

并行与分布式计算基础：第一部分补充资料

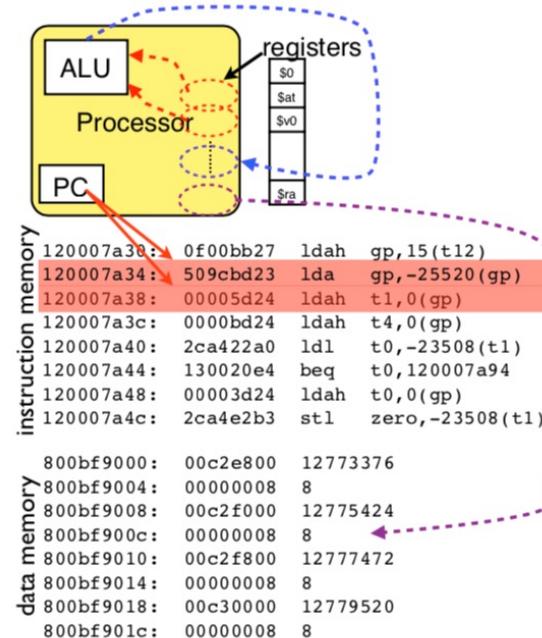
杨 超

北京大学数学科学学院

回顾：指令的执行方式

- 大致分四个步骤：取指、译码、执行、写回

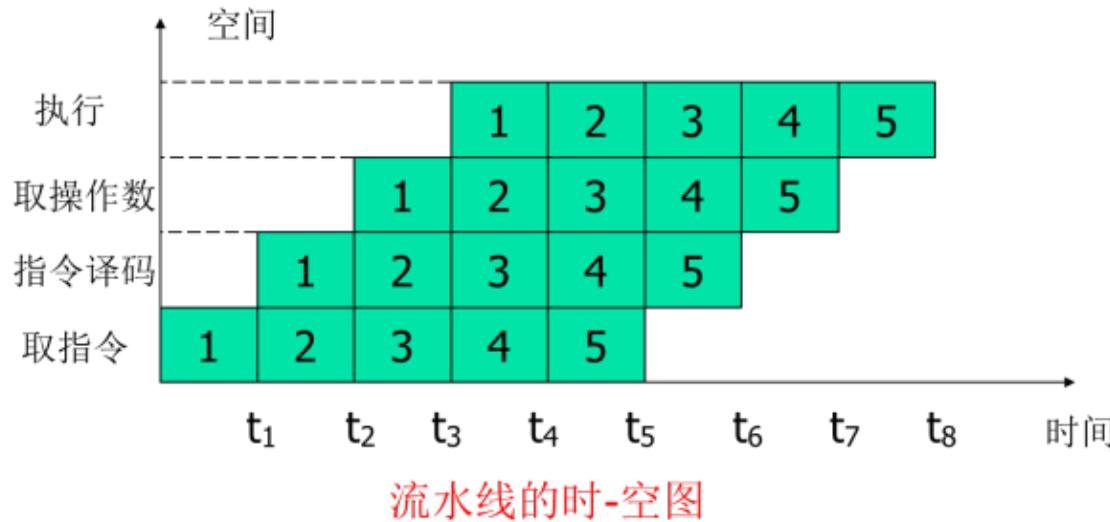
- Instruction fetch: where?
instruction memory
- Decode:
 - What's the instruction? **registers**
 - Where are the operands? **ALUs**
- Execute
- Memory access **data memory**
 - Where is my data?
- Write back **registers**
 - Where to put the result
- Determine the next PC



寄存器 (register) : 是CPU进行运算和保存结果的临时存储空间

指令流水线

- 例如某处理的所有指令分解为 $l = 4$ 个stage (深度), 每个stage花费 1 个时钟周期
- 下图展示了 $n = 5$ 条 (宽度) 指令流水线的示意图
- 暂不考虑流水线启动时间, 时间缩短为单条流水线的: $\frac{8}{20} = 40\%$



- 如果继续增大指令流水线条数, 时间缩短为原来的多少?

