并行处理系统

并行处理系统概述

多处理器系统

多计算机系统

并行处理编程模式简介

多处理器系统和多计算机系统简介

主　要　内　容

- 并行处理系统概述
  - 并行处理的主要技术问题
  - 并行处理系统的分类
- 多处理器系统
  - UMA多处理器结构
  - NUMA多处理器结构
  - 多处理器系统中的互连网络
  - 片级多处理器和多线程技术
  - 共享存储器的同步控制
- 多计算机系统
  - 集群多计算机系统
  - 网格多计算机系统

并行处理的主要技术问题

- 互连
  - 并行处理：将多个计算模块和存储模块互连，通过控制这些它们的并行工作来提高处理速度。
- 数据一致性
  - 在不同的计算模块中会设置共享高速缓存和主存，因此，在并行处理系统中存在复杂的数据一致性问题。
- 同步控制
  - 共享存储时，多个计算模块访问同一块数据需解决数据的互斥访问。
  - 分布计算和存储时，需要对各个模块进行同步控制。
- 任务划分
  - 大任务分解成子任务并行执行，包含算法分解和数据划分两个方面。
- 并行程序设计
  - 对系统中运行的程序进行并行化处理描述，以说明哪些处理逻辑段可并行执行、哪些有先后顺序关系，以及处理的数据可以怎样划分等。

并行处理的主要技术问题

- 资源调度和管理
  - 计算资源的调度和存储资源的管理，比在串行处理系统中复杂得多。
- 容错性和安全性
  - 预防由于单个节点失效可能带来的数据丢失、程序出错或系统崩溃等。
  - 要求系统必须考虑良好的可靠性设计、失效检测和恢复机制。
- 性能分析与评估
  - 并行处理性能通常用加速比来度量
    - 串行系统与并行系统中执行时间比值
    - 并行系统与串行系统的作业吞吐量比值
  - 评价并行系统的指标包括可用性、可扩展性、负载均衡、可靠性等

并行处理系统的分类

- 按指令和数据处理方式划分（Flynn分类）
  - SISD（单指令流单数据流）：指令级并行（超流水、超标量、动态调度）
  - SIMD（单指令流多数据流）：数据级并行（如Intel的MMX、SSE、AVX等）
  - MIMD（多指令流多数据流）：多计算机和多处理器系统
- 按地址空间的访问方式划分（MIMD）
  - 多计算机系统：计算节点有各自独立编址的存储器，通过消息传递方式访问其他节点的存储器，也称消息传递系统
  - 多处理器系统：各计算节点共享统一编址的存储器，可通过LOAD和STORE指令访问系统中的存储器，也被称为共享存储多处理器（Shared memory multiProcessor, SMP）系统
- 按访存时间是否一致划分（多处理器系统）　(Non-) Uniform Memory Access
  - 一致性访存（UMA）：每个处理器访存时间一致，通常共享一个存储器
  - 非一致性访存（NUMA）：处理器的访存时间不一致，与存储器位置有关
  - CC-NUMA：Cache-Coherent NUMA，支持cache一致性的NUMA

并行处理系统的分类

- 按处理单元的位置及其互连方式划分

下面先介绍多处理器系统

- 多核：一个CPU芯片中含多个（2、4、8等）核，共享LLC和主存 属多处理器
- 对称多处理器（Symmetric MultiProcessor, SMP）：相同类型CPU通过总线互连，并等同地位共享所有存储资源。即多个CPU对称工作。可见SMP就是一种UMA结构多处理器。PC、工作站和服务器等多采用SMP结构。
- 大规模并行处理机（MPP）：以专用内联网络连接数量众多处理单元而构成的并行计算系统。例如，可通过专用互连网络（如Mesh、交叉开关）将数量达几百甚至几千个的SMP服务器连接成MPP，SMP服务器之间协同工作，以完成同一个任务。
- 集群（Cluster）：以高速网卡将若干PC或SMP服务器或工作站连接而成的并行计算系统，其中每个节点有各自的独立编址的内存和磁盘，属于紧耦合同构多计算机系统（消息传递机制）。
- 网格（Grid）：以因特网等广域网将远距离分布的一组异构计算机系统连接而成的分布式并行处理系统，属于松耦合异构多计算机系统。 属多计算机
- 众核：一个GPU芯片中含几百个简单核，众多并行线程同时执行
- APU：CPU+GPU融合

回顾：并行处理系统的分类

- 按指令和数据处理方式划分（Flynn分类）
  - SISD（单指令流单数据流）：指令级并行（超流水、超标量、动态调度）
  - SIMD（单指令流多数据流）：数据级并行（如Intel的MMX、SSE、AVX等）
  - MIMD（多指令流多数据流）：多计算机和多处理器系统
- 按地址空间的访问方式划分（MIMD）
  - 多计算机系统：计算节点有各自独立编址的存储器，通过消息传递方式访问其他节点的存储器，也称消息传递系统
  - 多处理器系统：各计算节点共享统一编址的存储器，可通过LOAD和STORE指令访问系统中的存储器，也被称为共享存储多处理器（Shared memory multiProcessor, SMP）系统
- 按访存时间是否一致划分（多处理器系统）　　(Non-) Uniform Memory Access
  - 一致性访存（UMA）：每个处理器访存时间一致，也称为对称多处理器（SMP）
  - 非一致性访存（NUMA）：处理器的访存时间不一致，与存储器位置有关。
  - CC-NUMA：Cache-Coherent NUMA，支持cache一致性的NUMA

