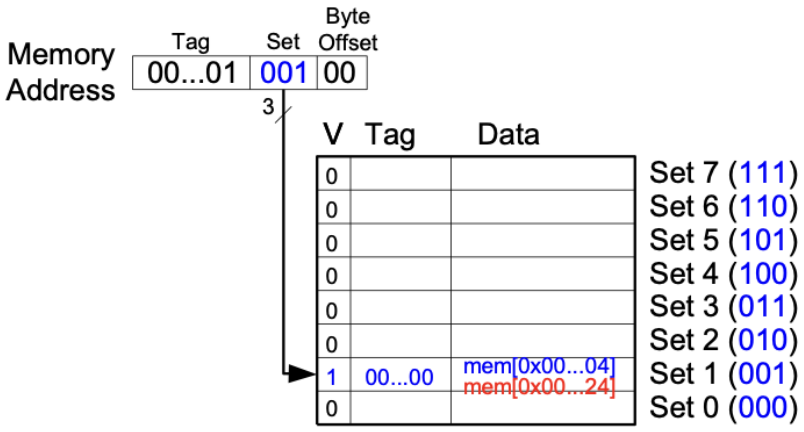
Direct Mapped Cache: Conflict

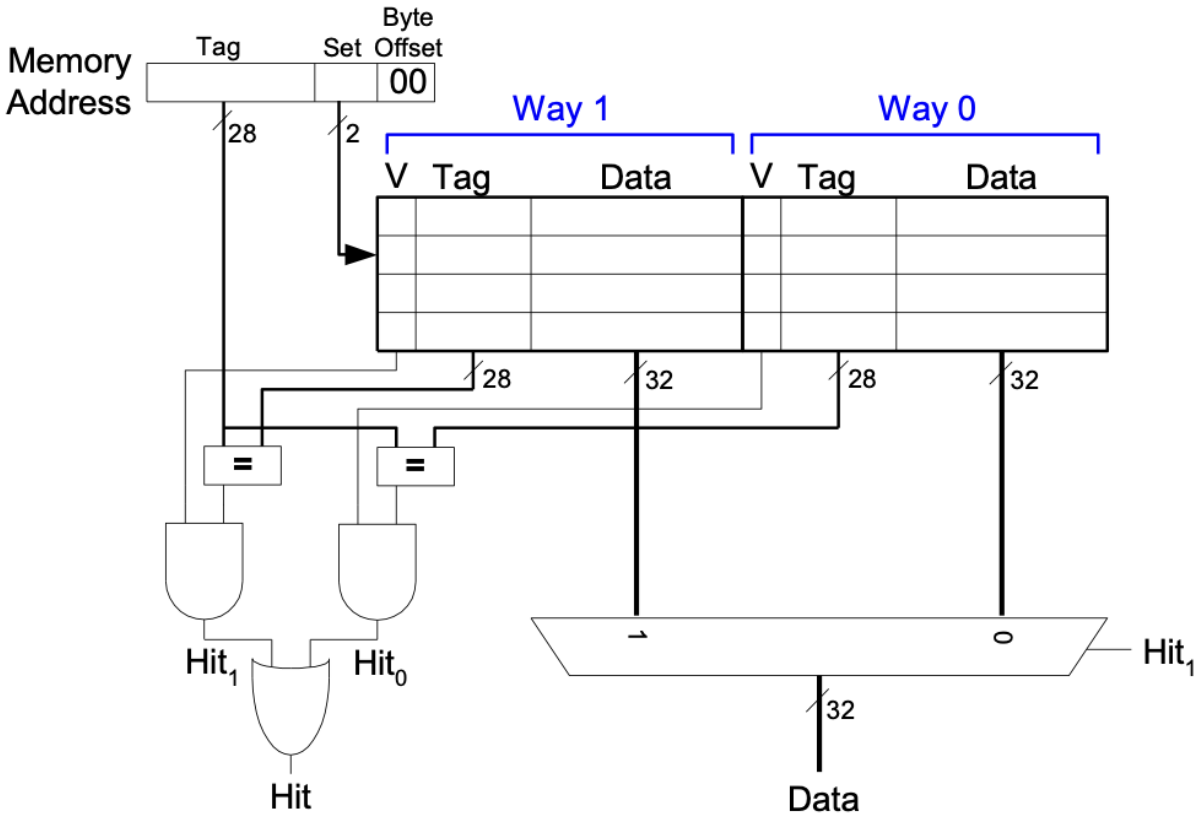



```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 10/10
= 100%

Conflict Misses

n-way set



N-way Set Associative Performance

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 2/10

= 20%

Associativity reduces conflict misses

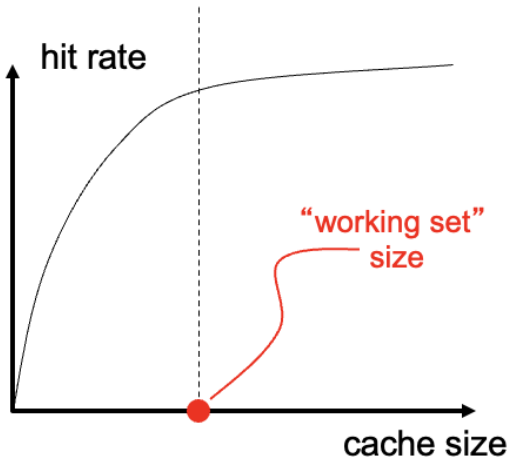
	Way 1			Way 0		
	V	Tag	Data	V	Tag	Data
Set 3	0			0		
Set 2	0			0		
Set 1	1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]
Set 0	0			0		

Size

Cache Size：关键是匹配working set的大小

Cache Size

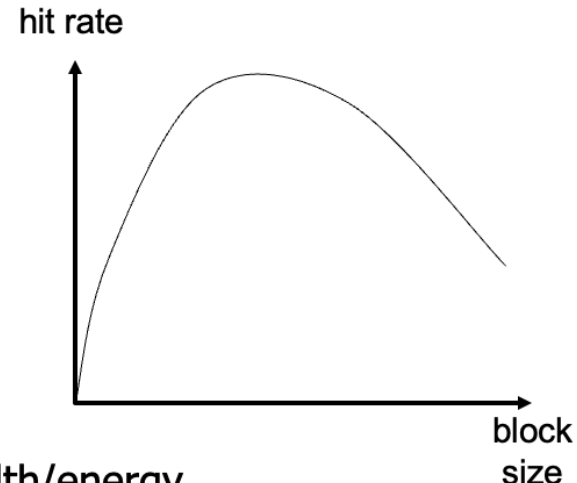
- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often
- Working set: the whole set of data the executing application references
  - Within a time interval



Block Size

Block Size

- Block size is the data that is associated with an address tag
- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead
- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high

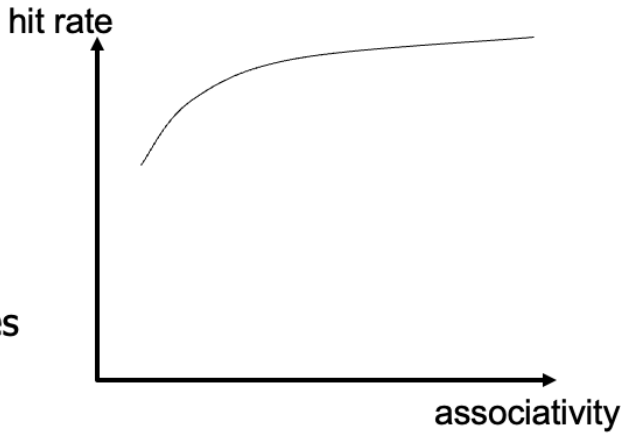


Associativity

- How many blocks can be present in the same index (i.e., set)?

- Larger associativity
  - lower miss rate (reduced conflicts)
  - higher hit latency and area cost (plus diminishing returns)

- Smaller associativity
  - Higher hit rate
  - lower cost
  - lower hit latency
    - Especially important for L1 caches



Cache in Multi-Core CPU （Coherence）

- Cores want a consistent view of memory.

资源共享的好处

- Resource sharing improves utilization/efficiency throughput
  - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data
- Reduces communication latency
  - For example, data shared between multiple threads can be kept in the same cache in multithreaded processors
- Compatible with the shared memory programming model

资源共享的坏处

- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it
- Sometimes reduces each or some thread’s performance
  - Thread performance can be worse than when it is run alone
- Eliminates performance isolation -> inconsistent performance across runs
  - Thread performance depends on co-executing threads
  - Uncontrolled (free-for-all) sharing degrades QoS
  - Causes unfairness, starvation

Private vs. Shared Caches

- **Advantages:**
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
    - If one core does not utilize some space, another core can
  - Easier to maintain coherence (a cache block is in a single location)

- **Disadvantages**
  - Slower access (cache not tightly coupled with the core)
  - Cores incur conflict misses due to other cores' accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

- 一般低级的cache是私有，高级的cache是共享。

Memory Consistency vs. Cache Coherence

Coherence

- 定义：保证单个内存位置的读写操作在所有处理器核心中的可见性顺序是一致的。
- 核心问题：解决多核环境下，同一内存地址的多个副本（如缓存中的副本）如何保持一致。

Consistency

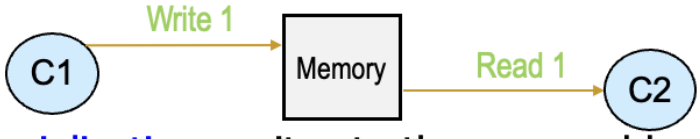
- 定义：规定多个内存位置的操作在全局执行时的可见顺序，即程序执行的正确性模型。
- 核心问题：确定不同内存地址的读写操作在不同核心中观察到的顺序是否符合预期（如程序顺序或特定模型要求）。

Cache Coherence的特征

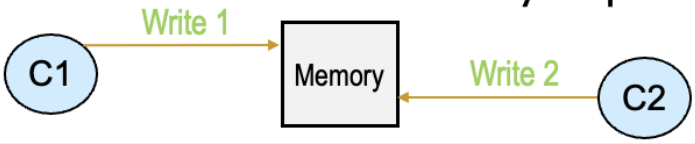
Features of Cache Coherence

- **Cache Coherence**: Multiple cores have a consistent state of the last written value from any core to a memory address.
  - Program order preservation: core C writes to the address and then reads from the same address, C gets value written.

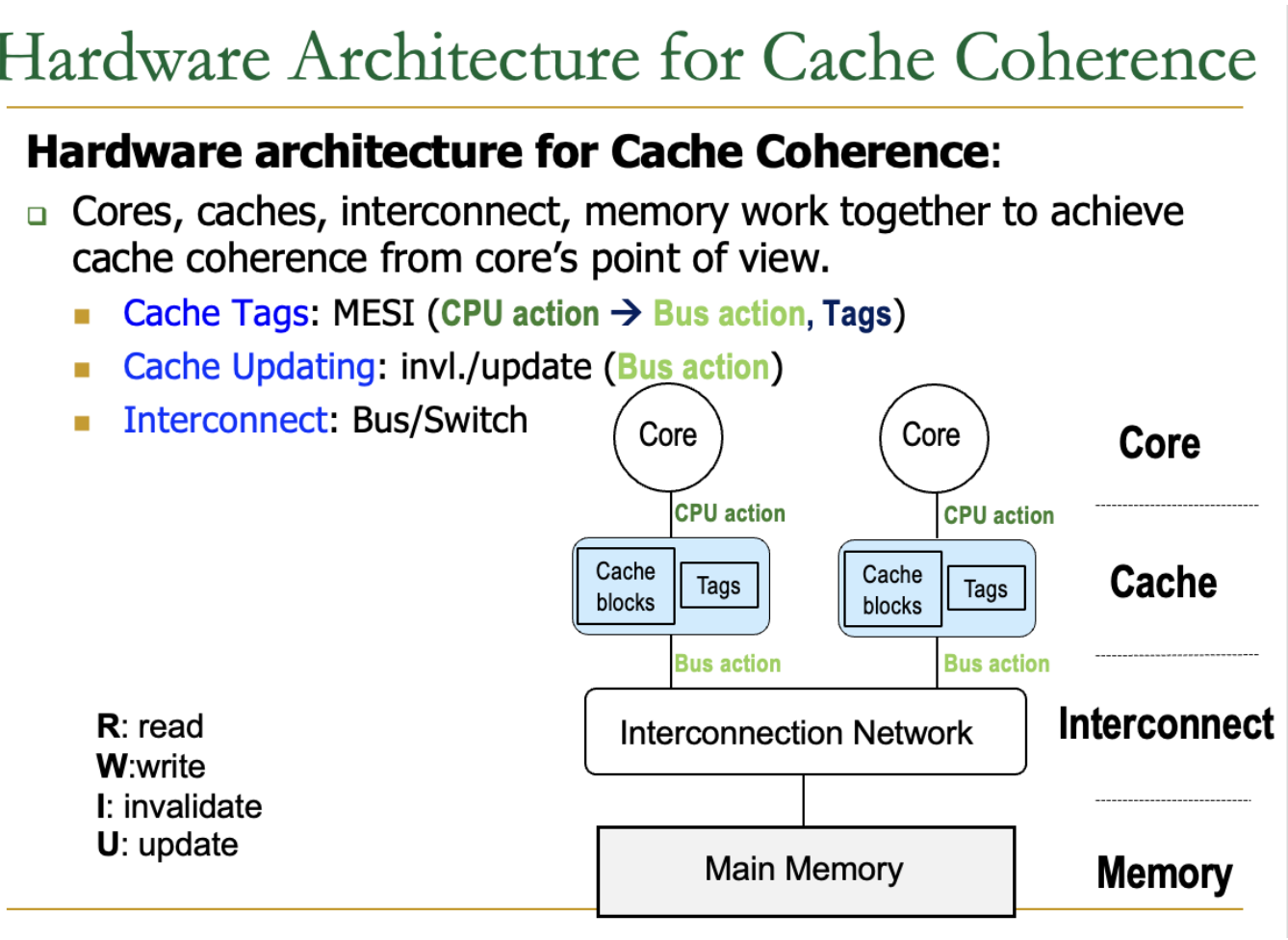
  - Coherent memory view: if C1 performs "mem[X] = 1", after a sufficient time, C2 will read 1 from "mem[X]".



  - Write serialization: writes to the same address by different processors are seen in same order by all processors.



硬件结构



InterConnect（互联）

Bus-based Protocol

- 1, A cache arbitrates for bus access, waiting until 2 happens
- 2, A cache is granted bus access
- 3, A cache places command on bus, waiting until 4 happens
- 4, Other caches place responses on bus

Switch-based protocol

- Each core pair can independently communicate with each other.

Update Policy：分为update和invalidate两种

Where and When: On a bus action write miss

Update Protocol

- Broadcast written data and address to cores
- Cores update the data in their caches if block is present

Invalidate Protocol

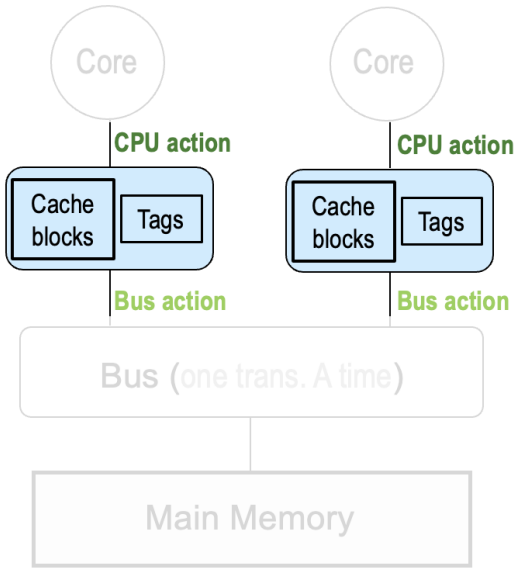
- Broadcast written data and address to cores
- Cores update the data in their caches if block is present

优缺点对比

- **Update Protocol**
  - \+ If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reacquire (broadcast update pattern)
  - – If data is rewritten without intervening reads by other cores, updates would be useless
  - – Write-through cache policy ➔ bus becomes bottleneck
- **Invalidate Protocol**
  - \+ After invalidation broadcast, core has exclusive access rights
  - \+ Only cores that keep reading after each write retain a copy
  - – If write contention is high, leads to ping-ponging (rapid invalidation-reacquire traffic from different processors)

Cache Tags：MSI Protocol

- **MSI Protocol**: safely update replicated data in caches (goal).
  - I(nvalid): block is not in cache, need to fetch from memory or other cache
  - S(hared): in >=1 caches, clean, local cores can read it w/o bus action
  - M(odified): in 1 cache, core can read/write it w/o bus action
- **Input**:
  - CPU action of one requested core
- **Output**:
  - Bus action of one requested core
- **Changed States**:
  - Modified cache tag



- Bus actions affects the overall performance of multi-core CPU.

状态转化图

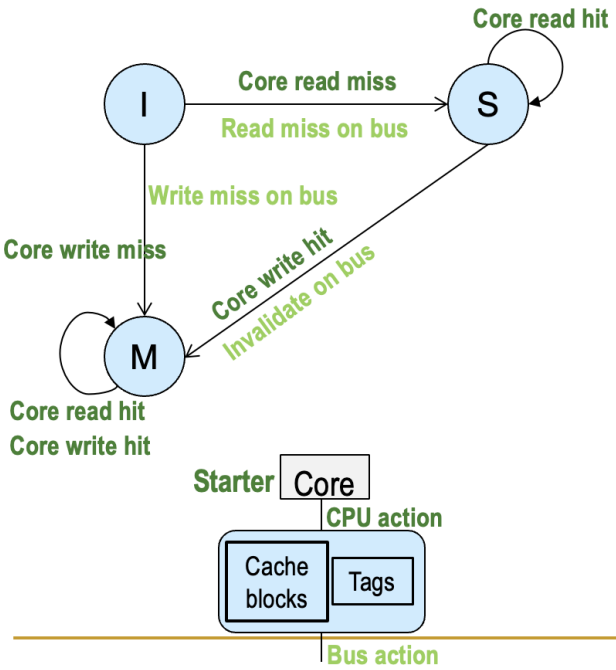
State Diagrams for CPU and Bus Requests

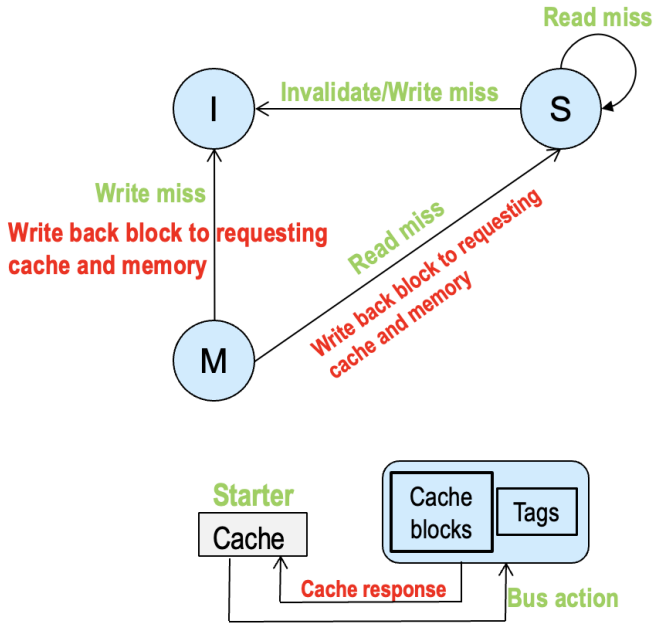
Core's memory read/write → Cache states, Bus action:

1, Miss in local cache
2, Hit in local cache

Bus action → Cache states:

1, Invalidate
2, Write miss
3, Read miss





MSI存在的问题

A block is not in cache at the beginning. On a read, the block immediately goes to the "**Shared**" state.

Problem: Core issues a bus action "invalidate" before writing the block to cache, even when only one cache copy exists.

