中国科学院大学计算机学院专业选修课

GPU架构与编程

第八课：PTX编程（二）

赵地
中科院计算所
2025年秋季学期

https://developer.nvidia.cn/

讲授内容

- 内存一致性模型（Memory Consistency Model）
- 内存一致性模型的适用范围与限制
- 内存操作的定义与核心机制
- 状态空间（State Spaces）
- 内存操作类型（Operation Types）
- 内存操作的作用域机制与分层同步控制（Scope）
- Morally strong operations
- Release-Acquire模式
- 内存操作顺序的层级约束机制（Ordering of memory operations）
- 内存一致性模型中的公理系统（Axioms）
- 内存模型中的边界场景处理（Special Cases）

内存一致性模型（Memory Consistency Model）

- ✓核心目标
  - ✓为多线程并行执行提供可预测的内存操作顺序，确保不同线程对共享内存的读写行为符合预期。
  - ✓适用于 sm_70 及以上架构，不涉及纹理/表面内存操作。

内存一致性模型（Memory Consistency Model）

- ✓关键基础概念
  - ✓内存操作（Memory Operations）
    - ✓基本单元：字节（8-bit），状态空间（如 .global, .shared）是连续的字节序列。
    - ✓操作类型
      - ✓atomic：原子读-修改-写（如 atom, red）。
      - ✓read：加载数据（如 ld, atom）。
      - ✓write：写入数据（如 st, 原子操作的结果）。
      - ✓volatile：带 .volatile 修饰符的指令，禁止重排序。
    - ✓重叠（Overlap）：两个内存操作地址区间有交集即重叠（完全重叠或部分重叠）。

内存一致性模型（Memory Consistency Model）

- ✓关键基础概念
  - ✓作用域（Scope）：定义操作可见性的线程范围
    - ✓.cta：同一线程块（CTA）内的线程。
    - ✓.gpu：同一GPU设备上的所有线程。
    - ✓.sys：包括所有GPU和主机线程的系统级范围。
  - ✓道德强操作（Morally Strong Operations）
    - ✓两个操作满足以下条件之一：
      - ✓同一线程按程序顺序执行。
      - ✓均为强操作（strong），作用域相互包含，且完全重叠（如两个线程写同一地址）。

内存一致性模型（Memory Consistency Model）

- ✓同步机制
  - ✓同步操作类型
    - ✓强操作（Strong Operations）：含 .relaxed, .acquire, .release, .acq_rel, .volatile 修饰符。
    - ✓弱操作（Weak Operations）：含 .weak 修饰符（无同步保证）。
    - ✓同步指令：bar（屏障）, fence（内存栅栏）, ld.acquire, st.release。

内存一致性模型（Memory Consistency Model）

- ✓同步机制
  - ✓Release-Acquire 模式
    - ✓Release Pattern：使当前线程此前操作对其他线程可见。例如：st.release [M] 或 fence; st.relaxed [M]。
    - ✓Acquire Pattern：使其他线程的操作结果对当前线程后续操作可见。例如：ld.acquire [M] 或 ld.relaxed [M]; fence。

内存一致性模型（Memory Consistency Model）

- ✓同步关系建立
  - ✓Fence-SC 顺序：fence.sc 操作按程序顺序和因果顺序一致。
  - ✓屏障同步：bar.sync 使同一屏障上的线程互相等待。
  - ✓Release-Acquire 同步：若 release 写操作被 acquire 读操作观察到，则同步成立。

内存一致性模型（Memory Consistency Model）

- ✓内存操作顺序约束
  - ✓程序顺序（Program Order）
    - ✓同一线程内操作按指令顺序执行（线程间无序）。
  - ✓因果顺序（Causality Order）
    - ✓基于同步操作传递的偏序关系：
      - ✓Base Causality：通过同步操作链（如 X sync-with Y → Y sync-with Z）建立。
      - ✓扩展因果：若读操作 R 观察到写操作 W，则 W 因果先于 R。

内存一致性模型（Memory Consistency Model）

- ✓内存操作顺序约束
  - ✓通信顺序（Communication Order）
    - ✓写操作 W 先于读操作 R → R 读取 W 的值。
    - ✓写操作 W 先于写操作 W' → W 在一致性顺序中先于 W'。

内存一致性模型（Memory Consistency Model）

- ✓关键公理（Axioms）
  - ✓9条公理确保行为一致性
    - ✓一致性（Coherence）：因果顺序在先的写操作，在一致性顺序中必须在前。
    - ✓Fence-SC 约束：fence.sc 的顺序不能违背因果顺序。

内存一致性模型（Memory Consistency Model）

- ✓关键公理（Axioms）
  - ✓9条公理确保行为一致性
    - ✓原子性（Atomicity）：
      - ✓单副本原子性：道德强操作对同一地址的读写不可分割。
      - ✓RMW原子性：重叠的原子操作与写操作互斥（全执行或全不执行）。

内存一致性模型（Memory Consistency Model）

- ✓关键公理（Axioms）
  - ✓9条公理确保行为一致性
    - ✓原子性（Atomicity）：
      - ✓无凭空值（No Thin Air）：值不能“无中生有”（禁止自循环依赖产生非法值，如Load Buffering测试）。
      - ✓按位置顺序一致（Sequential Consistency Per Location）：同一地址的道德强操作需按程序顺序执行。

内存一致性模型（Memory Consistency Model）

- ✓关键公理（Axioms）
  - ✓9条公理确保行为一致性
    - ✓特殊约束
      - ✓混合大小数据竞争（Mixed-size Data Races）
        - ✓部分重叠的数据竞争：模型不保证其行为（如4字节写与8字节读部分重叠）。
        - ✓完全重叠的数据竞争：受公理约束（如两个线程写同一地址需符合一致性）。

内存一致性模型（Memory Consistency Model）

- ✓关键公理（Axioms）
  - ✓9条公理确保行为一致性
    - ✓特殊约束
      - ✓系统级原子性限制
      - ✓64位强操作在CPU-GPU通信时可能非原子

内存一致性模型（Memory Consistency Model）

- ✓总结
  - ✓PTX内存模型通过同步操作（release/acquire）、因果顺序和公理约束，在多线程环境中平衡性能与正确性。
    - ✓避免混合大小数据竞争；
    - ✓用release/acquire 或 fence 同步跨线程操作；
    - ✓同一地址的竞争操作需通过道德强关系或同步消除歧义。

讲授内容

- ➢内存一致性模型（Memory Consistency Model）
- ➢内存一致性模型的适用范围与限制
- ➢内存操作的定义与核心机制
- ➢状态空间（State Spaces）
- ➢内存操作类型（Operation Types）
- ➢内存操作的作用域机制与分层同步控制（Scope）
- ➢Morally strong operations
- ➢Release-Acquire模式
- ➢内存操作顺序的层级约束机制（Ordering of memory operations）
- ➢内存一致性模型中的公理系统（Axioms）
- ➢内存模型中的边界场景处理（Special Cases）

内存一致性模型的适用范围与限制

- ✓核心适用范围
  - ✓目标架构
    - ✓仅适用于 NVIDIA sm_70 及更高架构（如 Volta/Turing/Ampere）。
    - ✓在 sm_70 之前的架构（如 Pascal）上无约束力。
  - ✓程序兼容性
    - ✓适用于 所有 PTX ISA 版本 编写的程序（包括历史版本）。
    - ✓模型定义在硬件层面实现，与 PTX 指令版本无关。

内存一致性模型的适用范围与限制

- ✓关键限制
  - ✓内存操作排除项
    - ✓纹理（Texture）与表面（Surface）内存访问
      - ✓内存一致性模型约束仅针对常规内存操作（.global, .shared, .local 等），纹理/表面内存有独立访问规则（例如 tex 和 suld 指令不参与本模型定义的同步）。