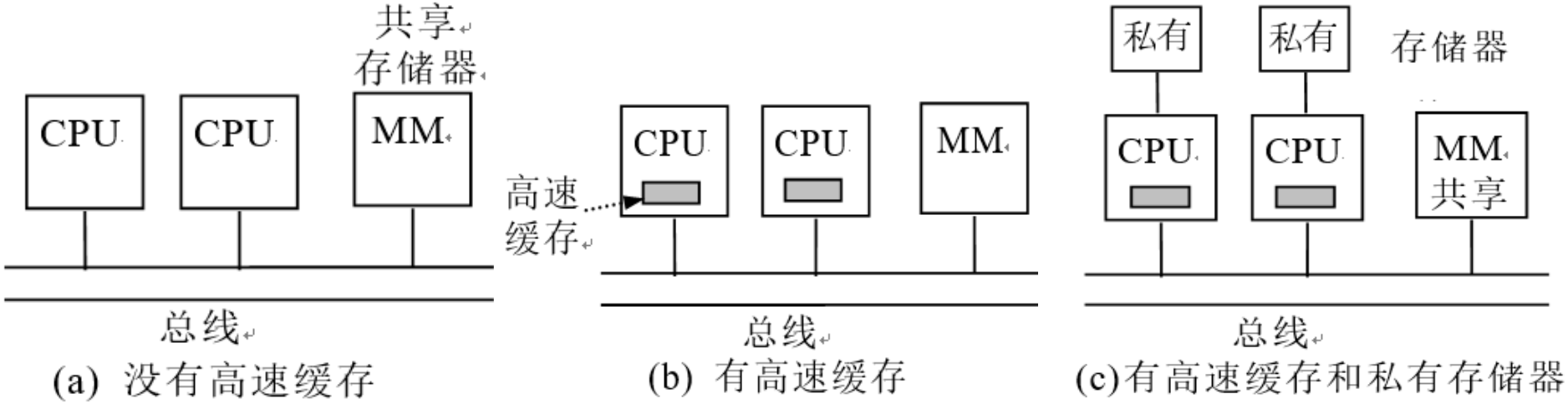
多处理器系统

编译器、链接器和操作系统等各种系统软件的功能需进行相应调整

- UMA多处理器结构
  - 基于总线连接的UMA系统

私有MM：存放程序、非共享变量、只读数据、栈内信息等

共享MM：存放可写的共享变量

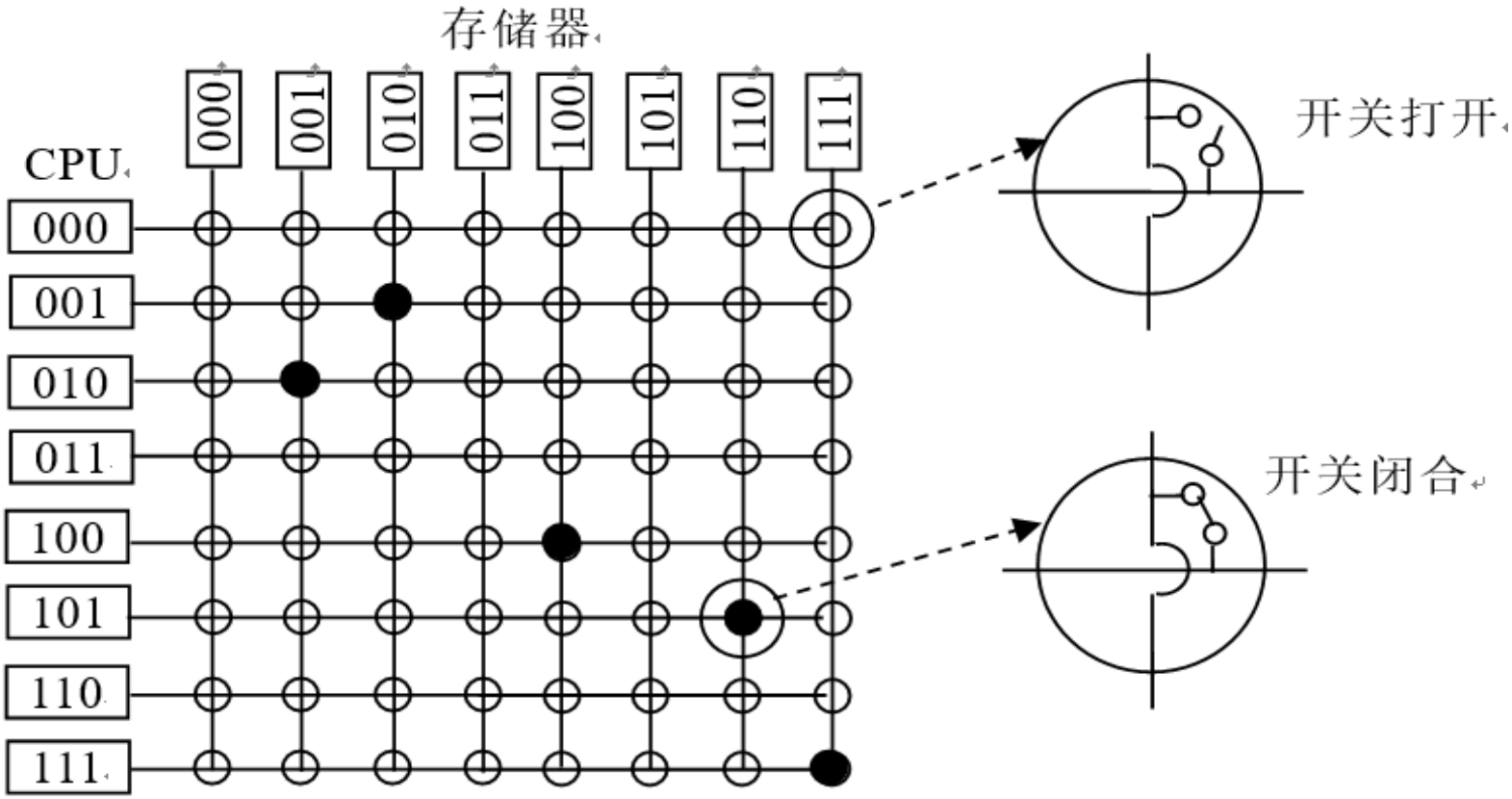


有Cache时：需Cache一致性协议，最常用的是侦听协议，如MESI协议，采用写无效策略（当侦听到一个写操作时，使其他副本无效）

缺点：总线繁忙，所连接模块数受限，因此有系统考虑基于交叉开关网络的UMA系统

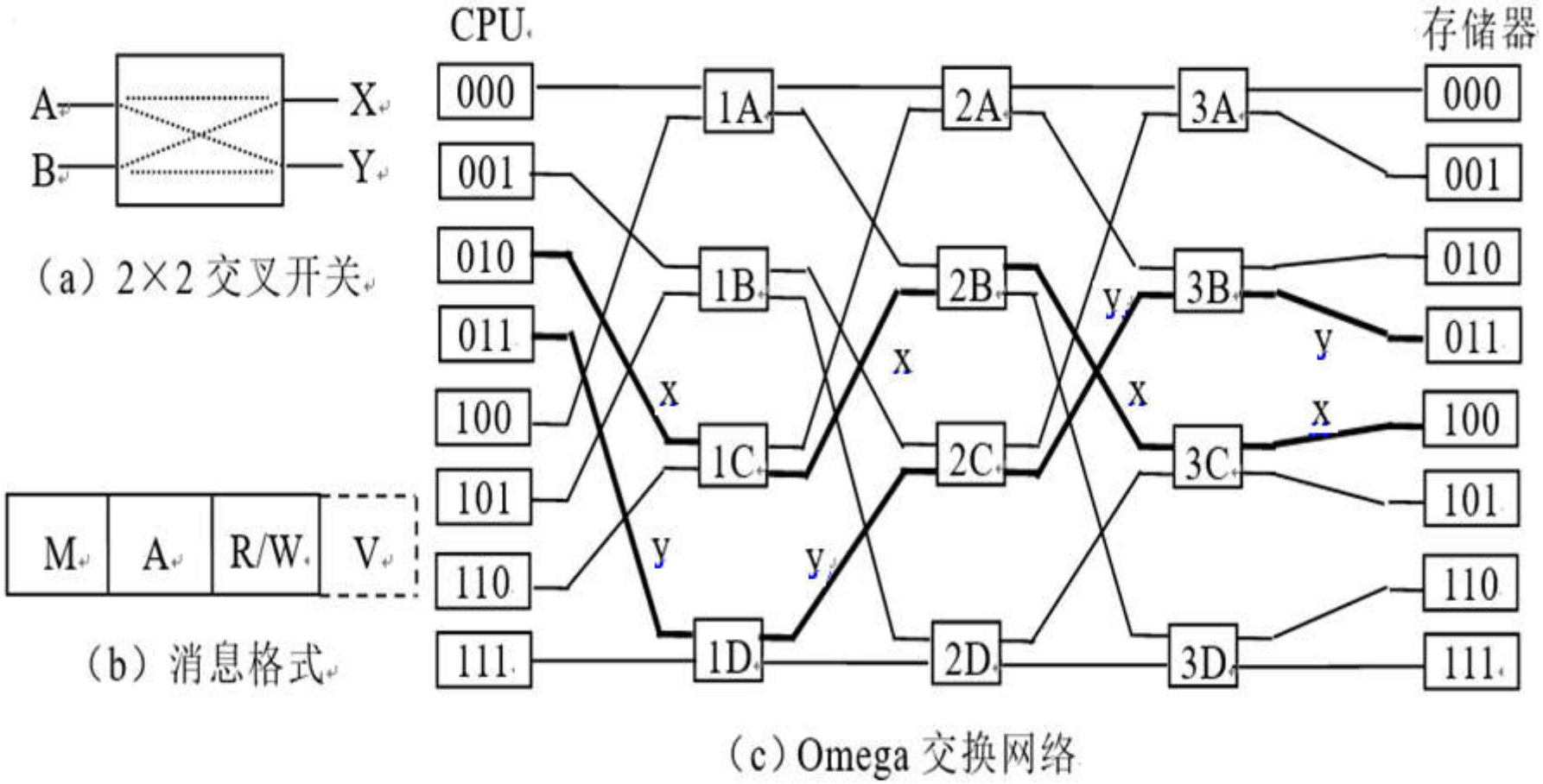
多处理器系统

- UMA多处理器结构
  - 基于交叉开关网络的UMA系统：连接m个CPU模块到n个存储器模块的最简单电路是交叉开关连接网路。交叉开关的数量为m×n，故规模受限于开关数量。



多处理器系统

- UMA多处理器结构
  - 基于多级交换网络的UMA系统：在CPU模块和存储器模块之间用多级小规模交叉开关（如2x2）网络连接，在数据访问过程中需要携带消息，消息中带有存储模块号、地址、读/写操作类型。



回顾：并行处理系统的分类

- 按指令和数据处理方式划分（Flynn分类）
  - SISD（单指令流单数据流）：指令级并行（超流水、超标量、动态调度）
  - SIMD（单指令流多数据流）：数据级并行（如Intel的MMX、SSE、AVX等）
  - MIMD（多指令流多数据流）：多计算机和多处理器系统
- 按地址空间的访问方式划分（MIMD）
  - 多计算机系统：计算节点有各自独立编址的存储器，通过消息传递方式访问其他节点的存储器，也称消息传递系统
  - 多处理器系统：各计算节点共享统一编址的存储器，可通过LOAD和STORE指令访问系统中的存储器，也被称为共享存储多处理器（Shared memory multiProcessor, SMP）系统
- 按访存时间是否一致划分（多处理器系统）　　(Non-) Uniform Memory Access
  - 一致性访存（UMA）：每个处理器访存时间一致，通常共享一个存储器
  - 非一致性访存（NUMA）：处理器的访存时间不一致，与存储器位置有关
  - CC-NUMA：Cache-Coherent NUMA，支持cache一致性的NUMA

多处理器系统

- NUMA多处理器结构
  - UMA系统的缺点：
    - 要保证每个处理器的访存时间与存储模块位置无关，因而其连接的处理器规模受到一定的限制。

    要构建更大规模的多处理器系统，须打破访存时间一致的限制，在保证单一地址空间的前提下允许访存时间不一致，因此，出现了非一致内存访问（NUMA）多处理机。
  - NUMA系统的特点：
    - 与UMA一样，所有存储器统一编址，都可用LOAD和STORE指令访问，因此，UMA程序可在NUMA系统上执行。
    - NUMA系统中，本地访问快于非本地访问。
    - 所连接处理器规模比UMA更大。
    - 在节点互连、并行编程、cache一致性方面与UMA多处理器不同。
  - 处理器中带有一致性高速缓存时，系统被称为CC-NUMA。

多处理器系统

- NUMA多处理器结构
  - 基于目录的CC-NUMA系统

目录：记录本地存储模块中各主存块的副本在哪个节点的Cache中，都不存在，则有效位 V=0.

例：节点18执行LOAD指令，其MMU将虚拟地址转换为物理地址480000C8H

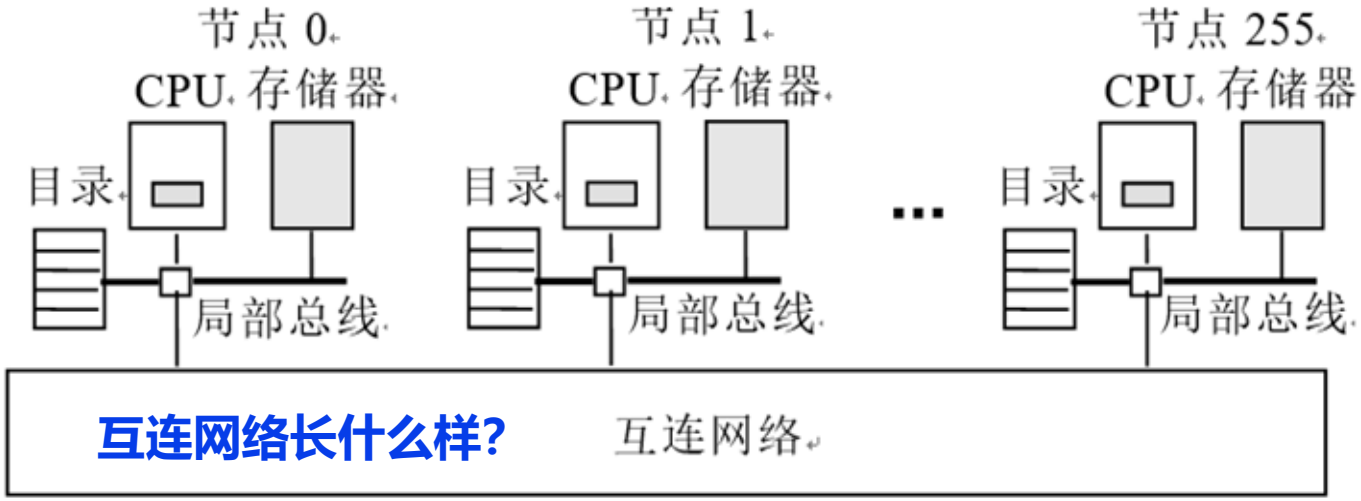
48H=72，

C8H=200(3,8)

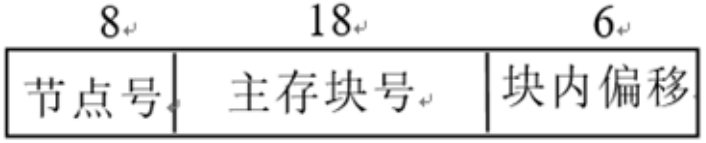
含义：从节点72的本地存储器第200单元读信息

查72#目录中第3项，V=0(miss).

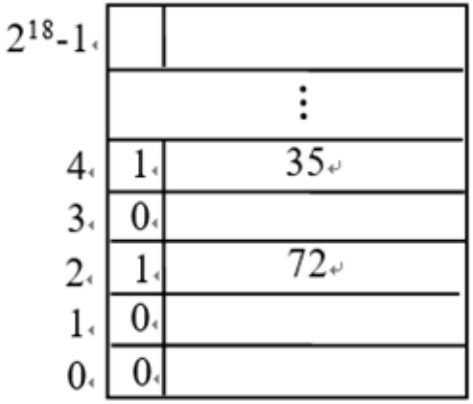
从72#的200号单元读出，送节点18中某Cache行，改第3项的V=1, 节点为18



(a) CC-NUMA 互连结构



(b) 主存地址划分

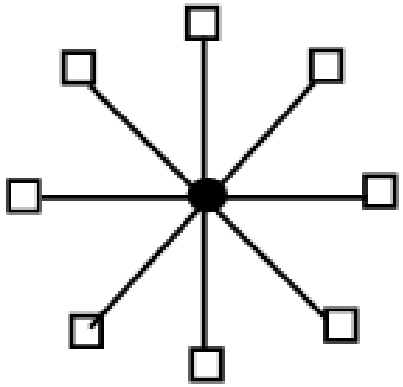


(c) 节点 72 中的目录

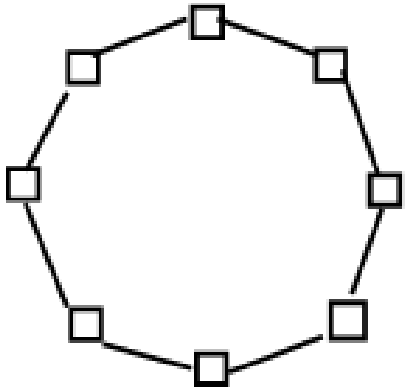
多处理器系统

- NUMA多处理器结构
  - NUMA互连网络结构

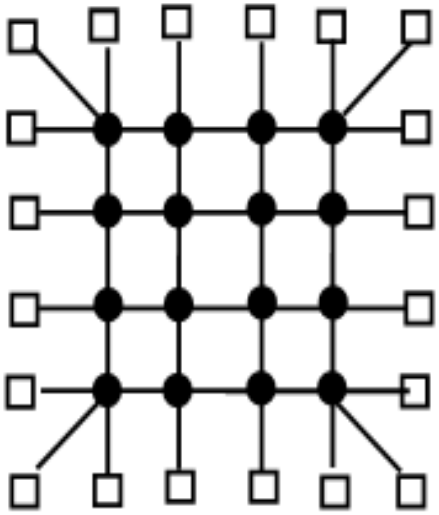
方块表示一个CPU-存储器节点，黑圆点表示交换器



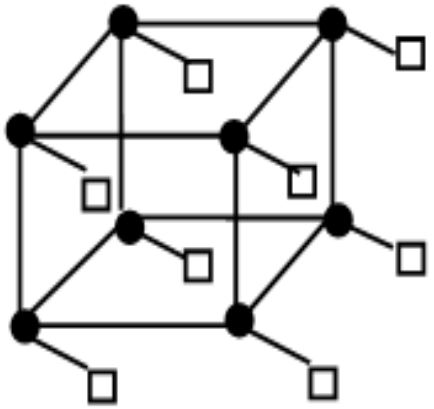
(a) 星形



(b) 环形



(c) 网格



(d) 立方体

回顾：并行处理系统的分类

- 按处理单元的位置及其互连方式划分

多核属于片级多处理器

- 多核：一个CPU芯片中含多个（2、4、8等）核，共享LLC和主存 属多处理器
- 对称多处理器（Symmetric MultiProcessor, SMP）：相同类型CPU通过总线互连，并等同地位共享所有存储资源。即多个CPU对称工作。可见SMP就是一种UMA结构多处理器。PC、工作站和服务器等多采用SMP结构。
- 大规模并行处理机（MPP）：以专用内联网络连接数量众多处理单元而构成的并行计算系统。例如，可通过专用互连网络（如Mesh、交叉开关）将数量达几百甚至几千个的SMP服务器连接成MPP，SMP服务器之间协同工作，以完成同一个任务。
- 集群（Cluster）：以高速网卡将若干PC或SMP服务器或工作站连接而成的并行计算系统，其中每个节点有各自的独立编址的内存和磁盘，属于紧耦合同构多计算机系统（消息传递机制）。
- 网格（Grid）：以因特网等广域网将远距离分布的一组异构计算机系统连接而成的分布式并行处理系统，属于松耦合异构多计算机系统。 属多计算机
- 众核：一个GPU芯片中含几百个简单核，众多并行线程同时执行
- APU：CPU+GPU融合

