

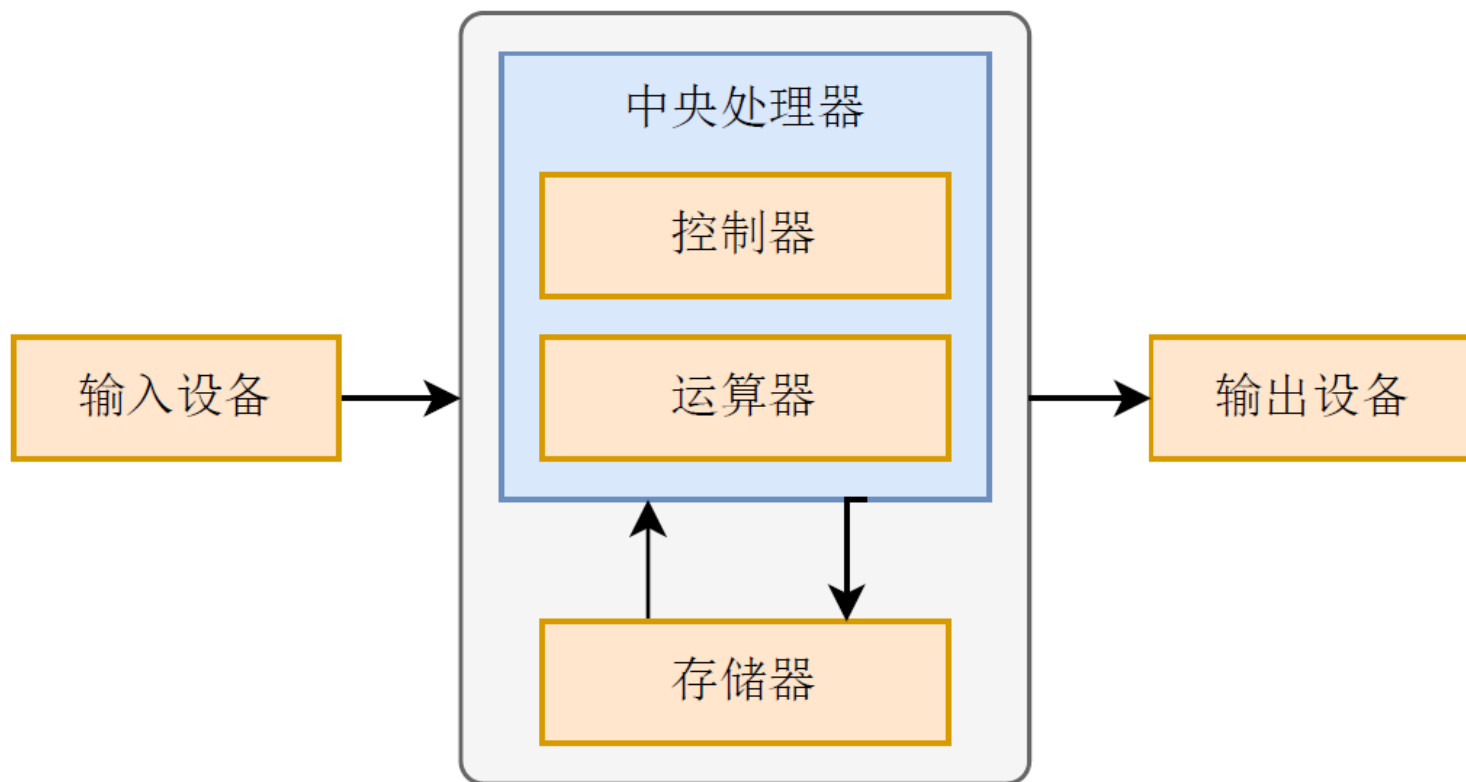
存储器优化

郭健美

2024年秋

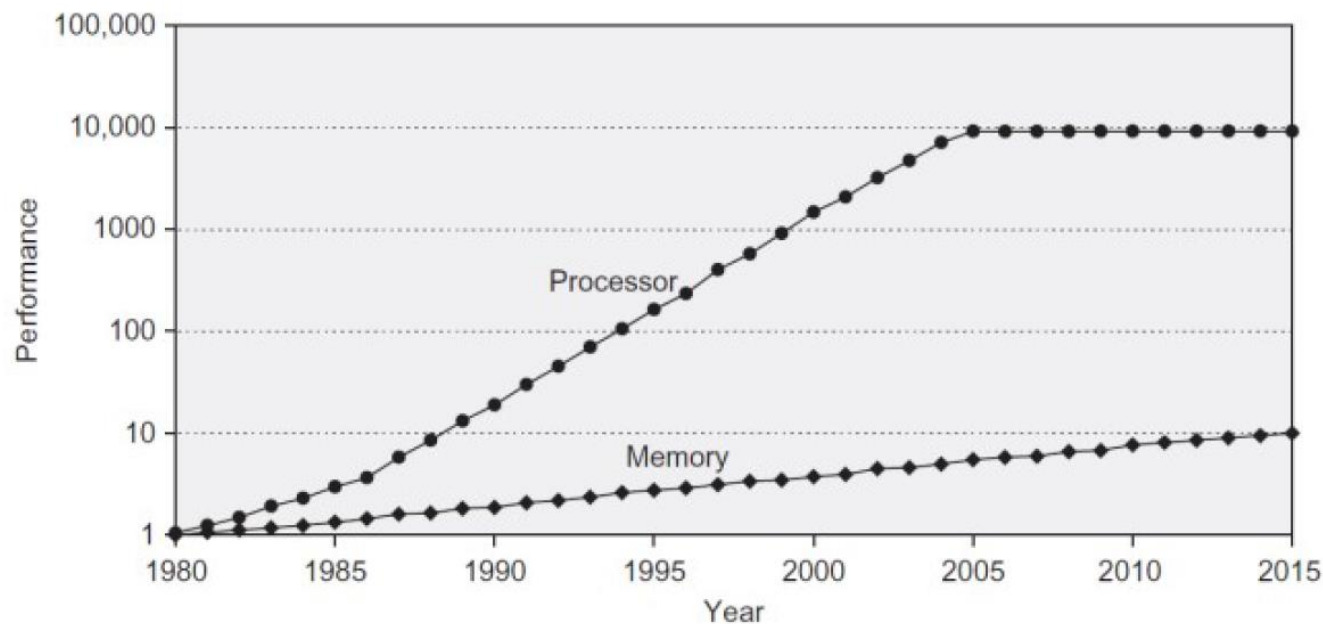
冯·诺依曼体系结构 / 普林斯顿体系结构

- 指令和数据都存在同一个存储器中



处理器与内存性能的“剪刀差”

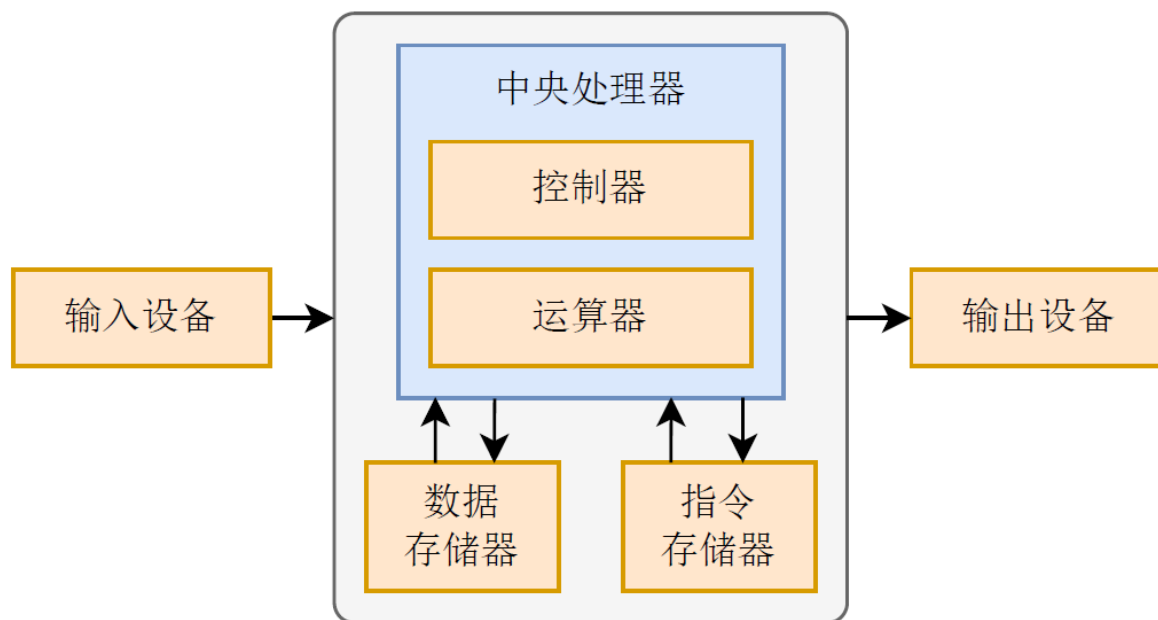
- 处理器与内存性能不匹配的问题造成了有名的**冯·诺依曼瓶颈**
- 如何解决？
 - 更先进的存储技术
 - 计算机体系结构的设计优化



