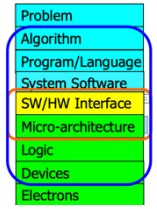
## Introduction



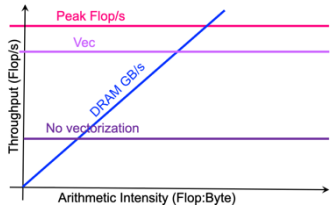

计算机架构的狭/广义定义（左图）

**Amdahl's Law**：加速=1/(1-f+(f/N))

无法完全并行（f 有限）的三个原因：Synchronization overhead； Load imbalance overhead ； Resource sharing overhead 。**Roofline model:** Attainable Flop/s = min( peak Flop/s, AI * peak GB/s )拐点左侧：Memory-bound 拐点右侧：Compute-bound

不同介质的斜率：cache>hbm>ddr

- No vectorization: none
- Vec: vectorization code
- Peak Flop/s: fused multiply-add + vectorization code




每次循环的内存占用：x[i]、y[i]、z[i],运算量 2 flops(加/乘)

```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

题目：1.理论吞吐量指的是 AI 处理其本身的硬件极限。对于某个算子来说，其在图中对应点离该 AI 下的 Throughput 极限越近，则说明优化的越好 2.对于内存受限算子来说，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 3.带宽的单位是 byte/s 4.只有增加 alu 数量才会影响 roofline model。乱序执行、超标量、多线程无法影响，多核、simd 会影响模型。

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

## Pipeline hazard+ROB

**冯诺伊曼模型：**内存、处理器、输入、输出、控制单元

两大性质：1.Stored program：内存对于数据和指令通用，依据控制信号被具体被解析。2. Sequential instruction processing

多周期 CPU：减少单周期时间，每条指令获取适合自己的执行周期数。

- Single-cycle: AS → AS' (transform AS to AS' in a single clock cycle)
- Multi-cycle: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

流水线 CPU：Increases instruction processing throughput (1/CPI)

**数据依赖：** 1flow-dependency 写后读 2output- ~ 写后写 3.anti- ~ 读后写只有 1 种是真依赖，后面 2 种是因为寄存器名不够

**Reorder Buffer：**顺序发射、乱序执行、顺序写回

题目:F 和 D 阶段都不需要等待，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 的进入 R 的同时，I2 就可以进入 EXE。

*具体作用*：1.正确地将指令重新排序回程序顺序 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 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 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。*工作方法*：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。*局限*：只解决假数据依赖。

**OoO**：out of order dispatch：解决了真实数据依赖。

## Tomasulo 算法

*首先等待直到保留站是否有空闲条目。对于每个源寄存器，检查寄存器文件(RF)中的有效位，若有效位 1:直接读取 RF 中该寄存器的值到保留站条目，标记保留站中的 v=1;若有效位 0:读取 RF 中该寄存器的标签记录到保留站中，并标记保留站中的 v=0，意思是该寄存器的值还没有准备好。对于目标寄存器，在 RF 中更新目标寄存器的标签字段为保留站条目标签(重命名)，更新 RF 中的 v=0。*总体来说属于 OoO 的一种具体算法，具体包括以下组件：前者实现了 OoO；后者实现了 in-order commitment（实际通过寄存器重命名）。