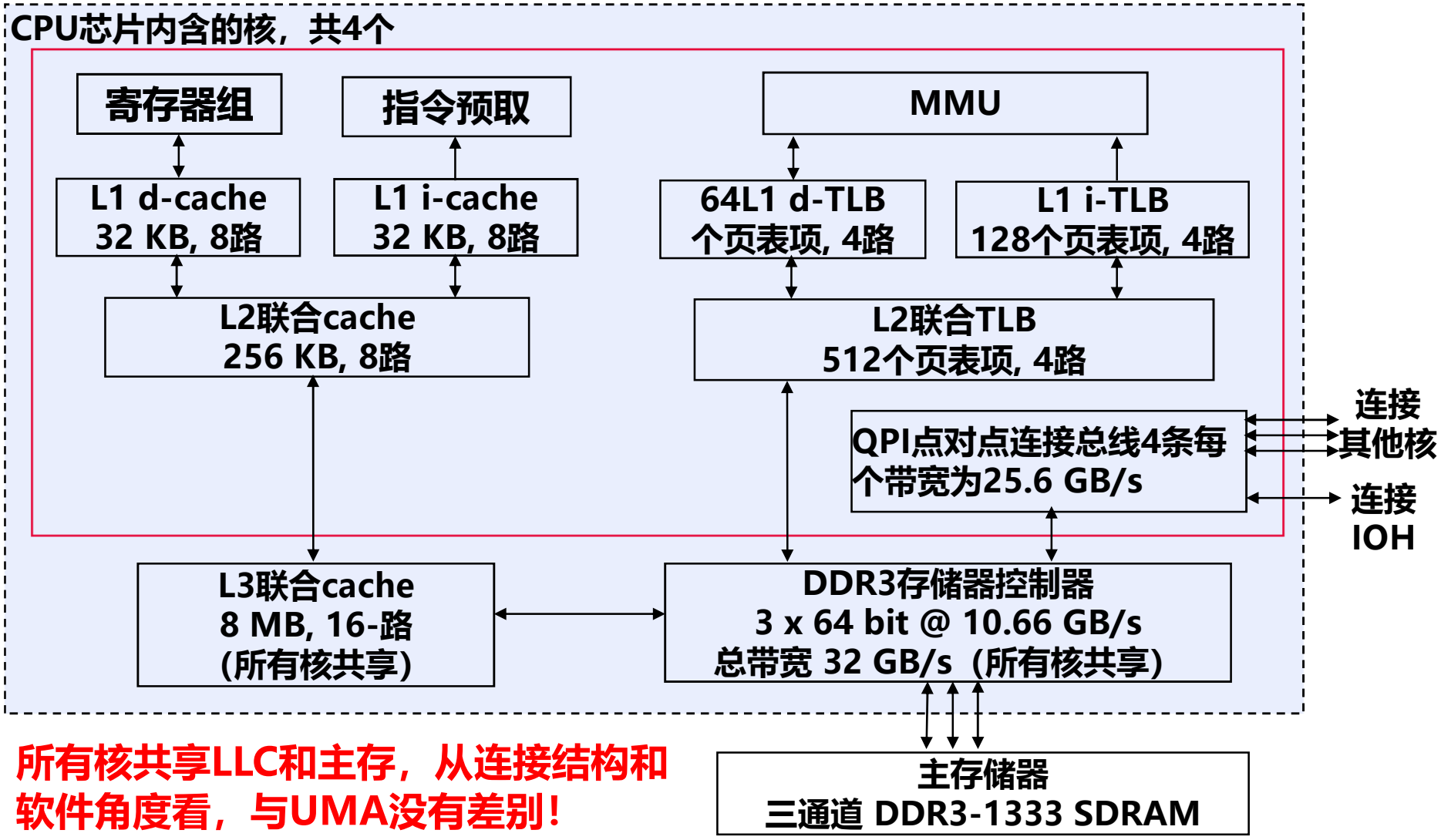
多处理器系统

SKIP

- 片级多处理器
  - 功耗墙：$P=CV^2f$ （P-功耗，C-时钟跳变时门电路电容，与集成度成正比；V是电压；f是主频）。功耗与集成度和主频成正比。集成度和主频的大幅度提升导致了功耗急剧增大。
  - 指令级并行墙：ILP技术发展到了极限
  - 2004年，Intel 4GHz主频处理器未能解决散热问题，标志升频技术时代的终结。
  - 2005年，Intel宣布从单处理器性能提升战略转向多核微处理器架构战略。
  - 多核：一个CPU芯片包含多个简单处理器核（Core），通过多核并行计算提高性能。也称为片级多处理器计算机（Chip-level MultiProcessors, CMP）。
- 硬件多线程（从CPU的角度看：一个线程就是一个指令序列）
  - 多线程并行：为每个线程提供单独的通用寄存器组、PC以及线程切换机制
  - 细粒度：在多个线程之间轮流交叉执行指令，能在每个时钟周期切换线程
  - 粗粒度：在某线程出现较大阻塞开销时才切换，例如，cache缺失时切换
  - 同时多线程（如Intel的超线程）：在单处理器或单核中设置多套线程状态部件而共享cache和功能部件。能在实现ILP的同时实现线程级并行，即: 同一时钟内在不同发射槽中发射不同线程中的指令。

回顾：Core i7的内部结构

BACK



所有核共享LLC和主存，从连接结构和软件角度看，与UMA没有差别！

从Core i7开始，北桥在CPU芯片内，CPU通过存储器总线（即内存条插槽，图中为三通道插槽）直接和内存条相连。3个存控包含在CPU芯片内。

共享存储器的同步控制

- 为什么需要同步控制?
  - 每个处理器核有一个或多个指令流（硬件多线程时）在执行。若多个核都要访问共享存储器，则由于每个核在执行程序时存在流水线阻塞、功能部件冲突、指令顺序调度等各种随机情况，多个核的程序之间访存顺序存在不确定性，从而造成执行结果的不确定。
- 如何进行同步控制?
  - ISA必须规定所用的存储器一致性模型（Memory Consistency Model），简称存储器模型，并提供一套配套同步指令。
- 三种代表性存储器模型
  - 顺序（Sequential）一致性模型：每个核完全按照程序指定的访存顺序执行，不可改变访存顺序。对存储器的访问都应该是串行的，效率低。
  - 宽松（Relaxed）一致性模型：允许在一定的条件下改变核内程序对于不同存储单元的访存顺序。通常用一个存储器屏障指令，设置一个同步点，前面的访存不能在屏障后进行、后面的也不能越过屏障提前进行。即双向限制!
  - 释放（Release）一致性模型：只限定一个方向的访存指令执行顺序。采用获取-释放机制实现，可用获取（Acquir）指令拦截后面的访存指令，或用释放（Release）指令拦截前面的访存指令。

共享存储器的同步控制

- 举例说明存储器一致性模型的使用

要求：核0将一块数据写入存储器中某个区域，然后，通知核1读取此数据块。

多核应用程序主要实现思想如下：

（1）核0和核1约定一个共享的全局变量作为同步标志。全局变量在主存中分配一个存储单元，两个核都可以访问该存储单元，以设置或读取其值；

（2）核0完成了写数据块的操作后，就将一个“特殊的值”写入共享变量单元中作为旗语；

（3）核1不断监测此共享变量的值，一旦检测到“特殊的值”，则认为可以安全地读取数据块。

因此，两个核上的程序可以各自抽象为以下操作序列。

- 核0：写入数据块→设置旗语。
- 核1：监测旗语→检测到“特殊的值”→读取数据块。

显然，核0的“写入数据块”和“设置旗语”的操作一定不能改变顺序

核1的“监测旗语”和“读取数据块”的操作也一定不能改变顺序

试分别描述顺序一致性模型、宽松一致性模型和释放一致性模型的实现。

共享存储器的同步控制

- 在顺序一致性模型的多核系统中，按以下操作序列进行。
  - 核0：写入数据块→设置旗语。
  - 核1：监测旗语→检测到“特殊的值”→读取数据块。

  核0和核1完全按照各自的顺序来访存，可能的顺序有：

  （1）写入数据块→设置旗语→监测旗语→检测到“特殊的值”→读取数据块。

  （2）写入数据块→监测旗语→设置旗语→检测到“特殊的值”→读取数据块。

  显然，以上两种顺序都没有问题。

- 在宽松一致性模型的多核系统中，由于数据块和共享变量所在存储地址不同，若不用屏障指令，则编译器或处理器可能会进行指令顺序调度优化，使程序最终执行的结果不满足程序员的预期。因此需要在程序中插入屏障指令FENCE。
  - 核0：写入数据块→FENCE指令→设置旗语。
  - 核1：监测旗语→检测到“特殊的值”→FENCE指令→读取数据块。

  显然，屏障指令（前、后不越屏障）保证了程序员所要求的访存顺序。

- 若将“FENCE指令”和“设置旗语”合成“释放旗语”指令；将“监测旗语”和“FENCE指令”合成“获取旗语”指令，就是释放一致性模型中的方案。

共享存储器的同步控制

- 在释放一致性模型的多核系统中，按以下操作序列进行。
  - 核0：写入数据块→释放旗语。
  - 核1：获取旗语→获取旗语检测到“特殊的值”→读取数据块。

  “释放旗语”指令拦截前面的访存，以保证前面的访存不会在其之后执行；

  “获取旗语”指令拦截后面的访存，以保证后面的访存不会在其之前执行。

  显然，上述操作序列满足程序员要求。
- 总结比较：
  - 顺序一致性模型：绝对保证访存顺序，不能进行指令调度优化操作，效率低。
  - 宽松一致性模型：不同存储地址的访存需要用屏障指令保证前、后两个方向的访存顺序。没有顺序要求时，可不用屏障指令，因而可进行指令调度优化。
  - 释放一致性模型：不同存储地址的访存操作可用“释放”或“获取”指令单方向拦截访存操作。没有顺序要求时，可不用“释放”和“获取”指令，因而可进行指令调度优化。比宽松一致性模型更加宽松！
  - 还有更多存储器一致性模型；共享存储器同步控制问题不仅限于多核处理器系统，在共享存储器多处理器系统中以及硬件多线程共享存储器时也一样。

RISC-V的共享存储器同步控制方式

- 在不同hart之间，使用宽松一致性模型进行同步控制。在RV32I中提供了fence屏障指令，用于约束数据访问指令的执行顺序。
- 针对事务处理和操作原子性需要，提供了扩展的32位架构原子操作指令集RV32A，以及64位架构原子操作指令（+RV64A），从而可进一步支持释放一致性模型。
- 举例说明：假定有三个处理器核C0、C1和C2需要共享一个数据块，并且要求某一时刻只能有一个核独占访问此共享数据块，要实现该功能可以采用以下的同步策略。

（1）定义一个共享的全局变量作为“锁”，该共享变量占用一个存储单元，三个核都可访问该存储单元，并约定：若锁值为0，表示当前共享数据块空闲，即没有被任何核独占；若锁值为1，表示当前共享数据块正在被某个核独占访问。

（2）当某核独占共享数据块完成相关操作后，便通过向锁中写0来立即释放数据块。

（3）当某个核独占共享数据块时，其他两个核都会不断地读取锁值，并判断锁值是否为0。一旦发现锁值为0，则立即向锁中写入1进行上锁，试图将共享数据块独占。

若采用普通Load/Store指令，则在执行Load指令（读锁为0）与下一条Store指令（写锁为1）之间，可能被插入其他核的Load指令读锁操作，结果两个核读的锁值都为0，都认为自己可独占数据块，并都将锁值设置为1。显然，这与要求实现的功能不相符。可用“原子操作”和“互斥操作”两种方式解决“上锁”问题！

RISC-V中如何解决“上锁”问题

- **原子操作方式**

  通过定义专门的原子操作指令，使得“读锁”和“写锁”不可分！例如：

  ARM中的原子交换（SWP）指令：将存储单元内容送结果寄存器，并将源寄存器内容写入同样的存储单元中。在读操作后，硬件根据总线上“Lock”信号线有效来锁定存储单元或cache行，使其他核不能访问，因而保证“读锁-写锁”过程不被打断。

  RISC-V架构提供了9条原子操作指令（RV32A指令集），其中，有一条原子交换(amoswap）指令。对应加锁功能的部分代码如下：

```
0:  00100293  li  t0, 1                          #初始化锁值
4:  0c55232f  amoswap.w.aq t1, t0, (a0)   #R[t1]←M[R[a0]], M[R[a0]]←R[t0]
8:  fe031ee3  bnez  t1, 4                        #若没有成功，则继续尝试
      ……操作共享的数据块（临界区代码）……
20:0a05202f amoswap.w.rl  x0, x0, (a0)   #释放锁：M[R[a0]]←R[x0](=0)
```

  三个核C0、C1和C2上的代码一样。若C0独占数据块，则可能C1和C2都正好要执行amoswap.w.aq（获取指令，拦截后面的访存，即获取后才能访存）。因为是原子操作，某一段时间一定只能C1或C2中的一个执行“读锁-写锁”操作，直到C0用amoswap.w.rl（释放指令，拦截前面的访存，即释放后便不能再访存）释放锁。

RISC-V中如何解决“上锁”问题

- 互斥操作方式

  定义一对互斥操作指令：互斥读（Load-Exclusive）和互斥写（Store-Exclusive）

  互斥读指令：与普通的Load指令类似，用于从存储单元中取出数据送寄存器。

  互斥写指令：与普通Store指令不同，不一定成功，结果寄存器中存放成功标志。

  监视器：监测同一个核的一条互斥读和一条互斥写指令是否成对地先、后访问同一个主存地址，且在此期间是否没有来自其他任何核或线程的写主存操作访问过该地址，且无任何异常和中断发生。如果是的话，说明互斥写指令的写主存操作执行成功。

