val, [addr]  // 仅当 p=True 时加载
setp.lt.s32       %p, %val, 100  // 比较生成新谓词
@%p st.shared.b32 [ptr], %val   // 仅当 %val<100 时存储
```

目标操作数（Destination Operand）

- ✓典型用例分析
  - ✓目标操作数（d）是 PTX 指令的结果存储位置；
  - ✓遵循 Load-Store 架构的Core原则；
  - ✓所有计算结果必须存回寄存器空间（.reg）内存写入需通过 st 指令显式完成，禁止直接修改内存数据。

目标操作数（Destination Operand）

- ✓目标操作数的关键特性
  - ✓存储位置限制

指令类型	目标操作数空间	示例	硬件关联
ALU 运算指令	.reg	add.s32 d, a, b	结果直写寄存器文件
数据移动指令	.reg	mov.f32 d, a	寄存器间传输
原子操作	.reg	atom.add.u32 d, [addr], val	结果先存寄存器，再可选存内存

目标操作数（Destination Operand）

- ✓多目标操作数支持
  - ✓双目标指令：用 | 分隔多个目标寄存器
  - ✓例如：setp.lt.s32 p|q, a, b // p = (a < b), q = !(a < b)
  - ✓仅限谓词比较指令（如 setp），物理实现依赖 SM 的 Predicate Register File。
- ✓结果丢弃机制
  - ✓位桶操作符 _：显式忽略结果
  - ✓例如：mul.wide.s32 _, a, b // 计算 a*b 但不存储 64 位结果
  - ✓优化作用：减少寄存器压力，避免无用写回。

目标操作数（Destination Operand）

- ✓多目标操作数支持
  - ✓双目标指令：用 | 分隔多个目标寄存器
    - ✓例如：setp.lt.s32 p|q, a, b // p = (a < b), q = !(a < b)
    - ✓仅限谓词比较指令（如 setp），物理实现依赖 SM 的 Predicate Register File。
  - ✓结果丢弃机制
    - ✓位桶操作符 _：显式忽略结果
    - ✓例如：mul.wide.s32 _, a, b // 计算 a*b 但不存储 64 位结果
    - ✓优化作用：减少寄存器压力，避免无用写回。

目标操作数（Destination Operand）

- ✓目标尺寸的放松规则
  - ✓当目标寄存器尺寸 大于 指令类型尺寸时，PTX 自动处理；
  - ✓例外：.f16 和 .f16x2 必须严格匹配目标尺寸。

指令类型	扩展方式	示例
有符号整数	符号扩展	cvt.s64.s32 d, a → d[63:32]=符号位
无符号整数	零扩展	cvt.u64.u16 d, a → d[63:16]=0
浮点/bit 类型	零扩展	mov.b64 d, f32_val → d[63:32]=0

目标操作数（Destination Operand）

- ✓与硬件模型的关联
  - ✓目标操作数的物理存储发生在 Streaming Multiprocessor (SM) 的寄存器文件中：
    - ✓标量寄存器：
      - ✓32 位/64 位数据存于 Scalar Processor (SP) 私有寄存器堆
      - ✓写回延迟：0 周期（结果直通至下一指令）
    - ✓谓词寄存器：
      - ✓1 位结果存于 Predicate Register File
      - ✓单周期写回，支持条件执行反馈

目标操作数（Destination Operand）

- ✓关键约束与优化
  - ✓子字存储限制：
    - ✓8/16 位目标需通过 st 显式截断：

```
cvt.u16.u32  %r1, %r2    // 32→16 位转换
st.shared.b16 [ptr], %r1 // 显式存储 16 位
```

目标操作数（Destination Operand）

- ✓关键约束与优化
  - ✓向量目标对齐要求：
    - ✓.v4 向量目标需 16 字节对齐

```
.align 16
.reg .v4.f32 V
ld.global.v4.f32 V, [addr] // addr 需对齐 16 字节
```

目标操作数（Destination Operand）

- ✓关键约束与优化
  - ✓寄存器重用优化：
    - ✓SM 架构支持 Register File Bypass，避免写后读冲突。