内存一致性模型的适用范围与限制

- ✓关键限制
  - ✓系统级原子性限制
    - ✓CPU-GPU 通信限制：当使用 .sys 作用域的 64 位强操作（如 atom.acq_rel.sys）与主机 CPU 通信时：
      - ✓某些系统可能无法保证原子性（如跨 PCIe 总线）。
      - ✓需参考 CUDA 编程指南确认具体平台的原子性支持。（例如：原子操作可能被拆分为多个子操作）

内存一致性模型的适用范围与限制

- ✓与PTX编程的关联
  - ✓与执行模型的关联
    - ✓模型依赖 CTA 线程协作机制：作用域 .cta 对应线程块内同步，.gpu 对应设备级同步。（如 barrier.sync 指令建立显式同步点）
  - ✓与指令集的关联
    - ✓强操作依赖于指令修饰符：
      - ✓.acquire/.release：ld.acquire.global, st.release.shared
      - ✓.acq_rel：atom.acq_rel.gpu

内存一致性模型的适用范围与限制

- ✓实践建议
  - ✓在 CPU-GPU 共享内存场景：
    - ✓优先使用 .cta 或 .gpu 作用域原子操作。
    - ✓避免依赖 .sys 作用域的 64 位原子操作跨设备通信。（可通过分块计算+归约减少跨系统原子依赖）

内存一致性模型的适用范围与限制

- ✓迁移兼容性
  - ✓针对 sm_70+ 设备的代码需显式声明：
    - ✓.target sm_70 // 必须指定目标架构
- ✓实践建议
  - ✓在 CPU-GPU 共享内存场景：
    - ✓优先使用 .cta 或 .gpu 作用域原子操作。
    - ✓避免依赖 .sys 作用域的 64 位原子操作跨设备通信。（可通过分块计算+归约减少跨系统原子依赖）

内存一致性模型的适用范围与限制

- ✓总结：硬件微架构的约束规范，而非编程接口。
  - ✓仅 sm_70+ 硬件强制遵循此模型；
  - ✓纹理/表面内存需单独处理；
  - ✓跨系统原子操作存在平台差异性。
    - ✓通过 cuDeviceGetAttribute(cudaDevAttrComputeCapabilityMajor) 查询设备架构版本以验证兼容性。

讲授内容

- ➢内存一致性模型（Memory Consistency Model）
- ➢内存一致性模型的适用范围与限制
- ➢内存操作的定义与核心机制
- ➢状态空间（State Spaces）
- ➢内存操作类型（Operation Types）
- ➢内存操作的作用域机制与分层同步控制（Scope）
- ➢Morally strong operations
- ➢Release-Acquire模式
- ➢内存操作顺序的层级约束机制（Ordering of memory operations）
- ➢内存一致性模型中的公理系统（Axioms）
- ➢内存模型中的边界场景处理（Special Cases）

内存操作的定义与核心机制

- ✓内存操作
  - ✓本质：内存操作是 GPU 并行编程中访问共享数据的原子单元
  - ✓构成要素：
    - ✓地址（Address）：内存空间的字节地址（8-bit 对齐）。
    - ✓数据类型（Data Type）：定义操作的内存位置范围（起始地址 + 数据类型宽度）。
    - ✓操作类型（Operation Type）：读（read）、写（write）或原子读-修改-写（atomic）。

```
ld.global.b32 %r0, [addr];  // 读操作：地址=addr, 类型=.b32, 位置=[addr, addr+3]
st.shared.f64 [0x100], %d0; // 写操作：地址=0x100, 类型=.f64, 位置=[0x100, 0x107]
```

关键概念解析

- ✓关键概念解析
  - ✓内存位置（Memory Location）
    - ✓定义：从地址 A 开始、长度为 N 字节的连续空间（N=数据类型宽度）。
    - ✓作用：
      - ✓决定操作的作用域范围（如 4 字节写入影响 4 个字节）。
      - ✓作为冲突检测的基础单位（见下文重叠规则）。

关键概念解析

- ✓关键概念解析
  - ✓重叠（Overlap）
    - ✓两个操作冲突当且仅当：
      - ✓地址区间有交集（部分或完全重叠）；
      - ✓至少一个是写操作。

关键概念解析

- ✓关键概念解析
  - ✓向量数据类型（Vector Data-Type）
    - ✓处理规则：向量操作（如 .v4.f32）被拆解为多个标量操作，执行顺序未定义。

```
st.v4.f32 [addr], {f1, f2, f3, f4};
// 等价于 4 个独立写操作（顺序随机）
```

关键概念解析

- ✓关键概念解析
  - ✓初始化（Initialization）
    - ✓机制：程序启动前，所有字节由虚拟写操作 W0 初始化：
      - ✓若变量有初值：W0 写入该值；
      - ✓否则：写入未知常量（平台相关）。
    - ✓意义：避免未初始化内存的未定义行为。

关键概念解析

✓操作类型分类

操作类型	对应指令/行为	同步性
atomic	atom, red（原子读-修改-写）	强同步
read	ld, atom（读取数据）	弱/强可选
write	st, 原子操作的结果	弱/强可选
volatile	带 .volatile 修饰符的指令	禁止重排序
acquire/release	.acquire/.release 修饰符	建立同步
fence	fence.sc（内存屏障）	全局同步

关键概念解析

- ✓操作类型分类
  - ✓强操作（Strong Operations）：含 .relaxed, .acquire, .release, .acq_rel, .volatile 修饰符，受内存模型严格约束。
  - ✓弱操作（Weak Operations）：含 .weak 修饰符（如 ld.weak），无同步保证。

关键概念解析

- ✓操作类型分类
  - ✓强操作（Strong Operations）：含 .relaxed, .acquire, .release, .acq_rel, .volatile 修饰符，受内存模型严格约束。
  - ✓弱操作（Weak Operations）：含 .weak 修饰符（如 ld.weak），无同步保证。

关键概念解析

- ✓开发注意事项
  - ✓地址对齐要求
    - ✓所有操作需自然对齐（地址 = 数据类型宽度的整数倍）。
    - ✓未对齐操作行为未定义（可能静默掩码或触发异常）。
  - ✓冲突操作风险
    - ✓完全重叠写操作：需通过原子操作或同步避免竞争。
    - ✓部分重叠操作：

```
thread0: st.b32 [addr], %r0;  // 写入 [addr, addr+3]
thread1: ld.b64 %d0, [addr+2]; // 读取 [addr+2, addr+9]（部分重叠）
```

关键概念解析

- ✓开发注意事项
  - ✓向量操作陷阱
    - ✓避免对同一地址的向量读写拆分：

```
// 错误！可能部分写入后被打断
st.v4.f32 [addr], {f1,f2,f3,f4};
```

关键概念解析

✓总结
- ✓内存操作是 GPU 并行内存模型的原子基石，核心要点：
  - ✓位置 = 地址 + 数据类型 → 定义影响范围；
  - ✓冲突由地址重叠 + 写操作触发 → 需同步或原子操作；
  - ✓向量操作非原子 → 拆分为标量执行（顺序随机）；
  - ✓强/弱操作决定是否参与同步 → 根据场景选择修饰符。
- ✓最佳实践：
  - ✓使用 .b32/.b64 访问非类型化数据，避免部分重叠；
  - ✓对共享数据优先使用 atom 或 fence + .release/.acquire。

讲授内容

- ➢内存一致性模型（Memory Consistency Model）
- ➢内存一致性模型的适用范围与限制
- ➢内存操作的定义与核心机制
- ➢状态空间（State Spaces）
- ➢内存操作类型（Operation Types）
- ➢内存操作的作用域机制与分层同步控制（Scope）
- ➢Morally strong operations
- ➢Release-Acquire模式
- ➢内存操作顺序的层级约束机制（Ordering of memory operations）
- ➢内存一致性模型中的公理系统（Axioms）
- ➢内存模型中的边界场景处理（Special Cases）