  Lock信号线：互斥读/写指令时有效。用于区分普通的主存读/写指令对应的操作。

- RISC-V架构扩展指令集RV32A（原子操作指令集）中，提供了两条互斥操作指令，可在指令中设置“获取”（aq字段为1）或“释放”（rl字段为1）属性。

  Load-Reserved（lr.w）：将存储单元中的锁值读到某个寄存器中。

  Store-Conditional（sc.w）：将源寄存器内容试图写入锁值所在内存单元，若成功写入，则将0存入结果寄存器中。

RISC-V中如何解决“上锁”问题

° 下列RISC-V程序段给出了利用互斥操作指令实现加锁操作的部分代码：

```
0:   li t0, 1                  #初始化锁值为1
4:   lr.w  a3, (a0)            #读出锁值到a3寄存器
8:   bne a3, x0, 4             #若读出的锁值不等于0，则转地址4继续读锁值
c:   sc.w.aq  a3, t0, (a0)     #尝试将1写入锁中，若成功，则a3中为0，拦截后面访存
10: bnez  a3, 4                #若没有成功，则转地址4处继续从读锁值开始
      ……操作共享的数据块（临界区代码）……
28: lr.w.rl  a3, (a0)          #读出锁值到a3，并拦截前面的访存
2c: sc.w.aq  a3, x0, (a0)      #尝试将0写入锁中，以释放锁，并拦截后面的访存
30: bnez  a3, 28               #若没有成功，则转地址28处继续尝试释放锁
```

若核C0独占数据块，则C1和C2可能正好在执行前5条指令进行“读锁-写锁”操作。C1和C2可能交错执行这些指令，导致C1和C2都刚好要执行“sc.w.aq a3, t0, (a0)”指令，以尝试写锁。不过，多线程硬件能保证总是有一个核先写锁，系统中的“监测器”能保证先执行该指令的核（或线程）获得锁，而另一个则不能成功写锁，从而保证一旦C0释放锁，那么C1和C2中至多只有一方能够获得锁，而不会发生C1和C2同时都获得锁的情况。临界区操作结束后，先通过设置rl=1来拦截前面的访存（即共享数据块操作结束），再将0（x0的值）成功写入锁中来释放锁。

回顾：并行处理系统的分类

- 按处理单元的位置及其互连方式划分

下面介绍多计算机系统

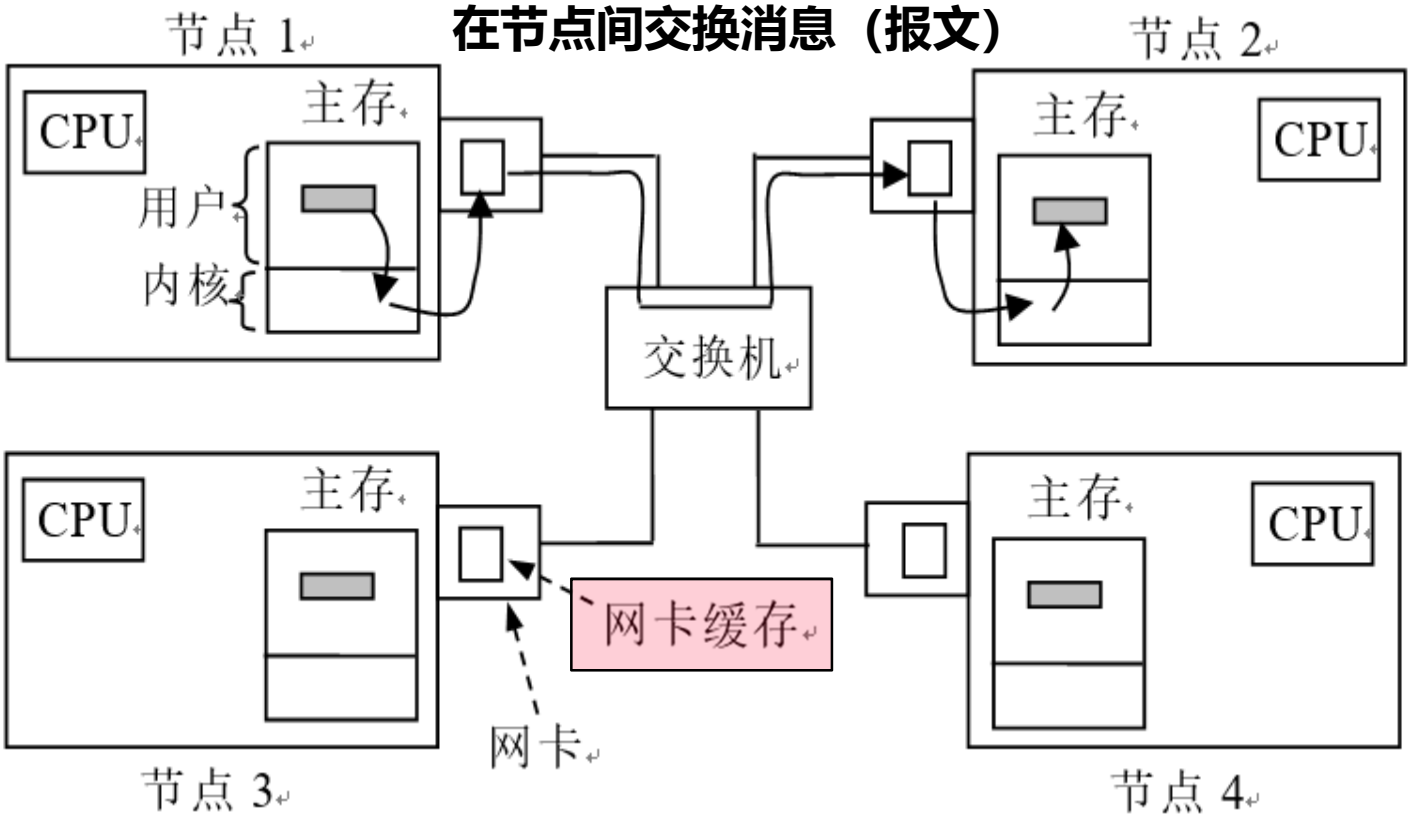
  - 多核：一个CPU芯片中含多个（2、4、8等）核，共享LLC和主存
  - 对称多处理器（Symmetric MultiProcessor, SMP）：相同类型CPU通过总线互连，并等同地位共享所有存储资源。即多个CPU对称工作。可见SMP就是一种UMA结构多处理器。PC、工作站和服务器等多采用SMP结构。
  - 大规模并行处理机（MPP）：以专用内联网络连接数量众多处理单元而构成的并行计算系统。例如，可通过专用互连网络（如Mesh、交叉开关）将数量达几百甚至几千个的SMP服务器连接成MPP，SMP服务器之间协同工作，以完成同一个任务。
  - 集群（Cluster）：以高速网卡将若干PC或SMP服务器或工作站连接而成的并行计算系统，其中每个节点有各自的独立编址的内存和磁盘，属于紧耦合同构多计算机系统（消息传递机制）。
  - 网格（Grid）：以因特网等广域网将远距离分布的一组异构计算机系统连接而成的分布式并行处理系统，属于松耦合异构多计算机系统。

属多计算机

  - 众核：一个GPU芯片中含几百个简单核，众多并行线程同时执行
  - APU：CPU+GPU融合

多计算机系统

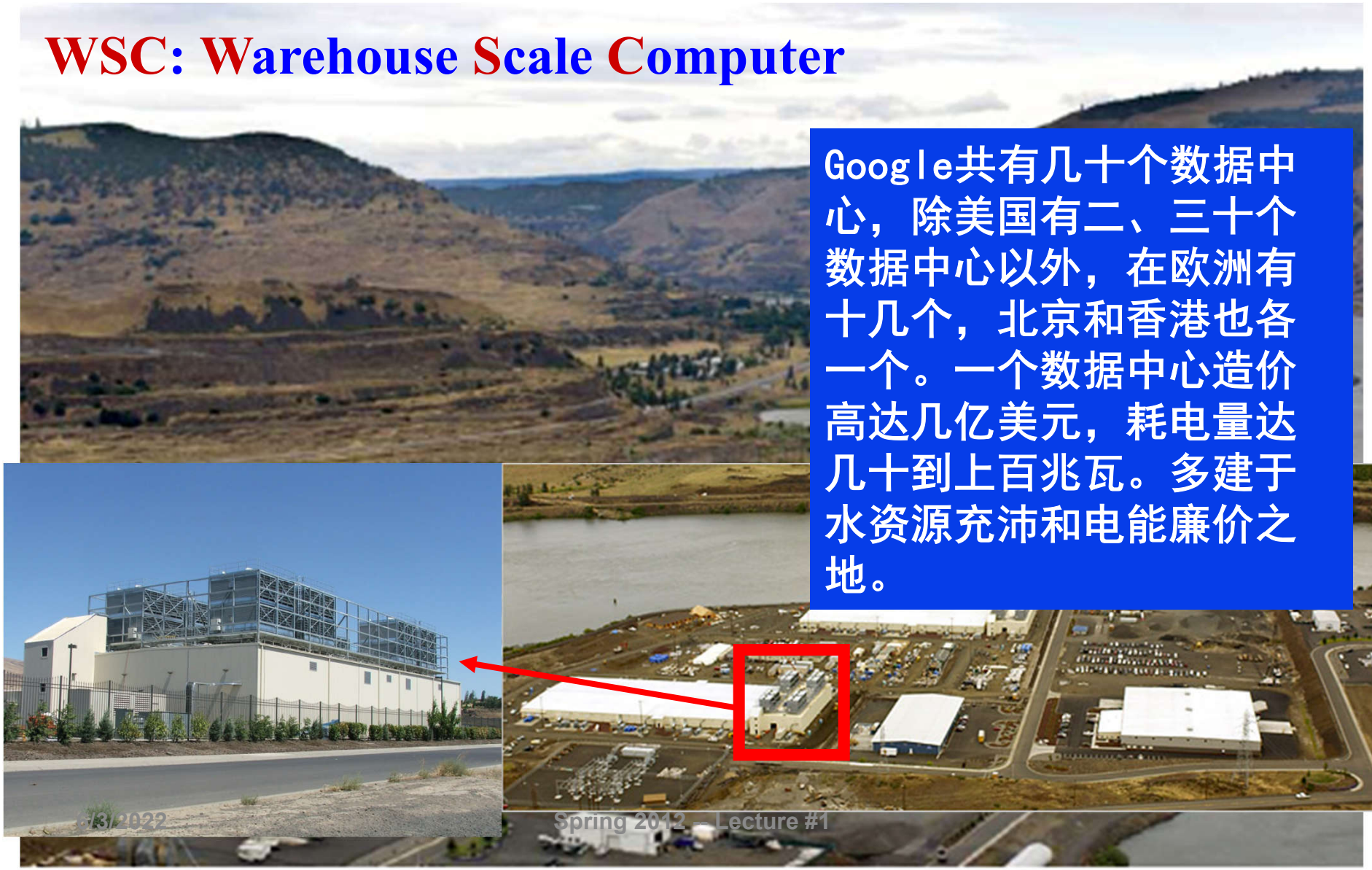
- 集群（Cluster）
  - 互连方式：局域网连接普通PC、通过消息传递进行通信，比多处理器系统中的互连网络慢，集群中信息访问为微秒级，多处理器中为纳秒级
  - 体积：集群在一个或多个房间甚至更大空间中，而多处理器通常在一个机箱中
  - 应用广泛：数据库系统、文件服务器、Web服务器等



网卡中通常包含多个DMA通道或专门的I/O处理器.