Time	P1 op.	P2 op.	State A in P1	State B in P2	Bus action
t0			I	I	
t1	Read A		S	I	Read miss A
t2	Write A		M	I	Invalidate
t3		Read B	M	S	Read miss B
t4		Write B	M	M	Invalidate

MESI Protocol

- **MESI Protocol**: Illinois protocol (ISCA, 84)
  - I(nvalid): block is not in cache, need to fetch from memory or other cache
  - S(hared): in >1 caches, clean, local cores directly reads it w/o bus action
  - M(odified): in 1 cache, local core can read/write it w/o bus action
  - E(xclusive): in 1 cache, clean, local core reads/writes it w/o bus action

- **Key Differences** from MSI Protocol:
  - Local core reads block in state E, the state holds
  - Local core writes block in state E → state M, without bus action
  - Remote core reads, via read miss on bus, block in state E → state S
  - Remote core writes, via write miss on bus, block in state E → state I

MSI vs MESI

MSI协议

MSI协议是一种基本的缓存一致性协议，它定义了三种缓存块状态：

1. **M(Modified/修改)**: 缓存块已被修改，与主内存不一致，只有当前缓存拥有有效副本
2. **S(Shared/共享)**: 缓存块未被修改，与主内存一致，可能存在于多个缓存中
3. **I(Invalid/无效)**: 缓存块无效或不在当前缓存中

MSI协议通过总线监听机制维护一致性，当发生读写操作时，会根据当前状态和操作类型进行状态转换。

MESI协议

MESI协议是MSI协议的扩展，由伊利诺伊大学在1984年提出，新增了E(Exclusive/独占)状态：

1. **M(Modified)**: 仅存在于一个缓存中，已被修改，与主内存不一致
2. **E(Exclusive)**: 仅存在于一个缓存中，未被修改(干净)，与主内存一致
3. **S(Shared)**: 可能存在于多个缓存中，未被修改(干净)
4. **I(Invalid)**: 缓存块无效或不在当前缓存中

MESI与MSI的关键区别

1. **E状态的引入**：MESI通过E状态可以识别"干净且独占"的情况，此时本地核心可以无需总线操作直接读写
   - 本地读E状态块：保持E状态
   - 本地写E状态块：转为M状态，无需总线操作
2. **对远程请求的响应**：
   - 远程读E状态块：转为S状态
   - 远程写E状态块：转为I状态

MESI over MSI

MSI:

Time	P1 op.	P2 op.	State A in P1	State B in P2	Bus action
t0			I	I	
t1	Read A		S	I	Read miss A
t2	Write A		M	I	Invalidate
t3		Read B	M	S	Read miss B
t4		Write B	M	M	Invalidate

Time	P1 op.	P2 op.	State A in P1	State B in P2	Bus action
t0			I	I	

MESI:

t1	Read A		E	I	Read miss A
t2	Write A		M	I	
t3		Read B	M	E	Read miss B
t4		Write B	M	M	

10.Cache Coherence and Cache Consistency

Snoop-Based Cache Coherence （本质上是Bus类型）

动态播放ppt10的P26、27

- BUS存在的问题
  1. **Ordering**：
     - 总线对请求排序，但一致性仅要求对**同一地址的请求**严格排序，不同地址的请求可以乱序执行。
     - 这种宽松的排序要求提高了系统并行性。
  2. **Communication**：
     - 总线虽然使用广播，但一致性**不需要广播**，只需与持有数据的缓存（sharers）通信。
     - 实际系统中，可以通过更精细的机制（如目录）优化通信，避免不必要的广播。

Directory-Based Cache Coherence

- goal：snoop依赖总线的全对全广播（all-to-all broadcast），每个缓存操作需要通知所有其他缓存，无论它们是否持有相关数据；而directory则是在缓存操作前先查询目录，仅与实际的共享者通信（而非广播）。
- idea：

1. **逻辑中心化的目录（Logically-central Directory）**
   - **核心思想**：系统维护一个**全局目录**，记录每个缓存块（Cache Block）的所有副本位置（即哪些处理器/节点的私有缓存中有该块的副本）。
   - **作用**：
     - 当某个缓存需要访问数据时（如写入），先查询目录，确定其他缓存中是否存在副本，从而避免不一致。
     - 目录是“逻辑上中心化”的，但物理上可以是分布式的（例如分片存储在不同节点上）。

3种Node的定义

Home Node（主节点/归属节点）

- **定义**：
  - 每个缓存块在系统中有一个唯一的**Home Node**，该节点**物理上存储该块的目录信息**（如状态、副本位置等）。
  - 不同缓存块的Home Node可以不同（例如通过地址分片或哈希分布）。
- **职责**：

    - 维护目录：记录该缓存块的当前状态（如共享、独占、无效）以及哪些节点（Local Nodes）持有副本。
    - 处理一致性请求：当其他节点（Local Nodes）发起读写请求时，Home Node根据目录信息协调一致性操作（如发送无效化消息、转发数据等）。
- **关键点**：
    - Home Node是**目录协议的权威中心**，但对不同缓存块是分布式的（避免单一节点成为瓶颈）。

Local Node（本地节点/请求节点）

- **定义**：
    - 发起对某个缓存块的**读/写请求**的节点。
    - 可能是处理器核心的私有缓存（L1/L2 Cache）或NUMA架构中的本地内存控制器。
- **职责**：
    - 向Home Node发送请求（如读请求、写请求、升级请求等）。
    - 根据Home Node的响应执行操作（如加载数据、无效化本地副本等）。
- **关键点**：
    - Local Node是**主动方**，其行为触发一致性协议的执行。

Remote Node（远程节点/响应节点）

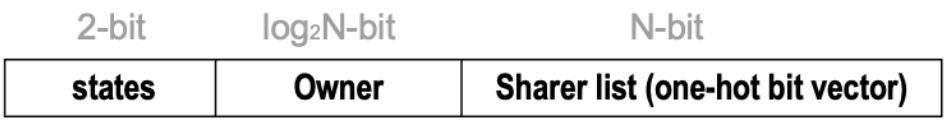
- **定义**：
    - **被动响应**Home Node指令的节点，通常是当前持有该缓存块副本的节点。
    - 不主动发起请求，仅在Home Node协调下参与一致性操作。
- **职责**：
    - 接收Home Node的请求（如“提供数据”或“无效化副本”），并执行相应动作。
    - 例如：
        - 若缓存块处于**共享状态**，Remote Node可能直接向Local Node发送数据副本。
        - 若缓存块处于**独占状态**，Remote Node需将数据写回Home Node并降级状态。
- **关键点**：
    - Remote Node是**被动方**，其行为由Home Node的一致性协议驱动。

三者的交互流程示例

以**写请求**为例：

1. **Local Node**发起写请求，向该块的**Home Node**查询。
2. **Home Node**检查目录：
    - 若有其他节点（Remote Nodes）持有副本，发送**无效化消息**给这些节点。
    - 等待所有Remote Nodes确认无效化完成。
3. **Remote Nodes**收到无效化消息后：
    - 若缓存块被修改过（脏数据），将数据写回Home Node。
    - 将本地副本标记为无效，并发送确认。
4. **Home Node**收到所有确认后：
    - 更新目录状态为“独占”，并授权Local Node执行写操作。

Directory结构



- **Detailed directory for each cache line:**
  - Each cache block needs $N+\log_2 N + 2$ bits for its directory, which resides at the home node.

- 具体变化见第10章ppt的第36、37页。

Cache Consistency

- **Consistency**: A contract between programmer and microarchitect.
  - Preserving an "expected" (more accurately, "agreed upon") order simplifies programmer's life
    - Ease of debugging; ease of state recovery, exception handling
  - Preserving an "expected" order usually makes the hardware designer's life difficult
    - Especially if the goal is to design a high performance processor: Recall load-store queues in out of order execution and their complexity

Four Types of Memory Barrier

- load对应读，store对应写
  - Load-Load:
    - Effectively prevents ordering of loads performed before the barrier with loads performed after the barrier
  - Load-Store:
    - Effectively prevents ordering of loads performed before the barrier with writes performed after the barrier
  - Store-Store:
    - Effectively prevents ordering of stores performed before the barrier with stores performed after the barrier
  - Store-Load:
    - Effectively prevents ordering of stores performed before the barrier with loads performed after the barrier

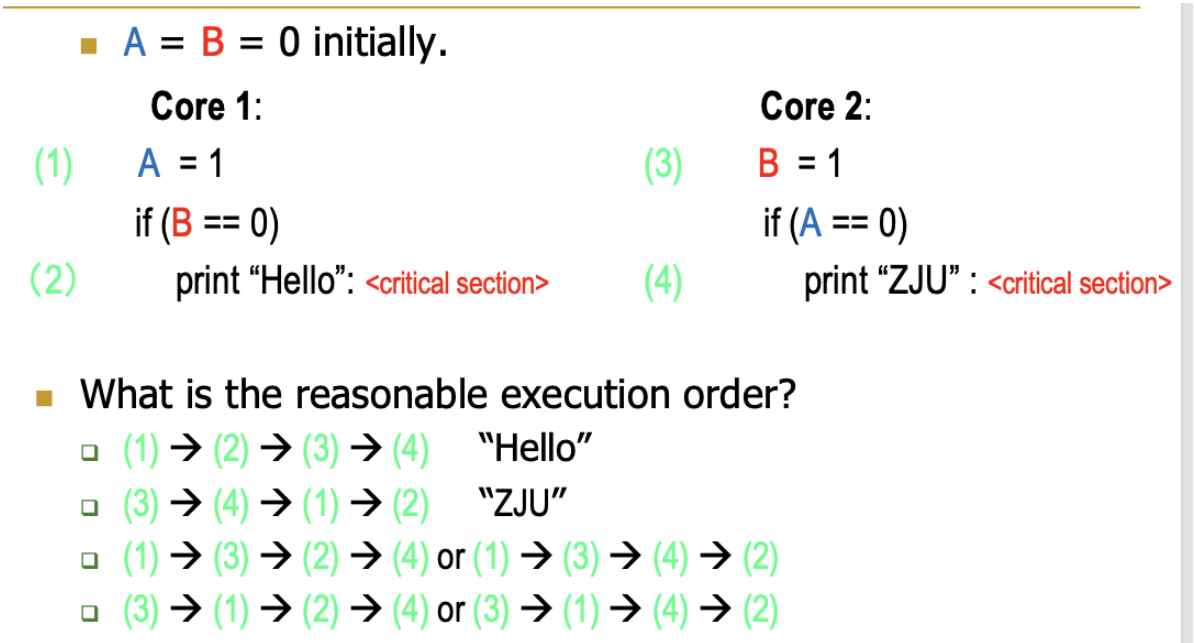
- 一致性越差，性能越好；
- 一致性越好，编程越容易；

Four Memory Barriers vs. Consistence Model

Load-Load	Load-Store	Store-Store	Store-Load	Consistence Model	CPU
√	√	√	√	Sequential Consistency	Dual 386
