人工智能芯片与系统

Grading Policy

5% pop quiz+checking
35% labs(40% report + 60% check): lab1-3: 15%+10%+10%
60% final exam(1 A4 memo)
5%-10% bonus

1.Introduction

3 success factors for machine learning:

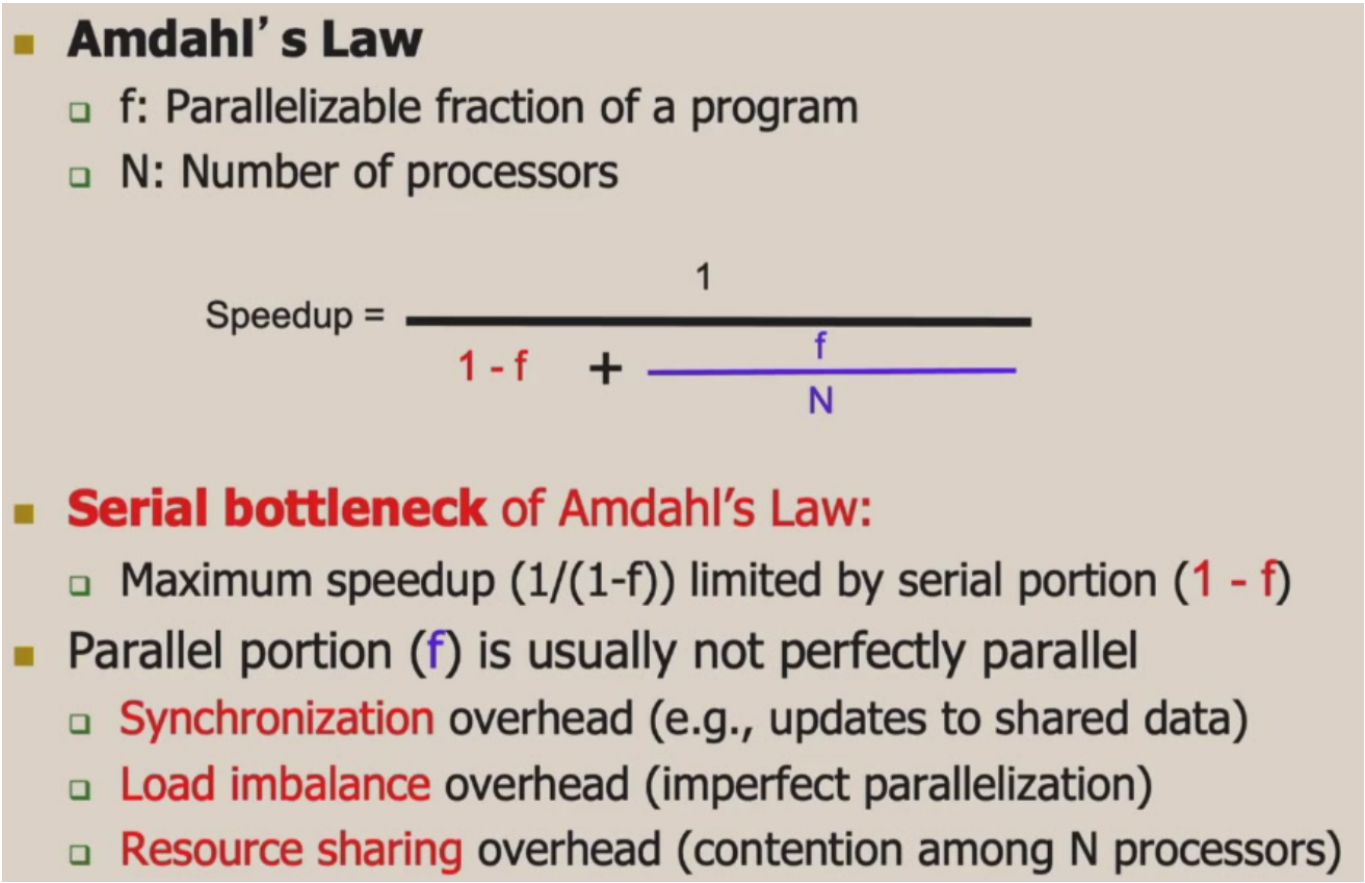
- big data
- algorithm
- computer power

1.Amdahl's Law

定律本身

加速上限

无法完美并行的三个原因



2.Roofline Model

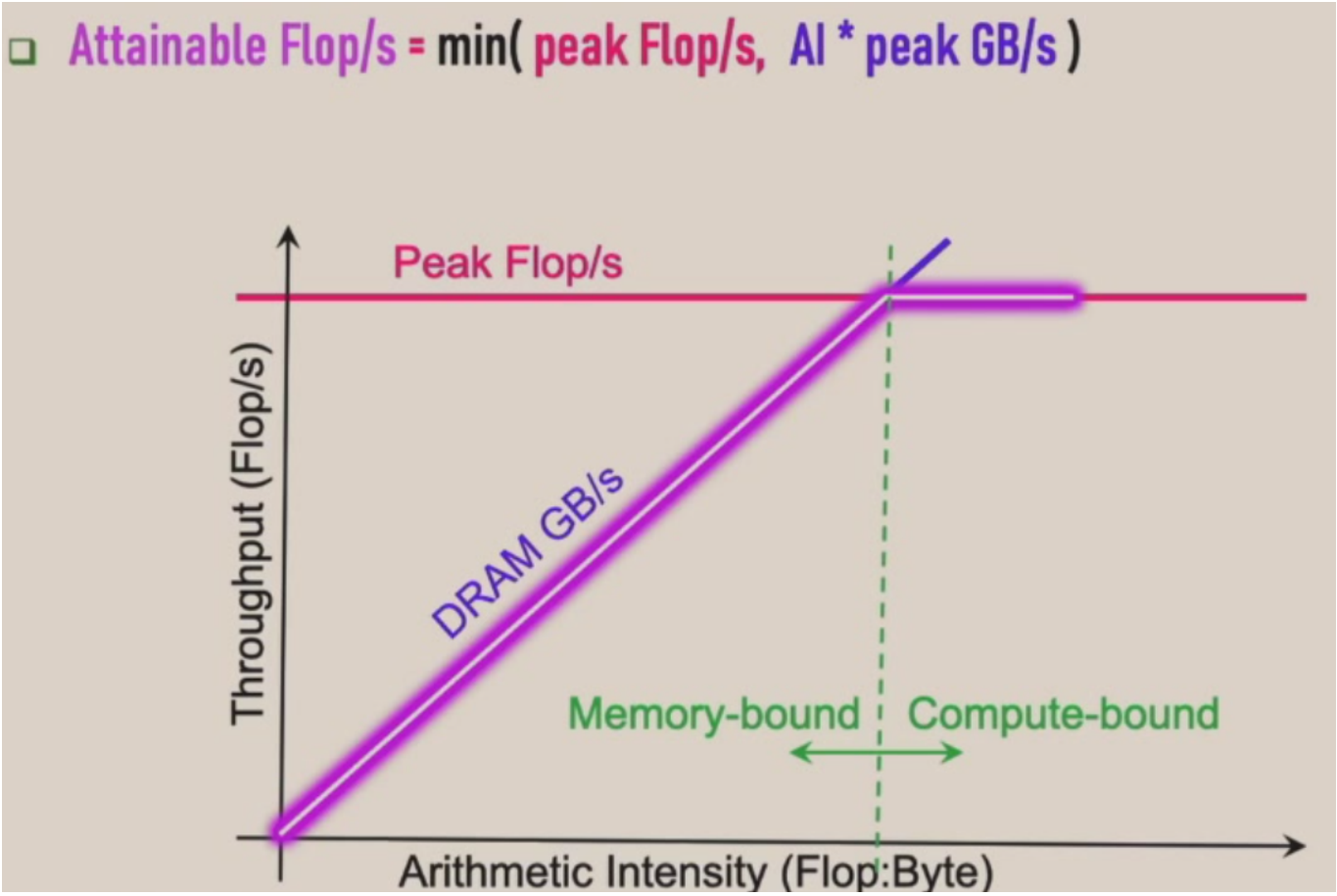
选择Roofline Model的三个原因

- 1, computing regime: Latency-limited → throughput-limited
  - Original latency-oriented performance model does not work
- 2, Target processor's perspective
  - Showing inherent hardware limitations (or bound), in term of compute and memory
- 3, Compute kernel's perspective
  - Showing the priority of optimizations for a given compute kernel running on a given processor

Arithmetic Intensity(AI:计算密度)：单位内存需要的算力

- Definition: AI = Total Flops / Total Memory Bytes
- Arithmetic intensity describes the characteristics of a compute kernel running on a given processor
  - Large AI → Compute-bound
  - Small AI → Memory-bound

实际图



1. 定义与核心区别

Arithmetic Intensity	Throughput
每字节数据移动（如内存访问）对应的计算量（FLOP/Byte）。	单位时间内完成的任务量（如FLOPS、IOPS）。
计算与数据移动的平衡（是否受限于内存带宽）。	**绝对性能**（系统能处理的最大速率）。
FLOP/Byte	FLOPS（浮点运算/秒）、TOPS（操作/秒）等。

---

2. 具体差异

(1) Arithmetic Intensity（算术强度）

- **本质**：衡量算法或计算任务的**计算受限（Compute-Bound）**还是**内存受限（Memory-Bound）**。
  - **高算术强度**：大量计算但数据移动少（如矩阵乘法、深度学习训练）。
    示例：AI模型的矩阵运算可能达到100 FLOP/Byte。
  - **低算术强度**：频繁数据访问但计算简单（如向量加法、稀疏计算）。
    示例：流媒体处理可能仅0.1 FLOP/Byte。
- **用途**：
  - 指导优化（如优化内存访问、选择适合的硬件）。
  - 预测性能上限（根据硬件带宽计算理论峰值性能）。

(2) Throughput（吞吐量）

- **本质**：描述系统在单位时间内的**最大处理能力**，与资源利用率无关。

3.Little's Law

Little's Law: L = λ * W (buffer size = throughput * latency)

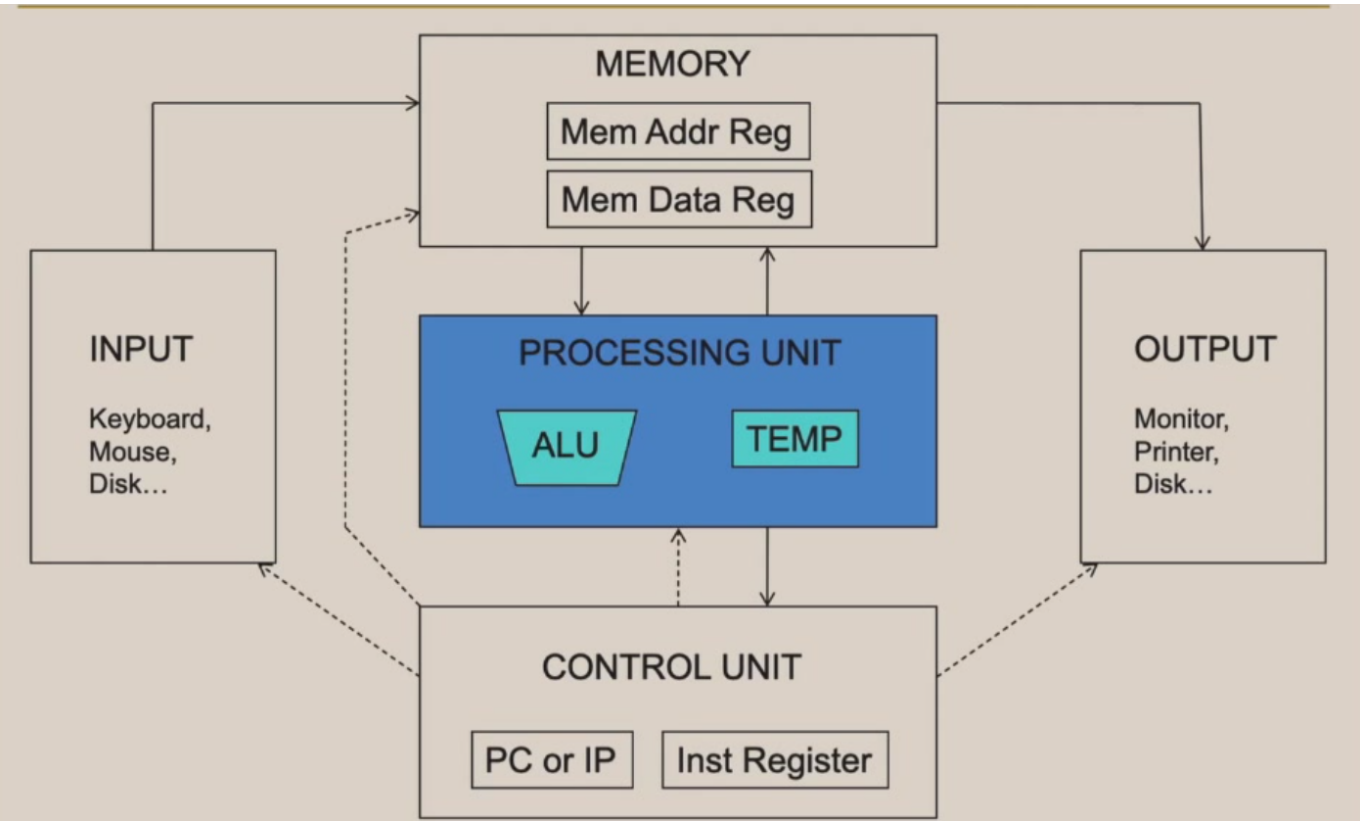
Image the services provided by counters in the bank.

- **Arrival rate**: one customer/min;
- **Counter's average serve time**: 6 mins;
- **Question**: how many counters are needed for people who need the service? (**Cond: The customer will leave if no counter is available**.)
- **Answer**: 6 counters (one slot for one person, then no customer will leave).

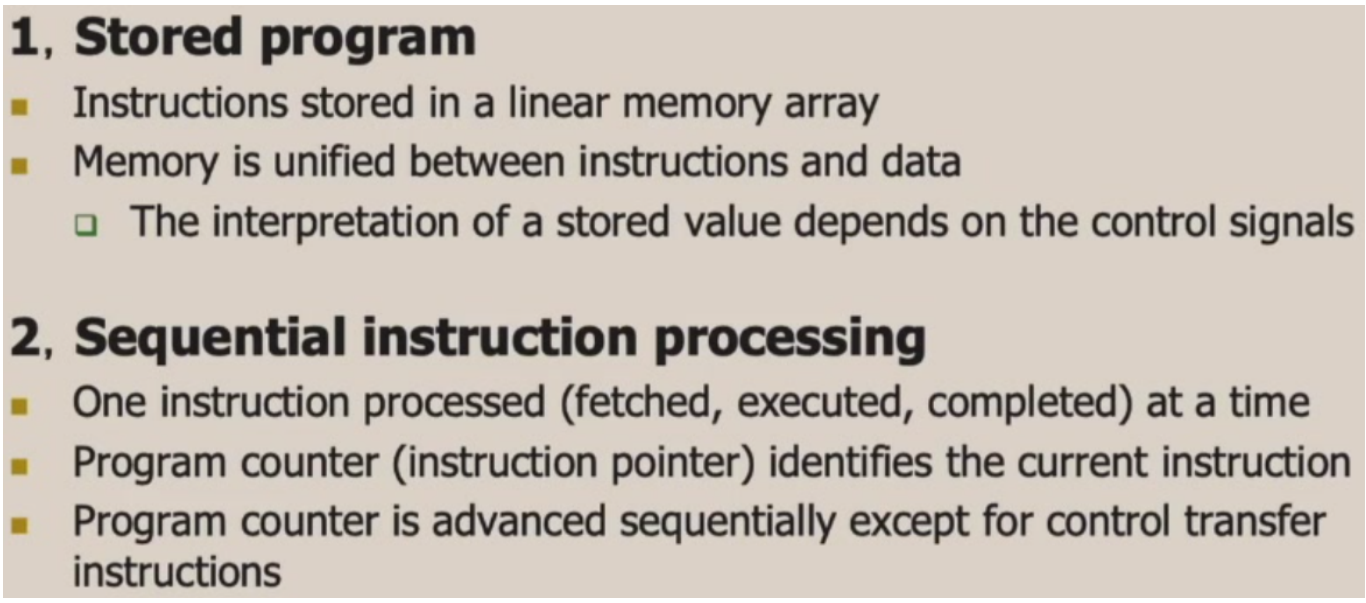
- **Throughput**: 12GB/s;
- **Latency**: 100ns;
- **Buffer Size** (concurrency): 100ns * 12GB/s = 120B