交换机可采用分组交换（将消息划分成更小的“分组”）和电路交换两种机制

集群举例：Google's Oregon WSC



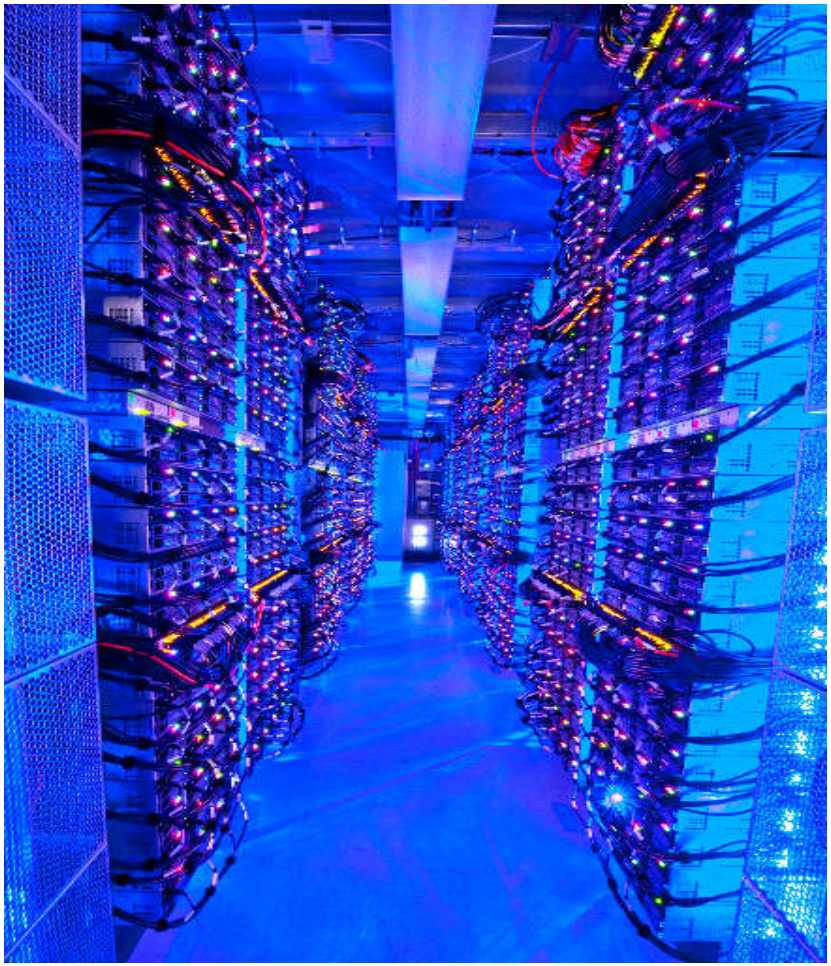
Containers in WSCs
（数据中心内的集装箱）

WSC内部

Google用运输集装箱来存放服务器及冷却设备，所耗电能非常少。

集装箱之间互联，里面有很多机柜，构成机柜阵列，也称为集群。

集装箱内部



Server, Rack, Array (Claster)

服务器、机架、阵列 (集群)

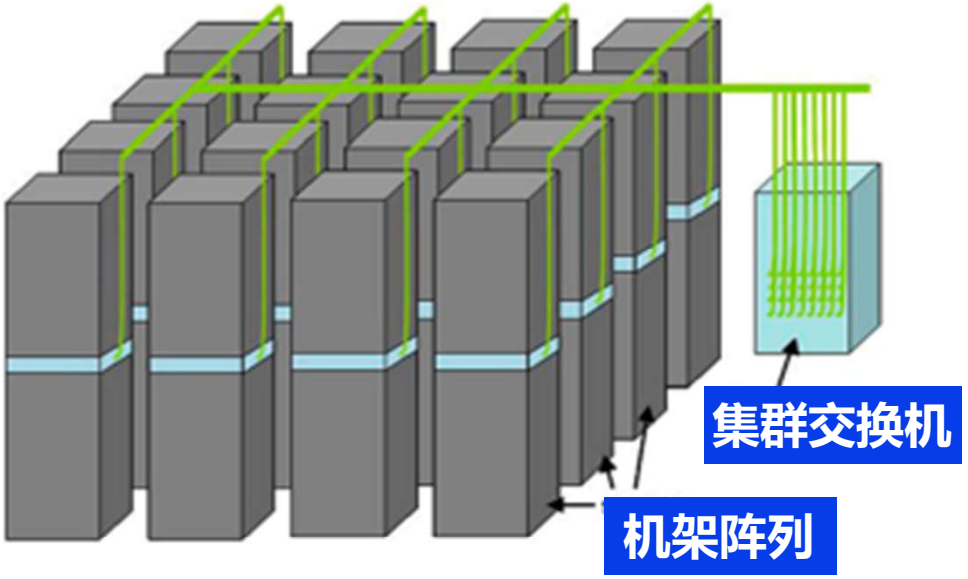
据称，目前Google服务器总数超过几百万台！

Google Server Internals
（Google服务器内部）

Google量身定做的服务器电源整合了蓄电池功能，可作为不间断电源（UPS）使用，确保服务器正常运转

Equipment Inside a WSC

Server(服务器):
8 cores(处理器核心)
16 GB DRAM(主存)
4x1 TB disk(磁盘)



Array (cluster):
16-32 server racks +
larger local area network
switch (“array switch”)

7 foot Rack: 40-80 servers + Ethernet
local area network (1-10 Gbps) switch

哪位已算出一个集装箱规模有多大?
几十个集装箱呢? 几十个数据中心呢?

一个大集装箱
盘容量可
达几EB!

32x80x8=20 480
32x80x16GB≈40TB
32x80x4TB ≈10PB

多计算机系统

- 网格（Grid）
  - 互连方式：利用互联网把分散在不同地理位置的多个异构计算机组成一台逻辑上的“虚拟超级计算机”
  - 体积：物理上可能位于世界各地
  - 构建思路：不同的管理系统上运行相应的中间件，以使用户和程序可以通过方便的、一致的方式访问所有资源
  - 基本原理：在网格中的每个计算机中运行一个特殊的程序，这个程序可以用来管理计算机并使计算机加入到网格中。因此，这个程序通常需要处理用户认证及远程登录、资源发布与发现、作业调度与分配等。当网格中的某个用户需要计算机完成某个任务时，网格软件决定何处有空闲的硬件、软件和数据资源，然后将作业迁移到有资源的计算机处，安排执行并收集处理结果返回给用户。
  - 预期目标：实现可靠的、具有较高容错和容灾性的系统，并且可以节省资源，实现资源共享
  - 应用困难：还存在许多技术问题，设想难以实现！

向量处理机和SIMD技术

主　要　内　容

- 向量处理机简介
- Intel架构中的SIMD技术
- GPU架构简介

向量处理机简介

- 在空气动力学、原子物理学、核物理学、气象学和化学等科学计算中，涉及到线性规划、傅里叶变换、滤波计算以及矩阵、线性代数、偏微分方程、积分等数学问题的求解，这些求解问题大都要求能对大量结构相同的数据进行高精度的浮点运算。
- 为了解决这些科学计算问题，历史上曾有一些公司研制出了一种超级（巨型）计算机----向量处理机（vector computer），它借助向量计算结构的数据并行技术，可以完成每秒钟上亿次的运算。
  - TI公司的ASC(1972年)
  - CDC公司STAR-100 (1973年)
  - 中国国防科大的“银河”机（1983年）
  - 到1982年底，世界上约有60台巨型机，大多是向量处理机。Cray公司的超级计算机多属于向量结构计算机。

向量处理机简介

- 向量处理机的基本思想
  - 面向向量型数据的并行计算：各分量彼此无关、各自独立
  - 主要用于求解大型问题，须具有集中式大容量高吞吐量的主存和向量寄存器，以源源不断地供给操作数和取走运算结果
- 举例：考察一个C语言循环程序（大约需运行700--800条标量指令）

  for (i=0; i<64; i++) c[i]=a[i]+b[i];
- 在向量处理机（32个向量寄存器，每个存放64个32位数据字，装入/存储指令（v_lw和v_sw）可从内存连续256个单元装入数据到一个向量寄存器或将向量寄存器内容存到内存连续单元中，运算指令（如v_add）可同时对两个向量寄存器中的64个32位字并行计算，并将结果存到另一个向量寄存器），则上述C语言循环程序段对应的指令只有4条。
  - v_lw　　$v_t1, 0($s1)　　#将a[0]到a[63]取到$v_t1中
  - v_lw　　$v_t2, 0($s2)　　#将b[0]到b[63]取到$v_t2中
  - v_add　　$v_t3, $v_t1, $v_t2　　#计算a[i]+b[i]，存到$v_t3中
  - v_sw　　$v_t3, 0($s3)　　#将a[i]+b[i]的值存到c[i]中

回顾：X87浮点指令、MMX和SSE指令

° IA-32的浮点处理架构有两种：

- 浮点协处理器x87架构（x87 FPU）

✓8个80位寄存器ST(0) ~ ST(7)（采用栈结构），栈顶为ST(0)

- 由MMX发展而来的SSE架构

✓MMX指令使用8个64位寄存器MM0~MM7，借用8个80位寄存器ST(0)~ST(7)中64位尾数所占的位，可同时处理8个字节，或4个字，或2个双字，或一个64位的数据

✓MMX指令并没带来3D游戏性能的显著提升，故相继推出SSE指令集，它们都采用SIMD（单指令多数据，也称数据级并行）技术

✓SSE指令集将80位浮点寄存器扩充到128位多媒体扩展通用寄存器XMM0~XMM7，可同时处理16个字节，或8个字，或4个双字（32位整数或单精度浮点数），或两个四字的数据，而且从SSE2开始，还支持128位整数运算或同时并行处理两个64位双精度浮点数

回顾：IA-32中通用寄存器中的编号

编号	8 位寄存器	16 位寄存器	32 位寄存器	64 位寄存器	128 位寄存器
000	AL	AX	EAX	MM0 / ST(0)	XMM0
001	CL	CX	ECX	MM1 / ST(1)	XMM1
010	DL	DX	EDX	MM2 / ST(2)	XMM2
011	BL	BX	EBX	MM3 / ST(3)	XMM3
100	AH	SP	ESP	MM4 / ST(4)	XMM4
101	CH	BP	EBP	MM5 / ST(5)	XMM5
110	DH	SI	ESI	MM6 / ST(6)	XMM6
111	BH	DI	EDI	MM7 / ST(7)	XMM7

反映了体系结构发展的轨迹，字长不断扩充，指令保持兼容

ST（0）~ ST（7）是80位，MM0 ~MM7使用其低64位

回顾：SSE指令（SIMD操作）

- 用简单的例子来比较普通指令与数据级并行指令的执行速度
  - ✓为使比较结果不受访存操作影响，下例中的运算操作数在寄存器中
  - ✓为使比较结果尽量准确，例中设置的循环次数较大: 0x4000000=2^{26}
  - ✓例子只是为了说明指令执行速度的快慢，并没有考虑结果是否溢出

以下是普通指令写的程序

```
080484f0 <dummy_add>:
 80484f0:  55              push  %ebp
 80484f1:  89 e5           mov  %esp, %ebp
 80484f3:  b9 00 00 00 04  mov  $0x4000000, %ecx
 80484f8:  b0 01           mov  $0x1, %al
 80484fa:   b3 00          mov   $0x0, %bl
 80484fc: 00 c3            add   %al, %bl
 80484fe: e2 fc            loop  80484fc <dummy_add+0xc>
8048500:  5d              pop           %ebp
 8048501: c3              ret
```

所用时间约为22.643816s

循环400 0000H=2^{26}次，每次只有一个数（字节）相加

回顾：SSE指令（SIMD操作）

以下是SIMD指令写的程序

所用时间约为1.411588s

```
08048510 <dummy_add_sse>:
 8048510:  55                   push  %ebp
 8048511:  b8 00 9d 04 10       mov $0x10049d00, %eax
 8048516:  89 e5                mov   %esp, %ebp
 8048518:  53                   push   %ebx
 8048519:   bb 20 9d 04 14      mov   $0x14049d20, %ebx
 804851e:  b9 00 00 40 00       mov   $0x400000, %ecx
 8048523:  66 0f 6f 00          movdqa  (%eax), %xmm0
 8048527:  66 0f 6f 0b          movdqa  (%ebx), %xmm1
 804852b: 66 0f fc c8           paddb    %xmm0, %xmm1
 804852f: e2 fa                 loop   804852b <dummy_add_sse+0x1b>
 8048531:  5b                   pop    %ebx
 8048532:  5d                   pop    %ebp
 8048533:  c3                   ret
```

SIDM指令

22.643816s/
1.411588s
≈16.041378,与
预期结果一致!
SIMD指令并行
执行效率高!