指令流水线

Operation timing:

$$\begin{cases} n & \text{operations} \\ \ell & \text{number of stages} \Rightarrow t(n) = n\ell\tau \\ \tau & \text{clock cycle} \end{cases}$$

With pipelining:

$$t(n) = [s + \ell + n - 1]\tau$$

where s is a setup cost

Asymptotic speedup is ℓ

$n_{1/2}$: value for which speedup is $\ell/2$

注: 时钟周期=1/主频

如: 主频 = 1GHz

时钟周期 = $1/(1\text{GHz}) = 1\text{ ns}$

指令流水线

Pipelining works for:
vector addition

Pipelining does not work for:
recurrences

```
for (i) {  
    x[i+1] = a[i]*x[i] + b[i];  
}
```

Transform:

$$\begin{aligned}x_{n+2} &= a_{n+1}x_{n+1} + b_{n+1} \\&= a_{n+1}(a_nx_n + b_n) + b_{n+1} \\&= a_{n+1}a_nx_n + a_{n+1}b_n + b_{n+1}\end{aligned}$$

指令流水线

- 当代处理器指令流水线支持的指令已经从单一指令类型到各种指令
- 由于理想的渐进加速比与流水线深度成正比，流水线深度也在增大
- 然而，更深的流水线需要更宽的流水线，即更多条相互独立的指令支撑，这越来越不现实，另一方面，分支预测错误导致的开销也会随着流水线加深而增大

Pentium III processor pipeline

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Pentium 4 processor pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

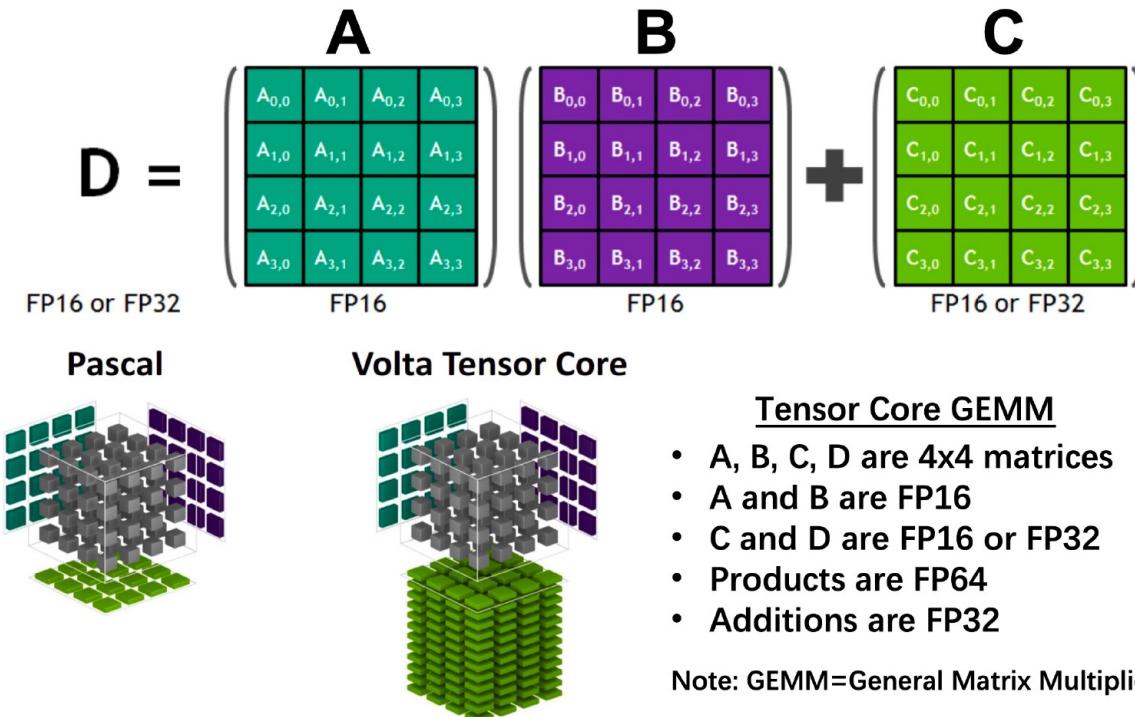
Misprediction is more “expensive” on Pentium 4’s.

当代处理器指令方面的其他变化

- 由于浮点计算的重要性，最重要的运算功能单元是Floating Point Unit (FPU)
- 从事乘加运算的功能单元称为Fused Multiply–Add (FMA)单元
- 除法往往计算速度很慢，一般需要10–20个时钟周期，也就是说，比加法和乘法慢10–20倍
- 流水线的超集是脉动阵列 (systolic array)，这一概念是80年代初提出来的，近年被Google的TPU采用
- 双精度一般指占据 $64\text{bit}=8\text{byte}=1\text{word}$ 内存地址的数据，单精度减半
- 随着机器学习的蓬勃发展，半精度甚至更低的精度逐渐得到了更大的重视

