中国科学院大学计算机学院专业选修课

GPU架构与编程

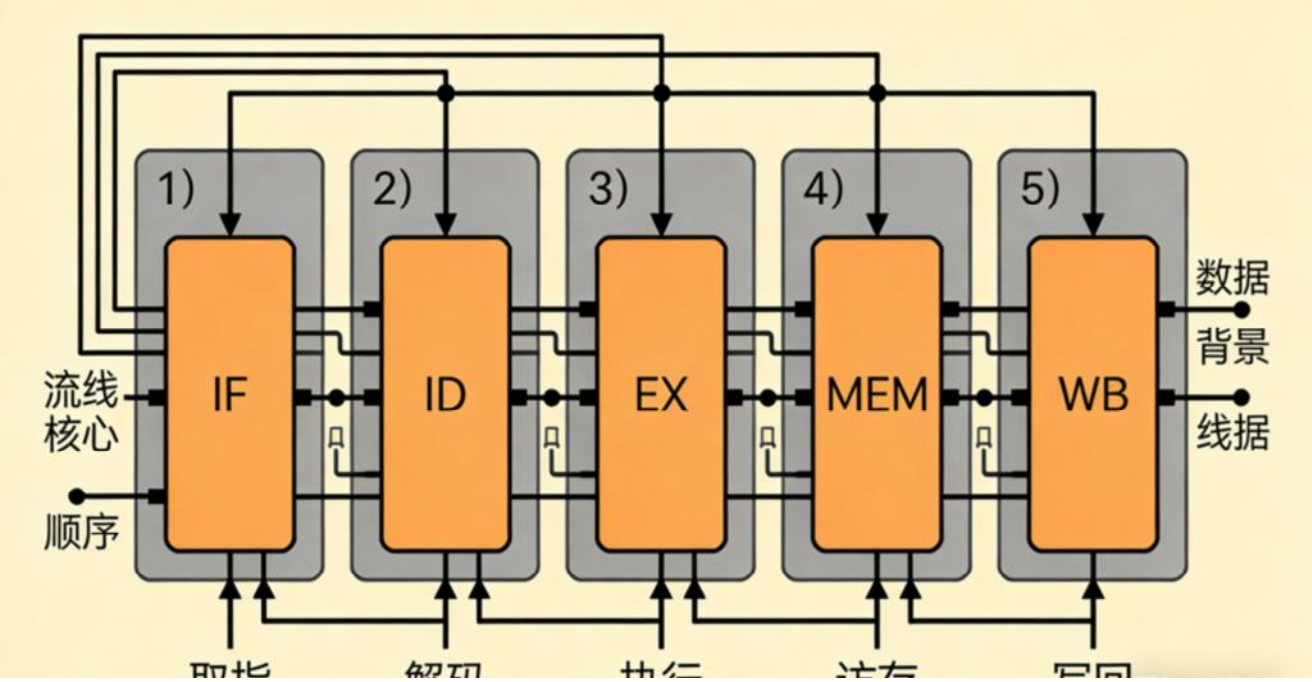
第十三课：GPU架构简介（二）

赵地
中科院计算所
2025年秋季学期

讲授内容:The SIMT Core

- One-Loop Approximation
  - Introduction
  - SIMT Stack: SIMT Execution Mask
  - SIMT Deadlock and Stackless SIMT Architectures
  - Warp Scheduling
- Two-Loop Approximation
  - Scoreboard
- Three-Loop Approximation
  - Operand Collector
  - Instruction Replay: Handling Structural Hazards

单指令顺序核CPU流水线



One-Loop Approximation: Introduction

✓指令获取：warp的“程序计数器”会访问指令存储器，找到该warp接下来要执行的指令。

✓指令解码与操作数读取：获取指令后，先对指令进行解码，再从“寄存器文件”中取出该指令需要的“源操作数寄存器”。

✓并行处理：确定SIMT 执行掩码确定，在读取源操作数的同时，会同步确定 SIMT的执行掩码值；这个掩码用于控制warp中哪些线程实际执行当前指令。

Tor M. Aamodt , Wilson Wai Lun Fung , Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018.

One-Loop Approximation: Introduction

✓执行方式：当“执行掩码”和“源寄存器”都准备就绪后，指令会以“单指令多数据（SIMD）”的方式推进执行。

✓线程的执行条件：每个线程会在与“通道（lane）”关联的功能单元（硬件运算模块）上运行，但只有当“SIMT执行掩码”被激活（对应线程的掩码位处于有效状态）时，该线程才会实际执行当前指令。

Tor M. Aamodt , Wilson Wai Lun Fung , Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018.

One-Loop Approximation: Introduction

✓功能单元的异构性

- ✓与现代 CPU 架构类似，GPU 的功能单元通常是“异构”：即每个功能单元仅支持特定子集的指令，而非通用所有指令。这种设计的目的是让不同类型的运算（如计算、存储、特殊数学运算）匹配专用硬件单元，提升执行效率。

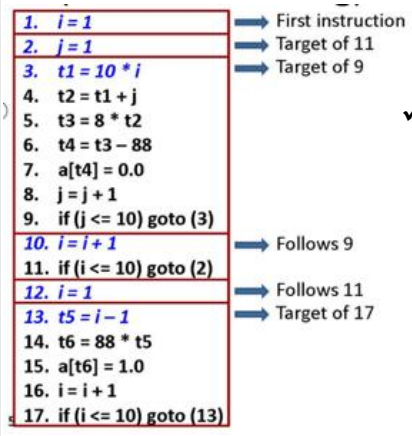
✓NVIDIA GPU 的功能单元：

- ✓特殊功能单元（SFU）：处理三角函数、对数等特殊数学运算；
- ✓加载 / 存储单元：负责内存数据的读写操作；
- ✓浮点功能单元：执行浮点运算；
- ✓整数功能单元：处理整数运算；
- ✓张量核心（Tensor Core）：自Volta架构起引入，专门加速矩阵运算，支撑深度学习等场景的高效计算。

One-Loop Approximation: Control Flow

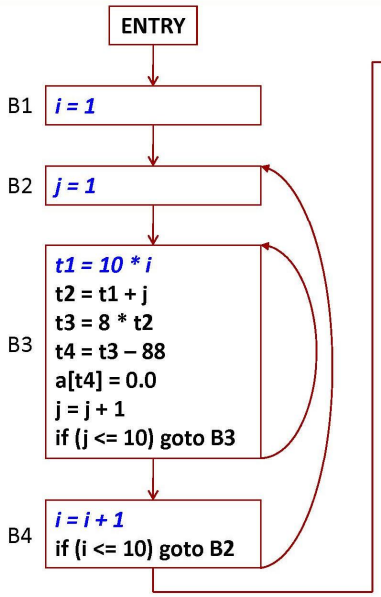
- ✓控制流图（CFG）的构建基础
  - ✓基本块（Basic block）：“连续指令的最大序列”，需满足两个控制流约束：
    - ✓控制流只能从块的第一个指令进入（不能跳转到块的中间位置）；
    - ✓控制流只能从块的最后一个指令退出。
    - ✓（注：“三地址指令” 是编译器常用的中间代码形式，每条指令通常包含一个操作、两个操作数和一个结果）
  - ✓Leader：基本块的起始指令。在给定完整指令序列的前提下，先找到所有基本块的起始指令（即 Leader），包含三类：
    - ✓整个指令序列的第一个指令；
    - ✓任何条件 / 无条件跳转指令的目标指令；
    - ✓紧跟在条件 / 无条件跳转指令之后的指令。

One-Loop Approximation: Control Flow



- ✓ 基本块的划分规则：在确定所有 “Leader（基本块起始指令）” 后，每个 Leader 对应的基本块包含：
  - ✓ 该 Leader 自身；
  - ✓ 从该 Leader 开始，到 下一个 Leader 之前（但不包含下一个 Leader）的所有指令。
- ✓ 示例解析：左侧展示了一段指令序列，结合 Leader 标注与功能说明：
  - ✓ **Leader 标注**：指令 1（第一个指令）、2（指令 11 的跳转目标）、3（指令 9 的跳转目标）、10（紧跟指令 9 之后）、12（紧跟指令 11 之后）、13（指令 17 的跳转目标）均为 Leader。
  - ✓ **示例功能**：这段指令实现两个逻辑：
    - ✓ 将数组a[i][j]（满足$1 \le i,j \le 10$）的元素设为 0.0（对应指令 1-11）；
    - ✓ 将数组a[i][i]（满足$1 \le i \le 10$）的元素设为 1.0（对应指令 12-17）。
  - ✓ **数组访问说明**：指令 7、15 的数组访问，是按 “行主序、元素占 8 字节、数组索引从 1 开始” 的规则计算偏移量的。

One-Loop Approximation: Control Flow



Artificial ENTRY and EXIT nodes are often added for convenience.

There is an edge from B_p to B_q if it is possible for the first instruction of B_q to be executed immediately after the last instruction of B_p. This is conservative: e.g., **if (3.14 > 2.78)** still generates two edges.

One-Loop Approximation: Control Flow

- ✓ CFG 的结构组成
  - ✓ **节点**：以之前划分的基本块（如 B1-B6）作为 CFG 的节点，同时补充了人工的ENTRY（入口）和EXIT（出口）节点；这是为了统一表示程序的开始与结束，提升 CFG 的规范性。
  - ✓ **控制流边**：通过有向边连接不同基本块，体现指令执行的跳转关系（例如 B3 执行后，会跳回B3或进入B4；B6执行后会跳回B6或进入 EXIT）。
- ✓ CFG 中“边”的定义规则
  - ✓ 若基本块B_p的最后一条指令执行后，基本块Bq的第一条指令能**立即执行**，则在B_p与Bq之间建立一条边。
  - ✓ 该规则是“保守性”的：即使是恒成立 / 恒不成立的条件判断（比如if (3.14 > 2.78)），也会生成两条边（对应条件成立、不成立的分支），以覆盖所有理论上的执行路径。

One-Loop Approximation: Control Flow

- ✓ CFG 的图数据结构选择
  - ✓ 由于 CFG 属于稀疏图（边的数量相对较少），通常采用邻接表来表示；
  - ✓ 边数的上限为 “节点数的 2 倍”（因为每个基本块最多对应条件分支的 2 条出边），这也符合稀疏图的特征。
- ✓ CFG 的节点与边的管理规则
  - ✓ 节点与边的层级：CFG 的节点是 “基本块”，边连接的是基本块（而非单个指令）；
  - ✓ 节点内部的指令存储：每个基本块节点内部，需用额外数据结构（如 “指令链表”）管理块内的指令序列；
  - ✓ 块的邻接关系维护：通常会为每个基本块同时维护 “后继列表（出边对应的块）” 和 “前驱列表（入边对应的块）”，便于快速查询控制流的上下游关系。

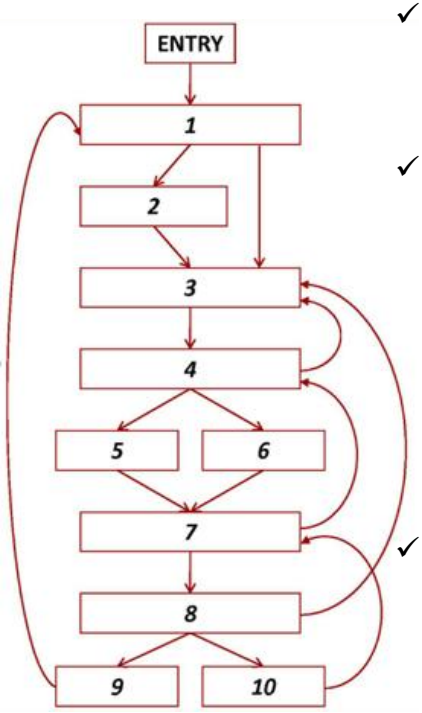
Atanas (Nasko) Rountev, Control-Flow Analysis, OSU CSE;

One-Loop Approximation: Control Flow

- ✓ **支配节点（Dominating Node）的定义**
  - ✓ CFG 中的节点d支配节点n（记为d dom n），当且仅当 “从ENTRY节点到n的所有路径，都必须经过d”。相关特性：
    - ✓ 隐含假设：每个节点都能从 ENTRY 到达（无死代码）；
    - ✓ 支配关系是自反的：任何节点d都支配自身（d dom d）。
- ✓ **直接支配节点（Immediate Dominator）**
  - ✓ 定义：节点m是n的直接支配节点，需满足 3 个条件：
    - ✓ $m \neq n$；
    - ✓ m dom n；
    - ✓ 对于任意支配n且不等于n的节点d，d都支配m。（即：直接支配节点是 “离n最近的、非自身的支配节点”）
  - ✓ 直接支配节点的唯一性
    - ✓ 每个节点都有唯一的直接支配节点，仅 ENTRY 节点例外；ENTRY 节点仅被自身支配，无其他直接支配节点。

Atanas (Nasko) Rountev, Control-Flow Analysis, OSU CSE;