[J. L. Hennessy, D. A. Patterson: Computer Architecture - A Quantitative Approach, 6th Edition. Morgan Kaufmann, 2017]

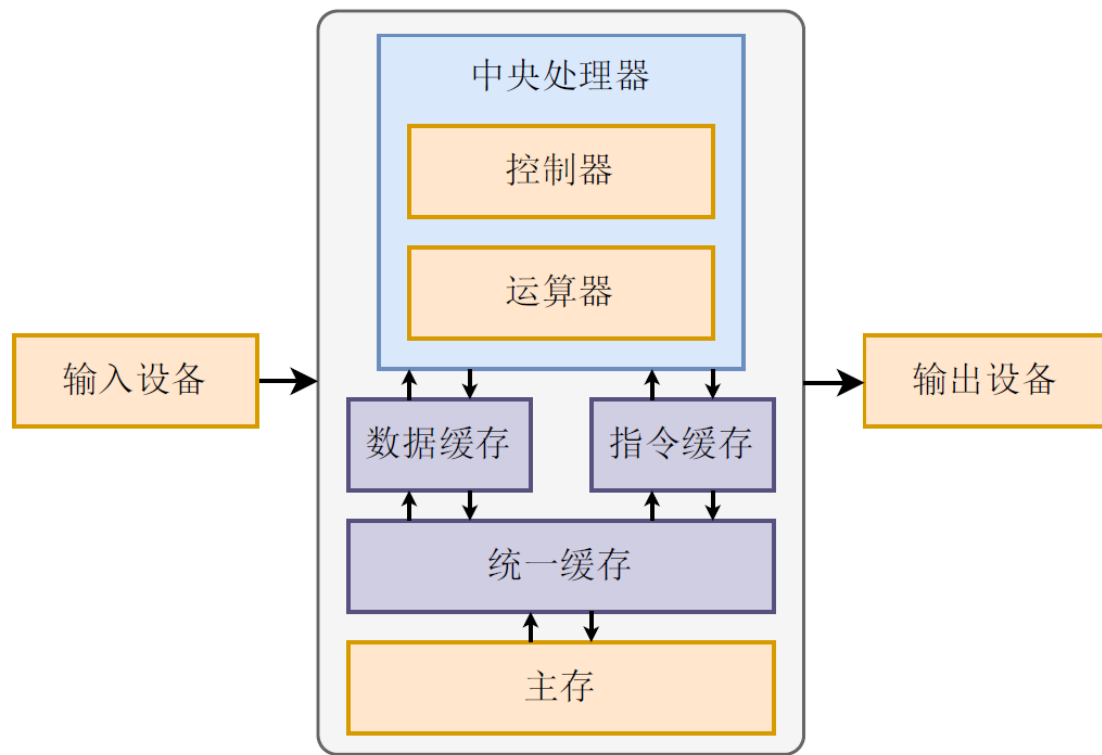
哈佛体系结构

- 将指令存储与数据存储完全分离，指令和数据各自拥有独立的地址空间和独立的总线
- 缓解了存储器带宽争抢的问题，但设计复杂性高、成本高



混合体系结构

- 采用多级高速缓存：L1 cache、L2 cache、L3/LL cache
- 兼具冯·诺依曼体系结构与哈佛体系结构的特性：分离的 L1 cache + 统一的 L2/L3 cache