状态空间（State Spaces）

- ✓关键特性
  - ✓跨空间同步机制
    - ✓因果顺序（Causality Order）全局生效：若线程 A 在 .global 空间的写操作同步到线程 B，则线程 B 在 .shared 空间的后续操作能观察到该结果。

```
Thread A: st.release.global.sys [g_addr], data  →
Thread B: ld.acquire.shared.cta [s_addr]  // 可观察到 g_addr 的写入
```

状态空间（State Spaces）

- ✓关键特性
  - ✓可见性边界
    - ✓操作仅对同空间访问者可见：
      - ✓.shared 空间的操作只能被同一 CTA 内的线程观察。
      - ✓即使指定 .sys 作用域：

```
ld.relaxed.shared.sys %r0, [s_addr]  // 实际同步范围仍限于 .cta
```

状态空间（State Spaces）

- ✓关键特性
  - ✓与纹理/表面空间的隔离
    - ✓纹理（.tex）和表面（.surface）内存 不参与此模型：
      - ✓tex 和 suld 等指令有独立内存规则。
      - ✓若混合访问需显式同步：

```
st.global.release.sys [g_addr], data  // 写全局内存
fence.sys                             // 确保后续纹理操作可见
tex.2d.v4.f32.s32 {...}, [tex_ref]    // 读取纹理
```

状态空间（State Spaces）

✓状态空间与作用域的关系

状态空间	可访问线程范围	实际最大作用域
.const	同一 GPU 所有线程	.gpu
.global	所有 GPU + CPU 线程	.sys
.shared	同一 CTA 内线程	.cta
.local	单一线程	无同步意义

状态空间（State Spaces）

✓状态空间与作用域的关系

✓重要说明：指定更大作用域（如 .sys）不突破空间物理限制：

```
// 实际同步仍限于 .cta（因 .shared 空间外部不可见）
atom.acq_rel.shared.sys %r0, [s_addr]
```

状态空间（State Spaces）

✓开发实践指导

✓跨空间同步策略

✓需通过全局内存中转实现跨 CTA 同步：

```
// CTA 0: 完成计算后写标志位
st.release.global.cta [flag_addr], 1

// CTA 1: 轮询等待标志位
loop:
  ld.acquire.global.cta %r0, [flag_addr]
  setp.eq.s32 %p, %r0, 0
  @%p bra loop
```

状态空间（State Spaces）

- ✓开发实践指导
  - ✓避免隐式假设
    - ✓错误：假设 .const 空间的写入能被其他线程立即可见（常量空间只读）。
    - ✓正确：通过 .global 空间传递更新通知。

状态空间（State Spaces）

- ✓总结
  - ✓状态空间是物理内存区域的逻辑抽象，而内存模型通过两点统一管理：
    - ✓关系定义全局化：因果/同步关系跨越所有状态空间；
    - ✓可见性本地化：操作仅对同空间访问者有效，作用域不突破物理边界。
  - ✓设计原则：跨空间通信必须通过 .global 内存，并用 fence + .release/.acquire 显式同步。

讲授内容

- 内存一致性模型（Memory Consistency Model）
- 内存一致性模型的适用范围与限制
- 内存操作的定义与核心机制
- 状态空间（State Spaces）
- 内存操作类型（Operation Types）
- 内存操作的作用域机制与分层同步控制（Scope）
- Morally strong operations
- Release-Acquire模式
- 内存操作顺序的层级约束机制（Ordering of memory operations）
- 内存一致性模型中的公理系统（Axioms）
- 内存模型中的边界场景处理（Special Cases）

内存操作类型（Operation Types）

✓操作类型核心分类

✓内存一致性模型将操作分为 10 类，定义其同步性和原子性特征：

操作类型	指令/行为	同步强度	典型修饰符
atomic	atom, red（原子读-修改-写）	强	无
read	ld, atom（读取数据）	弱/强可选	.relaxed, .acquire
write	st, 原子操作结果	弱/强可选	.relaxed, .release
volatile	带 .volatile 的指令	强	.volatile
acquire	含 .acquire 或 .acq_rel 的加载	强	.acquire
release	含 .release 或 .acq_rel 的存储	强	.release
strong	所有非 .weak 操作	强	无默认修饰符
weak	含 .weak 的加载/存储	弱	.weak
fence	fence.sc, fence.acq_rel（栅栏）	强	无
synchronizing	bar, fence, 释放/获取操作	强	无

内存操作类型（Operation Types）

- ✓关键类型深度解析
  - ✓原子操作（atomic）
    - ✓不可分割性：硬件保证执行过程不中断（如atom.add.u32）。
    - ✓作用：实现无锁数据结构（如计数器、队列）。
    - ✓限制：
      - ✓仅支持基础类型（.u32/.u64/.s32/.s64/.f32）
      - ✓地址需对齐（4字节对齐 for .u32，8字节对齐 for .u64）

内存操作类型（Operation Types）

- ✓关键类型深度解析
  - ✓强操作（strong）
    - ✓核心特征：
      - ✓参与内存同步模型约束（因果顺序/一致性）
      - ✓包含所有非弱操作（含 volatile/acquire/release）
    - ✓示例差异：

```
ld.relaxed.global.sys %r0, [addr]  // 强操作
ld.weak.global.sys %r1, [addr]     // 弱操作（无同步保证）
```

内存操作类型（Operation Types）

- ✓关键类型深度解析
  - ✓弱操作（weak）
    - ✓设计目的：性能优化（允许硬件重排序）
    - ✓使用场景：
      - ✓非共享数据（如线程局部缓存）
      - ✓不依赖顺序的批量加载
    - ✓风险：

```
st.weak.global [flag], 1     // 弱存储可能延迟
ld.weak.global %r0, [data]   // 可能读到旧值!
```

内存操作类型（Operation Types）

- ✓关键类型深度解析
  - ✓易失操作（volatile）
    - ✓强制行为：
      - ✓禁止编译器优化（如删除"冗余"加载）
      - ✓禁止硬件重排序（相对其他 volatile 操作）
    - ✓典型用例：内存映射 I/O 寄存器访问。

内存操作类型（Operation Types）

- ✓关键类型深度解析
  - ✓同步操作（synchronizing）
    - ✓建立跨线程可见性的操作：
      - ✓显式同步：bar.sync（线程块内屏障）
      - ✓隐式同步：st.release → ld.acquire 构成同步对

```
// Thread A
st.release.global.cta [X], data
// Thread B
ld.acquire.global.cta %r0, [X]  // 保证看到 A 的写入
```

内存操作类型（Operation Types）

- ✓操作类型交互规则
  - ✓强弱操作混合场景

操作序列	是否合法	风险
强读 → 弱写	√	弱写可能重排序到强读前
弱读 → 强写	√	弱读可能看到过时数据
强写 → 弱读	✗	违反可见性保证

内存操作类型（Operation Types）

- ✓原子操作边界约束
  - ✓原子性冲突：若原子操作与写操作完全重叠且为道德强操作
    → 全执行或全不执行。

```
Thread0: atom.add.u32 [addr], 1  // 原子增加
Thread1: st.u32 [addr], 42       // 冲突写入
→ 结果：要么原子操作生效，要么写入生效，不会混合。
```

内存操作类型（Operation Types）

- ✓开发者实践指南
  - ✓基础原则
    - ✓默认用强操作：优先 ld.relaxed/st.relaxed
    - ✓减少弱操作：仅在对性能有实测提升时使用
    - ✓避免混合：同一数据访问保持强度一致

内存操作类型（Operation Types）

- ✓开发者实践指南
  - ✓原子操作优化
    - ✓对高频计数器：用 atom.add 代替 ld+add+st
    - ✓对标志位：用 atom.exch 实现轻量锁

```
// 自旋锁实现
try_lock:
    atom.exch.b32 %r0, [lock_addr], 1  // 尝试置1
    setp.eq.u32 %p, %r0, 0             // 原值为0则获取成功
    @%p bra lock_acquired
    bra try_lock
```