One-Loop Approximation: Control Flow

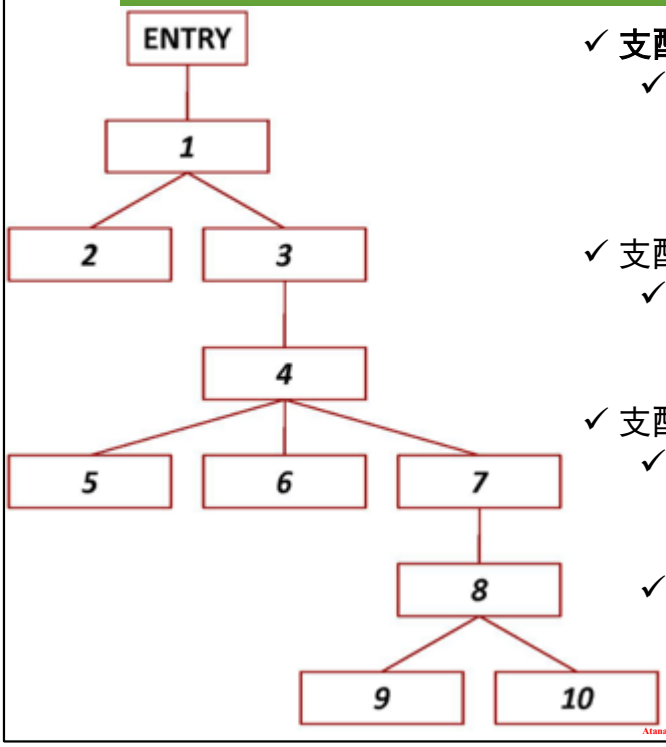


- ✓ 示例 CFG：
  - ✓ 无 EXIT 节点；
  - ✓ 节点 4、8 的出边数量超过 2 条（通常 CFG 节点出边最多 2 条，对应条件分支的“是 / 否”路径）。
- ✓ 支配关系（dom）说明
  - ✓ 各节点的支配范围：
    - ✓ ENTRY支配所有节点n；
    - ✓ 节点 1 支配除ENTRY外的所有节点n；
    - ✓ 节点 2 不支配任何其他节点；
    - ✓ 节点 3 支配节点 3、4、5、6、7、8、9、10；
    - ✓ 节点 4 支配节点 4、5、6、7、8、9、10；
    - ✓ 节点 5、6、9、10 不支配任何其他节点；
    - ✓ 节点 7 支配节点 7、8、9、10；
    - ✓ 节点 8 支配节点 8、9、10。
- ✓ 直接支配节点对应关系
  - ✓ 每个节点的唯一直接支配节点：
  - ✓ 节点 1 → ENTRY，节点 2 → 节点 1，节点 3 → 节点 1，节点 4 → 节点 3，节点 5 → 节点 4，节点 6 → 节点 4，节点 7 → 节点 4，节点 8 → 节点 7，节点 9 → 节点 8，节点 10 → 节点 8

One-Loop Approximation: Control Flow

- ✓ 支配关系的核心特性总结：
  - ✓支配者在无环路径中的出现规律
    - ✓对于从 ENTRY 到节点*n*的任意无环路径，*n*的所有支配者都会出现在这条路径上，且在所有这类路径中，支配者的出现顺序完全一致。
  - ✓支配关系的传递性
    - ✓支配关系是传递关系：若节点*a*支配*b*，且*b*支配*c*，则*a*必然支配*c*。
  - ✓支配关系的反对称性与偏序属性
    - ✓支配关系是反对称关系：若*a*支配*b*且*b*支配*a*，则*a*与*b*是同一个节点。结合支配关系的“自反性”（节点支配自身），支配关系满足“自反、反对称、传递”三大性质，因此是一种偏序关系。
  - ✓同一节点支配者的互斥性
    - ✓若*a*和*b*都是某节点n的支配者，则*a*与*b*之间必然存在支配关系： 要么*a*支配*b*，要么*b*支配*a*。

One-Loop Approximation: Control Flow

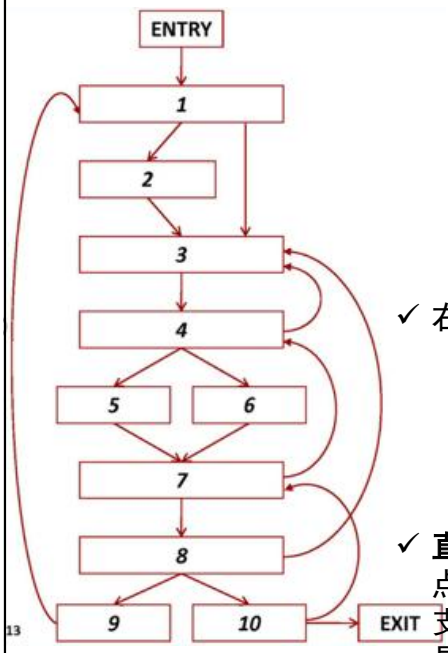


- ✓ **支配树（Dominator Tree）的结构规则**
  - ✓ 支配树中，节点n的父节点就是它的直接支配节点，树的根节点为ENTRY（对应之前示例的节点关系：如节点 1 的父节点是 ENTRY，节点 2/3 的父节点是 1，节点 4 的父节点是 3 等）。
- ✓ 支配树的核心特性
  - ✓ 从任意节点n到根节点（ENTRY）的路径，包含且仅包含n的所有支配者：这让支配关系可以通过树的路径直观体现。
- ✓ 支配树的构造算法
  - ✓ 经典算法：由 T. Lengauer 和 R. E. Tarjan 于 1979 年提出，发表于《ACM Transactions on Programming Languages and Systems》。
  - ✓ 其他算法：例如 K. D. Cooper 等人 2001 年提出的 “简单快速支配算法”，发表于《Software – Practice and Experience》。

One-Loop Approximation: Control Flow

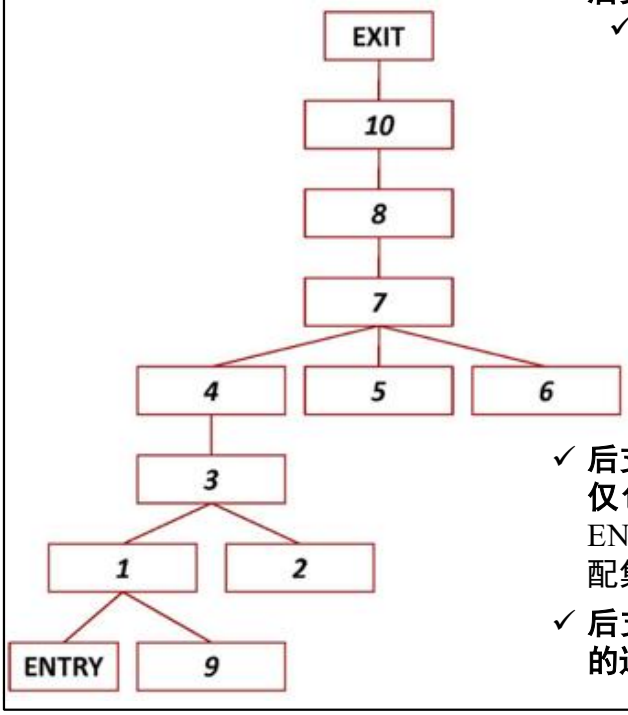
- ✓ **后支配（Post-Dominance）的定义**
  - ✓ CFG 中的节点d后支配节点n（记为d pdpm n），当且仅当 “从n到 EXIT 的所有路径，都必须经过d”。相关特性：
    - ✓ 隐含假设：每个节点都能到达 EXIT；
    - ✓ 后支配关系是自反的：任意节点d都后支配自身（d pdpm d）。
- ✓ **直接后支配（Immediate post-dominators）的定义**
  - ✓ 节点m是n的直接后支配节点，需满足：
    - ✓ $m \neq n$；
    - ✓ m后支配n；
    - ✓ 对于任意后支配n且不等于n的节点d，d都后支配m。
    - ✓ 每个节点都有唯一的直接后支配节点。
- ✓ 后支配与支配的关系
  - ✓ CFG 上的后支配关系，等价于反转 CFG（所有边反向）” 上的支配关系：即把控制流图的边方向颠倒后，后支配就对应原 CFG 的支配。
- ✓ 后支配树的结构规则：
  - ✓ 节点n的父节点是它的直接后支配节点；
  - ✓ 树的根节点是 EXIT（对应支配树的根是 ENTRY）。

One-Loop Approximation: Control Flow



- ✓ 左侧：控制流图（CFG）
  - ✓ 矩形代表基本块（编号 1-10），箭头是控制流的转移方向：
    - ✓ 入口是ENTRY，指向基本块 1；
    - ✓ 块 1 的控制流分向块 2、块 3；
    - ✓ 块 2→块 3；块 3→块 4；
    - ✓ 块 4 分向块 5、块 6；块 5、6→块 7；
    - ✓ 块 7→块 8；块 8 分向块 9、10；
    - ✓ 块 9回跳至块 1（形成循环）；块 10→EXIT（出口）；
    - ✓ 还有其他回边（如块 3、4、7、8 的跳转），体现循环结构。
- ✓ 右侧：**后支配（Post-dominance）**分析
  - ✓ “后支配（pdom）”指：若从节点 B 出发的所有路径都必须经过节点 A，则 A 后支配 B。
  - ✓ 各节点的后支配集：比如 “3 pdom ENTRY, 1, 2, 3, 9”，表示节点 3 后支配 ENTRY、1、2、3、9（这些节点的所有路径都经过 3）；最终EXIT后支配所有节点（所有路径的终点都是 EXIT）。
- ✓ **直接后支配（Immediate post-dominators）**：是后支配集中最接近该节点的节点（相当于 “直接后支配者”），比如：ENTRY 的直接后支配是 1；1 的直接后支配是 3；9 的直接后支配是 1（因为 9 的路径会回到 1）；10 的直接后支配是 EXIT。

One-Loop Approximation: Control Flow



- ✓ **后支配树（Post-Dominator Tree）**的结构与含义
  - ✓ **后支配树的根节点是 EXIT**（因为 EXIT 后支配所有节点），父子节点对应 “直接后支配” 关系（子节点的直接后支配者是父节点）：
    - ✓ 10 的父节点是 EXIT（10 的直接后支配是 EXIT）；
    - ✓ 8 的父节点是 10（8 的直接后支配是 10）；
    - ✓ 7 的父节点是 8（7 的直接后支配是 8）；
    - ✓ 4、5、6 的父节点是 7（这三个的直接后支配是 7）；
    - ✓ 3 的父节点是 4（3 的直接后支配是 4）；
    - ✓ 1、2 的父节点是 3（1、2 的直接后支配是 3）；
    - ✓ ENTRY、9 的父节点是 1（ENTRY、9 的直接后支配是 1）。
- ✓ **后支配树**的核心作用：**“从节点 *n* 到根（EXIT）的路径，包含且仅包含 *n* 的所有后支配者”**。例如，节点 ENTRY 的路径是 ENTRY→1→3→4→7→8→10→EXIT，这正好是 ENTRY 的后支配集（和之前控制流图的后支配分析结果一致）。
- ✓ **后支配树**的构建方法：**复用支配树的构建算法，只需把原控制流的边 “反向” 即可**。

讲授内容:The SIMT Core

- One-Loop Approximation
  - Introduction
  - SIMT Stack: SIMT Execution Mask
  - SIMT Deadlock and Stackless SIMT Architectures
  - Warp Scheduling
- Two-Loop Approximation
  - Scoreboard
- Three-Loop Approximation
  - Operand Collector
  - Instruction Replay: Handling Structural Hazards

SIMT Execution Mask

✓SIMT Stack的核心价值：

✓能高效处理 “线程独立执行” 时的两个关键问题：

✓问题 1、嵌套控制流指控制流的分支之间存在依赖关系（一个分支的执行由另一个分支决定），SIMT Stack可应对这种复杂的控制流逻辑。

✓问题 2、当warp里的所有线程都不执行某条控制流路径时，SIMT Stack会直接跳过该路径的计算。

✓实际优势：对于复杂控制流场景，能大幅节省计算资源。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Execution Mask

✓CUDA C code that contains two branches nested within a do-while loop:

```
1   do {
2     t1 = tid*N;        // A
3     t2 = t1 + i;
4     t3 = data1[t2];
5     t4 = 0;
6     if( t3 != t4 ) {
7       t5 = data2[t2]; // B
8       if( t5 != t4 ) {
9         x += 1;        // C
10      } else {
11        y += 2;        // D
12      }
13    } else {
14      z += 3;          // F
15    }
16    i++;               // G
17  } while( i < N );
```

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

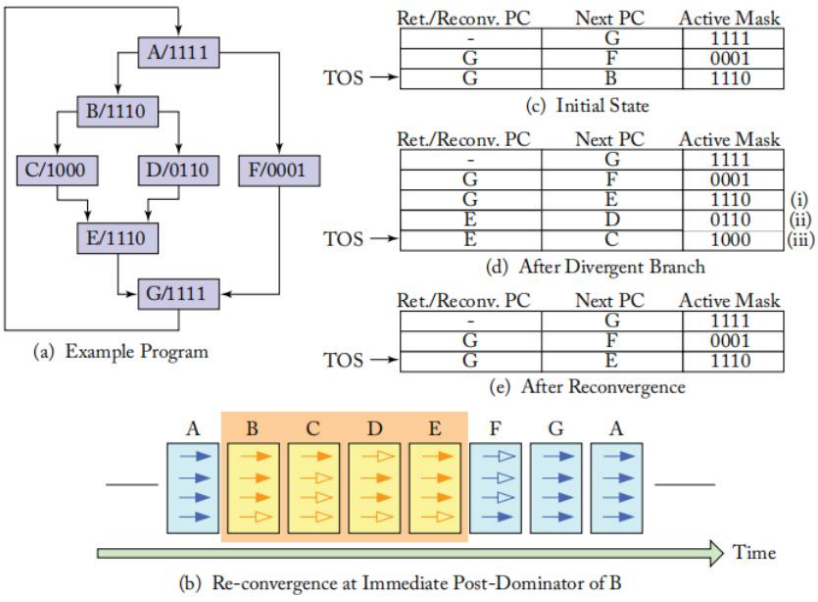
SIMT Execution Mask

✓PTX assembly code for illustrating SIMT stack operation:

	代码操作	逻辑说明
A	mul.lo.u32 t1, tid, N add.u32 t2, t1, i ld.global.u32 t3, [t2]	计算内存地址：t1=线程ID×N → t2=t1+i（i是循环变量），再从全局内存加载t3=[t2]
-	mov.u32 t4, 0 setp.eq.u32 p1, t3, t4	初始化t4=0，设置谓词p1：若t3==0，p1=真；否则p1=假
-	@p1 bra F	执行掩码控制分支：p1为真的线程跳转到F；p1为假的线程继续执行B
B	ld.global.u32 t5, [t2] setp.eq.u32 p2, t5, t4	再加载内存t5=[t2]，设置谓词p2：若t5==0，p2=真
-	@p2 bra D	掩码分支：p2为真的线程跳转到D；p2为假的线程执行C
C	add.u32 x, x, 1 bra E	x自增1，然后跳转到E
D	add.u32 y, y, 2	y自增2（跳转到此处的线程执行）
E	bra G	跳转到G，收敛分支
F	add.u32 z, z, 3	z自增3（跳转到此处的线程执行）
G	add.u32 i, i, 1 setp.le.u32 p3, i, N	i自增1，设置谓词p3：若i≤N，p3=真
-	@p3 bra A	掩码循环：p3为真的线程跳回A继续循环；p3为假的线程退出

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Execution Mask



✓Example of SIMT stack operation (based on Figure 5 from Fung et al. [2007]), illustrates how this code interacts with the SIMT stack assuming a GPU that has four threads per warp.