√	√	√		Total Store Order	X86/64
√	√			Partial Store Order	Arm
				Really weak memory model	DEC Alpha

Sequential Consistency 顺序一致性

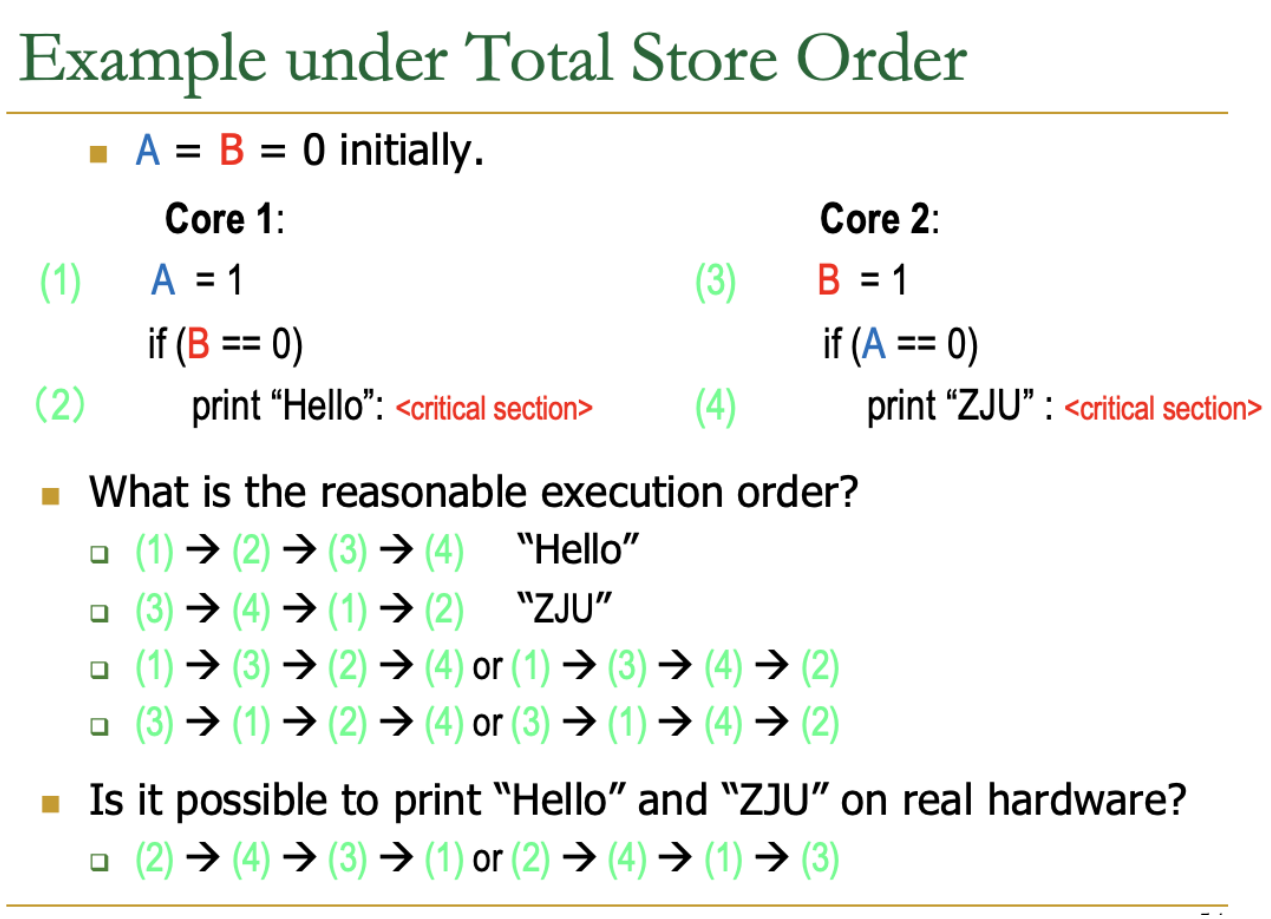
- 满足全部4种barrier
- Memory is a switch that services one load or store at a time from any processor
  - All processors see the currently serviced load or store at the same time
  - Each processor's operations are serviced in program order（每个处理器内部的顺序不能被打乱）



- 在保证顺序一致性的前提下：上面这种情况无法实现。

Total Store Order： Sequential Consistency + Store Buffer

- 不满足先写后读
- 在Sequential Consistency的基础上增加一个store buffer，每次写操作不是直接写入cache，而是写入一个store buffer内，store buffer满了再写入cache。好处是优化写入的性能，坏处是无法满足Store-Load Barrier。
- 注意，当store buffer满的时候，数据被写入cache中的顺序与先前数据被写入store buffer的顺序相同。
- 无法满足先写后读的原因：read操作是直接去cache里读，但是被写入的内容可能还在store buffer中未被写入。



- 在这种条件下，可能出现先2后1的情况，因此可以打印出HelloZJU。

Partial Store Order： Total Store Order + Write coalescing

- 不满足先写后读、先写后写
- Write coalescing: merge writes to the same cache line inside the write buffer to save memory bandwidth。
- 与Total Store Order的区别在于：数据从write buffer被写入cache时，顺序是固定的，与数据先前被写入store buffer的顺序无关，因此不满足store-store buffer。

- 手机一般是partial store order

Code:

A[0] = 8
A[5] = 8
A[11] = 8

Write buffer:

A[0] = 8			
	A[5] = 8		
			A[11] = 8

Executed:

A[0] = 8
A[5] = 8
A[11] = 8

Code:

A[0] = 8
A[5] = 8
A[3] = 8

Write buffer:

A[0] = 8			A[3] = 8
	A[5] = 8		

Executed:

A[0] = 8
A[3] = 8
A[5] = 8

- Writing to A[3] and A[5] is re-ordered

Order是通过排序shared data来影响程序正确性的！ 互斥原则(Mutual Exclusion)

- "Only one thread can execute a critical section at a given time" - 在任何给定时间，只能有一个线程执行临界区代码。这确保了共享数据不会被多个线程同时修改。
- 因此硬件原语要能支持原子操作就很重要。

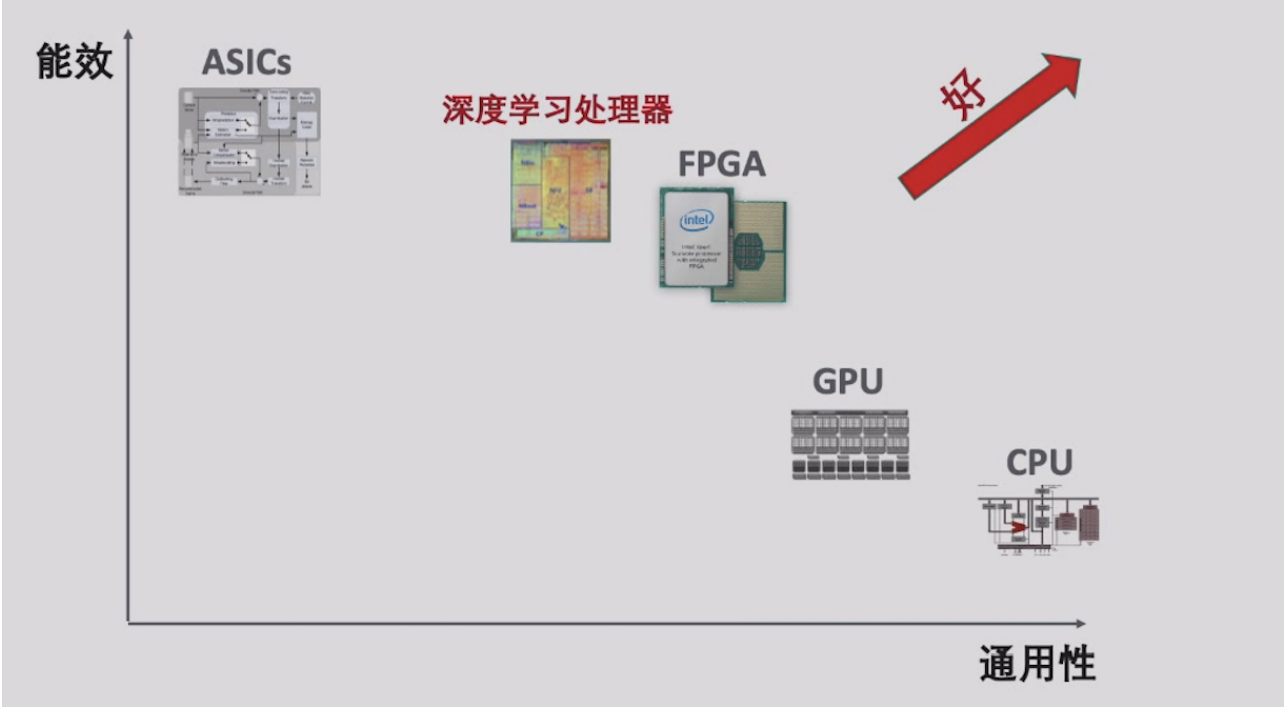
Multi-level Caching in a Pipelined Design

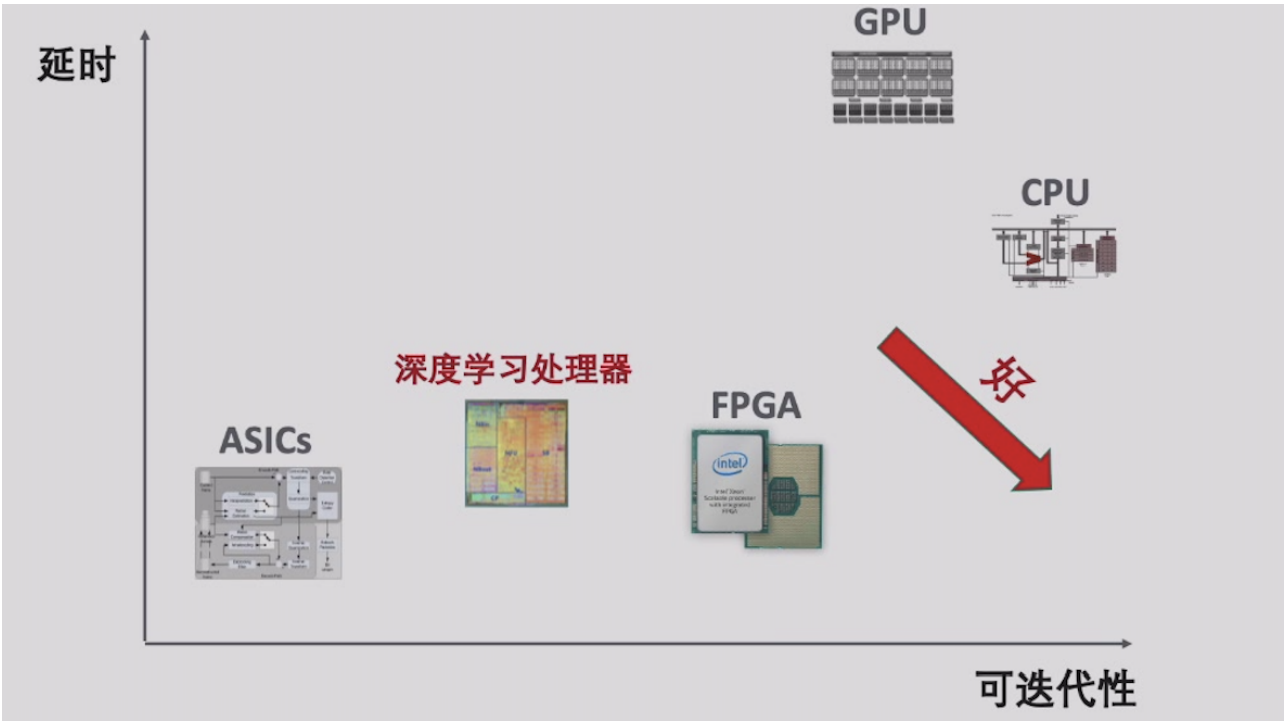
- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity; latency is critical
  - Tag store and data store usually accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store can be accessed serially
- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different

11.深度学习加速器 DL Accelerator

需要深度学习加速器的原因

- 1.深度学习市场大
- 2.利用CPU/GPU处理神经网络效率太低 **CPU/GPU/DL加速器 比较**





深度学习算子分析

两大特性

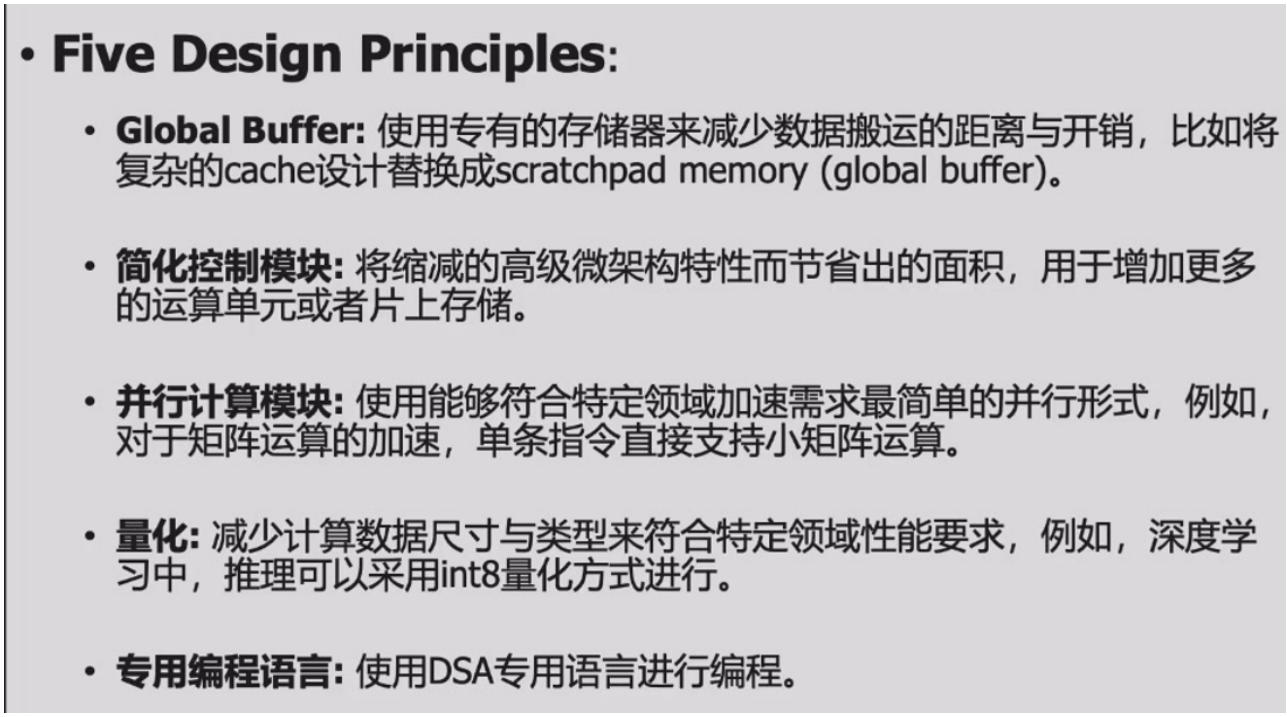
- 计算特性：是否存在重复、固定的计算模式（存在循环）？
- 访存特性：是否具有局部性（连续访问）？访存与计算的关系？ **以VGG19为例：**

- 矩阵运算占比高达90%

Operator	计算特性	访存特性
Conv	矩阵相乘	Burst+stride
Activation	单向量操作	Sequential
Pooling	单矩阵Reduce操作	Burst+stride
FC	矩阵相乘	Sequential

深度学习加速器设计思路

- DSA：Domain Specific Architecture



并行计算模块

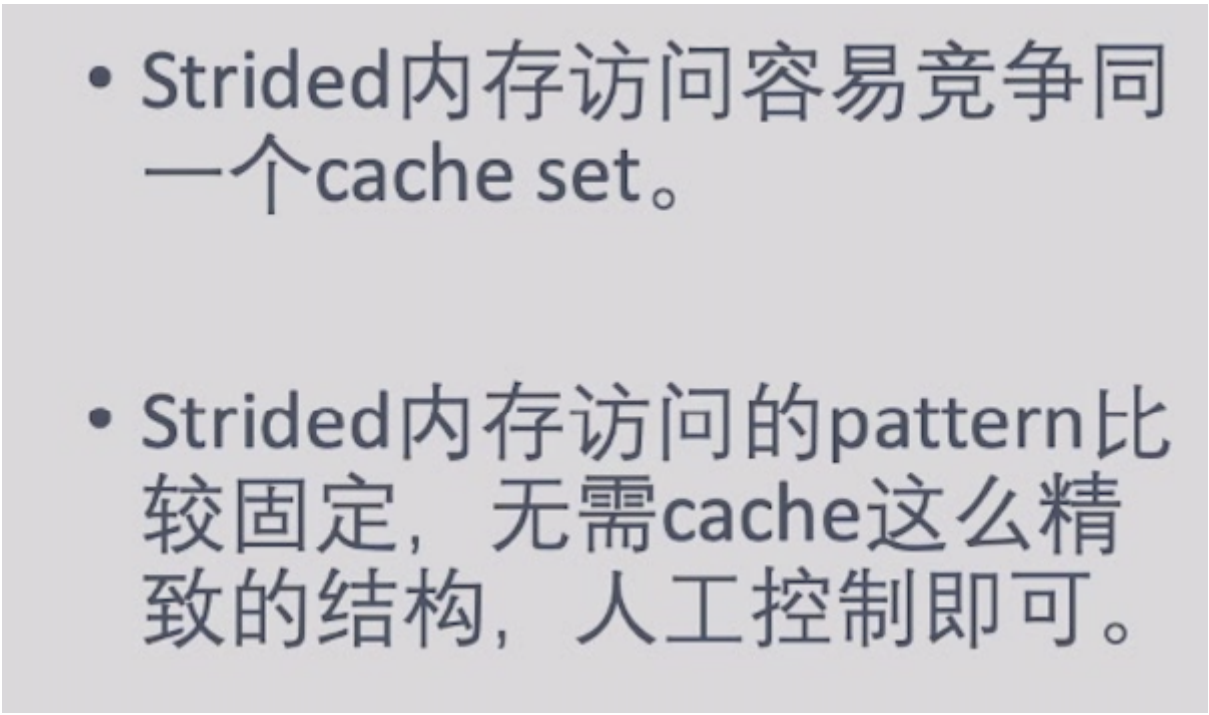
- 设计专门用于矩阵计算的单元。（CPU属于样样会，但不精通；而在加速器中，设计只会矩阵计算的模块，但非常高效）

简化控制模块

- 优化重点并不是提升指令的并行，也就是说不是在控制模块。（CPU的控制模块很复杂，因为它要处理许多不同的任务）

Global Buffer

- cache对stride很不友好！



- 有很多分块的global buffer ，用buffer代替cache

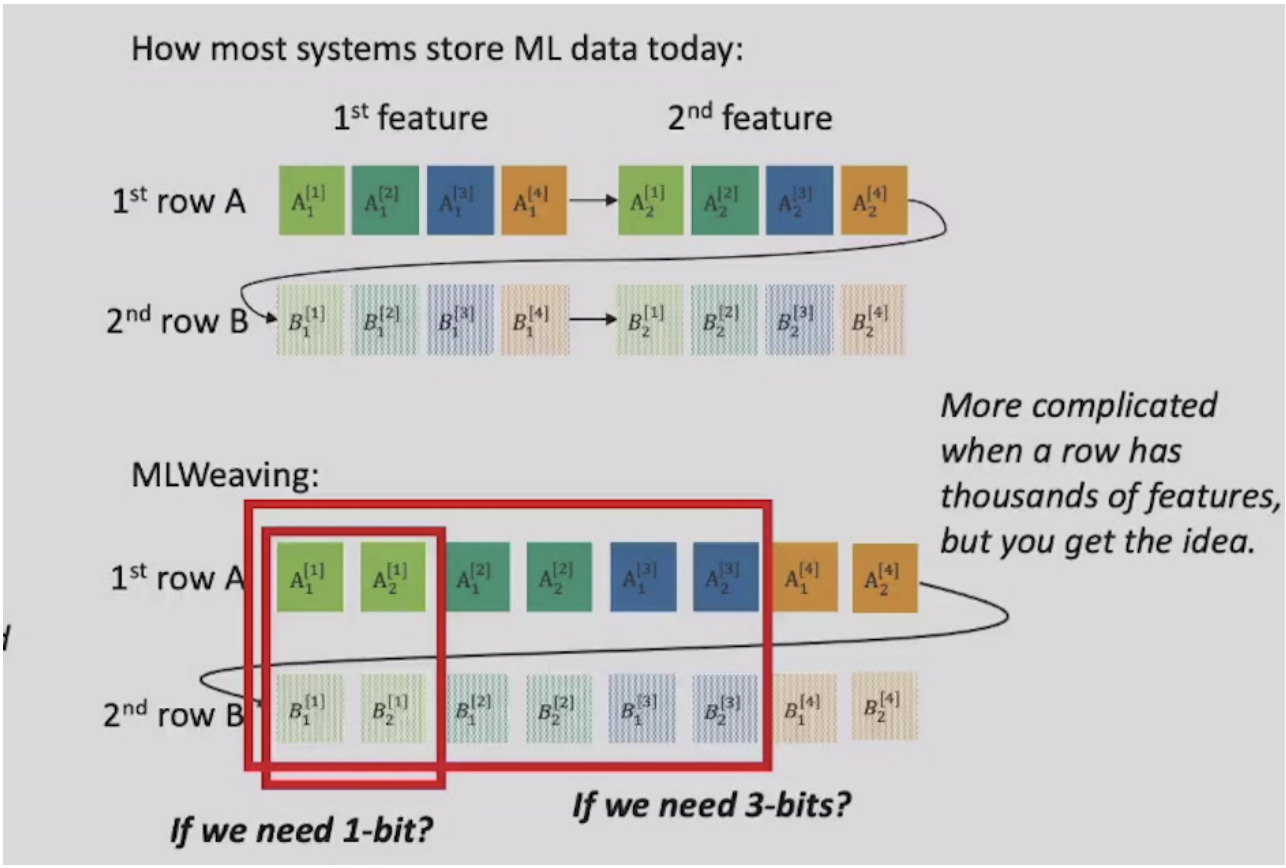
	Cache	Buffer
能耗	高	低
芯片面积	大	小
管理方式	自动	手动

- AI加速器的主要目标：提高算力、降低功耗！

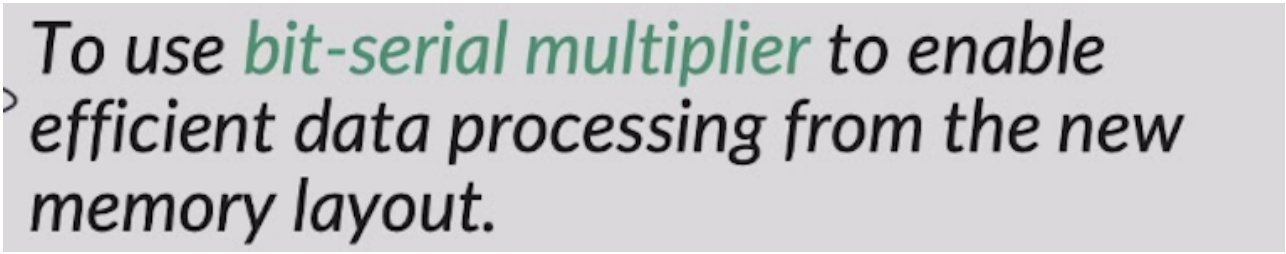
量化

- AI领域对精度要求不高。（比如图片二分类，只需要最后预测概率大于0.5，就可以算是预测正确；那么预测概率是0.777777和0.777778其实差别并不大）
- CPU/GPU都不太支持低精度，而AI处理器（比如TPU、Ascend等）能较好地支持低精度的运算。

- 一种新的数据储存方式：按数据位进行储存



- 上面那种数据储存方式下，硬件设计的核心理念：**BSM：bit serial multiplier**



具体方式见ppt。

专用编程语言

- 难编程，高性能，需要显式调用硬件（比如buffer）。
- 商家提供算子借口、调用库。

综合对比

	CPU	DSA
On-chip Memory	Cache	Global Buffer
Instruction Issue	Superscalar	In-order/simple
Parallelism	Inter-instruction	Intra-instruction
Fuctionality	Full	Partial
Optimization Purpose	Low Latency	High Throughput
Programming Language	General	Domain-specific

1. **On-chip Memory（片上存储器）**
   - **CPU**：使用缓存（Cache）来存储临时数据，以加快数据访问速度。
   - **DSA**：使用全局缓冲区（Global Buffer），这是一种更高效的内存结构，专为特定领域的高吞吐量需求设计。
2. **Instruction Issue（指令发射）**
   - **CPU**：支持超标量（Superscalar）架构，可以同时发射多条指令，利用指令级并行性提高性能。
   - **DSA**：通常采用顺序发射（In-order）或简单的指令发射方式，专注于特定任务的效率而非通用性。
3. **Parallelism（并行性）**
   - **CPU**：侧重于指令间并行（Inter-instruction），通过多线程、流水线等技术实现并行处理。
   - **DSA**：侧重于指令内并行（Intra-instruction），例如在单个指令中处理大量数据（如SIMD指令）。
4. **Functionality（功能性）**
   - **CPU**：功能完整（Full），支持通用计算任务，能够运行复杂的操作系统和应用程序。
   - **DSA**：功能部分（Partial），针对特定领域优化，牺牲通用性以换取更高的效率。
5. **Optimization Purpose（优化目标）**
   - **CPU**：优化目标是低延迟（Low Latency），确保快速响应和实时处理。
   - **DSA**：优化目标是高吞吐量（High Throughput），专注于在特定任务中处理大量数据。
6. **Programming Language（编程语言）**
   - **CPU**：支持通用编程语言（如C、C++、Python等），适用于广泛的应用场景。
   - **DSA**：通常使用领域特定语言（Domain-specific），例如用于图形处理的GLSL或用于张量计算的TensorFlow。

12.AI Processors

深度学习加速器设计目标

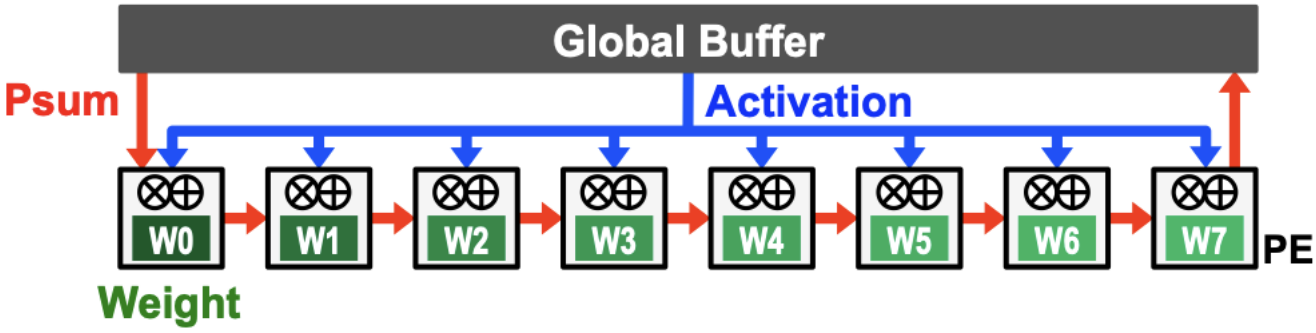
减少内存访问

- 当前的主要挑战是：算力不足，访存代价太大（跨芯片速度慢、耗能极大）
- 解决方案：用global buffer代替main memory（DRAM）
- MAC：Multiply-Accumulate，乘积累加运算，也就是矩阵运算。

减少Global Buffer访问

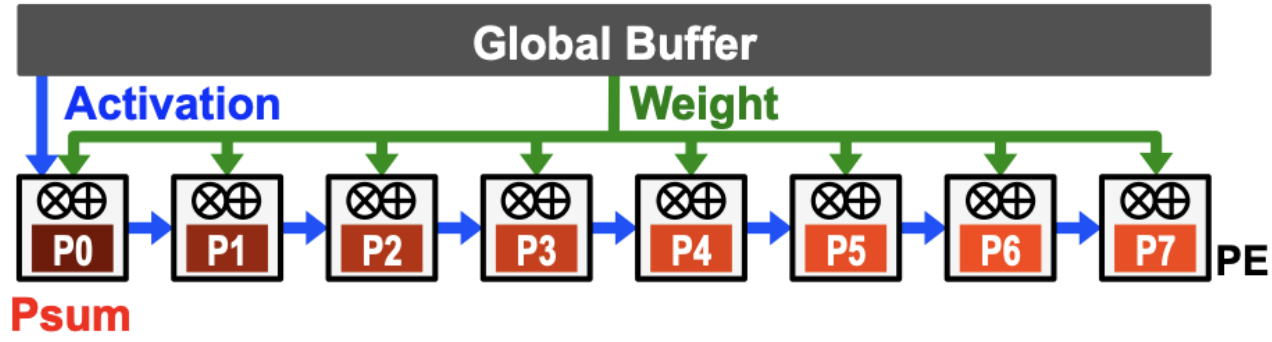
- 问题： Global Buffer的访问很贵
  - Flip-Flops
    - Very fast, parallel access
    - Very expensive (one bit costs tens of transistors)
  - Static RAM
    - Relatively fast, only one data word at a time
    - Expensive (one bit costs 6+ transistors)
  - Dynamic RAM
    - Slower, one data word at a time, reading destroys content (refresh), needs special process for manufacturing
    - Cheap (one bit costs only one transistor plus one capacitor)
  - Flash Memory
    - Much slower, access takes a long time, non-volatile
    - Very cheap (one transistor stores 16 bits or no transistors involved)
- 解决方案： 增加寄存器文件的使用，代替global buffer。也就是将数据留在处理器中，也就是留在寄存器中。具体以下四种方法：

Weight Stationary (WS)



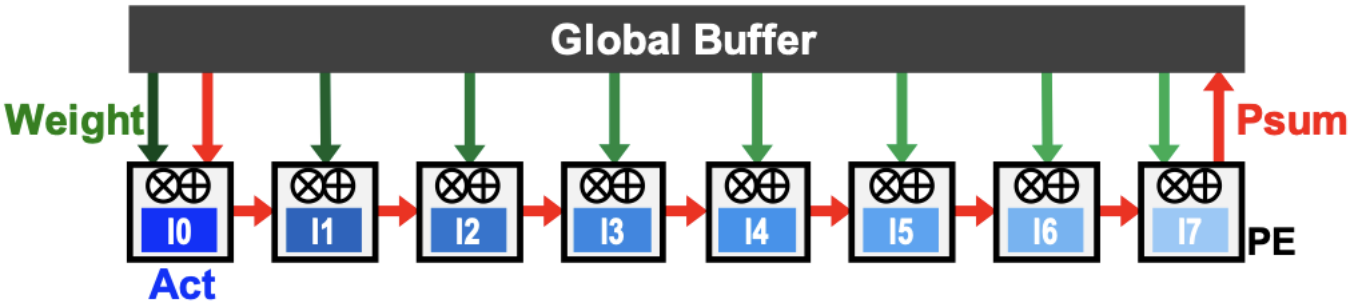
- **Key idea** (Systolic array):
  - 最大程度地减少从Global Buffer读取**Weight** (conv),
  - 广播Activations和沿着PE水平方向上累加**Psum**.