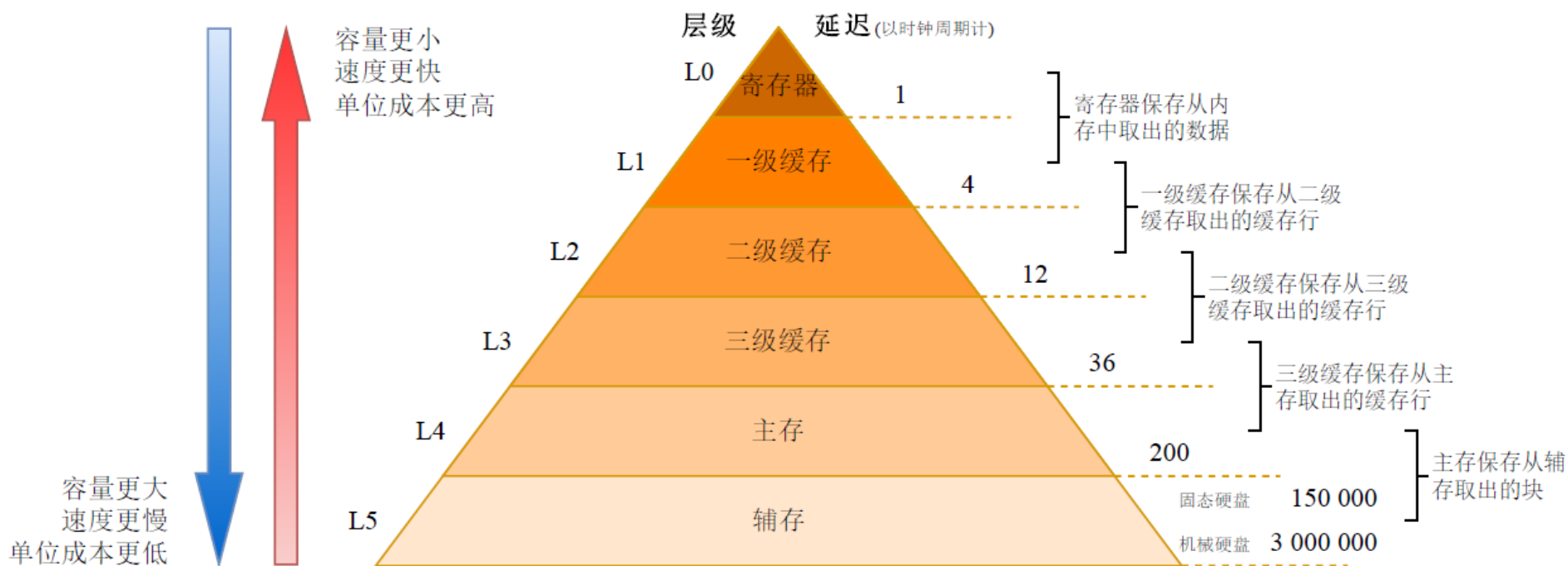
本课内容

- **高速缓存**
- 多核访存架构
- 编写缓存友好的代码

存储器层次结构

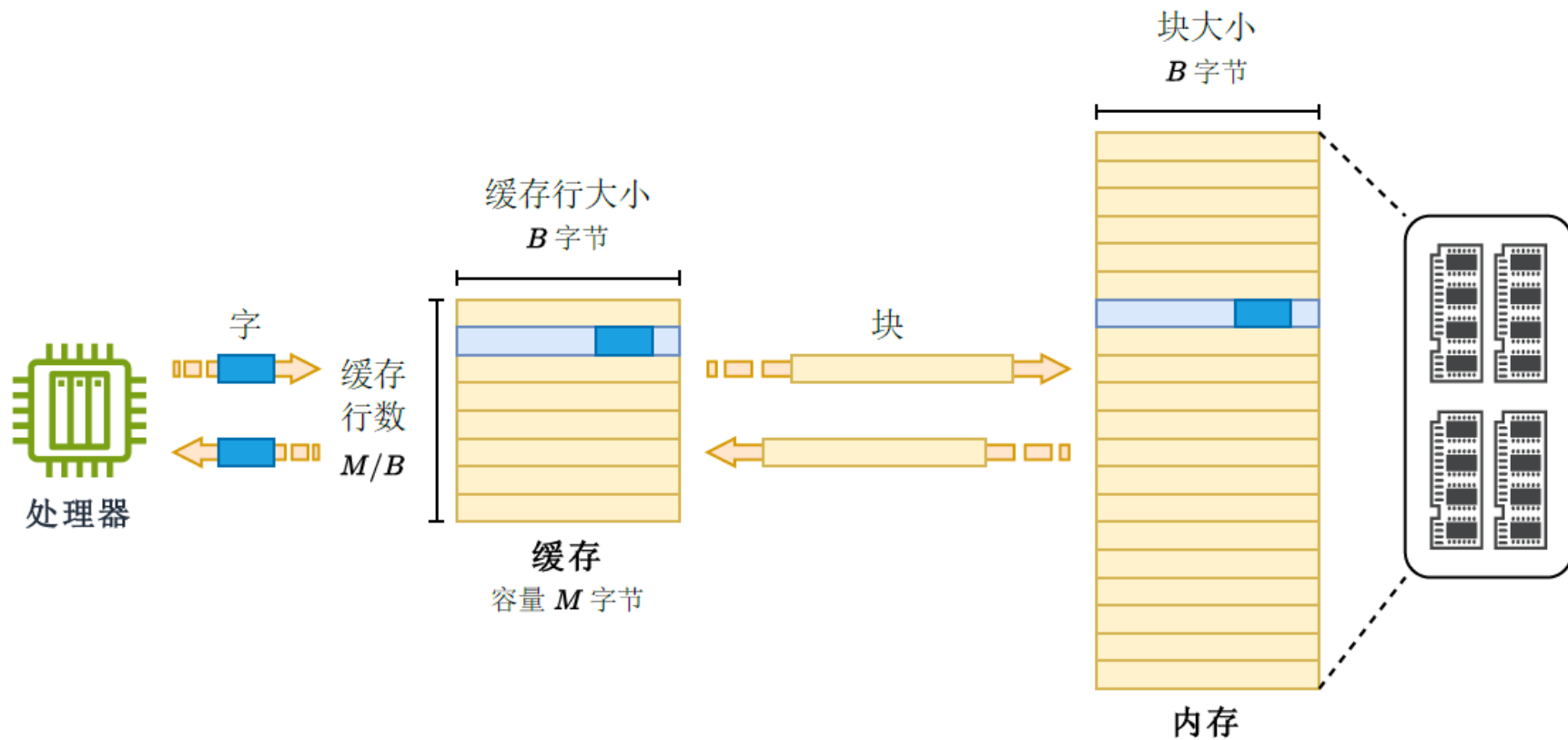
- 性能优化的核心思想

- 用位于高层级的、更小但更快的存储设备作为位于低层级的、更大但更慢的存储设备的缓存
- 当处理器需要访问数据时，应尽可能地使用高层级中被缓存的数据，从而降低访问低层级存储器的次数，以提高性能



高速缓存的组织结构

- 一个简化的模型



高速缓存的组织结构

当处理器发出访存请求时，

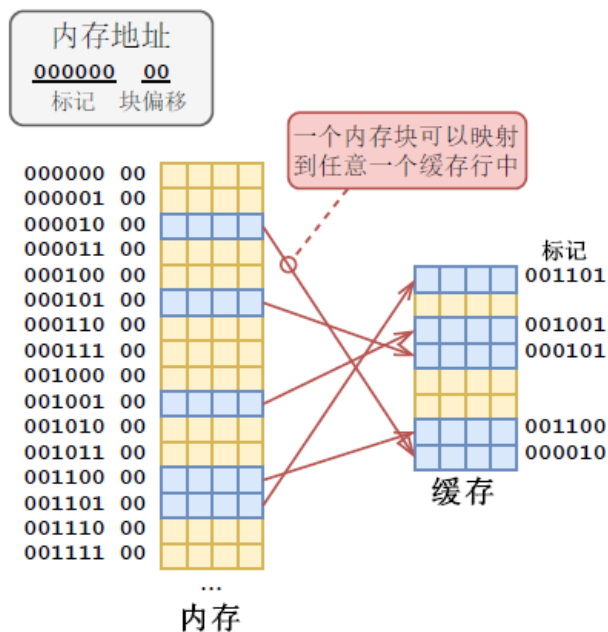
- **缓存命中 (hit)**：要访问的数据所在的内存块已经在缓存中了
- **缓存未命中 (miss)**：要访问的内存块不在缓存中，需要从内存中取出相应的块并放入缓存中进行访问
 - **强制性未命中 (compulsory miss) / 冷未命中 (cold miss)**：初始状态下缓存是空的 / 冷的，对任何内存块的访问都不会命中
 - **冲突未命中 (conflict miss)**：多个内存块映射到同一个或同一组缓存行，由于它们之间的竞争或冲突会导致缓存行的未命中和被替换
 - **容量未命中 (capacity miss)**：程序一段时间访问内存块的集合称之为工作集 (working set)，当工作集的大小超过缓存大小时，会出现容量未命中，即缓存不足以容纳整个工作集
 - **一致性未命中 (coherence miss)**：在多处理器系统中，每一个处理器核心都会拥有自己独立的缓存，为了保证数据一致性，当一个核心修改了共享数据，将导致其他核心的缓存数据失效，此时其他核心发出的对该数据的访问请求将会未命中

高速缓存的组织结构

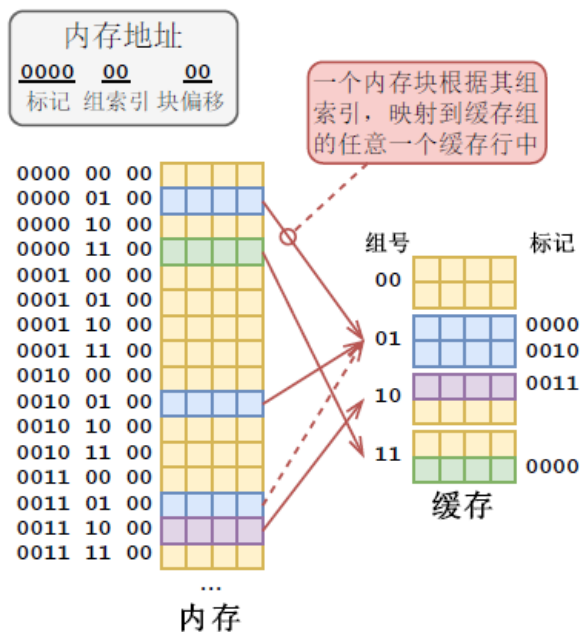
- **缓存替换策略**：当发生缓存未命中时，如果缓存已经满了，会有其它缓存行被替换
 - 例如，最近最少使用（Least Recently Used, LRU）策略
- **缓存放置策略**：当发生缓存未命中后，要访问的内存块将会被放置到缓存中
 - 缓存放置策略与缓存的组织方式有关

高速缓存的组织结构

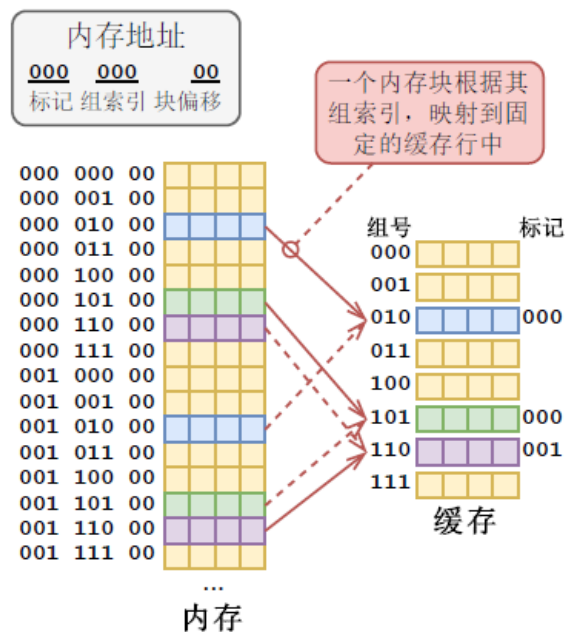
- **全相联**：整个缓存的所有缓存行就是一组，那么一个内存块可以放置到整个缓存中任意一个缓存行
- **k路组相联**：将整个缓存的所有缓存行分为若干组，每个组有k个缓存行。这里，k表示相联度
- **直接映射**：每个缓存行就是一个缓存组，一个内存块根据其地址将会被放置到一个特定的缓存行