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Execution Mask

✓“线程重汇聚”的规则：

- ✓重汇聚点（reconvergence point）：程序里的一个位置：当warp里的线程因分支（比如if-else）出现执行路径发散时，能在这里被强制回到锁步（lock-step）执行（即所有线程同步执行同一条指令）。
- ✓重汇聚点的选择原则：通常优先选最近的重汇聚点（这样能让线程尽早恢复同步，减少执行效率损耗）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Execution Mask

✓“线程重汇聚” 的规则：

✓重汇聚点的确定方式：在编译阶段就能确定；导致线程分支发散的那个分支的直接后支配点（immediate post-dominator），就是线程能再次锁步执行的 “最早位置”。

✓GPU处理warp内部分支发散”的机制：通过 “重汇聚点（通常是分支的直接后支配点）” 让发散的线程重新同步，保证 SIMT 的执行效率。

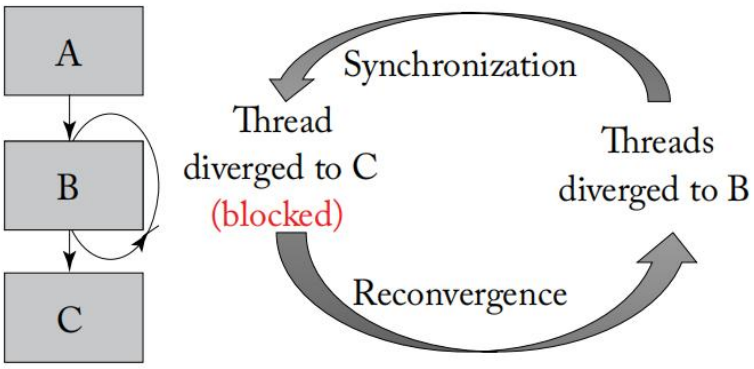
Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

讲授内容:The SIMT Core

- One-Loop Approximation
  - Introduction
  - SIMT Stack: SIMT Execution Mask
  - SIMT Deadlock and Stackless SIMT Architectures
  - Warp Scheduling
- Two-Loop Approximation
  - Scoreboard
- Three-Loop Approximation
  - Operand Collector
  - Instruction Replay: Handling Structural Hazards

SIMT Deadlock and Stackless SIMT Architectures: SIMT deadlock example

```
A: *mutex = 0
B: while(!atomicCAS(mutex, 0  ,1));
C: // critical section
    atomicExch(mutex, 0  ) ;
```



✓SIMT deadlock example (based on Figure 1 from ElTantawy and Aamodt [2016]).

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures: SIMT deadlock example

✓互斥锁（mutex）的初始化（Line A）

- ✓将共享变量mutex初始化为 0，以此标识 “锁处于空闲状态”，可供线程尝试获取。

✓atomicCAS操作（Line B）

- ✓warp中的每个线程，都会对mutex所在的内存地址执行atomicCAS（比较并交换）操作；
- ✓这是一种原子操作：能保证多线程环境下，对共享变量的读写不会被其他线程打断，避免数据竞争。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures: SIMT deadlock example

- ✓atomicCAS 的编译映射
  - ✓atomicCAS是编译器内建函数，最终会被翻译为 GPU 汇编指令（PTX）中的atom.global.cas指令，由 GPU 硬件直接支持。
- ✓SIMT死锁场景：线程通过atomicCAS竞争互斥锁，但由于 SIMT 的 “warp锁步执行” 特性，若多个线程同时抢锁，可能导致warp内部分线程持续等待（无法推进执行），最终引发死锁。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures: SIMT deadlock example

- ✓atomicCAS的逻辑执行步骤，atomicCAS是一个原子操作（不可被中断），其逻辑流程为：
  - ✓首先读取共享变量mutex（互斥锁）的当前内容；
  - ✓然后将读取到的内容与 “第二个输入参数（这里是0）” 进行比较。
- ✓比较后的更新逻辑，atomicCAS的第三个输入参数是1，对应互斥锁的 “占用态”：
  - ✓若mutex当前值等于第二个输入（即0，代表锁处于空闲态），则将mutex的值更新为第三个输入（即1，代表锁被占用）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures: SIMT deadlock example

- ✓atomicCAS的返回值规则：atomicCAS的返回结果是mutex的原始值（即操作执行前mutex的状态）：
  - ✓若返回值为0：说明操作前锁是空闲态，当前线程成功将锁设为占用态（加锁成功）；
  - ✓若返回值为1：说明操作前锁已被占用，当前线程加锁失败（需进入自旋等待）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures: SIMT deadlock example

- ✓该操作与 SIMT 死锁的关联，在 GPU 的 SIMT 架构中，warp内的32个线程会锁步执行atomicCAS：
  - ✓只有 1 个线程能成功获取锁（返回0），其余线程会返回1并进入自旋循环；
  - ✓由于 SIMT 的 “锁步执行” 特性，整个warp会同步执行自旋逻辑，持续占用计算资源且无法推进，最终可能引发死锁。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓基于无栈收敛屏障（stack-less convergence barrier）的SIMTwarp的分支发散处理机制
- ✓核心思想: 用 “warp收敛屏障” 替代传统栈结构来管理分支发散后的线程同步。

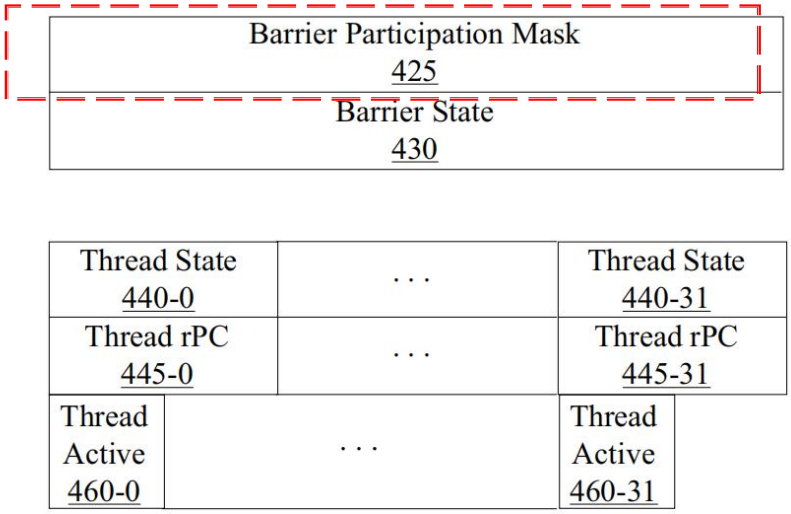


Figure 3.6

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓字段的存储与使用者
  - ✓这些字段（BPM、屏障状态、线程上下文信息、等）存储在寄存器中，由GPU硬件warp调度器直接使用。
- ✓BPM的作用
  - ✓每个屏障参与掩码（Barrier Participation Mask，BPM）的核心功能是：跟踪当前warp中，哪些线程需要参与某个特定的收敛屏障（即哪些线程因分支发散，需要在该屏障处同步重汇聚）。
- ✓warp的多掩码设计
  - ✓一个warp可能对应多个BPM。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓收敛屏障的常规工作场景：在常见情况下，被同一个屏障参与掩码（BPM）跟踪的线程，会在分支发散之后互相等待，直到所有线程都到达程序中的某个“公共点”（即重汇聚点），最终实现线程的重新同步执行（重汇聚）。

✓屏障状态字段的核心功能：为了支撑“线程互相等待、同步重汇聚”的过程，屏障状态（Barrier State）字段会用来跟踪哪些线程已经到达了当前的收敛屏障。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

```
1  // id = warp ID
2  // BB_A Basic Block "A"
3  if(id%2==0){
4      // BB_B
5  }else{
6      // BB_C
7      if(id==1){
8          // BB_D
9      }else{
10         // BB_E
11     }
12     // BB_F
13 }
14 // BB_G
```

Figure 3.7

✓Nested control flow example (based on Figure 6(a) from ElTantaway et al. [2014]).

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓基于收敛屏障（convergence barrier）的分支发散处理机制

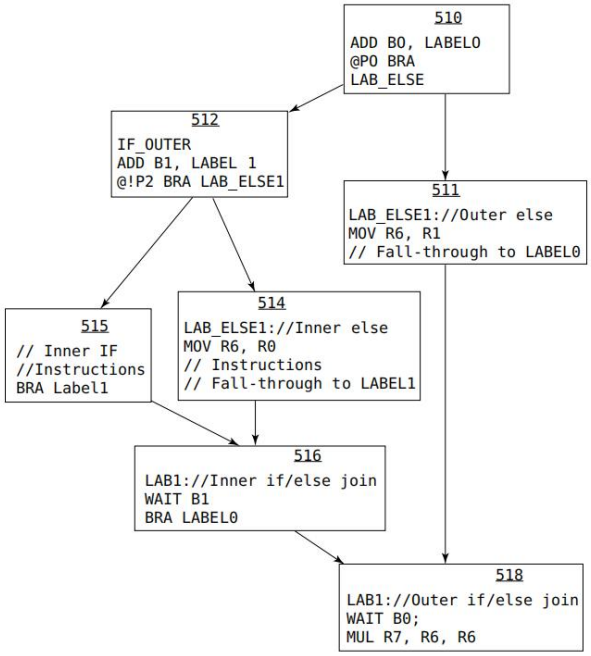


Figure 3.8

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓SIMT 收敛屏障中屏障参与掩码（BPM）的格式、含义

- ✓BPM的位数
  - ✓warp默认包含 32 个线程，因此BPM的宽度为32 位，与warp的线程数一一对应，保证每个线程都有专属的标识位。
- ✓BPM的位含义
  - ✓掩码的每一位对应warp中的一个线程：若某一位被置为 “1”，则表示该位对应的线程需要参与当前收敛屏障的同步重汇聚。

SIMT Deadlock and Stackless SIMT Architectures

- ✓线程发散的触发条件
  - ✓warp内的线程会在执行分支指令时发生发散，需要收敛屏障进行重汇聚管理。

SIMT Deadlock and Stackless SIMT Architectures

- ✓屏障参与掩码（BPM）的作用场景，以及嵌套控制流对多掩码的需求
  - ✓BPM的核心作用：BPM由warp调度器使用，用于让线程在特定的收敛屏障位置暂停，这个位置通常是分支的“直接后支配点”（即之前提到的重汇聚点），也可以是其他需要同步的位置，以此实现线程的重汇聚。
  - ✓多掩码适配嵌套控制流：同一warp在同一时间可能需要多个BPM，核心是为了支持嵌套控制流结构（例如，嵌套if语句），嵌套分支会产生多层发散，每层发散都需要独立的BPM来跟踪对应线程，因此单个warp需要同时维护多个掩码，实现分层的同步与重汇聚。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓SIMT无栈架构中屏障相关寄存器的实现方式
  - ✓屏障参与掩码（BPM）的存储冗余问题
    - ✓由于BPM仅32位宽（对应 32 线程的warp），若采用 “每个线程都存储一份掩码” 的方式（比如直接用通用寄存器文件存储），会造成存储冗余（因为掩码是warp级的全局信息，无需每个线程单独保存），因此实际不会采用这种朴素的存储方式。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓SIMT 架构中控制流嵌套对BPM管理方式的影响
  - ✓控制流的嵌套特性：SIMT 程序的控制流（如嵌套的条件分支、循环等）可以达到任意深度，不存在层数限制。
  - ✓对BPM的需求：由于控制流嵌套深度无上限，单个warp需要的BPM数量也会是任意的；每一层嵌套的分支都会对应一个独立的掩码，以实现分层重汇聚。
  - ✓掩码的管理方式：硬件难以预分配 “无限数量” 的存储资源来承载这些掩码，因此通过软件管理BPM成为更合理的选择，能适配任意嵌套深度的控制流需求。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓SIMT收敛屏障参与掩码（The barrier participation mask，BPM）的初始化，分支后线程发散的表现
  - ✓BPM的初始化指令：BPM的初始化，需通过一条特殊的 “ADD” 指令来完成。
  - ✓ADD 指令的作用逻辑：当warp执行该 ADD 指令时，当前处于活跃状态的所有线程，其对应的位会被置位到该 ADD 指令所指定的BPM中，这意味着这些活跃线程需要参与该收敛屏障的同步重汇聚。
  - ✓分支后的线程发散表现：执行分支指令后，部分线程会进入发散状态；具体表现为不同线程下一条要执行的指令地址（即程序计数器PC）出现差异，warp的执行路径从此分流。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓SIMT warp发散后的调度处理逻辑
  - ✓线程发散后的调度操作
    - ✓当warp内线程因分支执行出现PC（程序计数器）不一致（发散）时，调度器会：
      - ✓筛选出PC相同的线程子集；
      - ✓更新该子集线程对应的 “Thread Active” 字段，激活这些线程，使硬件能以锁步方式执行这部分PC一致的线程。
  - ✓学术界将这种 “因分支发散而拆分出的、PC相同的线程子集” 称为：warp拆分（warp split）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓基于收敛屏障的SIMT架构与传统栈式SIMT架构的调度差异
  - ✓调度灵活性的对比：与传统基于栈的SIMT实现不同，在基于收敛屏障的实现中，调度器可以自由切换不同的发散线程组（即之前提到的“warp split”），而栈式实现通常受栈的顺序限制，调度灵活性较低。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓基于收敛屏障的SIMT架构与传统栈式SIMT架构的调度差异
  - ✓基于收敛屏障的SIMT架构的核心优势：这种调度灵活性，能让warp在“部分线程已获取锁、部分未获取锁”的场景下，依然实现线程的向前推进（forward progress），调度器可切换处理已获取锁的线程组，避免因部分线程等待锁而导致整个warp停滞，从而缓解 SIMT 架构中常见的锁竞争导致的执行阻塞问题。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓SIMT收敛屏障机制中 “WAIT 指令” 的功能、设计与执行效果