内存操作类型（Operation Types）

- ✓开发者实践指南
  - ✓易失操作使用

```
// 读取硬件状态寄存器（必须用volatile）
ld.volatile.global.u32 %r0, [hw_status_reg]
```

内存操作类型（Operation Types）

✓总结
- ✓操作类型是内存模型的分类核心，其核心规则为：
  - ✓原子操作提供不可分割性 → 冲突操作全执行/全不执行；
  - ✓强弱操作决定同步性 → 强操作参与模型约束，弱操作可重排序；
  - ✓同步操作建立跨线程可见性 → release/acquire 构成同步对。
- ✓验证建议：使用 nv-nsight-cu-cli --metrics smsp__warp_issue_stalled_membar_per_op 检测同步操作性能瓶颈。

讲授内容

- ➢内存一致性模型（Memory Consistency Model）
- ➢内存一致性模型的适用范围与限制
- ➢内存操作的定义与核心机制
- ➢状态空间（State Spaces）
- ➢内存操作类型（Operation Types）
- ➢内存操作的作用域机制与分层同步控制（Scope）
- ➢Morally strong operations
- ➢Release-Acquire模式
- ➢内存操作顺序的层级约束机制（Ordering of memory operations）
- ➢内存一致性模型中的公理系统（Axioms）
- ➢内存模型中的边界场景处理（Special Cases）

内存操作的作用域机制与分层同步控制（Scope）

- ✓核心定义与作用
  - ✓作用域（Scope） 定义了内存操作（如 st.release 或 fence）的可见性边界，指定哪些线程能观察到该操作的内存同步效果。本质是通过分层约束减少同步开销，实现性能与正确性的平衡。

内存操作的作用域机制与分层同步控制（Scope）

- ✓三类作用域详解

作用域类型	覆盖范围	适用操作	典型用例
.cta	同一线程块（CTA）内所有线程	bar.sync, fence.cta	CTA内部共享内存（.shared）同步
.gpu	同一GPU设备上所有线程（含多个CTA）	atom.gpu, fence.gpu	全局内存（.global）跨CTA通信
.sys	全系统（包括所有GPU和CPU线程）	ld.acq.sys, st.rel.sys	GPU-CPU异构系统通信

内存操作的作用域机制与分层同步控制（Scope）

- ✓关键特性对比
  - ✓.cta 作用域
    - ✓硬件支持：通过L1缓存一致性协议实现，延迟最低（约数十时钟周期）。
    - ✓约束：线程块退出后同步效果失效（如 __syncthreads() 仅限块内）。
  - ✓.gpu 作用域
    - ✓硬件支持：依赖 GPU全局内存子系统（L2缓存+内存控制器）。
    - ✓延迟：数百至数千时钟周期（受L2缓存竞争影响）。
  - ✓.sys 作用域
    - ✓硬件支持：通过 PCIe/NVLink总线协议，强制刷新缓存。
    - ✓延迟：微秒级（跨设备通信），慎用！

内存操作的作用域机制与分层同步控制（Scope）

- ✓作用域与内存操作的关系
  - ✓操作类型的作用域兼容性

操作类型	有效作用域	非法组合示例
原子操作（atom）	.cta, .gpu, .sys	atom.shared.sys（物理不可达）
栅栏（fence）	.cta, .gpu, .sys	fence.shared（冗余）
加载/存储	必须匹配空间物理范围（见下表）	ld.acquire.shared.sys

内存操作的作用域机制与分层同步控制（Scope）

- ✓作用域与内存操作的关系
  - ✓状态空间与最大有效作用域
    - ✓重要规则：指定大于物理范围的作用域（如 st.shared.sys）自动降级为物理支持的最大作用域（此处降为 .cta）。

状态空间	实际最大作用域	原因
.shared	.cta	仅同一CTA线程可访问
.local	无效（无同步意义）	线程私有，无需同步
.global	.sys	所有设备+主机可访问
.const	.gpu	同一GPU内只读，主机可更新

内存操作的作用域机制与分层同步控制（Scope）

- ✓分层同步机制
  - ✓作用域层次结构
  - ✓自上而下覆盖：.sys 操作隐含 .gpu 和 .cta 的同步效果（但性能代价高）。
  - ✓自下而上隔离：.cta 操作不影响其他CTA或设备。

内存操作的作用域机制与分层同步控制（Scope）

✓开发实践指南

✓作用域选择原则

场景	推荐作用域	理由
CTA内共享变量更新	.cta	零额外开销，硬件自动保证
多CTA协作归约	.gpu	平衡性能与可见性
GPU-CPU数据交换	.sys	必须使用（但需最小化次数）

内存操作的作用域机制与分层同步控制（Scope）

✓开发实践指南

✓错误用法与修复

✗错误：跨CTA依赖 `.cta` 作用域

```
st.rel.global.cta [x], 1    // 仅当前CTA可见!
```

√修复：升级为 `.gpu`

```
st.rel.global.gpu [x], 1    // 全设备可见
```

内存操作的作用域机制与分层同步控制（Scope）

✓开发实践指南
  ✓性能优化技巧
    ✓作用域降级：

```
// 原：高开销操作
atom.acq_rel.global.sys %r0, [counter]
// 优化：限定GPU内部同步
atom.acq_rel.global.gpu %r0, [counter]
```

内存操作的作用域机制与分层同步控制（Scope）

✓开发实践指南
  ✓性能优化技巧
    ✓作用域范围测试：

```
int supportsSysScope;
cudaDeviceGetAttribute(&supportsSysScope,
                       cudaDevAttrManagedMemory,
                       devId); // 支持.unified内存的设备通常支持.sys
```

内存操作的作用域机制与分层同步控制（Scope）

✓总结

- ✓作用域是分层内存同步的核心控制机制，核心规则：
  - ✓物理边界优先：操作作用域不能超越内存空间的物理可访问范围；
  - ✓性能分级：.cta（纳秒级）→ .gpu（微秒级）→ .sys（毫秒级）；
  - ✓隐式降级：超范围声明自动降级，无错误但可能造成性能浪费。
- ✓黄金法则：从最窄作用域（.cta）开始验证，仅必要时升级作用域，通过 nvprof --metrics gst_transactions 检测全局内存同步开销。

讲授内容

- ➢内存一致性模型（Memory Consistency Model）
- ➢内存一致性模型的适用范围与限制
- ➢内存操作的定义与核心机制
- ➢状态空间（State Spaces）
- ➢内存操作类型（Operation Types）
- ➢内存操作的作用域机制与分层同步控制（Scope）
- ➢Morally strong operations
- ➢Release-Acquire模式
- ➢内存操作顺序的层级约束机制（Ordering of memory operations）
- ➢内存一致性模型中的公理系统（Axioms）
- ➢内存模型中的边界场景处理（Special Cases）

Morally Strong Operations

- ✓内存模型中的核心同步约束机制
  - ✓定义：
    - ✓道德强操作（Morally Strong Operations）：内存一致性模型中定义操作间可见性关系的核心概念。
    - ✓两个操作成为道德强操作（Morally Strong Operations）操作当且仅当满足以下任一条件：
      - ✓程序顺序关联：同一线程内按程序顺序执行的操作

```
st.relaxed.global [X], 1  // 操作A
ld.acquire.global %r0, [Y] // 操作B（与A形成道德强关系）
```

Morally Strong Operations

- ✓两个操作成为道德强操作（Morally Strong Operations）操作当且仅当满足以下任一条件：
  - ✓完全重叠的强操作：
    - ✓两个操作均为强操作（非 .weak）
    - ✓访问的内存位置完全重叠（相同起始地址和大小）
    - ✓作用域相互包含（如线程B在操作作用域内）
- ✓关键区别：道德强操作 ≠ 原子操作。前者是操作间关系，后者是操作属性。