(a) 全相联缓存



(b) 组相联缓存



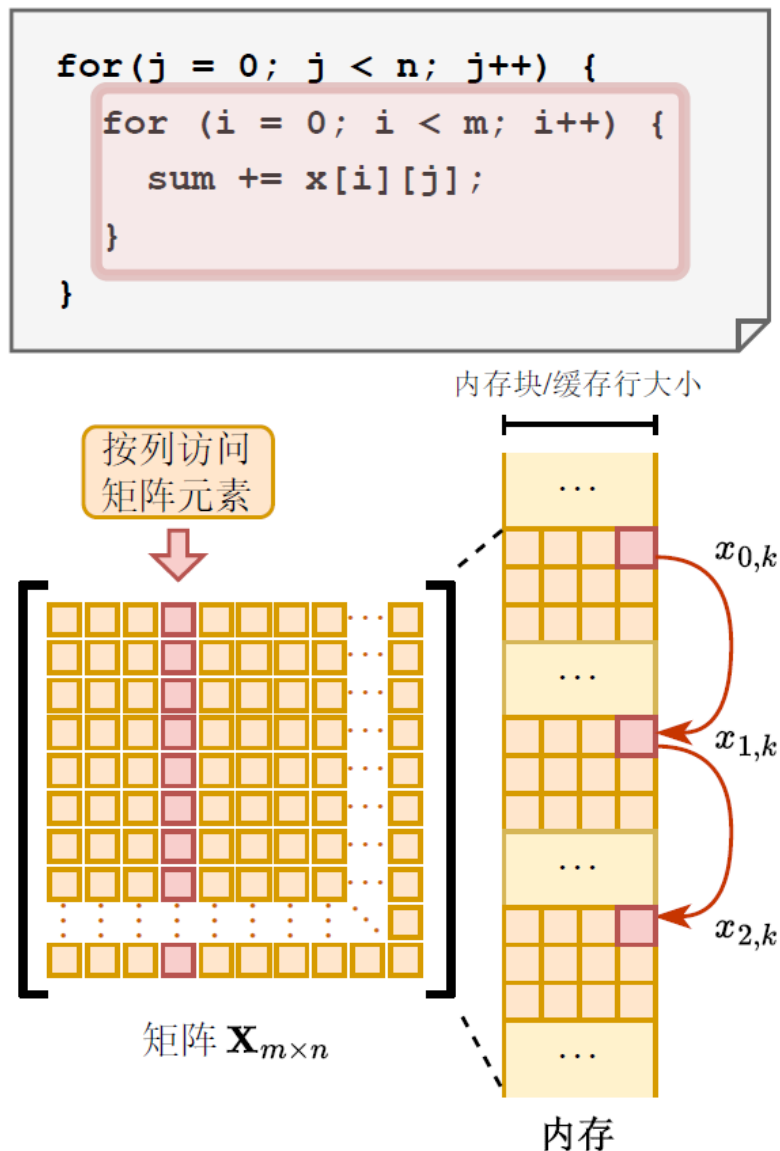
(c) 直接映射缓存

缓存预取

- 在流水线执行的处理器中，缓存未命中是导致流水线停顿的重要原因之一，因此，提高缓存命中率是优化程序性能的关键
- **预取（prefetching）**：提前将可能需要访问的指令或数据放到缓存中，而非等到缓存未命中的时候
 - 原理：在执行访存请求时，若缓存命中，则会很快地完成访存请求；若缓存未命中，则需要消耗额外的时间访问内存，这段额外的时间称为未命中惩罚（miss penalty）。如果在流水线中能提前充足的时间就发出预取请求，那么未命中惩罚的时间就可以在很大程度上被隐藏
 - 实现方式
 - 硬件预取
 - 软件预取