- ✓WAIT 指令的核心用途：当 warp拆分（warp split），即分支发散后形成的 PC 一致线程子集，到达收敛屏障时，通过 “WAIT” 指令让该warp拆分暂停执行，进入同步等待状态。
- ✓WAIT 指令的操作数设计：根据NVIDIA的专利申请，WAIT 指令包含一个操作数，其作用是指定当前需要同步的 “收敛屏障标识”：以此明确该warp拆分对应的是哪一个收敛屏障，避免与其他屏障混淆。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓SIMT 收敛屏障机制中 “WAIT 指令” 的功能、设计与执行效果

- ✓WAIT 指令的执行效果：执行WAIT指令后，会完成两个关键操作
  - ✓将当前warp split中的线程，添加到对应收敛屏障的 Barrier State 寄存器中（用于跟踪已到达该屏障的线程）；
  - ✓将这些线程的状态修改为阻塞（blocked），直到该收敛屏障的所有参与线程都到达后，才会解除阻塞并恢复执行。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

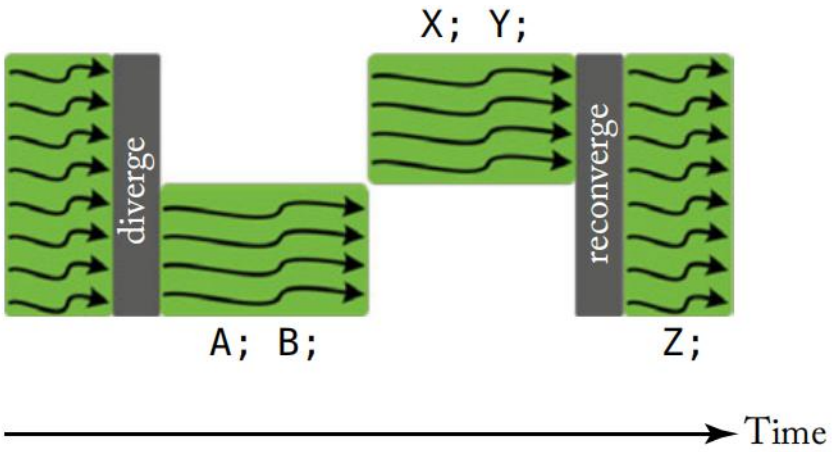


Figure 3.9

✓Example showing behavior of stack-based reconvergence (based on Figure 20 from Nvidia [2017]).

SIMT Deadlock and Stackless SIMT Architectures

✓SIMT 栈的作用

- ✓SIMT 栈用于记录分支的重汇聚点、线程活跃掩码，实现“先执行一个分支，再切换到另一个分支”的串行化执行，并保证所有线程最终在重汇聚点同步。
- ✓这个机制是早期GPU（如NVIDIA Kepler/Pascal）处理分支分歧的经典方式，但存在栈开销、嵌套分支深度限制等问题（后续架构如Volta引入了独立线程调度来优化）。

SIMT Deadlock and Stackless SIMT Architectures

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```

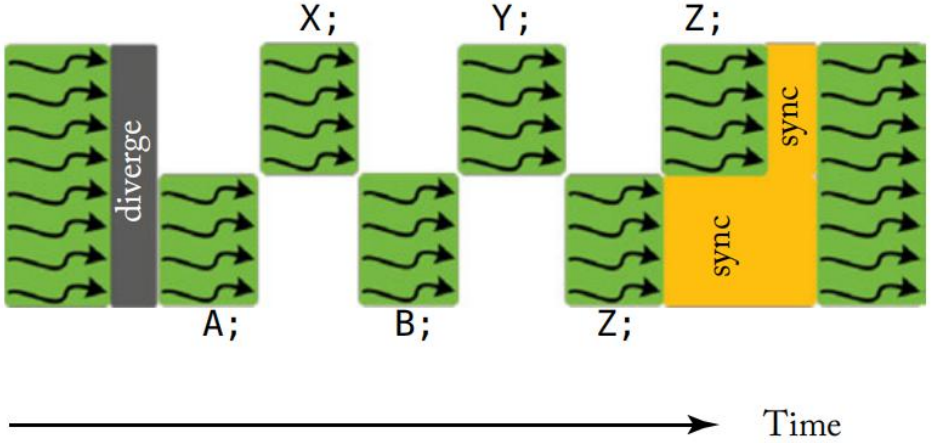


Figure 3.10

✓Example showing behavior of Volta reconvergence (based on Figure 23 from Nvidia [2017]).

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓NVIDIA Volta 架构下的SIMT分支重汇聚机制（对比之前的栈式重汇聚）

- ✓核心：基于独立线程调度（Independent Thread Scheduling, ITS）的分支处理方式：
  - ✓在原if-else后增加了__syncwarp()：这是 Volta 引入的warp显式同步指令，用于强制warp内的线程同步。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓NVIDIA Volta 架构下的 SIMT 分支重汇聚机制（对比之前的栈式重汇聚）
  - ✓执行逻辑（Volta 重汇聚）
    - ✓分支分歧（diverge），遇到if-else分支时，Volta 的线程不再依赖 “栈” 来串行化分支，而是线程独立执行：
      - ✓走if分支（threadIdx.x < 4）的线程，独立执行A; B;；
      - ✓走else分支的线程，独立执行X; Y;；
    - ✓（对比之前的栈式：栈式是先全线程执行完一个分支，再切换到另一个分支；而 Volta 中不同分支的指令可以并行推进）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓NVIDIA Volta架构下的SIMT分支重汇聚机制（对比之前的栈式重汇聚）
  - ✓执行逻辑（Volta 重汇聚）
    - ✓同步与重汇聚：
      - ✓当线程执行到Z或__syncwarp()时，会触发显式同步（sync）：所有线程需等待分支内的指令执行完毕，在同步点（Z/__syncwarp()）重新对齐，之后再继续锁步执行后续逻辑。
    - ✓核心优势（Volta vs 栈式重汇聚）
      - ✓摆脱了 “栈记录重汇聚点” 的限制，分支可以并行执行，减少了分支串行化的性能开销；
      - ✓支持更灵活的同步（通过__syncwarp()显式控制），适配复杂分支、嵌套分支的场景。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

对比维度	栈式SIMT重汇聚（早期架构：Kepler/Pascal）	Volta重汇聚（独立线程调度）
调度核心	warp锁步执行 + SIMT栈串行化分支	独立线程调度（ITS）
分支执行模式	先全warp执行一个分支，再切换执行另一个分支（串行）	不同分支的线程独立并行推进（并行）
同步方式	隐式同步（分支结束后自动在重汇聚点对齐）	隐式同步 + 显式同步（__syncwarp()指令）
依赖组件	需SIMT栈记录重汇聚点、线程活跃掩码	无需SIMT栈，靠线程状态追踪
分支性能开销	分支串行化导致开销较高	分支并行执行，开销显著降低
嵌套分支支持	受SIMT栈深度限制，复杂嵌套易受限	无栈深度限制，支持灵活嵌套

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Warp Scheduling

✓Example showing behavior of an academic mechanism similar to convergence barrier on spin lock code from Figure 3.5 (based on Figure 6(a) from ElTantawy and Aamodt [2016]).

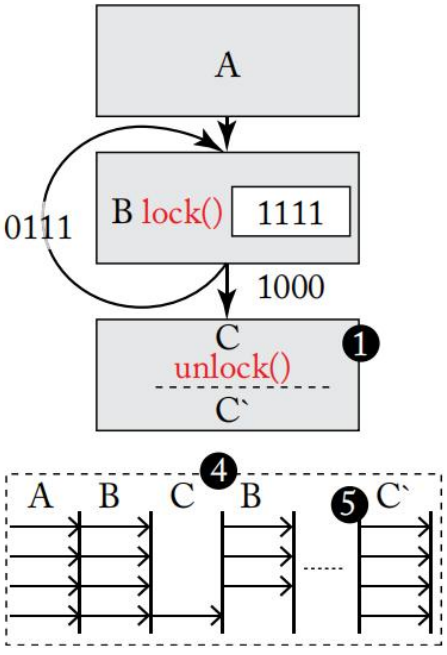


Figure 3.11

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Warp Scheduling

- ✓基于收敛屏障（convergence barrier）和自旋锁（spin lock） 的warp调度机制（ Figure 6(a) from ElTantawy and Aamodt [2016]）
  - ✓核心思想：warp被调度器多次调度（比如忙等时重试 lock 阶段），同时通过同步机制保证锁操作的正确性；

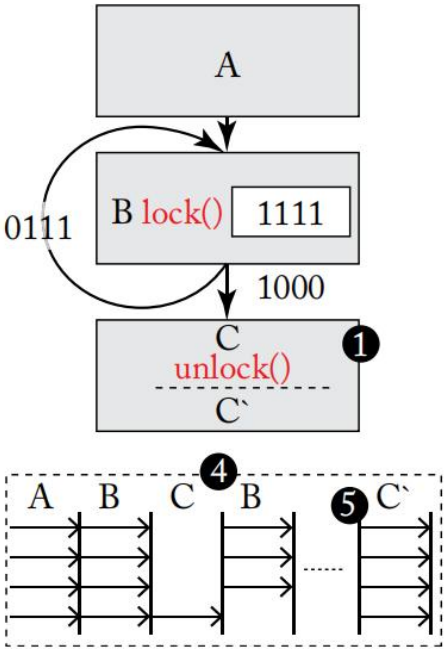


Figure 3.11

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Warp Scheduling

- ✓核心概念
  - ✓Warp
  - ✓自旋锁（spin lock）：一种“忙等”锁机制（线程循环检查锁是否可用，不挂起）。
  - ✓收敛屏障（convergence barrier）：让warp内的线程同步、收敛到同一执行路径的机制（避免线程分支导致的效率损失）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Warp Scheduling

✓流程图

- ✓A→B→C：warp的执行阶段
  - ✓B 阶段：执行lock()（加锁），此时锁的状态标记为1111（可理解为“全锁定”，对应warp内多个线程的锁状态）。
  - ✓C 阶段：执行unlock()（解锁），标记1代表解锁后的状态；同时存在循环箭头（0111/1000），表示自旋锁的“忙等”行为（若锁未获取成功，warp会重新调度到B阶段重试 lock）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Warp Scheduling

✓调度时序图

- ✓先调度包含「A→B→C」的warp（前几行）。
- ✓步骤 4：再次调度 B 阶段（对应自旋锁的“忙等重试”，重新尝试加锁）。
- ✓步骤 5：调度C'阶段（解锁后的后续执行阶段）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓基于栈的重收敛的时序（Volta架构之前的GPU调度方式）：warp内的线程分支后，会通过“栈”保存执行状态，等分支结束后统一“重收敛”（恢复warp的同步执行）。

✓NVIDIA Volta架构的“独立线程调度”（来自 Volta 白皮书）：Volta让warp内的每个线程可独立调度，不再严格绑定成束的同步收敛。

✓两种调度方式的核心行为差异：Volta 的调度会让“语句 A、B”与“语句 X、Y”交错执行；基于栈的重收敛中，这些语句是按warp的同步逻辑顺序执行的 。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓无栈架构的作用

- ✓无栈 SIMT 架构的执行逻辑：通过特定调度方式，执行自旋锁代码，以此避免 SIMT 死锁。
- ✓无栈架构不需要依赖“栈”保存warp的执行状态，因此在处理自旋锁这类“忙等”代码时，能更灵活地调度warp，规避线程同步导致的死锁风险。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓经过SMIT调度机制的解释，各位同学理解了：基于warp同步和SIMT栈的机制、Volta架构的独立线程调度（ITS）机制
  - ✓请问：Volta架构的独立线程调度（ITS）机制会不会造成传统的SIMT Deadlock问题呢？

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓答案：Volta架构的独立线程调度（ITS）机制不会造成传统的SIMT Deadlock问题，原因是：
  - ✓每个线程拥有独立的PC、寄存器和执行状态，可独立调度（不再严格绑定warp的同步）；
  - ✓用“汇合屏障（convergence barriers）”替代了SIMT栈，允许线程在分支后灵活切换执行路径，打破了之前导致死锁的调度约束。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

讲授内容:The SIMT Core

- One-Loop Approximation
  - Introduction
  - SIMT Stack: SIMT Execution Mask
  - SIMT Deadlock and Stackless SIMT Architectures
  - Warp Scheduling
- Two-Loop Approximation
  - Scoreboard
- Three-Loop Approximation
  - Operand Collector
  - Instruction Replay: Handling Structural Hazards

SIMT Deadlock and Stackless SIMT Architectures

✓流处理器上的warp：每个流处理器SM包含大量warp，多个warp会在SM中等待分配资源。

✓核心问题：warp scheduler如何为warp分配计算资源？

✓讨论假设：

✓每个warp被调度时，只发射一条指令；

✓这条指令执行完成前，该warp不能发射下一条指令。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓理想的内存系统下warp调度
  - ✓理想内存系统与延迟隐藏：假设内存系统是“理想的”（即内存请求的延迟固定、可预测），理论上可以通过细粒度多线程来设计GPU核心
    - ✓让核心支持足够多的warp，当某个warp等待内存请求完成时，立即调度另一个warp执行；
    - ✓以此“隐藏”内存延迟（把等待时间用在其他warp的计算上）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓理想的内存系统下warp调度
  - ✓轮询调度的硬件优势：在这种理想场景下，用 轮询（round robin）顺序调度warp
    - ✓保证目标吞吐量（因为warp切换足够频繁，内存延迟被有效隐藏）；
    - ✓简化调度逻辑（轮询是简单的循环调度方式），从而减少芯片的硬件面积（不需要复杂的调度决策模块）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓轮询（round robin）warp调度
  - ✓轮询调度的定义：轮询调度会给warp设定固定的顺序（比如按线程ID递增排序），调度器会严格按照这个预设顺序来选择下一个要执行的warp。
  - ✓比如把warp按ID从1到*N*排好队，调度器会循环依次选 1→2→…→*N*→1→2… 的顺序分配资源。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓轮询（round robin）warp调度
  - ✓轮询调度的核心特性
  - ✓关键特点：能让每个已发射的指令都获得大致均等的时间来完成执行。不会出现某个warp的指令被长期搁置的情况，时间资源在各warp间的分配相对公平。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓GPU核心中warp数量与吞吐量的关联
  - ✓执行单元持续忙碌的条件：
    - ✓若“核心内的warp数量 × 每个warp的发射时间”超过了内存延迟，核心的执行单元就会始终保持忙碌。
      - ✓原理：内存延迟是warp等待内存请求的时间；当warp数量足够多，在一个warp等待内存的时段内，调度器能依次发射足够多的其他warp，让执行单元不会出现空闲。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓GPU核心中warp数量与吞吐量的关联
  - ✓执行单元持续忙碌的条件：
    - ✓warp数量对吞吐量的提升作用：将warp数量增加到满足上述条件的临界点，理论上可以提升每个核心的吞吐量；
      - ✓原因：执行单元被持续利用、无空闲时间，单位时间内处理的任务 / 指令会更多。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓“增加GPU核心内warp数量” 的权衡代价
  - ✓warp高效切换的前提：独立寄存器
    - ✓要实现每个周期都让不同warp发射指令，必须让每个线程拥有独立的寄存器；
    - ✓这样warp切换时，不用在寄存器和内存之间拷贝 / 恢复状态，能直接切换执行，保证调度效率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