2.Pipeline Hazard + Reorder Buffer

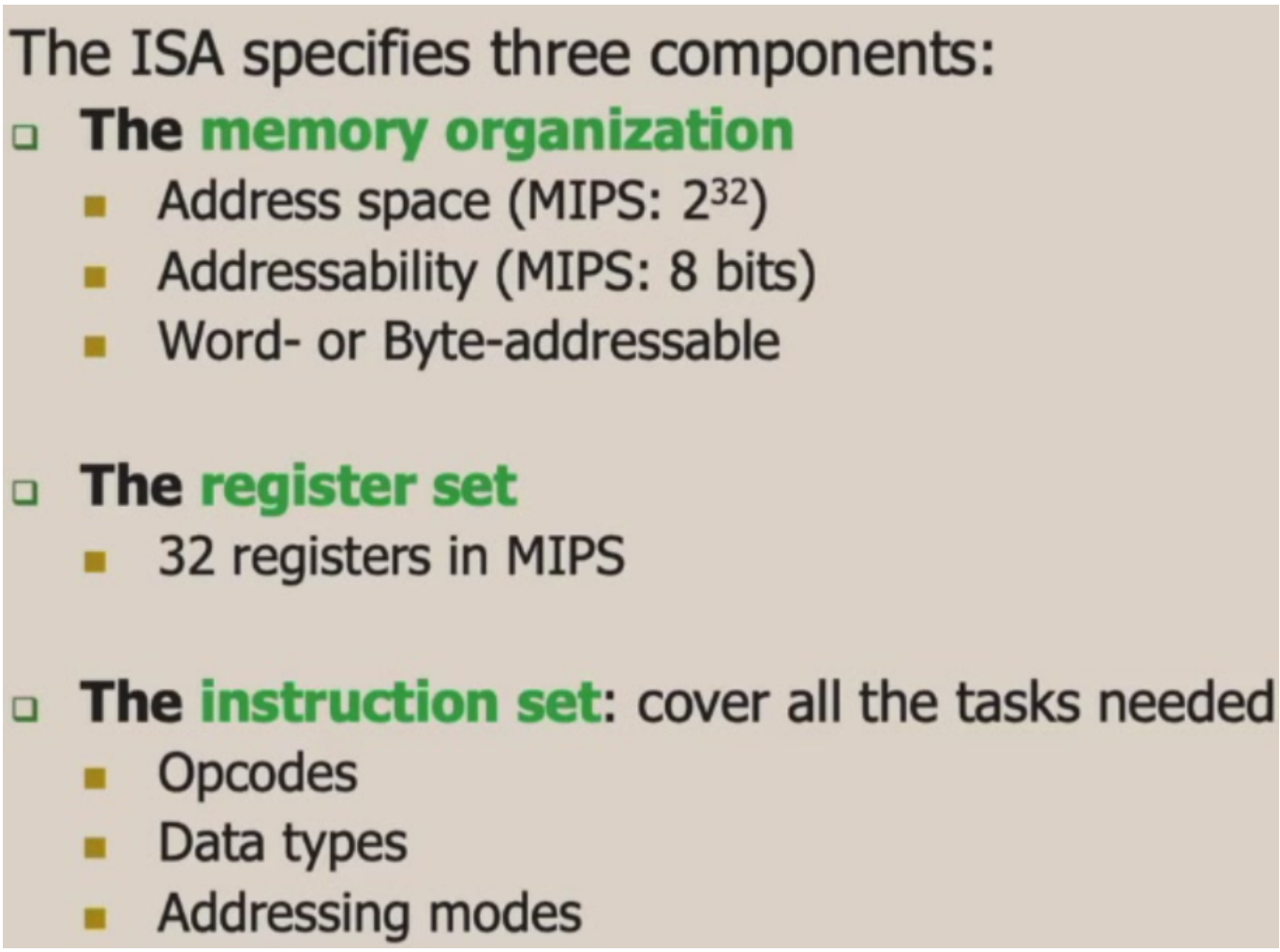
Von Neumann Model



2 key points of Von Neumann Model

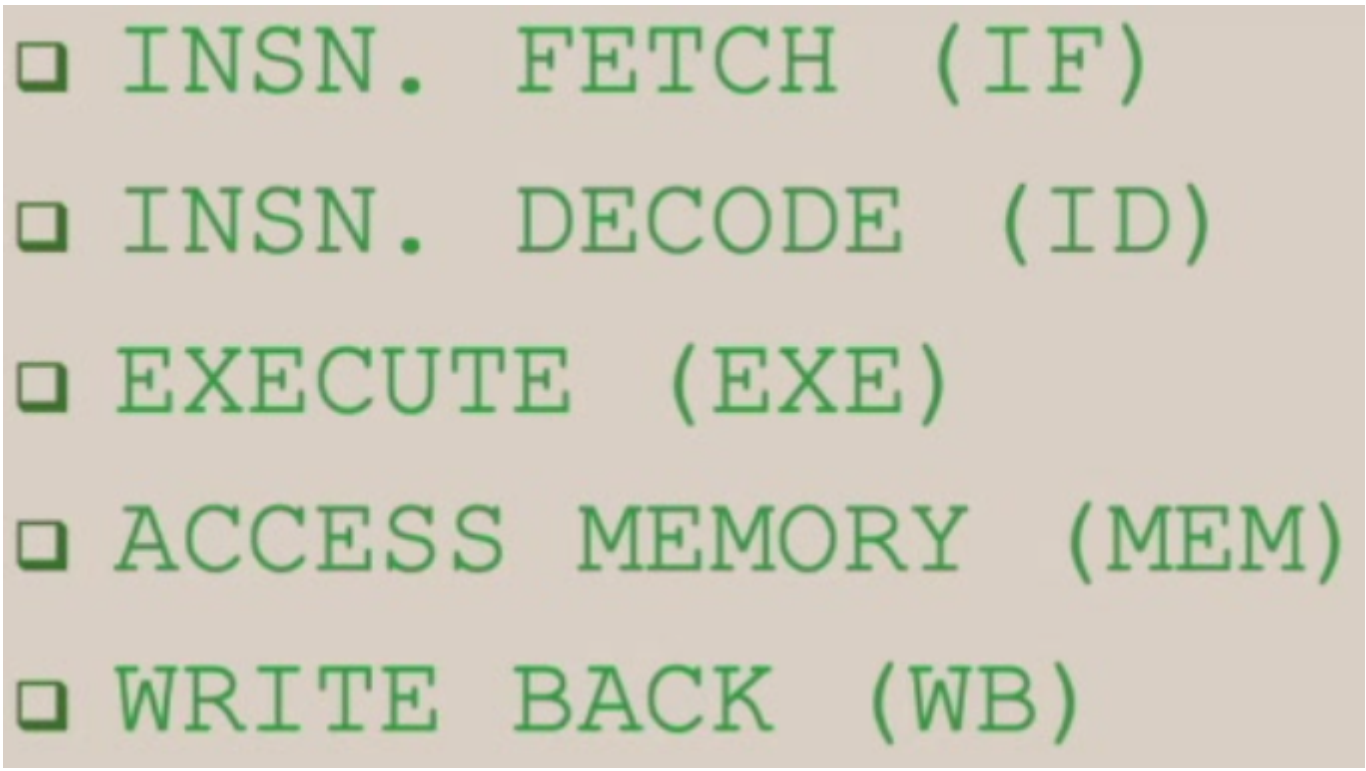


ISA



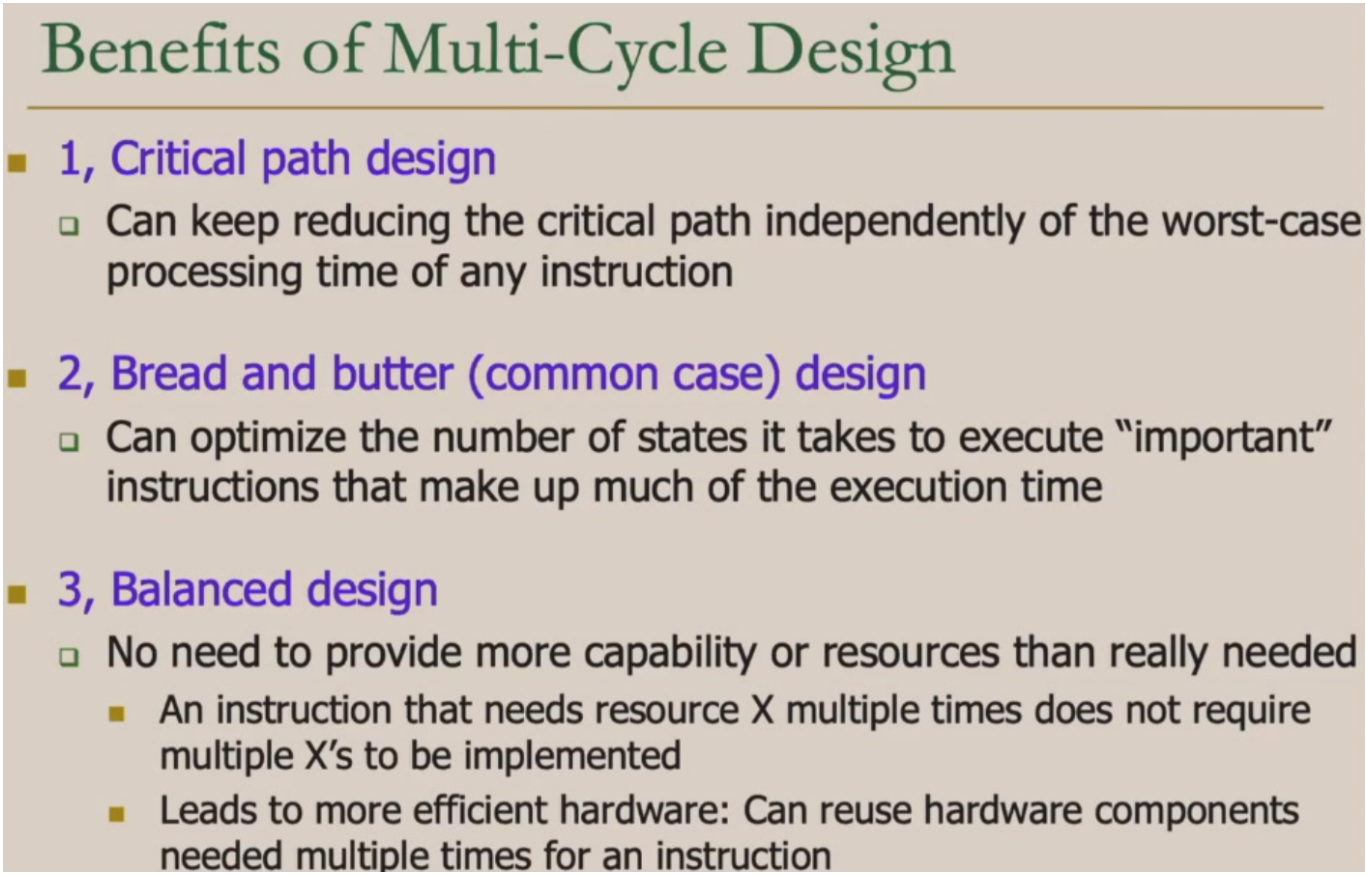
ABI:application binary interface between 2 binary program modules.

Instruction Cycle

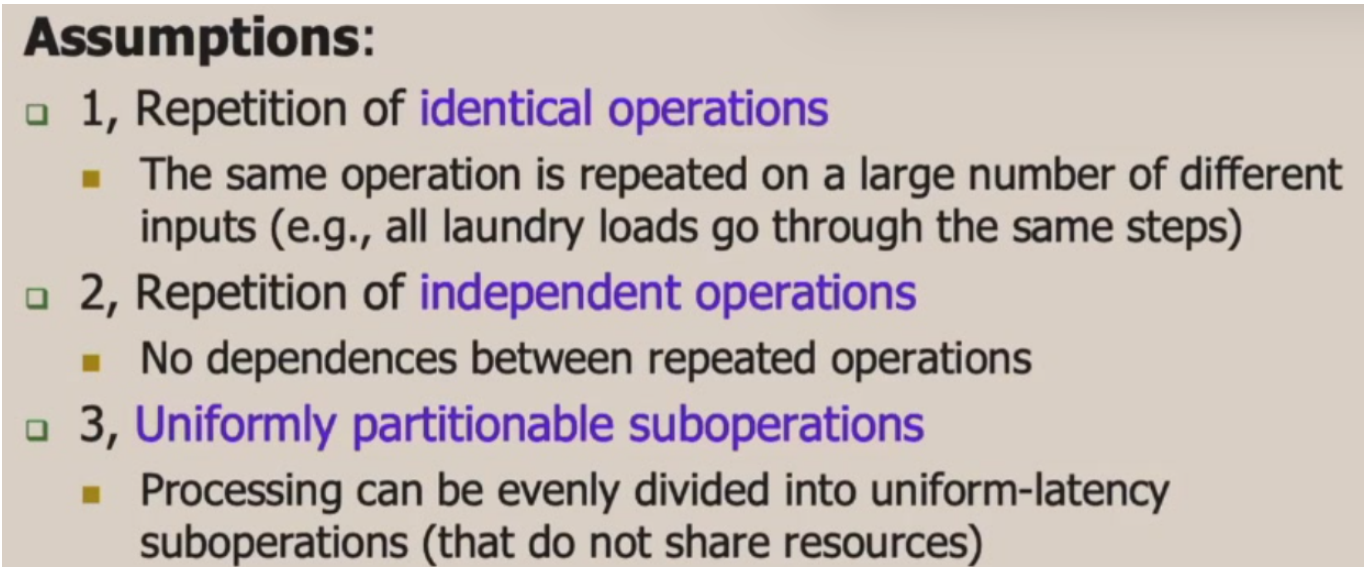


Single Cycle CPU

Multi-Cycle CPU



Pipeline CPU：increase throughput



Pipeline Hazard

structural hazard

data hazard

control hazard

Reorder Buffer

Multi-Cycle Execution

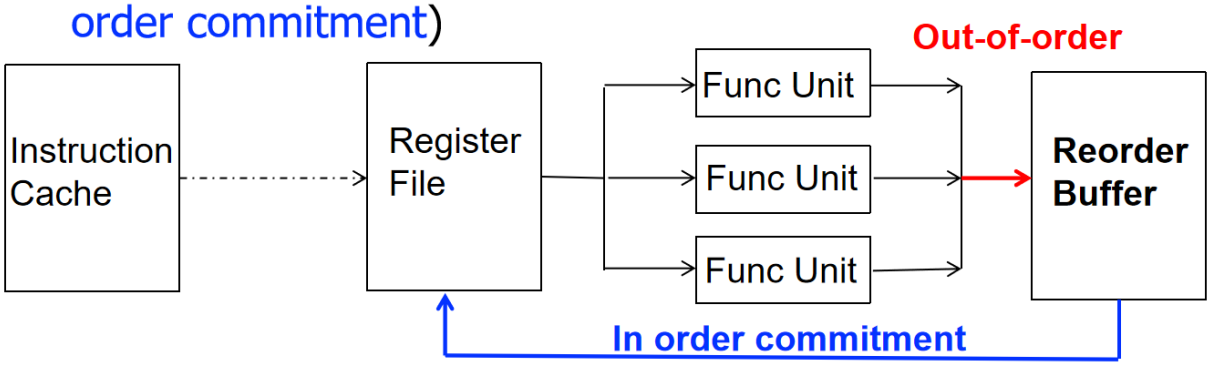
Not all instructions take the same amount of time for “execution”. Question: How to address multi-cycle execution issue? Answer: Have multiple different functional units that take different number of cycles

Exception and Interrupt

False Dependencies (WAW & WAR)

Definition of Reorder Buffer

- **Key Idea**: Complete instructions out-of-order, but reorder them before writing results to architectural state (Commit).
  - 1, When instruction is **decoded,** it reserves the next-sequential entry in the ROB.
  - 2, When instruction **completes out-of-order**, it writes result into ROB entry.
  - 3, When instruction **oldest in ROB** and it has completed without exceptions, its result writes to reg. file or memory (In order commitment)



这段话描述的是一种超标量处理器中指令乱序执行（Out-of-Order Execution）以及指令提交（Commit）的过程。下面是对每个步骤的中文解释：

key ideas

1. **当指令解码时，它会在重排序缓冲区（ROB）中预留下一个顺序的条目**：
   - 在指令解码阶段，处理器会为每个指令分配一个ROB条目，这个条目会按照指令到达的顺序预留位置，用于保存指令的执行状态和结果。虽然这些指令在执行时可能会乱序，但ROB会按顺序管理指令。
2. **当指令完成乱序执行时，它将结果写入ROB条目**：
   - 一旦指令完成了执行（可能是乱序完成的），它会将结果写入ROB的相应条目。这时，指令的结果已经准备好，但它并没有立即提交到寄存器文件或者内存中。
3. **当ROB中最老的指令完成并且没有异常时，它的结果会按照顺序写入寄存器文件或内存（顺序提交）**：
   - 当ROB中排在最前面的指令已经执行完成，并且没有发生任何异常时，它的结果会按照程序顺序提交到寄存器文件或内存中。这是一个顺序提交的过程，确保程序执行的顺序性。

ROB function

1. **正确地将指令重新排序回程序顺序**：
   - 重排序缓冲区（ROB）的一个重要功能是确保乱序执行的指令最终按照它们在程序中的原始顺序提交。尽管指令可能在执行过程中乱序执行，但ROB会负责按程序的顺序重新安排它们的结果。
2. **在指令可以没有任何问题地退休时，更新架构状态并写入指令结果**：

- 当指令在执行过程中没有发生任何异常（如溢出、缺页等），并且它已经完成执行，ROB将指令的结果更新到架构状态中。这通常意味着将结果写入寄存器文件或者内存，以便后续的指令可以依赖这些结果。

3. **如果需要处理异常/中断，在指令退休之前准确地处理它们**：
   - 如果在指令执行过程中发生了异常或中断，ROB需要确保在指令提交之前正确处理这些问题。这意味着如果指令执行时出现错误或中断，系统需要准确地捕获并处理这些问题，而不能让错误的指令被提交到架构状态中。
4. **使用有效位来跟踪结果的准备情况，并判断指令是否完成执行**：
   - ROB使用有效位（valid bits）来标识每个指令的执行结果是否准备好。有效位可以帮助判断指令是否已经完成执行并且可以提交。如果指令尚未完成，ROB会标记其状态为未完成；只有当指令完成且结果准备好时，才会将其提交。

实现

- Output and anti dependences are **not true dependences**
  - WHY? The same register refers to values that have nothing to do with each other
  - **They exist due to lack of register ID's (i.e. names) in the ISA**
- RB eliminates anti and output dependences
  - Gives the illusion that there are a large number of registers

具体来说，当指令解码时，处理器将寄存器ID"重命名"成一个物理寄存器ID，这个ID并不直接指向一个固定的架构寄存器，而是指向ROB中的一个条目。ROB条目会保存指令执行过程中该寄存器的结果。通过这种方式，处理器能够避免寄存器ID冲突，消除由于共享寄存器ID而产生的依赖问题。

如何使用ROB?

- **Idea**: Use indirection

- 1, Access register file first (check if the register is valid)
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry: Register file maps the register to a reorder buffer entry if there is an in-flight instruction writing to the register

- 2, Access reorder buffer next

1. **寄存器文件**：首先检查寄存器是否有效（是否存有最新的值）。
2. 如果寄存器的值无效，寄存器文件会存储该寄存器值所在的 **重排序缓冲区条目 ID**，即该寄存器正在等待的指令执行结果。
3. 然后，访问 **重排序缓冲区** 来获取该寄存器的实际值，这个值会在指令执行时产生，并最终写回寄存器文件。

只用ROB的问题：

由于真实数据依赖（True Data Dependence），会导致较年轻的指令（即后续执行的指令）在派发到功能单元（Functional Units，即执行单元）时发生阻塞（stalls）。

True Data Dependence（真实数据依赖）： 或者称为流依赖（Flow Dependence），发生在两条指令之间，当一条指令的输出（写入的结果）是另一条指令所需要的输入（读取的操作数）时。也就是说，后一条指令依赖于前一条指令的结果。这种依赖关系表示程序中必须按顺序执行这两条指令。例如，指令A的结果需要作为指令B的输入，这就是一种真实数据依赖。

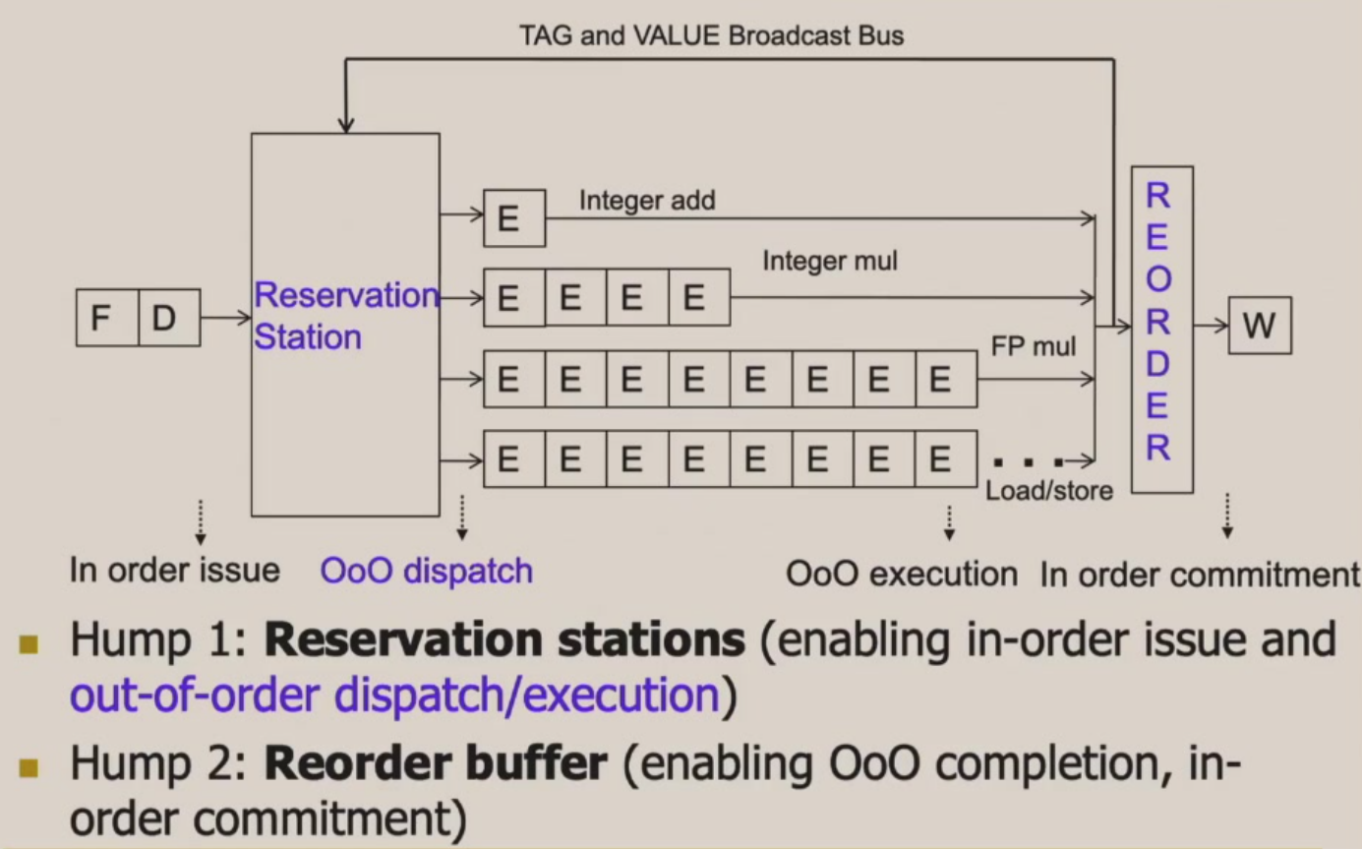
- **Problem**: **in-order** dispatch (scheduling, or execution)

- **Solution**: **out-of-order** dispatch (scheduling, or execution)

3.Tomasulo 算法

Tomasulo's Algorithm 是一种用于 **乱序执行（Out-of-Order Execution）** 的硬件调度算法，旨在提高处理器的吞吐量和资源利用率。它通过动态地重新调度指令，减少数据依赖引起的停顿，从而加快指令的执行速度。

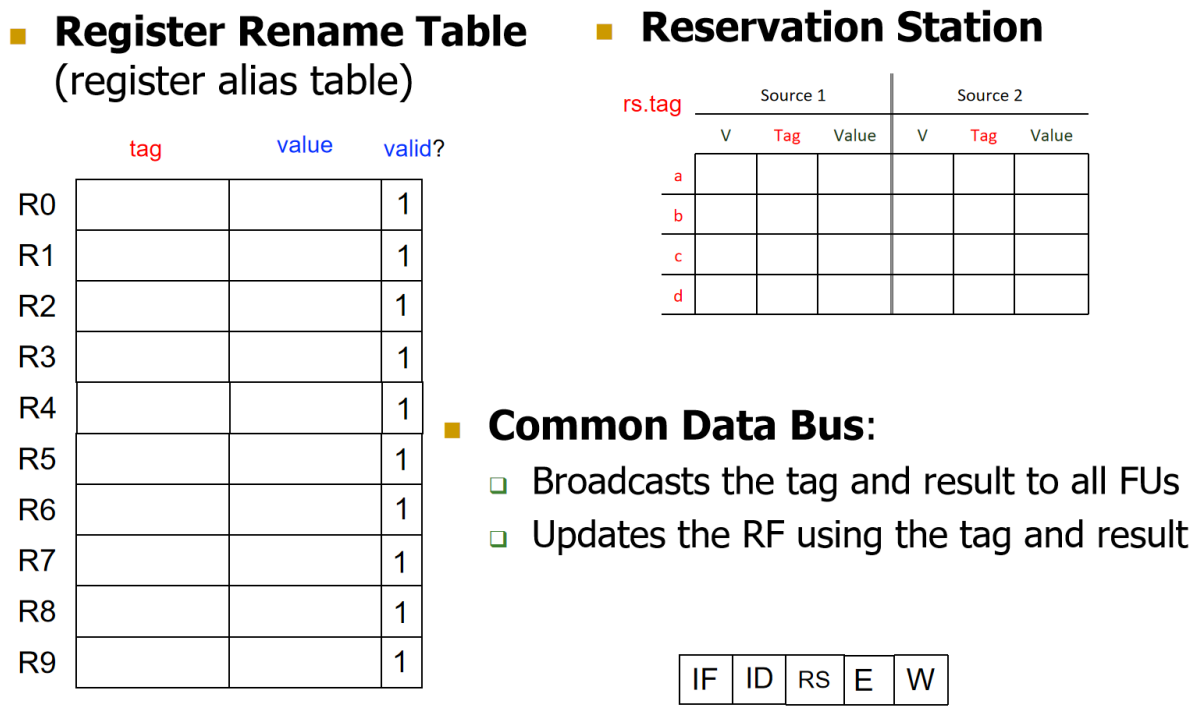
Tomasulo's Algorithm的工作原理



Tomasulo的算法的基本思想是，通过**寄存器重命名**和**预约站**来动态地调度和执行指令。它使用了几个关键组件来实现乱序执行：

1. **保留站（Reservation Stations）**：
    - 保留站用于存储尚未执行但已经解码的指令。每个功能单元（如加法单元、乘法单元）都有一个对应的保留站。
    - 每个保留站存储着操作数和操作指令，直到操作数准备好（即没有数据依赖）时，指令才会被派发到功能单元执行。
2. **寄存器重命名（Register Renaming）**：
    - 处理器通过寄存器重命名来解决数据依赖问题。寄存器重命名通过为每个目标寄存器分配一个唯一的标识符（称为**标签**），避免了寄存器之间的直接依赖冲突。
    - 这意味着两个指令可以写入相同的物理寄存器，而不会发生冲突。指令的输出可以写入新的物理寄存器（重命名后的寄存器），而不是原始寄存器。
3. **Common Data Bus (CDB)**：
    - CDB是一个共享总线，它用于将执行单元的结果传输到寄存器和保留站。如果某个指令完成了执行，它的结果将通过CDB广播到其他指令所需的位置。

- 保留站和寄存器都可以通过CDB接收结果，并继续执行。



Register Rename Table：tag对应Reservation Station里的rs.tag一列，value是原来的值。如果valid=1，说明value值没有被改变，直接可被读走；但如果valid是0，则需要根据tag去Reservation Station里找。

Reservation Station:tag对应的是Reservation Station里rs.tag里的其他数据，代表本条指令需要其它指令的结果。

Tomasulo's Algorithm的执行流程

Tomasulo's Algorithm

- **ID**: If reservation station entry available before renaming dest. register
  - Occupy a RS entry for the instruction
  - **For each source register in the RS entry:** if the valid bit of source register in RF is 1, RS.source.v = 1 and RS.source.value=source register; else RS.source.v = 0 and RS.source.tag = source register.tag.
  - **For dest. register in RF**: Rename to the tag of the corresponding RS entry, set the valid bit to 0.

  Else stall
- **RS**: While in reservation station, each instruction:
  - **Update**: Watches common data bus (CDB) for tag of its sources. When tag seen, grab value for the source and keep it in the reservation station (.v = 1).
  - **Issue**: When both operands available, instruction ready to be dispatched to FU
- **EXE**: Execute the instruction in FU, produce its broadcast tag and value
- **WB**: After instruction finishes in the Functional Unit
  - a, Arbitrate for CDB
  - b, Put broadcast tag and its broadcast value onto CDB (tag broadcast)
  - c, Update register file connected to the CDB
    - If the tag in the RF matches the broadcast tag, write broadcast value into register (and set valid bit)
  - d, Update reservation station connected to the CDB
    - If the broadcast tag matches the tag of any source in a RS entry, write the broadcast value to the source and set the valid bit of the source.