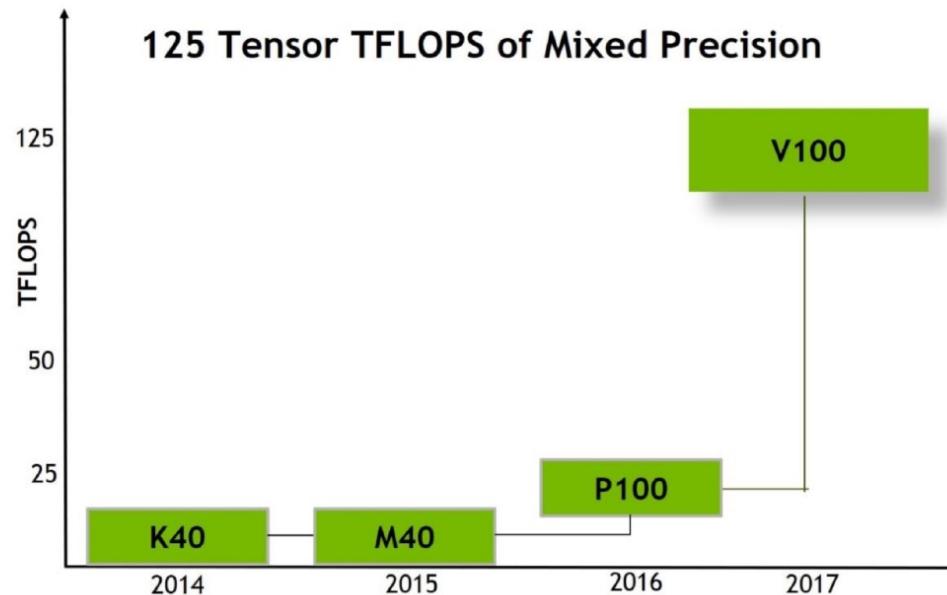
SIMD和混合精度的一个代表性例子：Tensor Core

- NVIDIA公司的Tesla V100 GPU配备了Tensor Core，可以在一个时钟周期完成如下混合精度的矩阵乘法运算，大幅提升了机器学习性能



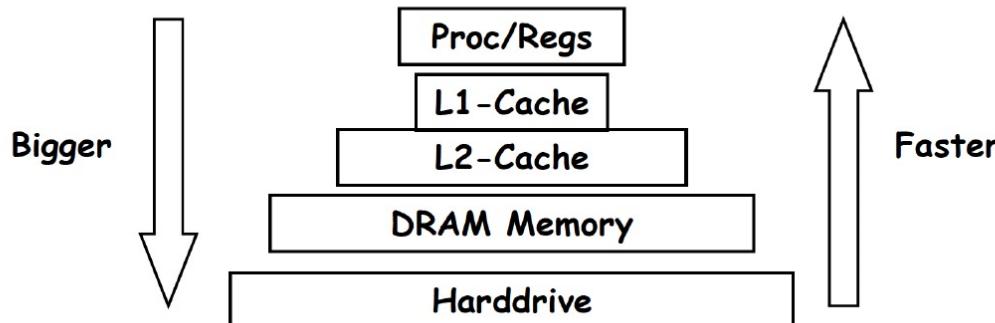
V100与之前几代GPU性能对比

- 双精度峰值 (FP64) 7.8 TFLOPS (P100: 5.3 TFLOPS)
- 单精度峰值 (FP32) 15.7 TFLOPS (P100: 10.6 TFLOPS)
- 半精度峰值 (FP16*) 125 TFLOPS (P100: 21.2 TFLOPS)
*: 基于 Tensor Core 张量计算核心的混合精度实现



回顾：多级存储技术

小（快）→ 大（慢）：寄存器 → 各级缓存 → ... → 内存 → 外存



This is an
AMD Operton CPU

Total Area: 193 mm²

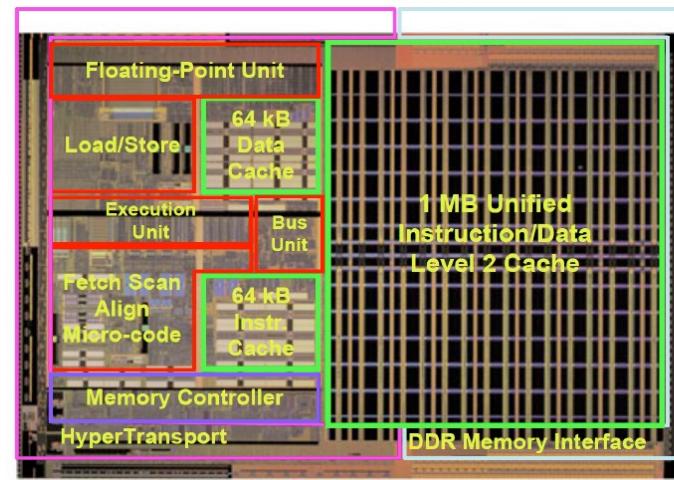
Look at the relative
sizes of each block:

50% cache

23% I/O

20% CPU logic

+ extra stuff



寄存器

- 寄存器 (register): 是CPU进行运算和保存结果的临时存储空间
- 容量极小、速度极快，是最为珍贵的存储资源，在编程时要充分利用
- 寄存器资源不够用时会发生寄存器漫溢 (spilling)，这尤其在GPU平台上较为常见

```
a := b + c  
d := a + e
```

大多时候编译器会设法把常用数据驻留在寄存器中

a stays resident in register, avoid store and load

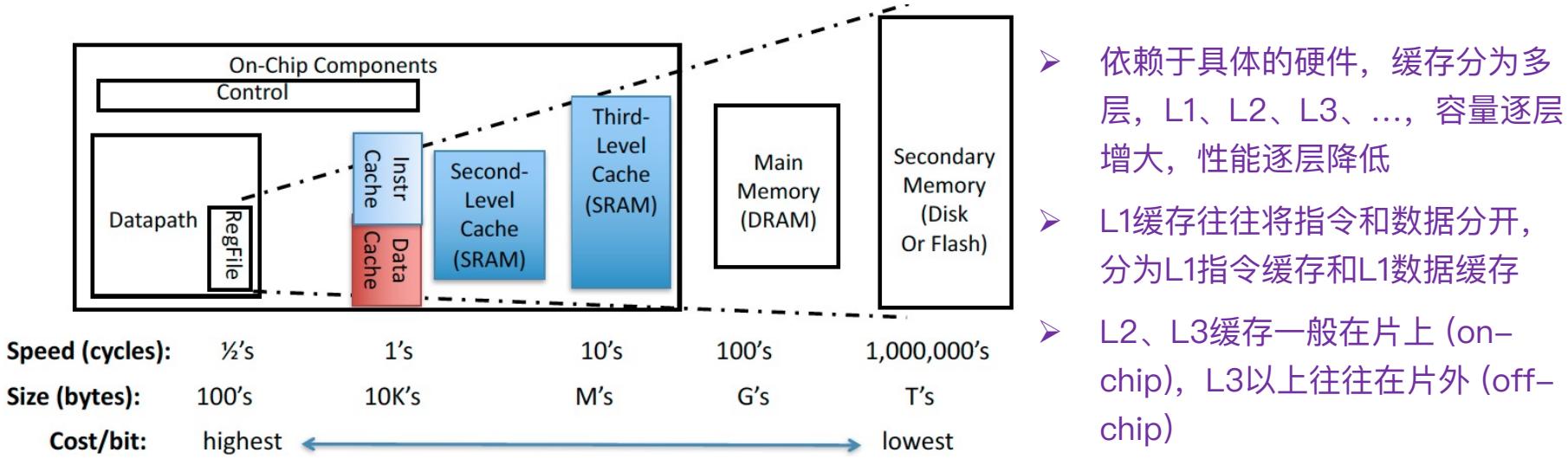
亦可通过声明寄存器类型变量来
人为指定变量存储于寄存器中

Hint to the compiler: declare *register variable*

```
register double t;
```

回顾：缓存

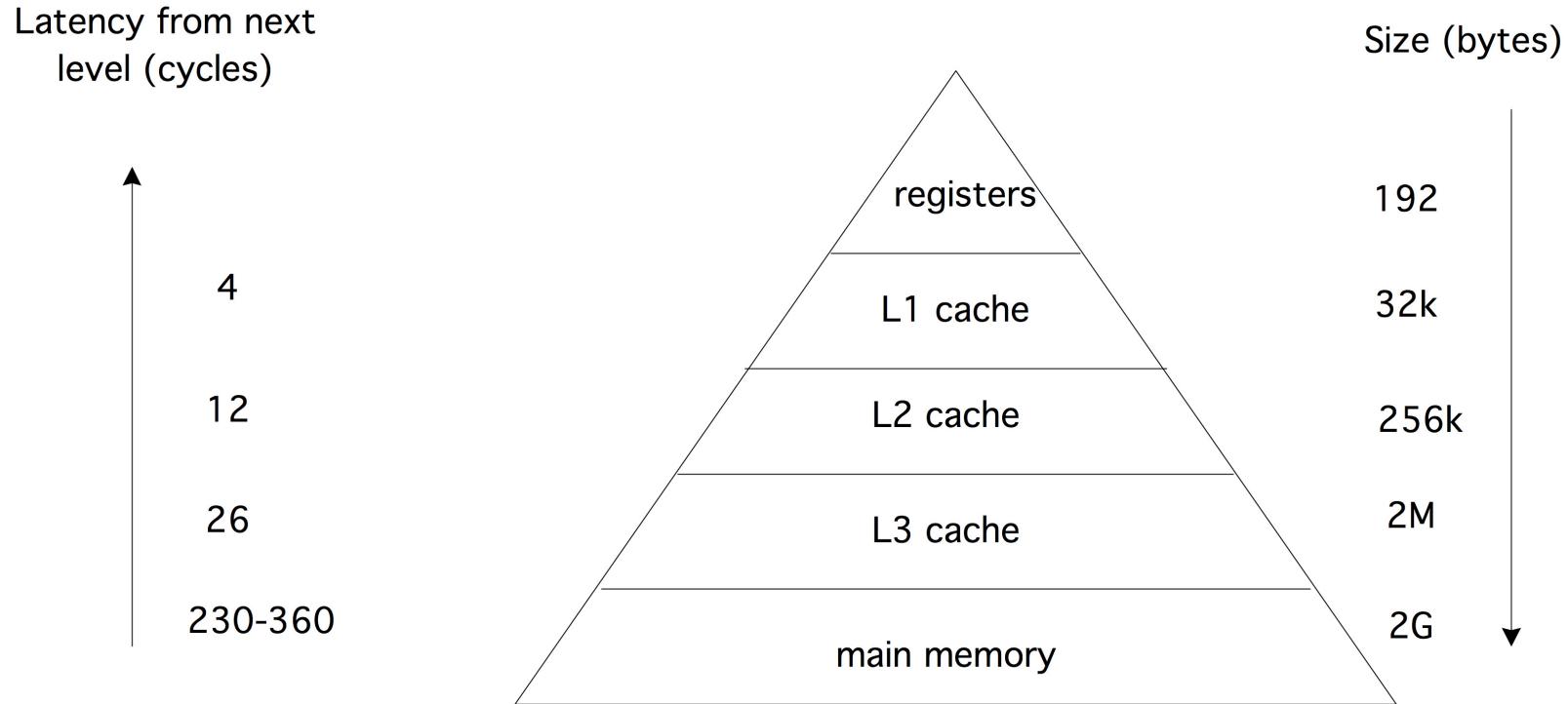
- 缓存 (cache): 位于寄存器和内存之间的用于保存指令和数据的高速存储空间



注：不同于内存采用的Dynamic Random–Access Memory (DRAM) 技术，缓存一般采用Static Random–Access Memory (SRAM) 技术，造价更高，耗费的晶体管更多，发热量也更大，也就是说性能高但做不大！

缓存在分级存储中的位置举例

