???????

- Memory Consistency and Cache Coherence
  - Memory Consistency and Cache Coherence 定义
  - TSO（Total Store Ordering）内存模型
  - x86的多核宽松内存一致性模型
- einsum
- RAM
- NoC
- Cache写机制 Write-through与 Write-back
- 人工智能的产业
- Nand flash
- acquire release 实现内存一致性

Memory Consistency and Cache Coherence

Memory Consistency and Cache Coherence ??

1. Memory Consistency???????

- 定义：内存一致性是指多核或多处理器系统中，多个处理器对共享内存的访问顺序是否一致，以及这些访问操作是否满足特定的规则或模型。
- 关注点：内存操作的全局可见顺序，确保所有处理器对内存的读写操作按照一致的顺序进行。
- 问题背景：在多核系统中，不同处理器可能同时访问共享内存，如果没有明确的内存一致性模型，可能会导致程序行为不可预测。
- 内存一致性模型：
  - 顺序一致性（**Sequential Consistency**）：所有处理器的内存操作按照一个全局顺序执行，且每个处理器的操作顺序保持不变。
  - 弱一致性模型（**Weak Consistency**）：允许某些操作乱序执行，但需要通过同步操作（如内存屏障）来保证一致性。
- 示例：
  - 如果两个处理器分别执行写操作和读操作，内存一致性模型定义了读操作是否能立即看到写操作的结果。

---

2. Cache Coherence???????

- 定义：缓存一致性是指多核或多处理器系统中，每个处理器的私有缓存与共享内存之间数据的一致性，确保所有处理器看到的数据是最新的。
- 关注点：多个缓存副本之间的数据一致性，避免脏数据或过时数据。
- 问题背景：每个处理器都有自己的缓存，如果某个处理器修改了缓存中的数据，其他处理器的缓存可能仍然保存旧值，导致数据不一致。
- 缓存一致性协议：
  - **MESI**协议：通过标记缓存行的状态（Modified、Exclusive、Shared、Invalid）来维护一致性。
  - 写失效（**Write Invalidate**）：当一个处理器修改数据时，使其他处理器的缓存副本失效。
  - 写更新（**Write Update**）：当一个处理器修改数据时，将新值广播给其他处理器的缓存。
- 示例：
  - 如果处理器**A**修改了某个内存位置的值，处理器**B**的缓存中该位置的旧值必须被更新或失效。

3. ????

特性	**Memory Consistency** （内存一致性）	**Cache Coherence** （缓存一致性）
关注点	内存操作的全局顺序和可见性	多个缓存副本之间的数据一致性
范围	涉及所有内存操作（读/写）的顺序	仅涉及缓存和内存之间的数据一致性
问题背景	多个处理器对共享内存的访问顺序是否一致	多个处理器的缓存中数据是否一致
解决方案	内存一致性模型（如顺序一致性、弱一致性）	缓存一致性协议（如 MESI、写失效、写更新）

4. ??

- **Memory Consistency** 关注的是多处理器系统中内存操作的全局顺序和可见性。
- **Cache Coherence** 关注的是多处理器系统中缓存副本之间的数据一致性。

Memory Consistency and Cache Coherence

TSO?Total Store Ordering?????

1. TSO（Total Store Ordering）是一个被广泛使用的内存模型
2. 并在x86架构中使用，RISC-V也提供了TSO扩展，即RVTSO，人们普遍认为x86内存模型等同于TSO，然而Intel和AMD从来没有保证这一点
3. x86选择放弃SC（顺序一致性sequential consistency），以更好地支持基于FIFO的write buffer，用于加速性能
4. TSO和SC最关键的区别就是store可能被放入write buffer中且允许load的bypass
    1. 对于单核视角来说，和SC没有区别，执行顺序和程序顺序一致
    2. 对于多核来说，需要额外的手段来保证memory consistency
5. 由于TSO只允许store-load的重排，因此需要使用FENCE的场合不多，FENCE的实现方式也不是很重要。一种简单的实现是，当FENCE被执行时，清空write buffer，并在FENCE提交之前不再执行后面的load指令。

Memory Consistency and Cache Coherence

x86????????????