Hump 1: **Reservation station** (scheduling window)

Hump 2: **Reorder Buffer** ( aka instruction window or active window)

题目：1.在非理想的情况下：每个指令执行的周期数都会与理想条件下不同！

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）！3.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

## Superscalar + SIMD + Multi-core

**超标量：**每个 cycle 执行多条指令。 **Flynn's Taxonomy of Computers:** sisd/simd/misd/mimd **向量指令**：SIMD 的一种：1.增加 ALU 的数量，实现并行。2.新增向量寄存器文件。3.内存需新增读写口。

**Multi-threading**(单核)：*优*：1.无需指令间依赖性检查。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。线程内的数据依赖和结构依赖自然消失。2.无需分支预测逻辑：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。3. 利用空闲周期执行其他线程的指令，显著提高流水线利用率。4. 提升系统吞吐量和延迟容忍能力。*劣*：硬件复杂度增加；单线程性能下降；线程资源竞争；跨线程的依赖检查残留。

**Multi-Core：***优*：单个核心更简单；多任务吞吐量更高；*劣*：依赖并行编程；资源竞争影响单线程性能；共享资源管理复杂；IO 带宽限制（引脚瓶颈）

## Memory

理想内存：零延迟，大容量，大带宽，零花费。

sram/hbm/dram/ssd/disk:*capacity*:10mb/10gb/100gb/1tb/10tb; *bandwidth* 1tb/s\100gb/s\10gb/s\1gb/s\10mb/s; *latency*:1ns/10ns/100ns/1us/1ms

**memory array**：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

word line 定位行；bit line 定位列。

**Memory Bank：**把内存分为 n 块，可并行访问读写。

注意：内存的访问比浮点数计算等都要消耗更多能量！

**SRAM：**六个晶体管，更简单。**DRAM：**两个元件（一个晶体管、一个电容），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；由于 DRAM 比较大，因此要分层次进行管理，从大到小分别为：channel/dimm/rank/chip/bank/row&column

**HBM**:高带宽内存(高速专用通道)为 ai/gpu高带宽场景量身定制: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.

**DDR**:时钟上下沿都能读写，经济型高速通道。

**DRAM 访问的 3 种状态（延时依次升高）**：*1.Page Hit* 当内存事务访问的 row 已经在其所属的 bank 中处于打开状态时。可以直接进行 column access。*2.Page Closed*：当内存事务访问的行所在的 bank 处于关闭状态时。需要先发送 Activate 命令打开目标行，然后才能进行列访问。*3.Page Miss:* 当内存事务访问的行与当前 bank 中已打开的行不匹配时。需要先发送 Precharge 预充电命令关闭当前打开的行，然后发送 Activate 打开目标行，最后才能进行列访问。

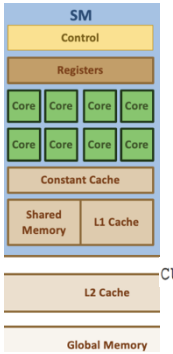
**DRAM 访问特性**：每次读出来的都是一行，整行的数据都被存在 row buffer 中。 因此当连续访问同行的不同列时，延迟很低，因为可以直接在 buffer 中找到； 突然访问其他行，很费时。