硬件预取

- 由硬件预取器实现（通常在BIOS中设置），会监测过去一段时间内存的访问，识别常见的、预先定义的访问模式
- 通过消耗部分内存带宽以获得更低的内存访问延迟



软件预取

- 通过在程序代码中手动插入预取指令执行预取
- 软件预取则给了程序员更大的灵活性，但也对程序员提出了更高的要求，需要确定合适的时间预取合适的数据

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++) {  
        prefetch(&x[i + 1][j]);  
        sum = sum + x[i][j];  
    }  
}
```

本课内容

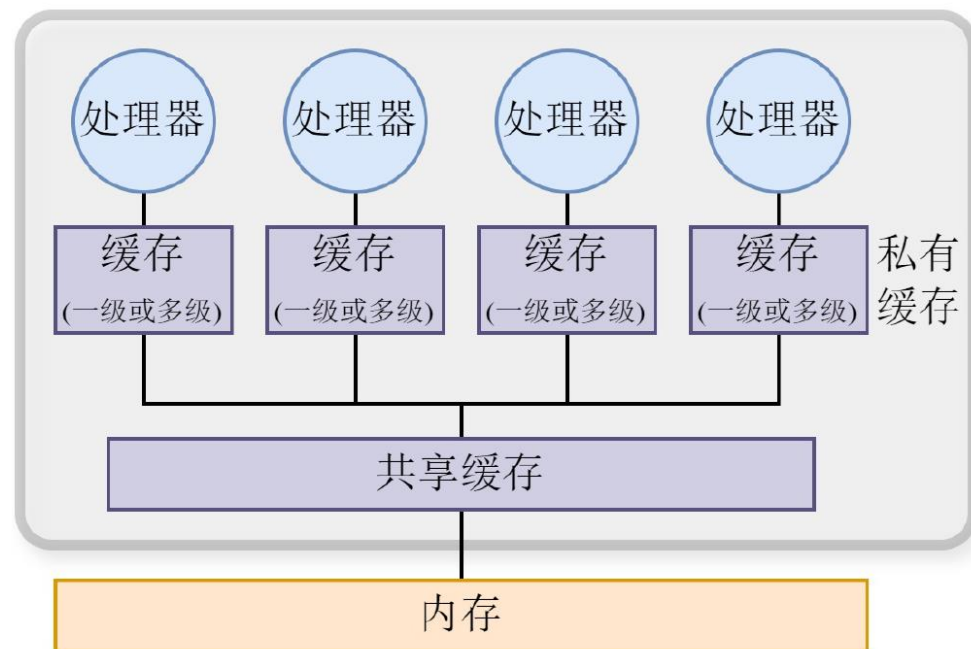
- 高速缓存
- **多核访存架构**
- 编写缓存友好的代码

多处理器系统架构

- 多处理器（multiprocessor）系统架构是一种具有多个处理器的计算机体系结构设计
- 每个处理器可以是单核的，也可以是多核的
- 多处理器系统通常由单个操作系统控制，通过共享地址空间来实现内存的共享
 - 集中式
 - 分布式

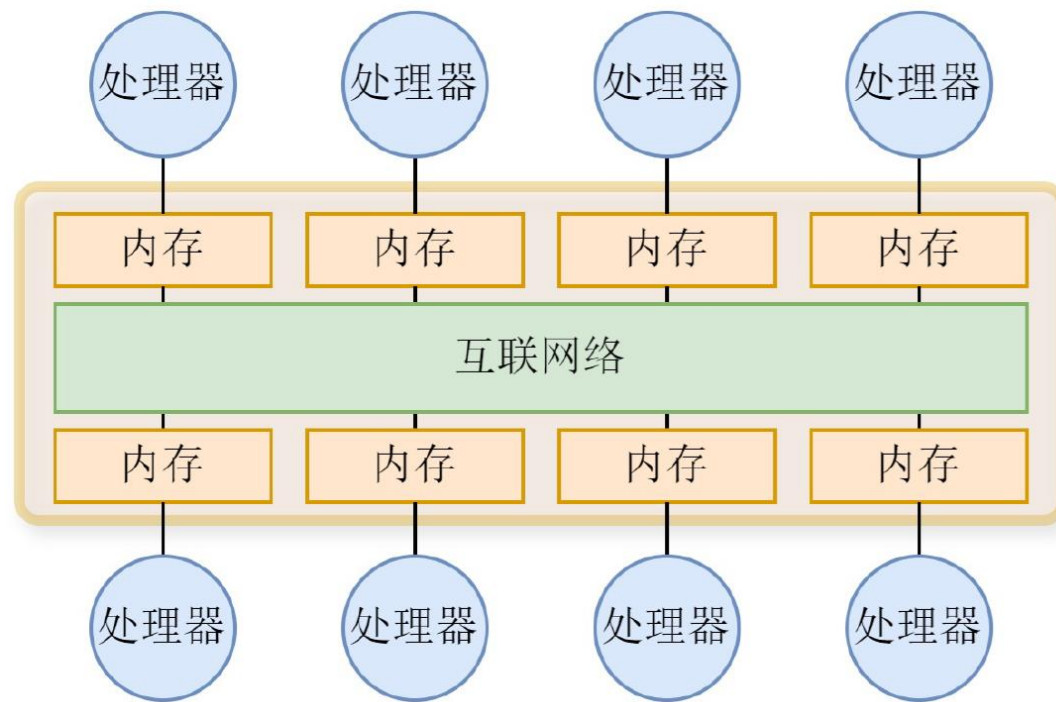
集中式共享内存（Centralized Shared Memory, CSM）架构

- 所有处理器连接并共享一个集中式内存，并可以对等地访问它
- 这种访存模式也称为**一致内存访问（Uniform Memory Access, UMA）**，这里的“一致”指的是所有处理器访问内存的延迟是一致的
- 一种常见实现是**对称多处理（Symmetric Multi-Processing, SMP）系统**，即每个处理器有对称的地位和权限，可以独立执行任务，并且共享系统资源



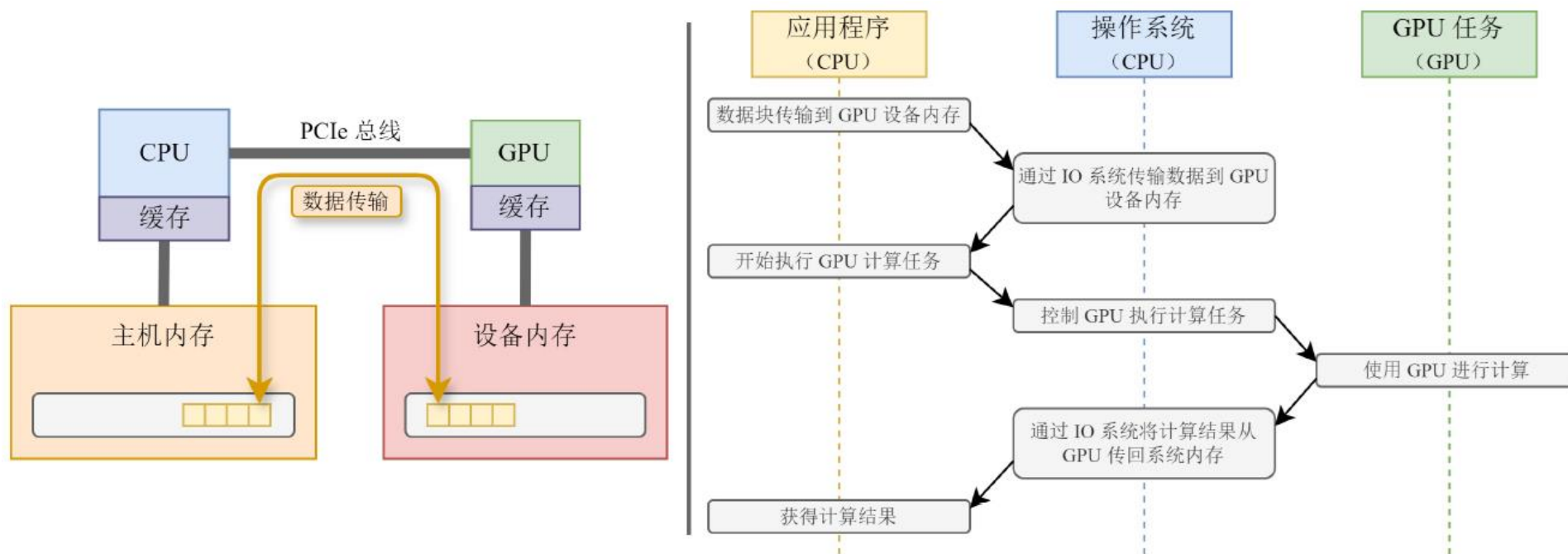
分布式共享内存（Distributed Shared Memory, DSM）架构

- 物理内存分布在各个处理器上，通过互联网络相互连接，被所有处理器共享访问
- 处理器访问本地直连内存的延迟通常明显低于访问远端通过互联网络连接内存的延迟，这种访存模式也称为**非一致内存访问（Non-Uniform Memory Access, NUMA）**
- 一种普遍实现是缓存一致的**非一致内存访问（cache coherent Non-Uniform Memory Access, ccNUMA）**系统，通过专门的硬件保持缓存中数据的一致性



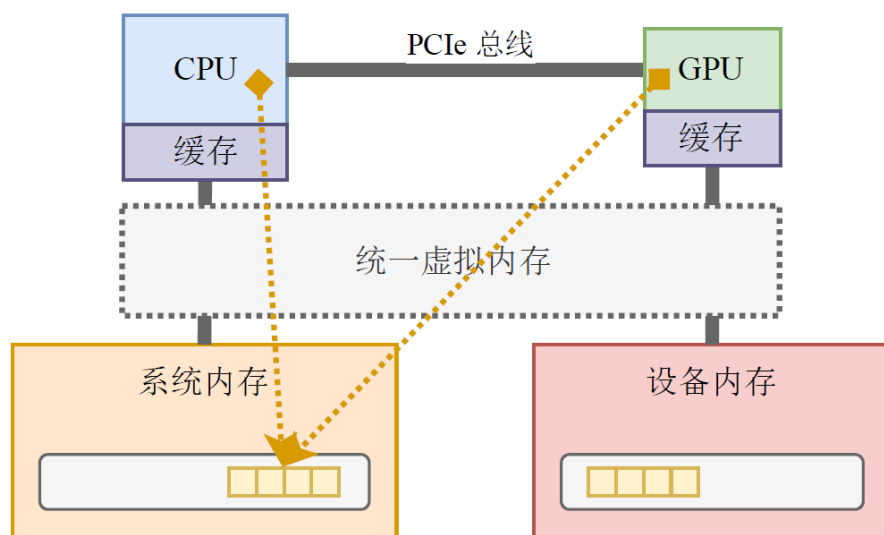
异构系统架构 (Heterogeneous Systems Architecture, HSA)

- 同时集成多种异构计算单元的系统架构
 - CPU + GPU / FPGA / DSP / ASIC / DPU / NPU

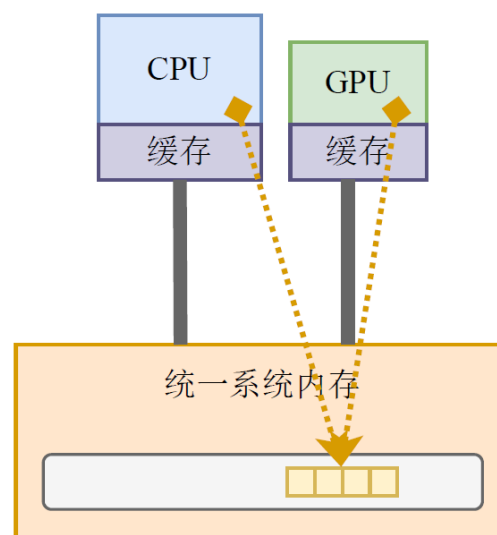


异构系统的内存访问

- 对于异构系统上的应用程序，程序员需要手动管理主机内存和设备内存
- 为了简化编程模型，许多异构系统采用了**统一内存访问（unified memory access）**模型
 - **统一虚拟内存**：例如，在软件层面，用 Intel oneAPI 申请一块统一虚拟内存
 - **统一系统（物理）内存**：例如，AMD Accelerated Processing Units（APU）的硬件实现