dqa：两个对齐四字

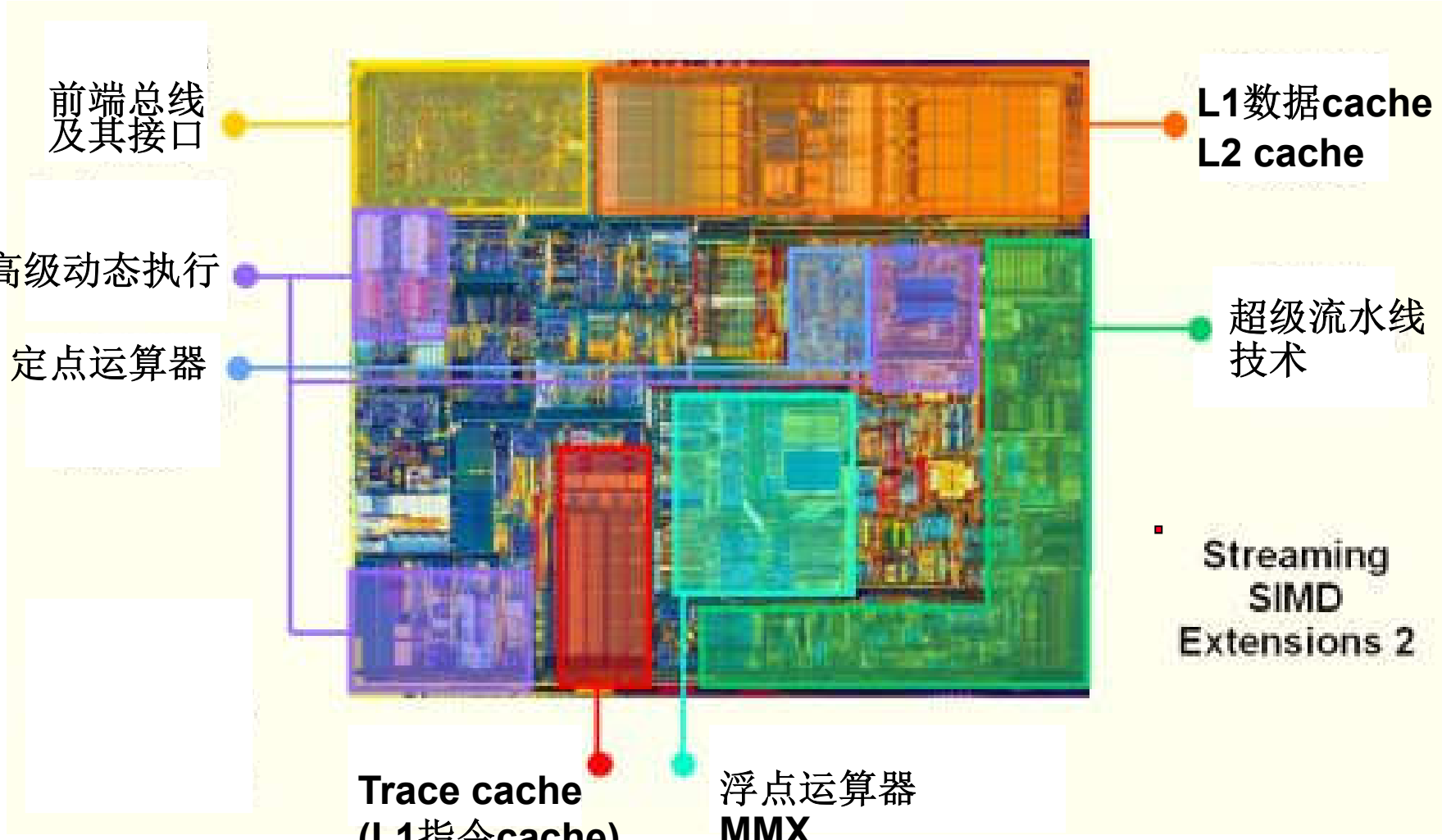
循环400000H=2^{22}次，每次同时有128/8=16个数（字节）相加

GPU简介

- CPU中的MMX、SSE以及AVX等采用SIMD技术
  - 针对绘图运算、3D游戏加速、视频编码等图形处理能力的增强而提出
  - 包含在Intel CPU中，但CPU不专用于图形处理，需从CPU中分离出来
- 随着计算机游戏产业的不断发展，形成了专门的图形处理功能部件：GPU（Graphics Processing Unit）芯片
  - 1999年NVIDIA发布第一款GPU
  - 早期GPU主要用于3D图形渲染等图形处理
  - 随着以CUDA为代表的GPU通用计算API的普及，GPU被广泛应用于石油勘测、天文计算、流体力学模拟、分子动力学仿真、生物计算等科学计算领域
  - GPU的含义也从原来专门的图形处理器转变为GPGPU（General-Purpose computing on GPU）

SKIP

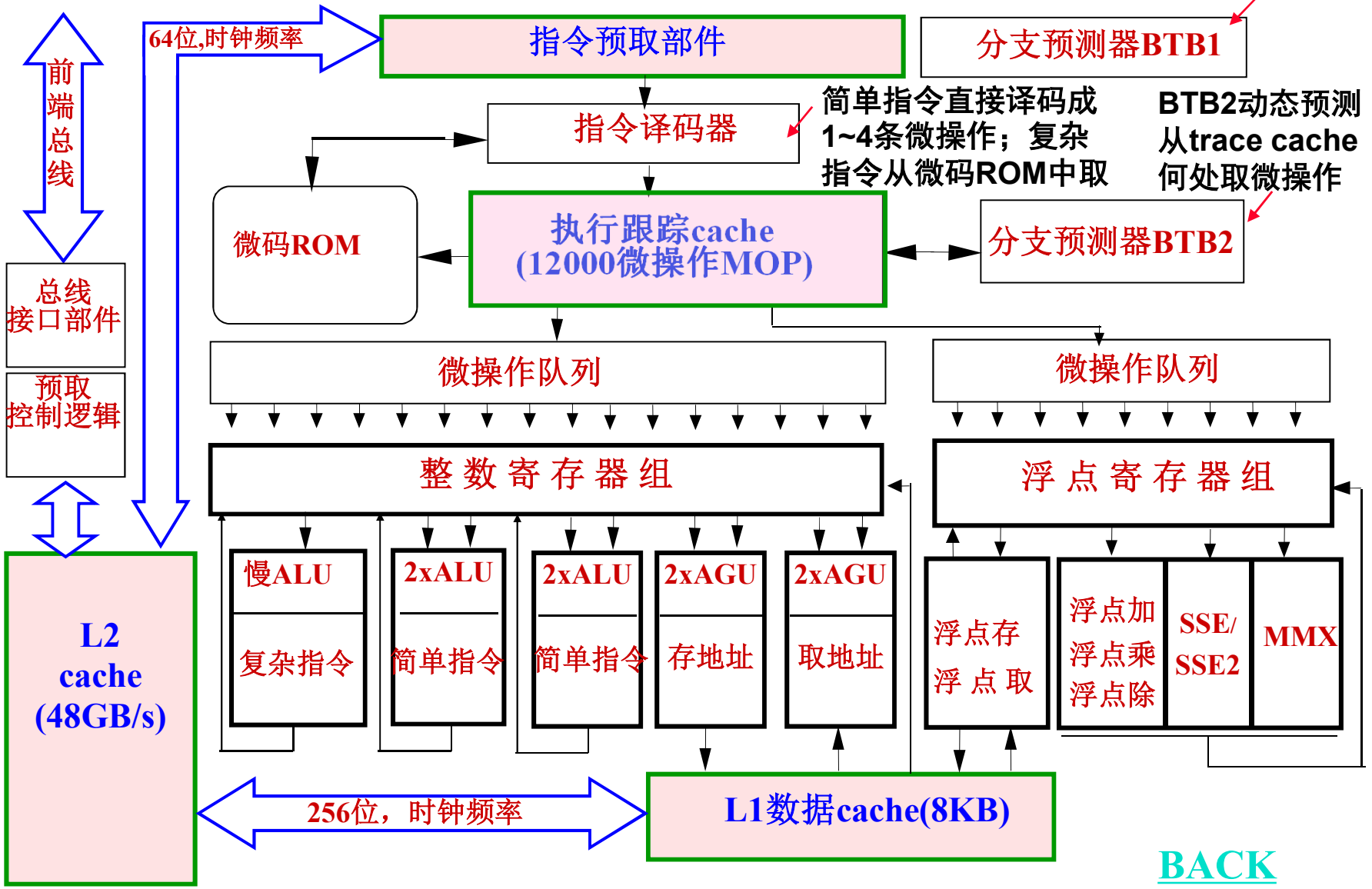
回顾：Pentium 4 处理器的芯片布局



回顾：Pentium 4 处理器的逻辑结构

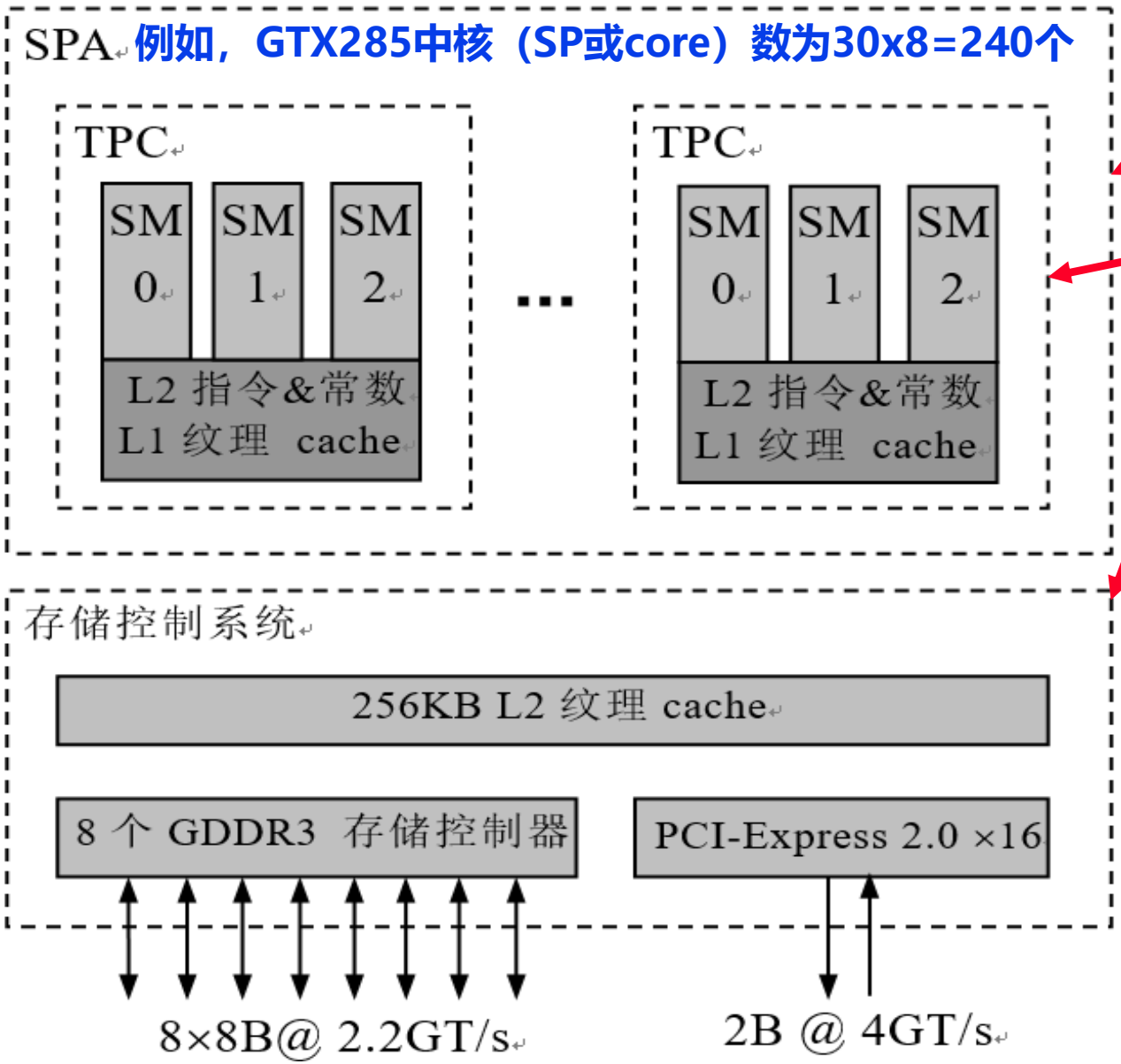
每个MOP相当于一条RISC指令，格式没公开

BTB1静态预测从L2的何处预取指令



BACK

NVIDIA GPU GT200 Tesla架构



- Tesla架构包含：
  - 流处理器阵列SPA
  - 存储控制系统

TPC：线程处理器群
Thread Processor Cluster

同一TPC中3个SM共享指令和常数L2cache、纹理L1cache、纹理处理部件

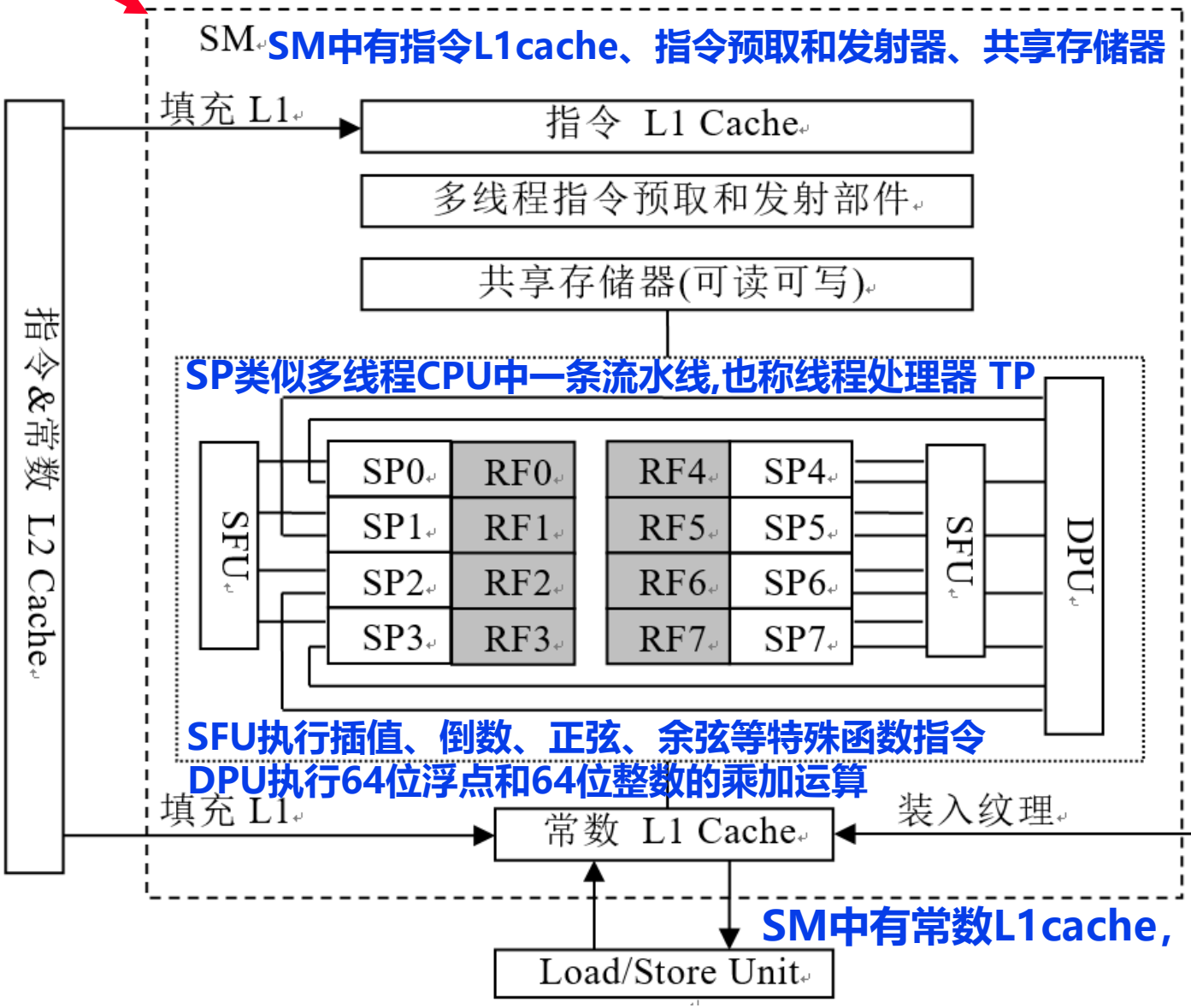
如：GTX285有10个TPC、8个存储控制器，每个SPA包含30个SM（流多处理器）