```
__shared__ float A_s[TILE_DIM][TILE_DIM], 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) {
    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();
    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;
```

## GPU Architechture

CPU:核少;大 cache;内存大且慢;功能多;性能差。gpu 相反。

1GPU=30sm=30*32warps=30*32*32threads=30K 线程




左图是 **GPU Memory，Hardware Execution Model** 是由众多 sm 构成；对应 **Cuda Programming Model** 的是：gpu - grid / sm - thread block / cuda core - thread

**warp：**32 个线程共享指令流，是 gpu 的调度单位。

**编程模型**有很多，包括：sequential（sisd）；data parallel（simd），一条指令完成向量操作；multithread（spmd）；

**SPMD：**不同指令流执行相同程序（cuda 是其中一种）

```
float *A_d, *B_d, *C_d; cudaFree(A_d);
cudaMalloc((void**) &A_d, N*sizeof(float));
kernel<<< #blocks, #threads >>>(args);
cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
```

## GPU OPtimization

**FGMT：Fine-grained multithreading：**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。**Occupancy**：active warps 的比例。

**Memory Coalescing**：同一个 warp 内的 thread 访问的 memory 的地址是连续的！也就意味着访问同一个 row 中的数据；（在 memory 的同一个 busrt 中）也就意味着：Only one DRAM transaction is needed。

**Shared Memory**：本质是 SRAM。共享内存是一种交错式（分 Bank）存储结构。每个 Bank 每个时钟周期可以响应一个地址的访问。NVIDIA GPU 中，通常有 32 个 Bank。Bank 编号 = 内存地址 % 32。Bank 冲突（也就是串行化）只可能发生在一个线程束（Warp）内部。不同 Warp 之间不会发生 Bank 冲突(不同 Warp 的请求在时间上错开)。

**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 workset 拆分成多个 chunk, 使得每个小块的数据都在芯片的高速存储上，避免不同计算块之间对片上芯片的访问冲突。

**Divergence**：同一个 warp 内执行的代码不同，效率低下。

Cache cache 题目如下所示

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。

**direct-mapped/fully-associative/n-way set associative**，读写单位是 block! 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。

**PLRU**：查表选择某路，该路的所有 bit 都要取反。**Belady's OPT**：最优策略

**Set Thrashing**：工作集>set associativity -> LRU 并不是一定比 random 好

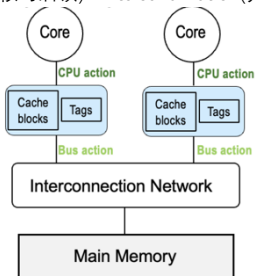
**Write Policy**:store 类指令：*write-allocate/write-no-allocate:*后者直接写入 memory；前者是写入 cache，但是在写入前要先从 memory 读入整个 cache block(cache 的读写单位),因为 block 要大于 byte（内存读写单位），防污染。cache 写入 memory 的策略*：write-back (*写入 cache，满了更新到内存）和 *write-through* (同时写入 cache 和内存)；前者需要标记 cache 是否被污染。*data-cache 和 instruction-cache:* 高级 cache 统一设计；底层分开设计。

随着 cache-size/block-size/associativity 上升,hit rate 上升/先升后降/上升。

Coherence and Consistency

**coherence：**同一内存地址的多个副本保持一致

3种类型：program order preservation(同核写后读)/coherent memory view(异核写后读)/write serialization(异核写后写)；硬件架构如下：




从上到下 4 层分别是：core/cache/interconnect/memory

**interconnect:**分为 bus/switch。前者：不同核通过申请使用总线，完成通信；后者：每一对核之间都可以完成独立通信。

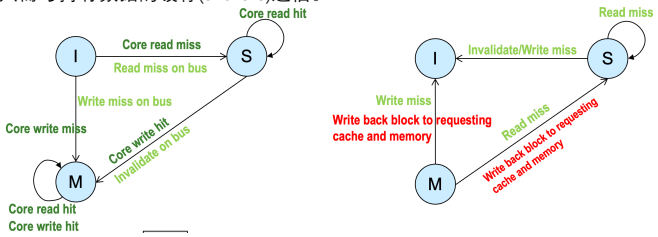
**cache update:**分为 update/invalidate update 优点：减少一致性通信的开销，因为数据变更时会主动更新所有副本。缺点：不必要的更新可能浪费带宽；写透策略下所有写入都要通过总线。 另一汇方法相反。

**MSI：**1.M: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: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:缓存块无效或不在当前缓存中。更新策略：只要从 S 发生了写，发出 invalidate 信号。*存在问题：*数据第一次被读如 cache，尽管只有唯一副本，但还是被标记为 S(share)状态，*进而引发问题*：该数据如果发生了写操作，也会引起 bus 上的 invalidate(误判为要更新其他共享此副本的 cache)，但根本没有其他副本。

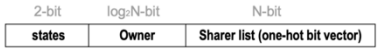