Morally Strong Operations

- ✓核心作用
  - ✓冲突检测的基石
    - ✓当两个操作访问相同内存地址且至少一个是写操作时：
      - ✓若构成道德强关系 → 受模型约束（如顺序性）
      - ✓否则 → 属于数据竞争（Data Race），行为未定义

```
线程A: st.global [addr], 1  // 强操作
线程B: ld.global [addr]     // 强操作 & 完全重叠 → 道德强关系
```

Morally Strong Operations

- ✓核心作用
  - ✓原子性的边界条件
    - ✓对完全重叠的道德强操作，硬件保证：
      - ✓原子操作 vs 写操作：全执行或全不执行
      - ✓读操作 vs 写操作：不会看到中间状态

Morally Strong Operations

✓技术细节解析
  ✓完全重叠要求

场景	道德强操作	原因
相同地址+相同大小访问	√	完全重叠
相同地址+不同大小访问	✗	部分重叠（如4B写 vs 8B读）
相邻地址访问	✗	无重叠

Morally Strong Operations

✓技术细节解析
  ✓完全重叠要求
    ✓部分重叠风险：以下代码不构成道德强关系 → 数据竞争！

```
thread0: st.b64 [addr], %d0    // 写入8字节 [addr, addr+7]
thread1: ld.b32 %r0, [addr+4]  // 读取 [addr+4, addr+7]（部分重叠）
```

Morally Strong Operations

✓技术细节解析
  ✓作用域互包含规则
    ✓有效：线程B在操作A的作用域内

```
A: st.release.cta [X], 1   // 作用域=CTA
B: ld.acquire.cta [X]      // 同CTA → 作用域互包含
```

Morally Strong Operations

✓技术细节解析
  ✓作用域互包含规则
    ✓无效：线程B超出作用域

```
A: st.release.cta [X], 1   // 作用域=CTA
C: ld.acquire.gpu [X]      // 不同CTA → 不构成道德强关系
```

Morally Strong Operations

✓技术细节解析
  ✓弱操作排除规则
    ✓含 .weak 修饰符的操作永不参与道德强关系

```
st.weak.global [X], 1  // 弱存储
ld.relaxed.global [X]   // 强加载 → 仍不构成道德强关系
```

Morally Strong Operations

✓开发者实践指南
  ✓道德强关系建立策略

场景	推荐方法	示例指令
同线程内操作	自然满足（程序顺序）	st → ld
同CTA跨线程	bar.sync + 相同作用域	bar.cta → st.rel.cta
跨CTA通信	全局内存 + .gpu作用域	st.release.gpu → ld.acquire.gpu

Morally Strong Operations

✓开发者实践指南
  ✓需规避场景
    ✓规避部分重叠：

```
// 安全：相同类型访问
st.b32 [addr], %r0
ld.b32 %r1, [addr]

// 危险：混合尺寸访问
st.b64 [addr], %d0
ld.b32 %r1, [addr+4]   // 部分重叠!
```

Morally Strong Operations

✓开发者实践指南
  ✓需规避场景
    ✓强制对齐访问：

```
__align__(8) int2 data; // 确保8字节对齐
```

Morally Strong Operations

- ✓开发者实践指南
  - ✓需规避场景
    - ✓调试检测工具
      - ✓使用 cuda-memcheck --tool racecheck 识别未受保护的冲突访问：

```
# 输出示例
Race detected at address 0x7fcd4a000010
Access 1 by thread (1,0,0) in kernel.cu:20
Access 2 by thread (0,0,0) in kernel.cu:25
```

Morally Strong Operations

- ✓总结
  - ✓道德强操作是内存模型的顺序性保障基石，其核心规则：
    - ✓关系建立：仅限程序顺序或完全重叠的强操作；
    - ✓冲突安全：道德强关系内的冲突操作受硬件原子性保护；
    - ✓开发关键：
      - ✓避免部分重叠访问
      - ✓跨线程通信需显式作用域控制
      - ✓弱操作（.weak）不提供保护
  - ✓终极准则：对共享数据的并发访问，必须通过道德强关系或显式同步（如fence）保护。通过 nvcc --extended-notation -Xptxas -v 编译时检查未保护的内存访问。

讲授内容

- 内存一致性模型（Memory Consistency Model）
- 内存一致性模型的适用范围与限制
- 内存操作的定义与核心机制
- 状态空间（State Spaces）
- 内存操作类型（Operation Types）
- 内存操作的作用域机制与分层同步控制（Scope）
- Morally strong operations
- Release-Acquire模式
- 内存操作顺序的层级约束机制（Ordering of memory operations）
- 内存一致性模型中的公理系统（Axioms）
- 内存模型中的边界场景处理（Special Cases）

Release-Acquire模式

- ✓定义与作用
  - ✓Release-Acquire模式是建立跨线程内存操作可见性的核心机制，通过配对使用两种操作：
    - ✓Release Pattern（释放模式）：
      - ✓确保当前线程之前的所有操作对其他线程可见
      - ✓类型：st.release, atom.release, fence + st.relaxed
    - ✓Acquire Pattern（获取模式）：
      - ✓确保当前线程之后的所有操作能看到其他线程的修改
      - ✓类型：ld.acquire, atom.acquire, ld.relaxed + fence
  - ✓核心目标：解决写后读（Store-Load）和读后写（Load-Store）的重排序问题，建立跨线程的happens-before关系。

Release-Acquire模式

✓模式组成规则

✓Release Pattern的合法形式

组合方式	示例指令	同步范围
单一释放操作	st.release.global [X], 1	Release点之前的所有操作
栅栏 + 普通存储	fence.sc; st.relaxed [X], 1	fence前所有操作
原子释放操作	atom.acq_rel.global [X], %r0	原子操作前的所有操作

Release-Acquire模式

✓模式组成规则

✓Acquire Pattern的合法形式

组合方式	示例指令	同步范围
单一获取操作	ld.acquire.global %r0, [X]	Acquire点之后的所有操作
普通加载 + 栅栏	ld.relaxed [X]; fence.sc	fence后的所有操作
原子获取操作	atom.acq_rel.global [X], %r0	原子操作后的所有操作

Release-Acquire模式

- ✓同步原理与硬件行为
  - ✓同步触发条件
    - ✓当以下两条同时满足时，建立同步关系：
    - ✓写操作在Release Pattern内

```
// Thread A
st.release.global [X], data  // Release Pattern
```

Release-Acquire模式

- ✓同步原理与硬件行为
  - ✓同步触发条件
    - ✓读操作在Acquire Pattern内且读到Release的写入值

```
// Thread B
ld.acquire.global %r0, [X]   // Acquire Pattern
setp.eq.u32 %p, %r0, data    // 读到Thread A写入的值
```

Release-Acquire模式

- ✓同步原理与硬件行为
  - ✓硬件保证
    - ✓禁止重排序：
      - ✓Release前的操作 → Release操作 → Acquire操作 → Acquire后的操作
    - ✓缓存一致性：
      - ✓Release操作刷新写缓存（Write Buffer）
      - ✓Acquire操作失效读缓存（Invalidate Cache）

Release-Acquire模式

- ✓分层作用域控制
  - ✓作用域（Scope）决定同步的传播范围：

作用域	同步范围	延迟特性
.cta	同一线程块内线程	纳秒级（L1缓存）
.gpu	同一GPU设备内所有线程	微秒级（L2缓存）
.sys	全系统（GPU+CPU）	毫秒级（PCIe/NVLink）