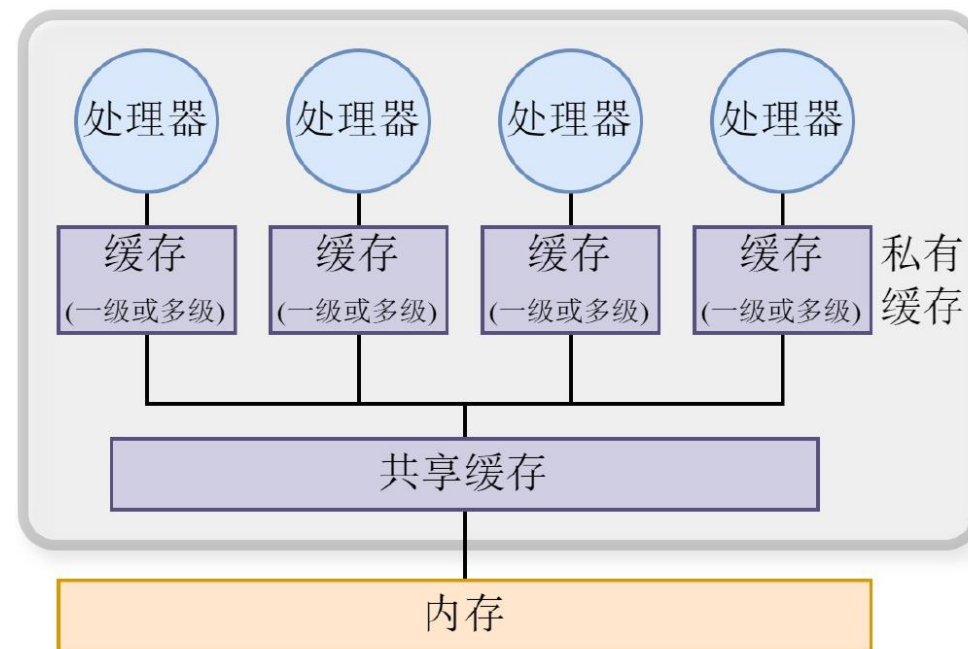
(a) 统一虚拟内存



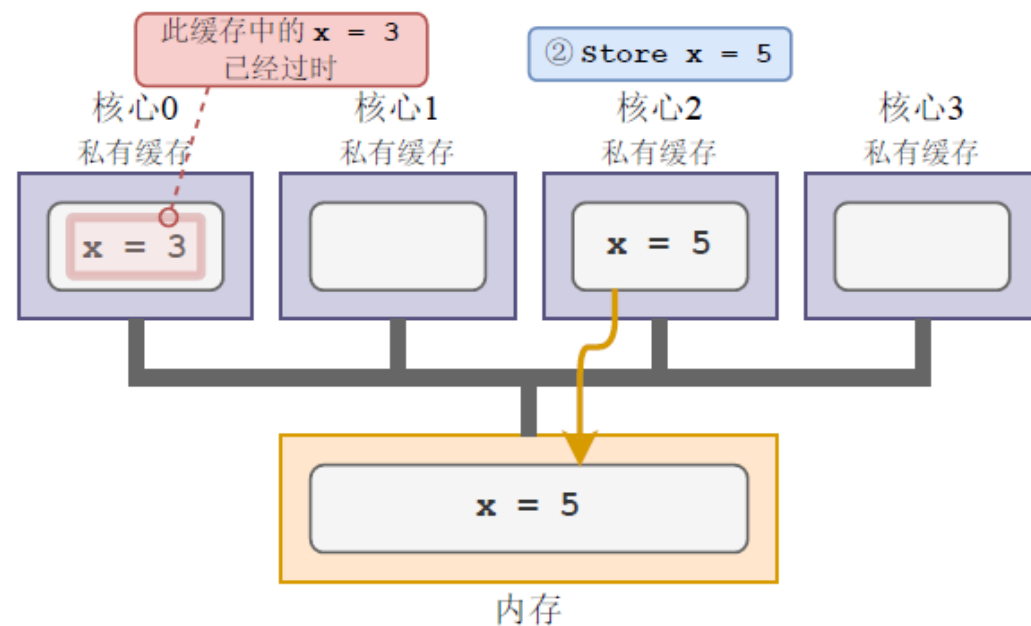
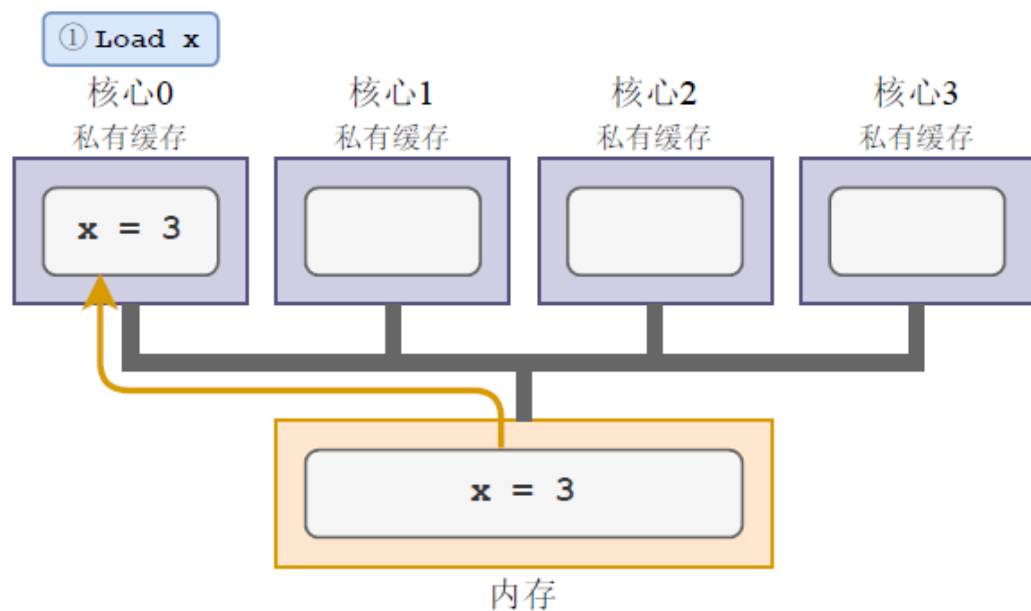
(b) 统一系统（物理）内存

缓存一致性

- 对于单片多处理器 CMP 系统，假设每个处理器都是单核的，且每个处理器核心都有各自独立的私有缓存
- 对于共享的任意一个内存块，其副本可能存在于多个核心的私有缓存中，在这种情况下，多个副本可能会存在缓存数据不一致的问题



缓存不一致的示例



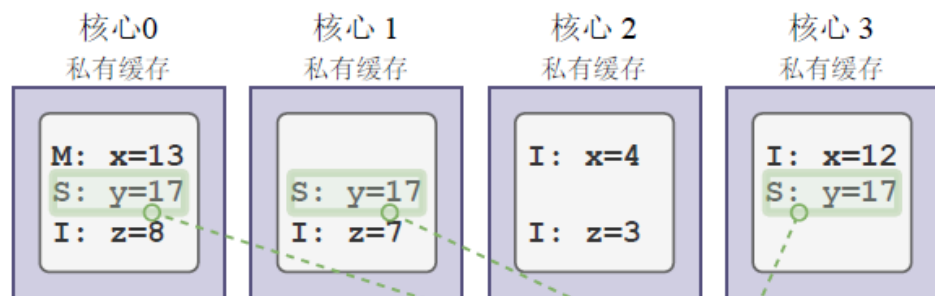
缓存一致性协议

- 确保不同处理器或核心看到的共享数据是一致的
- 同一内存地址的数据在缓存中应保持一致的状态
 - 读操作的一致性：当一个处理器或其他组件从内存中读取数据时，要确保它获取的是最新的数据，而不是过时或失效的数据。
 - 写操作的一致性：当一个处理器或其他组件向内存中写入数据时，要确保这个写操作被同步到所有相关的缓存中，以避免其他处理器或组件读取到过期的数据

MSI 缓存一致性协议

- Modified (M) : 当某个处理器写入数据到一个缓存行时, 该行被标记为M 状态。这表示该处理器拥有对这个缓存行的独占写权限, 并且缓存数据与内存中的数据不一致, 需要写回到相应的内存块中。此时其他处理器缓存中对应相同数据的缓存行不会处于M 或S 状态。
- Shared (S) : 当某个处理器读取一个缓存行的数据, 而该数据在其他处理器的缓存中也存在时, 这个缓存行被标记为S 状态。此时多个处理器的缓存共享相同的数据。
- Invalid (I) : 当某个处理器写入数据到一个缓存行时, 其他处理器缓存中对应相同数据的缓存行被标记为I 状态。这表示其他缓存中的数据已经过时, 需要从新写入的缓存或内存重新加载数据。

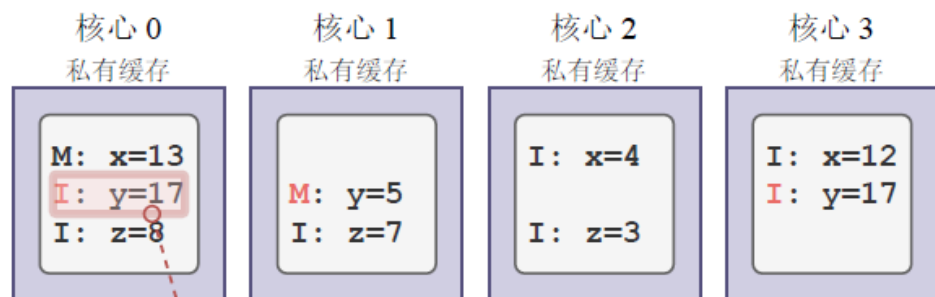
MSI 协议的应用示例



初始状态时，部分处理器从内存中读取同一个数据，这些缓存行被标记为 S 状态

(a)

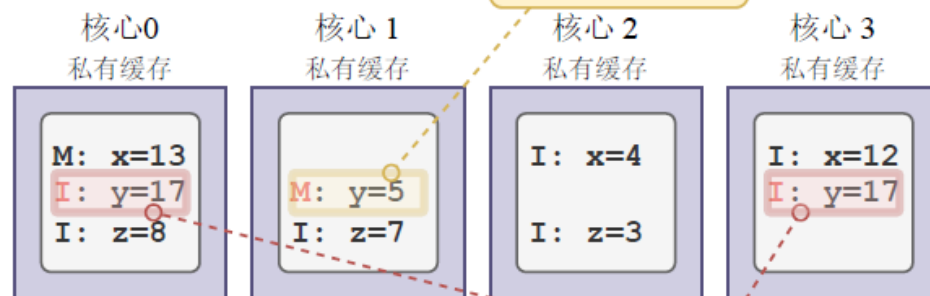
执行 Load y



由于该行为 I 状态，将发生缓存未命中

(c)

执行 Store y = 5

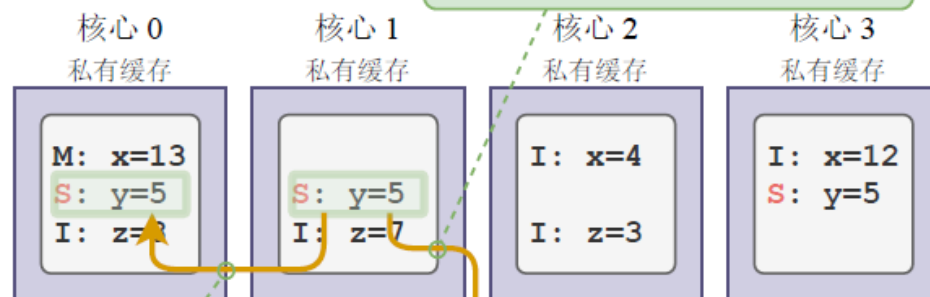


① 写入缓存行后，该行被标记为 M 状态

② 其他处理器缓存的同一行被标记为 I 状态，表示已经过时

(b)

执行 Load y



① 其他处理器标记为 M 状态的同一缓存行写回内存，该缓存行标记为 S 状态

② 从其他核心的缓存中重新加载该缓存行，并标记为 S 状态

内存

(d)

其它缓存一致性协议

- MESI (Modified, Exclusive, Shared, Invalid)
- MOESI (Modified, Owned, Exclusive, Shared, Invalid)
- VI (Valid, Invalid)
- ...

缓存一致性未命中

- 当一个处理器修改了共享数据，而导致其他处理器的缓存数据失效
- 并行程序性能问题的主要成因之一，也是性能优化的关键点
- 两种类型
 - 真共享未命中 (true sharing miss)
 - 伪共享未命中 (false sharing miss)

真共享未命中

- 多核处理器修改缓存行上相同变量的数据造成的一致性未命中