```
mad.rn.f32  res, a, b, res  // res 同时作源和目标
```

复合数据结构的访问规则

- ✓定义：PTX 中复合数据结构的访问规则，涵盖地址计算、数组索引和向量操作三大关键能力
- ✓设计目标：在保持硬件效率的前提下，提供类似高级语言的抽象能力通过统一的寻址模型（Generic Addressing）和向量化指令支持 SIMD 并行。

复合数据结构的访问规则

- ✓Addresses as Operands
  - ✓地址格式与寻址模式

语法格式	示例指令	硬件行为
[var]	ld.shared.u16 r0, [x]	直接变量地址访问
[reg]	st.global.f32 [ptr], f1	寄存器含绝对地址
[reg + immOff]	ld.const.s32 q, [tbl+12]	基址寄存器+常量偏移（32位有符号）
[var + immOff]	mov.u32 %ptr, A+20	变量地址+编译时常量偏移
[immAddr]	ld.local.b64 x, [240]	立即数地址（32位无符号）

复合数据结构的访问规则

- ✓通用寻址（Generic Addressing）
  - ✓机制：将 .const/.local/.shared 映射到虚拟地址空间窗口
  - ✓示例：ld.u32 %r1, [gaddr] // gaddr 自动映射到实际物理空间
  - ✓约束：地址必须按访问尺寸对齐（未对齐行为未定义）

复合数据结构的访问规则

- ✓Arrays as Operands
  - ✓数组操作规则

操作类型	语法示例	底层实现
声明	.local .u16 kernel[19][19]	连续内存分配（361个16位元素）
直接索引	ld.global.u32 s, a[0]	基址 + 索引×元素尺寸
寄存器索引	mul.wide.u32 %off, %idx, 4	需显式计算字节偏移
	ld.global.f32 f, [a + %off]	

复合数据结构的访问规则

- ✓Arrays as Operands
  - ✓特殊初始化

```
.global .s32 offset[][2] = {
    {-1, 0},
    {0, -1},  // 自动填充为 4×2 数组
};
```

复合数据结构的访问规则

✓Vectors as Operands
✓向量支持架构

特性	指令示例	硬件加速机制
声明限制	.reg .v4 .f32 V	最大 128 位（4×32位）
元素访问	mov.f32 %f1, V.x	寄存器文件并行读写
内存宽访问	ld.global.v4.f32 {f1,f2,f3,f4}, [addr]	合并内存事务（Coalescing）
结构化存取	mov.b32 {x,y,z,w}, %r1	位域解构（零周期开销）

复合数据结构的访问规则

✓Vectors as Operands
✓内存对齐要求

```
.align 16        // 128位向量需16字节对齐
.global .v4 .f32 V
ld.global.v4.f32 V, [0x10000]  // 地址必须对齐16字节
```

复合数据结构的访问规则

- ✓Labels and Function Names as Operands
  - ✓符号操作数使用场景

类型	合法指令	非法用法
代码标签	bra L1	add.s32 L1, a, b
函数名	call myFunc, (a, b)	st.global.f32 [f], cos
函数地址	mov.u64 %ptr, @sin	atom.add.u32 [sin], v

复合数据结构的访问规则

- ✓Labels and Function Names as Operands
  - ✓动态并行支持（PTX 3.1+）