24

1. 保留站条目分配与寄存器处理

- **条件检查**

  当指令进入流水线时，首先检查保留站是否有空闲条目：
  - **有空闲条目**：占用该条目，继续处理。
  - **无空闲条目**：流水线停顿（Stall），直到保留站释放条目。
- **源寄存器处理**

  对指令的每个源寄存器，检查寄存器文件（RF）中的有效位：
  - **有效位为1**：直接读取值到保留站条目，标记为有效（`RS.source.v = 1`）。
  - **有效位为0**：记录该寄存器的标签（`RS.source.tag`），标记为无效（`RS.source.v = 0`），后续需通过公共数据总线（CDB）监听该标签的值。
- **目标寄存器重命名**

  将目标寄存器重命名为当前保留站条目的标签（例如`RS5`），并在RF中：
  - 更新目标寄存器的标签字段为保留站条目标签。
  - 设置目标寄存器的有效位为0（表示值未就绪）。

2. 保留站中的指令等待

- **监视公共数据总线（CDB）**
  指令在保留站中等待时，持续监视CDB：
  - 若某个源寄存器的标签出现在CDB上，立即捕获对应值，更新保留站中的源字段，并标记为有效（`v = 1`）。
- **发射条件**
  当所有源操作数的有效位均为1（`v = 1`），指令准备好被派遣到功能单元（FU）执行。

---

3. 执行阶段（EXE）

- **功能单元执行**
  指令在FU中执行，生成结果值及其对应的标签（即保留站条目标签）。

---

4. 写回阶段（WB）

- **仲裁CDB使用权**
  多个FU可能同时完成执行，需仲裁决定哪个结果优先广播到CDB。
- **广播结果**
  将标签和结果值通过CDB广播，供所有相关组件捕获。
- **更新寄存器文件（RF）**
  - 若RF中某个寄存器的标签与广播标签匹配，将值写入该寄存器，并设置有效位为1（`valid = 1`）。
- **更新保留站（RS）**
  - 若保留站中的源字段标签与广播标签匹配，更新该字段的值并标记为有效（`v = 1`）。

```
# 重点！！！
# 具体看PPT的一个例子！
# 人工智能芯片与系统 2024-03-12第3-5节 1小时15分开始
```

Tomasulo算法的优势

1. **提高指令并行性**：
   - Tomasulo通过乱序执行和寄存器重命名，能够有效地消除数据依赖带来的性能瓶颈，从而提高指令并行性。
2. **减少停顿**：
   - 通过保留站、寄存器重命名和CDB，Tomasulo算法能够让指令在依赖数据准备好之前并行执行，最大限度地减少停顿时间。
3. **解决数据冲突**：

- 通过寄存器重命名和保留站管理，Tomasulo有效地解决了多个指令可能依赖同一个寄存器值的问题。

4.Superscalar + SIMD + Multi-core

Superscalar超标量

定义：每个cycle执行多条指令。
IPC：instruction per cycle。
存在的问题：1.需要硬件实现； 2.解决冲突很复杂。
（超标量和乱序执行两者没有任何关系）

Advantages

Higher instruction throughput

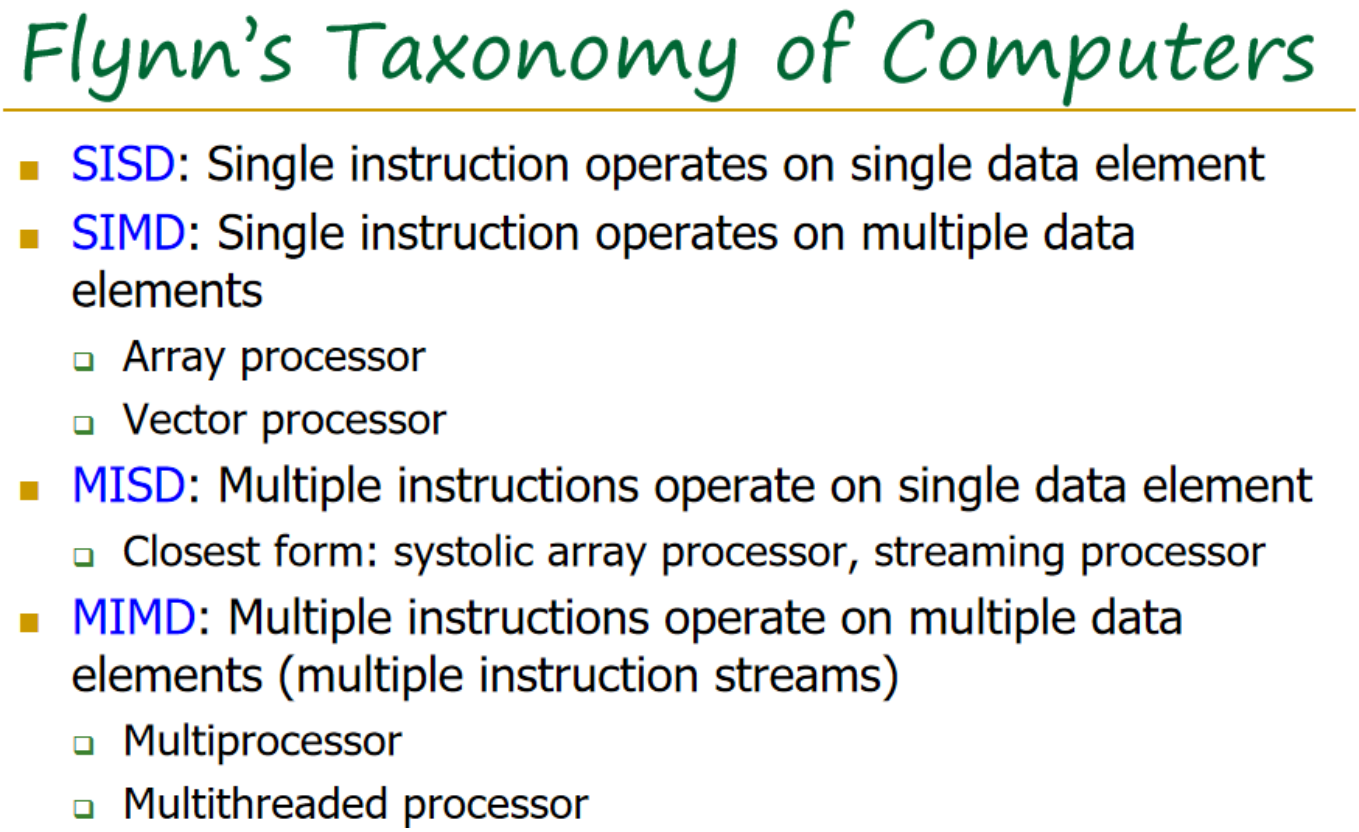
Disadvantages

Higher complexity for dependence checking
More hardware resources needed

乱序执行和超标量都无法影响Roofline Model！

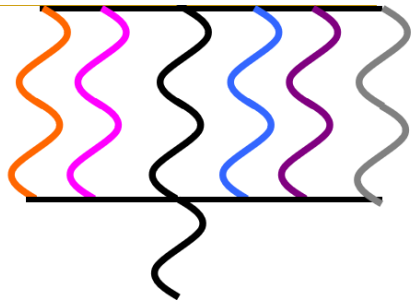
Vector Instruction 向量指令

CPU的4种分类



一个很严重问题：智能加速并行部分，串行部分成为限制提速的最大瓶颈。

- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors



$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Maximum speedup limited by serial portion: Serial bottleneck

比如：一个周期能做四个加法（并行），但是每个周期只能load一个数据，那么就会受限制。（实质上：去CPU上跑"并行"本身就不合适。）

向量指令基本概念

向量指令是一种利用SIMD（单指令多数据）技术，允许单条指令同时对多个数据元素进行相同操作的CPU指令。 具体而言，需要增加ALU的数量，实现并行。 此外需要修改两样东西： **1**： 寄存器文件：在原有的标量寄存器文件的基础上，新增一个向量寄存器文件。 **2**： 内存：新增读写口。

Memory:
1, array of storage locations
indexed by an address;
2, Multiple bank design.

Registers:
- General purpose register file
- Vector register file

SIMD可以提升roofline model的性能（因为提升了ALU的数量）

Multithreading（SMT：同步多线程）

同一个核跑多个线程（目前通用的是一个核跑两个线程）： **核心思想**
硬件设计采用多线程上下文（程序计数器PC+寄存器组），每个时钟周期从不同线程抓取指令。当某线程的分支/指令结果尚未解析完成时，流水线会暂停从该线程抓取新指令，转而执行其他线程的指令。通过这种方式，将某个线程的延迟（如分支预测失败、数据依赖等待）与其他线程的有效工作重叠，从而隐藏延迟。 **工作原理**

1. **轮询式线程切换**
   - 每个时钟周期，取指单元切换到不同线程，确保同一线程的指令不会同时在流水线中并发执行。
   - 例如：若线程A的指令因分支未解析而停滞，后续周期会抓取线程B、C、D的指令，直到线程A的分支解析完成。
2. **消除线程内依赖**
   - 由于同一线程的指令不会同时在流水线中，无需处理线程内的控制依赖（如分支跳转）和数据依赖（如寄存器写后读冲突），硬件设计得以简化。

优点（Advantages）

1. **无需指令间依赖性检查**

- **原因**：同一时刻，流水线中每个线程最多只有一条指令在执行（例如，线程A的第N条指令必须执行完成后，第N+1条指令才会进入流水线）。
- **效果**：由于指令之间没有重叠执行，**线程内的数据依赖（如写后读RAW）和结构依赖（如功能单元冲突）自然消失**，硬件无需复杂的依赖检测逻辑（如乱序执行中的动态调度）。

2. **无需分支预测逻辑**

- **原因**：如果某线程遇到分支指令，流水线会直接切换到其他线程，直到该分支结果解析完成。
- **效果**：分支延迟被其他线程的计算覆盖，无需预测分支方向或处理预测失败的惩罚（如清空流水线）。

3. **利用空闲周期执行其他线程的指令**

- **原因**：当某线程因缓存未命中、分支停滞或数据依赖暂停时，原本空闲的流水线阶段（称为“气泡”）会被其他线程的指令填充。
- **效果**：显著提高流水线利用率，减少资源浪费。例如，线程A等待内存时，线程B的指令可以占用执行单元。

4. **提升系统吞吐量和延迟容忍能力**

- **原因**：通过多线程并行掩盖单个线程的延迟（如内存访问、分支解析）。
- **效果**：整体系统吞吐量（单位时间完成的任务数）提高，尤其适合高并发场景（如服务器、图形渲染）。

缺点（Disadvantages）

1. **硬件复杂度增加**

- **具体开销**：
  - 每个线程需要独立的硬件上下文（程序计数器PC、寄存器文件、状态寄存器等）。
  - 需要线程选择逻辑（如轮询调度器）决定每个周期从哪个线程取指。
- **影响**：芯片面积和功耗上升，设计验证成本增加。

2. **单线程性能下降**

- **原因**：单个线程的指令被分散到多个周期执行（例如，一个4线程系统中，每个线程每4个周期才能取一条新指令）。
- **影响**：单线程任务的执行时间延长，不适合延迟敏感型应用（如实时控制系统）。

3. **线程间资源争用**

- **表现**：
  - **缓存争用**：多个线程共享缓存，可能导致频繁的缓存驱逐（如线程A的数据被线程B覆盖）。
  - **内存带宽争用**：多线程同时访问内存时，带宽可能成为瓶颈。
- **影响**：实际性能提升可能低于理论预期，需通过优化缓存策略缓解。

4. **跨线程的依赖检查残留**

- **问题**：
  - **存储一致性**：需确保不同线程的访存操作顺序正确（如线程A写入内存后，线程B需看到最新值）。
  - **同步操作**：若线程间需要通信（如共享数据），仍需硬件支持原子操作或内存屏障。

- **影响**：仍需少量依赖检查逻辑（如Load/Store队列的冲突检测），但复杂度低于单线程乱序执行。

不会提升roofline model，因为只是提升了利用率，而没有提升alu数量。

Multi-Core

与单核超标量相比，多核架构的优势与劣势

优势：

1. **核心更简单**：
   - **能效更高**：单个核心复杂度低，功耗更优。
   - **设计更容易**：核心可模块化复制，减少设计成本。
   - **频率更高**：结构更小、连线更短，时钟频率可提升。
2. **多任务吞吐量更高**：
   - **减少上下文切换**：多程序并行运行时，核心可独立执行任务，降低切换开销。
   - **提升并行应用性能**：若程序能分解成多线程/进程，多核可显著加速计算。

劣势：

1. **依赖并行编程**：
   - 必须显式编写多线程/并行代码才能利用多核性能，否则单线程任务无法加速。
2. **资源竞争影响单线程性能**：
   - 共享缓存、内存带宽等资源可能被其他核心占用，导致单线程性能下降。
3. **共享资源管理复杂**：
   - 缓存一致性（Cache Coherence）、内存访问冲突、总线仲裁等需额外硬件/软件支持。
4. **I/O 带宽限制（引脚瓶颈）**：
   - 核心数增加后，数据需求（如内存访问、I/O）可能超出芯片引脚（Pins）的供给能力，形成瓶颈。

能够提升roofline model

5.Memory

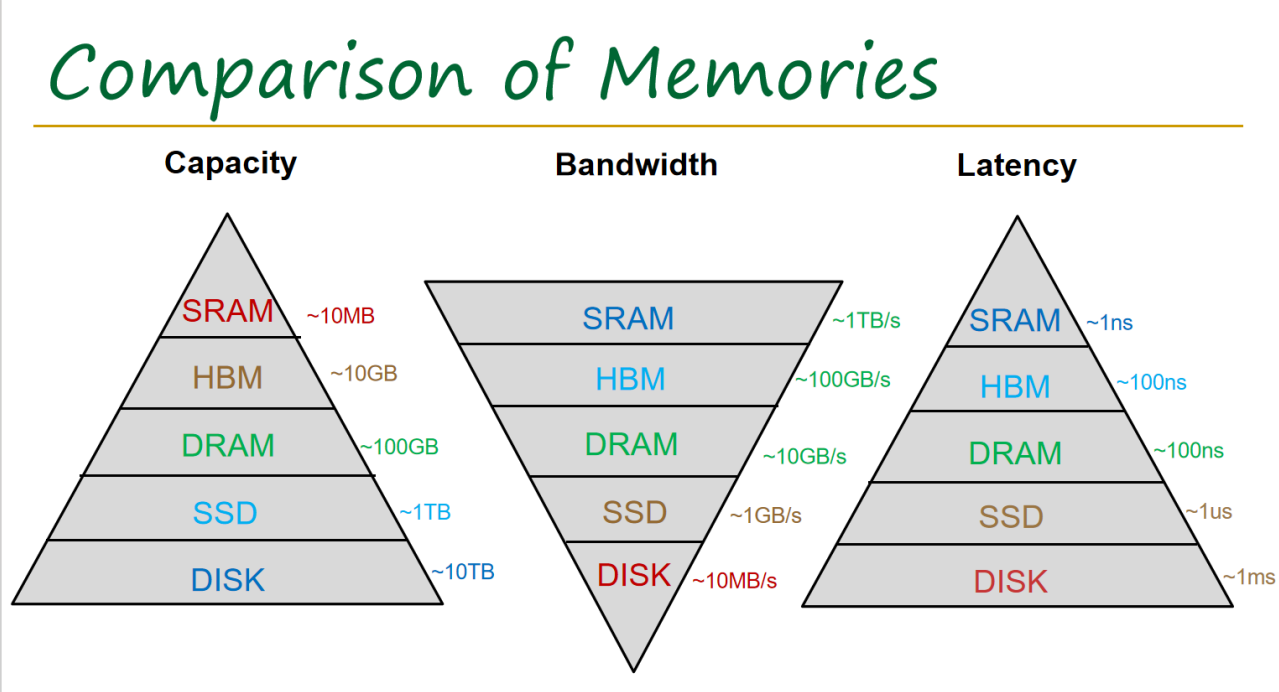
理想的内存：

- **Four properties of ideal memory:**
  - Zero latency: zero access time
  - Infinite capacity: no swap out
  - Infinite bandwidth: to support multiple accesses in parallel
  - Zero cost: provide as many as needed

但是各条件之间存在矛盾：

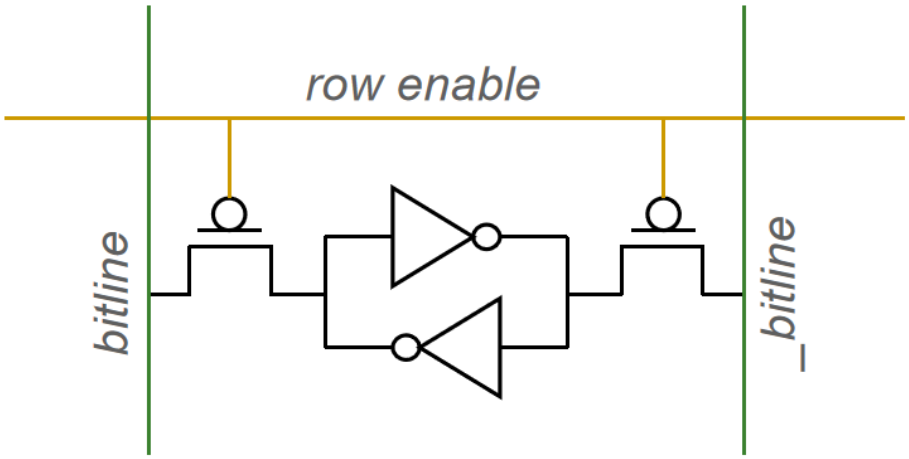
- Bigger is slower
- Faster is more expensive

- Higher bandwidth is more expensive



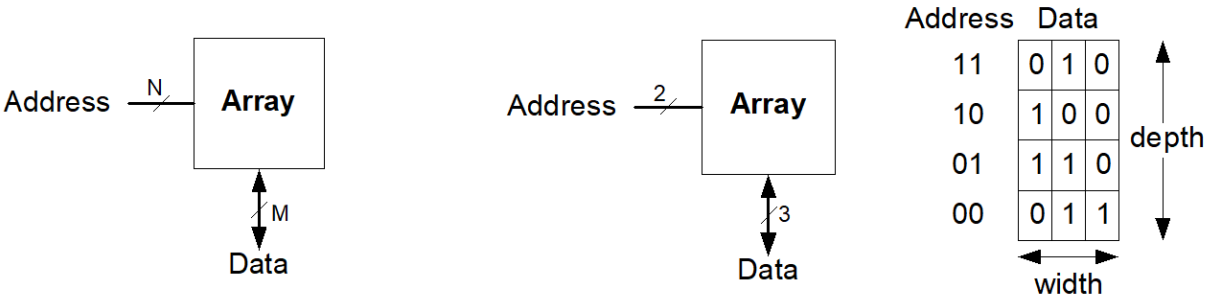
需要记住大概的大小

SRAM

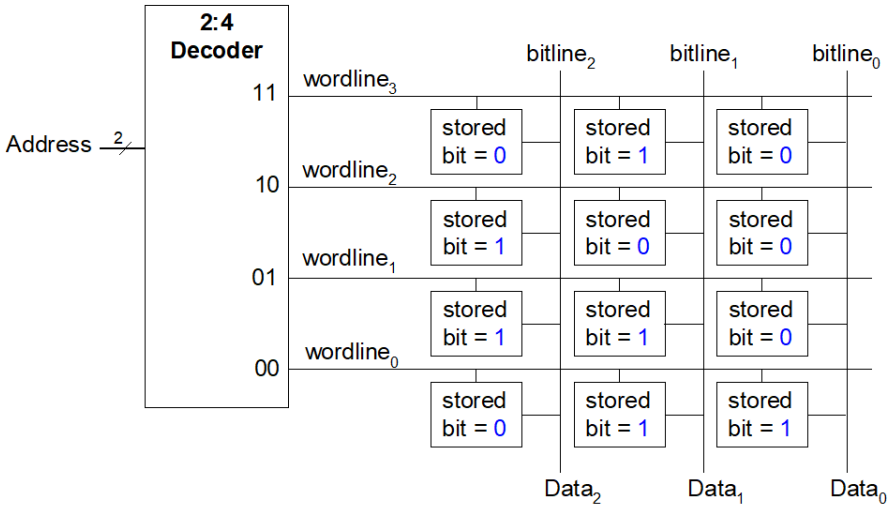


Memory结构

- Two-dimensional array of bit cells
  - Each bit cell stores one bit
- An array with N address bits and M data bits:
  - 2^N rows and M columns
  - **Depth**: number of rows (number of words)
  - **Width**: number of columns (size of word)
  - **Array size**: depth × width = 2^N × M

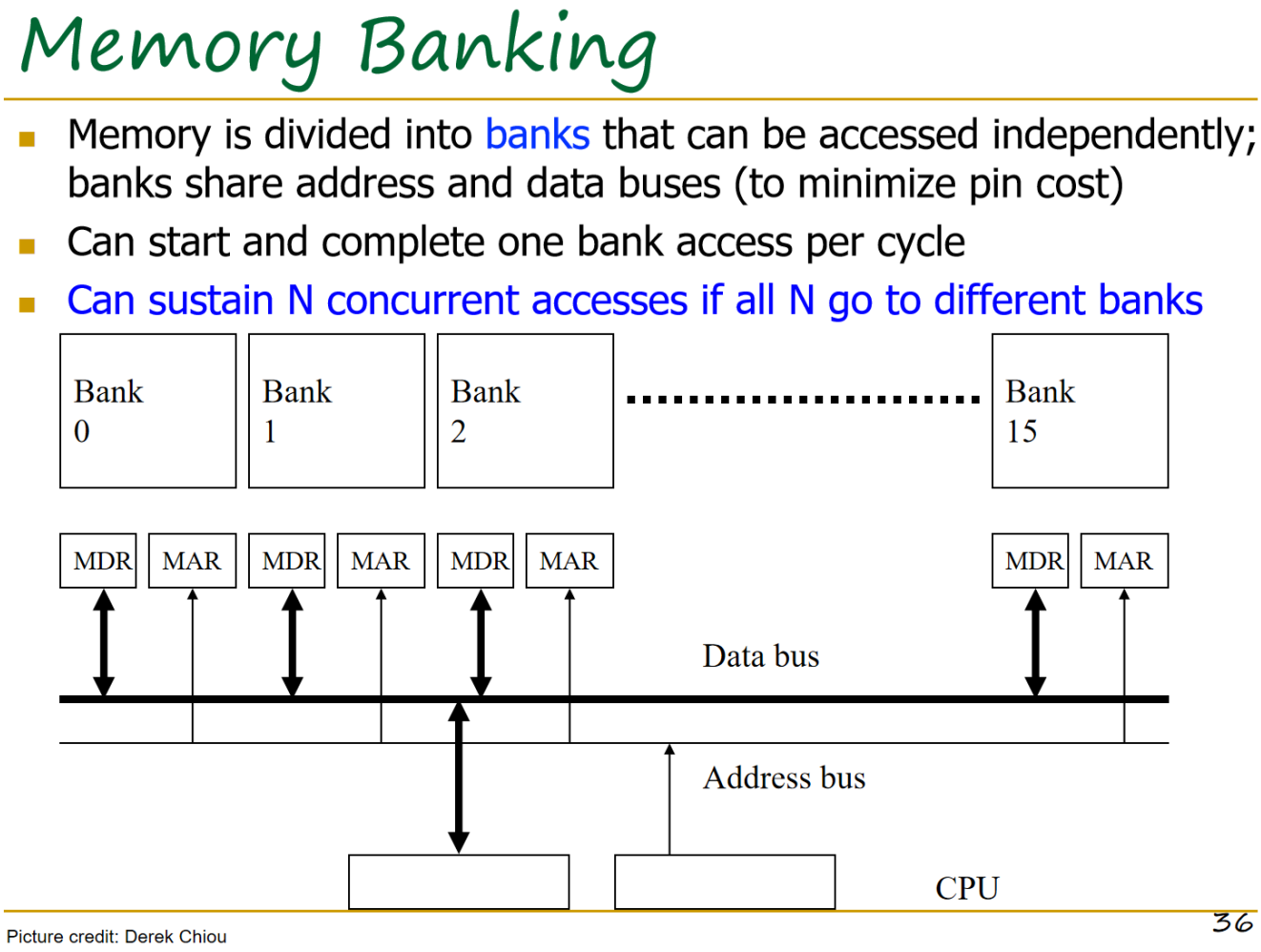


- Memory Array:
  - **Bitline**: Storage nodes in one column connected to one bitline
  - **Wordline**: Address decoder activates only ONE wordline, content of one line of storage available at output