1. 被修饰的汇编指令成为“原子的”
    1. 本身是原子指令，比如“XCHG”和“XADD”汇编指令
    2. 本身不是原子指令，但是被`LOCK`指令前缀修饰后成为原子指令，比如 LOCK CMPXCHG
    3. 被修饰的汇编指令A在执行期间，会在内存总线上声言一个`#LOCK`信号，该信号导致内存被锁住，此时内存不能再被其他汇编指令存取，直到A执行完成。经过分析可知，A的执行效果与“暂停执行其他所有汇编指令直到A执行完成等价，因此此时A是原子的
2. fence
    1. sfence: 在 sfence指令前的写操作当必须在sfence指令后的写操作前完成
    2. lfence: 在 lfence指令前的读操作当必须在lfence指令后的读操作前完成
    3. mfence: 在 mfence指令前的读写操作当必须在mfence指令后的读写操作前完成
3. C++11中支持的内存模型
    1. Acquire-Release能保证不同线程之间的Synchronizes-With关系，同时也约束到同一个线程中前后语句的执行顺序。
    2. Release-Consume只约束有明确的carry-a-dependency关系的语句的执行顺序，同一个线程中的其他语句的执行先后顺序并不受这个内存模型的影响

```
enum memory_order {
    memory_order_relaxed, // Relaxed
    memory_order_consume, // Release-Consume
    memory_order_acquire, // Acquire-Release 用来修饰一个读操作，表示在本线程中，所有后续的关于此变量的内存操作都必须在本条原子操作完成后执行
    memory_order_release, // Acquire-Release 用来修饰一个写操作，表示在本线程中，所有之前的针对该变量的内存操作完成后才能执行本条原子操作
    memory_order_acq_rel, // Acquire-Release 同时包含memory_order_acquire和memory_order_release标志
    memory_order_seq_cst  // 顺序一致性模型
};
```

einsum

??????

自由索引（***Free indices***）和求和索引（***Summation indices***）：

- 自由索引，出现在箭头右边的索引，比如上面的例子就是 i 和 j；
- 求和索引，只出现在箭头左边的索引，表示中间计算结果需要这个维度上求和之后才能得到输出，比如上面的例子就是 k；

??????

- 规则一，equation 箭头左边，在不同输入之间重复出现的索引表示，把输入张量沿着该维度做乘法操作，比如还是以上面矩阵乘法为例，"ik,kj->ij"，k 在输入中重复出现，所以就是把 a 和 b 沿着 k 这个维度作相乘操作；
- 规则二，只出现在 equation 箭头左边的索引，表示中间计算结果需要在这个维度上求和，也就是上面提到的求和索引；
- 规则三，equation 箭头右边的索引顺序可以是任意的，比如上面的 "ik,kj->ij" 如果写成 "ik,kj->ji"，那么就是返回输出结果的转置，用户只需要定义好索引的顺序，转置操作会在 einsum 内部完成。

????

特殊规则有两条：

- equation 可以不写包括箭头在内的右边部分，那么在这种情况下，输出张量的维度会根据默认规则推导。就是把输入中只出现一次的索引取出来，然后按字母表顺序排列，比如上面的矩阵乘法 "ik,kj->ij" 也可以简化为 "ik,kj"，根据默认规则，输出就是 "ij" 与原来一样；
- equation 中支持 "..." 省略号，用于表示用户并不关心的索引，比如只对一个高维张量的最后两维做转置可以这么写：

```
a = torch.randn(2,3,5,7,9)
# i = 7, j = 9
b = torch.einsum('...ij->...ji', [a])
```

?????bilinear transformation?

```
np_a = a.numpy()
np_b = b.numpy()
np_c = c.numpy()
np_out = np.empty((2, 5), dtype=np.float32)
```

```
np_out = torch.einsum('ik,jkl,il->ij', [a, b, c]).numpy()
# ik broadcast成ikl
# il broadcast成 ikl
# 'ik,jkl,il->ij'可以理解成'ikl,jkl,ikl->ij'

for i in range(0, 2):
    for j in range(0, 5):
        # 求和索引内循环 这里是 k 和 l
        sum_result = 0
        for k in range(0, 3):
            for l in range(0, 7):
                sum_result += np_a[i, k] * np_b[j, k, l] * np_c[i, l]
        np_out[i, j] = sum_result
```