- ✓“增加GPU核心内warp数量” 的权衡代价
  - ✓warp数量增加的面积代价
    - ✓增加每个核心的warp数量，会让芯片面积中寄存器文件（存储寄存器的硬件模块）的占比上升，相对地，执行单元（负责计算的硬件）的面积占比会减少。
    - ✓原因：每个线程都需要独立寄存器，warp越多→线程总数越多→所需寄存器总量越大→寄存器文件的硬件面积就得扩大。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

SIMT Deadlock and Stackless SIMT Architectures

✓“增加GPU核心内warp数量” 的权衡代价

✓warp数量与核心总数的冲突

✓若芯片总面积固定，每个核心的warp数量增加（导致单个核心面积变大），那么芯片上能容纳的核心总数会减少。

✓总结：增加warp数量虽能提升单个核心的吞吐量，但会带来 寄存器面积占比升高、核心总数减少的代价，这是GPU硬件设计中需要平衡的关键取舍。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

讲授内容:The SIMT Core

- ➢One-Loop Approximation
  - ➢Introduction
  - ➢SIMT Stack: SIMT Execution Mask
  - ➢SIMT Deadlock and Stackless SIMT Architectures
  - ➢Warp Scheduling
- ➢Two-Loop Approximation
  - ➢Scoreboard
- ➢Three-Loop Approximation
  - ➢Operand Collector
  - ➢Instruction Replay: Handling Structural Hazards

Two-Loop Approximation: Motivations

- ✓双循环近似（Two-Loop Approximation）
  - ✓核心需求：减少warp数量的同时隐藏长延迟
    - ✓为了降低每个核心需要支持的warp数量（避免之前提到的“warp过多导致寄存器面积占比过高”的问题），理想的方式是：让同一个warp的后续指令，能在其早期指令还未完成时就发射执行（即指令级并行）。
    - ✓不用依赖大量warp切换来填充延迟，少量warp也能隐藏长执行延迟。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

- ✓双循环近似（Two-Loop Approximation）
  - ✓单循环微架构的局限：因为单循环微架构的调度逻辑只能获取 线程 ID和下一条要发射的指令地址，缺少跟踪同一个warp内未完成指令的能力，必须等一个warp的指令执行完成后，才能调度它的下一条指令（或切换到其他warp），所以“单循环微架构”做不到这种指令重叠执行；
  - ✓解决方案：突破单循环架构的调度限制，实现warp内的指令重叠执行，从而减少核心所需的warp数量，同时高效隐藏延迟。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

- ✓单循环微架构缺陷：缺乏指令依赖感知能力
  - ✓单循环微架构的调度逻辑，不仅只能获取线程ID和下一条指令地址（上一张提到的局限），单循环微架构无法判断“warp的下一条待发射指令”，是否依赖于该warp中还未执行完成的早期指令。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

- ✓单循环微架构缺陷：缺乏指令依赖感知能力
  - ✓因为调度器不知道指令间的依赖关系，为了避免执行错误（比如后续指令用到未完成指令的结果），调度器必须等早期指令执行完毕后，才能发射该warp的下一条指令
  - ✓彻底阻止了“同一个warp内指令重叠执行”的可能，只能依赖多warp切换来隐藏延迟（进而带来warp过多、寄存器面积占比过高的问题）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

✓单循环微架构缺陷：缺乏指令依赖感知能力

✓双循环近似的关键原因：双循环架构需要增加“指令依赖跟踪”的能力，从而安全的实现warp内的指令并行，减少核心所需的warp数量。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

✓ “双循环近似”实现“指令依赖感知”的具体技术方案

✓获取依赖信息的前提：先取指令

✓先从内存中取出指令：只有拿到指令本身，才能分析其操作数、资源需求，进而识别依赖关系。

✓识别指令间的数据冒险（数据依赖，比如后续指令用到前序指令的结果）或结构冒险（硬件资源冲突，比如多条指令争抢同一执行单元）

Two-Loop Approximation: Motivations

- ✓ “双循环近似” 实现 “指令依赖感知” 的具体技术方案
  - ✓GPU 的硬件支撑：指令缓冲区
    - ✓指令缓冲区（instruction buffer）：为了暂存已取出的指令（方便后续分析依赖），指令经过缓存访问后，会被暂存到这个缓冲区中，而非直接发射到执行流水线。
  - ✓调度逻辑的升级：独立调度器
    - ✓专门设计一个独立的调度器，负责分析指令缓冲区中各指令的依赖状态、资源可用性，最终决定 “哪条指令可以安全发射到后续流水线”（比如无依赖冲突、资源充足的指令）。

Two-Loop Approximation: Motivations

- ✓ “双循环近似” 实现 “指令依赖感知” 的具体技术方案
  - ✓双循环近似的核心目标：通过 “先取指令→缓冲缓存→独立调度器分析依赖” 的机制，让GPU能感知指令间的约束，从而实现warp内的指令并行。

Two-Loop Approximation: Motivations

- ✓“双循环近似（Two-Loop Approximation）”指令存储架构与指令缓冲区的延迟隐藏能力：
  - ✓指令缓存（instruction buffer）的架构设计：指令内存采用“多级缓存层级”实现
    - ✓最前端是一级指令缓存，直接为GPU核心提供指令；
    - ✓一级缓存的后端，由“一级或多级二级缓存”提供支撑，这类二级缓存通常是统一缓存（即同时存储指令与数据）
      - ✓该设计可提升缓存资源的复用率，适配GPU指令、数据访问频繁的场景。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

- ✓指令内存的实现方式
  - ✓指令缓冲区的额外作用：隐藏缓存缺失延迟；
  - ✓指令缓冲区不仅能暂存指令、辅助分析依赖，还能结合指令缺失状态保持寄存器（MSHR），MSHR是1981年Kroft提出的技术；

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

- ✓“双循环近似” 指令缓冲区的工作流程与组织形式
  - ✓指令缓冲区的组织形式
    - ✓指令缓冲区的设计没有固定模式，可采用多种组织方式；
    - ✓其中一种简单直接的方案是：为每个warp分配能存储 1 条或多条指令的空间。
      - ✓这种设计契合 GPU按warp调度的核心逻辑，方便管理单个warp的待执行指令，也便于后续调度器针对warp内的指令做依赖分析与并行发射。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

- ✓“双循环近似” 指令缓冲区的工作流程与组织形式
  - ✓指令进入缓冲区的时机
    - ✓当指令发生缓存命中（直接从指令缓存拿到指令），或缓存缺失后的填充完成（从后端缓存 / 内存取到指令）后，指令信息会被存入指令缓冲区：这是指令从 “存储层” 进入 “调度层” 的关键步骤，为后续的依赖分析、并行调度做准备。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

- ✓指令内存的实现方式
  - ✓MSHR隐藏指令缓存缺失的延迟：
    - ✓当指令缓存缺失（需要从更后端的缓存 / 内存取指令）时，MSHR会记录这个缺失请求的状态；
    - ✓此时指令缓冲区可以继续调度已缓存的其他指令，避免整个执行流水线因缓存缺失而停滞，从而“隐藏”缓存缺失的等待时间。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Motivations

- ✓CPU的两种依赖检测方法
  - ✓保留站（Reservation Station）
    - ✓作用：消除“名称依赖”（如寄存器重命名解决的写后写、读后写冲突）；
    - ✓缺点：需要复杂的关联逻辑，硬件面积与能耗成本较高。
  - ✓记分板（Scoreboard）
    - ✓特点：设计灵活性强，既支持按序执行，也可适配乱序执行，是更通用的依赖检测方案。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

✓单线程顺序CPU的记分板结构

✓记分板以“寄存器”为单位，为每个寄存器分配1 个标记位：当某条指令要写这个寄存器时，记分板会把该寄存器对应的位置为“已占用”状态（置位）。

✓记分板的调度规则（暂停机制）：若后续指令想要读 / 写这个寄存器，会先检查记分板中对应的位

✓如果位是“置位（已占用）”状态，这条指令会被暂停（stalled）；直到之前写该寄存器的指令执行完成，把这个位清除（置为空闲），后续指令才能继续执行。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

✓单线程顺序CPU的记分板结构

✓核心作用：防止数据冒险，避免两种经典的数据冒险（指令执行时因数据依赖导致的错误）

✓写后读（RAW）冒险：指令B要读寄存器X，但指令A还没写完X，此时记分板会让B暂停，等A写完（位清除）再执行；

✓写后写（WAW）冒险：指令B要写寄存器X，但指令A还没写完X，此时记分板会让B暂停，避免A、B的写操作顺序混乱。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

✓“双循环近似”记分板（Scoreboard）

✓防数据冒险的作用：当它和 “按序指令发射” 结合时，能避免 “写后读（WAR）冒险”（一种指令间的数据冲突）

✓ 前提是寄存器文件的读取也按顺序进行，这在按序CPU设计里是常规操作。

✓采用原因：这个设计是最简单的，因此占用的硬件面积和能耗最少，刚好匹配GPU需要大量并行单元、控制逻辑尽量简化的需求，所以GPU会用 “按序记分板”。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

✓按序记分板的两个核心问题

✓硬件开销大：

✓现代 GPU 的寄存器数量极多：每warp最多有128个寄存器，每个核心最多支持64个warp，因此单个核心的记分板需要占用128×64 = 8192位的硬件资源，对面积 / 存储的消耗较高。

✓依赖查询效率低：当指令遇到数据依赖时，必须反复去记分板中查询其操作数的状态，直到它所依赖的前序指令把结果写入寄存器文件，这个过程会产生重复查询的额外开销。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

- ✓按序记分板的优化设计（解决之前硬件开销大的问题）
  - ✓优化思路：不再采用“每个warp的每个寄存器都分配 1 位”的旧方式（旧方式硬件开销大），而是给每个warp只设置少量条目（近年研究显示约3-4个）。
  - ✓条目的作用：已经发射、但还没完成执行的指令，其要写入的寄存器的标识（ID），即跟踪这些“待写”的寄存器状态。
  - ✓通过只跟踪当前未完成指令的目标寄存器，大幅减少了记分板的硬件资源占用。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

- ✓常规按序记分板的访问时机
  - ✓在两个阶段被访问：
    - ✓“指令发射”阶段
    - ✓“指令写回”阶段
- ✓Coon 等人设计的记分板的访问时机
  - ✓目的：更高效的管理指令与寄存器的状态交互
  - ✓在这两个阶段：
    - ✓“指令被放入指令缓冲区”阶段
    - ✓“指令将结果写入寄存器文件”阶段。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

- ✓Coon 记分板的具体操作流程（指令入缓冲区阶段）
  - ✓记分板条目与寄存器的对比：
    - ✓当指令从指令缓存被取出、放入指令缓冲区时，会把 “该指令对应的warp” 的记分板条目，和这条指令的源寄存器、目的寄存器做匹配对比。
  - ✓生成短位向量：
    - ✓对比后会得到一个短位向量：向量里的每个位，对应这个warp记分板中的一个条目（因为每个warp的记分板只有3-4 个条目，所以位向量通常只有3或4位）。

Two-Loop Approximation: Scoreboard

- ✓短位向量的规则与后续处理
  - ✓位向量的置位规则：如果记分板中对应的条目，与当前指令的任意一个操作数（源 / 目的寄存器）匹配，那么位向量里对应的位就会被 “置位”（设为 1）。
  - ✓位向量的后续处理：这个记录了依赖关系的位向量，会和对应的指令一起，被复制到指令缓冲区中（跟着指令流转）。
  - ✓位向量的目的：让指令在后续阶段，能快速通过这个位向量判断自己与未完成指令的依赖状态，提升处理效率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

✓双循环近似（Two-Loop Approximation）”记分板（Scoreboard）机制的3个规则：

✓指令的调度条件：指令要被调度器处理，必须等其对应的 “标志位全部清零”

✓通过将标志位向量的每一位输入或非门（NOR gate）来判断是否全部清零；

✓没清零的指令不具备调度资格。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

✓双循环近似（Two-Loop Approximation）”记分板（Scoreboard）机制的3个规则：

✓依赖位的清除时机：指令缓冲中的依赖位（Dependency bits），会在指令将结果写入寄存器文件（register file）时被清除。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

✓双循环近似（Two-Loop Approximation）”记分板（Scoreboard）机制的3个规则：

✓资源耗尽的处理：若某个warp的所有条目都被用完，要么所有warp的取指操作暂停，要么该指令被丢弃、之后需要重新取指。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

✓双循环近似（Two-Loop Approximation）”记分板（Scoreboard）机制的3个规则：

✓GPU指令调度中记分板的工作逻辑：通过标志位控制指令调度、随结果写回清除依赖、资源不足时暂停取指或重取指令。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

- ✓双循环近似记分板（Two-Loop Approximation Scoreboard）中“指令写回寄存器文件时的操作规则”，核心逻辑是：
  - ✓当已完成执行的指令准备向寄存器文件写入结果时，会执行两个操作：
    - ✓清除该指令在记分板中被分配的对应条目；
    - ✓清除同一warp中、暂存于指令缓冲里的其他指令的对应的依赖位。
  - ✓对之前“依赖位在写回时清除”的细化：不仅要清理自身在记分板的资源，还要同步解除同warp内待调度指令的对应依赖，保证后续指令能正确判断调度资格。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Two-Loop Approximation: Scoreboard