Memory Bank

把内存分为n块，可以同时访问、同时读写。



SRAM使用的地方

DRAM

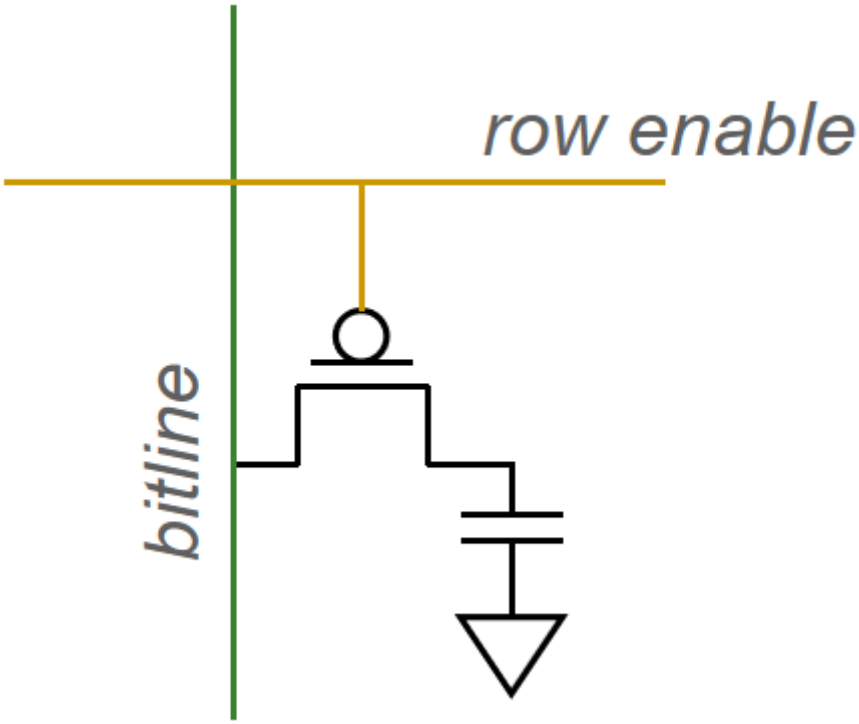
比如手机电脑的内存16G指的都是DRAM。

Motivation and Goals

- Application Perspective 各种现有的大模型都对内存有需求。
- Performance Perspective 研究发现内存才是限制算力的bottleneck。
- Energy Perspective 内存的访问才是最消耗能量的（而不是浮点数运算等）。
- Reliability Perspective 同面积下（内存）芯片数量越多，越容易出错。因此要提高内存的质量。

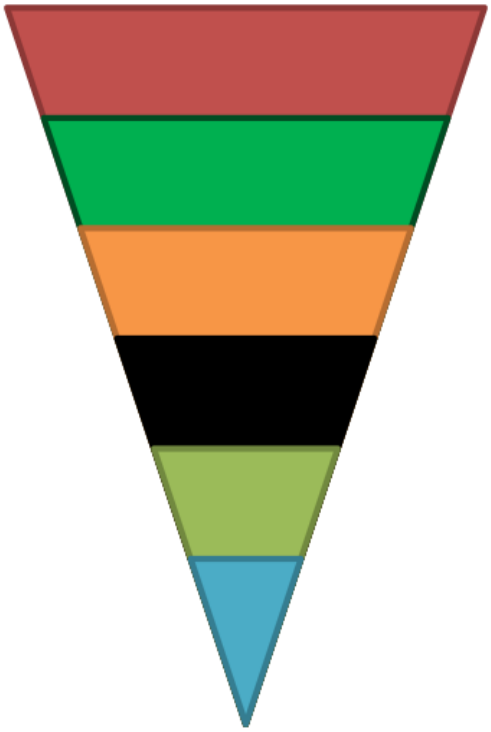
Background and Architecture of Memory

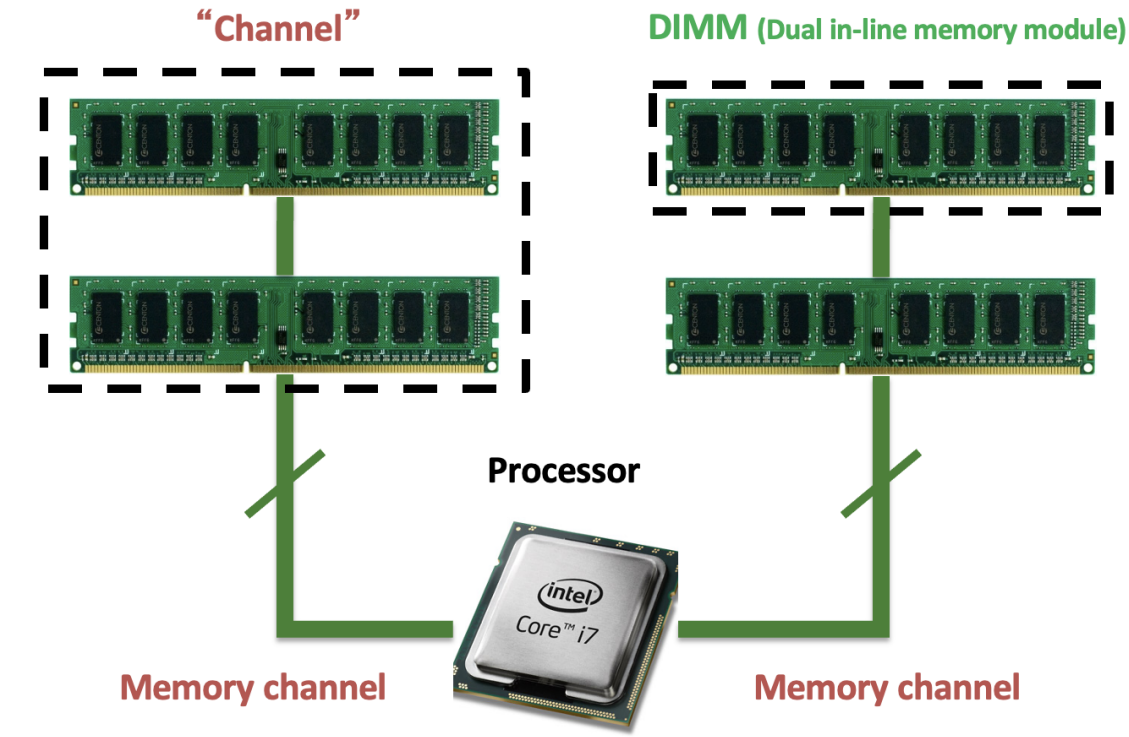
- High Level Abstraction OS将内存区分为物理内存和虚拟内存。

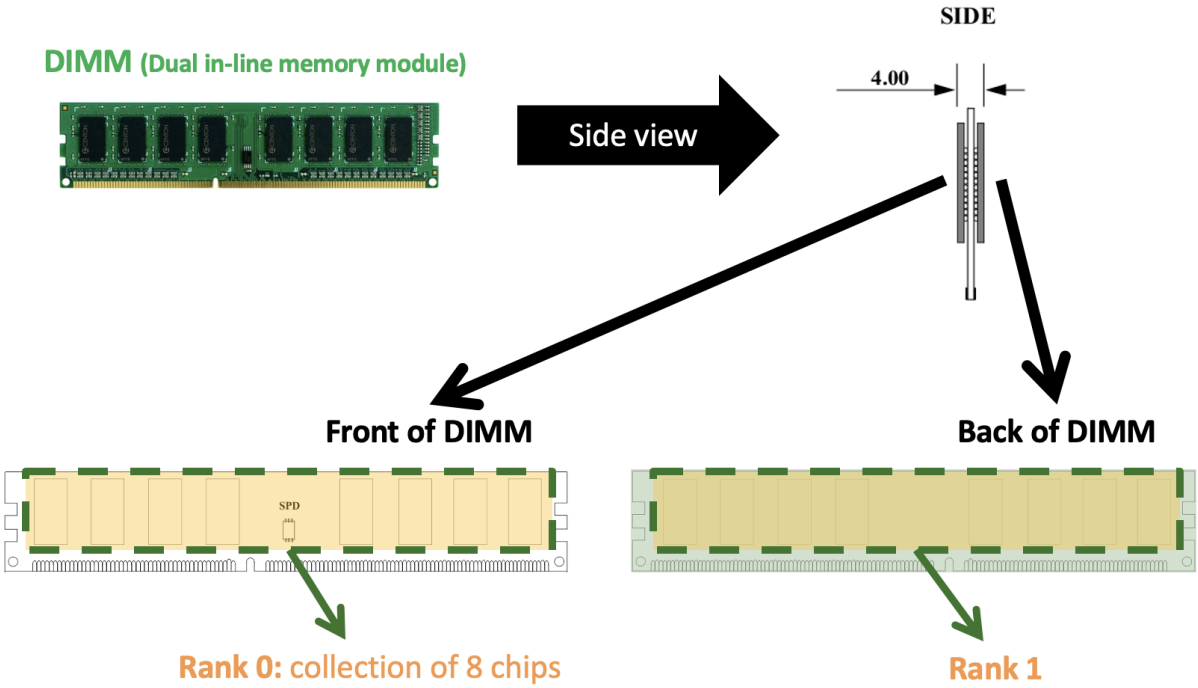


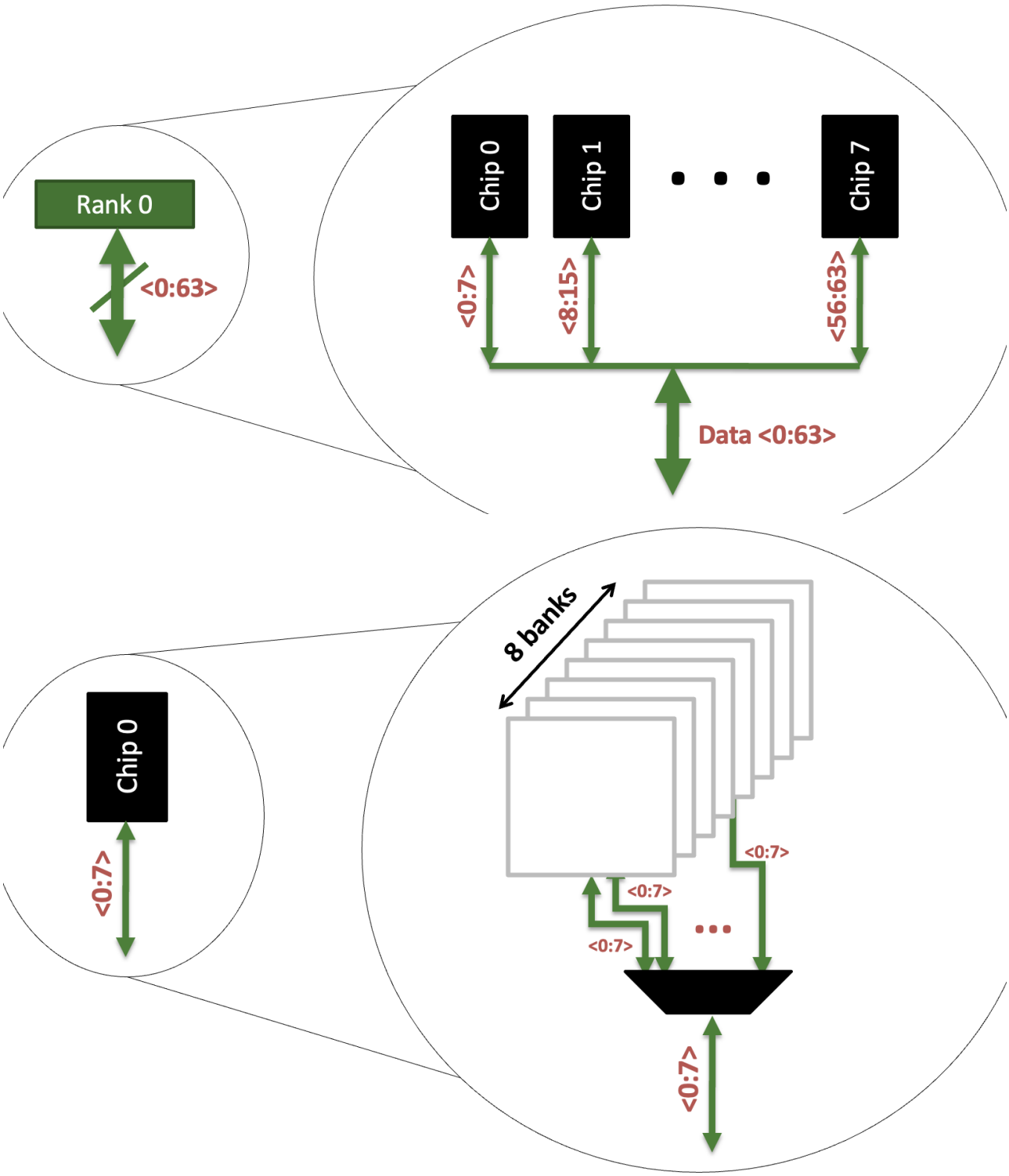
- DRAM 结构比
  SRAM（六个晶体管）更简单，两个元件（一个晶体管、一个电容），因此相对的容量就更大。 但是缺点是电容会有漏电，需要及时充电。
- Architecture of DRAM 由于DRAM比较大，因此要分层次进行管理。

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column




"Channel"
DIMM (Dual in-line memory module)
Processor
intel
Core™ i7
Memory channel
Memory channel


DIMM (Dual in-line memory module)
Side view
SIDE
4.00
Front of DIMM
Back of DIMM
SPD
Rank 0: collection of 8 chips
Rank 1



Burst:给定一个地址，给出的不是一个数据而是八个数据，再在里面选。

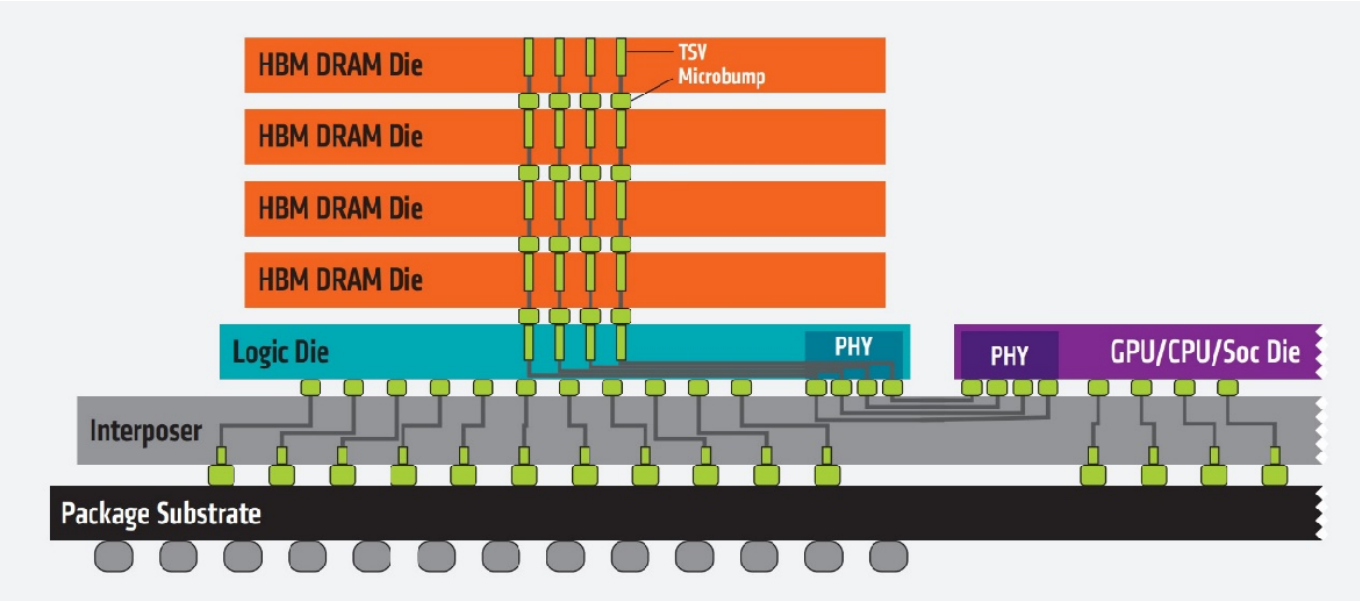
DDR：时钟上下延都能读写数据

HBM：高带宽内存

HBM（高带宽内存）堆栈：
这是一种与高端GPU（图形处理器）、AI专用芯片（ASIC）和现场可编程门阵列（FPGA）配合使用的高性能内存技术。

每颗HBM堆栈由以下部分组成：

- **4或8个DRAM内存芯片（dies）**：垂直堆叠，提供高密度数据存储。
- **1个逻辑控制芯片（logic die）**：负责管理内存访问和数据传输。



- **Advantage of HBM:**
  - High bandwidth: ~500GB/s per stack.
  - Low power consumption: due to running without termination.

- **Disadvantage of HBM:**
  - Less flexibility: fixed, in the same package with compute chip.
  - Low capacity: really close to compute chip.
  - High cost: strict condition.

- DDR 是“经济型大容量高速公路”，适合通用计算。HBM 是“超高速专用通道”，为AI/GPU等高带宽场景量身定制。

1. Page Hit（页命中）

- **定义**：当内存事务访问的行（row）已经在其所属的存储体（bank）中处于打开状态时，称为页命中。
- **操作**：此时不需要发送预充电（Precharge）和行激活（Activate）命令，可以直接进行列访问（column access）。
- **延迟**：由于跳过了预充电和激活步骤，访问延迟最小（即最快）。

2. Page Closed（页关闭）

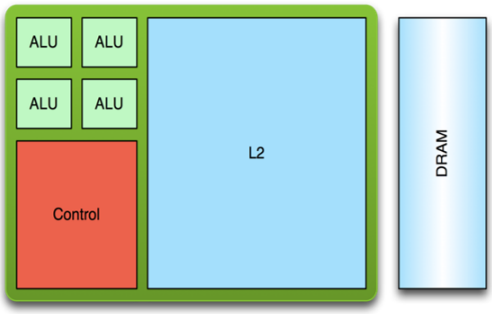
- **定义**：当内存事务访问的行所在的存储体（bank）处于关闭状态时，称为页关闭。
- **操作**：需要先发送行激活命令（Activate）打开目标行，然后才能进行列访问。
- **延迟**：比页命中多了一个激活步骤的延迟，但比页缺失（Page Miss）快，因为不需要预充电。

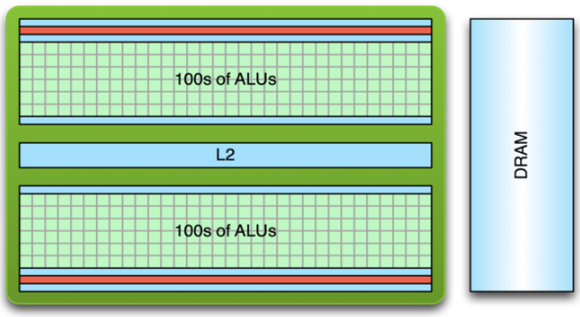
3. Page Miss（页缺失）

- **定义**：当内存事务访问的行与当前存储体（bank）中已打开的行不匹配时，称为页缺失。
- **操作**：需要先发送预充电命令（Precharge）关闭当前打开的行，然后发送激活命令（Activate）打开目标行，最后才能进行列访问。
- **延迟**：由于需要预充电和激活两个额外步骤，延迟最大（即最慢）。

6.GPU Architechture

CPU vs GPU





- **CPU:**
  - Few complex cores
  - Larger cache for low memory latency
  - Large and slow memory
- **GPU:**
  - Lots of simple cores
  - Small cache for low memory latency
  - Small and fast memory

Three steps:

- CPU-GPU data transfer (1)
- GPU kernel execution (2)
- GPU-CPU data transfer (3)

CPU-GPU Co-processing:

CPU: 串行部分 以及 简单的并行任务 GPU: 大规模并行任务

Programming Model vs Hardware Execution Model

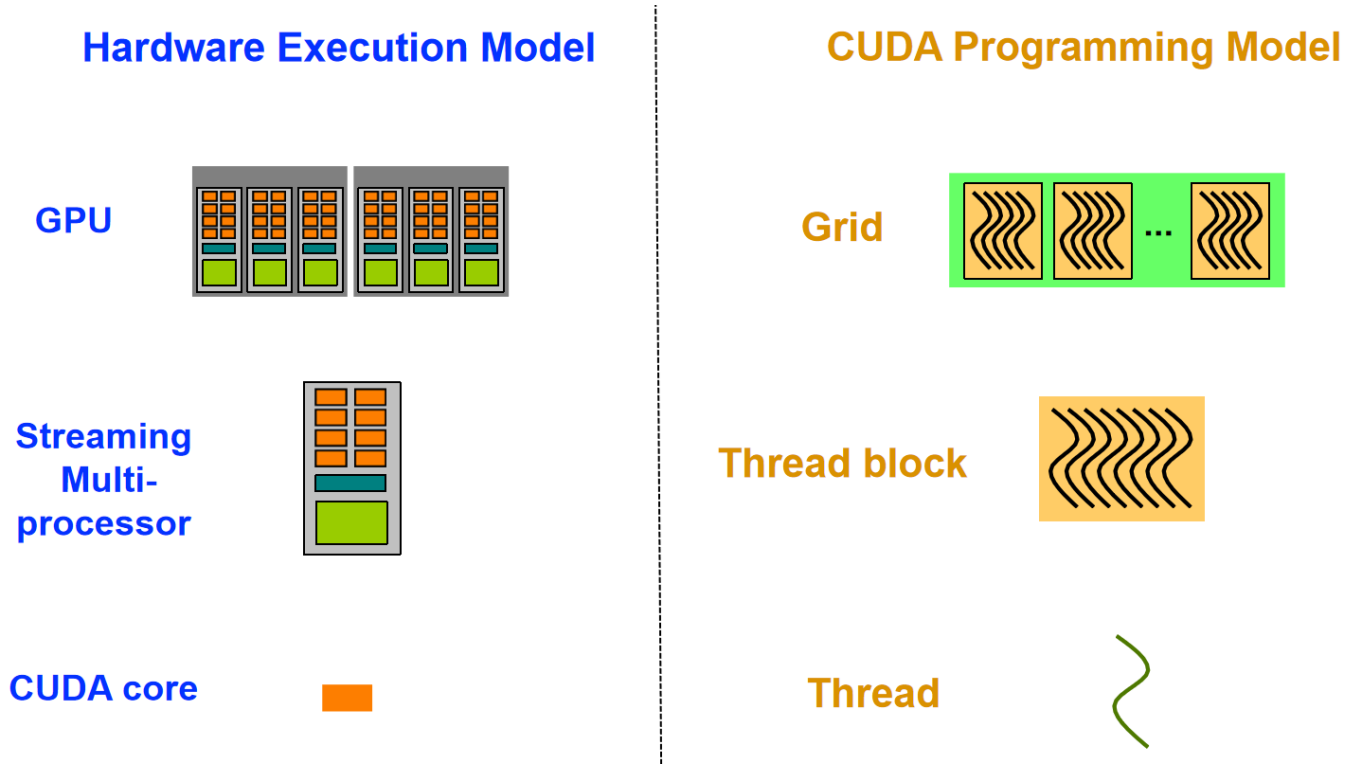
Programming Model： how the programmer expresses the code