??

```
a = torch.rand(2,3)
b = torch.rand(3,4)
c = torch.einsum("ik,kj->ij", [a, b])
# 等价操作 torch.mm(a, b)
```

equation 中的字符也可以理解为索引，就是输出张量的某个位置的值，是怎么从输入张量中得到的，比如上面矩阵乘法的输出 c 的某个点 c[i, j] 的值是通过 a[i, k] 和 b[k, j] 沿着 k 这个维度做内积得到的。

RAM

DRAM

1. 电容
2. 带宽不是很高
3. 需要刷新，会有颠簸

SRAM

1. 面积和功耗不能和工艺平行
2. 类型
    1. Cpu register Flip Flops 每个 bit都有一读一写
    2. L1/L2 SRAM 6个晶体管，一般最多每个 bank一读一写
    3. L3/L4 eDRAM/GCRAM 4晶体管 /电容 读写速度明显降低，也会有使用上的问题

NoC

OpenSMART

https://github.com/hyoukjun/OpenSMART/tree/master

connect

https://users.ece.cmu.edu/~mpapamic/connect/

https://github.com/crossroadsfpga/connect/tree/main

Flexnoc

is a commercial NoC generator by Arteris which generates a customized topology for each SoC,

Cache??? Write-through?Write-back

Cache写机制分为 **write through**和 **write back**两种。

Write-through: Write is done synchronously both to the cache and to the backing store. Write-back (or Write-behind) : Writing is done only to the cache. A modified cache block is written back to the store, just before it is replaced. Write-through（直写模式）在数据更新时，同时写入缓存 Cache 和后端存储。此模式的优点是操作简单；缺点是因为数据修改需要同时写入存储，数据写入速度较慢。

Write-back（回写模式）在数据更新时只写入缓存 Cache。只在数据被替换出缓存时，被修改的缓存数据才会被写到后端存储。此模式的优点是数据写入速度快，因为不需要写存储；缺点是一旦更新后的数据未被写入存储时出现系统掉电的情况，数据将无法找回。

Write-misses写缺失的处理方式 对于写操作，存在写入缓存缺失数据的情况，这时有两种处理方式：

Write allocate (aka Fetch on write) – Datum at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read-misses. No-write allocate (aka Write-no-allocate, Write around) – Datum at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, actually only system reads are being cached. Write allocate方式将写入位置读入缓存，然后采用 write-hit（缓存命中写入）操作。写缺失操作与读缺失操作类似。

No-write allocate

方式并不将写入位置读入缓存，而是直接将数据写入存储。这种方式下，只有读操作会被缓存。

无论是 **Write-through**还是 **Write-back**都可以使用写缺失的两种方式之一。只是通常 Write-back采用 Write allocate方式，而 Write-through采用 No-write allocate方式；因为多次写入同一缓存时，Write allocate配合 Write-back可以提升性能；而对于 Write-through则没有帮助。

处理流程图

Write-through模式处理流程： A Write-Through cache with No-Write Allocation

[![image.png](Cache写机制 Write-through与 Write-back/BLAimage-png.png)](Cache写机制 Write-through与 Write-back/BLAimage-png.png)

Write-back模式处理流程： A Write-Back cache with Write Allocation

[![image.png](Cache写机制 Write-through与 Write-back/aCyimage-png.png)](Cache写机制 Write-through与 Write-back/aCyimage-png.png)

???????

1. 模型算法
   1. 科研
   2. 企业商用
   3. 数据收集、标注
2. 软件框架
   1. 科研
   2. 商业部署
3. 加速芯片
   1. 云训练芯片
   2. 云推理
   3. 边沿推理
4. 云服务
   1. 基础软件框架
   2. 硬件
   3. 服务
5. 应用
   1. 各种AI云服务
   2. 边缘AI
   3. 特定领域
      1. 基础设施：安防
      2. 自动驾驶、机器人
      3. 工业、医疗、教育、金融