```
1 int sum = 0;
2 #pragma omp parallel for num_threads(4) reduction(+: sum)
3 for (i = 1; i < n; i++)
4     sum += a[i];
```

伪共享未命中

- 多个处理器访问的是相同缓存行上的不同数据
- 注意，现代处理器缓存行大小通常是 64 字节，不同的变量可能会存放在同一个缓存行中
- 伪共享未命中应该消除！
- 可以通过 Intel VTune、Linux perf 等性能分析工具进行检测

```
1 struct S {
2     int sumA;
3     int sumB;
4 } s;
5 ...
6 #pragma omp parallel
7 {
8     #pragma omp sections
9     {
10         #pragma omp section
11         {
12             for (int i = 0; i < n; i++)
13                 s.sumA += a[i];
14         }
15         #pragma omp section
16         {
17             for (int i = 0; i < n; i++)
18                 s.sumB += b[i];
19         }
20     }
21 }
```

本课内容

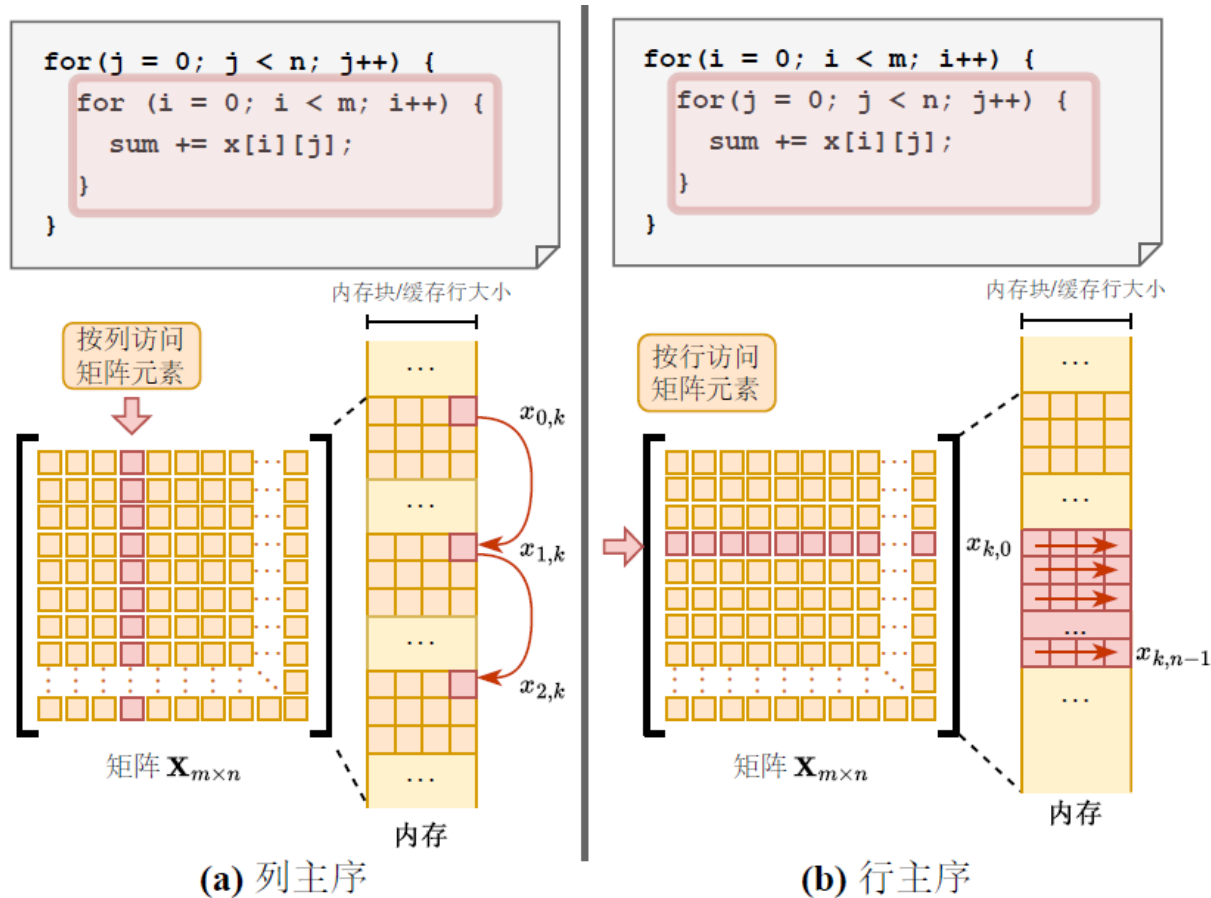
- 高速缓存
- 多核访存架构
- **编写缓存友好的代码**

编写缓存友好的代码

- 关键在于代码具有良好的局部性，尽可能利用高速缓存获取数据
- **时间局部性**：当访问某个内存位置时，在不久的将来很可能会再次访问同一位置。理想情况下，我们总是希望下次需要时，缓存中仍保留这些数据
- **空间局部性**：当某个内存位置被访问时，附近的位置很可能在不久的将来也会被访问。通常，缓存行或内存块的大小大于程序需要读取的变量大小，当程序从内存中读取一个数据时，其相邻的数据也会被放在缓存行中。当程序需要这些相邻的数据时，缓存中就有这些数据了

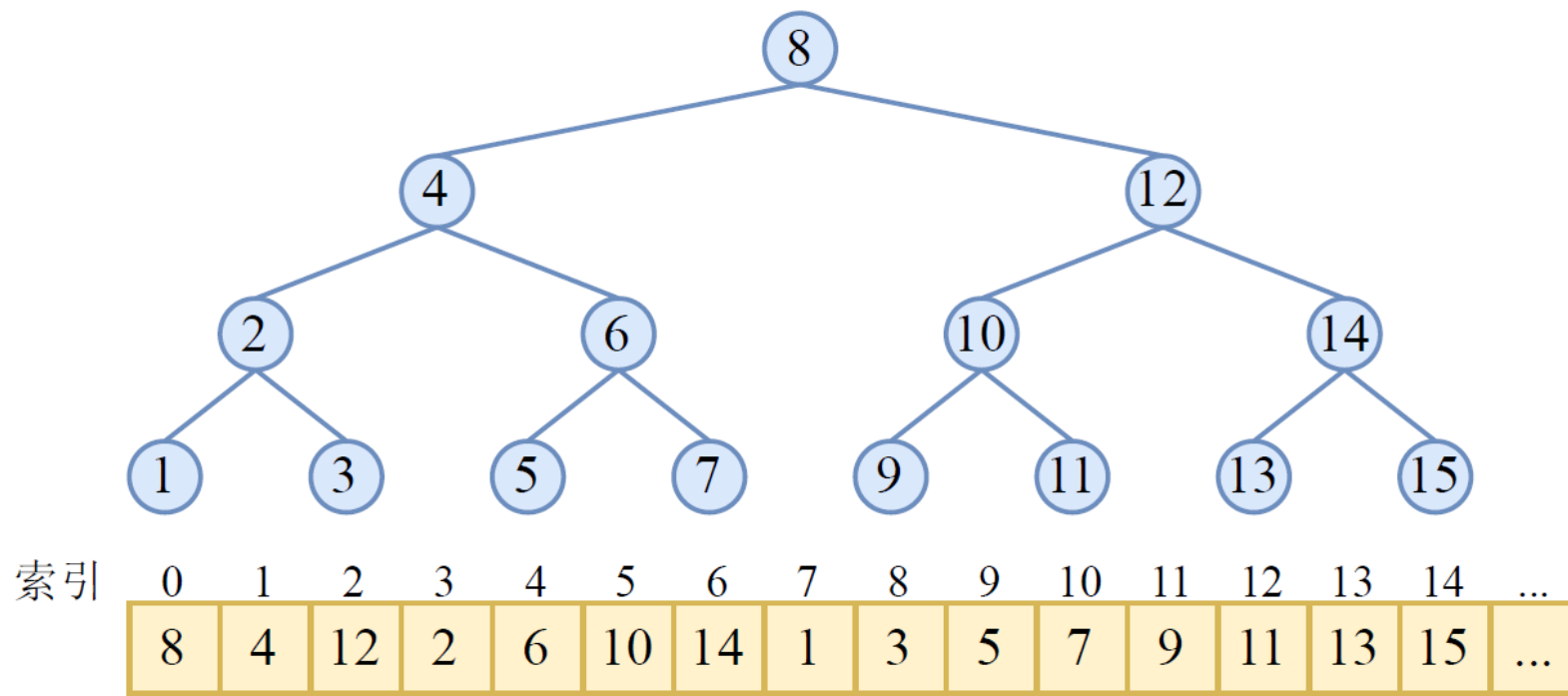
顺序访问数据

- 利用空间局部性原理的最佳方法



顺序访问数据

- 除了调整数据访问顺序以外，还能够通过调整数据结构的存储方式，以提升数据访问时的空间局部性
- 例如，二叉查找树的 Eytzinger 布局




数据打包 (packing)

- 将数据按照一定规则整理、组织和压缩的过程，以提高数据的存储效率、传输效率或者处理效率
- 例如，使用位域 (bitfields)