- E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), …

Hardware Execution Model： how the hardware executes the code underneath

- E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, …

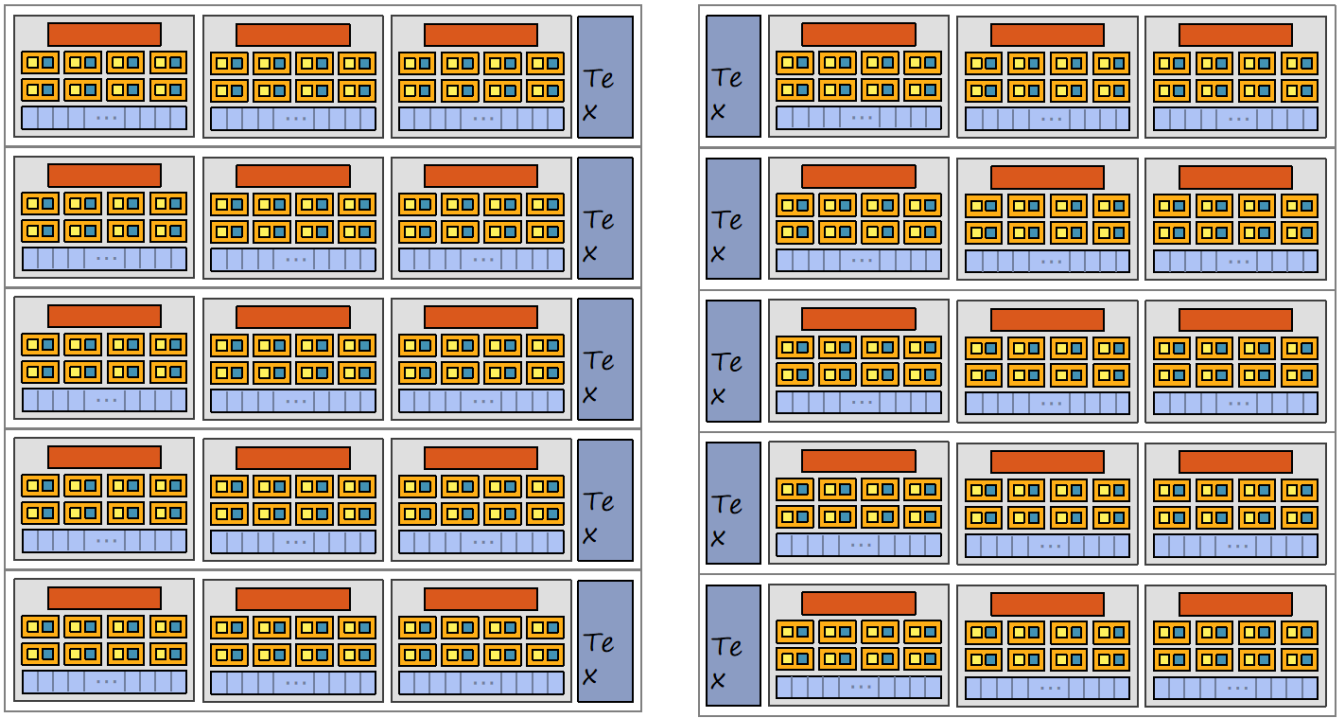
GPU架构



- 一般将SM作为“ core ”。

Hardware Execution Model

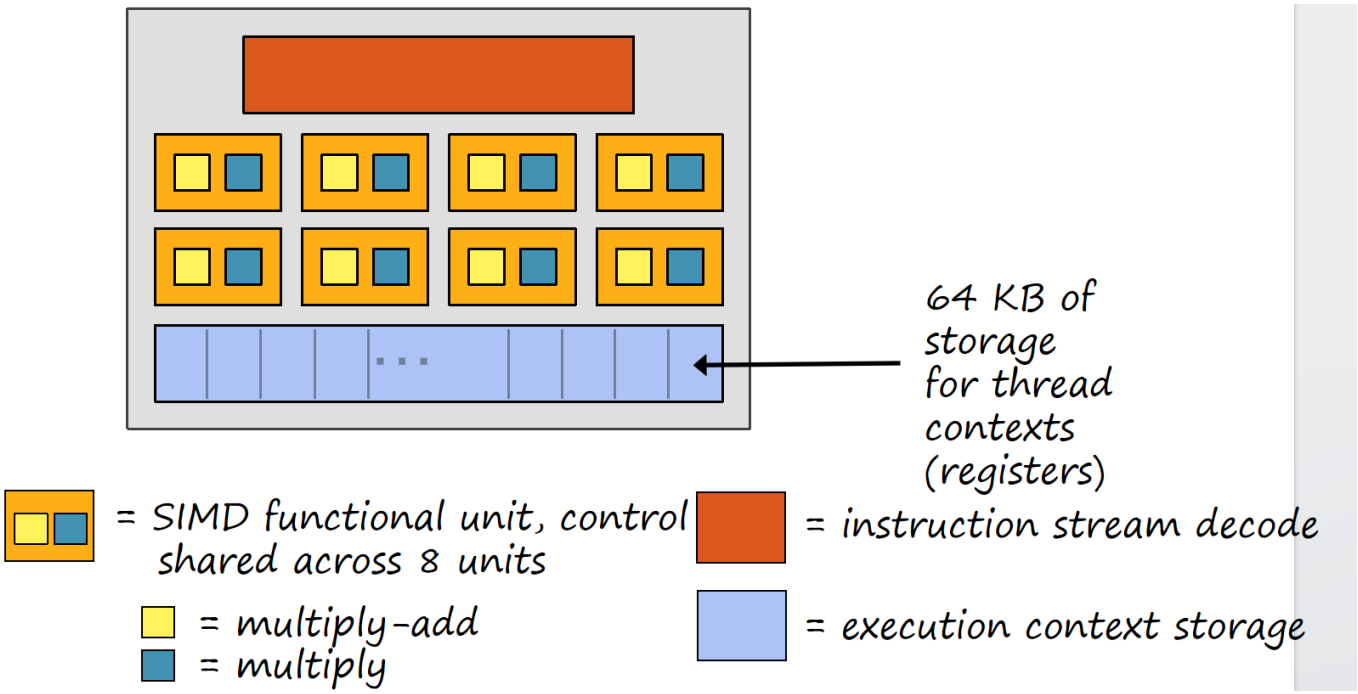
GPU示意图



- 一个GPU有30个core（也就是30个sm）。

SM(Streaming Multi-processor)示意图

硬件执行角度



- warp：一定数量的thread被集中在一起，共享instruction stream。

软件编程角度

- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

  - SIMT:同一个指令使用多个thread。

SIMT & Warp

- **SIMT**: **S**ingle **I**nstruction **M**ultiple **T**hread
  - More precisely, SIMD (Single Instruction Multiple Data)
  - Key Feature: 16 CUDA cores in a SM are executed in a lock step.

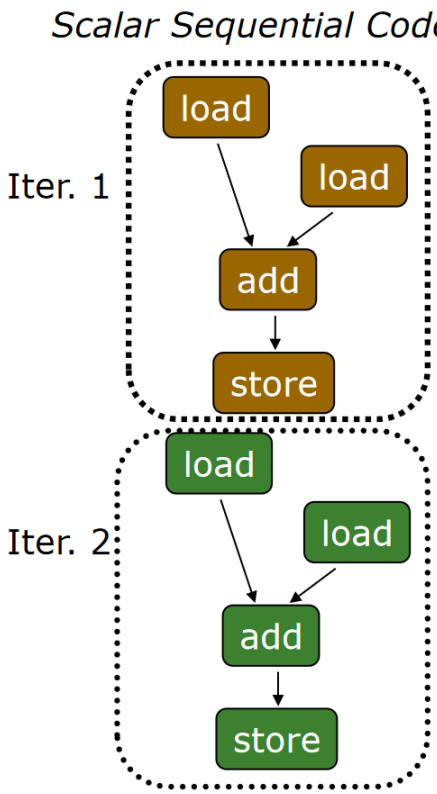
- **Warp:**
  - A warp, a basic execution unit, consists of 32 consecutive threads
  - A thread block is divided into warps for SIMT execution.

这一类GPU因此就有30K个threads （1 GPU = 30 CORES = 30*1024 THREADS）

Programming Model

Prog. Model 1: Sequential (SISD)

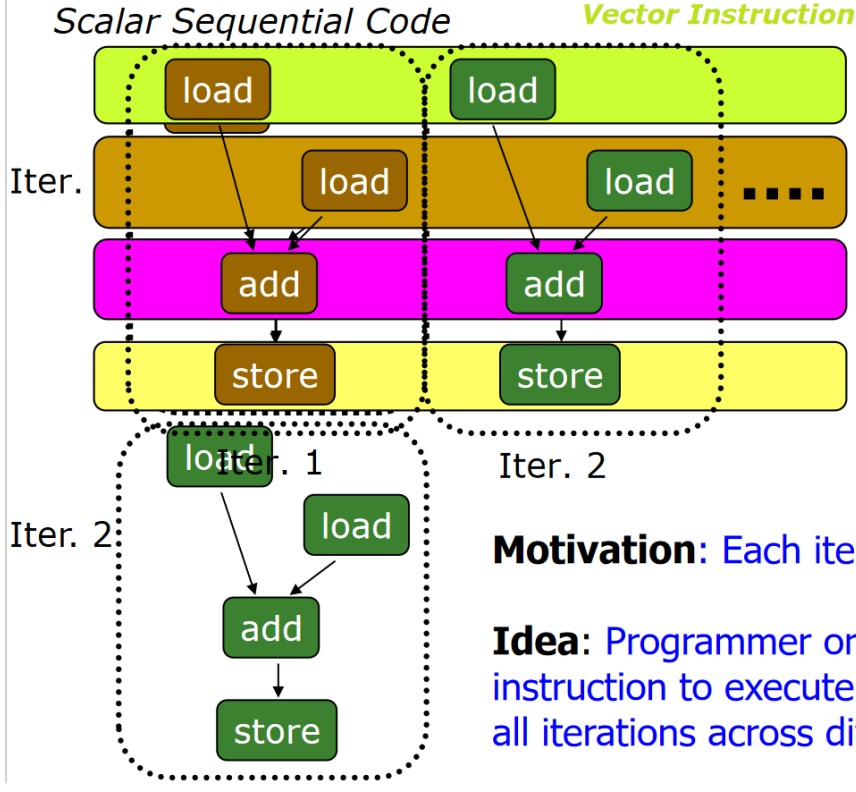
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



- Can be executed on thee processors:
- 1, Pipelined processor
- 2, Out-of-order execution processor
  - Independent instructions executed when ready
  - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
- 3, Superscalar or VLIW processor
  - Can fetch and execute multiple instructions per cycle

Prog. Model 2: Data Parallel (SIMD)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Vectorized Code

VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

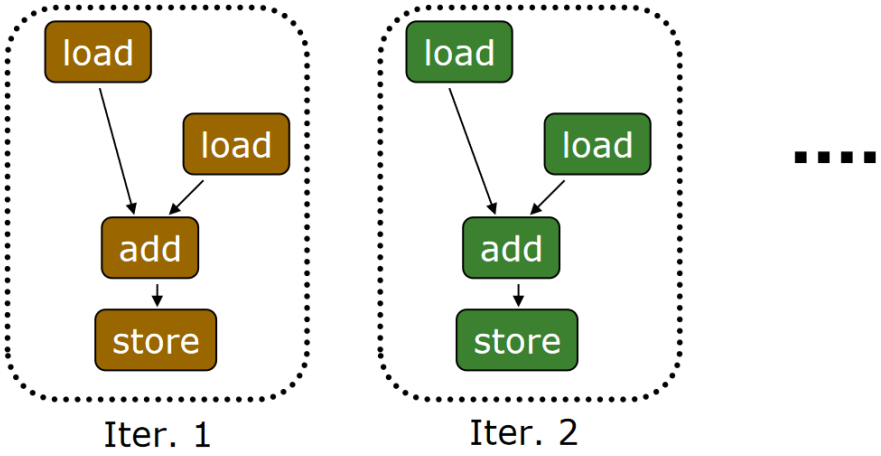
VST V3 → C

Motivation: Each iteration is independent

Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Realization: Each iteration is independent

This programming model (software) is called:

SPMD: Single Program Multiple Data

- 第三种：每层迭代都有一个单独的thread执行。
- 注意将第三种与SIMD区分。

SPMD

SPMD

- **SPMD**: Single procedure/program, multiple data
  - This is a **programming model** rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

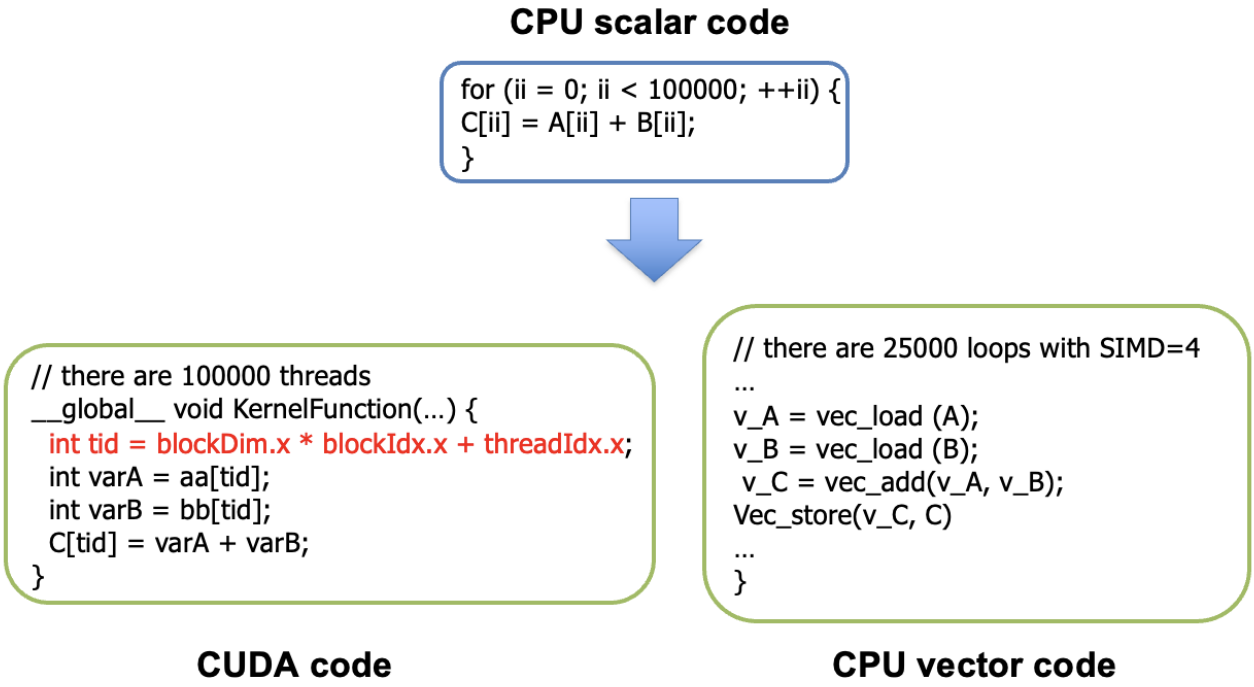
- **Key Idea of SPMD**: multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware

SPMD 和 SIMD 的区别

1. **并行粒度**：SIMD是细粒度(指令级)，SPMD是粗粒度(程序级)
2. **控制流**：SIMD必须同步执行，SPMD可以有不同的控制路径
3. **硬件支持**：SIMD通常由专用硬件实现，SPMD更抽象
4. **灵活性**：SPMD能处理更复杂、不规则的问题

SIMT 和 SIMD 的区别

Reall: SIMT Code vs. SIMD Code



Cuda：SPMD的一种

- The device (typically GPU) executes CUDA kernels
  - Grid
  - Thread Block
    - CUDA runtime schedules at granularity of thread block.
    - A thread block is a programming abstraction that represents a group of threads that can be executed in parallel.
    - Within a block, shared memory, and synchronization.
  - Thread
    - A thread corresponds to an iteration.

Cuda编程

- Function prototypes

```
float serialFunction(…);
__global__ void kernel(…);
```

- `main()`
  - 1) Allocate memory space on the device – `cudaMalloc(&d_in, bytes);`
  - 2) Transfer data from host to device – `cudaMemCpy(d_in, h_in, …);`
  - 3) Execution configuration setup: #blocks and #threads
  - 4) Kernel call – `kernel<<<execution configuration>>>(args…);`
  - 5) Transfer results from device to host – `cudaMemCpy(h_out, d_out, …);`

- Kernel – `__global__ void kernel(type args,…)`
  - Automatic variables transparently assigned to registers
  - Shared memory: `__shared__`
  - Intra-block synchronization: `__syncthreads();`

CUDA Programming Language

- Memory allocation

```
cudaMalloc((void**)&d_in, #bytes);
```

- Memory copy

```
cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
```

- Kernel launch

```
kernel<<< #blocks, #threads >>>(args);
```

- Memory deallocation

```
cudaFree(d_in);
```

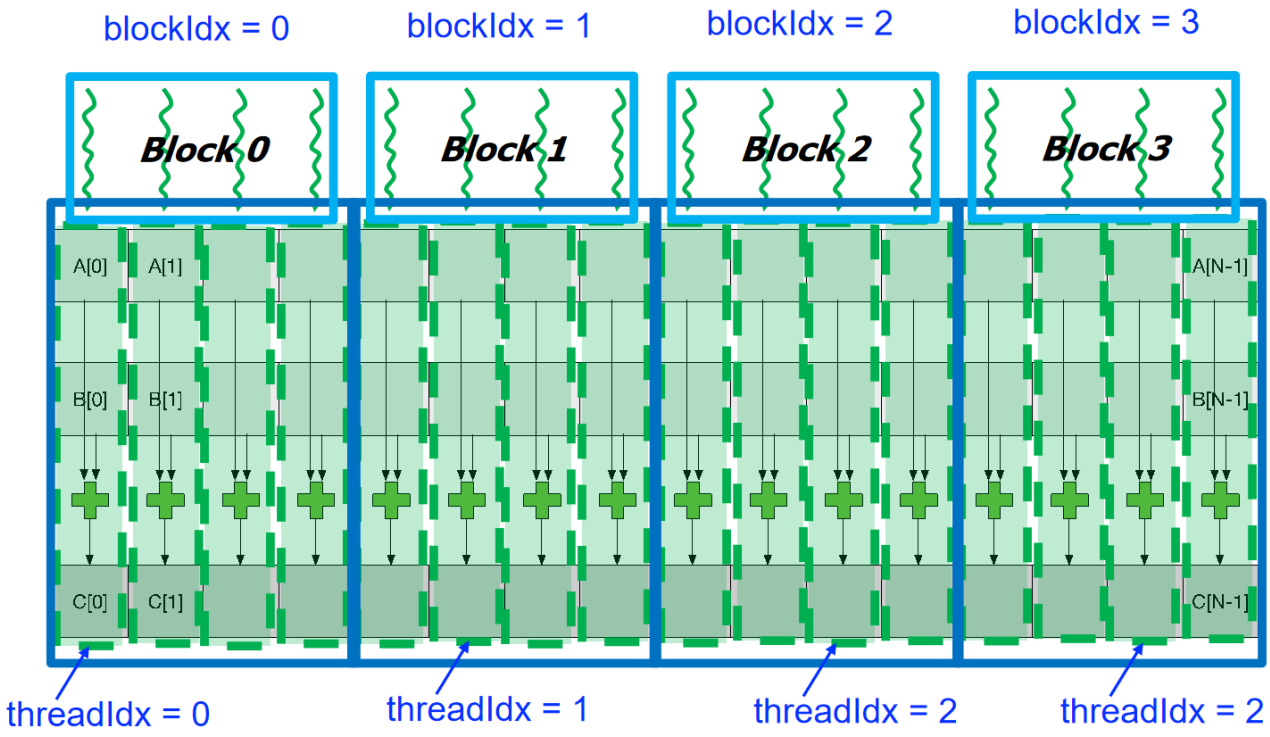
- Explicit synchronization

```
cudaDeviceSynchronize();
```

Example：向量加法

First GPU Example: Vector Addition (III)

- We group threads into blocks



Host Code Example: Vector Addition

```
void vecadd(float* A, float* B, float* C, int N) {

    //1, Allocate GPU memory
    float *A_d, *B_d, *C_d;
    cudaMalloc((void**) &A_d, N*sizeof(float));
    cudaMalloc((void**) &B_d, N*sizeof(float));
    cudaMalloc((void**) &C_d, N*sizeof(float));

    //2, Copy data to GPU memory
    cudaMemcpy(A_d, A, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, N*sizeof(float), cudaMemcpyHostToDevice);

    //3, Perform computation on GPU
    const unsigned int numThreadsPerBlock = 512;
    const unsigned int numBlocks = N/numThreadsPerBlock;
    vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);

    //4, Copy data from GPU memory
    cudaMemcpy(C, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);

    //5, Deallocate GPU memory
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

```
__global__ void vecadd_kernel(float* A, float* B, float* C, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