Nand flash

LUN → CE → Die → Plane → Bank → Block → Page

1. Block是擦除操作的最小单位
2. Page是读写操作的最小单位，常见的Nand Flash多数是2KB，最新的是4KB、8KB
3. 硬件电路只支持**“**一次性操作一个**Page**的所有单元**”**，因此读写必须以**Page**为单位。
4. 写入前必须先擦除对应的Block
5. 栅极堆叠（Gate-Stacked）和沟道堆叠（Channel-Stacked）两种结构均需解决“垂直互联”，如何连接不同层的字线WL和位线BL
6. WL BL SSL三个控制线构成三维浮栅晶体管的索引信号，确定WL和SSL，读写所有的BL，也就是一个Page
7. 不同WL的晶体管是串联在一起的，除了需要读取的管，其他的会在读取的时候被强制导通，避免影响需要读取的晶体管

WL?Word Line????

- 位置：WL是平行于浮栅晶体管控制栅的金属或多晶硅连线，贯穿整个存储阵列的行方向（类似“横线”）。
- 连接方式：同一行（Row）中所有浮栅晶体管的控制栅都与同一条WL相连。一个**Page**内的所有存储单元通常属于同一行，共享一条**WL**
- 写入：向WL施加高电压（如10-20V），通过“热电子注入”或“F-N隧穿效应”，将电子注入浮栅（存储“0”）。
- 读取：向WL施加中等电压（如2-5V），根据晶体管是否导通判断浮栅是否带电荷（即区分“0”和“1”）。
- 擦除：擦除操作针对整个Block（块），此时WL接地，衬底施加高电压，通过F-N隧穿效应让浮栅释放电荷（恢复为“1”）。

BL?Bit Line????

- 位置：BL是垂直于WL的金属连线，沿存储阵列的列方向延伸（类似“竖线”）。
- 连接方式：同一列（Column）中所有浮栅晶体管的漏极通过接触孔（Contact）与同一条BL相连。
- 材料：通常采用铝或铜，以降低连线电阻，提高信号传输速度。
- BL是数据信号的**“**传输通道**”**，负责将存储单元的状态（导通/截止）转化为电信号并传递给外部电路
- 一条BL对应一列存储单元，每列中的每个单元对应一个“位”（bit）

SSL GSL??????????

- 通过TSG管选择一个string

MAC??

1. input：SSL
    1. 每个SSL表示input的一位
2. output：BL
    1. 每个BL表示output的1位，
    2. 8个连续的BL组成一个8位的输出
    3. BL经过高精度ADC量化后，再根据每个bit的二进制单位分别进行左移，再累加，得到乘法的结果
3. weight：cell中存储的数据
4. WL：选中一批权重

??MAC????

1. 矩阵乘法MNK
2. SSL input: 所有乘完需要累加的input的指定位，input矩阵的一行（K个）的每个指定位
    1. 拆分input的K
3. BL output: 一个BL表示一批input的一位和weight的位乘加的结果，一次性最多输出N个结果的1位
    1. 拆分output的N
4. WL weight: 选中一批权重的指定位，可以用来选定权重批次，或者权重的位index
    1. 支持用于拆分权重的N或者bit index

[![b81dc059-2002-4e2b-932e-073924c856af.jpg](Nand flash/b81dc059-2002-4e2b-932e-073924c856af.jpg)](Nand flash/b81dc059-2002-4e2b-932e-073924c856af.jpg)

acquire release ???????

??