  - ✓作用域匹配规则：Release与Acquire的作用域必须兼容（如 .gpu Release只能被 .gpu/.sys Acquire观测）

Release-Acquire模式

✓开发实践与出错规避

✓正确同步模板

```
// Thread A: 发布数据
st.relaxed.global [data_addr], data_value  // 写数据
st.release.cta [flag_addr], 1              // Release标记

// Thread B: 获取数据
loop:
   ld.acquire.cta %r0, [flag_addr]        // Acquire标记
   setp.ne.u32 %p, %r0, 1
   @%p bra loop
ld.relaxed.global %r1, [data_addr]        // 保证看到最新数据
```

Release-Acquire模式

✓开发实践与出错规避

✓典型错误与修复

错误模式	后果	修复方案
Release/Acquire作用域不匹配	同步失效	统一为 .gpu 或 .sys
未通过标记值触发同步	Acquire可能读到旧值	用独立标志变量作为触发器
混合弱操作（.weak）	同步链断裂	避免在同步链中用 .weak

Release-Acquire模式

✓开发实践与出错规避
  ✓性能优化技巧
    ✓作用域最小化：

```
// 原：全设备同步（高开销）
st.release.gpu [flag], 1
// 优化：限定线程块内
st.release.cta [flag], 1   // 延迟降低10-100倍
```

Release-Acquire模式

✓开发实践与出错规避
  ✓性能优化技巧
    ✓ 批量操作合并：

```
// 多次Release合并为一次
st.relaxed [data1], %r0
st.relaxed [data2], %r1
fence.release.cta          // 单次Release
st.relaxed [flag], 1
```

Release-Acquire模式

- ✓调试与验证工具
  - ✓cuda-memcheck：检测未配对的Release/Acquire操作

```
cuda-memcheck --tool synccheck ./app
```

- ✓Nsight Compute
  - ✓指标 l1tex__t_sectors：监测缓存失效开销
  - ✓指标 sm__pipe_alu_cycles_active：分析同步等待

Release-Acquire模式

- ✓总结
  - ✓Release-Acquire模式是跨线程内存可见性的黄金标准：
    - ✓核心机制：
      - ✓Release刷新写缓存 → Acquire失效读缓存
      - ✓通过标志变量传递同步信号
    - ✓三大铁律：
      - ✓作用域必须兼容
      - ✓必须通过独立标志触发
      - ✓禁止混用弱操作（.weak）

Release-Acquire模式

- ✓总结
  - ✓Release-Acquire模式是跨线程内存可见性的黄金标准：
    - ✓性能关键：
      - ✓用最小作用域（优先 .cta）
      - ✓合并多次操作为单次Release
    - ✓终极准则：对共享数据的跨线程传递，必须通过Release-Acquire对保护通过 nvcc -Xptxas -Werror=cross-scope-sync 编译检查作用域错误。

讲授内容

- ➢内存一致性模型（Memory Consistency Model）
- ➢内存一致性模型的适用范围与限制
- ➢内存操作的定义与核心机制
- ➢状态空间（State Spaces）
- ➢内存操作类型（Operation Types）
- ➢内存操作的作用域机制与分层同步控制（Scope）
- ➢Morally strong operations
- ➢Release-Acquire模式
- ➢内存操作顺序的层级约束机制（Ordering of memory operations）
- ➢内存一致性模型中的公理系统（Axioms）
- ➢内存模型中的边界场景处理（Special Cases）

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓内存顺序的核心层级
  - ✓PTX 内存模型通过三级顺序约束操作可见性：
    - ✓程序顺序（Program Order）
      - ✓定义：单线程内操作按指令流顺序执行
      - ✓约束范围：仅本线程（不约束其他线程）

```
st.global [X], 1    // 操作A
ld.global %r0, [Y]  // 操作B → 保证A在B前执行
```

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓内存顺序的核心层级
  - ✓PTX 内存模型通过三级顺序约束操作可见性：
    - ✓因果顺序（Causality Order）
      - ✓定义：通过同步操作建立跨线程 happens-before 关系
      - ✓传递性：若 A→B 且 B→C，则 A→C

```
// Thread 0
st.release.cta [flag], 1  // 操作A
// Thread 1
ld.acquire.cta %r0, [flag] // 操作B（A→B）
st.release.cta [data], 42  // 操作C（B→C）
// 则 A→C 成立
```

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓关键顺序规则
  - ✓程序顺序约束

操作类型	是否可重排序	示例
同地址写后读	❌	st [X] → ld [X]
同地址写后写	❌	st [X] → st [X]
同地址读后写	❌	ld [X] → st [X]
不同地址操作	√	st [X] → ld [Y]（可能重排）

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓关键顺序规则
  - ✓因果顺序强制约束
    - ✓若因果顺序 A→B，则：
      - ✓写操作可见性：A 的写入对 B 可见
      - ✓操作顺序保留：硬件/编译器禁止重排破坏 A→B 关系

```
graph LR
    A[写操作W] -->|st.release| B[同步点]
    B -->|ld.acquire| C[读操作R]
    D[其他操作] --> B
    B --> E[其他操作]
    // W的写入对R可见
```

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓关键顺序规则
  - ✓通信顺序运行时特性
  - ✓写操作提交顺序：由内存控制器决定
  - ✓读操作观测顺序：受缓存一致性协议影响

```
Thread0: st [X]=1 → st [Y]=1
Thread1:          ld [Y]=1 → ld [X]=0?  // 可能看到旧值!
// 需显式同步消除歧义
```

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓顺序性公理（Axioms）
  - ✓写操作原子性（Atomicity）
    - ✓对完全重叠的道德强操作：
      - ✓原子操作 vs 写操作 → 全执行或全不执行
      - ✓读操作 vs 写操作 → 不会看到中间状态
  - ✓顺序一致性（Sequential Consistency）
    - ✓对同一内存位置的道德强操作：
      - ✓所有线程观测到相同操作顺序

```
线程A: st [X]=1 → st [X]=2
线程B: 只能看到 1→2 或 2→1（全序一致）
```

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓顺序性公理（Axioms）
  - ✓无循环依赖（No Thin Air）
    - ✓禁止凭空产生值：

```
// 非法执行:
Thread0: if (y==1) x=1
Thread1: if (x==1) y=1
// 初始x=y=0 → 结果不能出现x=1且y=1
```

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓开发者控制策略
  - ✓程序顺序强化工具

机制	指令示例	作用
内存屏障	fence.sc	禁止前后操作重排序
依赖链	add.dep %r1, %r0, 1	强制数据依赖顺序
易失操作	ld.volatile	禁止编译器优化重排

内存操作顺序的层级约束机制（Ordering of memory operations）

✓开发者控制策略

✓因果顺序建立方法

```
// 跨线程同步链
Thread0:
   st.global [data], value
   fence.release.sys      // 建立释放点
Thread1:
   fence.acquire.sys      // 建立获取点
   ld.global %r0, [data]  // 保证看到Thread0的写入
```

内存操作顺序的层级约束机制（Ordering of memory operations）

✓开发者控制策略

✓通信顺序观测控制

✓写操作提交：fence.sc 强制刷新写缓存

```
st.global [X], 1
fence.sc.sys  // 确保写入全局可见
```

✓读操作刷新：ld.acquire 失效本地缓存

```
ld.acquire.global %r0, [X]  // 从全局内存加载最新值
```

内存操作顺序的层级约束机制（Ordering of memory operations）

✓调试与验证
  ✓顺序违规检测工具

```
cuda-memcheck --tool racecheck --print-level all
// 输出示例:
Witnessed Order:
  Thread(1,0,0) store [0x1000] = 1
  Thread(0,0,0) load [0x1000] = 0  // 违反顺序!
```

内存操作顺序的层级约束机制（Ordering of memory operations）

✓调试与验证
  ✓硬件计数器监测
    ✓指标 l1tex__t_sectors：L1缓存失效次数（反映缓存一致性开销）

```
nv-nsight-cu-cli --metrics "l1tex__t_sectors"
```