但上面的代码存在一个问题：如果输入的大小不是每个block的线程数量的整数倍？解决方案：使用block的数量始终+1，保证block充足：

- **Host code:**

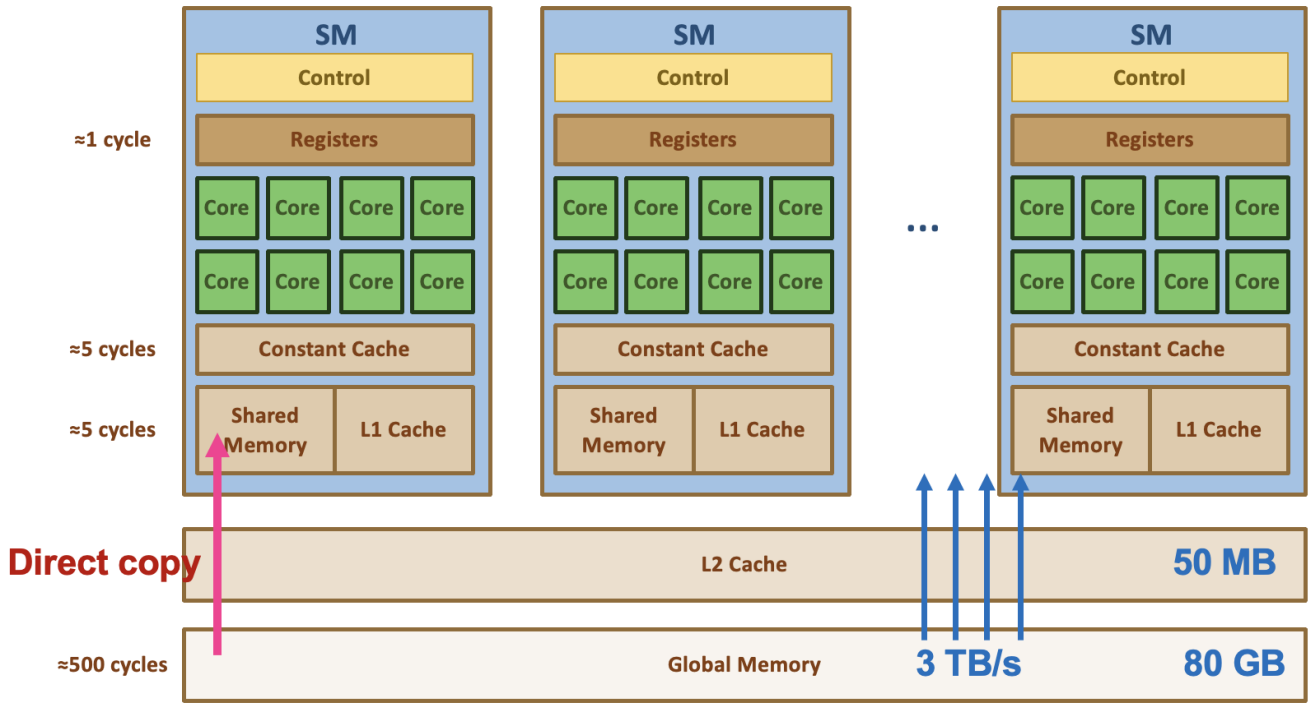
```
const unsigned int numBlocks = (N +numThreadsPerBlock - 1)/numThreadsPerBlock;
vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);
```

- **Kernel code:**

```
__global__ void vecadd_kernel(float* A, float* B, float* C, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if(i < N) {
        C[i] = A[i] + B[i]; }
}
```

7.GPU Optimization

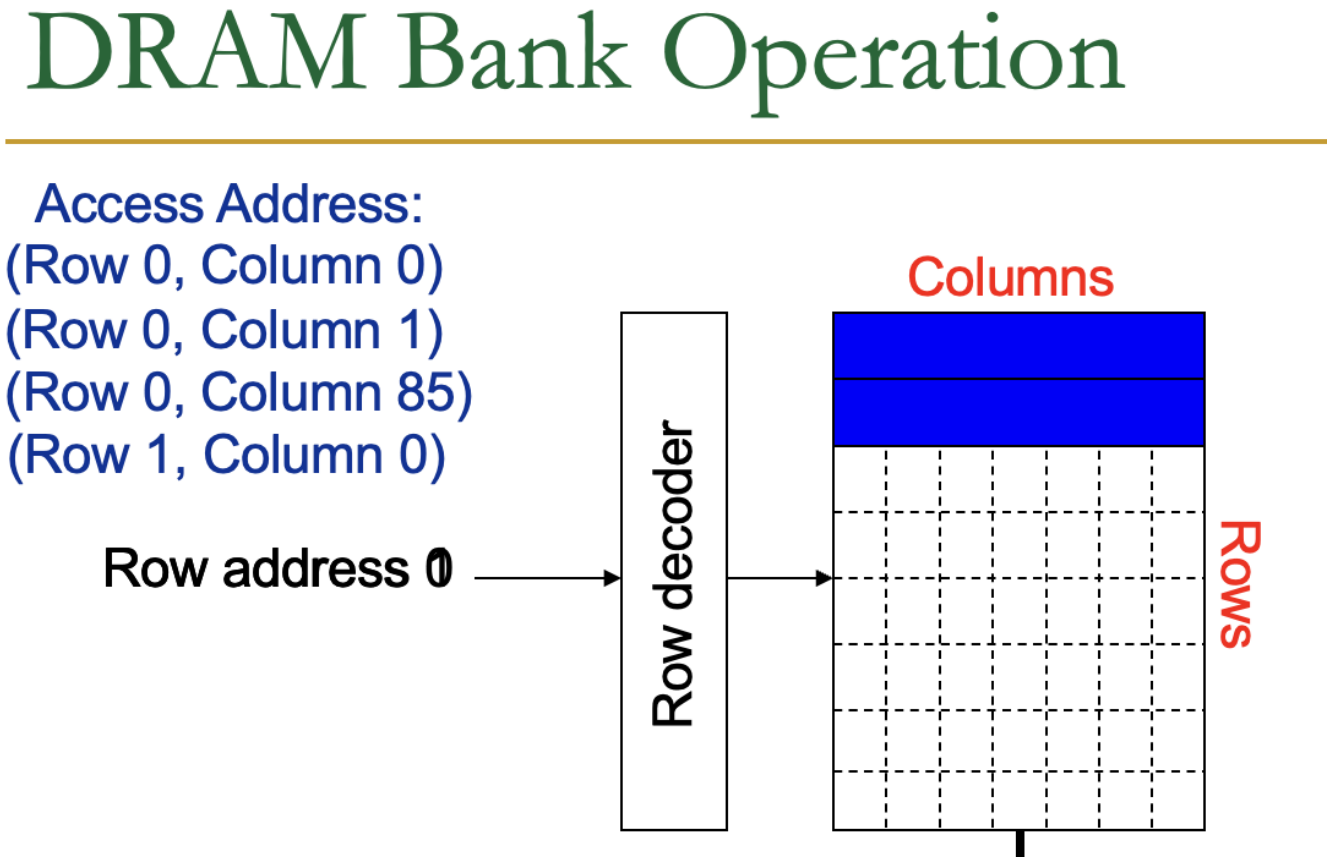
GPU Memory 示意图



不同变量的区别

Variable declaration		Memory	Scope	Lifetime
	`int LocalVar;`	register	thread	thread
	`int localArr[N];`	global	thread	thread
`__device__ __shared__`	`int SharedVar;`	shared	block	block
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

DRAM访问



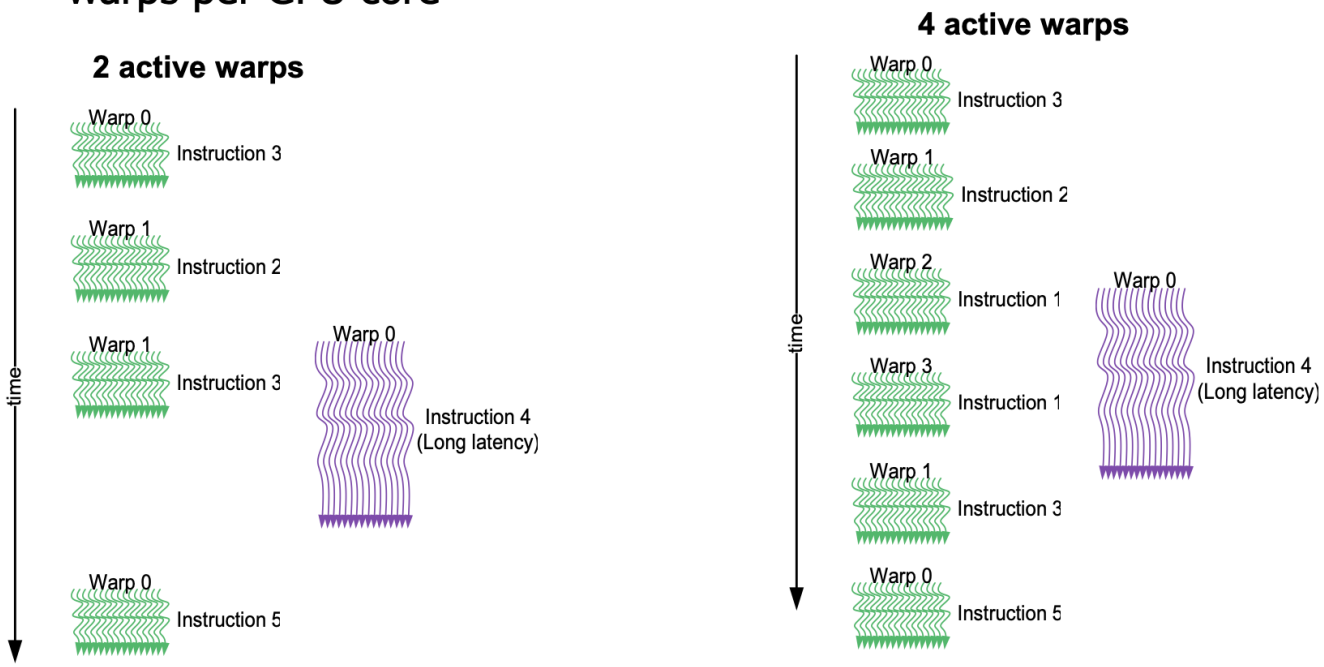
注意：每次读出来的都是一行（row），整行的数据都被存在row buffer中。 因此当连续访问row0的不同column时，延迟很低，因为可以直接在buffer中找到； 突然访问row1，则需要去内存中取出具体的数值，因此会多花约10倍的时间。

Optimization of Memory System

Multi-threading

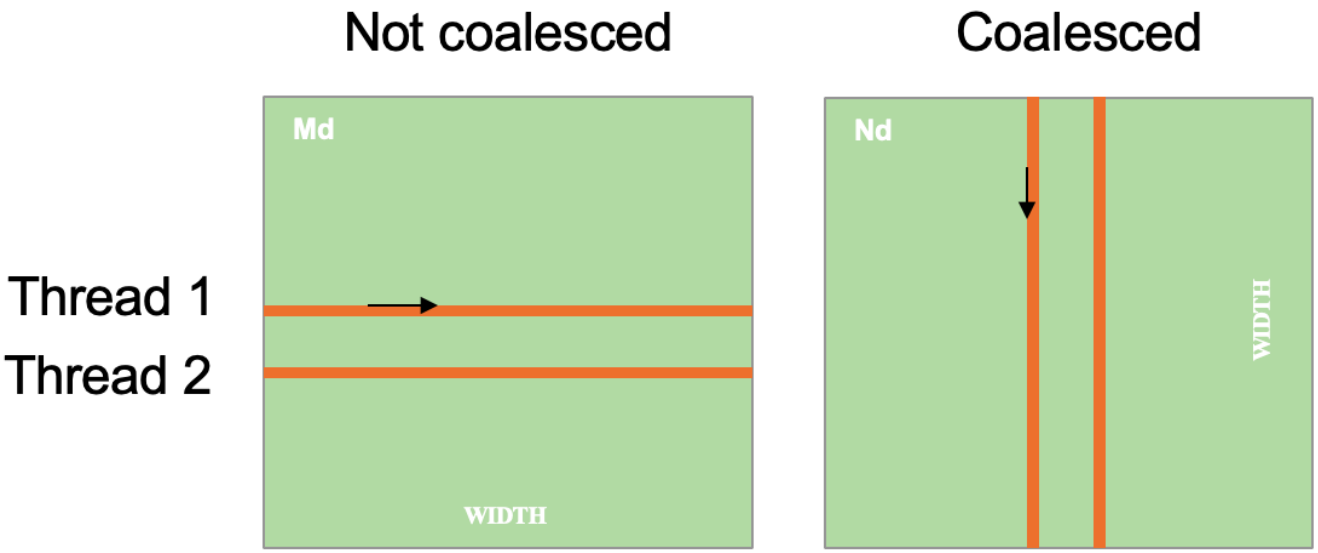
本质：统筹规划！ 也就是说在某个warp进行长延时的操作（比如访问内存）的时候，其他warp执行与长延时操作无冲突的命令，提高GPU利用率。FGMT:Fine-grained multithreading

- FGMT can hide long latency operations (e.g., memory accesses)
- Occupancy: ratio of active warps to the maximum number of warps per GPU core

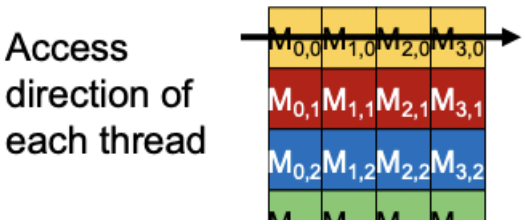


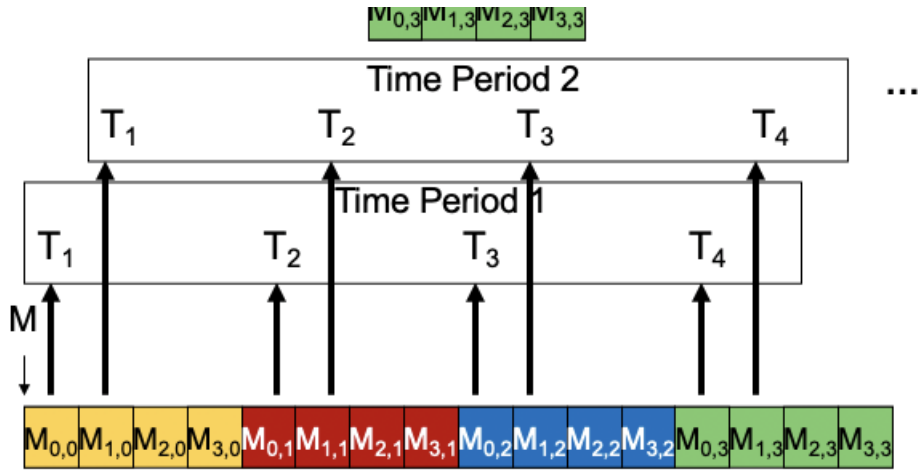
Memory Coalescing(联合)

本质：同一个warp内的thread访问的memory的地址是连续的！（在memory的同一个busrt中）
也就意味着：Only one DRAM transaction is needed.



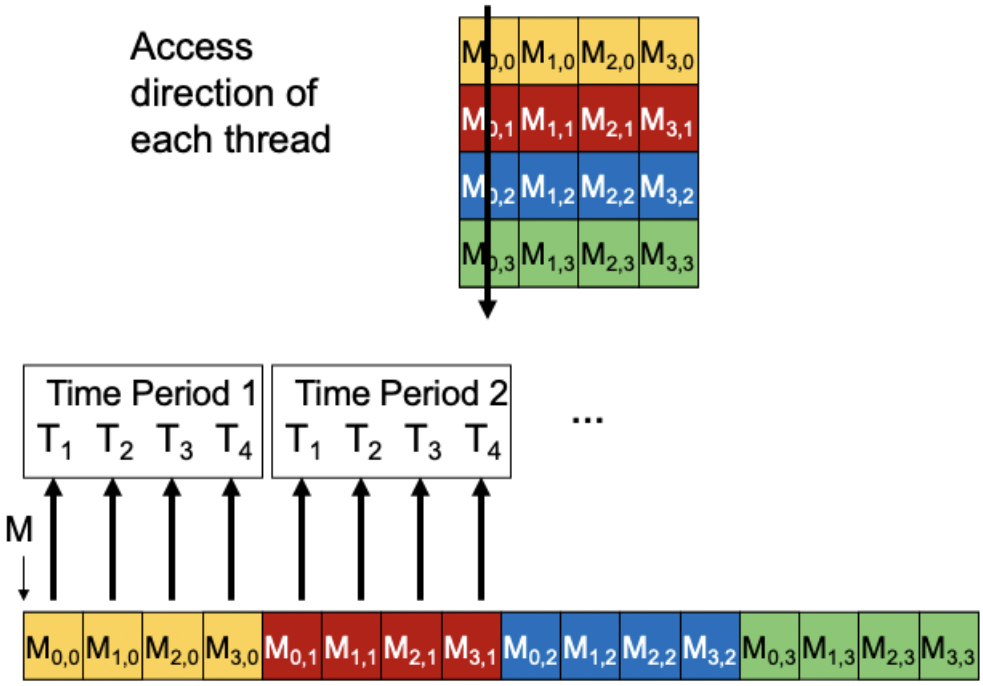
Uncoalesced Memory Accesses





Slide credit: Hwu & Kirk 37

Coalesced Memory Accesses



Slide credit: Hwu & Kirk 38

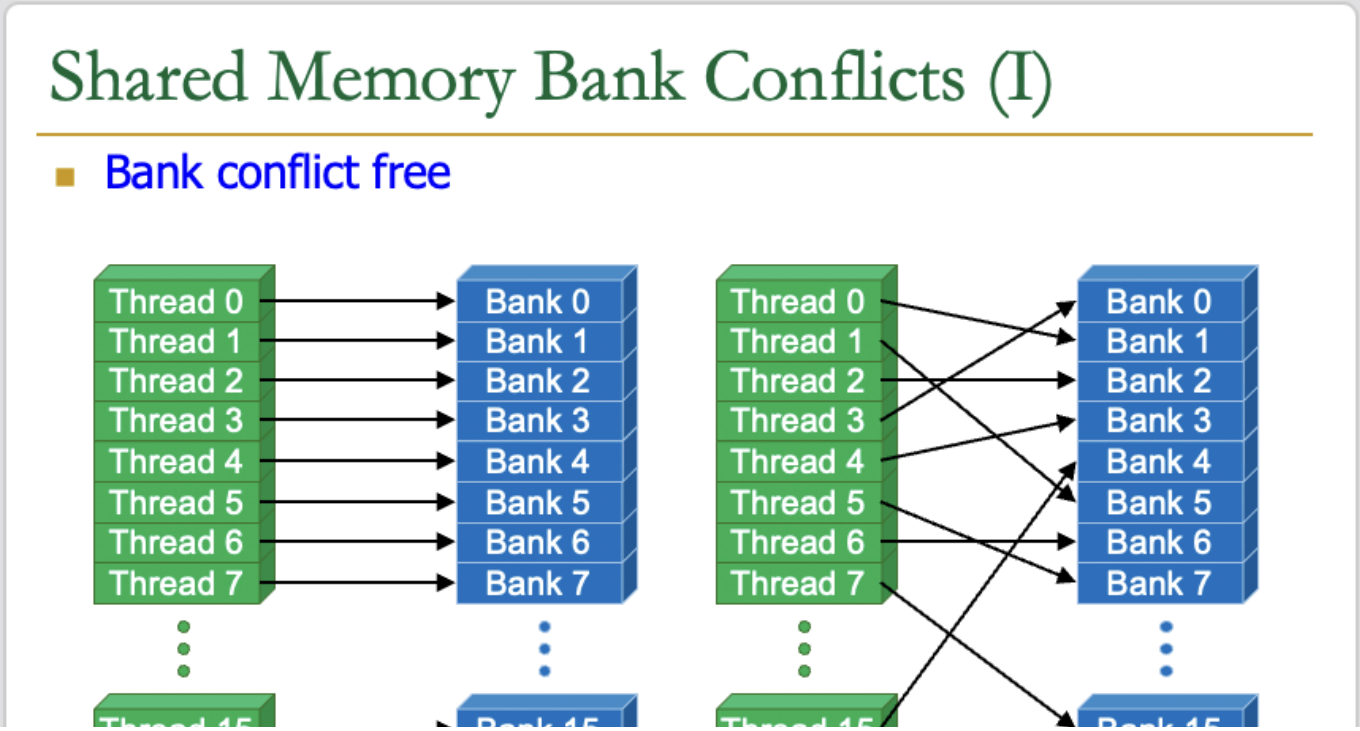
Shared Memory

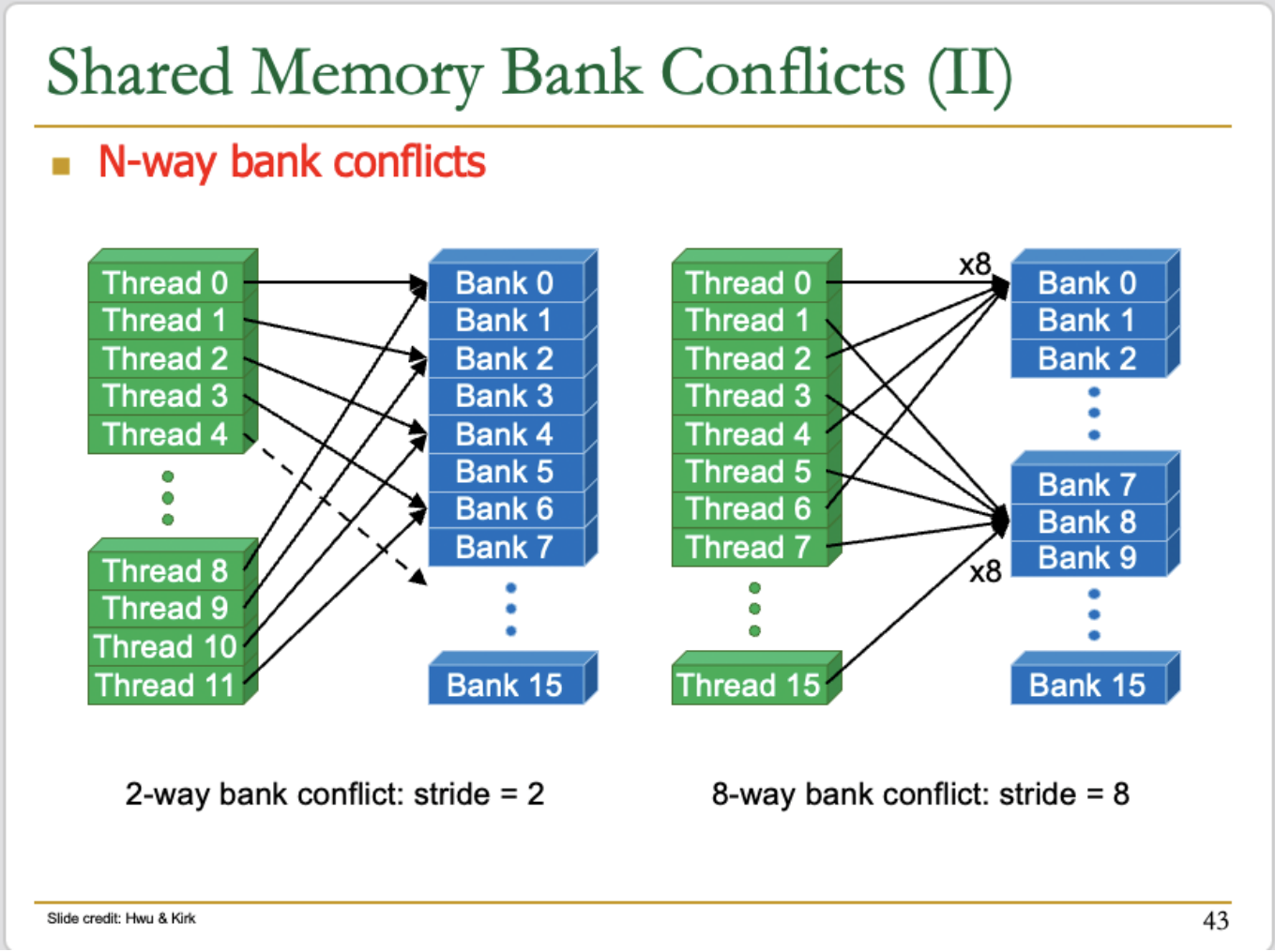
本质：SRAM快，DRAM慢，因此尽可能多访问前者。共享内存共享的就是SRAM。