◎ 举例：Memory hierarchy of an Intel Sandy Bridge



缓存的命中和失效

- 访问数据时，会从寄存器到各级缓存逐级查找，如果在缓存中，称为缓存命中 (cache hit)，否则缓存不命中/失效 (cache miss)，缓存失效的种类有：
 - 强制失效 (compulsory miss): 数据第一次被访问，且不在缓存中
 - 容量失效 (capacity miss): 数值此前在缓存中，但是由于缓存容量限制被冲掉——与缓存替换策略相关
 - 冲突失效 (conflict miss): 两个不同的数据项被映射到了同一个缓存位置——与缓存映射策略相关
 - 无效失效 (invalidation miss): 由于另一个处理器核心改写了数据，导致数据在本地缓存中的拷贝无效
- 缓存命中一个诱因是数据的复用 (reuse)，矩阵乘向量 $y = A x$ 操作的向量 x 的每个元素被访问了 n 次， n 为 A 的行数，因此有数据复用的机会

缓存线与数据局部性

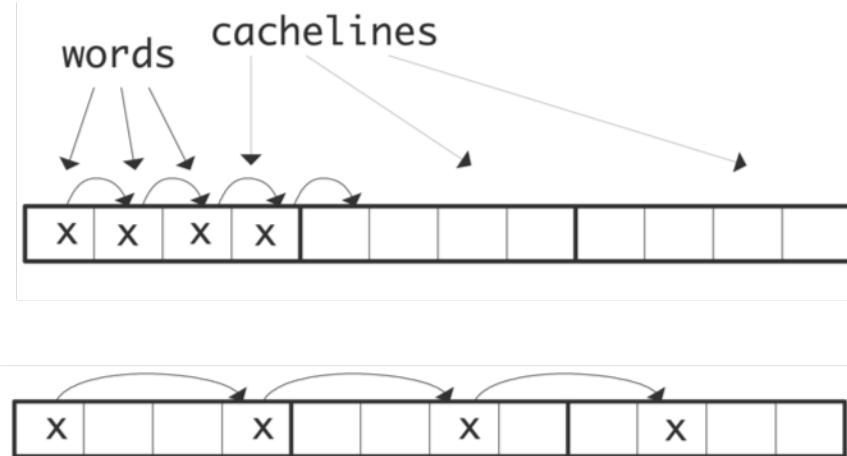
- 缓存线 (cache line): 数据进入和清出缓存的最小单位，一般是64或128 Byte，相当于8或16个双精度浮点数
- 缓存命中中的另一个诱因是数据局部性 (data locality)，即访问的数据在内存中的相邻性，由于缓存线的存在，算法的数据局部性越高，数据进入缓存的机会就越大
- 例如，某数组的跨步访问 (stride access)，跨步越小越好：