```
mov.u64 %kptr, kernel_entry  // 获取内核入口地址
// 用于CUDA Dynamic Parallelism系统调用
```

类型转换（Type Conversion）

- ✓定义：在 PTX 中通过 cvt 指令显式完成
- ✓Core功能：在不同类型/尺寸的标量数据间进行位模式转换所有转换行为遵循 IEEE 754 和 PTX 类型兼容性规则（表 13），不支持隐式转换。

类型转换（Type Conversion）

✓Scalar Conversions

✓转换规则矩阵

目标类型 ↓	.sX	.uX	.fX
.sX	符号扩展/截断	零扩展/截断	✗ 禁止
.uX	零扩展/截断	零扩展/截断	✗ 禁止
.fX	有符号整数转浮点	无符号整数转浮点	精度调整/舍入

类型转换（Type Conversion）

- ✓关键转换语义
  - ✓整数 → 浮点：
    - ✓例子：cvt.f32.s32 %f, %i // 32位有符号整数→单精度浮点
    - ✓超出浮点范围的值转为 ±Inf
    - ✓非正规数（Subnormal）处理依赖 .ftz 修饰符
  - ✓浮点 → 整数：
    - ✓例子：cvt.rni.s32.f32 %i, %f // 单精度浮点→32位有符号整数（就近取偶）
    - ✓必须指定整数舍入修饰符（.rni/.rzi等）
    - ✓NaN 转换为未定义值，超出范围时饱和到极值
  - ✓同尺寸转换：
    - ✓例子：
    - ✓cvt.rn.f32.f32 %f2, %f1 // 浮点舍入（需显式指定）
    - ✓cvt.u32.u32 %u2, %u1 // 无操作（冗余指令）

类型转换（Type Conversion）

- ✓Rounding Modifiers
  - ✓浮点转换必须显式指定舍入模式（如 cvt.rn.f32.f64）

修饰符	行为	IEEE 754 等效	硬件实现
.rn	最近偶数值（默认）	RoundTiesToEven	SPMD Core原生支持
.rz	向零舍入	RoundTowardZero	截断低位
.rm	向负无穷舍入	RoundTowardNegative	增加符号逻辑
.rp	向正无穷舍入	RoundTowardPositive	增加符号逻辑

类型转换（Type Conversion）

- ✓Rounding Modifiers
  - ✓整数舍入模式，仅适用于浮点→整数转换

修饰符	数学行为	示例（输入 2.5）
.rni	就近取偶	→ 2.0
.rzi	向零截断	→ 2.0
.rmi	向负无穷（floor）	→ 2.0
.rpi	向正无穷（ceil）	→ 3.0

类型转换（Type Conversion）

- ✓硬件实现细节
  - ✓关键路径延迟：
    - ✓整数转换：1 周期
    - ✓浮点转换：4-8 周期（依赖精度）
  - ✓非正规数处理（.ftz）
    - ✓启用条件：.ftz 修饰符 + 单精度浮点转换
    - ✓行为：if (abs(input) < FLT_MIN) output = signed_zero;
    - ✓架构差异：

目标架构	默认行为	.ftz 效果
sm_1x	强制冲洗为零	冗余（无变化）
sm_20+	保留非正规数	启用冲洗为零

类型转换（Type Conversion）

- ✓特殊案例解析
- ✓案例 1：双精度→单精度饱和转换
  - ✓例子：cvt.rn.ftz.f32.f64 %f32, %f64 // 含非正规数冲洗
    - ✓位模式转换（64b→32b 尾数截断）
    - ✓就近取偶舍入
    - ✓若结果小于 2^{-126}，符号位为零
- ✓案例 2：浮点→8 位整数
  - ✓例子：cvt.rzi.s8.f32 %s8, %f32
    - ✓向零舍入到整数
    - ✓饱和到 [-128, 127]
    - ✓截断到 8 位存储

操作数开销（Operand Costs）

- ✓定义：量化了从不同状态空间（State Spaces）访问数据的时间开销与资源消耗
- ✓核心目标：为编译器/程序员提供性能预测模型，指导寄存器分配、数据布局和指令调度优化。

操作数开销（Operand Costs）

✓开销分类与量化模型

✓操作数开销矩阵

- ✓ 延迟：从发起访问到数据就绪的时钟周期
- ✓ 吞吐量：每 SM 每周期可完成的访问次数

状态空间	访问类型	延迟周期	吞吐量	硬件路径	优化建议
.reg	寄存器读写	0	极高	Register File Bypass	优先使用，减少内存访问
.sreg	特殊寄存器读	0	中	常量缓存（Constant Cache）	只读，避免频繁读取动态值
.const	常量缓存命中	1–2	高	L1 常量缓存	小常量优先放 .const
	常量缓存未命中	100+	低	全局内存→L2→常量缓存	合并访问，利用空间局部性
.shared	Bank 无冲突访问	0–5	高	共享内存阵列	避免 Bank Conflict
	Bank 冲突访问	10–30	骤降	Bank 仲裁延迟	使用跨步/广播优化
.global	L1/L2 缓存命中	20–50	中	L1→L2→SM Core	利用缓存行（128B 对齐）
	全局内存访问	100–300	低	DRAM→PCIe（如适用）	合并访问（Coalescing）
.local	栈帧访问	20–100	中低	L1/L2 或全局内存	减少栈深度，复用寄存器

操作数开销（Operand Costs）

✓硬件实现机制

✓内存层次与访问路径

SM Core → Register File (0cy)
→ Shared Memory (1-30cy)
→ L1 Cache (20-50cy) → L2 Cache (50-100cy) → Global Memory (100-300cy)
→ Constant Cache (1-100cy)

操作数开销（Operand Costs）

- ✓关键路径：
  - ✓寄存器文件：零延迟（直通路径）
  - ✓共享内存：Bank 并行访问（32 Banks @ 4B/Bank）
  - ✓常量缓存：只读广播，支持单周期多线程同地址访问

操作数开销（Operand Costs）

- ✓性能降低案例
  - ✓Bank Conflict：