1. **"Shared memory is an interleaved (banked) memory"**
    - **共享内存是一种交错式（分 Bank）存储结构。**

- 即共享内存被划分为多个独立的存储单元（Bank），数据按一定规则分布在这些 Bank 上。

2. **"Each bank can service one address per cycle"**
   - **每个 Bank 每个时钟周期可以响应一个地址的访问。**
   - 即同一周期内，一个 Bank 只能处理一个内存请求，如果多个请求访问同一个 Bank，就会发生冲突。
3. **"Typically, 32 banks in NVIDIA GPUs"**
   - **在 NVIDIA GPU 中，通常有 32 个 Bank。**
   - 这是硬件设计上的常见配置（如 CUDA 架构中的共享内存）。
4. **"Successive 32-bit words are assigned to successive banks"**
   - **连续的 32 位字（4 字节）会被分配到连续的 Bank 中。**
   - 例如，地址 0 → Bank 0，地址 1 → Bank 1，…，地址 31 → Bank 31，地址 32 → Bank 0（循环分配）。
5. **"Bank = Address % 32"**
   - **Bank 编号 = 内存地址 % 32（取模运算）。**
   - 即 Bank 的分配是通过地址对 32 取模决定的，确保均匀分布。
6. **"Bank conflicts are only possible within a warp"**
   - **Bank 冲突只可能发生在一个线程束（Warp）内部。**
   - 因为 Warp（通常 32 个线程）是 GPU 调度的基本单位，同一 Warp 中的线程如果访问同一个 Bank 的不同地址，会导致串行化（冲突）。
7. **"No bank conflicts between different warps"**
   - **不同 Warp 之间不会发生 Bank 冲突。**
   - 因为 GPU 的调度机制会保证不同 Warp 的请求在时间上错开，即使访问相同 Bank 也不会冲突。





Bank Conflict Free： 不同thread访问不同bank -> 一个周期就完成

N-Way Bank Conflict： N个周期完成

Tiling or Blocking

将原本在大型数组上连续执行的循环计算拆分成多个小块（Chunk），使得每个小块的数据能够完全放入芯片上的高速存储（如片上 RAM、缓存或 Scratchpad Memory），从而减少对慢速主存的访问，提升计算效率。

1. **"Divide loops operating on arrays into computation chunks"**
    - **将针对数组的循环计算拆分成多个计算块（Chunk）。**
    - 例如，一个大型矩阵乘法可以被分解为多个小矩阵块的乘法。
2. **"so that each chunk can hold its data in the on-chip RAM"**

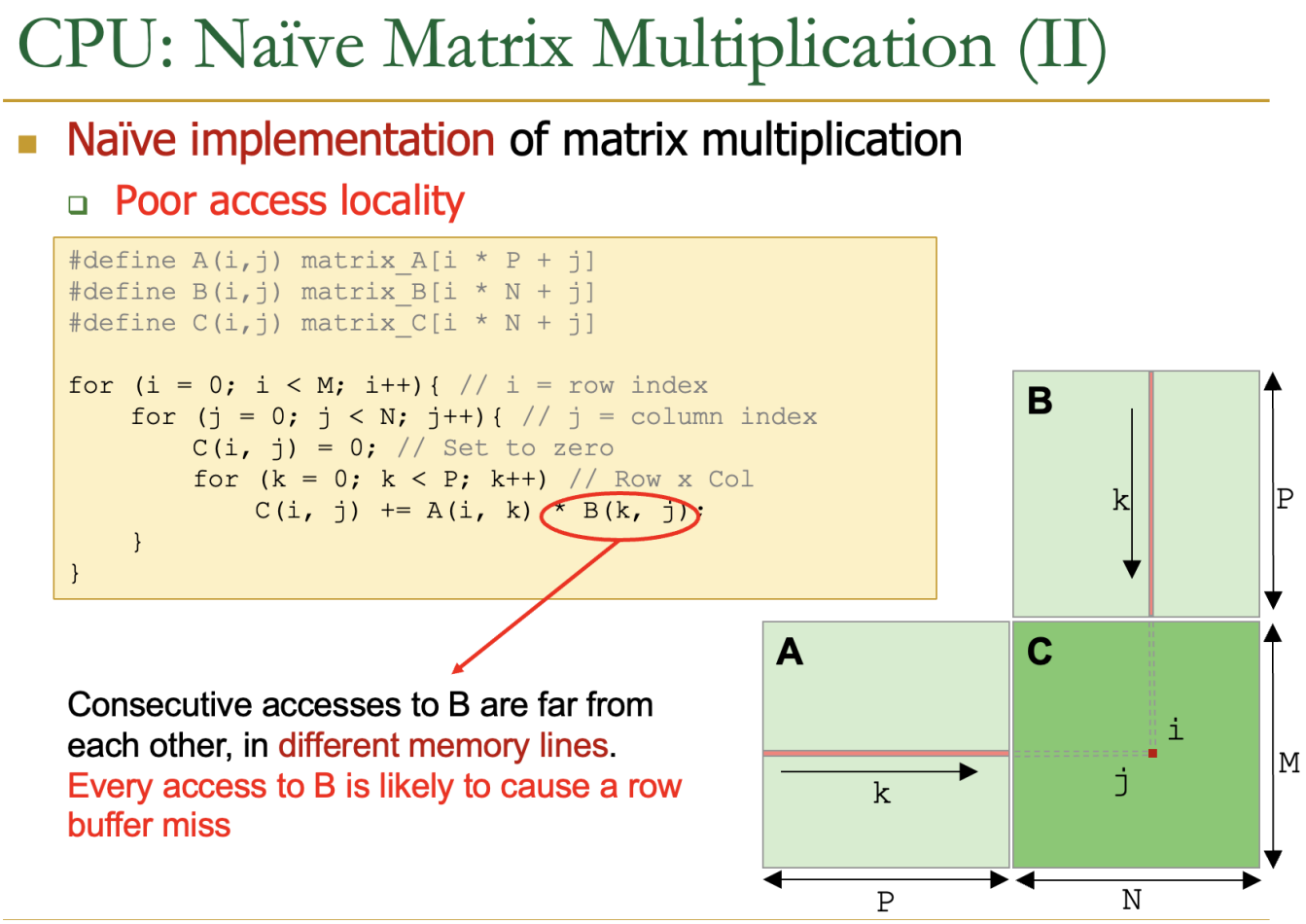
- **确保每个计算块的数据能够完全存储在片上 RAM（或其他高速存储，如 Scratchpad Memory）中。**
- 这样计算时无需频繁访问较慢的全局内存（DRAM），减少延迟。

3. **"Avoids on-chip RAM conflicts between different chunks of computation"**
   - **避免不同计算块之间对片上 RAM 的访问冲突。**
   - 由于每个块的数据独立存储在高速存储中，不同块的计算不会互相争抢带宽。
4. **"Essentially: Divide the working set so that each piece fits in the on-chip RAMs"**
   - **本质：将工作集（Working Set）划分成若干小块，确保每块数据能完全放入片上 RAM。**
   - 这样可以利用局部性原理（Locality），提高数据复用率，减少内存瓶颈。

Tiling 的一个经典例子：矩阵乘法

算法比较

传统做法



经典做法下：每次访问B的数据，locality都非常差！

tiling做法

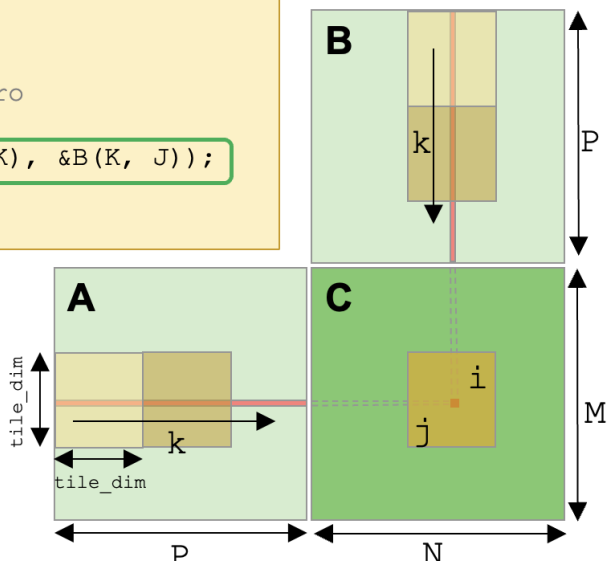
CPU: Tiled Matrix Multiplication (II)

- Tiled implementation operates on submatrices (tiles or blocks) that fit fast RAMs (cache, scratchpad, RF)

```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (I = 0; I < M; I += tile_dim){
    for (J = 0; J < N; J += tile_dim){
        Set_to_zero(&C(I, J)); // Set to zero
        for (K = 0; K < P; K += tile dim)
            Multiply_tiles(&C(I, J), &A(I, K), &B(K, J));
    }
}
```

Multiply small submatrices (tiles or blocks) of size `tile_dim x tile_dim`



改成一块一块地操作。

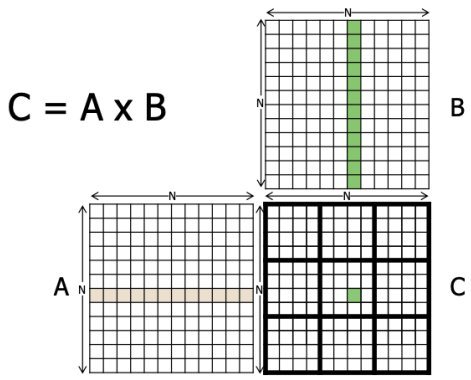
普通的N*N矩阵乘法的代码

```
__global__ void mm_kernel(float* A, float* B, float* C, unsigned int N) {

    unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for(unsigned int i = 0; i < N; ++i) {
        sum += A[row*N + i]*B[i*N + col];
    }
    C[row*N + col] = sum;

}
```



tiling具体实现

GPU: Tiled Matrix-Matrix Multiplication (V)

```
__shared__ float A_s[TILE_DIM][TILE_DIM];
__shared__ float B_s[TILE_DIM][TILE_DIM];

unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;

float sum = 0.0f;

for(unsigned int tile = 0; tile < N/TILE_DIM; ++tile) {

    // Load tile to shared memory
    A_s[threadIdx.y][threadIdx.x] = A[row*N + tile*TILE_DIM + threadIdx.x];
    B_s[threadIdx.y][threadIdx.x] = B[(tile*TILE_DIM + threadIdx.y)*N + col];
    __syncthreads();

    // Compute with tile
    for(unsigned int i = 0; i < TILE_DIM; ++i) {
        sum += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];
    }
    __syncthreads();

}

C[row*N + col] = sum;
```

Declare arrays in shared memory

Threads wait for each other to finish loading before computing

Threads wait for each other to finish computing before loading

变量	计算方式	作用
`row`	`blockIdx.y * blockDim.y + threadIdx.y`	当前线程负责的 `C` 的行
`col`	`blockIdx.x * blockDim.x + threadIdx.x`	当前线程负责的 `C` 的列
`A` 的索引	`row * N + tile * TILE_DIM + threadIdx.x`	加载 `A` 的分块到 `A_s`
`B` 的索引	`(tile * TILE_DIM + threadIdx.y) * N + col`	加载 `B` 的分块到 `B_s`
`A_s` 访问	`A_s[threadIdx.y][i]`	计算时读取 `A_s` 的行
`B_s` 访问	`B_s[i][threadIdx.x]`	计算时读取 `B_s` 的列

Synchronization Function

- `void __syncthreads();`
  - Synchronizes all threads in a block

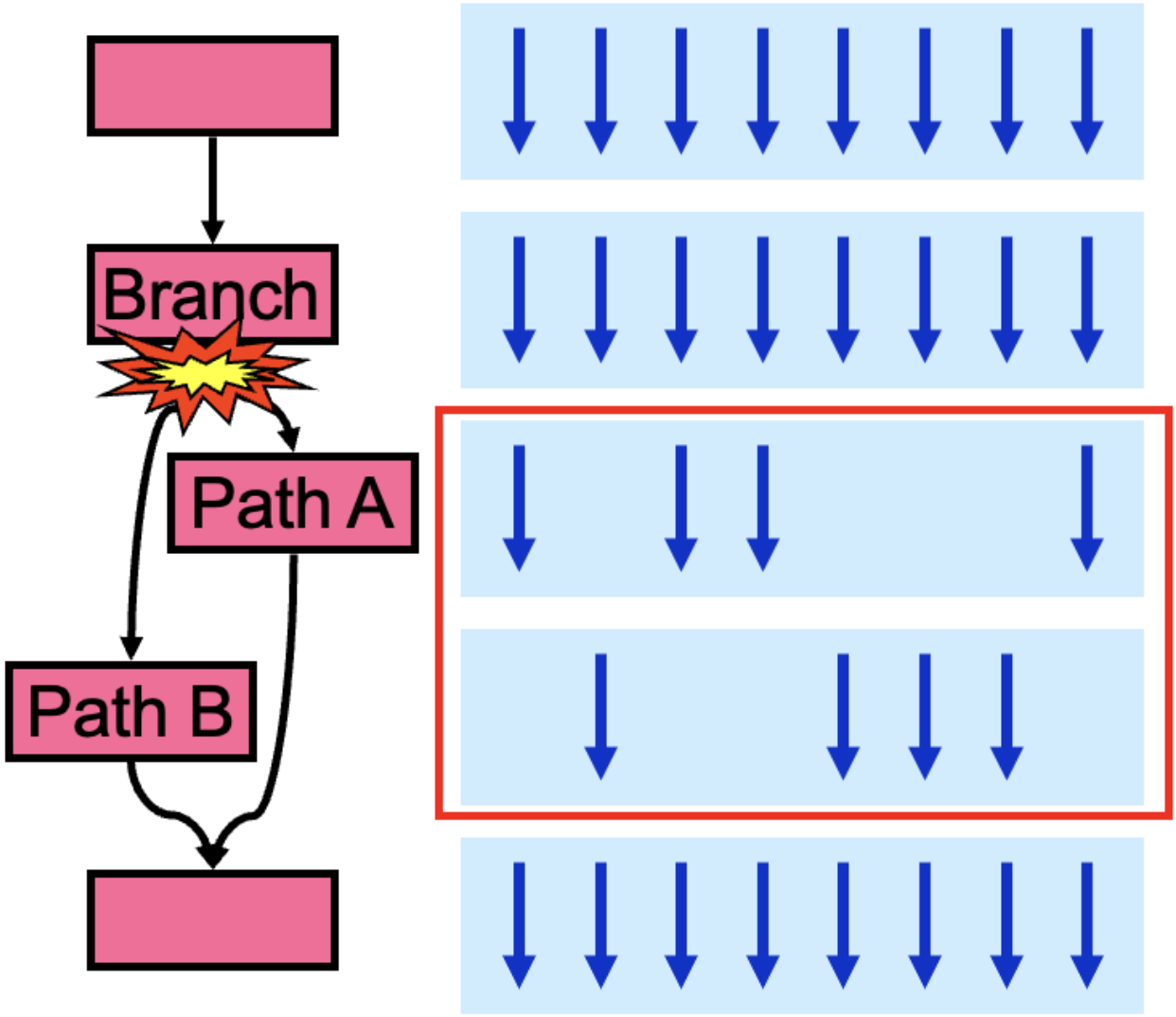
- Once all threads in a block have reached this point, execution resumes normally

- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory

SIMT Efficieny

Divergency(分支)

不同的thread如果执行的代码不同，怎么解决？（比如出现if else）
Branch divergence occurs when threads inside warps branch to different execution paths
解决办法：出现分支的时候选择部分thread进行分叉。

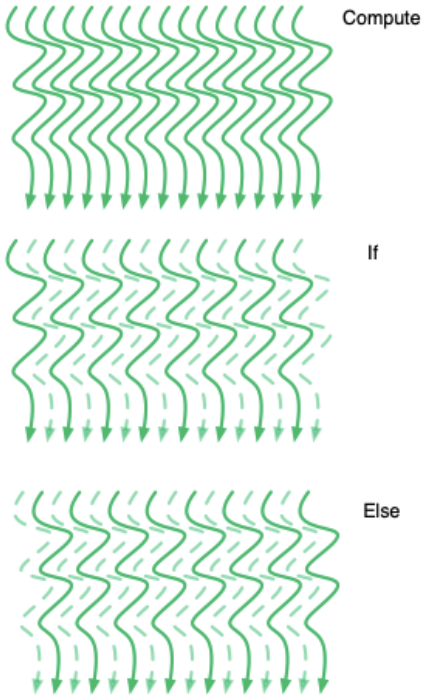


但是会损失一定的性能，因此在写代码的时候要注意避免，下面这张图就显示了不同的代码有不同的性能（前者分叉，后者不分叉）：

SIMT Utilization

- Intra-warp divergence

```
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
  Do_this(threadIdx.x);
}
else{
  Do_that(threadIdx.x);
}
```

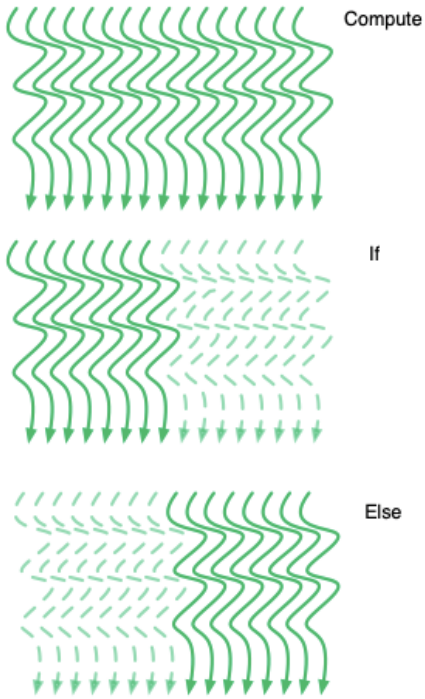


67

Increasing SIMT Utilization

- Divergence-free execution

```
Compute(threadIdx.x);
if (threadIdx.x < 32){
  Do_this(threadIdx.x * 2);
}
else{
  Do_that((threadIdx.x%32)*2+1);
}
```



举例：向量减法

Atomic

Atomic operations serialize the execution if there are atomic conflicts. 如果存在原子冲突，则原子作将序列化执行，影响效率。

CPU-GPU Transfer

如果本身任务并不大，那么数据传输的时间本身就会使GPU的使用不划算。

Key Messages:

- Programming model is the key success of Nvidia, rather than the GPU itself.
- GPU has an order of magnitude higher memory bandwidth and compute power than CPU.
- Offloading a task to GPU pays off only when the task has enough compute intensity.
- AI task needs compute-intensive accelerators, e.g., GPU and AI processor.

8.Memory Hierarchy and Caches

Memory Hierachy

Ideal Memory

- zero access time (latency)
- infinite capacity
- infinite bandwidth
- zero cost 所以理想内存很难实现的原因是：几个性质之间是互相矛盾的（单层内存无法同时保证 大+快） -> 多级内存：cache满足fast， main memory 满足large。

Cache

cache的作用： 利用locality实现既fast又large的内存访问

把最近访问的东西用cache存起来。

Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i, The **perceived access time** T_i is longer than t_i
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate hr_i) you "hit" and access time is t_i
  - a chance (miss-rate mr_i) you "miss" and access time $t_i + T_{i+1}$
  - $hr_i + mr_i = 1$
- Thus

$$T_i = hr_i \cdot t_i + mr_i \cdot (t_i + T_{i+1})$$
$$T_i = t_i + mr_i \cdot T_{i+1}$$

因此要优化两样内容：

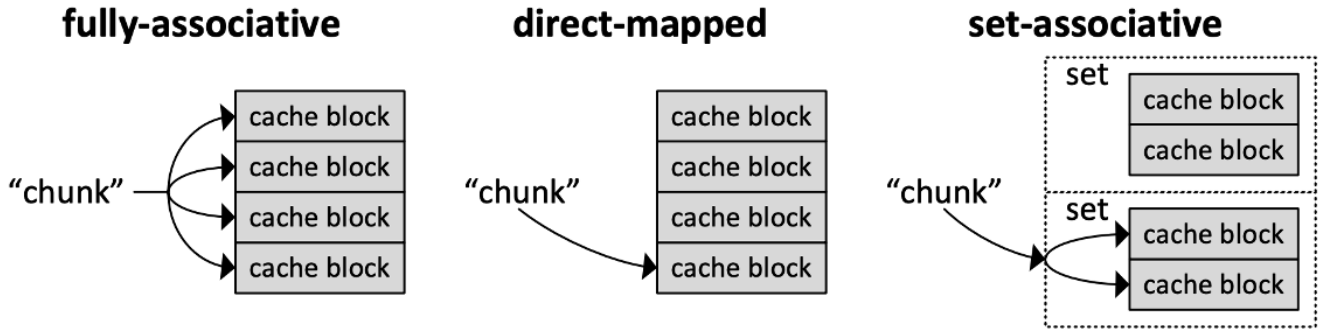
- 1.降低miss rate
- 2.降低Ti

  - $T_i \approx t_i$ is desirable, when we
    - Keep mr_i low
      - increasing capacity C_i lowers mr_i, but beware of increasing t_i
      - lower mr_i by smarter cache management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)
    - Keep T_{i+1} low
      - faster lower hierarchies, but beware of increasing cost
      - introduce intermediate hierarchies as a compromise