- ✓双循环近似架构（Two-Loop Architecture）中“双循环”的具体职责
  - ✓第一个循环（First Loop）：负责取指
    - ✓选择“指令缓冲有空闲空间”的warp，读取该warp的程序计数器，再通过访问指令缓存，获取该warp的下一条指令。
  - ✓第二个循环（Second Loop）：负责指令调度
    - ✓从指令缓冲中，挑选“没有未解决依赖”的指令，将其发送到执行单元执行。
- ✓总得来说，双循环的分工是：第一个循环管“从指令缓存取指令到缓冲”；第二个循环管“把缓冲里无依赖的指令发去执行”，二者配合完成指令的流转。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

讲授内容:The SIMT Core

- One-Loop Approximation
  - Introduction
  - SIMT Stack: SIMT Execution Mask
  - SIMT Deadlock and Stackless SIMT Architectures
  - Warp Scheduling
- Two-Loop Approximation
  - Scoreboard
- Three-Loop Approximation
  - Operand Collector
  - Instruction Replay: Handling Structural Hazards

Three-Loop Approximation

✓三循环近似（Three-Loop Approximation）架构中“隐藏长内存延迟”的设计要求与硬件支撑

✓为了掩盖长内存延迟，需要满足两个运行特性：

✓每个核心支持大量warp（many warps per core）；

✓支持warp之间的逐周期切换（cycle by cycle switching between warps）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

- ✓三循环近似（Three-Loop Approximation）架构中“隐藏长内存延迟”的设计要求与硬件支撑
  - ✓要实现这两个特性，硬件上的必要条件是：配备一个大寄存器文件（large register file），且这个寄存器文件要为每个正在执行的warp，分配独立的物理寄存器。
  - ✓简单说，“多warp + 逐周期切换”是隐藏内存延迟的策略，而“大寄存器文件 + warp独立物理寄存器”是实现该策略的硬件基础。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

- ✓三循环近似架构中“寄存器文件的硬件实现成本逻辑”，核心包含两个关键关联：
  - ✓SRAM面积与端口的关系：SRAM存储器的芯片占用面积，和SRAM存储器的端口数量成正比（端口越多，SRAM 需要的面积就越大）。
  - ✓寄存器文件的朴素实现端口需求：如果用简单（naive）方案实现寄存器文件，需要的端口数是每个周期内、每条被发射的指令，其每个操作数都对应一个独立端口。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

✓三循环近似架构中“寄存器文件的硬件实现成本逻辑”，核心包含两个关键关联：

✓“大寄存器文件”的硬件代价：朴素实现的端口需求会让 SRAM 面积急剧增加，这也是三循环近似架构需要优化寄存器文件设计的原因之一。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

✓三循环近似架构中寄存器文件的面积优化方案：

✓寄存器文件的面积优化方法：为了缩小寄存器文件的面积，采用“多个单端口存储器 bank”来模拟多端口功能（替代面积过大的多端口 SRAM）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

- ✓三循环近似架构中寄存器文件的面积优化方案：
  - ✓操作数收集器的角色：虽然可以将这些bank暴露给指令集架构来实现效果，但部分 GPU 设计会用“操作数收集器（operand collector）”来更透明地完成该功能（无需指令集感知bank，对上层逻辑更友好）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

- ✓三循环近似架构中寄存器文件的面积优化方案：
  - ✓第三循环的来源：这个操作数收集器，实际上构成了三循环近似中的第三个调度循环。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

✓操作数收集器的核心作用：

- ✓支撑寄存器文件面积优化：配合 “多个单端口存储器 bank”，模拟多端口功能，从而减少寄存器文件的SRAM面积开销；
- ✓提升实现透明性：无需将存储器bank暴露给指令集架构，对上层逻辑（如指令调度）更友好，无需让指令感知底层存储结构；
- ✓构成第三调度循环：是三循环近似架构中“第三个调度循环”的载体，承担对应的调度职责。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Naive banked register file microarchitecture

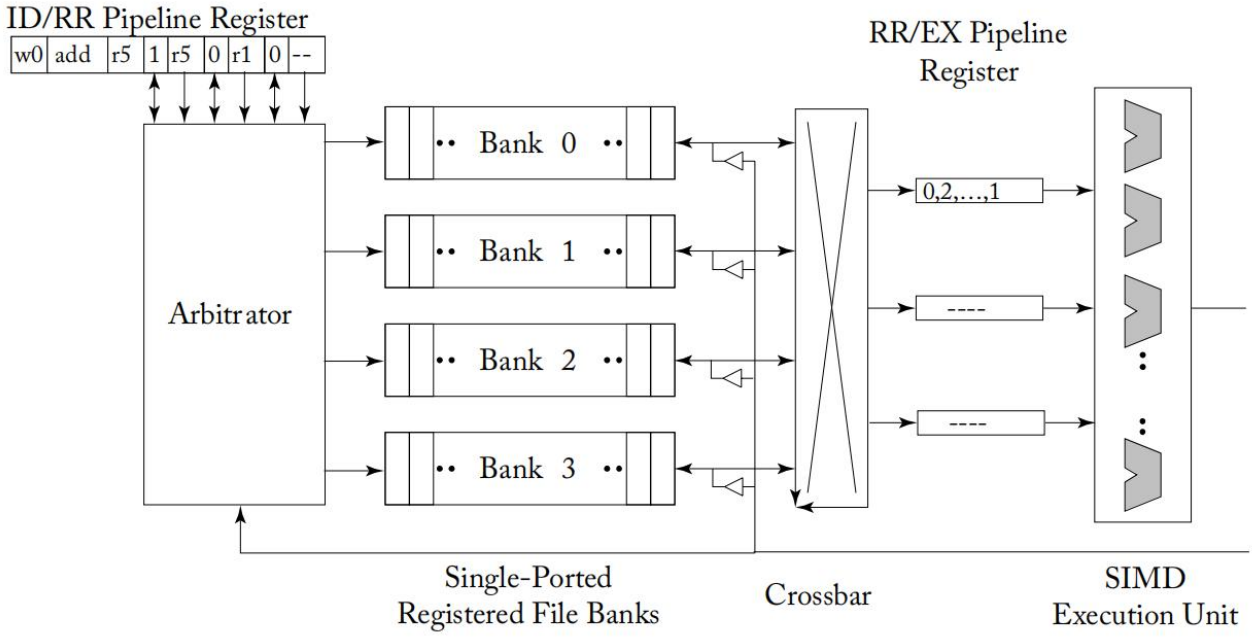


Figure 3.12

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

- ✓朴素分Bank寄存器文件（Naive Banked Register File，NBRF）微架构：
  - ✓输入：ID/RR流水线寄存器
    - ✓存储warp（如 w0）的指令信息（如 add 指令）及对应的寄存器操作数标识，是指令从“译码 / 读寄存器”阶段传递来的信息。
  - ✓仲裁器（Arbiter）
    - ✓接收 ID/RR 阶段的指令请求，将寄存器访问请求分配到对应的单端口寄存器bank（Bank 0~3）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

- ✓朴素分Bank寄存器文件（Naive Banked Register File，NBRF）微架构：
  - ✓单端口寄存器Bank（Single-Ported Registered File Banks）
    - ✓由多个单端口寄存器bank组成（图中是 Bank 0~3），替代面积较大的多端口SRAM，以此缩小寄存器文件的硬件面积。
  - ✓锁存 + 交叉开关（Crossbar）
    - ✓每个bank的输出会先锁存，再通过交叉开关完成数据路由：将不同bank的操作数，对应到指令所需的位置。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Three-Loop Approximation

✓NBRF微架构：

- ✓输出：RR/EX流水线寄存器+ SIMD执行单元
  - ✓交叉开关的输出存入“读寄存器→执行”阶段的流水线寄存器，最终将操作数送到SIMD执行单元完成指令执行。
- ✓NBRF微架构的核心逻辑是：用“仲裁器 + 多单端口 bank + 交叉开关”的组合，实现寄存器文件的多端口访问能力，同时控制硬件面积开销。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Naive banked register file microarchitecture

✓NBRF微架构

- ✓架构定位：NBRF微架构是为了提升寄存器文件的带宽，同时也是理解“操作数收集器解决了什么问题”的前置背景。
- ✓对应流水线阶段：NBRF微架构属于GPU指令流水线的“寄存器读阶段”，寄存器文件由 4 个单端口的逻辑 bank 组成。
- ✓实际扩展方式：由于寄存器文件的规模通常很大，实际中每个逻辑 bank 还会进一步分解为更多物理 bank。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Naive banked register file microarchitecture

- ✓ NBRF微架构的交叉开关与仲裁器的作用：
  - ✓逻辑bank与暂存寄存器的连接方式
    - ✓逻辑bank通过交叉开关，连接到“暂存寄存器（即流水线寄存器）”；
    - ✓这些暂存寄存器会先缓存源操作数，再将其传递给 SIMD 执行单元。
  - ✓仲裁器的核心职责
    - ✓仲裁器要完成两件事：
      - ✓一是控制对各个bank的访问权限，
      - ✓二是通过交叉开关将结果路由到对应的暂存寄存器。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Naive banked register layout

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Figure 3.13

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Naive banked register file microarchitecture

- ✓ NBRF微架构的布局：
  - ✓布局规则：
    - ✓每个warp（如 w0、w1）的寄存器，会按 寄存器编号的 “模 bank 数”分配到不同逻辑 bank：
      - ✓以 4 个 bank 为例：寄存器 r0→Bank0、r1→Bank1、r2→Bank2、r3→Bank3；r4（4 mod4=0）→Bank0、r5→Bank1，以此循环分配。
      - ✓图中 w0 的 r0/r4 在 Bank0，r1/r5 在 Bank1；w1 的 r0/r4 也遵循同样的分配规律。
    - ✓布局的作用：把不同寄存器分散到多个 bank，避免单个 bank 被集中访问导致的冲突，从而提升寄存器文件的带宽利用率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Naive banked register layout

- ✓NBRF布局
  - ✓NBRF布局定位：这是 “每个warp的寄存器分配到逻辑 bank” 的简单布局。
  - ✓分配示例：以warp w0为例，NBRF的r0寄存器存在Bank0的第一个位置，r1 寄存器存在Bank1的第一个位置，后续寄存器会按 “寄存器编号对应bank序号”的规律，依次分配到不同bank的对应位置。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Naive banked register layout

✓NBRF的环绕分配规则

- ✓当计算所需的寄存器数量超过逻辑 bank 的数量时，寄存器的分配会“环绕复用”之前的bank（不会新增 bank，而是重复使用已有的bank）。
- ✓示例验证：以4个逻辑bank为例：w0的r0-r3已分到Bank0-Bank3的第一个位置；r4（寄存器数≥bank 数）会“环绕”分配到Bank0的第二个位置。
- ✓价值：环绕分配规则让有限数量的bank，能支撑更多寄存器的存储，同时保持分bank的布局逻辑。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

```
i1:   mad   r2₂, r5₁, r4₀, r6₂
i2:   add   r5₁, r5₁, r1₁
```

Decode Cycle	Warp	Instruction
0	w3	i1: mad r2, r5, r4, r6
1	w0	i2: add r5, r5, r1
4	w1	i2: add r5, r5, r1

Cycle

Bank	1	2	3	4	5	6
0	w3:i1:r4					
1	w3:i1:r5	w0:i2:r1	w0:i2:r5	w1:i2:r1	w0:i2:r5	w1:i2:r5
2	w3:i1:r6		w3:i1:r2			
3						
EU		w3		w0		

Figure 3.14

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓NBRF的时序
  - ✓核心：呈现不同warp的指令，在分 bank 架构下对寄存器 bank 的访问节奏，以及执行单元的调度流程
    - ✓涉及的指令与warp
      - ✓指令：i1（mad操作，涉及寄存器r2/r5/r4/r6）、i2（add操作，涉及寄存器r5/r5/r1）
      - ✓warp：w3、w0、w1

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓NBRF的时序
  - ✓核心：呈现不同warp的指令，在分bank架构下对寄存器bank的访问节奏，以及执行单元的调度流程
    - ✓译码阶段的周期分配：不同warp的指令在不同周期完成译码
      - ✓Cycle 0：w3 的i1完成译码
      - ✓Cycle 1：w0 的i2完成译码
      - ✓Cycle 4：w1 的i2完成译码

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

✓NBRF的时序

- ✓核心：呈现不同warp的指令，在分 bank 架构下对寄存器bank的访问节奏，以及执行单元的调度流程
  - ✓各 Bank 的访问时序（按 Cycle）
    - ✓每个单端口 Bank 同一周期只能处理一个访问，时序：
      - ✓Cycle 1：Bank0（w3:i1→r4）、Bank1（w3:i1→r5）、Bank2（w3:i1→r6）被访问
      - ✓Cycle 2：Bank1（w0:i2→r1）被访问
      - ✓Cycle 3：Bank1（w0:i2→r5）、Bank2（w3:i1→r2，结果写回）被访问
      - ✓Cycle 4：Bank1（w1:i2→r1）被访问
      - ✓Cycle 5：Bank1（w0:i2→r5）被访问
      - ✓Cycle 6：Bank1（w1:i2→r5）被访问

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

✓NBRF的时序

- ✓核心：呈现不同warp的指令，在分bank架构下对寄存器bank的访问节奏，以及执行单元的调度流程
  - ✓执行单元（EU）的调度
    - ✓Cycle 2：EU执行w3的指令
    - ✓Cycle 4：EU执行w0的指令

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ NBRF的补充说明
  - ✓NBRF微架构可能会对性能造成负面影响。
  - ✓涉及的指令范围：案例包含图片顶部展示的两条指令。
  - ✓指令i1的寄存器分布：i1是乘加操作，其读取的寄存器 r5、r4、r6，分别被分配在Bank1、Bank0、Bank2