stride = 1 (连续访问): 缓存线被充分利用

```
for (i=0; i<N; i++)  
    ... = ... x[i] ...
```

stride = 3: 缓存线的1/3被利用

```
for (i=0; i<N; i+=stride)  
    ... = ... x[i] ...
```



缓存的替换与映射策略

- 缓存的替换 (eviction): 缓存容量小于内存，因此不可能所有数据永远驻留在缓存中，需要基于某种策略进行替换，具体策略有
 - 最近最少使用 (Least Recently Used, LRU) 替换
 - 先进先出 (First In First Out, FIFO) 替换
 - 随机替换
- 缓存的映射 (mapping): 缓存容量小于内存，需要确定内存中的数据地址与缓存中的地址对应关系，即映射策略，具体策略有
 - 直接 (direct) 映射
 - 全关联 (fully associative) 映射
 - k路关联 (k -way associative) 映射
- 一般而言，缓存的替换和映射都是系统底层确定的，程序员无法更改

缓存映射策略比较

- 例如：某缓存有12个缓存线，左图是直接映射(类似round-robin)，把数据地址按12取模后映射到不同的缓存线，同一缓存线中的数据相互之间有缓存冲突的风险

{0, 12, 24, ... }
{1, 13, 25, ... }
{2, 14, 26, ... }
{3, 15, 27, ... }
{4, 16, 28, ... }
{5, 17, 29, ... }
{6, 18, 30, ... }
{7, 19, 31, ... }
{8, 20, 32, ... }
{9, 21, 33, ... }
{10, 22, 34, ... }
{11, 23, 35, ... }

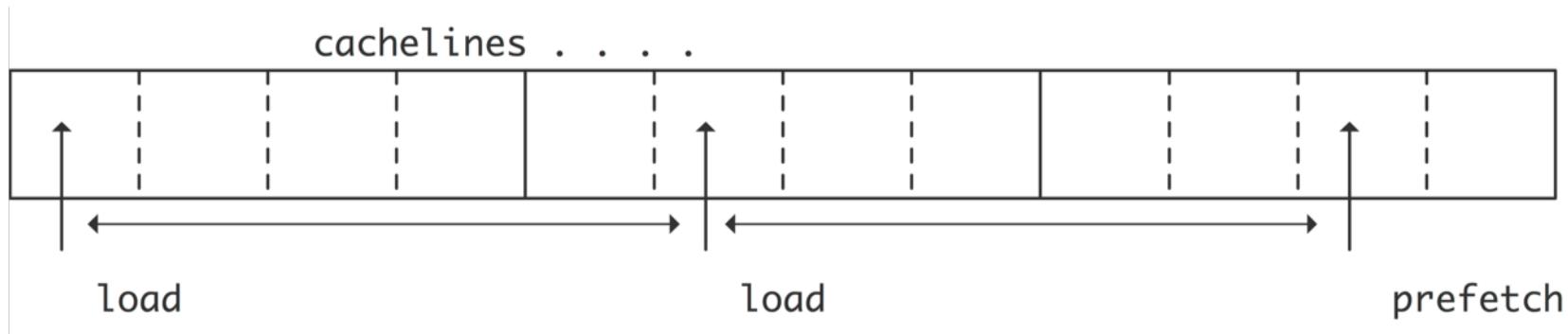
{0, 12, 24, ... } {4, 16, 28, ... }
{8, 20, 32, ... }
{1, 13, 25, ... } {5, 17, 29, ... }
{9, 21, 33, ... }
{2, 14, 26, ... } {6, 18, 30, ... }
{10, 22, 34, ... }
{3, 15, 27, ... } {7, 19, 31, ... }
{11, 23, 35, ... }