**MESI**:1.M 仅存在于一个缓存中，已被修改，与主内存不一致 2.E:仅存在于一个缓存中，未被修改(干净)，与主内存一致 3.4.S/I 与 MSI 一致。E 状态带来的好处：仅有"多核且其中一个核发生写操作"时，才会触发 invalidate 信号；避免"唯一核更新"也触发 invalidate 信号，浪费总线资源。

*注意：write-allocate：写入 cache 前，要先从 memory 中读取整个 block!*

**snoop-based cache coherence(bus)**缺点：1. 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.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。




**Directory-Based Cache Coherence(switch):** 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）。3 种 node：1.home-node：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 directory（如副本位置等）。2.local-node：向 home node 发送请求（如读写请求等）。根据 home node 的响应执行操作（如无效化本地副本等）。3.remote-node：被动响应 Home Node 指令的节点(持有该缓存块副本的节点)。比如:remote node 需将脏数据写回 home node 并降级状态。




**无论是 snoop 还是 directory-based,只要是读操作,一定要在读取到 local 的同时写回内存！因为 m/e 状态是 dirty，而 shared 状态是 clean！**

**consistency**：不同内存地址的读写在不同核中观察顺序一致

**4barries**：读后读/后写/写后读/写后写。一致性越差，性能越好，编程越难；

**Sequential Consistency:**每个处理器内部的顺序不能被打乱；满足全部 4 种。(All processors see the currently serviced load or store at the same time)

**Total Store Order:**Sequential Consistency + Store Buffer:不满足先写后读.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,优化了写入的性能。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。*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。*

**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。也就是说*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。

*Order 是通过排序 shared data 来影响程序正确性的!*

**Mutual Exclusion：**在任何给定时间，只能有一个线程执行临界区代码。

**多级 cache:**1 级 cache:容量小,associativity 小,tag store 和 data store 并行访问,追求低延迟；2 级 cache:容量大,关联度高，标签和数据串行访问(先检查标签,命中后再访问数据)不追求低延迟。只有 1 级 miss 时，才会访问 2 级。

DL Accelerator

| 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 |

**两大特性**：计算特性：重复的计算模式(循环)；访存特性：局部性（连续访问）

**设计思路**包括以下 5 部分：并行计算模块；简化控制模块；专用编程语言；global buffer(因为 cache 对 stride 很不友好,1 个 stride 内容易造成同一个 set 竞争且访问方式固定，可以直接手动控制而不需要 cache 那么复杂的设计,和 cache 相比 buffer 能耗低、面积小)；量化(低精度运算->按位储存数据,需要多少精度就取几位-> bit serial multiplier)。与 cpu 相比，ai 处理器：指令内并行！目标高吞吐！

AI Processor

**减少内存访问:**1.用 global buffer 代替 dram;2.mac:multiply-accumulate 矩阵运算。减少 **global buffer** 访问：增加 PE（processing element）自带的寄存器文件的使用，代替 global buffer。*4 种 Stationary：1.Weight S*：把网络权重始终留在寄存器文件中，避免从 global buffer 中读取；*2.Input S：*把 activation 留在 pe 内；*3.Output S：*把 Psum 留在 pe 内；*4.Row S：*从 Global Buffer 读出 Filter 中的一行和 Activation 的一个滑窗，留在 PE 内。

**增加计算模块：**矩阵乘法单元 Cube：单周期内完成，算力密度极高。