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ NBRF的补充说明
  - ✓指令 i2 的寄存器分布
    - ✓i2 是加法指令，其读取的两个寄存器（r5、r1）均被分配在Bank1中。
    - ✓由于该架构的 bank 是单端口设计，同一 bank 在同一周期仅能处理一次访问，因此i2的两个寄存器同属一个bank，会引发访问冲突，这也是该微架构性能受损的原因之一。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ NBRF的细节补充
  - ✓指令发射的周期与延迟
    - ✓cycle 0：warp3发射指令i1；
    - ✓cycle 1：warp0发射指令i2；
    - ✓cycle 4：warp1的i2发射出现延迟，延迟原因是后续会说明的bank冲突。
  - ✓NBRF架构下bank冲突会直接导致指令发射的滞后，是性能损耗的直接表现。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ NBRF的时序的细节补充
  - ✓cycle1中i1的无冲突访问
    - ✓cycle1时，warp3的i1能一次性读取所有3个源寄存器；因为这些寄存器对应不同的逻辑bank，单端口bank之间无访问冲突，因此可并行完成读取。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ NBRF中bank 冲突导致的指令访问延迟，核心时序与逻辑如下：
  - ✓cycle2的i2访问限制
    - ✓warp0 的i2指令有 2 个源寄存器，但两者均映射到Bank1（单端口设计），因此cycle2仅能读取其中 1 个寄存器；这是 bank 冲突直接限制了寄存器访问效率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ NBRF中bank 冲突导致的指令访问延迟，核心时序与逻辑如下：
  - ✓cycle3的延迟读取与并行操作
    - ✓该i2的第二个源寄存器延迟到cycle3才被读取，同时该周期还与warp3的i1指令的结果写回操作并行执行（体现 bank 在部分操作类型下的并行能力，但未解决冲突导致的延迟）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ NBRF中bank 冲突导致的指令访问延迟，核心时序与逻辑如下：
  - ✓cycle4 的重复冲突
    - ✓warp1 的 i2 指令再次出现相同问题：两个源寄存器同属 Bank1，因此 cycle4 仅能读取第一个源操作数，第二个操作数的读取被 bank 冲突阻塞。
  - ✓“单端口分bank架构” 的核心缺陷：寄存器映射到同一bank会引发访问冲突，直接导致指令执行延迟、性能下降。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓cycle5 时NBRF架构下的bank冲突升级场景，核心逻辑如下：
  - ✓在 cycle5，warp1的i2指令的第二个源操作数无法从寄存器文件读取
    - ✓原因：对应的Bank1已被 “warp0 更早发射的 i2 指令的写回操作” 占用，且该写回操作具有更高优先级，因此warp1的读取请求被阻塞。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓cycle6的最终读取：
  - ✓warp1的i2指令的第二个源操作数，经过多周期的bank冲突阻塞后，终于在cycle6完成了寄存器文件的读取。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓不仅“同指令的多个寄存器映射到同一bank”会引发冲突，不同warp的不同操作（读操作与写回操作）也会因共享同一bank产生冲突；
- ✓同时操作的优先级机制会加剧读取延迟，导致指令执行的性能损耗进一步扩大。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ NBRF架构性能缺陷总结：
  - ✓仅3条指令完成源寄存器读取，就消耗了6个周期；同时这段时间内多数bank处于未被访问的空闲状态。
  - ✓ NBRF架构的双重低效：
    - ✓延迟高：少量指令的寄存器访问需要多周期完成；
    - ✓资源利用率低：bank资源未被充分利用，却因冲突导致指令阻塞。
- ✓结论：引入高性能的寄存器文件方案“操作数收集器”

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

讲授内容:The SIMT Core

- ➢One-Loop Approximation
  - ➢Introduction
  - ➢SIMT Stack: SIMT Execution Mask
  - ➢SIMT Deadlock and Stackless SIMT Architectures
  - ➢Warp Scheduling
- ➢Two-Loop Approximation
  - ➢Scoreboard
- ➢Three-Loop Approximation
  - ➢Operand Collector
  - ➢Instruction Replay: Handling Structural Hazards

Operand collector microarchitecture (based on Figure 6 from Tor M. Aamodt *et al.*)

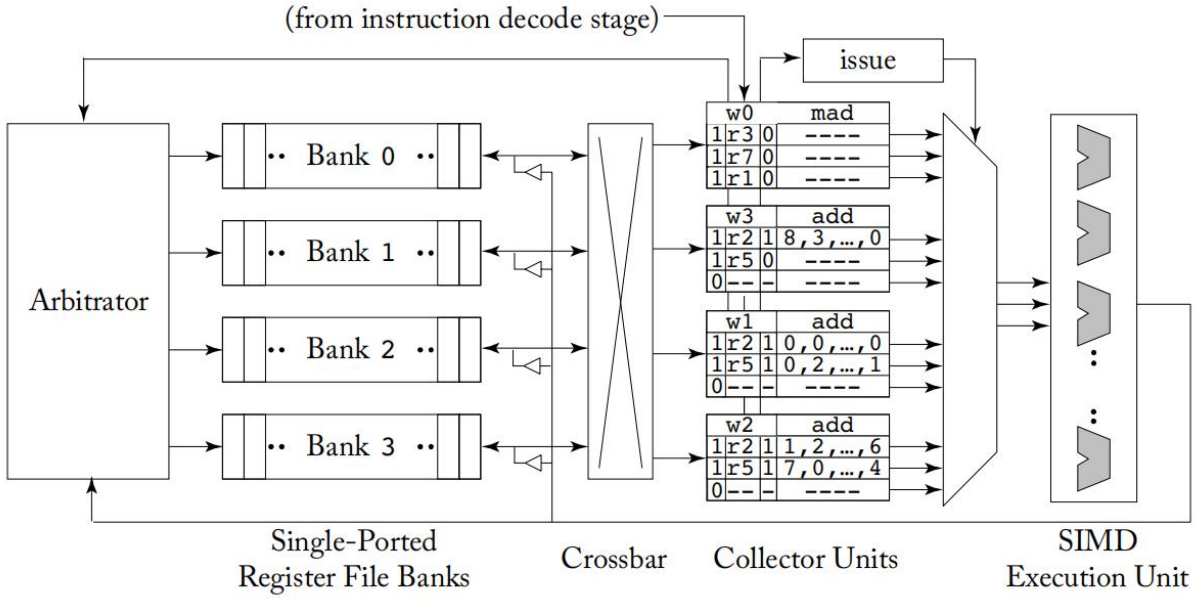


Figure 3.15

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Operand collector microarchitecture

- ✓针对NBRF缺陷，操作数收集器：
  - ✓核心组件
    - ✓仲裁器（Arbitrator）：控制对4个单端口寄存器Bank（Bank0~3）的访问请求分配；
    - ✓单端口寄存器 Bank：存储寄存器数据，通过单端口提供访问（维持面积优势）；
    - ✓交叉开关（Crossbar）：将不同Bank的输出路由到对应的收集器单元；

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Operand collector microarchitecture

✓针对NBRF缺陷，操作数收集器：

- ✓核心组件
  - ✓收集器单元（Collector Units）：按warp（如 w0、w3、w1、w2）分类，汇聚对应warp指令所需的所有源操作数；
  - ✓SIMD 执行单元：接收收集器单元输出的完整操作数集，执行指令。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Operand collector microarchitecture

✓针对NBRF缺陷，操作数收集器：

- ✓数据流
  - ✓仲裁器将寄存器访问请求分配到对应 Bank；
  - ✓Bank的输出通过交叉开关，被路由到对应warp的收集器单元；
  - ✓收集器单元持续汇聚该warp指令所需的所有操作数，待操作数齐全后，由issue信号触发；
  - ✓完整的操作数集被发送到SIMD执行单元执行指令。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Operand collector microarchitecture

- ✓针对NBRF缺陷，操作数收集器：
  - ✓核心优化作用：解决了NBRF架构的冲突与延迟问题。
    - ✓通过收集器单元汇聚操作数，让指令无需因 bank 冲突分周期读取操作数，而是等所有操作数从不同Bank读取并汇聚后再执行，既保留了单端口Bank的面积优势，提升了指令执行的效率与资源利用率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ 操作数收集器微架构 的核心说明
  - ✓核心架构变化
    - ✓NBRF架构相比，关键修改是将 “暂存寄存器” 替换为 “收集器单元”
    - ✓解决此前bank冲突、操作数读取延迟问题的核心设计调整。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓ “操作数收集器微架构” 的核心说明
  - ✓收集器单元的分配逻辑
    - ✓每个指令在进入 “寄存器读阶段” 时，会被分配一个独立的收集器单元，确保该指令的所有源操作数能被单独汇聚，避免不同指令的操作数访问相互干扰。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓操作数收集器微架构中 “多收集器单元” 的设计价值
  - ✓多收集器单元的设计
    - ✓架构中配备了多个收集器单元，支持多个指令的源操作数读取过程重叠执行。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓操作数收集器微架构中 “多收集器单元” 的设计价值
  - ✓对吞吐量的提升作用
    - ✓即使单个指令的源操作数之间存在 bank 冲突，多收集器单元也能让不同指令的操作数读取并行推进，从而在冲突场景下仍能提升整体的处理吞吐量。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓操作数收集器微架构中 “多收集器单元” 的设计价值
  - ✓“操作数收集器”弥补了NBRF架构的低效性：通过 “并行重叠执行” 的方式，在保留单端口 bank 面积优势的同时，缓解了 bank 冲突对整体性能的拖累。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓操作数收集器的优点：
  - ✓收集器单元的缓存能力，优化了指令执行延迟
    - ✓每个收集器单元内置缓存空间，可容纳一条指令执行所需的全部源操作数
    - ✓这让指令无需分周期读取操作数，而是等所有操作数从对应 bank 读取并缓存后，再传递给执行单元，避免了NBRF架构中的分周期延迟。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of naive banked register file

- ✓操作数收集器的优点：
  - ✓bank 级并行性的提升，提高硬件资源利用率
    - ✓由于多个指令的源操作数总量较多，仲裁器更易调度这些操作数对应的不同 bank，从而实现更高的“bank级并行性”，支持同时访问多个寄存器 bank
    - ✓解决了NBRF中bank空闲率高的问题，大幅提升了 bank 资源的利用率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Figure 3.16

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR（Swizzled banked register）的核心内容：
  - ✓布局规则
    - ✓与NBRF的 “所有warp共用同一寄存器”不同，bank 映射规则” 不同，SBR对不同warp采用差异化的bank 映射规则：
      - ✓warp w0：寄存器按 “r编号 mod 4” 分配 bank（如 r0→Bank0、r1→Bank1、r4→Bank0）；
      - ✓warp w1：寄存器按 “(r编号 + 1) mod 4” 分配 bank（如 r0→Bank1、r1→Bank2、r3→Bank0）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR（Swizzled banked register）的核心内容：
  - ✓设计目的
    - ✓通过差异化的映射规则，让不同warp的同编号寄存器（如w0:r0与w1:r0）分散到不同bank，避免NBRF布局中“多warp同编号寄存器集中在同一bank”导致的冲突，从而降低跨warp的bank冲突概率，提升寄存器访问的并行性与资源利用率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR的研究背景:
  - ✓操作数收集器的局限
    - ✓操作数收集器仅通过调度机制“容忍”已发生的bank冲突，而非从根源上减少冲突的发生。
  - ✓待解决的核心问题
    - ✓这一局限引出了新的需求：如何从设计层面减少bank 冲突的数量。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

✓SBR设计核心思路

- ✓ 将不同warp的等价寄存器（即不同 warp 中编号相同的寄存器）分配到不同的bank，从寄存器映射层面分散访问压力。

✓SBR示例

- ✓warp0的r0被分配到Bank0；
- ✓warp1的r0被分配到Bank1。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

✓SBR的作用

- ✓通过让不同warp的同编号寄存器分散在不同bank，避免了多warp同时访问同一bank的同编号寄存器的场景，从而降低跨warp的bank冲突概率，提升寄存器文件的访问并行性与资源利用率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR适用范围：
  - ✓存在的局限性：该布局无法解决单条指令内部寄存器操作数之间的 bank 冲突（即同一条指令的多个源寄存器仍可能映射到同一 bank）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR适用范围：
  - ✓核心优化作用
    - ✓有效减少不同warp的指令之间的 bank 冲突
    - ✓通过将不同warp的等价寄存器分配到不同bank，避免了多warp指令同时访问同一bank的场景，提升了跨warp指令的寄存器访问并行性。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR优势场景：所有warp进度相对均匀；
- ✓SBR优势场景下的调度策略
  - ✓轮询调度（round robin scheduling）
  - ✓两级调度（two-level scheduling，参考Narasiman et al. 2011 的研究）：取指组内的单个warp会按轮询顺序被调度。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR的收益：让不同warp的指令更大概率同时发起寄存器访问，此时SBR “将不同 warp 的等价寄存器分配到不同 bank” 的设计，能有效减少跨warp指令之间的bank冲突，最大化寄存器访问的并行性。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Timing of operand collector

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Cycle →

Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

Figure 3.17

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓操作数收集器的时序图：指令、寄存器 bank 访问与执行单元调度的节奏
  - ✓涉及的指令与译码周期分配
    - ✓指令：i1（add操作，r1=r2+r5）、i2（mad操作，r4=r3*r7+r1）
    - ✓译码周期：不同 warp 的指令在不同周期完成译码
      - ✓Cycle 0：w1的i1
      - ✓Cycle 1：w2的i1
      - ✓Cycle 2：w3的i1
      - ✓Cycle 3：w0的i2

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓操作数收集器时序图：指令、寄存器 bank 访问与执行单元调度的节奏
  - ✓寄存器bank的访问时序（Cycle1~6）：各单端口 bank 在不同周期被不同warp的指令访问，实现了多warp操作数访问的重叠执行
    - ✓Cycle 1：Bank3（w1→r2）
    - ✓Cycle 2：Bank2（w1→r5）、Bank3（w2→r5）
    - ✓Cycle 3：Bank1（w3→r2）、Bank3（w0→r3）
    - ✓Cycle 4：Bank0（w3→r5）、Bank2（w1→r1）、Bank3（w2→r1）
    - ✓Cycle 5：Bank3（w0→r7）
    - ✓Cycle 6：Bank0（w3→r1）

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓操作数收集器时序图：指令、寄存器 bank 访问与执行单元调度的节奏
  - ✓执行单元（EU）的调度：执行单元按周期调度已完成操作数收集的指令，节奏紧凑
    - ✓Cycle 2：执行w1的i1
    - ✓Cycle 3：执行w2的i1
    - ✓Cycle 4：执行w3的i1