  - 例子: TPU [Jouppi, ISCA, 2017]

Output Stationary (OS)



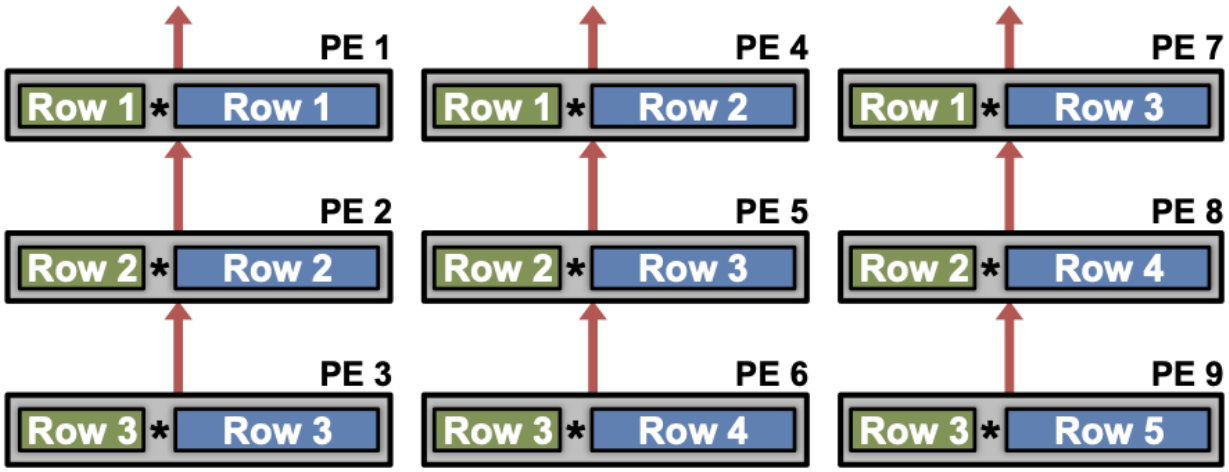
- Key idea:
  - 最大程度地减少从Global Buffer读取和存储**Psum**, 尽量把**Psum**留在PE内。
  - 广播**Weight**和沿着PE水平方向上复用**Activation**。
  - 例子: [Moons, VLSI, 2016]

Input Stationary (IS)



- Key idea:
  - 最大程度地减少从Global Buffer读取**Activation**, 尽量把**Activation**留在PE内。
  - 并行读**Weight**, 沿着PE水平方向上累加**Psum**。
  - 例子: [SCNN, ISCA, 2017]

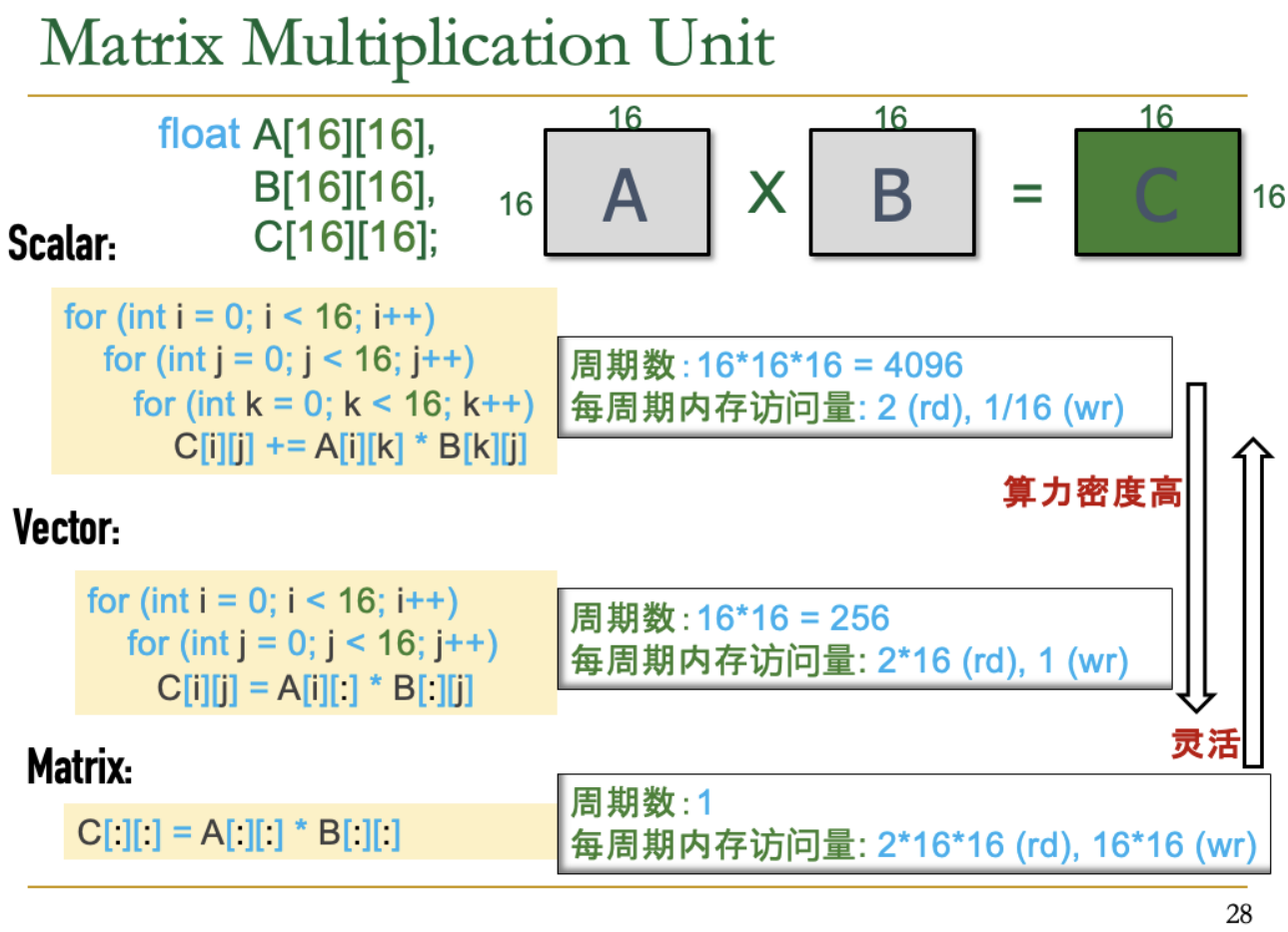
Row Stationary (RS)



- Key idea:
  - 从Global Buffer读出Filter中的一行和**Activation**的一个滑窗, 留在PE内。
  - 尽量减少从Global Buffer的整体读出量，而不只是一个维度的。
  - 例子: [Chen, ISCA, 2016]

增加计算模块

- 引入**矩阵乘法单元：Cube**

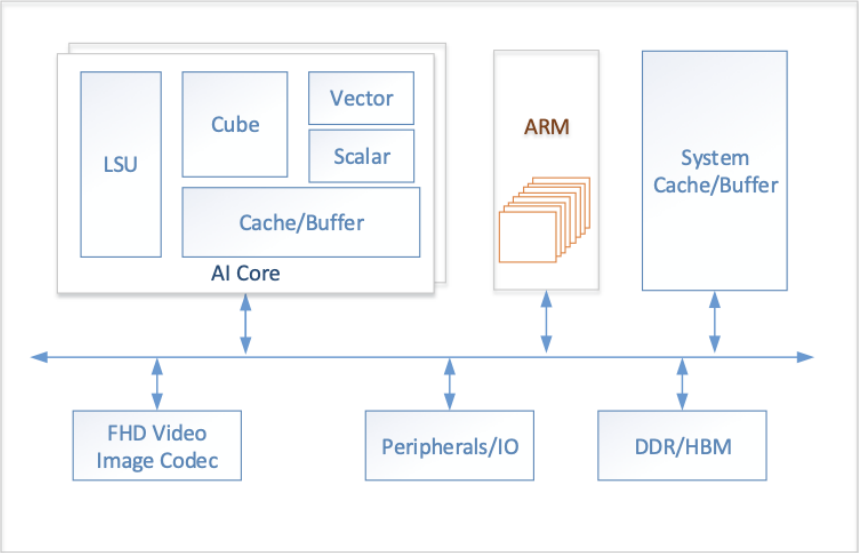


常见AI芯片比较

华为Ascend

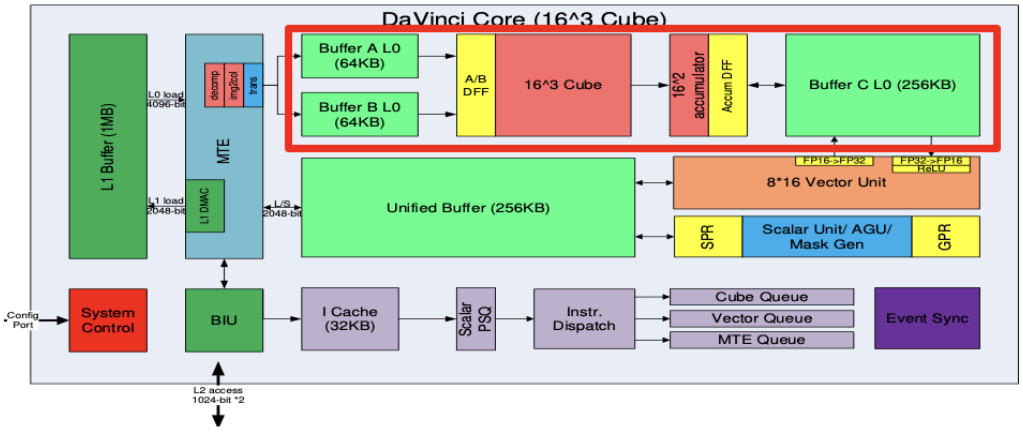
- 关键在于Cube，实现了单周期内的小矩阵直接乘法运算。

晟腾310/910 芯片结构示意图



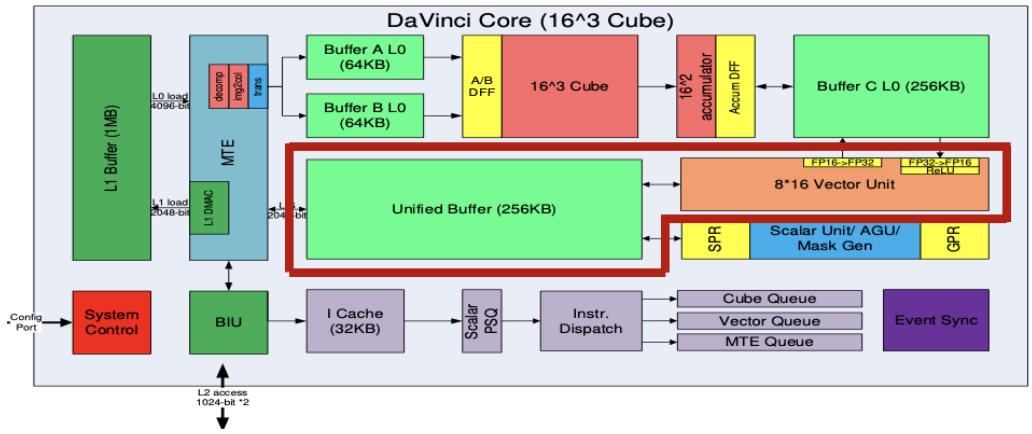
- **L2 Buffer vs. L2 Cache**
  - 同一个介质，两种使用模式
  - **Buffer**:程序员可见并可以直接读写(地址空间和DDR/HBM不重合)
  - **Cache**: 作为DDR/HBM高速缓存，程序员不可见
- **DDR/HBM**
  - **DDR**: 普通内存，带宽低/价格低，在推理芯片310中
  - **HBM**: High Bandwidth Memory, 带宽高，成本高，在训练芯片310中

Cube模块（矩阵运算，算力担当）



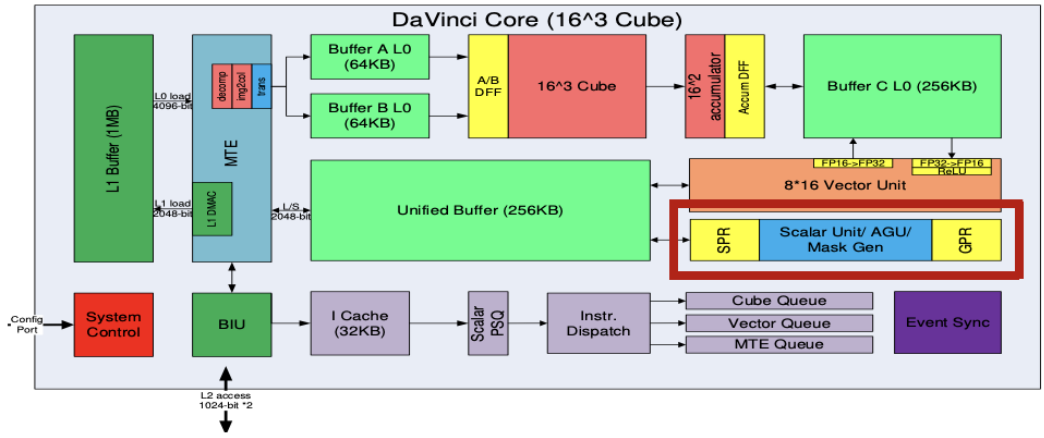
- **矩阵乘运算单元Cube**：一拍完成一个fp16的 2个16x16矩阵相乘；**C = A * B**; 如果是int8输入,则一拍完成 16*32 与 32*16 矩阵乘。
- **累加器Accumulator**: 把当前矩阵乘的结果与前次计算的中间结果相加（**C = A * B + C**），可以用于完成卷积中加bias操作。
- **L0A/L0B/L0C Buffer**: L0A 存储矩阵乘的左矩阵数据，L0B 存储矩阵乘的右矩阵数据，L0C 存储矩阵乘的结果和中间结果。
- **A/B DFF**: 数据寄存器，缓存当前计算的16*16 左/右子矩阵。
- **Accum DFF**：数据寄存器，缓存当前计算的16*16结果矩阵。

Vector模块（向量运算，多面手）



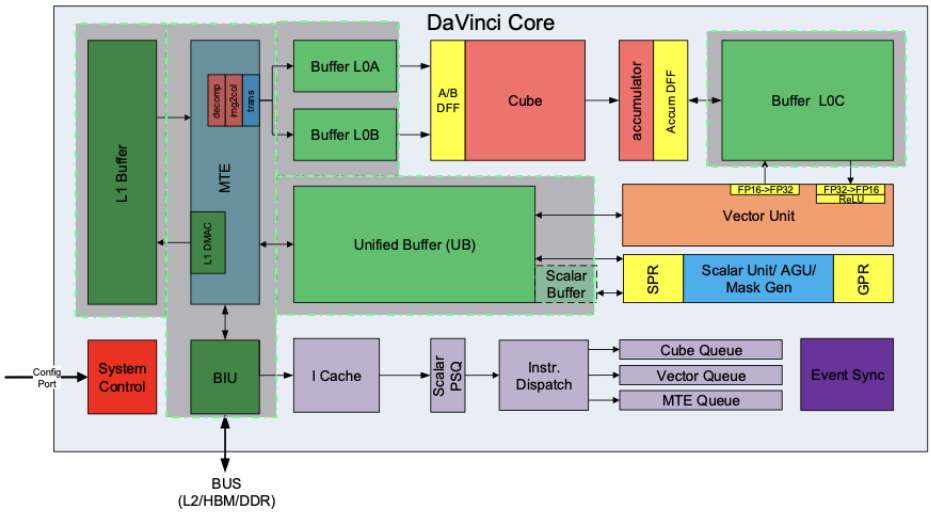
- **向量运算单元Vector Unit**：覆盖各种基本的计算类型和许多定制的计算类型，主要包括FP16/FP32/int32/Int8等数据类型的计算，支持连续或者固定间隔寻址；或者VA寄存器寻址（不规则向量运算）
- **SIMD长度**：一条Vector指令可以完成两个128长度fp16类型的向量相加/乘，或者64个fp32/int32类型的向量相加/乘
- **Unified Buffer(UB)**：保存Vector运算的源操作数和目的操作数；一般要求32Byte对齐；
- **数据从L0C->UB**：随数据搬运在Vector Unit完成一些RELU/数据格式转换等操作

Scalar模块（标量运算，司令部）



- **Scalar Unit**：负责完成AICore中的标量运算，功能上可以看做一个小CPU；完成整个程序的循环控制、分支判断、CUBE/Vector等指令的地址和参数计算以及基本的算术运算等；
- **Unified Buffer or Scalar Buffer**: 晟腾310/910 Scalar Unit不能直接访问外面的DDR/HBM，需要预留UB的一部分(310)或者使用专门的Scalar Buffer(910)用作Scalar Unit的堆栈空间
- **GPR**：通用寄存器，目前包含32个通用寄存器
- **SPR**：专用寄存器，为了支持指令集一些指令的特殊需要，Davinci设计了许多专用寄存器，比如CoreID, BLOCKID, VA，STATUS, CTRL等寄存器

MTE/BIU和片上高速存储(Buffer)



- **BIU (Bus Interface Unit):** AICore 的“大门”，与总线交互的接口。AICore从外部（L2/DDR/HBM）读取、写入数据的出入口。负责把AICore的读写请求转换为总线上的请求并完成协议交互等工作。
- **MTE (Memory Transfer Unit):** 也被称作 LSU (Load Store Unit), 负责AICore内部数据在不同Buffer之间的读写管理，以及完成一些格式转换的操作，比如padding, 转置, Img2Col, 解压等
- **L1 Buffer**: AICore内最大的一块数据中转区(1MB)，可以用来暂存AICore需要反复使用的一些数据从而减少从总线读写；Img2col操作等MTE的数据格式转换功能需源数据必须位于L1 Buffer
- **L0A/L0B/L0C/UB/Scalar Buffer**: 前面已介绍

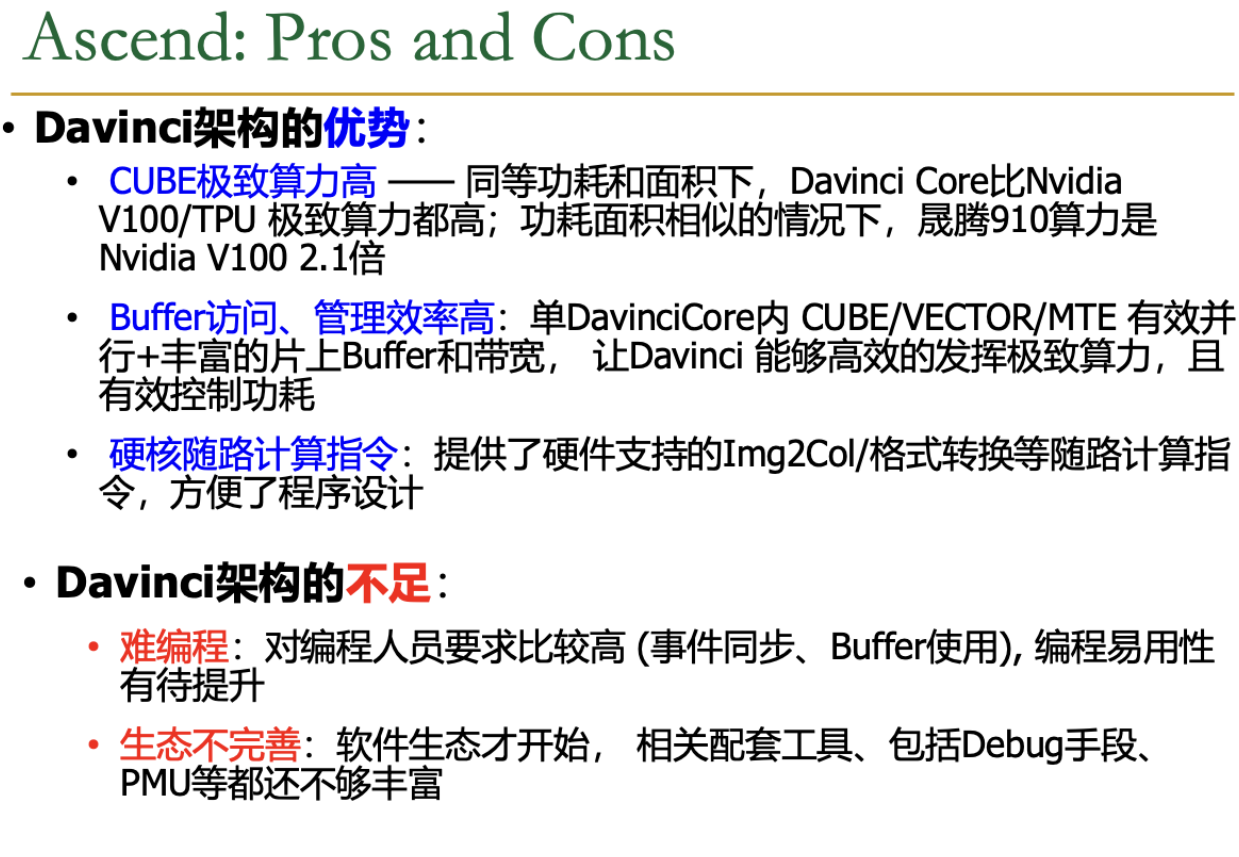
Event Sync: 用于控制不同队列指令(也叫做不同指令流水)之间的依赖和同步的模块

barrier()

set_flag.PIPE_dst.PIPE_src

wait_flag.PIPE_dst.PIPE_src

- 异步发起，同步跑。

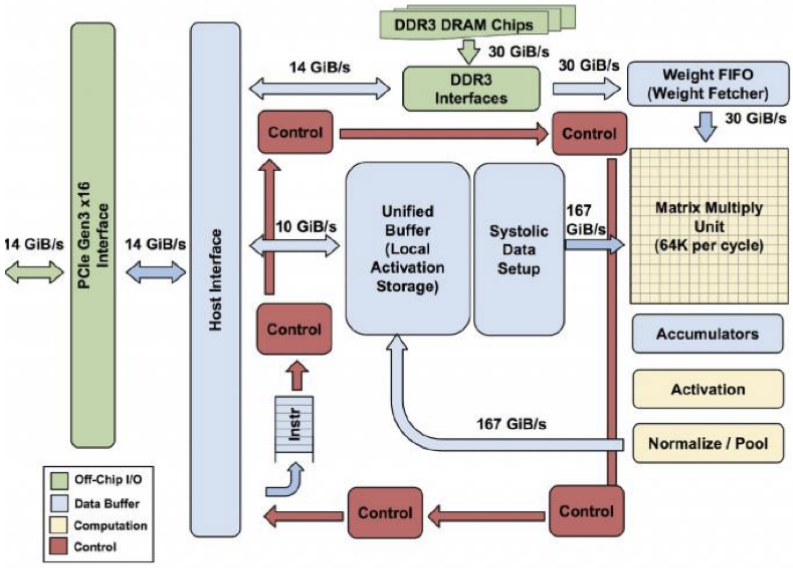


Google TPU

- TPU v1
  - Inference only
- TPU v2
  - Support Training
- TPU v3
  - Support Training