```
// 所有线程访问同一 Bank（Bank 0）
ld.shared.u32 %r0, [%s + threadIdx.x * 4]  // 32 线程冲突 → 32 周期串行化
```

  - ✓未合并全局访问：

```
// 线程 i 访问 addr[i*128] → 每线程跨越 128B 缓存行
ld.global.f32 %f, [addr + tid*512]        // 无合并 → 32 次独立内存事务
```

操作数开销（Operand Costs）

✓优化策略与示例

✓策略 1：寄存器重用

✓收益：减少 1 次共享内存访问（节省 5-30 周期）

```
// 低效：多次访问共享内存
ld.shared.f32 %a, [x]
add.f32 %b, %a, 1.0
ld.shared.f32 %c, [y]
mul.f32 %d, %b, %c

// 优化：寄存器缓存中间值
ld.shared.f32 %a, [x]
ld.shared.f32 %c, [y]       // 提前加载
add.f32 %b, %a, 1.0         // 寄存器操作（0cy）
mul.f32 %d, %b, %c
```

操作数开销（Operand Costs）

✓优化策略与示例

✓策略 2：共享内存 Bank 冲突消除

```
// 冲突访问：线程 i 访问 bank = (i % 16)
ld.shared.u32 %r, [base + tid * 4]

// 优化：填充使步长=128B（Bank 数×4B）
ld.shared.u32 %r, [base + tid * 128]  // 步长 128B → 所有线程访问不同 Bank
```

操作数开销（Operand Costs）

- ✓优化策略与示例
  - ✓策略 3：全局内存合并访问
    - ✓收益：吞吐量提升 32 倍（理想情况）

```
// 未合并：分散地址
ld.global.f32 %f, [addr + tid*4]  // 32 线程 → 32 次内存事务

// 合并：连续 128B 块（32 线程×4B）
ld.global.v4.f32 {f0,f1,f2,f3}, [addr + warp_id*128]  // 1 次事务
```

操作数开销（Operand Costs）

- ✓架构差异
  - ✓架构演进影响

特性	sm_1x (Fermi)	sm_20+ (Kepler+)
.const 未命中	无常量缓存	L1→L2 级缓存
.local 访问	全局内存路径	专用 L1 缓存
原子操作	全局内存串行	共享内存硬件加速

操作数开销（Operand Costs）

- ✓架构差异
  - ✓常见陷阱
    - ✓误用 .sreg：mov.u32 %r, %clock // 高频读取 %clock → 成为性能瓶颈
    - ✓忽略 .param 开销：ld.param.f32 %f, [param_ptr] // 内核参数频繁访问 → 实际通过常量内存访问 // 优于多次重复访问

操作数开销（Operand Costs）

- ✓调试与性能工具建议
  - ✓NSight Compute：
    - ✓分析内存事务效率（Global/Local Memory Transactions）
    - ✓检测 Shared Memory Bank Conflict
  - ✓PTXAS 警告：ptxas --warn-on-local-memory-usage
  - ✓SASS 指令检查：nvdisasm --print-instruction-latency kernel.o

讲授内容

- CUDA编程补充知识：寄存器编程
- PTX编程的主要内容
- PTX 编程模型
- PTX 机器模型
- PTX语言的语法规则（Syntax）
- PTX存储模型和数据类型
- Instruction Operands
- 二进制接口（ABI）

二进制接口（ABI）

- ✓核心目标与背景
  - ✓ABI 抽象旨在隐藏底层硬件调用约定（Calling Convention），为 PTX 提供跨架构统一的函数接口规范。
  - ✓Core作用：
    - ✓函数参数传递
    - ✓栈帧管理
    - ✓寄存器保存/恢复：使 PTX 代码独立于 GPU 架构演进（如 sm_20 → sm_80），避免重写高级语言生成的中间代码。

二进制接口（ABI）

- ✓函数（包括内核和设备函数）的声明与定义
- ✓函数声明与定义：在 PTX 中用于封装可重用代码单元
- ✓核心目标
  - ✓支持跨调用（Call）的代码复用
  - ✓抽象底层 ABI（应用二进制接口）细节
  - ✓统一参数传递、栈帧管理和寄存器使用规则

通过 .entry（内核函数）和 .func（设备函数）指令实现，隐藏硬件差异（如 sm_20 与 sm_80 的调用约定区别）。

二进制接口（ABI）

✓内核函数（Kernel Functions）

✓语法与声明：