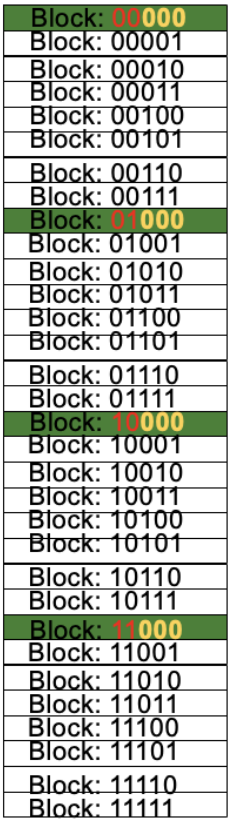
cache的3种类型

Three Cache Organization Methods

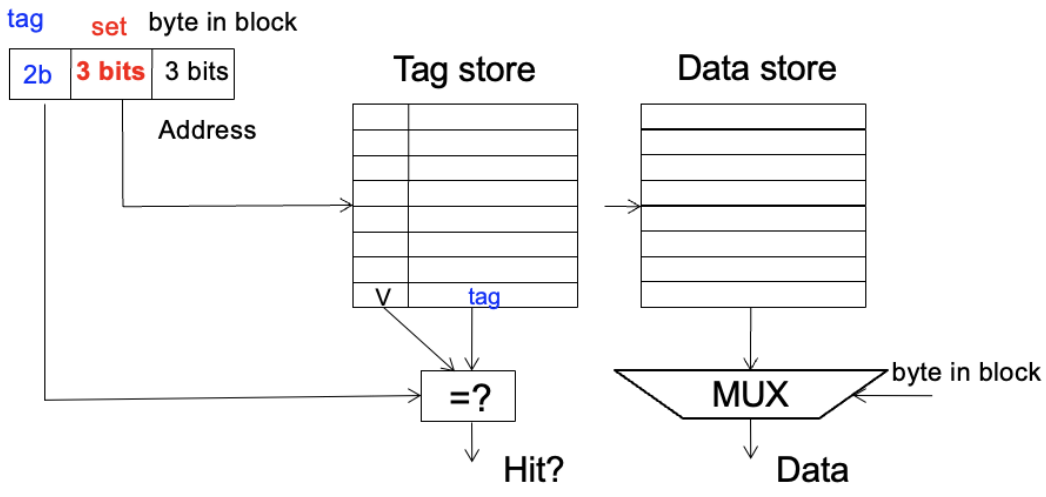
- Direct-mapped:
  - A chunk can go to **only one** cache block in the cache. (Another extreme)
- Fully-associative:
  - A chunk can go to **any** cache block in the cache. (One extreme)
- Set-associative:
  - A chunk can go to **N** cache blocks in the **N**-way set-associative cache. (Best choice)



Direct-Mapped Cache: Placement and Access



- Direct-mapped (A block can go to only one location)
  - Assume memory: 256 bytes, 8-byte blocks → 32 blocks
  - Assume cache: 64 bytes, 8 blocks
  - Blocks with same index contend for the same cache location
    - Cause conflict misses when accessing blocks in green consecutively

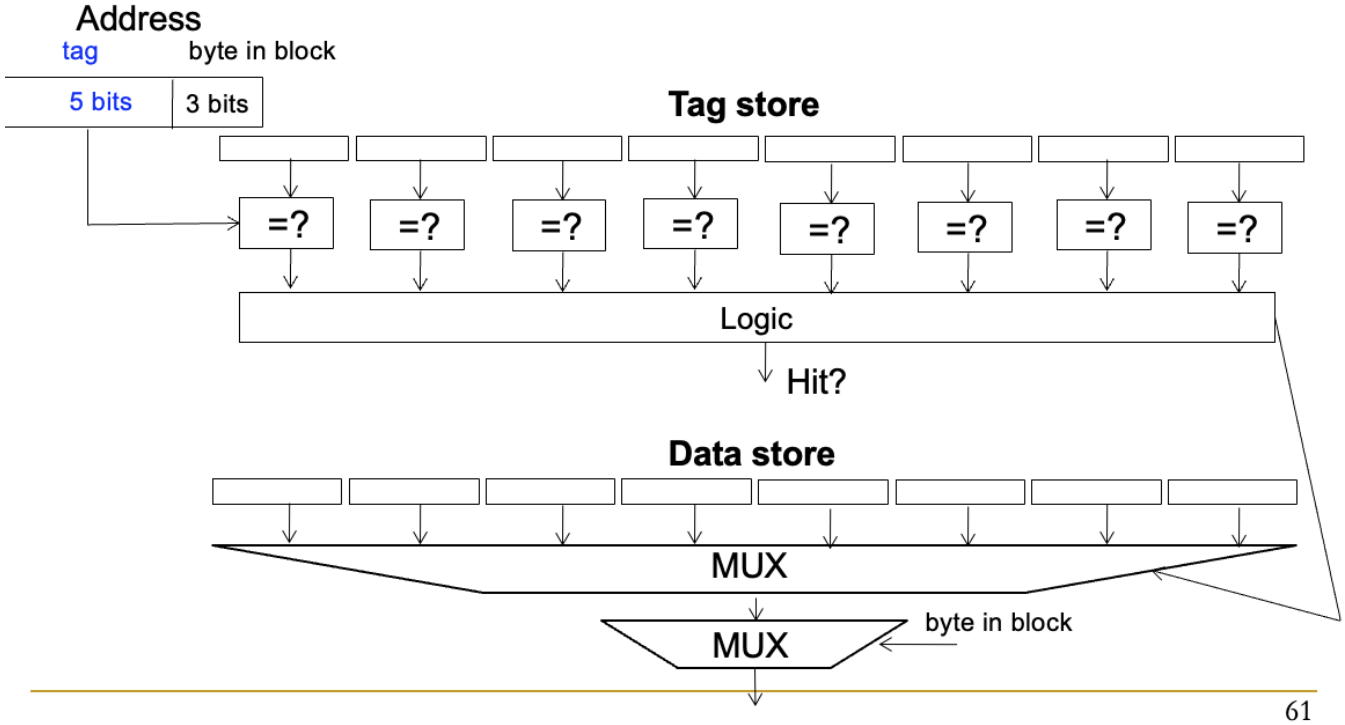


Advantage and Issue of Direct-Mapped Caches

- **Direct-mapped cache**:
  - Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time.
    - One index → one entry
- Main advantage of direct-mapped cache:
  - Easy to implement
- Main issue of direct-mapped cache:
  - Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
    - Assume addresses A and B have the same index bits but different tag bits
    - A, B, A, B, A, B, A, B, … → conflict in the cache index
    - All accesses are conflict misses

Full Associativity

- **Fully-associative cache**
  - A block can be placed in any cache location



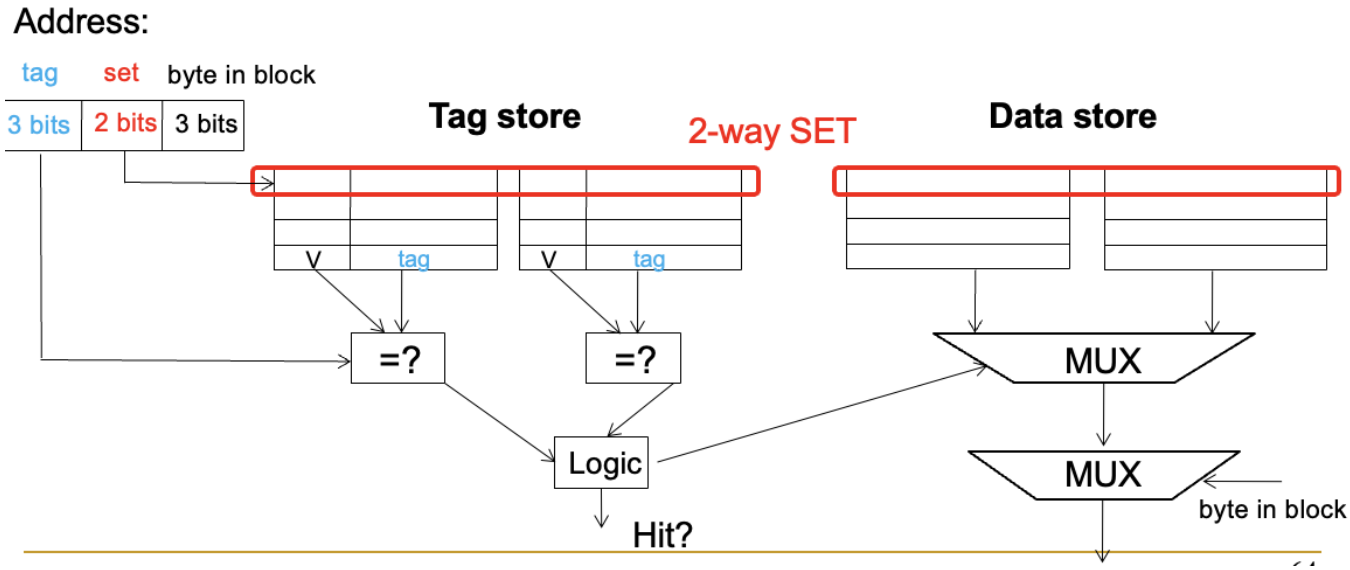
61

Advantage and Issue of Fully-associative Caches

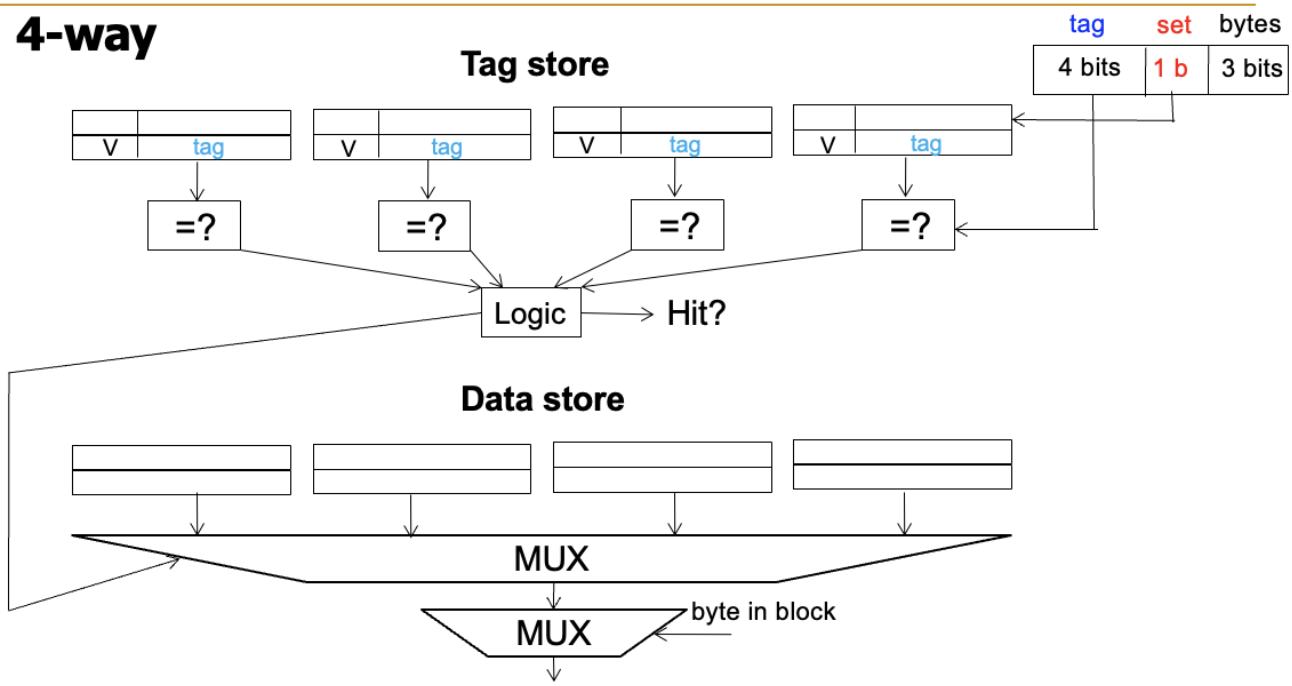
- **Fully-associative cache**:
  - A block can be placed in any cache block.
- Main advantage of fully-associative cache:
  - Highly utilization of cache blocks (global view)
- Main issue of fully-associative cache:
  - Can lead to extremely difficult to implement when the number of cache blocks in the cache is large.
    - Number of cache blocks in modern CPU reaches 32M/64=512K.
    - Choosing one out of 512K cache lines is extremely costly.

2-way Set-Associative Cache: Structure

- **Set-Associative Cache**
  - A block can be placed in any of N blocks of N-way set-associative cache
  - Example of 2-way cache:
    - Instead of having one column of 8, have 2 columns of 4 blocks



4-way Set Associativity



+ Likelihood of conflict misses even lower
-- More tag comparators and wider data mux; larger tags

Set-Associative Cache: Advantage and Issue

- **Set-Associative Cache**
  - **Key Idea:** Associative memory within the set

- **Advantage** of Set-Associative Cache
  - Accommodates conflicts better (fewer conflict misses)
    - Assume addresses A and B have the same index bits but different tag bits
    - A, B, A, B, A, B, A, B, … → store in the cache set
    - All accesses are cache hit

- **Issue** of Set-Associative Cache
  - More complex, slower access, larger tag store

- N-Way组相连，理解为一个set由N个line组成。Way是line的单位名称。
- set是书架个数，way是每个书架上能放书的数量。
- 注意看way数多少和tag位数、set位数的关系！

Higher associativity Means

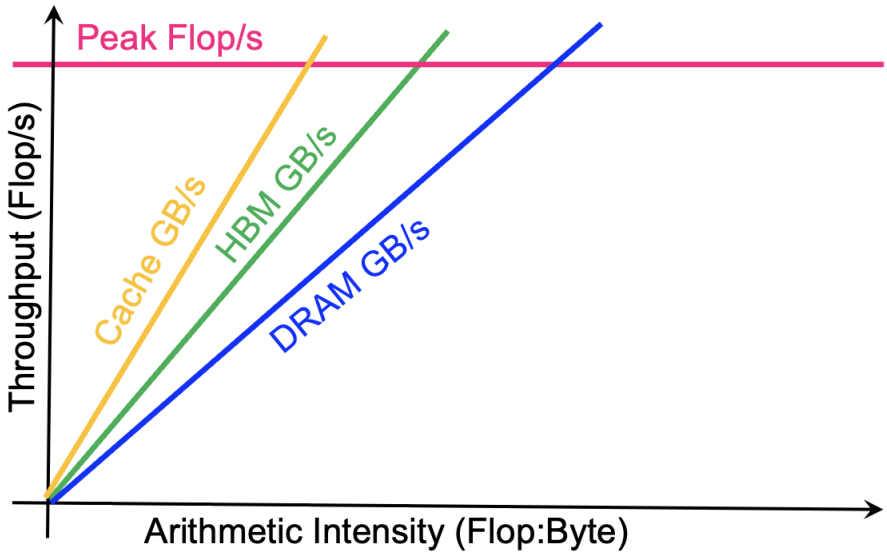
- ++ Higher hit rate
- -- Slower cache access time (hit latency and data access latency)
- -- More expensive hardware (more comparators)

Memory Roofline Model

Memory Roofline Model

Associativity (and Tradeoffs)

- **Memory Roofline Model**:
  - DRAM: limited memory bandwidth;
  - HBM: medium memory bandwidth;
  - Cache: large memory bandwidth



9.Cache and Cache Coherence

Cache Terminology

- Capacity (C):
  - the number of data bytes a cache stores
- Block size (b):
  - bytes of data brought into cache at once
- Number of blocks ($B = C/b$):
  - number of blocks in cache: $B = C/b$
- Degree of associativity (N):
  - number of blocks in a set
- Number of sets ($S = B/N$):
  - each memory address maps to exactly one cache set

- 关键问题： 替换cache里的哪个?

Replacement Policy

- Random
- FIFO
- Least recently used (how to implement?)
- Hybrid replacement policies
- Optimal replacement policy?

LRU

- 完全的LRU太难实现
- 严格意义上的LRU不一定比近似的更好

一种特殊的LRU近似：PLRU

- **Pseudo LRU for 8-way set-associated cache:**
  - Assume 8 blocks (L0~L7) for a set, 7 bits for rule (B0~B6).
  - **PLRU Replacement Way Selection**: choosing the suitable way (L0~L7) based on the PLRU bits.
  - **PLRU Bits Updating Rule**: updating PLRU bits after replacing the way (L0~L7).

PLRU Replacement Way Selection:

If the PLRU bits are:						Then the way selected for replacement is:
B0	0	B1	0	B3	0	L0
	0		0		1	L1
	0		1	B4	0	L2
	0		1		1	L3
	1	B2	0	B5	0	L4
	1		0		1	L5
	1		1	B6	0	L6
	1		1		1	L7

PLRU Bits Updating Rule:

If the current access is to:	Then the PLRU bits in the set are changed to the following[1]:						
	B0	B1	B2	B3	B4	B5	B6
L0	1	1	x	1	x	x	x
L1	1	1	x	0	x	x	x
L2	1	0	x	x	1	x	x
L3	1	0	x	x	0	x	x
L4	0	x	1	x	x	1	x
L5	0	x	1	x	x	0	x
L6	0	x	0	x	x	x	1
L7	0	x	0	x	x	x	0

[1] x = Does not change.

- 可以看到，如果选择某一路，那么这一路对应的B都要取反。

LRU不一定比Random好：Set Thrashing

- LRU vs. Random: LRU is not always better.
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the “program working set” in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs

Optimal Replacement Policy： Belady’s OPT

- Replace the block that is going to be referenced furthest in the future by the program
- 需要预知未来，无法达到，但是规定了上限

Write Policy

- **Step 1**: store insn. → cache, either policy works:
  - Write-allocate policy (memory, default):
    - Allocate the cache line (put it in the cache).
    - Issue: Read an entire cache block from memory
  - Write-no-allocate policy (PCIe/IO):
    - Write it directly to memory without allocation in cache.
    - Ignore cache.

- **Step 2**: cache → memory, either policy works:
  - Write-back policy (default):
    - Writes it to the cache and wait until the cache kicks the cache block out
  - Write-through policy (streaming write instruction):
    - Writes it to the cache and memory right away

- write-allocate为什么一定要在写入cache前从memory读入这个那个cache block?
- 因为cache的读写单位是block（64B），大于memory读写的单位B；如果不从memory读入，那么在cache写回memory时，除了cache要更新memory的那部分（假设是6B），其余memory对应的部分（64-6=48B）会被cache剩下的部分覆盖掉，由于没有提前从memory读取，可能是随机值，会污染memory。

1. Cache 的基本设计：以 Block 为单位管理

Cache 并不是以单个字节或字（word）为单位存储数据的，而是以 **Cache Block（缓存行）** 为单位（通常 64B）。这是因为：

- **空间局部性（Spatial Locality）**：程序访问的数据往往集中在相邻的内存区域，一次性加载整个 Block 可以减少未来访问的 Cache Miss。
- **减少管理开销**：如果 Cache 允许部分写入而不加载整个 Block，就需要更复杂的机制来跟踪哪些部分有效，这会增加硬件复杂度。

因此，Cache 必须保证整个 Block 是完整的，即使只修改其中的一部分。

Write-back vs. Write-through

- **Write-back**:
  - Write goes to cache; cache writes to main memory (evicted)

  + Can combine multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy

  -- Need a bit in the tag store indicating the block is “dirty/modified”

- **Write-through**:
  - Write goes to memory and cache

  + Simpler

  + Evictions do not need to write to memory

  + All levels are up to date

    Consistency: Simpler cache coherence because no need to check close-to-processor caches’ tag stores for presence

  -- More memory bandwidth intensive; no combining of writes

Data Cache vs Instruction Cache

分开设计还是统一设计?

- **Core question**: Separate or Unified?

- Pros and Cons of Unified Cache:

  + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., separate I and D caches)

  -- Instructions and data can thrash each other (i.e., no guaranteed space for either)

  -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- Modern CPU:
  - First level caches are almost always split
  - Higher level caches are almost always unified

低级cache分开设计，高级cache统一设计。

Cache Miss

Cache Miss 分类

- Compulsory miss
  - Defined as the first reference to an address (block), always resulting in a miss
- Capacity miss
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity
  - Cause: cache is too small to hold everything needed
- Conflict miss
  - defined as any miss that is neither a compulsory nor a capacity miss

减少CacheMiss的方法

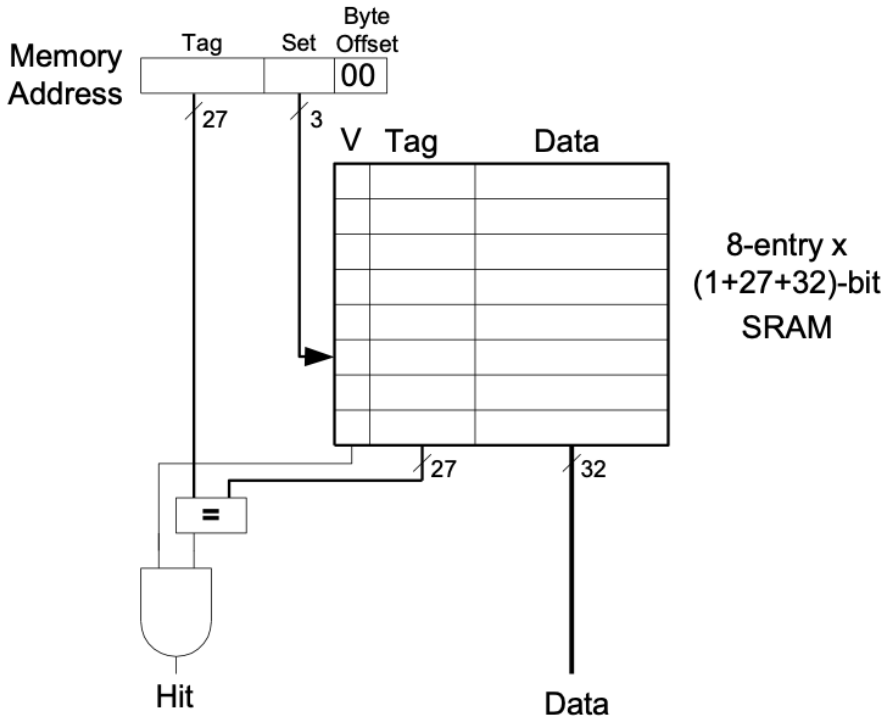
- Compulsory miss
  - Caching cannot help
  - Prefetching can: Anticipate which blocks will be needed soon
- Conflict miss
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Better, randomized indexing
    - Software hints?
- Capacity miss
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set and computation such that each “computation phase” fits in cache

---

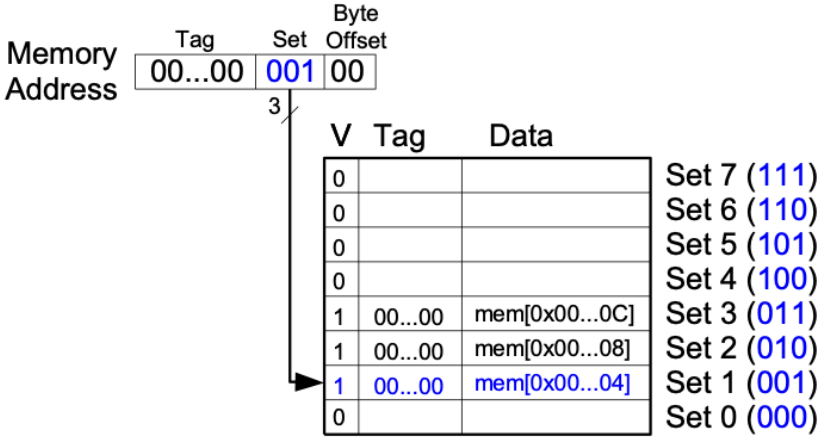
具体计算（期末考！）

direct mapped

Direct Mapped Cache Hardware



Direct Mapped Cache Performance




```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 3/15

= 20%

Temporal Locality
Compulsory Misses