右图是3路关联映射，缓存线以3个一组与内存数据映射，可以避免同组缓存冲突小于3个的情况。

全关联映射则可以将任意内存数据映射给任意缓存线，能够最大程度降低缓存冲突失效，但造价高昂

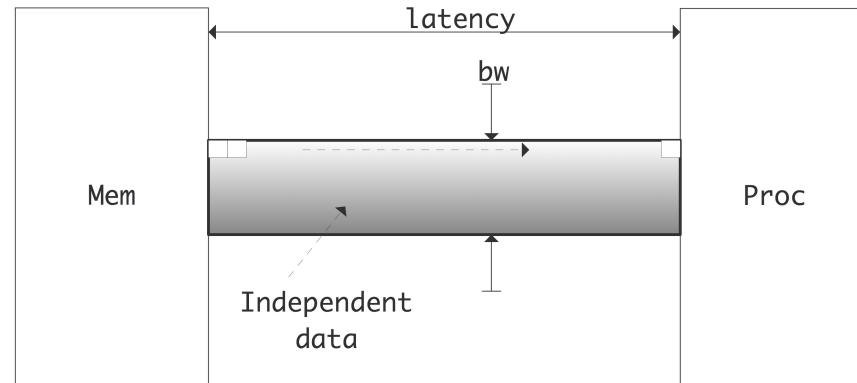
数据预取

- 预取 (pre-fetch): 如果程序顺序访问连续的缓存线, 系统会触发预取功能, 将之后的相邻数据从内存读入下一个缓存线, 从而隐藏后继内存访问的开销
- 这种硬件自动完成的数据预取称为硬件预取, 它能够触发的重要前提是数据局部性
- 有些系统支持程序员实现软件预取, 一般需要采用源语 (intrinsic) 实现, 即系统提供的某种底层编程接口, 有时也可以通过添加编译指示 (compiling hint) 实现



内存bank

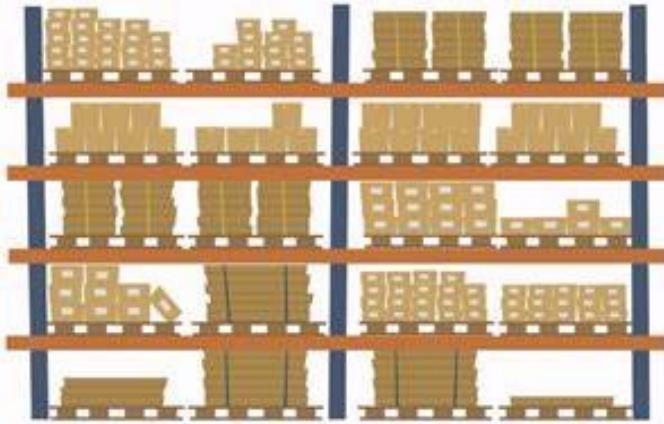
- 回忆：前端总线——类比为马路，为了实现高效的数据访问：需要缩短延迟——路要短，提高带宽——路要宽，提高并发度——车要多



- 内存中数据的组织按照bank进行分配——类比为车道，数据依次按照round-robin顺序与bank映射
- 如果访问的数据来自同一个bank，会引发bank conflict，导致性能降低，这在GPU上较为明显，在多级缓存系统上受缓存影响可以一定程度降低

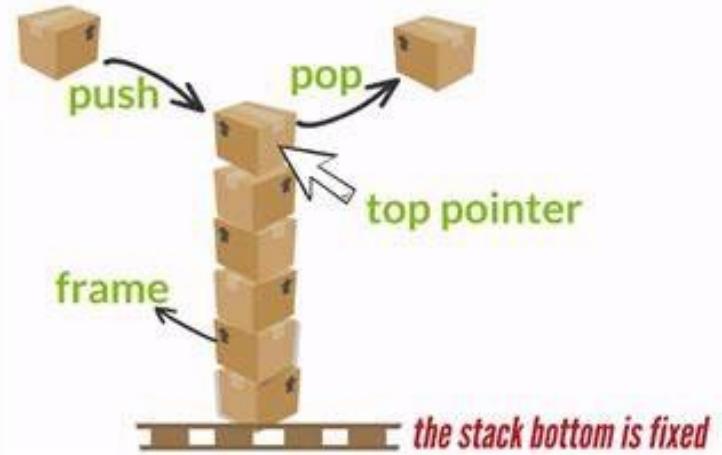
堆 (heap) 和栈 (stack)