一个SM包含8个流处理器SP、2个特殊功能处理器SFP和1个双精度处理器DFU

存储控制系统包括存储控制器、光栅操作处理器ROP和二级纹理缓存

NVIDIA GPU GT200 Tesla架构

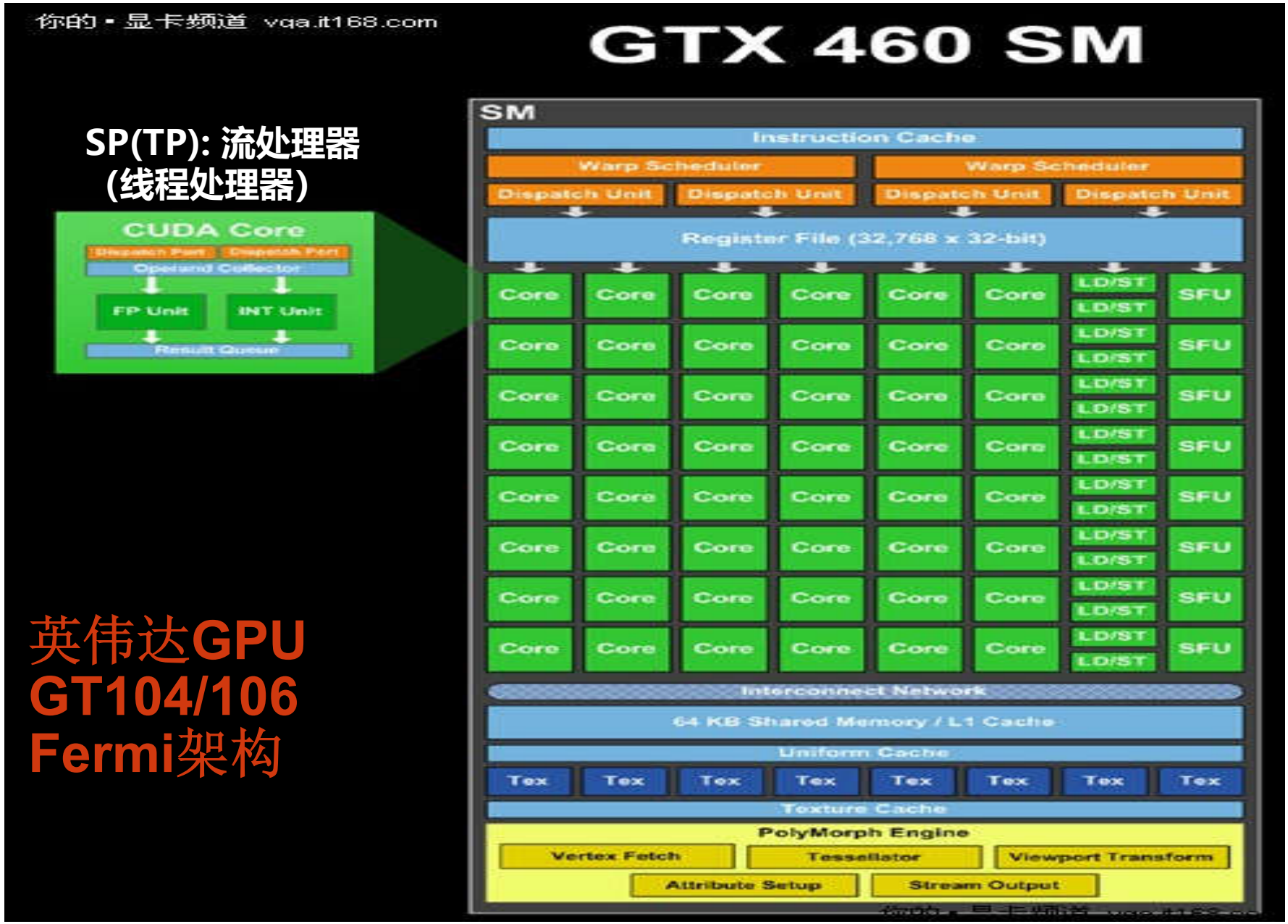
SM模型与传统SIMD有区别，NVIDIA称其为单指令多线程SIMT



SP核包含标量单精度浮点数和整数处理单元，可实现大部分运算指令，最多可并发执行64个线程，每个SP有1024个32位通用寄存器，根据SP所分配的线程数来分配寄存器。程序会声明其寄存器需求，通常编译器会为每个线程分配16~64个寄存器。例如，对于像素渲染程序，通常使用16个或更少的寄存器，这样，每个SP可运行64个像素渲染线程

每个SM包含8个SP，每个SP最多有64个线程，故一个SM上最多有512个线程

NVIDIA GPU GT104/106 Fermi架构



并行编程模式简介

- **应用可分以下几种类型：**
  - **计算密集型（Computation-Intensive）**
    - **数据量不大，但计算较为复杂**
    - **如：一个计算圆周率至小数点一千位以下的程序，在执行过程中绝大部份时间用在三角函数和开根号计算**
  - **数据密集型（Data-Intensive）**
    - **数据量巨大、但计算相对简单**
    - **如：海量网页词频统计问题**
    - **需要频繁读取数据，也称为I/O密集型**
  - **数据密集与计算密集混合型**
    - **数据量巨大且计算复杂**
    - **如：3D电影渲染**

针对不同特点的应用问题和不同的并行处理结构，可以有不同的并行编程模型以及并行程序设计语言

并行编程模式简介

- 几种主要的并行处理程序设计方式：
  - 共享存储变量方式（多线程并行程序设计）
    - 用于共享存储器多处理器系统
    - 通常将一个任务分解成若干个处理逻辑段（称为一个线程）
    - 线程间通过共享存储变量进行数据交换
    - Pthread：在串行程序中加入Pthread函数调用实现多线程并行处理。
    - 所有Pthread线程含有：

      一个线程标识符（有专门的线程创建、线程终止等Pthread函数）

      包括程序计数器PC的一组寄存器（用于存放各线程的现场信息）

      一组存储在结构（struct）类型变量中的属性，这些属性包括栈大小、调度参数以及线程所需的其他信息
    - OpenMP：在串行程序中插入编译指导命令指示哪个程序段可并行执行

      例如：在for语句前加编译指导语句“#pragma omp parallel for”，告诉编译器将for语句编译成多线程并行代码。在编译指导语句中需指定并行线程个数、线程中私有变量、对哪个变量进行规约以及如何规约等。OpenMP中还提供API函数，包括运行环境设置函数、锁操作函数和时间操作函数等

并行编程模式简介

- **几种主要的并行处理程序设计方式：**
  - 共享存储变量方式（多线程并行程序设计）
  - **消息传递方式**
    - **用于分布式存储器访问的并行处理系统（如集群）**
    - **通过消息传递（Message Passing）方式进行数据的交换。（采用分布式存储访问，无法通过执行LOAD/STORE指令访问另一节点私有存储器，因此，无法通过共享存储变量来交换数据）**
    - **每个子任务作为一个进程在各自独立的计算节点上并行执行，因此也可以被狭义地理解为多进程并行程序设计方式**
    - **最典型的是MPI（Message Passing interface，消息传递接口）标准**
    - **MPI：在串行程序中加入MPI并行编程接口函数，实现多进程并行处理，所有节点运行同一个程序。并行程序段部分由不同计算节点处理不同数据由专门的API接口函数来指定哪一段是并行程序段、共有多少计算节点/进程数目、如何进行同步通信以传递消息等**

并行编程模式简介

- **MPI编程的简单例子:**

```
#include <mpi.h>
#include <stdio.h>
main(int argc, char **argv)
{
  int num, rk;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, & num);
  MPI_Comm_rank(MPI_COMM_WORLD, &rk);

  printf("Hello world from Process %d of %d\n",rk,num);

  MPI_Finalize();
}
```

```
Hello world from Process 0 of 5
Hello world from Process 1 of 5
Hello world from Process 2 of 5
Hello world from Process 3 of 5
Hello world from Process 4 of 5
```

并行编程模式简介

- **几种主要的并行处理程序设计方式：**
  - 共享存储变量方式（多线程并行程序设计）（Pthread、OpenMP）
  - 消息传递方式（MPI）
  - **MapReduce大规模并行计算框架**
    - **由谷歌提出的一种面向大规模数据处理的并行处理模式和方法**
    - **可构建数百甚至数千个节点的基于集群的高性能并行计算平台**
    - **程序员仅需要编程实现Map和Reduce两个基本操作接口**

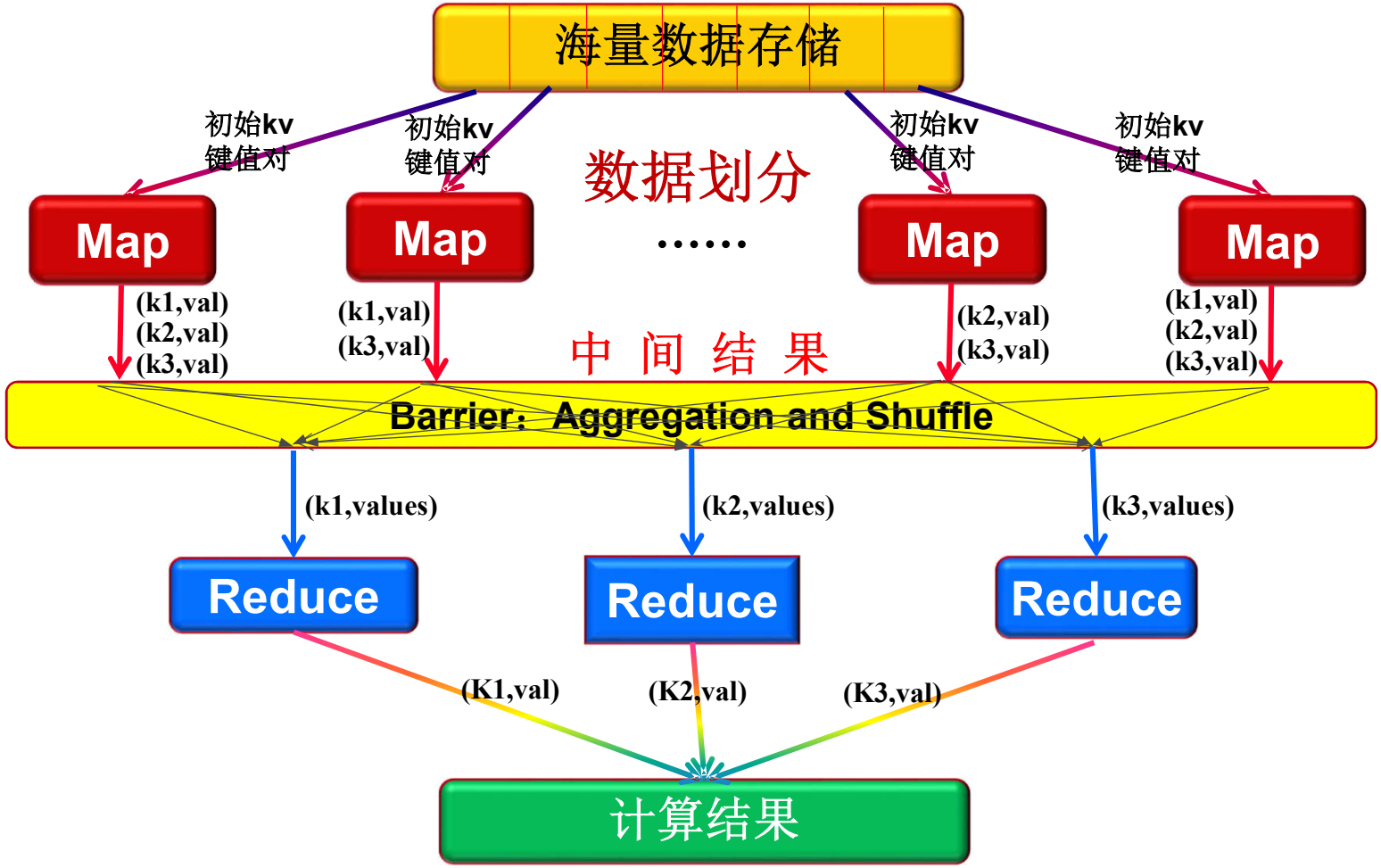
    Map操作主要负责对一组数据记录进行某种重复处理

    Reduce操作主要负责对Map操作的中间结果进行某种规约并输出结果

    进行reduce处理之前，必须等到所有map函数完成，故进入reduce前需有一个同步障(barrier)；这个阶段也负责对map的中间结果数据进行收集整理(aggregation & shuffle)处理，以便reduce能够计算最终结果，最终汇总所有reduce的输出结果即可获得最终结果