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓架构优化的体现：对比NBRF架构，该时序体现了操作数收集器的核心优势
  - ✓多warp的操作数访问可重叠执行，提升了bank资源利用率
  - ✓指令执行的周期更紧凑，避免了大量空闲等待，有效提升了整体吞吐量

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR指令与寄存器分配的补充说明
  - ✓指令发射信息：warp0的 i2 指令在cycle3完成发射。
  - ✓寄存器的bank分配特点：add 指令（i1）的目标寄存器 r1，与该指令的源寄存器 r5，对于任意一个warp而言，都会被分配到同一个 bank。
  - ✓SBR的局限性：仅解决不同warp之间的bank冲突，而单条指令内部的寄存器（如i1的r1与r5）仍可能处于同一bank，因此无法避免单指令内部的 bank 冲突。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓NBRF与SBR的区别
  - ✓核心设计差异：在SBR中，不同warp（warps）会访问不同的bank；
  - ✓冲突优化效果：这一设计能有效减少“某一warp的写回操作”与“其他warp的源操作数读取操作”之间的bank冲突，提升了跨warp操作的并行性。

Swizzled banked register layout

- ✓SBR+ 操作数收集器架构下bank访问时序的细节说明
  - ✓时序区域的功能：展示了操作数收集器作用下的bank级访问时序，体现该架构对寄存器访问节奏的优化。
  - ✓cycle1的具体访问：在cycle1，warp1的寄存器r2完成对Bank3的读取操作。

Swizzled banked register layout

- ✓SBR+ 操作数收集器架构下 bank 访问时序的细节说明
  - ✓cycle4的并行操作（核心优化体现）：cycle4时，warp1的寄存器r1写回操作，与warp3的r5读取操作、warp0的r3读取操作并行执行
  - ✓这是SBR（分散不同 warp 的寄存器到不同 bank）与操作数收集器（调度多操作重叠）结合的效果，既避免了bank冲突，又提升了bank资源的利用率。

Swizzled banked register layout

- ✓当前设计的操作数收集器没有对不同指令的“就绪发射顺序”做约束，这可能导致读后写（WAR）冒险（一种指令间的数据冲突）
- ✓操作数收集器的作用是为指令准备所需的操作数，但如果操作数收集器不控制指令发射的顺序，可能出现“后一条指令先写寄存器，而前一条指令后读同一寄存器”的情况（读后写WAR 冒险），破坏指令执行的正确性。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

✓SBR下读后写WAR 冒险具体触发场景：

- ✓当同一warp的两条指令同时存在于操作数收集器中时，如果第一条指令要读取某个寄存器，而第二条指令要写入同一个寄存器；由于操作数收集器不约束指令的发射顺序，第二条指令可能先完成写操作，导致第一条指令读到被提前修改的寄存器值，最终触发“读后写（WAR）冒险”。
- ✓简单说：同一warp的“读寄存器指令”和“写同个寄存器的指令”同时在操作数收集器里，顺序乱了就会出 WAR 问题（这是GPU架构中寄存器冲突的典型场景之一）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

✓SBR的Bank冲突加剧WAR冒险

- ✓寄存器是按bank存储的，若第一条指令的源操作数访问重复命中同一 bank（即出现 bank 冲突），会导致这条指令的读操作被延迟；
- ✓此时，第二条要写同一寄存器的指令，可能在第一条指令还没读到“原本该读的旧值”之前，就完成了写操作；
- ✓最终第一条指令读到的是被第二条指令提前修改的新值，触发“读后写（WAR）冒险”。

✓简单说：bank冲突让“读寄存器”的指令变慢，给“写同个寄存器”的指令留出了抢先执行的时间，最终导致数据读错。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

✓SBR下WAR 冒险的预防策略：

✓基础预防方法：要求同一warp的指令，从操作数收集器发送到执行单元时，必须遵循 “程序编写的原始顺序”；通过强制指令按逻辑顺序发射，避免 “后写指令抢先执行” 的情况，从根源减少WAR冒险。

✓相关研究：Mishkin 等人在 2016 年探索了三种硬件复杂度较低的解决方案，同时评估了这些方案对性能的影响（既控制硬件成本，又兼顾执行效率）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

✓SBR下WAR 冒险的预防策略：Mishkin 等人提出的第一个低硬件复杂度解决方案（release-on-commit warpboard）

✓方案规则：“提交后释放的warp板” 要求每个warp同一时间最多只能有一条指令在执行；必须等当前指令执行完成并提交后，才会释放下一条指令执行。

✓性能影响：这个方案虽然硬件复杂度低，但会严重限制指令并行度，在部分场景下性能几乎下降一半，负面影响很明显。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR下WAR冒险的预防策略：Mishkin 等人提出的第二个解决方案（release-on-read warpboard）
  - ✓方案规则：“读操作后释放的warp板” 要求每个warp同一时间，仅允许一条指令在操作数收集器中执行操作数收集；等这条指令完成读操作后，再释放下一条指令进入收集阶段。
  - ✓性能影响：相比第一个方案，这个策略的性能损失小得多，在研究的工作负载中，性能下降较多。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR下WAR冒险的预防策略：Mishkin 等人提出的第三个解决方案（bloomboard 机制）
  - ✓方案设计：为了在操作数收集器中保留指令级并行性（避免前两个方案限制并行的问题），这个机制使用bloom过滤器（bloom filter）来跟踪 “未完成的寄存器读操作”; 通过监测读操作状态，既防止WAR冒险，又不阻碍指令并行执行。
  - ✓性能影响：相比 “放任WAR冒险（错误执行）” 的情况，这个方案的性能损失很小，是三个方案中性能影响最小的。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

方案名称	核心规则	硬件复杂度	性能损失情况
release-on-commit warpboard	每个warp同一时间最多仅1条指令执行	低	部分场景性能几乎下降一半
release-on-read warpboard	每个warp同一时间仅1条指令在操作数收集器中收集操作数	低	最多仅10%的性能下降
bloomboard机制	用小型bloom过滤器（bloom filter）跟踪未完成的寄存器读操作，保留指令级并行	低	仅不到几个百分点的性能损失

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Swizzled banked register layout

- ✓SBR下WAR冒险的解决方法：Maxwell GPU “读依赖屏障”
  - ✓NVIDIA的Maxwell架构GPU引入了“read dependency barrier（读依赖屏障）”：由特殊的“控制指令”管理，作用是避免WAR冒险（Write-After-Read，写后读冒险）；WAR冒险这是指令执行乱序时可能出现的冲突（比如指令B要读寄存器X，但指令A还没写完X，就会导致错误）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

讲授内容:The SIMT Core

- One-Loop Approximation
  - Introduction
  - SIMT Stack: SIMT Execution Mask
  - SIMT Deadlock and Stackless SIMT Architectures
  - Warp Scheduling
- Two-Loop Approximation
  - Scoreboard
- Three-Loop Approximation
  - Operand Collector
  - Instruction Replay: Handling Structural Hazards

Instruction Replay: Handling Structural Hazards

✓“结构冒险”是处理器流水线的典型问题之一（因硬件资源不足导致指令执行冲突），而“指令重放”是 GPU 解决流水线冲突的一种策略；。

✓GPU 流水线存在多种结构冒险的潜在原因

✓结构冒险：硬件资源（如功能单元、寄存器相关组件等）不足，导致指令无法按流水线流程执行的冲突。

✓寄存器读取阶段可能会耗尽“操作数收集器单元”

✓操作数收集器是负责从寄存器中提取指令所需操作数的硬件模块，若该单元数量不足，后续指令就会因抢不到资源而阻塞，触发结构冒险。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Instruction Replay: Handling Structural Hazards

- ✓单条内存指令需拆分执行
  - ✓当一个warp执行单条内存指令时，这条指令通常需要被拆分成多个独立的子操作（原因：内存访问的粒度、硬件资源限制等；比如内存指令对应的地址可能分布在不同内存块，需分步骤完成访问）。
- ✓拆分后的子操作高效利用流水线
  - ✓每个拆分出的独立子操作，能在某个时钟周期内充分利用流水线的一部分资源（作用：通过拆分操作，避免硬件资源“不够用”的结构冒险，同时提升流水线资源的利用率）。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Instruction Replay: Handling Structural Hazards

- ✓核心问题：GPU流水线中，当指令遇到结构冒险时，会如何处理？
  - ✓单线程顺序CPU的解决方案：在单线程、按顺序执行的 CPU 流水线中，处理结构冒险的标准方式是：暂停后续（“更年轻的”）指令的执行，直到遇到冒险的指令能获取足够资源、继续推进。
  - ✓这种“暂停后续指令”的思路，对GPU（高线程吞吐量架构）来说“不太理想”：因为GPU的设计目标是同时处理大量线程，这种暂停会严重影响整体吞吐量，具体原因会后续说明。
  - ✓这里的“结构冒险”依旧是硬件资源不足导致的指令执行冲突；“高线程吞吐量架构”是指GPU通过同时调度大量线程来提升计算效率。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Instruction Replay: Handling Structural Hazards

✓CPU与GPU处理结构冒险的策略对比

对比维度	单线程顺序CPU架构	高线程吞吐量GPU架构
架构类型	单线程、按指令顺序执行的流水线	多线程并行、大规模吞吐的并行架构
核心目标	降低单线程的执行延迟	提升海量线程的整体计算吞吐量
结构冒险处理策略	暂停后续（“更年轻的”）指令，等待冲突指令获取资源后再推进	不采用“暂停后续指令”（该方案会严重降低吞吐量），后续采用“指令重放”等适配高线程的策略
策略适配性	适配单线程场景，保证单线程执行连贯性	“暂停”策略完全不适配：会阻塞大量线程，违背GPU的高吞吐设计目标

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Instruction Replay: Handling Structural Hazards

✓“指令重放”解决GPU的结构冒险时，会遇到“影响关键路径”“增加芯片面积”这两个实际问题。

- ✓“首先，由于寄存器堆（register file）规模较大，且完整图形流水线需要很多流水线阶段；此时传递“暂停信号（stall signal）”，可能会影响关键路径（处理器中延迟最长的电路路径，决定时钟周期快慢）。”
  - ✓简单说：图形流水线的寄存器堆大、阶段多，传递暂停信号的路径太长，会拖慢处理器的核心运行节奏。
- ✓对“暂停周期分配”做流水线化处理后，需要引入额外的缓冲（buffering），这会增加芯片的面积开销。
  - ✓简单说：为了协调流水线里的暂停周期，得加更多缓冲存储数据，代价是芯片要占用更大物理空间。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Instruction Replay: Handling Structural Hazards

- ✓用指令重放处理结构冒险时，会出现“线程束暂停的连锁反应”，引发不必要的执行停滞，进而降低GPU的运行效率。
  - ✓暂停某一个warp中的指令，可能会导致其他线程束的指令也跟着被暂停。
    - ✓当一个warp的指令因资源冲突（结构冒险）被暂停时，可能会连带其他warp的指令一起停滞（比如受流水线调度机制的影响）。
  - ✓如果这些被连带暂停的指令，并不需要“导致暂停的那条指令”所占用的资源，那么（处理器的）吞吐量就会受损。
    - ✓简单说：本来其他线程束的指令可以正常执行，但被无辜“连累”暂停了，相当于浪费了执行资源，最终导致处理器的整体处理效率（吞吐量）下降。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Instruction Replay: Handling Structural Hazards

- ✓指令重放是GPU解决结构冒险的方案，同时也是CPU应对“投机调度 + 可变延迟指令”的恢复机制，是跨处理器架构的一种技术。
  - ✓为了规避之前提到的那些问题，GPU采用了一种“指令重放”的方案。
    - ✓结构冒险导致的“暂停信号影响关键路径”“额外缓冲增加芯片面积”“线程束连锁暂停降低吞吐量”等问题；GPU用“指令重放”来解决这些问题。
  - ✓指令重放在部分CPU设计中也存在，它被用作一种‘恢复机制’：当CPU投机调度（提前安排）某个‘依赖指令’时，若这个指令所依赖的‘更早指令’具有可变延迟（执行耗时不确定），此时就用指令重放来处理潜在的执行问题。
    - ✓简单说：CPU提前调度指令时，若依赖的指令延迟不稳定，可能导致调度出错，这时候指令重放就是“兜底恢复”的手段。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Instruction Replay: Handling Structural Hazards

- ✓CPU的缓存访问为例，解释指令重放适用场景中 “可变延迟”
  - ✓加载指令（loads，从存储设备取数据到寄存器）执行时，数据可能‘命中一级缓存’（数据在缓存里，执行快），也可能‘未命中一级缓存’（数据不在缓存，得去更慢的存储层级取，执行慢）；而高时钟频率的CPU，会把‘一级缓存访问’的过程流水线化，拆分到多达 4 个时钟周期里完成。
  - ✓CPU 中 “加载指令” 的延迟是可变的（命中 / 未命中缓存的耗时差异大），再加上高频CPU把缓存访问拆成多流水线阶段（这里是 4 个周期），进一步让指令延迟变得不确定。这种 “可变延迟”，正是CPU需要用 “指令重放” 做恢复机制的场景之一。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;

Instruction Replay: Handling Structural Hazards

- ✓CPU 与 GPU 在 “投机调度” 和 “指令重放” 上的设计取舍
  - ✓关于 CPU 的操作
    - ✓有些 CPU 会 “投机唤醒” 依赖加载指令的后续指令；也就是提前调度这些依赖指令，目的是提升单线程性能（让单个线程的执行节奏更快）。
  - ✓关于 GPU 的选择
    - ✓相比之下，GPU会避免投机操作：因为投机对GPU（主打大规模多线程并行）来说，既会浪费能量，又会降低整体吞吐量（反而拖累并行效率）。
  - ✓GPU用指令重放的作用
    - ✓GPU转而使用指令重放，是为了避免流水线堵塞，同时规避 “暂停操作带来的电路面积 / 时序开销”（对应之前提到的暂停导致的芯片面积增加、关键路径延迟等问题）。
- ✓总得来说，CPU靠 “投机调度” 优化单线程速度，GPU则放弃投机、用 “指令重放” 来优化流水线和降低暂停代价；这是两者（单线程 vs 多线程并行）差异带来的设计选择。

Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Springer Cham, 2018;


THANKS