  - More Computing Power
- TPU v4
  - TPU4: for Training

- TPU4i: for Inference

TPU v1



- **Matrix Multiply Unit**
  - 256x256 MACs
  - Systolic Array
  - 24% area
- **Unified Buffer**
  - 24 MB
  - 29% area

- **TPU v1**
  - For inference, model is pre-stored in DDR3, and data is from the host via PCIe

Systolic Arrays:

1. 设计目标

- **简单规则化设计**：减少独特部件的数量，采用重复性高的模块（如处理单元阵列）。这能降低设计复杂度，提高可扩展性和制造效率。
- **高并发性**：通过并行处理提升性能，利用多个处理单元同时工作。
- **计算与I/O带宽平衡**：避免计算单元因等待数据而闲置，确保内存带宽足以喂饱计算资源。

---

2. 核心思想

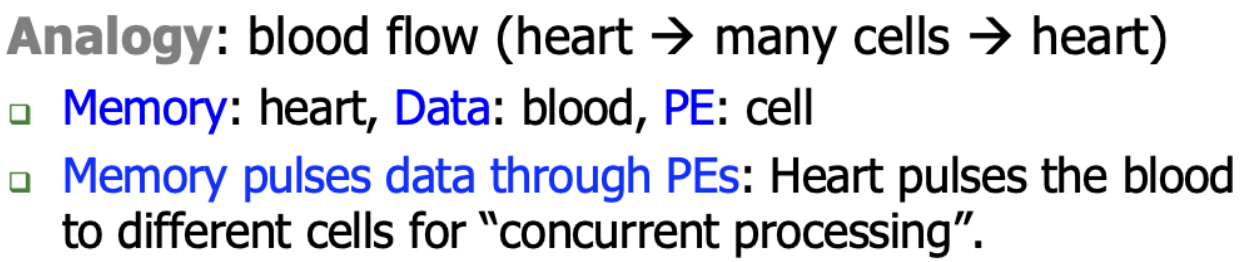
- **处理单元阵列（PE Array）**：
  用多个简单的处理单元（PE）组成规则网格（如脉动阵列或SIMD阵列），替代单一复杂PE。每个PE执行相同或类似操作，但处理不同数据。
- **数据流编排（Data Orchestration）**：
  数据在PE间按固定路径流动（如流水线或相邻传递），每经过一个PE就被逐步处理。这样，单个数据元素会被多个PE依次加工，**最大化数据复用**，减少内存访问次数。

---

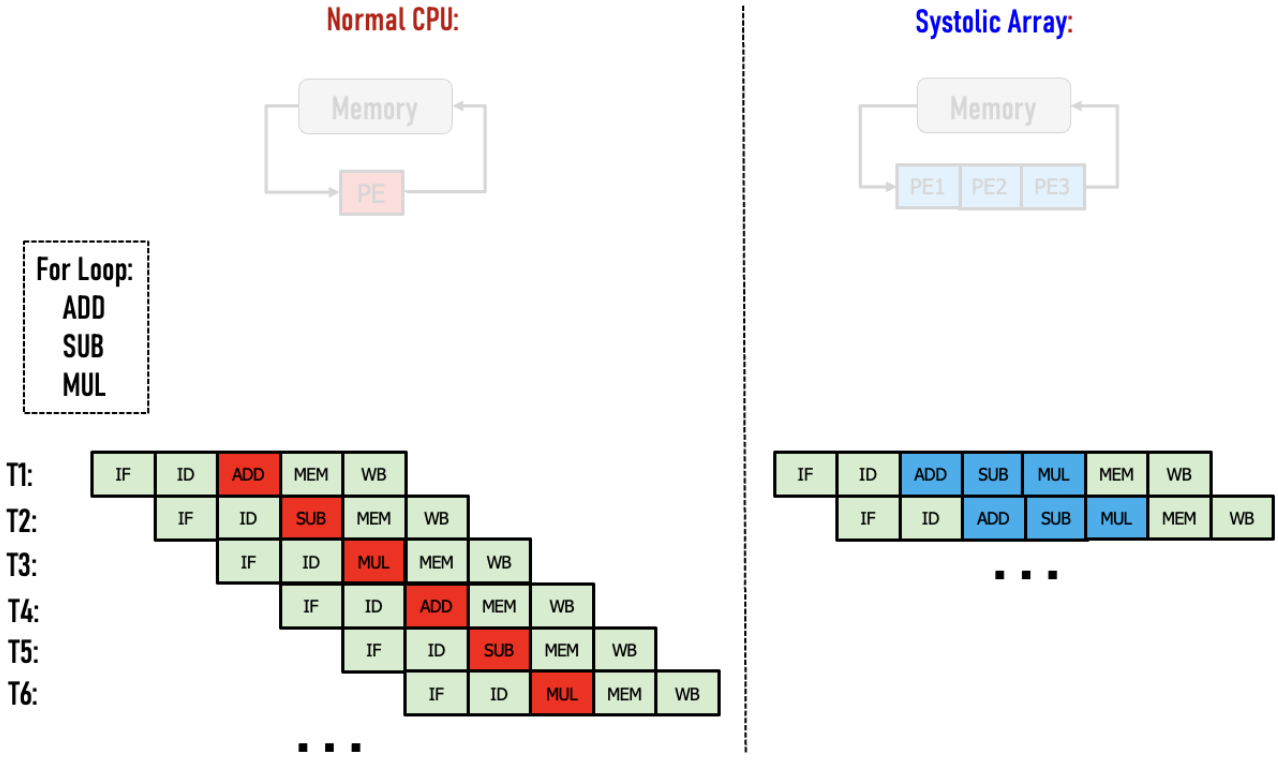
3. 关键优势

- **计算/内存比优化**：
  传统架构中，数据从内存加载后可能只经过少量计算就被写回，导致内存带宽成为瓶颈。而PE阵列让数据在计算单元间流动，**多次计算后才写回内存**，显著提升计算密度。
- **局部性优化**：
  数据在PE间直接传递（而非通过全局内存），减少高延迟的片外访问，适合矩阵乘法、卷积等具有数据重用模式的计算。

- 一个类比



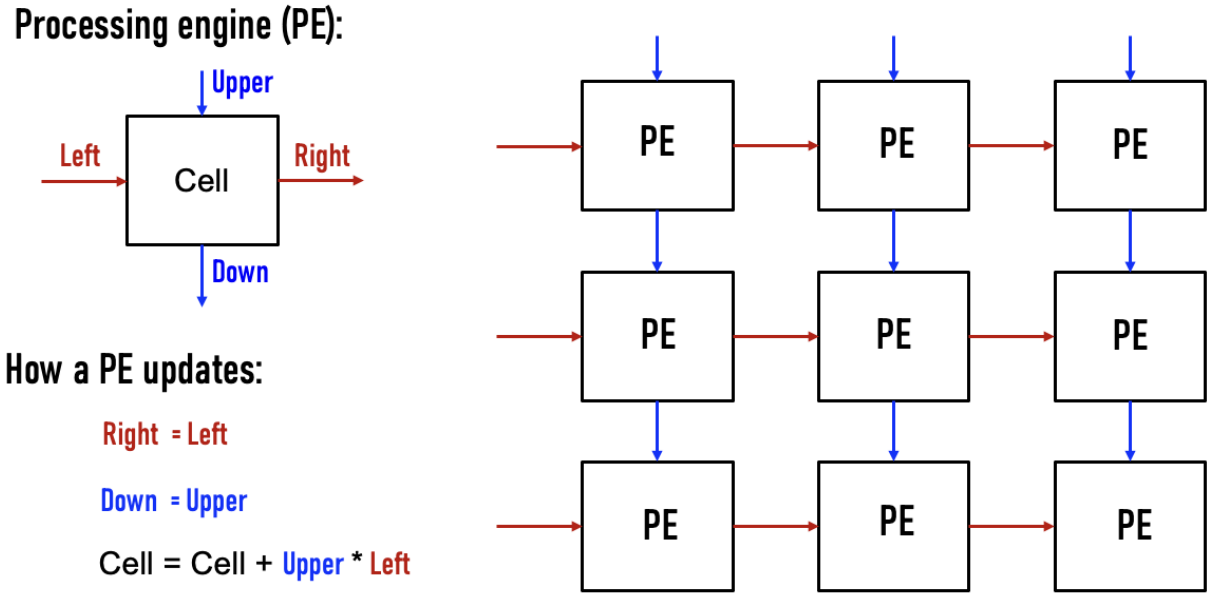
Systolic Arrays: Benefit (Intuition)

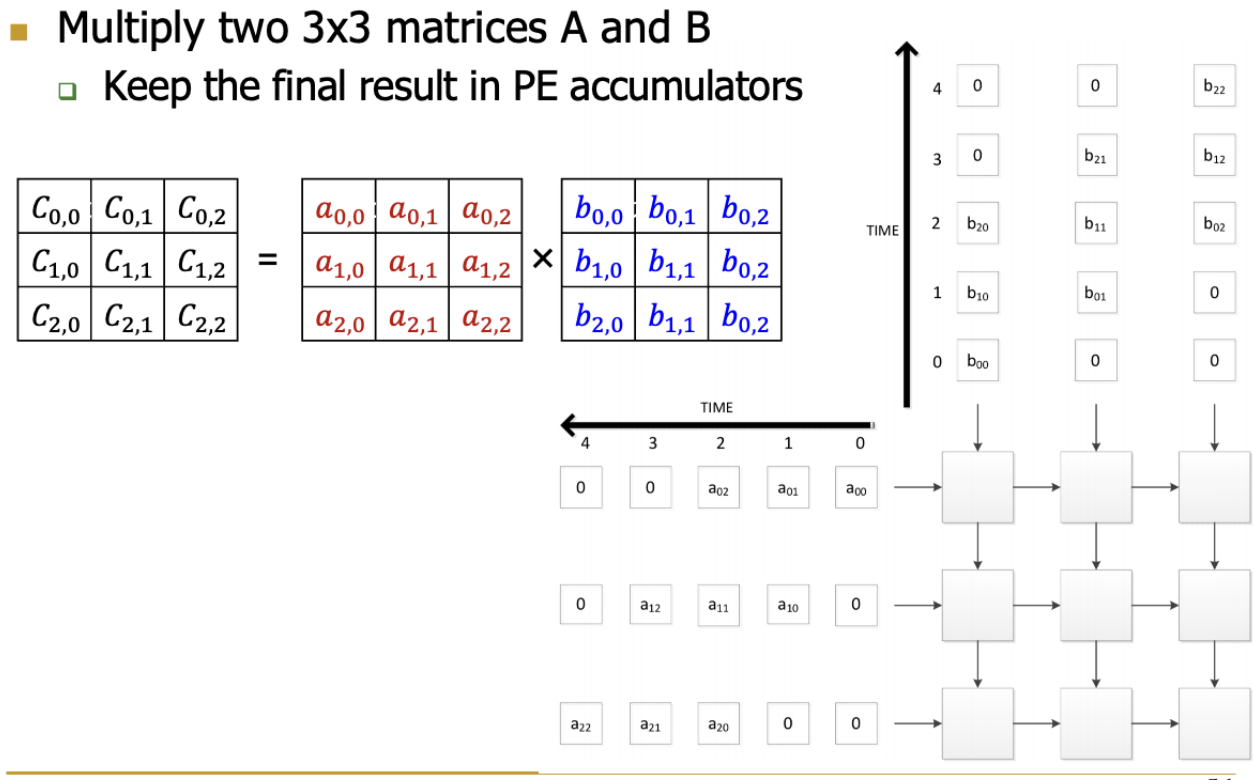


- 具体更新模式

Systolic Arrays in AI Accelerator

- Systolic array can be multi-dimensional
  - The most popular one used by AI accelerator is two-dimensional.





具体看第12章ppt的第57-64页。

- TPU v1-v4的变化看第12章ppt的第65-74页。

为啥AI加速器只要集中在推理(Inference)而不是训练(Training)?

AI模型训练中，内存带宽往往是整体性能的瓶颈，而AI加速器并不能很明显地提高内存带宽的利用效率。

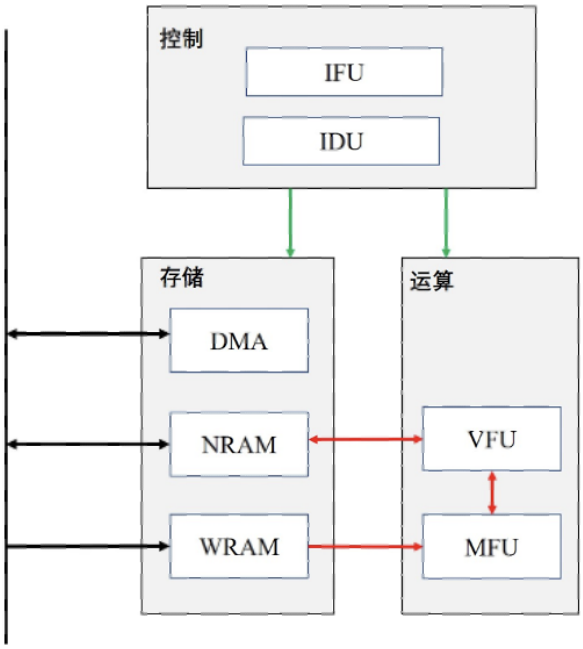
寒武纪Cambridge（DLP）

- 分为单核和多核

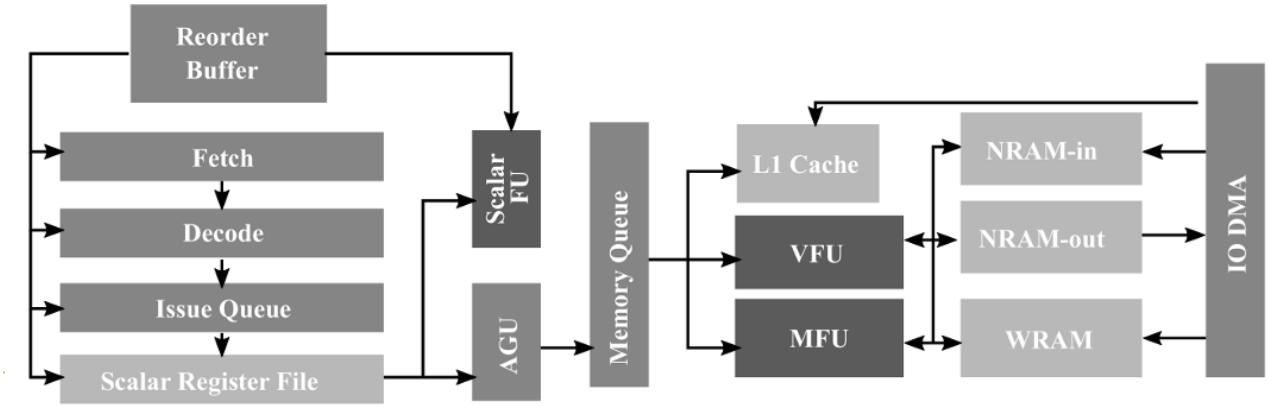
单核 DLP-S

- 结构总览

  - Control Module
    - IFU (Instruction Fetch Unit)
    - IDU (Instruction Decode Unit)
  - Compute Unit
    - VFU (Vector Function Unit)
    - MFU (Matrix Function Unit)
  - SRAM Unit
    - WRAM (Weight RAM)
    - NRAM (Neuron RAM)
    - DMA (Direct Memory Access)



- 具体结构

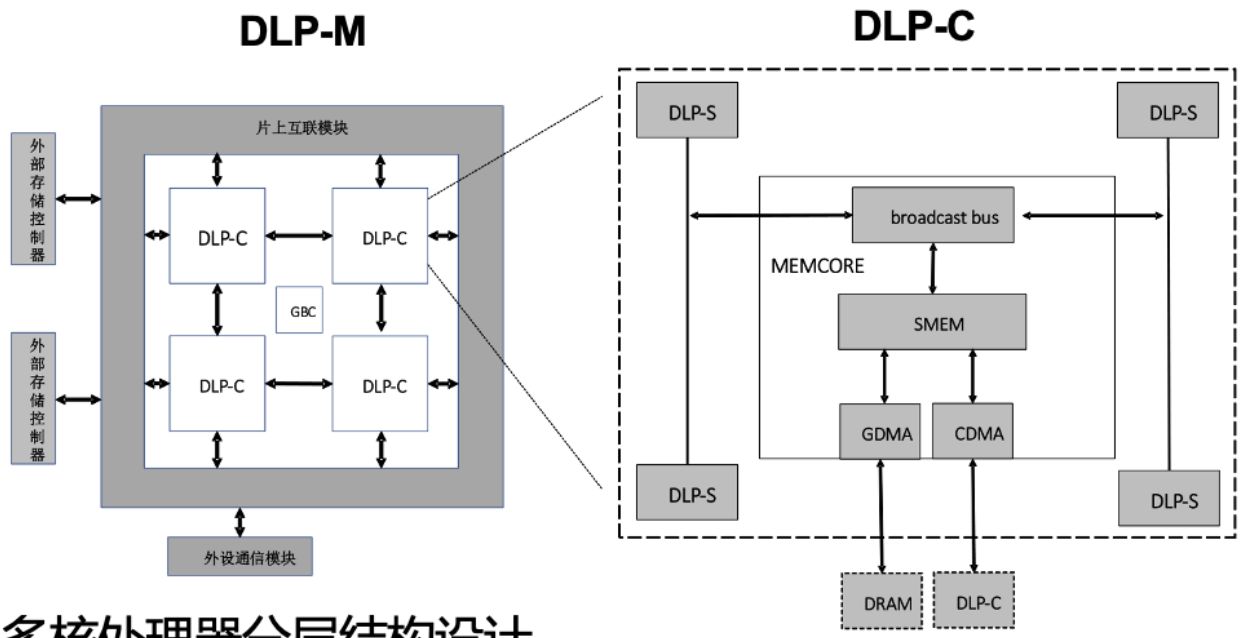


  - 左侧：控制模块
  - 中间：运算模块
  - 右侧：储存模块
- 具体工作流见底13章ppt对第27-33页。

- 指令集ISA

指令类型		例子	操作对象
控制指令		跳转(JUMP)，条件分支(CB)	寄存器(标量值)，立即数
数据转移指令	矩阵(Matrix)	矩阵 取(MLOAD)/存(MTORE)/移动(MMOVE)	寄存器(矩阵 地址/大小，标量值)，立即数
	向量(Vector)	向量 取(VLOAD)/存(VSTORE)/移动(VMOVE)	寄存器(向量 地址/大小，标量值)，立即数
	标量(Scalar)	标量 取(SLOAD)/存(SSTORE)/移动(SMOVE)	寄存器(标量值)，立即数
计算指令	矩阵(Matrix)	矩阵乘向量(MMV)，向量乘矩阵(VMM)，矩阵乘标量(MMS)，外积(OP)，矩阵相加(MAM)，矩阵相减(MSM)	寄存器(矩阵/向量 地址/大小，标量值)
	向量(Vector)	向量基本运算(加(VAV)、减(VSV)、乘(VMV)、除(VDV))，向量超越函数(指数(VEXP)、对数(VLOG))，内积(IP)，随机向量(RV)，向量最大值(VMAX)，向量最小值(VMIN)	寄存器(向量 地址/大小，标量值)
	标量(Scalar)	标量基本运算，标量超越函数	寄存器(向量 地址/大小， 标量)
逻辑运算指令	向量(Vector)	向量比较(大于(VGT),等于(VE)),向量逻辑操作(与(VAND)，或(VOR)，取反(VNOT))，向量最值归约(VGTM)	寄存器(向量 地址/大小，标量)
	标量(Scalar)	标量比较，标量逻辑运算	寄存器(标量)，立即数

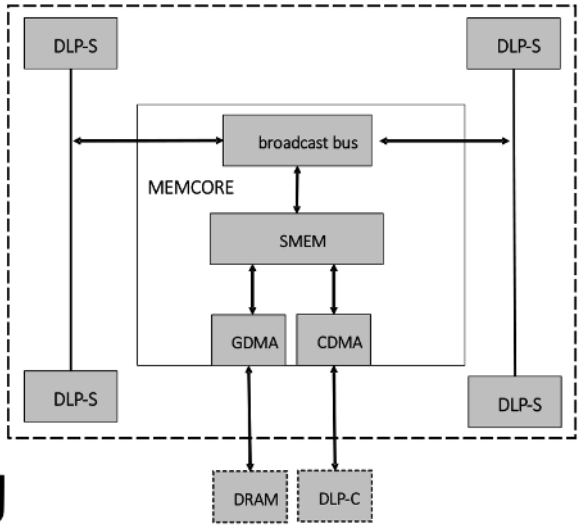
多核 DLP-M



- 多核处理器分层结构设计
  - 一个DLP-M由多个DLP-C构成
  - 一个DLP-C由多个DLP-S构成

为什么需要进行分层结构设计？ 减少NoC的负载核开销

DLP-C（cluster）



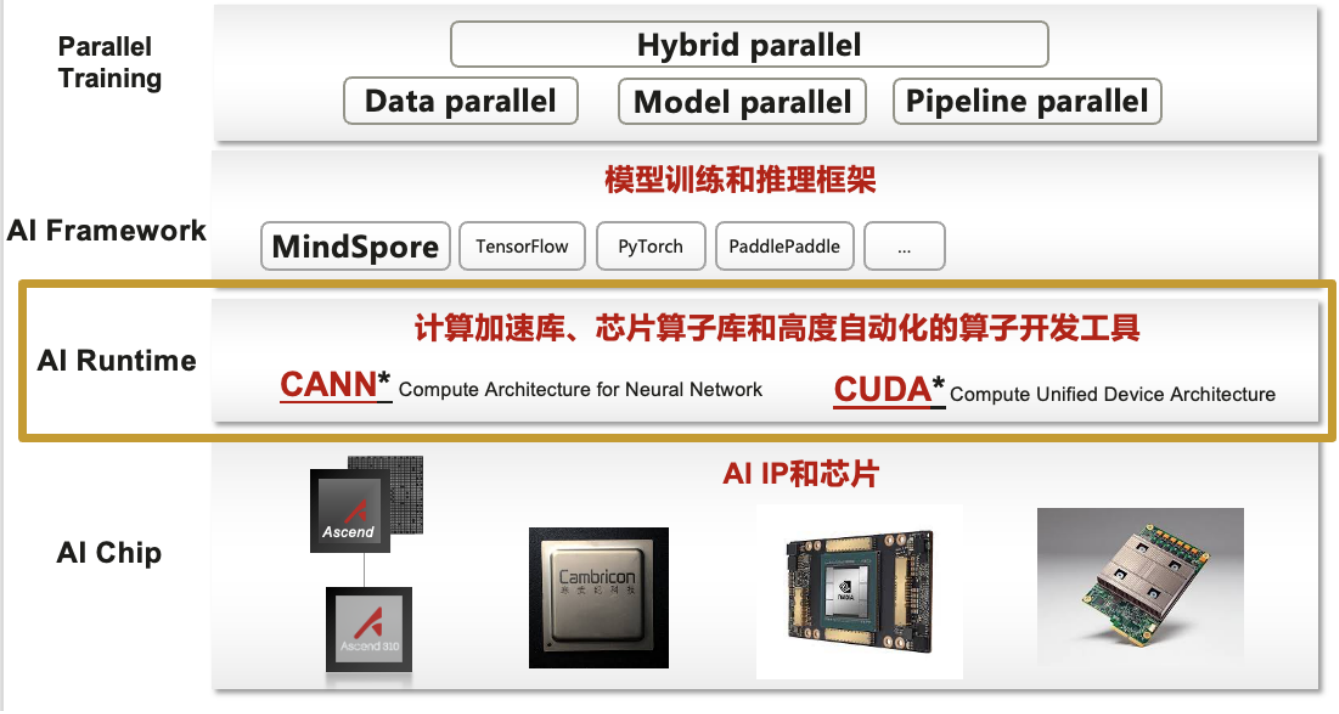
- **DLP-C整体架构**
  - 四个DLP-S
  - 存储核MEMCORE（Memory Core）
    - **存储SMEM**：DLP-S共享数据
    - **通信**：
      - GDMA: DLP-C与片外DRAM
      - CDMA: DLP-C之间，多个DLP-S之间

选择同构还是异构?