**Ascend 芯片：***L2buffer 和 L2 Cache*：同一种介质，两种使用模式，前者程序员可见而后者不可；*L1buffer*：最大的一块数据中转区，MTE 数据格式转化功能的元数据必须位于其中；*L0buffer*：用于储存矩阵运算用的矩阵；*UnifiedBuffer*:用于储存向量运算的向量,部分可用 scalar 运算；*ScalarBuffer*:专门用于标量运算；*Cube：*矩阵乘法单元；*Accumulator：*矩阵相加单元；*A/B/Accum DFF：*数据寄存器，缓存左/右/结果矩阵；*VectorUnit：*向量运算单元；*GPR*通用寄存器；*SPR：*专用寄存器；*BIU*：ai-core 和 bus 的交互接口；*MTE：*也被称作 *LSU*，管理芯片内数据在不同 buffer 中的管理及格式转化(padding，转置等)。*异步发起同步跑。优势*是算力极高、buffer 访问管理效率高、提供随路指令，*缺点*是编程难度高、不完善。

**GoogleTPU：Systolic Array** 通过规则排列的简单处理单元 PE 实现高吞吐量的计算(采用重复性高的模块，降低设计复杂度、提高可扩展性)。*优势：*1.传统架构中数据从内存加载后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。2.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。*类比*：内存是心脏，数据是血液，处理单元是细胞。

**二维列阵**：*right=left/down=upper/cell=cell+upper*left*

**Cambridge**:*单核 DLP*-S:3 个模块:1 控制模块(取指令、解码);2 计算模块(向量乘法、矩阵乘法);3SRAM 模块(权重 ram,神经元 ram,direct memory ram)。*DLP-C*:4 个单核组成和核内存(4 单核的共享内存);*DLP-M* 由 4 个-C 组成

Runtime(cuda/cann 等) & Framework(pytorch/tensorflow 等)

**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。向下使能处理器并行加速，向上使能高效开发。

**Ascend 算子库包括**：NN 库(用于处理 pytorch 等类型)、BLAS 库(处理线性代数)、DVPP 库(视频、图片等编码解码裁剪缩放)、AIPP 库(图片预处理尺寸、色域等)、HCCL 库(用于多机多卡通信)

**Tensor 属性：**名称(不同 tensor 唯一)；形状(1024,1024)；数据类型；数据排布格式(N,H,W,C)。**CANN 开发方式：**TBE DSL / TIK / AI CPU 3 种,区别在于：语言 Python/Python/C++；计算单元:AI Core/ai core/AI CPU；运用场景：简单的向量、矩阵及池化运算/复杂算子开发(排序类操作)/极复杂场景或连接网络的场景；入门难度：低/高/中。

*-> tik 处理细碎复杂的小问题，dsl 处理大型运算。*

**计算图引擎：**通过将计算过程抽象为有向无环图，实现优化。优化方式：**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。

**算子融合 Intuition**:算子都是去 buffer 中获取数据，融合算子就可以减少数据在 buffer 和 memory 之间的搬运。

**Framework：**将算法中常用的操作封装成组件。

**Mindspore 优势**：1.自动并行;2.二阶优化;3.动静态图结合;4.AI+科学计算

Parallel Training

Y=X*W 只需要 1 次运算；但 BP 时，*dW=dY*X^T , dX=W^T*dY* 需要两次。

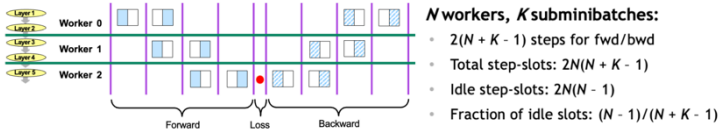
**Data Parallel:** N 个 worker *各自持有模型的完整副本，并处理不同的数据子集*(数据分片)，独立计算其本地数据的梯度。所有 worker 需要共享梯度，将自己的梯度分成 1/N 份，然后将其中(N-1)份发送给其他 worker；同时从其他 worker 接收对应的梯度分片并将它们相加，得到全局梯度的平均值。

**Ring Allreduce** 算法：总共 2(N-1)步，每一步只向一个邻居发送数据(weight)，并接收另一个邻居的数据。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。

**Strong Scaling:**增加 worker 数量，保持 minibatch 大小不变，缩短计算时间，目标是缩短计算时间；**Weak Scaling**：增加 worker 数量也增加 minibatch 的大小，目标是问题规模扩大的时候保持处理时间不变。

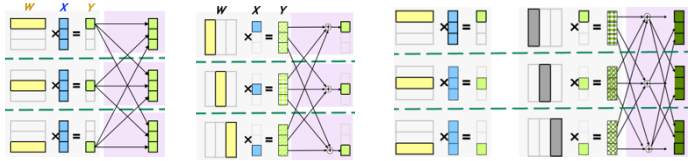
**Model Parallel**:分为 层间并行(流水线并行) 和 层内并行。

**Inter-layer/Pipeline(层间)**：每个 worker 负责某几层 layer。




*存在的挑战:*负载不均衡(不同层计算量差异大)；计算通信重叠难(通信延迟无法被计算完全隐藏)；扩展效率低(Worker 空闲导致资源浪费)；子小批量权衡(过小的子批量会引入类似强数据并行的通信瓶颈)。

**Intra-layer：**每个 worker 负责所有层的某几部分。两种分割方式：




row-wise：按行分割，通信方式为 **Allgather**；column-wise：按列分割，通信方式为 **ReduceScatter。**按这两种方法每 1 层通信一次。但是如果交替使用 Row-wise 和 Col-wise ： 两次通信被优化为一次通信：**Allreduce**

在 BP 过程中，row-wise 会变成 col-wise，反之亦然。

*Data-parallel can be overlapped with computation, but Model-parallel can't.*

原因是对于数据并行的某个 worker 而言，某一批次训练完可以立即执行下一批次的训练；而模型并行不然，前向传播的计算是层间串联的，必须等待前一层的输出到达才能开始下一层计算。

**ZeRO:**zero redundancy optimizer：普通的数据并行中:每个 GPU 存储完整的模型，显存占用高；ZeRO 的核心思想是消除冗余存储，将优化器状态分区储存到不同 GPU，仅在需要时通过通信重建完整数据。