```
.entry kernel_name (parameter_list) .attribute_list
{
    // 函数体
}
```

✓.entry：标识 GPU 内核入口点（由主机调用）。

✓parameter_list：参数必须使用 .param 状态空间声明。示例：.param .u32 N;

二进制接口（ABI）

✓内核函数（Kernel Functions）

✓属性（.attribute_list）：

属性	作用	示例
.reqntid	指定线程块最小尺寸	.reqntid .x 128
.maxntid	指定线程块最大尺寸	.maxntid .x 1024
.minnctapersm	指定每 SM 最小线程块数	.minnctapersm 4
.maxnreg	限制寄存器使用数量	.maxnreg 32

二进制接口（ABI）

- ✓内核函数（Kernel Functions）
  - ✓关键特性：
    - ✓不可嵌套：内核函数不能调用其他函数（包括自身）。
    - ✓参数传递：
      - ✓参数通过常量内存（.const）传递，只读且跨 CTA 共享。
      - ✓访问需用 ld.param：ld.param.u32 %r0, [N]; // 加载标量参数
    - ✓启动约束：
      - ✓线程块维度通过 %ntid 等特殊寄存器隐式传递。
      - ✓网格维度通过 %nctaid 传递。

二进制接口（ABI）

✓设备函数（Device Functions）
✓语法与定义

```
.func return_type func_name (parameter_list) .attribute_list
{
    // 函数体
    ret;
}
```

二进制接口（ABI）

✓设备函数（Device Functions）

✓语法与定义

✓.func：标识设备级函数（可被内核或其他设备函数调用）。

✓参数列表（parameter_list）：

✓标量参数：使用 .reg 寄存器传递（如 .reg .u32 a）。

✓结构体/数组参数：使用 .param 空间传递（如 .param .b8 data[]）。例如：.func foo(.reg .u32 x, .param .b8 buf[12])

✓返回值：通过 .reg 寄存器返回（单值），或多值通过 .param 结构体。例如：.func (.reg .f32 ret) bar(.reg .f32 a) { ... }

二进制接口（ABI）

- ✓设备函数（Device Functions）
  - ✓ABI 抽象机制
    - ✓参数传递标准化：
      - ✓编译器自动处理寄存器分配（如 .s32 参数映射到物理寄存器 R0-R7）。
      - ✓大结构体通过栈传递（.param 空间），隐藏栈指针管理细节。
    - ✓调用约定：
      - ✓调用方（Caller）：保存易失寄存器（Caller-Saved）。
      - ✓被调用方（Callee）：保存非易失寄存器（Callee-Saved）。
    - ✓栈帧管理：
      - ✓动态分配通过 %alloca 指令
      - ✓栈增长方向由 ABI 定义（sm_20+ 向下增长）。

二进制接口（ABI）

✓函数属性与修饰符
✓可见性控制

修饰符	作用	示例
.visible	函数可跨模块链接（默认）	.func visible foo(...)
.weak	弱符号（允许多重定义）	.func weak bar(...)
.extern	声明外部函数（不定义）	.extern .func baz(...);

二进制接口（ABI）

- ✓函数属性与修饰符
  - ✓对齐与初始化
    - ✓对齐属性。例如：.func .align 16 my_func(...) // 函数入口 16 字节对齐
  - ✓初始化限制：
    - ✓函数内局部变量（.local）不支持静态初始化（需运行时赋值）。
    - ✓全局变量（.global）可初始化，但函数内不可直接访问。

二进制接口（ABI）

- ✓函数作用域与生命周期
  - ✓作用域规则
    - ✓模块级作用域：
      - ✓函数默认在 PTX 模块内可见。
      - ✓通过 .visible/.weak 控制跨模块链接。
    - ✓局部标签：