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓总结
  - ✓内存操作顺序的三级约束机制：
    - ✓程序顺序：线程内基础顺序 → 编译器/硬件可重排无关操作
    - ✓因果顺序：跨线程 happens-before 关系 → 通过同步操作显式建立
    - ✓通信顺序：运行时实际可见顺序 → 由缓存一致性协议决定

内存操作顺序的层级约束机制（Ordering of memory operations）

- ✓总结
  - ✓注意事项：
    - ✓对共享数据的跨线程访问必须用 release/acquire 或 fence 建立因果顺序
    - ✓同地址操作天然有序，异地址操作需显式依赖控制
    - ✓通过 fence.sc 强制全局顺序一致性（牺牲性能换取强保证）

```
// 强顺序一致性模式
st.global [X], 1
fence.sc.sys        // 全局内存屏障
st.global [Y], 2    // 保证所有线程先见X=1后见Y=2
```

讲授内容

- 内存一致性模型（Memory Consistency Model）
- 内存一致性模型的适用范围与限制
- 内存操作的定义与核心机制
- 状态空间（State Spaces）
- 内存操作类型（Operation Types）
- 内存操作的作用域机制与分层同步控制（Scope）
- Morally strong operations
- Release-Acquire模式
- 内存操作顺序的层级约束机制（Ordering of memory operations）
- 内存一致性模型中的公理系统（Axioms）
- 内存模型中的边界场景处理（Special Cases）

内存一致性模型中的公理系统（Axioms）

- ✓公理的本质与作用
  - ✓公理（Axioms） 是 PTX 内存一致性模型的基础数学规则，定义了内存操作的全局行为约束。这些公理不是可选的优化策略，而是硬件和编译器必须遵守的强制性规则，确保程序在多线程环境下的行为可预测且无矛盾。
  - ✓公理的核心作用包括：
    - ✓消除歧义：解决并发操作间的冲突和可见性问题。
    - ✓保证正确性：防止数据竞争、死锁和未定义行为。
    - ✓跨平台一致性：在不同 GPU 架构（如 sm_70 和 sm_80）上提供统一语义。
  - ✓公理 vs 指令：公理是底层规则（如“写操作必须有全局顺序”），而指令（如 fence）是实现公理的工具。

内存一致性模型中的公理系统（Axioms）

- ✓六大公理详解
  - ✓Coherence（连贯性）
    - ✓定义：对同一内存位置的写操作，所有线程观测到相同的全局顺序。
    - ✓作用：解决“写后写”冲突（如两个线程同时写同一地址）。
    - ✓违反后果：线程可能看到乱序写入（如 Thread2 看到 X=1 而 Thread3 看到 X=2）。
    - ✓示例：

```
// Thread0: st.global [X], 1
// Thread1: st.global [X], 2
// 所有线程看到 X 的值要么是 1→2，要么是 2→1（全局一致）
```

内存一致性模型中的公理系统（Axioms）

- ✓六大公理详解
  - ✓Fence-SC（Fence 顺序一致性）
    - ✓定义：fence.sc 操作（全系统内存屏障）的执行顺序必须匹配因果顺序。
    - ✓作用：确保屏障操作不破坏 happens-before 关系。
    - ✓硬件支持：通过刷新缓存和序列化内存请求实现。
    - ✓示例：

```
// Thread0: st.global [A], 1; fence.sc.sys; st.global [B], 2
// Thread1: ld.global [B]; fence.sc.sys; ld.global [A]
// 若 Thread1 读到 B=2，则必须读到 A=1（fence 强制顺序）
```

内存一致性模型中的公理系统（Axioms）

- ✓六大公理详解
  - ✓Atomicity（原子性）
    - ✓定义：对道德强操作（Morally Strong Operations） 的冲突访问：
      - ✓原子操作 vs 写操作 → 全执行或全不执行。
      - ✓读操作 vs 写操作 → 不会看到中间状态。
    - ✓作用：保障共享数据操作的完整性。
    - ✓示例：

```
// Thread0: atom.add.u32 [X], 1  // 原子加
// Thread1: st.u32 [X], 42       // 冲突写入
// 结果：要么原子操作生效（X=原值+1），要么写入生效（X=42），不会混合。
```

内存一致性模型中的公理系统（Axioms）

- ✓六大公理详解
  - ✓No Thin Air（无凭空值）
    - ✓定义：寄存器的值不能“凭空产生”，必须来自先前可见的写入或初始化。
    - ✓作用：防止循环依赖导致的非法值传播。
    - ✓合规方案：通过显式同步打破循环依赖。
    - ✓示例：

```
// 非法场景：
Thread0: if (y==1) x=1
Thread1: if (x==1) y=1
// 初始 x=y=0 → 结果不能出现 x=1 且 y=1（违反公理）
```

内存一致性模型中的公理系统（Axioms）

- ✓六大公理详解
  - ✓Sequential Consistency Per Location（每个位置的顺序一致性）
    - ✓定义：对同一内存位置的道德强操作，所有线程观测到相同的操作序列。
    - ✓作用：简化对单个变量的并发推理。
    - ✓示例：

```
// Thread0: st [X], 1 → st [X], 2
// Thread1: st [X], 3
// 所有线程看到 X 的写入序列为全序（如 1→3→2 或 3→1→2）
```

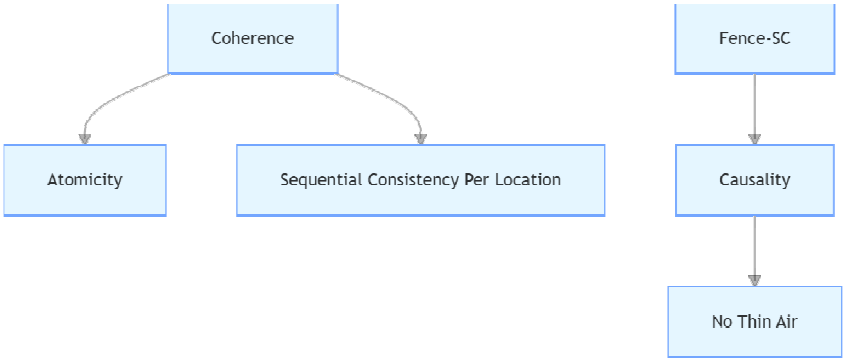
内存一致性模型中的公理系统（Axioms）

- ✓六大公理详解
  - ✓Causality（因果性）
    - ✓定义：若操作 A happens-before 操作 B，则 A 的副作用必须对 B 可见。
    - ✓作用：确保同步操作（如 release/acquire）传递数据。
    - ✓ 示例：

```
// Thread0: st.release [flag], 1  // A
// Thread1: ld.acquire [flag]     // B（读到 1）
// Thread1: ld [data]             // C → 必须看到 Thread0 在 A 前的写入
```

内存一致性模型中的公理系统（Axioms）

- ✓公理间的交互与层级
  - ✓公理依赖关系
    - ✓基础层：Coherence 和 Causality 确保操作顺序。
    - ✓增强层：Atomicity 和 No Thin Air 防止数据损坏。
    - ✓最高层：Fence-SC 和 Sequential Consistency 提供全局保证。



内存一致性模型中的公理系统（Axioms）

- ✓公理间的交互与层级
  - ✓违反公理的后果
    - ✓硬件未定义行为：结果可能为任意值（e.g., 读取到撕裂值）。
    - ✓编译器优化失效：循环或条件分支可能被错误优化。
    - ✓调试工具检测：cuda-memcheck --tool synccheck 可识别部分违规。

内存一致性模型中的公理系统（Axioms）

✓开发者实践指南
  ✓公理合规策略

公理	合规方法	指令示例
Coherence	对共享变量使用原子操作	atom.add.u32 [X], 1
No Thin Air	避免未初始化的数据依赖	mov.u32 %r0, 0（显式初始化）
Causality	用 Release-Acquire 对建立顺序	st.release → ld.acquire