- Homogeneous Architecture (Huawei and Nvidia)
- Heterogeneous architecture (Cambricon)
- 同构的好处
- 异构的好处：效率高，但是编程难度极大，因此不如同构

13. Runtime + Framework

RunTime



- 连接芯片和框架的算子开发工具。

- 对底层的计算资源进行抽象。

- The motivation of NN operator library:
  - 1, NN tasks are composed of NN operators
  - 2, AI chips are difficult to program, we cannot let AI programmer directly program AI chips

- The goal of NN operator library:
  - Performance + Usability: provide high-performance, well-documented NN library for the upper AI framework such as MindSpore.

CANN：向下使能处理器并行加速，向上使能高效开发

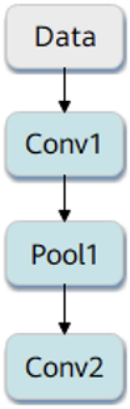
Ascend NN Operator Library

昇腾算子库包含了丰富的高性能算子：

- **NN（Neural Network）算子库**：覆盖了包括TensorFlow、Pytorch、MindSpore、ONNX等框架的常用深度学习算法的计算类型，在算子库中占有最大比重。
- **BLAS（Basic Linear Algebra Subprograms）算子库**：基础线性代数程序集，是进行向量和矩阵等基本线性代数操作的数值库。
- **DVPP（Digital Video Pre-Processor）算子库**：提供高性能的视频编解码、图片编解码、图像裁剪缩放等预处理能力。
- **AIPP（AI Pre-Processing）算子库**：主要实现改变图像尺寸、色域转换（转换图像格式）、减均值/乘系数（图像归一化），并与模型推理过程融合，以满足推理输入要求。
- **HCCL（Huawei Collective Communication Library）算子库**：提供单机多卡以及多机多卡间的Broadcast，allreduce，reducescatter，allgather等集合通信功能，在分布式训练中提供高效的数据传输能力。

■ 算子名称（Name）

算子的名称，用于标志网络中的某个算子，同一网络中算子的名称需要保持唯一。如右图所示Conv1，Pool1，Conv2都是此网络中的算子名称，其中Conv1与Conv2算子的类型为Convolution，表示分别做一次卷积运算。

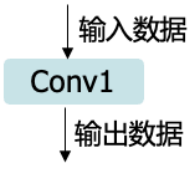


■ 算子类型（Type）

网络中每一个算子根据算子类型进行算子实现的匹配，相同类型的算子的实现逻辑相同。在一个网络中同一类型的算子可能存在多个，例如右图中名称为Conv1的算子与Conv2算子的类型都为Convolution。

■ 数据容器（Tensor）

张量（Tensor）是承载算子数据的容器。如右图所示，算子在网络中执行时，输入数据是一个tensor，算子执行完后，输出数据也是一个tensor。



算子基本概念-Tensor

张量(Tensor)是存储算子输入数据与输出数据的容器，

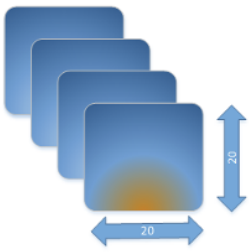
而张量描述符(TensorDesc)是对输入数据与输出数据的描述，

张量描述符的数据结构包含如下属性：

属性	定义
名称（name）	用于对Tensor进行索引，不同Tensor的name需保持唯一。
形状（shape）	Tensor的形状，比如（10,）或者（1024, 1024）或者（2, 3, 4）等。 形式：(i1, i2,...in)，其中i1到in均为正整数
数据类型（dtype）	指定Tensor对象的数据类型。 例如：float16, float32, int8, int16, int32, uint8, uint16, bool等。 不同计算操作支持的数据类型不同。
数据排布格式（format）	数据的物理排布格式，定义了解读数据的维度。

算子基本概念-Tensor

- **数据排布格式(format)：**
- 在深度学习领域，多维数据通过多维数组存储，比如卷积神经网络的特征图（Feature Map）通常用四维数组保存，即4D格式：
  - N：Batch数量，例如图像的数目。
  - H：Height，特征图高度，即垂直高度方向的像素个数。
  - W：Width，特征图宽度，即水平宽度方向的像素个数。
  - C：Channels，特征图通道，例如彩色RGB图像的Channels为3。



- 不同深度学习框架会按照不同的顺序存储特征图数据：
  - Caffe的排列顺序为[Batch, Channels, Height, Width]即NCHW
  - TensorFlow的排列顺序为[Batch, Height, Width, Channels] 即NHWC

- 注意：tensor储存在device memory

CANN开发方式

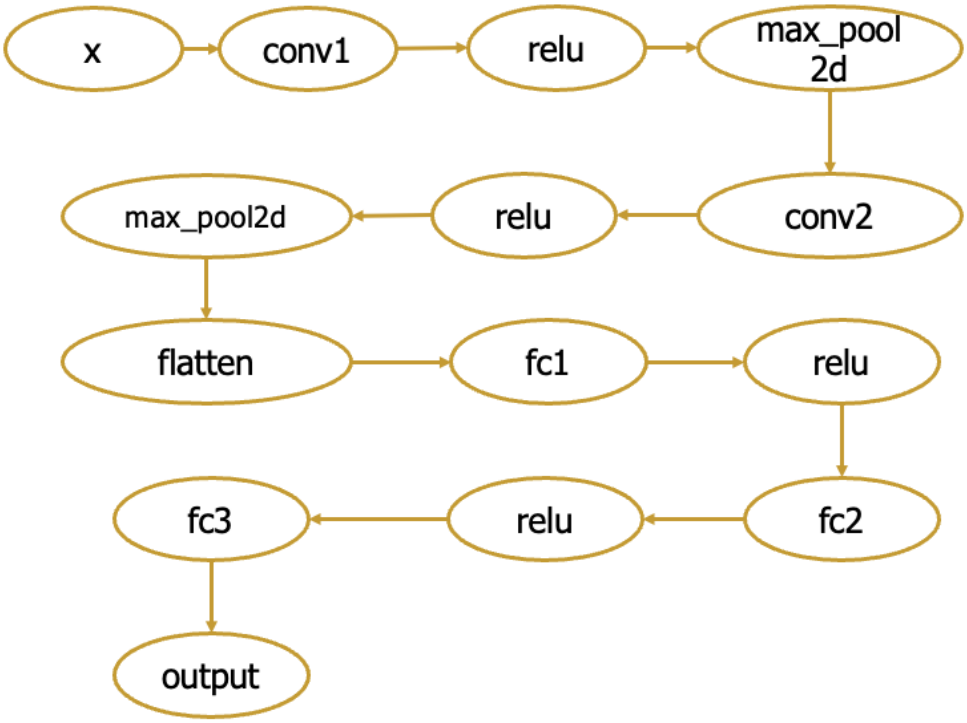
参数	TBE DSL方式	TIK方式	AI CPU方式
语言	Python	Python	C++
计算单元	AI Core	AI Core	AI CPU
运用场景	常用于各种算术逻辑简单向量运算，或内置支持的矩阵运算及池化运算	适用各类算子的开发，对于无法通过lambda表达描述的复杂计算场景也有很好的支持，例如排序类操作	某些场景下，无法通过AI Core实现的自定义算子，或者需要临时快速打通网络的场景下使用
入门难度	较低	较高	中等
适用人群	入门用户，需要了解NN、TBE DSL相关知识	高级用户，需要了解NN，深入理解昇腾AI处理器架构、指令集、数据搬运等相关知识	具备C++程序开发能力，对机器学习、深度学习、AI CPU开发流程有一定的了解
特点	TBE DSL接口已高度封装，用户仅需要使用DSL接口完成计算过程的表达，后续的Schedule创建、优化及编译都可通过已有接口一键式完成	入门难度高，程序员直接使用TIK提供的API完成计算过程的描述及Schedule过程，需要手工控制数据搬运的参数和Schedule。用户无须关注Buffer地址的分配及数据同步处理，由TIK工具进行管理	开发的流程和DSL都是类似的，不需要了解AI Core的内部架构设计，入门较快
不足	某些场景下性能可能较低，复杂算子逻辑无法支持表达	需要开发者手工控制数据搬运的参数和Schedule过程。	无封装的计算接口，计算过程相对繁琐，另外AI CPU性能较低。

- tik处理细碎复杂的小问题
- dsl处理大型运算

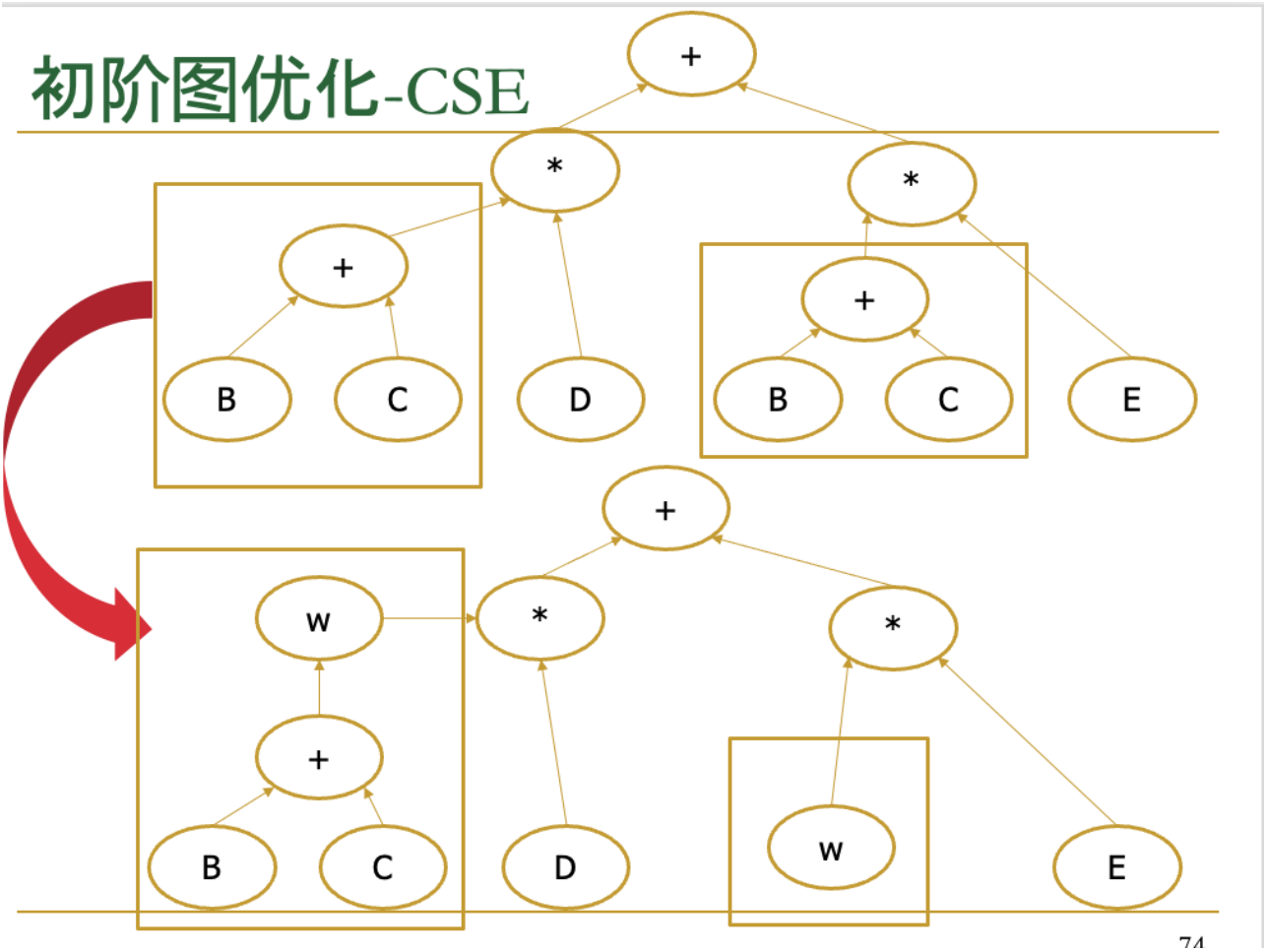
计算图引擎GE

Lenet5

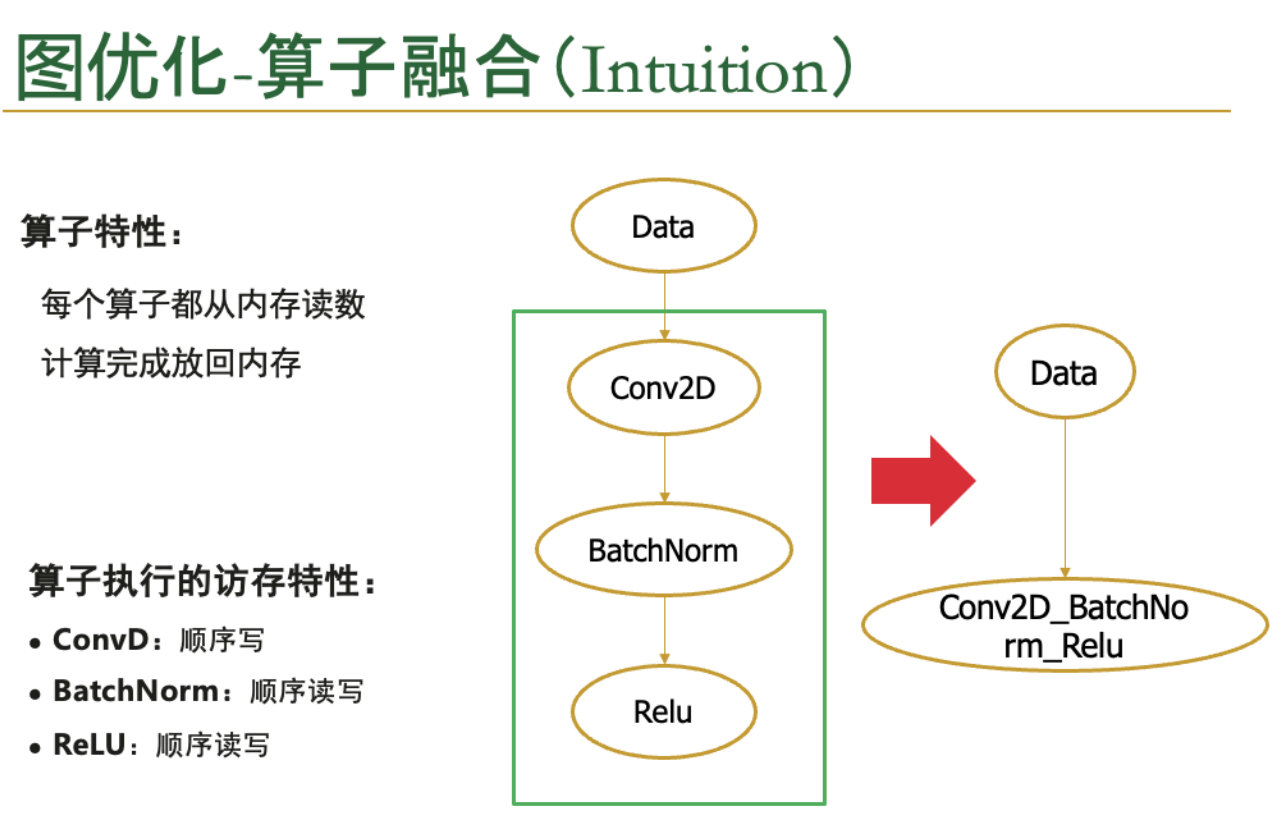
```
def construct(self, x):
    # 使用定义好的运算构建前向网络
    x = self.conv1(x)
    x = self.relu(x)
    x = self.max_pool2d(x)
    x = self.conv2(x)
    x = self.relu(x)
    x = self.max_pool2d(x)
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    x = self.relu(x)
    x = self.fc3(x)
    return x
```



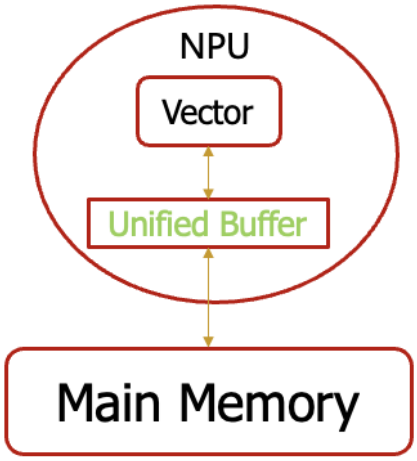
- CSE（Common-Subexpression Elimination），公共子表达式消除。简单而言就是将相同输入的表达式进行消除，复用计算结果。


初阶图优化-CSE
+
*
*
+
+
B
C
D
B
C
E
+
w
*
*
+
B
C
D
w
E
74

- 算子融合 -> 减少访存


图优化-算子融合（Intuition）
算子特性：
每个算子都从内存读数
计算完成放回内存
算子执行的访存特性：
ConvD：顺序写
BatchNorm：顺序读写
ReLU：顺序读写
Data
Conv2D
BatchNorm
Relu
Data
Conv2D_BatchNo
rm_Relu

图优化-算子融合（UB融合）



以一个简单的Vector算子计算为例，其计算过程通常包含以下几个步骤：

1. 计算任务和数据在片上的上下文切换
2. 新的算子所需数据从主存搬运到Unified Buffer（以下简称UB）
3. Vector读取UB中的数据进行计算，并将结果存回UB
4. 计算结果从UB搬出到主存

Key Idea of UB融合：泛指片上缓存级别的融合，即数据搬进芯片后，下发的算子计算任务是由多个小算子融合而成的大算子。

UB融合具体计算步骤：

1. 计算任务和数据在片上的上下文切换
2. 新的算子所需数据从主存搬运到Unified Buffer（以下简称UB）
3. Vector读取UB中的数据进行算子1计算，并将结果存回UB
4. Vector读取UB中的数据进行算子2计算，并将结果存回UB
5. Vector读取UB中的数据进行算子3计算，并将结果存回UB
6. 计算结果从UB搬出到主存

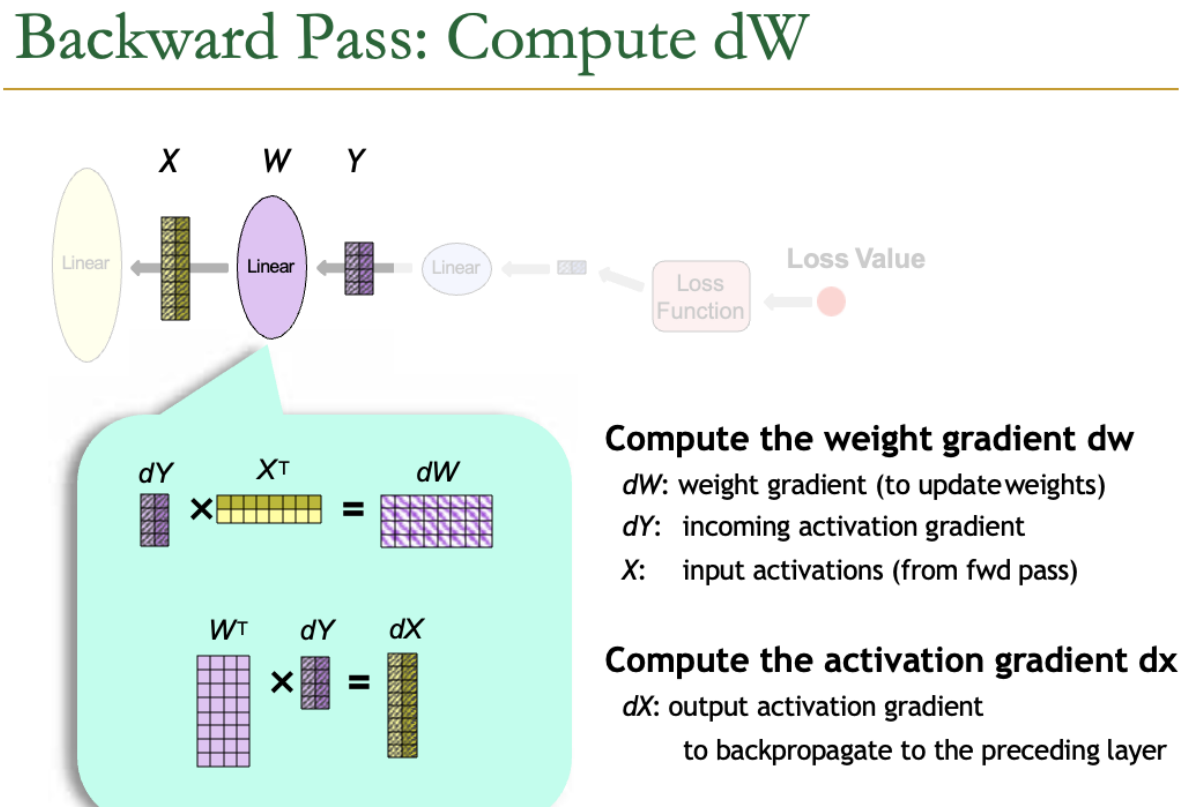
Framework

- Reasons:
  - AI algorithms are gaining great attention.