1. 在单线程场景中，CPU 通常会保证程序顺序（**Program Order**）的可见性，即单线程内的指令会按照代码编写的顺序执行（或看起来像是按顺序执行）存储器读写的结果也会符合单线程的预期
    1. 即使 CPU有乱序功能，也会通过 scoreboard等方式来处理 data hazard，address hazard 等，确保单线程内的内存访问都是保续的。即使现代的 CPU都是超标量处理器。
2. 但在 多线程或多处理器（**multi-hart**）场景中，要实现多线程同时正确的对一个内存操作就会遇到问题
    1. 乱序执行（Out-of-Order Execution）
    2. 缓存分层（Cache Hierarchy），
    3. 存储缓冲区（Store Buffer）**，**CPU有各级的存储器 cache，in-flight指令暂存器
    4. 其他线程观察到的实际操作顺序与当前线程的程序顺序不一定一致
    5. 希望实现一个功能，利用一片多核都可以访问到的存储器来实现同步，这个存储器实现 **“锁”**功能
        1. 处理器 1，通过读“锁”，确定没人用，写“锁”，确保占用，不被别人使用
        2. 在处理器，读-写中间，处理器 2可能就会写“锁”，从而造成错误
        3. 处理器 1，准备通过写“锁”，来释放占用。但是写操作可能被乱序到当前线程的前面去执行，从而达不到保护的目的。
    6. 所以希望实现“原子指令”，实现获取“锁”，和释放“锁”的中间不会被其他的线程干扰，产生错误。
3. 在多线程间建立“同步关系”，通过限制 CPU 的重排序和缓存行为，确保
    1. 使用 `acquire` 和 `releasey` 语义，确保单线程内，和 LR.W 和 SC.W 指令之间是保序的。单线程内的指令实际执行顺序能被 LR.W 和 SC.W 控制，不收缓存和乱序的影响。
    2. 然后通过 LR.W 和 SC.W 指令，通过特殊的多线程都唯一的硬件，实现多线程之间的竞争。
4. 从而确保
    1. 一个线程的写操作结果能被另一个线程的读操作正确感知
    2. 跨线程的操作顺序符合预期（避免“先写后读”变成“先读后写”的错乱）

`LR.W`(Load-Reserved)??

????????

????????

1. 加载值：从内存地址 `rs1` 读取 32位数据到寄存器 `rd`。
2. 设置保留标记：
    - 在缓存子系统（如 L1 Cache）中标记该地址为 **Reserved**（保留状态）。
    - 硬件会记录当前核心（Hart）对该地址的独占访问权。
3. 隐式 **acquire**语义（若未显式指定）：

    - 后续内存操作不会被重排序到`LR.W`之前（保证加载操作的可见性）。

??????????

- 缓存行状态：目标地址对应的缓存行会被置为 **Exclusive** 或 **Modified** 状态（取决于一致性协议，如MESI）。
- 保留标记的存储：
    - 通常由缓存控制器维护一个 **Reservation Register**（保留寄存器），记录保留地址和核心ID。
    - 其他核心的写操作会清除该地址的保留标记（通过缓存一致性协议广播Invalidate）。

??`SC.W`(Store-Conditional)??

????????

????????

1. 检查保留标记：
    - 若地址`rs1`的保留标记仍被当前核心持有（未被其他核心修改）：
        - 执行存储：将`rs2`的值写入`rs1`。
        - 清除保留标记。
        - 向`rd`写入`0`（成功）。
    - 若保留标记已失效（其他核心修改了地址`rs1`）：
        - 放弃存储。
        - 向`rd`写入`非0值`（通常为1，失败）。
2. 隐式 **release**语义（若未显式指定）：
    - 之前的内存操作不会被重排序到`SC.W`之后（保证存储操作的全局可见性）。

??????????

- 原子性保证：
    - 通过缓存一致性协议（如MESI）锁定缓存行，确保“检查-存储”的原子性。
    - 若其他核心发起写请求，缓存控制器会清除保留标记并响应Invalidate。
- 总线事务：
    - 成功时生成 **Atomic Write** 总线事务。
    - 失败时不发起存储请求。

????????????

```
# 加锁（使用aq/rl语义确保屏障）
lock:
  lr.w.aq t0, (a0)        # 带acquire的加载保留
  bnez    t0, lock        # 检查锁是否被占用（t0!=0则重试）
  li      t1, 1
  sc.w.rl t0, t1, (a0)    # 带release的条件存储
  bnez    t0, lock        # 若存储失败（t0!=0），重试
```

```
# 解锁
unlock:
  sw.rl   zero, (a0)     # 带release的存储，释放锁
```

??????????

```
Core 0                          Core 1
======                          ======
1. lr.w t0, (x)                3. lr.w t0, (x)
   - 加载x=0，设置保留标记          - 加载x=0，设置保留标记
2. sc.w t1, 1, (x)             4. sc.w t1, 1, (x)
   - 保留有效，存储成功（x=1）       - 保留已失效（因Core 0修改x），存储失败
   - t1=0                          - t1=1
```