并行编程模式简介

基于Map和Reduce的并行计算模型



并行编程模式简介

基于MapReduce的处理过程示例--文档词频统计：WordCount

MapReduce处理方式

使用4个map节点：

map节点1:

输入：(text1, “the weather is good”)

输出：(the, 1), (weather, 1), (is, 1), (good, 1)

map节点2:

输入：(text2, “today is good”)

输出：(today, 1), (is, 1), (good, 1)

map节点3:

输入：(text3, “good weather is good”)

输出：(good, 1), (weather, 1), (is, 1), (good, 1)

map节点4:

输入：(text3, “today has good weather”)

输出：(today, 1), (has, 1), (good, 1), (weather, 1)

并行编程模式简介

基于MapReduce的处理过程示例--文档词频统计：WordCount

MapReduce处理方式

使用3个reduce节点：

reduce节点1:

输入：(good, 1), (good, 1), (good, 1), (good, 1), (good, 1)

输出：(good, 5)

reduce节点2:

输入：(has, 1), (is,1), (is,1), (is, 1),

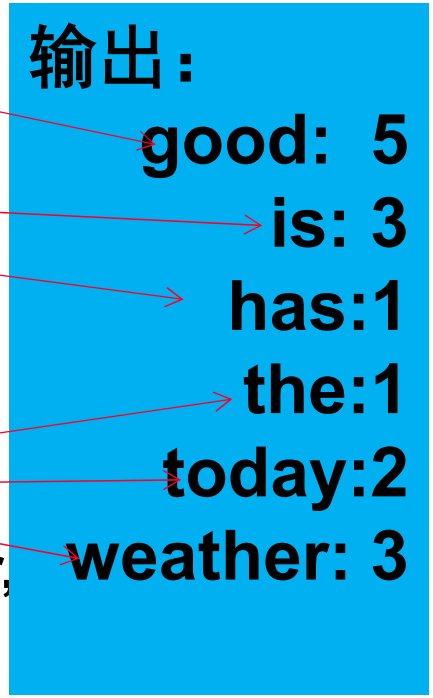
输出：(has, 1), (is, 3)

reduce节点3:

输入：(the, 1), (today, 1), (today, 1)

(weather, 1), (weather,1), (weather,

输出：(the, 1), (today, 2), (weather, 3)

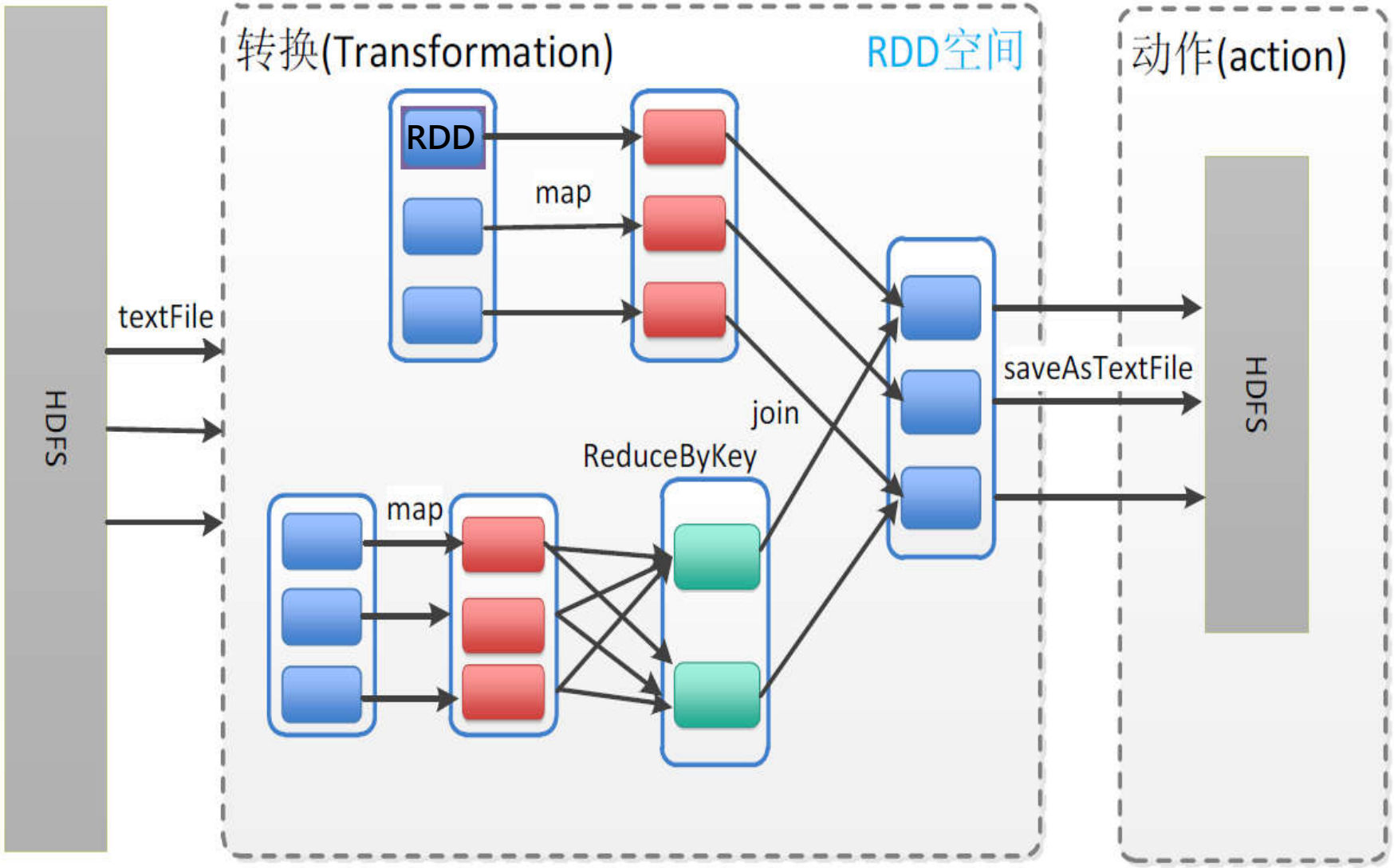


并行编程模式简介

- **几种主要的并行处理程序设计方式：**
  - 共享存储变量方式、消息传递方式（MPI）、MapReduce并行计算框架
  - **Spark大规模并行计算框架**
    - **MapReduce 容错性和负载均衡性好、编程方便，但每次计算都需从磁盘读数据和写结果，I/O开销很大**
    - **Spark 提出基于分布式内存的弹性数据集RDD，中间工作集以RDD的结构存储在分布式内存中，避免了重复读写磁盘的操作**
    - **RDD的创建：从HDFS加载、从内存数据分片处理、由已有RDD转换**
    - **转换(Transformation)操作: map、reduceByKey、join、filter 等**
    - **动作(action)和控制(control)操作：没有返回值**
    - **以上三类操作都可以并行执行**
    - **除了 RDD，Spark 还提供了一种重要的抽象共享变量机制 — Broadcast，以提供一种在各个节点上共享数据的机制**

并行编程模式简介

- **Spark编程模型**



并行编程模式简介

- **几种主要的并行处理程序设计方式：**
  - 共享存储变量方式、消息传递方式（MPI）、
  - MapReduce、Spark大规模并行计算框架
  - **CUDA并行编程模型**
    - **用于众核GPU+多核CPU的异构系统**
    - **对C/C++语言扩展了三类功能：层次结构线程组、同步栅和共享内存**
    - **应用“任务—数据块—数据元素”**
    - **GPU“TPC—SM—SP（SFU、DFU、LD/ST）”**
    - **CUDA“网格（grid）—线程块（thread block）—线程”**
    - **可在串行程序中调用一个并行执行的内核（kernel）函数程序，网格就是执行相同内核函数程序的一组线程块，相当于处理一个子任务**

层次结构线程组

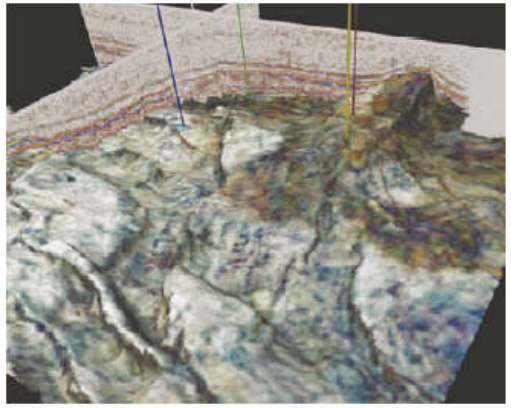
举例：一个应用包含两个顺序执行的子任务：

kernelA<<<6,10>>>(params)：含6个一维线程块，各线程块有10个线程

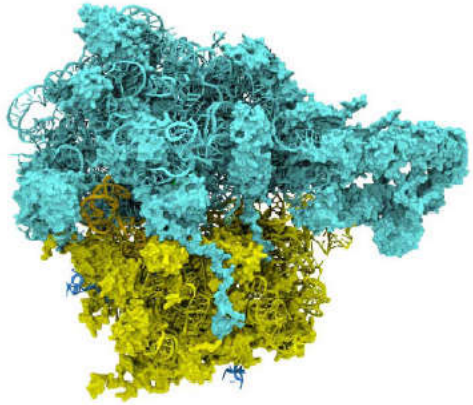
kernelB<<<(6,4), (8,3)>>>(params)：含6×4的二维线程块，各线程块含8×3个线程

必须在A和B之间设置同步栅

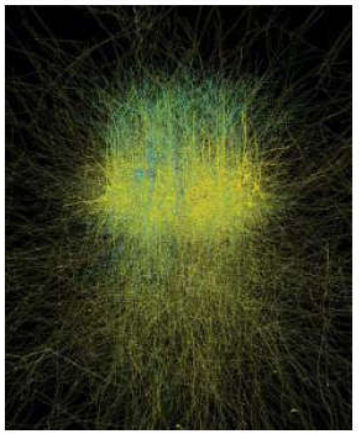
GPGPU应用举例



(a)



(b)



(c)

GPU及CUDA应用举例：

a、地震图像建模

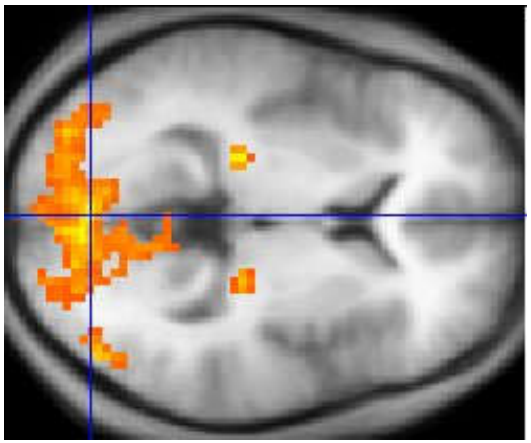
b、分子动力计算

c、脑神经电路模拟

d、衣物外观和运动

e、超声波图像处理

(d)



(e)

应用瓶颈：既懂应用算法，又懂编程模型和底层硬件结构的人不多

本章小结

- 按指令和数据处理方式划分（Flynn分类）
  - SISD、SIMD、MIMD
- 按地址空间的访问方式划分（MIMD）
  - 多处理器系统（共享存储）、多计算机系统（消息传递）
- 按访存时间是否一致划分（多处理器系统，在一个机箱中）
  - UMA：基于总线（侦听协议，如MESI）连接、基于互连网络（交叉开关、多级交换网络）连接，对称多处理器（SMP）是一种UMA系统
  - NUMA、CC-NUMA：本地和非本地的存储访问时间不一致，采用基于目录的cache一致性协议
  - 片级多处理器系统：多核系统，是一种UMA系统，如基于Core i7的计算机
  - 多线程技术：细粒度（时钟级切换）、粗粒度（阻塞时）、同时多线程
- 按连接方式的不同划分（多计算机系统，在一个或多个房间中，或分散在世界各地）
  - 集群：通过局域网互连。应用广泛，如google数据中心采用集群系统。
  - 网格：通过互连网互连。由于存在许多技术问题，故应用没有得到广泛开展。
- 向量处理机和Intel处理器中的SIMD技术、GPU（SIMT）
- 并行编程技术
  - 共享变量（多线程并行：Pthread、OpenMP）、消息传递（MPI）、MapReduce、Spark、CUDA（GPGPU）