```
.func foo {
    .reg .pred p;
    @p bra LOCAL_LABEL;  // 标签作用域限于函数内
LOCAL_LABEL:
    ...
}
```

二进制接口（ABI）

- ✓函数作用域与生命周期
  - ✓生命周期管理
    - ✓栈帧生命周期：
      - ✓函数调用时动态分配栈帧（通过 %alloca）。
      - ✓返回时自动释放（编译器插入释放指令）。
    - ✓寄存器保存：
      - ✓非易失寄存器（如 %r8-%r15）由被调用方保存/恢复。
      - ✓易失寄存器（如 %r0-%r7）由调用方负责保存。

二进制接口（ABI）

- ✓函数作用域与生命周期
  - ✓注意事项
    - ✓递归限制：
      - ✓PTX 不支持直接递归（无自动栈展开）。
      - ✓手动递归需显式管理栈深度（易出错）。
    - ✓参数对齐：
      - ✓.param 数组需按元素尺寸对齐（如 .f64 需 8 字节对齐）。
      - ✓未对齐访问导致未定义行为。
    - ✓ABI 兼容性：
      - ✓混合 sm_1x 与 sm_80 代码时需统一调用约定（如寄存器使用）。
      - ✓使用 .callprototype 强制类型检查。例如：.callprototype _Z3fooiPfi int(int, float*);
    - ✓调试支持。
      - ✓通过 .loc 指令映射源码位置。例如：.loc 1 20 "kernel.cu"

二进制接口（ABI）

- ✓可变参数函数（Variadic Functions）
- ✓PTX ISA 6.0 已正式移除可变参数函数支持
  - ✓现代替代方案（PTX 6.0+）
    - ✓结构体打包法
      - ✓优势：类型安全（编译时检查）、单指针传递（零拷贝）
    - ✓多态函数重载
      - ✓适用场景：参数类型有限且已知、需避免内存访问开销
    - ✓缓冲区描述符

二进制接口（ABI）

- ✓可变参数函数（Variadic Functions）
  - ✓PTX ISA 6.0 已正式移除可变参数函数支持
    - ✓现代替代方案（PTX 6.0+）
      - ✓结构体打包法
        - ✓优势：类型安全（编译时检查）、单指针传递（零拷贝）
      - ✓多态函数重载
        - ✓适用场景：参数类型有限且已知、需避免内存访问开销
      - ✓缓冲区描述符

二进制接口（ABI）

- ✓Alloca指令
  - ✓核心概念与作用
    - ✓alloca 指令用于在函数栈帧中动态分配可变大小的内存空间
  - ✓Core特性：
    - ✓运行时按需分配栈内存为局部数组、可变长结构体或暂存缓冲区提供临时存储
    - ✓生命周期与函数调用绑定（自动释放）

二进制接口（ABI）

- ✓Alloca指令
  - ✓指令语法与操作语义
    - ✓基础语法：mov.u64 %dst, %alloca(size);
      - ✓%dst：目标寄存器（存储分配地址，.u64 类型）
      - ✓size：动态大小（.u32 或 .u64 寄存器/立即数），字节对齐
    - ✓硬件行为：
      - ✓调整栈指针 %stack_ptr
      - ✓返回新分配空间的起始地址
      - ✓地址对齐至 8 字节（PTX ABI 要求）

二进制接口（ABI）

- ✓Alloca指令
  - ✓关键约束与边界条件
    - ✓大小限制

```
// 低效：多次小分配
mov.u64 %buf1, %alloca(64);
mov.u64 %buf2, %alloca(128);

// 高效：单次合并分配
mov.u64 %buf, %alloca(192);  // 64+128
```

二进制接口（ABI）

- ✓Alloca指令
  - ✓关键约束与边界条件
    - ✓对齐预计算

```
// 避免碎片化
.reg .u32 aligned_size = (size + 7) & ~7;  // 8 字节对齐
mov.u64 %buf, %alloca(aligned_size);
```

二进制接口（ABI）

✓Alloca指令
  ✓调试与错误处理
    ✓栈溢出检测

```
// 安全分配检查
.reg .pred %ok;
mov.u32 %max_stack, 0x10000;     // 预设阈值
setp.lt.u32 %ok, %size, %max_stack;
@%ok bra ALLOC_OK;
trap;                            // 溢出陷阱
ALLOC_OK:
    mov.u64 %buf, %alloca(%size);
```

二进制接口（ABI）

✓Alloca指令
  ✓调试支持
    ✓Nsight 工具链：
      ✓可视化栈帧布局
      ✓标记 alloca 分配区域
    ✓PTXAS 警告：ptxas --warn-on-stack-size // 检测潜在溢出

二进制接口（ABI）

- ✓Alloca指令
- ✓与静态分配对比

特性	alloca	.local 静态分配
内存来源	栈空间	本地内存（显存）
分配时机	运行时	编译时
最大尺寸	16 KB	512 KB (sm_80)
访问速度	寄存器级延迟（L1）	显存延迟（100+周期）
适用场景	小规模临时变量	大规模持久化数据