```
1 struct S {  
2     unsigned a;  
3     unsigned b;  
4     unsigned c;  
5 } // 结构体 S 的大小是 sizeof(unsigned) * 3 字节
```

```
1 struct S{  
2     unsigned a:4;  
3     unsigned b:2;  
4     unsigned c:2;  
5 } // 结构体 S 的大小仅为 1 字节
```

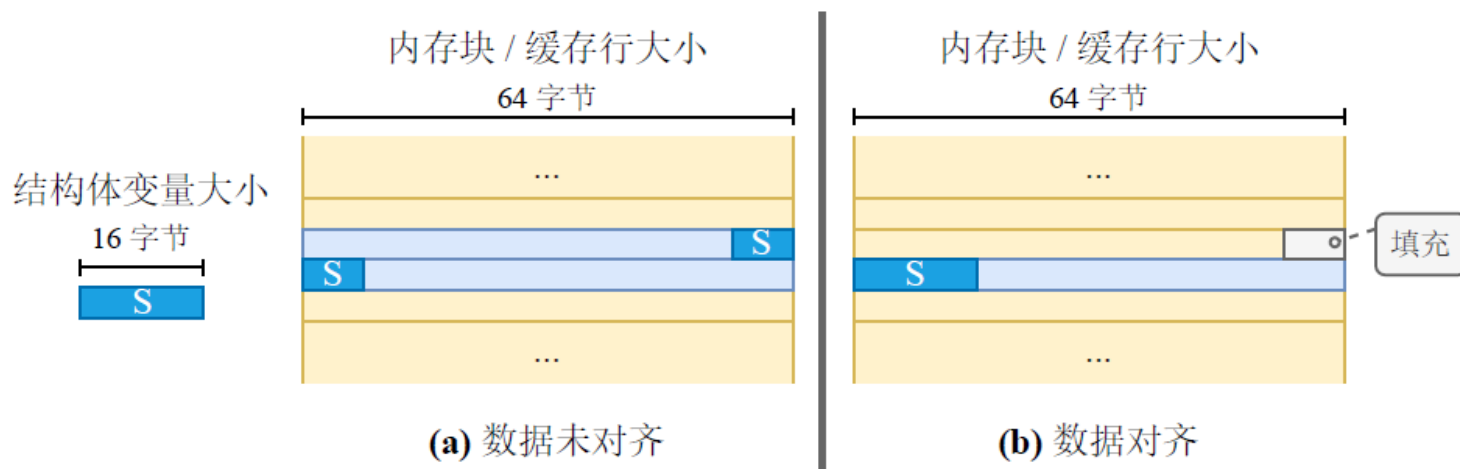


数据对齐 (aligning) 与填充 (padding)

- 如果变量存储在一个可被变量大小整除的内存地址上，那么访问变量的效率会较高
- 例如，一个双精度浮点型 (double) 变量需要 8 字节的存储空间，因此，最好将其存放在能被 8 整除的地址处，这样的对齐称为 8 字节对齐

数据对齐与填充

- 使用填充使得缓存行中数据对齐
- 对齐和填充会使得部分字节未被使用，形成空洞（hole）



- 在 C++ 中，可以使用 alignas


```
1 alignas(16) int16_t a[N];  
2  
3 struct alignas(64) S {  
4     ...  
5 }
```

数据对齐与填充

- 对齐与填充可以用于两个变量之间，以避免缓存争用或伪共享（false sharing）等

```
1 struct S {  
2     int a;           // 被线程 A 写入  
3     int b;           // 被线程 B 写入  
4 }
```

```
1 struct S {  
2     int a;           // 被线程 A 写入  
3     alignas(64) int b; // 被线程 B 写入  
4 }
```



小结

- 为了优化冯·诺依曼瓶颈，根据局部性原理，现代计算机普遍采用高速缓存构成存储器层次结构，其性能优化的核心思想是用位于高层级的、更小但更快的存储设备作为位于低层级的、更大但更慢的存储设备的缓存
- 缓存未命中是导致流水线停顿的重要成因之一，提高缓存命中率是优化程序性能的关键，具体手段包括：软/硬件预取，以及顺序访问数据、数据打包、对齐与填充等程序代码层面的优化方法
- 多处理器系统包括集中式和分布式两种共享内存架构，它们分别对应一致内存访问和非一致内存访问两种模式，同时，系统需要保证缓存一致性
- 异构系统需要程序员手动管理主机内存和设备内存，也可以采用统一内存访问模型来简化编程