  - More and more companies and programmers are using them.
- 有必要将算法中的常用操作封装成组件提供给程序员，以提高深度学习算法开发效率和性能。
- Mindspore 优势：
  - 自动并行：整图切分，感知集群拓扑，实现通信开销最小，融合数据并行与模型并行；
  - 二阶优化：利用二阶计算修正梯度更新方向，找到训练梯度最优下降路径，从而加速训练收敛过程；

- 动静态图结合：统一自动微分引擎支持动静态图，一行代码完成模式切换，兼顾模型开发和执行效率；
- AI+科学计算，场景应用创新，拓展MindSpore的边界

14.Parallel Training

- 4 Components of AI System(model training)
  - Storage
  - Computing
  - Networking
  - Compiling

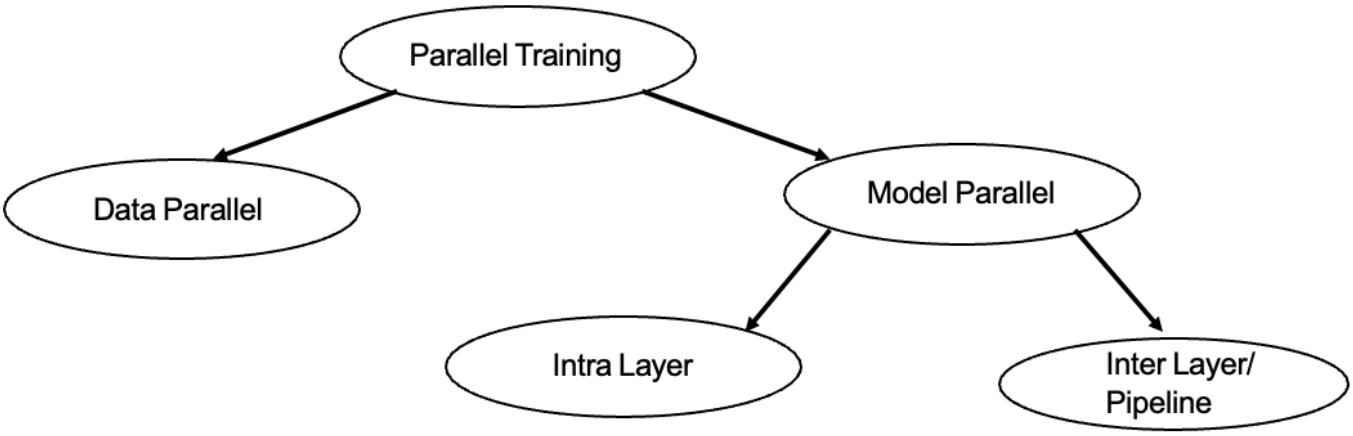


- 反向传递梯度时：从dY传递到dX需要两次矩阵计算；但是在正向传递时，只有Y=X*W这一次矩阵运算。

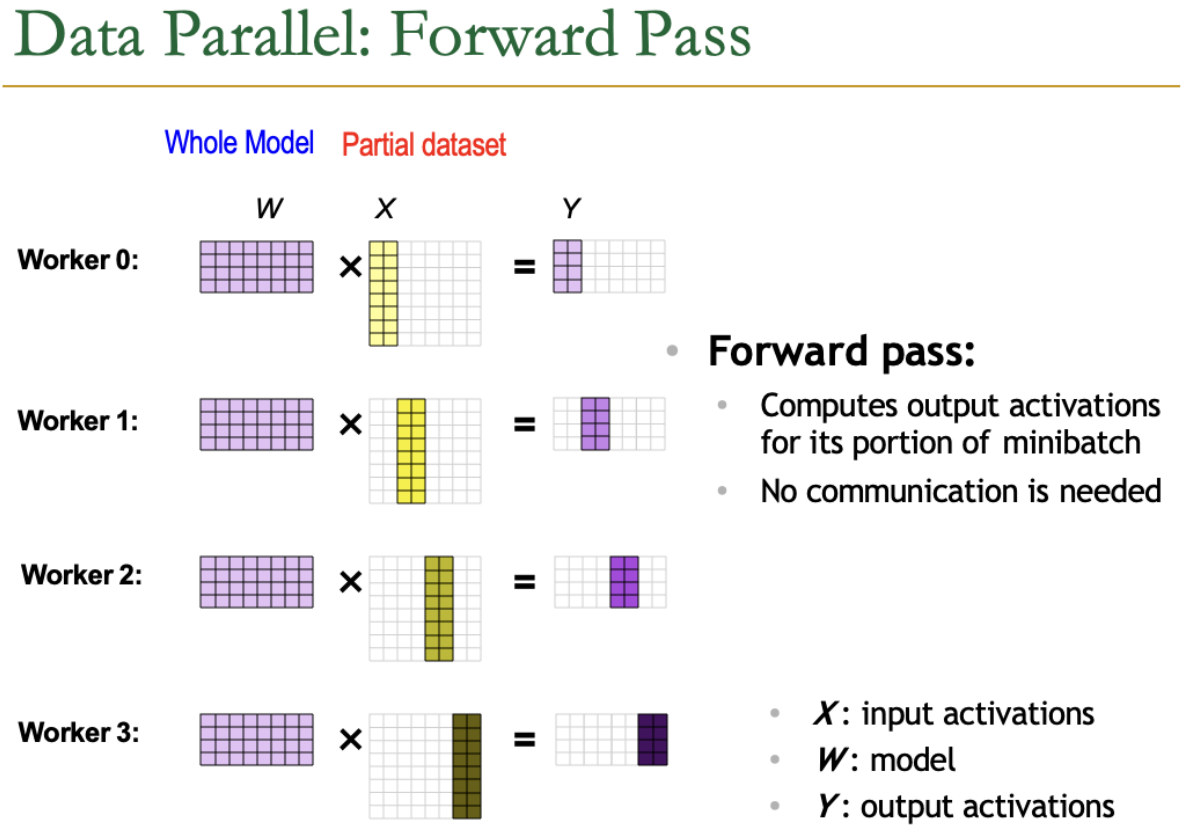
Why Distributed Training?

- Challenge from Model Side: Larger models
- Challenge from Dataset Side: Larger datasets
- Challenge from System Side:

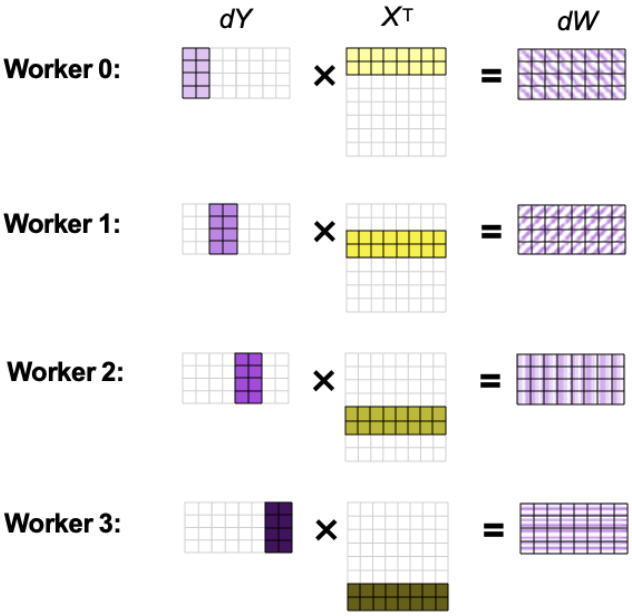
Data Parallelism



- Each worker:
  - Model: has a copy of the entire neural network model
  - Dataset: responsible for compute of a portion of data (training minibatch)



Data Parallel: Backward Pass

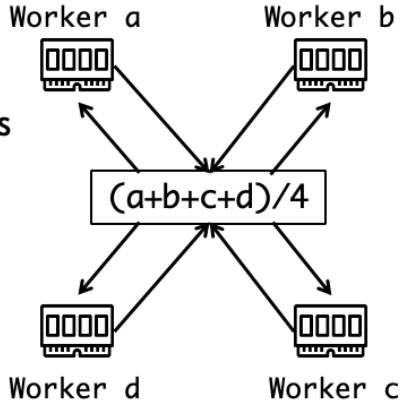


- **Backward pass:**
  - Computes activation gradients for its portion of minibatch
  - Computes contribution to the weight gradient based on its portion of minibatch
    - All workers' contributions must be summed before weight update

- X: input activations
- W: model
- Y: output activations

Data Parallel Training: Weight Update

- **Weight update:**
  - **1, Each of N workers accumulates gradients:**
    - Summing $1/N$ gradients collected from $(N - 1)$ peers
  - **2, Each worker updates its model:**
    - Each worker updates its copy of the model with combined gradients from all 4 workers



1. 每个 worker 累积梯度

- **背景**：在数据并行训练中，有 **N 个 worker**（通常是 GPU 或计算节点），每个 worker 持有模型的完整副本，并处理不同的数据子集（数据分片）。
- **梯度计算**：每个 worker 独立计算其本地数据的梯度（即损失函数对模型参数的导数）。
- **梯度聚合**：为了保持一致性，所有 worker 需要共享梯度，使得每个 worker 最终得到的是 **全局平均梯度**（即所有 worker 梯度的平均值）。
  - 具体操作：每个 worker 将自己的梯度分成 **1/N** 份，然后将其中 **(N-1) 份**发送给其他 worker，同时从其他 worker 接收对应的梯度分片。
  - **结果**：每个 worker 最终会收集到来自所有其他 worker 的梯度分片，并将它们相加，得到全局梯度的平均值。

Ring AllReduce 算法

1. 拓扑结构（1D Torus/Ring）

- **环形拓扑**：所有工作节点（worker）排列成一个逻辑环，每个节点只与左右两个相邻节点直接通信。
- **通用性**：只要网络拓扑包含一个环形结构（即使物理连接更复杂），就可以应用此算法。

---

2. 通信模式

- **邻居通信**：每个步骤中，节点只向一个邻居发送数据，并从另一个邻居接收数据（例如，顺时针发送，逆时针接收）。
- **分块传输**：数据被均分为N份（N=节点数），每次只传输1/N的数据量，减少单次通信的带宽压力。

---

3. 步骤与同步

- **总步骤数**：`2(N−1)`步。分为两个阶段：
  - **Scatter-Reduce**（N-1步）：逐步聚合部分结果（如求和）。
  - **Allgather**（N-1步）：将最终结果广播到所有节点。
- **同步要求**：每一步必须等待所有节点完成通信后才能继续，因此需要严格的同步（共`2(N−1)`次同步）。

- 第14章ppt的p39-p48
  - **Strong scaling (increase the number of workers, keep minibatch size constant)**
    - Certain layers require minimum minibatch sizes to properly operate
      - Example: batch normalization (BN) generally requires 16+ samples
    - Maybe lower GPU utilization
  - **Weak scaling (increase the number of workers, increase minibatch size)**
    - Training networks with large minibatches requires hyper-parameter adjustment
      - Learning rate schedule, BN decay, …
      - Example: R50 (SGD up to bs=16K, LARS above 16K, …)
    - Often increase the amount of work required to reach the same model accuracy

5. 直观对比表

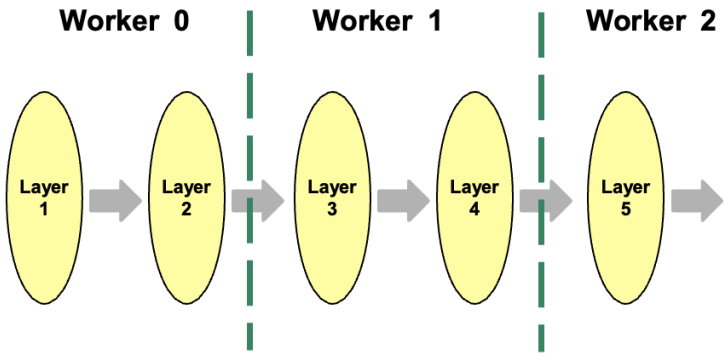
维度	Strong Scaling	Weak Scaling
问题规模	固定	随处理器数量线性增加
目标	缩短计算时间	扩大问题规模，保持时间不变
理想加速	线性加速比（时间减半）	恒定效率（时间不变）
瓶颈	通信开销、串行部分	内存带宽、全局通信

Model Parallelism

Model Parallel Training

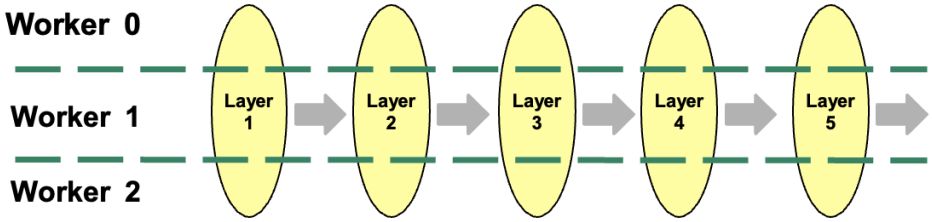
Inter-layer Parallel (aka Pipeline Parallel):

A worker is responsible for its portion of the layers



Intra-layer Parallel

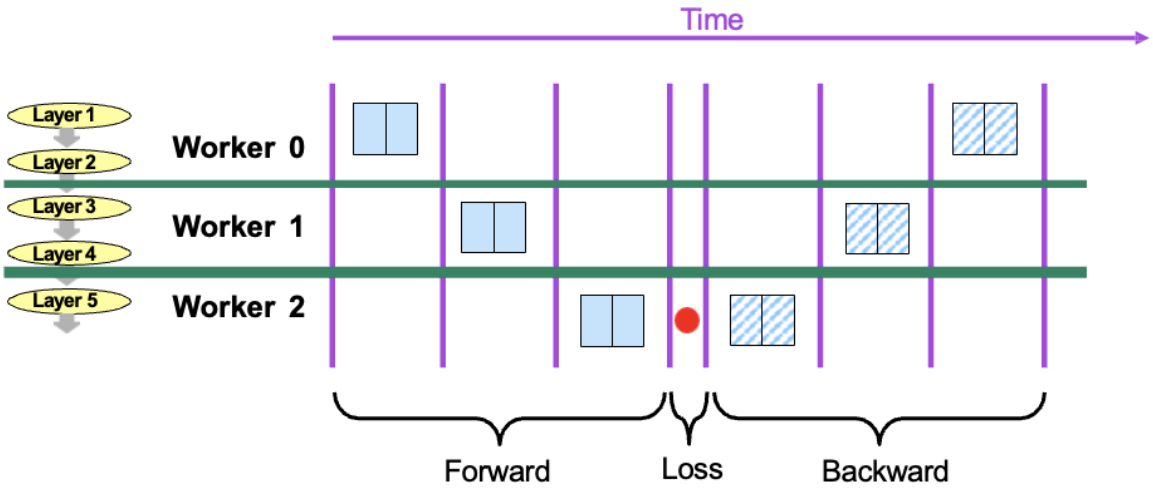
A worker is responsible for its portion of each layer



Inter-layer / Pipeline

- 存在bubble

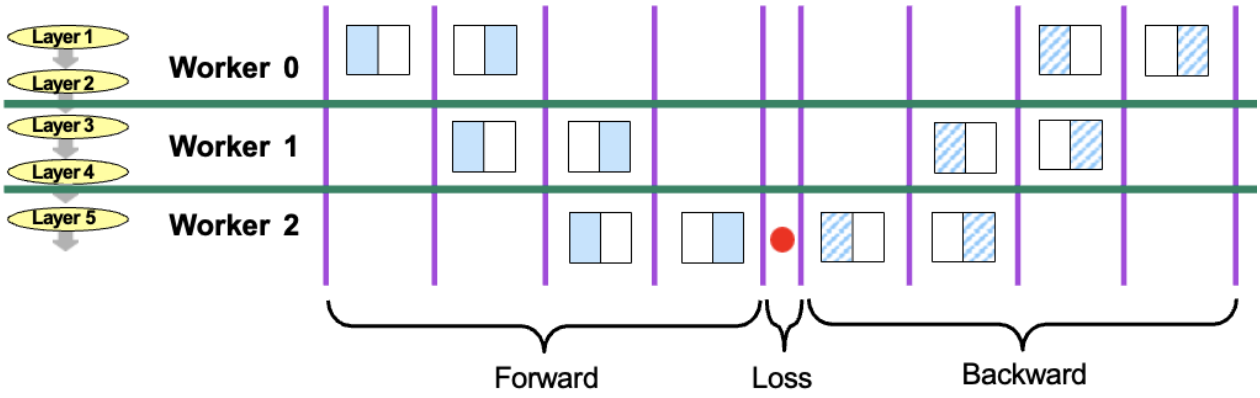
Pipeline Parallel Training



- **Idle bubbles:**
  - 67%: 12/18 step-slots
- **For *N* workers:**
  - $(N - 1)/N$ idle slots

- GPipe

Pipeline Parallel Training: GPipe



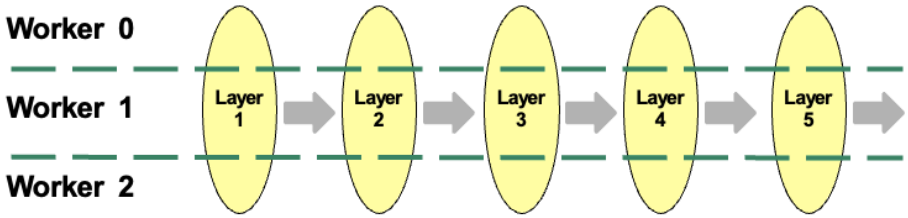
- ***N* workers, *K* subminibatches:**
  - $2(N + K - 1)$ steps for fwd/bwd
  - Total step-slots: $2N(N + K - 1)$
  - Idle step-slots: $2N(N - 1)$
  - Fraction of idle slots: $(N - 1)/(N + K - 1)$
- **As *N* grows:**
  - $K = N \rightarrow$ 50% idle slots
  - $K = 4N \rightarrow$ 20% idle slots

Pipeline Parallel: Challenges

- **Load balancing workload across workers is difficult**
  - Different layers of a network can take different amounts of time
  - Leads to even busy slots for other workers idling for portions of time
- **Lots of computation to hide communication**
- **Idle slots reduce scaling efficiency**
  - Many subminibatches help with this, but run into the same problems as strong-scaling of data-parallel.