内存一致性模型中的公理系统（Axioms）

✓开发者实践指南
  ✓公理合规策略

公理	合规方法	指令示例
Coherence	对共享变量使用原子操作	atom.add.u32 [X], 1
No Thin Air	避免未初始化的数据依赖	mov.u32 %r0, 0（显式初始化）
Causality	用 Release-Acquire 对建立顺序	st.release → ld.acquire

内存一致性模型中的公理系统（Axioms）

- ✓开发者实践指南
  - ✓调试与验证工具
    - ✓Nsight Compute：
      - ✓指标 l1tex__t_sectors：监测缓存失效（反映 Coherence 开销）。
      - ✓指标 sm__pipe_alu_cycles_active：分析原子操作延迟（Atomicity 成本）。
    - ✓cuda-memcheck：

```
cuda-memcheck --tool racecheck  # 检测 Coherence 和 Atomicity 违规
```

内存一致性模型中的公理系统（Axioms）

- ✓开发者实践指南
  - ✓常见错误案例
    - ✓违反 No Thin Air：

```
// ❌错误：未初始化导致循环依赖
ld.global %r0, [X]
add.u32 %r1, %r0, 1
st.global [X], %r1  // 若 X 未初始化，%r0 可能为“凭空”值
```

- ✓修复：

```
mov.u32 %r0, 0     // 显式初始化
st.global [X], %r0  // 确保写入初始值
```

内存一致性模型中的公理系统（Axioms）

- ✓公理系统的重要性总结
  - ✓公理是内存模型的数学根基，确保：
    - ✓可预测性：多线程程序行为与开发者预期一致。
    - ✓可移植性：代码在 sm_70 到 sm_90 等架构间行为一致。
    - ✓可调优性：开发者可在公理约束下优化性能（如缩小作用域）。
  - ✓终极准则：所有并发访问必须通过同步操作（如原子指令或 fence）满足公理约束。
  - ✓编译时启用 nvcc -Xptxas -Werror=axiom-violation 可检测潜在违规。

内存一致性模型中的公理系统（Axioms）

- ✓内存模型中的边界场景处理（Special Cases）
  - ✓零长度操作（Zero-Length Operations）
    - ✓定义：访问长度为 0 的内存区域（如 .b0 类型操作）。行为：
      - ✓无实际内存访问：不读取/写入任何数据
      - ✓顺序性保留：在程序顺序中仍占位（可能影响重排序）
    - ✓作用：用于占位或调试，无实际内存影响。

```
st.b0 [addr]  // 零长度存储（无效果但占用程序顺序）
ld.b0 %r0, [addr] // 零长度加载 → %r0 值未定义
```

讲授内容

- 内存一致性模型（Memory Consistency Model）
- 内存一致性模型的适用范围与限制
- 内存操作的定义与核心机制
- 状态空间（State Spaces）
- 内存操作类型（Operation Types）
- 内存操作的作用域机制与分层同步控制（Scope）
- Morally strong operations
- Release-Acquire模式
- 内存操作顺序的层级约束机制（Ordering of memory operations）
- 内存一致性模型中的公理系统（Axioms）
- 内存模型中的边界场景处理（Special Cases）

内存模型中的边界场景处理（Special Cases）

- ✓未初始化内存访问
  - ✓场景：读取从未写入过的内存位置。规则：
    - ✓返回值未定义：可能为任意值（包括陷阱值）
    - ✓不触发异常：除非访问越界（由地址检查独立处理）
  - ✓风险：
    - ✓若后续操作依赖该值 → 未定义行为
    - ✓解决方案：显式初始化或避免访问

```
ld.u32 %r0, [uninitialized_addr]
// %r0 可能为 0、旧数据或随机值
```

内存模型中的边界场景处理（Special Cases）

- ✓地址计算操作
  - ✓定义：计算内存地址但不执行访问（如 mov.u64 %ptr, addr）。
  - ✓特性：
    - ✓无内存交互：不参与内存一致性模型约束
    - ✓顺序自由：编译器/硬件可重排（除非数据依赖）

```
add.u64 %addr, %base, %offset  // 地址计算
st.global [%addr], %value      // 实际存储
// 地址计算可能被重排到存储之后
```

内存模型中的边界场景处理（Special Cases）

- ✓地址计算操作
  - ✓定义：计算内存地址但不执行访问（如 mov.u64 %ptr, addr）。
  - ✓约束：
    - ✓仅当存在数据依赖时保持顺序：

```
ld.shared %offset, [shared_addr]
add.u64 %addr, %base, %offset  // 依赖 %offset → 不可重排
st.global [%addr], %value
```

内存模型中的边界场景处理（Special Cases）

- ✓非冲突访问（Non-Conflicting Accesses）
  - ✓场景：多个操作访问无重叠的内存区域。规则：
    - ✓无同步要求：操作可任意重排序
    - ✓硬件优化自由：并行执行或乱序提交

```
// 无重叠地址 → 自由重排
st.global [X], 1    // 地址 A
st.global [Y], 2    // 地址 B（|A-B|>4）
```

内存模型中的边界场景处理（Special Cases）

- ✓非冲突访问（Non-Conflicting Accesses）
  - ✓例外：若操作含同步语义（如 release/acquire）

```
st.release.global [X], 1  // 同步操作
st.relaxed.global [Y], 2  // 可能被重排到 release 前
```

内存模型中的边界场景处理（Special Cases）

- ✓常量内存（Constant Memory）的特殊性
  - ✓特性：
    - ✓只读性：.const 空间在运行时不可写
    - ✓初始化要求：必须由主机初始化访问行为：
      - ✓读取未初始化区域 → 未定义值（同未初始化内存）
      - ✓写入尝试 → 编译错误或运行时陷阱
  - ✓优化提示：常量内存访问可被缓存，无一致性开销。

```
ld.const.u32 %r0, [const_addr]  // 合法
st.const.u32 [const_addr], 1    // ✗非法!
```

内存模型中的边界场景处理（Special Cases）

✓特殊操作汇总表

场景	关键规则	开发者注意事项
零长度操作	无内存影响，保留程序顺序	避免用于功能逻辑
未初始化内存读取	返回值未定义，不触发异常	必须显式初始化内存
地址计算	无内存交互，可自由重排	依赖链需显式控制顺序
非冲突访问	无同步要求，可并行执行	同步操作仍受约束
常量内存访问	只读，主机初始化，运行时不可写	禁止写入，缓存优化优先

内存模型中的边界场景处理（Special Cases）

✓开发者应对策略
- ✓零长度操作：仅用于调试占位，生产代码移除
- ✓未初始化内存：

```
// 安全初始化
mov.u32 %r0, 0
st.global [data], %r0  // 显式清零
```

内存模型中的边界场景处理（Special Cases）

✓开发者应对策略
- ✓地址计算重排：

```
// 添加伪依赖强制顺序
add.u64 %addr, %base, %offset
setp.ne.u64 %p, %addr, 0  // 伪条件
@%p st.global [%addr], %value
```

内存模型中的边界场景处理（Special Cases）

- ✓开发者应对策略
  - ✓常量内存优化：

```
ld.const.f32 %f0, [table+index*4]  // 常量表访问
// 利用硬件常量缓存（L1）加速
```

内存模型中的边界场景处理（Special Cases）

- ✓总结
  - ✓特殊场景揭示了内存模型的边界行为：
    - ✓零操作与未初始化内存：需显式规避 → 保证结果确定性
    - ✓地址计算与非冲突访问：理解重排自由 → 平衡性能与正确性
    - ✓常量内存：只读特性 → 专注访问优化
  - ✓终极实践：对非常规操作（零长度/未初始化访问）保持警惕，通过编译选项 -Xptxas -Werror=uninitialized 检测潜在风险。常量内存数据布局优先考虑空间局部性以提升缓存效率。


THANKS