Intra-layer

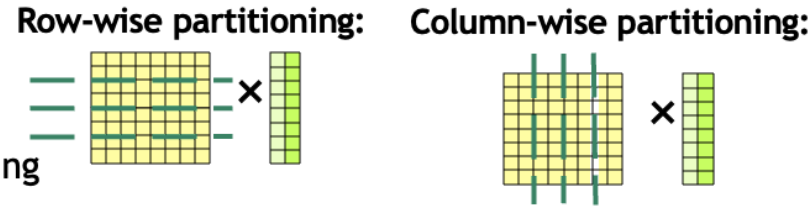
- 两种层内分割方法



- **Partition a given layer's weights among the workers**
- **Addresses some of the Pipeline Parallel challenges**
  - Idle slots, load imbalance

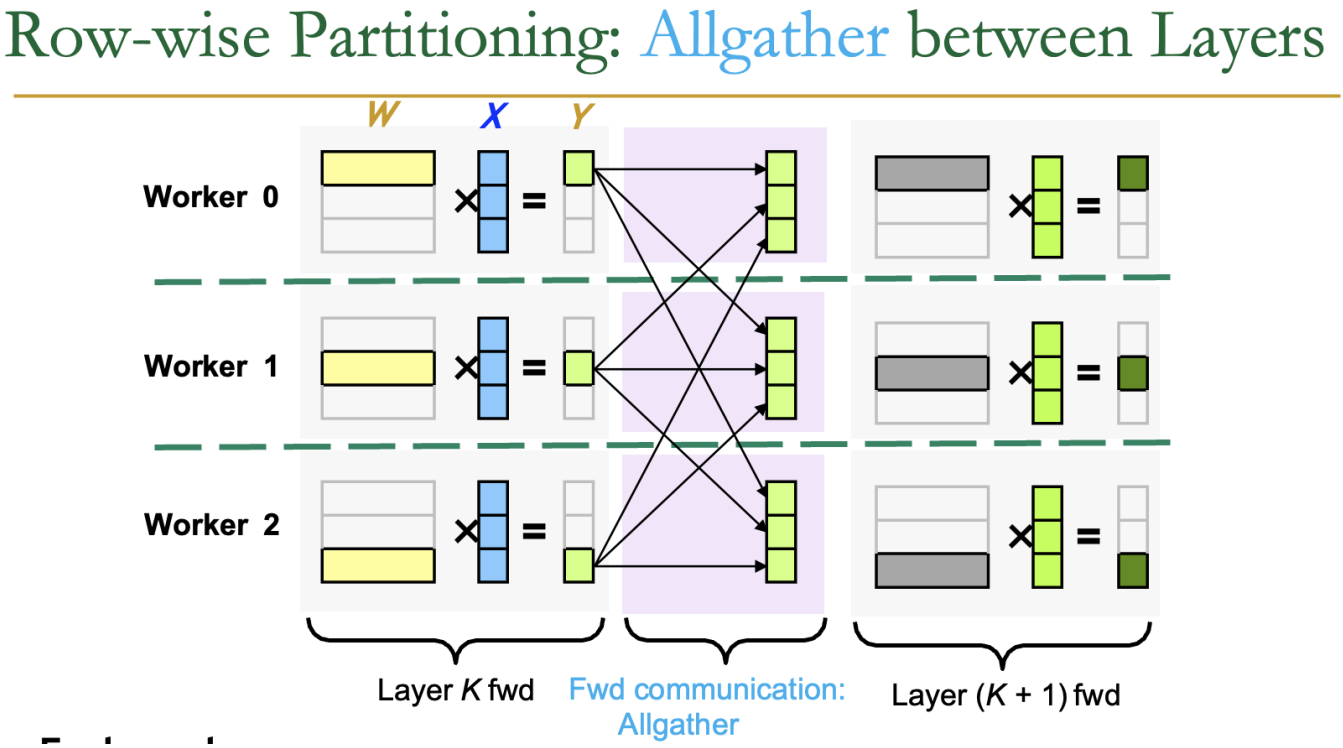
Two variants:

- Row-wise partitioning
- Column-wise partitioning

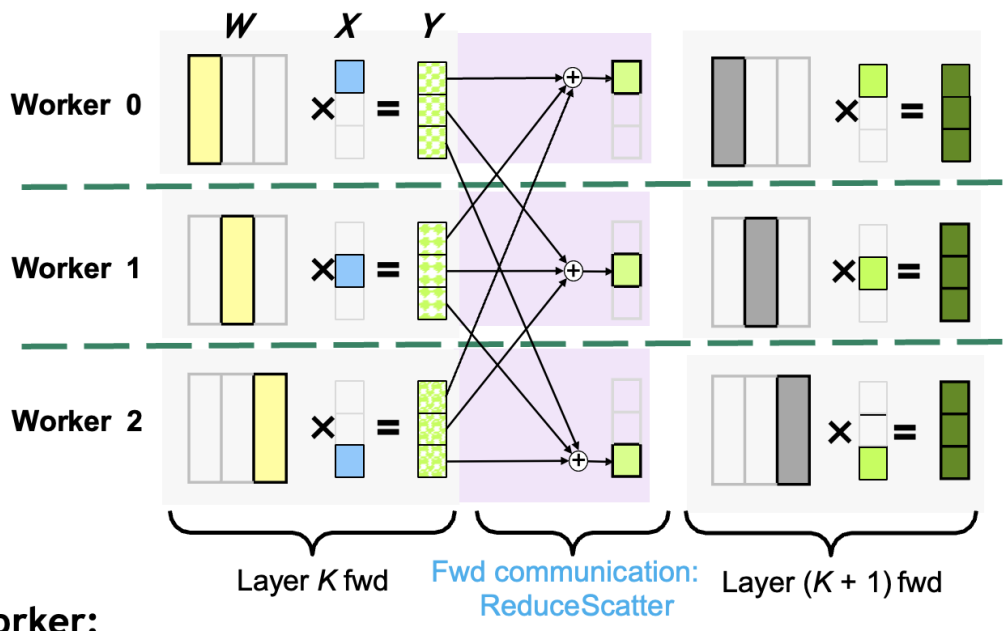


- 不同分割方法有不同的通信方式!

row-wise 的通信方式： allgather



column-wise 的通信方式： reducescatter

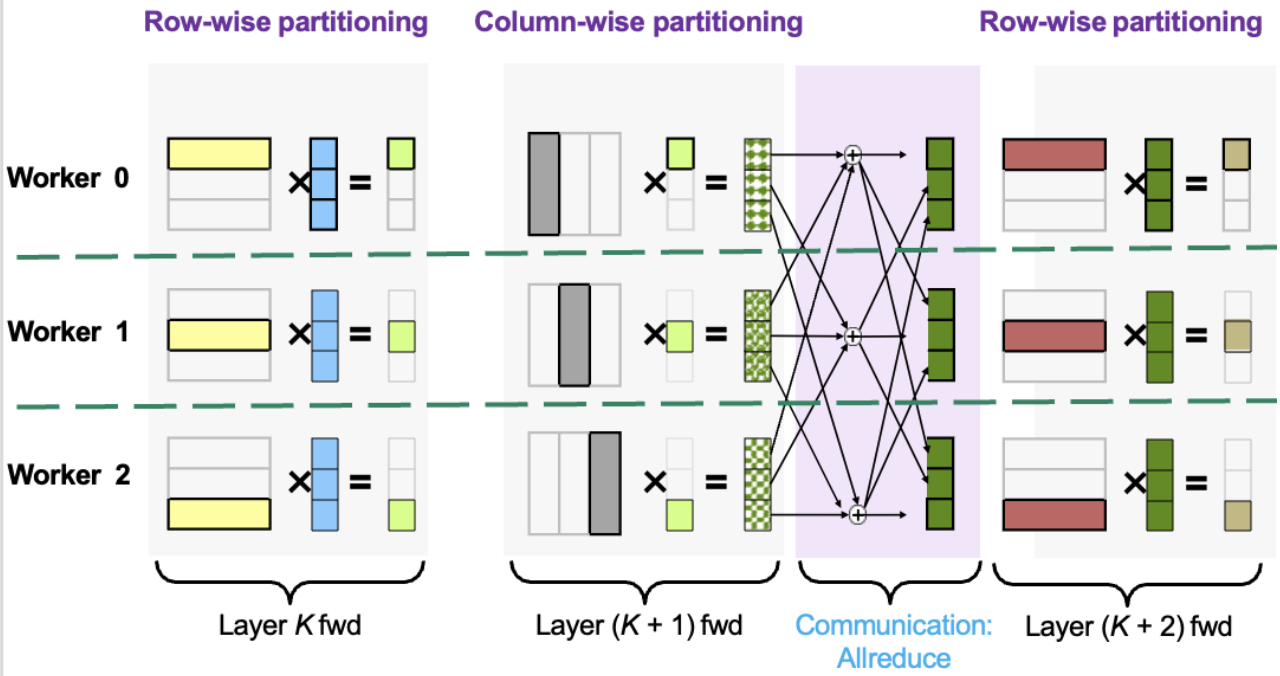


- **Each worker:**

优化的通信方式：allreduce

- 原先每一层通信一次，但是如果交替使用 Row-wise 和 Col-wise： 两次通信（reducescatter+allgather）被优化为一次通信：allreduce

Reducing Synchronization By Alternating Partitioning



Intra-Layer Parallel: Communication

- **Row-wise in forward becomes Col-wise in backward**
- **Col-wise in forward becomes Row-wise in backward**
- **Row-wise:**
  - **Fwd:** allgather
  - **Bwd:** reduce_scatter
- **Col-wise:**
  - **Fwd:** reduce_scatter
  - **Bwd:** allgather
- **When row- and col- are alternated:**
  - Allreduce every two layers, in fwd and bwd
  - Halves the synchronizations compared to not alternating

Communication Pattern Review

- **Data Parallel:**
  - *Allreduce* of weights
  - Can be overlapped with computation

- **Pipeline Parallel:**
  - *Point-wise* communication of activations and activation gradients
  - Hard to overlap with computation
  - Hard to load-balance

- **Intra-layer Parallel:**
  - *Allgather, Reduce_scatter* of activations and activation gradients
  - *Allreduce* if row-wise and col-wise partitioning is alternated
  - Hard to overlap with computation

ZeRO: Zero Redundancy Optimizer

- **Key Idea:**
  - Each GPU stores a subset of optimizer states, rather than the whole states like data parallel.

1.1 传统数据并行的内存问题

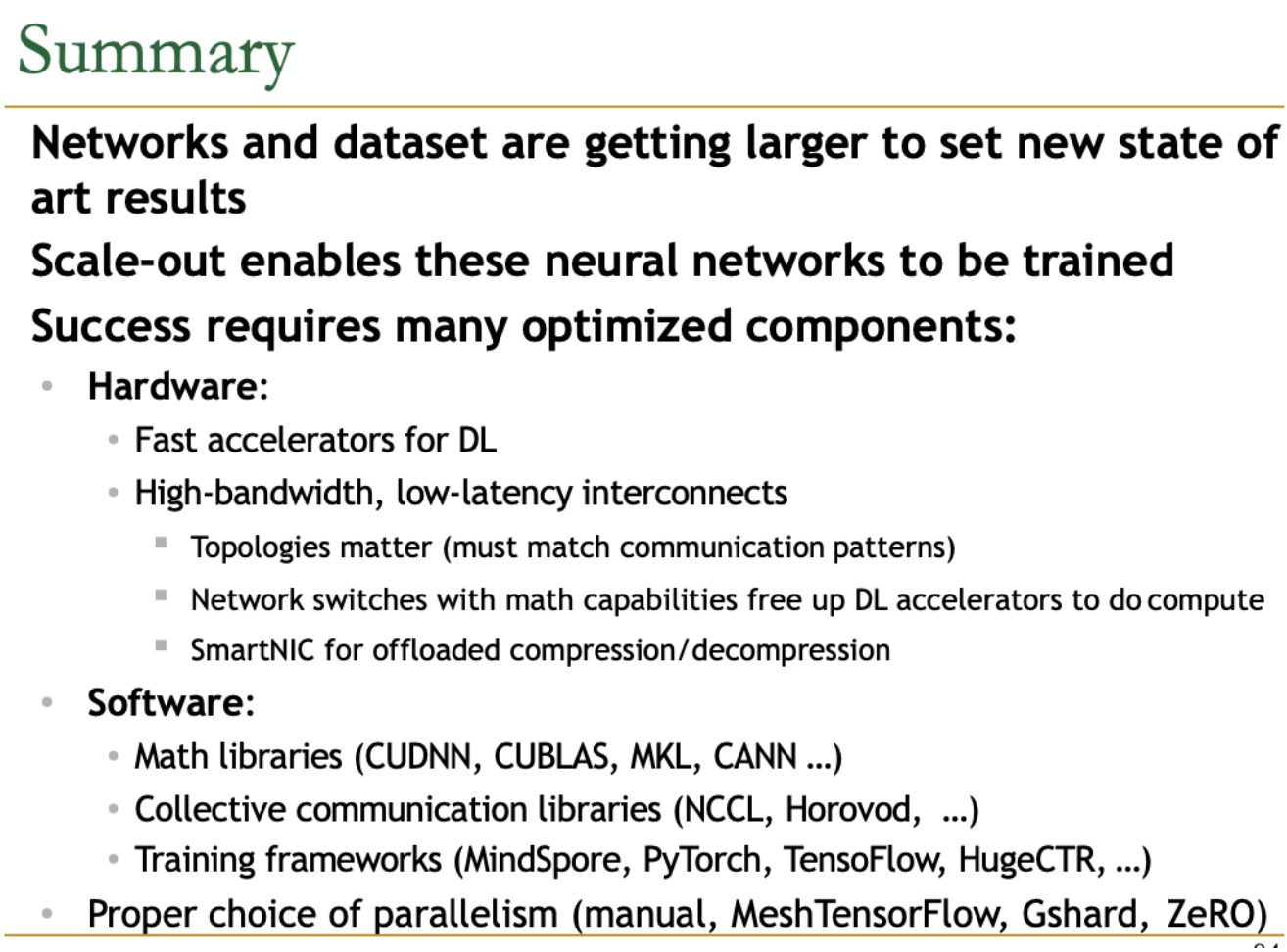
在普通的数据并行（Data Parallelism, DP）中：

- **每个GPU** 存储完整的模型参数、优化器状态和梯度。
- **显存占用高**：尤其是优化器状态（如Adam的动量、方差）可能占用大量显存，限制模型规模。

1.2 ZeRO的核心创新

ZeRO的核心思想是**消除冗余存储**，将优化器状态、梯度和参数**分区到不同GPU**，仅在需要时通过通信重建完整数据。

- **目标**：在不增加计算开销的前提下，**线性降低单卡显存占用**（如ZeRO-3可将显存降低至原来的1/N，N为GPU数量）。



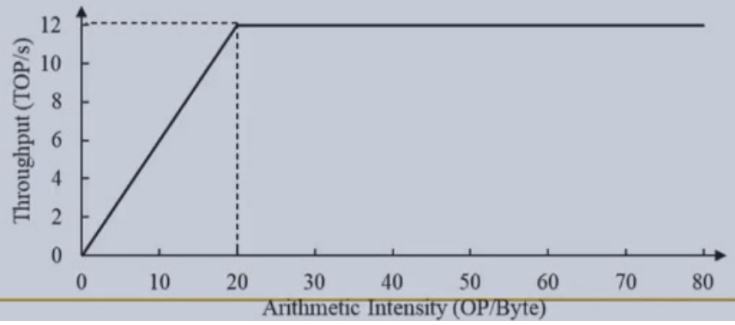
15.期末习题

15.1 roofline model

You're required to evaluate the performance of three operators (Conv, FC and Attention) on an AI processor. The chip manufacturer provides an empty roofline chart as below:

Performance benchmark results of the three operators are given as follow:
The operator Conv has **10000 operations** (OPs) per **1000-byte memory** access and achieves **5.8 TOP/s** on the AI processor.
The operator FC has **30000 operations** (OPs) per **1000-byte memory** access and achieves **7.9 TOP/s** on the AI processor.
The operator Attention has **50000 operations** (OPs) per **1000-byte memory** access and achieves **6.1 TOP/s** on the AI processor.

(a) Please calculate theoretical computing throughput and memory bandwidth of the processor.
(b) Please place each operator onto the roofline chart given above.
(c) Among the three operators, which operators are almost fully optimized and which are not? Please give the reason.
(d) Which operators are memory-bound and which are compute-bound? Please give the reason.
(e) If there exists another implementation of the Conv operator where computing units finish convolution in fewer clock cycles, will its throughput become higher or not? Please give the reason.



54

- （a）理论计算吞吐量（Compute Roof） 和 内存带宽（Memory Bandwidth） 指的是 AI 处理器本身的硬件极限，而不是某个具体 operator 的实测性能。因此答案为12TOP/s和600Byte/s。
- （b）
- （c）operator在图中对应点离改AI下的Throughput极限越近，则说明优化的越好。Cov对应ai下的极限throughput是6top/s，FC和Attention对应ai下的极限throughput都是12top/s，显然5.8/6要高于7.9/12和6.1/12，因此Cov优化得最好。
- （d）Cov是memory bound，FC和Attention是compute bound。
- （e）不会显著提高。Conv 是内存受限算子，其性能瓶颈在于内存带宽而非计算速度。即使减少计算时钟周期，内存访问的延迟仍会限制整体吞吐量。

15.2 little's law

- buffersize（concurrency）=latency*throughput

15.3 pipeline hazard

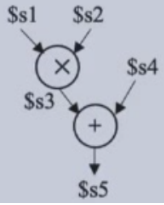
This problem deals with a processor with **out-of-order dispatch** and **precise exception** with 1 adder and 1 multiplier. The adder has a **two-cycle latency** and is fully pipelined, while the multiplier has a **four-cycle latency** and is fully pipelined. Consider the following instruction sequence:

I_1	ADD	$s3, $s1, $s2
I_2	IMUL	$s4, $s1, $s3
I_3	IMUL	$s1, $s3, $s4
I_4	ADD	$s4, $s5, $s3
I_5	IMUL	$s6, $s4, $s5

(a) [2 points] Give an example of write-after-write (WAW) hazard from the instruction sequence. What's the solution to this hazard?

(b) Draw a dataflow graph for the instruction sequence. An example is given as follows.

MUL $s3, $s1, $s2
ADD $s5, $s3, $s4



- （a）注意：第一个寄存器是source寄存器，因此I2和I4是WAW冲突。解决WAW的方法有两种：ROB和寄存器重命名。
- （b）

Please draw your own graph below.

(c) Simulate the instruction execution procedure. Complete the state of each instruction at each cycle in the following table. Use F, D, E, R and W to represent IF, ID, EXE, reorder buffer and WB stage. Use "—" to represent waiting state. You are not required to use all columns in the table.

Cycle	1	2	3	4	5	6
I_1	F	D	E	E	R	W
I_2		F	D			
I_3			F	D		
I_4				F	D	
I_5					F	D

(d) Given a processor with **in-order dispatch without reorder buffer**, while instruction sequence and computing units keep the same, how many cycles does the in-order processor take to finish the procedure? Please give your analysis. Note that precise exception should still be guaranteed here.

Cycle	1	2	3	4	5	6
I_1						
I_2						
I_3						
I_4						
I_5						

- （c）（d）注意：1.F和D阶段都是不需要等待的，只有E阶段需要等待源操作数正常。2.如果是有reorder buffer的情况下，比如I2需要I1的结果，那么只需要等到I1的R阶段结束即可，无需等到WB阶段。

15.4 Performance Analysis

> P is a multi-cycle processor with a clock cycle of **2ns**. Under ideal conditions (with a hit rate of 100%), P executes a load instruction in **4 cycles**, a store instruction in **6 cycles**, an arithmetic instruction in **2 cycles**, and a branch instruction in **3 cycles**.
>
> Let's consider an application called A with **20%** of the instructions being load instructions, **10%** being store instructions, **50%** being arithmetic instructions, and **20%** being branch instructions.
>
> (a) What is the CPI when running application A on processor P under ideal conditions?
>
> (b) P's memory access time for a miss is **100ns**, while the hit time is **1 clock cycle**. The cache is direct-mapped cache, it has a miss rate of **1.4%**. What is the average memory access time of P?
>
> (c) each instruction of application A requires an average of **1.3 memory accesses**, and A has **100 instructions**. What is the CPU time of process P to run application A, taking into account cache misses?
>
> (d) Replace the cache of P1 with a 2-way set-associative cache. It has a miss rate of **1.0%**. Due to the existence of multi-way selection, the CPU clock cycle increases to **1.05** times of the original. Which caching method has faster execution time for application A?

- (a):理想情况下的cpi=4*0.2+6*0.1+2*0.5+3*0.2=3(cycles/instruction)
- (b):1.在非理想的情况下：每个指令执行的周期数都会有所不同，题目给出的信息中每种指令的周期数在本小题已经失效了！2.Memory access time指的是访问内存本身的时间，也就是cache miss后额外花费的时间（miss penalty）！因此AMAT（average memory access time）=2+100*1.4%=3.4ns
- （c）：总时间=理想时间（cache不miss，即无memory访问）+ 真正访问内存带来的延时 = 指令数*理想cpi*单周期时间 + 理论内存访问次数 * caches miss rate（内存访问操作中，实际真正进行内存访问的比例）*内存访问延时 = 100*32 + （100*1.3）*1.4%*100=782ns；
- （d）：time=100*3（2*1.05）+ （100*1.3）*1.0%*100=760ns，相较于（c）更优

15.5 Cache

A computer system has a **32-byte** cache, the size of each block is **4 bytes**. The smallest addressable unit is **1 byte**. Given the following access sequence S1 and the cache is empty at the beginning.
S1: 0x8, 0x28, 0x8, 0x88, 0x8, 0x28

(a) If the cache is direct mapped. Analyze whether each memory access is hit, if an access causes a cache miss, what kind of miss it is, compulsory miss, capacity miss, or conflict miss? And explain why.

(b) if the cache is 2-way set-associative (using LRU replacement strategy), analyze whether each memory access is hit, if an access causes a cache miss, what kind of miss it is, compulsory miss, capacity miss, or conflict miss? And explain why.

(c) If the cache is fully associative (using LRU replacement strategy), analyze whether each memory access is hit, if an access causes a cache miss, what kind of miss it is, compulsory miss, capacity miss, or conflict miss? And explain why.

(d) Comparing the hit rates of three caches under the given access sequence in the question, which cache mapping policy has the highest hit rate.

解题关键在于分析结构：首先32byte的cache，block大小是4byte，也就意味着一共有8个block，编号分别为0-7。其次，每个block大小为4byte，也就意味着每个地址的后2位（0-1位）会用来储存在每个block内部的offset；对于direct mapped而言，一共8个block，也就意味着每个地址去掉offset后的倒数3位用来储存对应cache的编号（2-4位）；再往前的数据则用来储存tag；对于 2-way而言，8个block分为4set，因此对应cache编号的位数变成了2位；如果是full-associative的话，不需要任何储存cache编号的数据。最后，第一次填入cache属于compulsory miss；后续属于capacity miss；几乎看不到conflict miss。