Heap no order stores objects



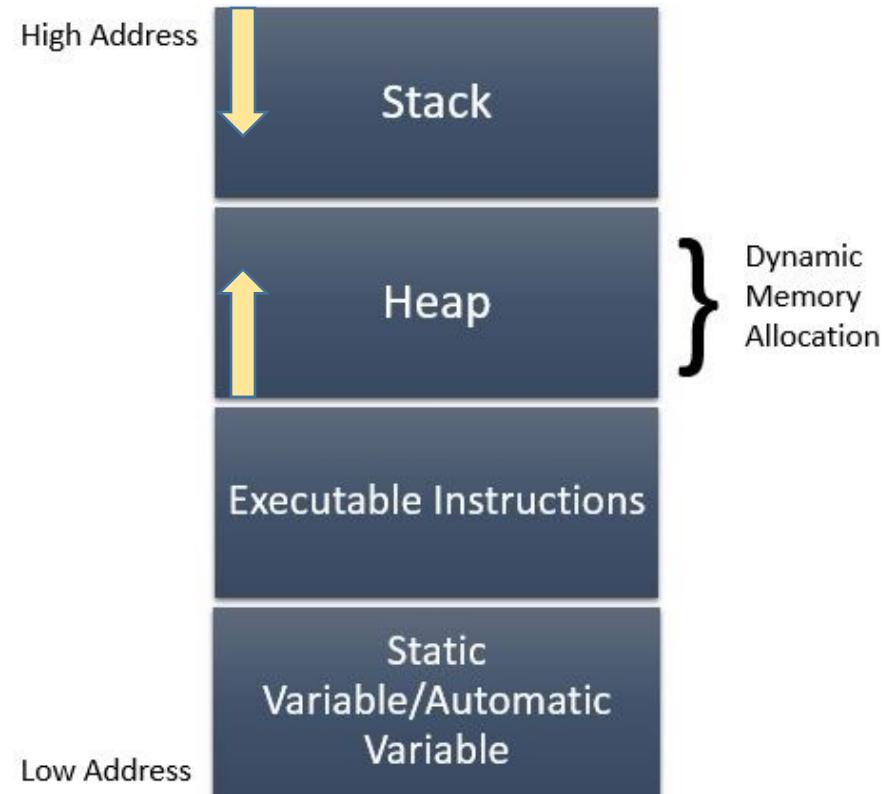
Stack

LIFO: last in, first out



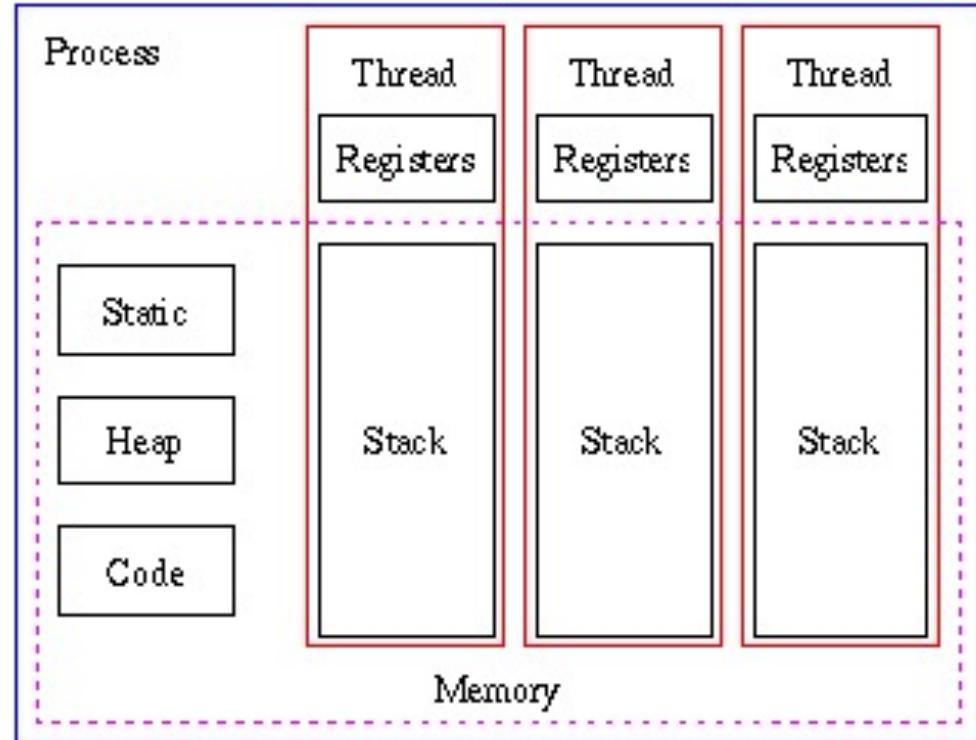
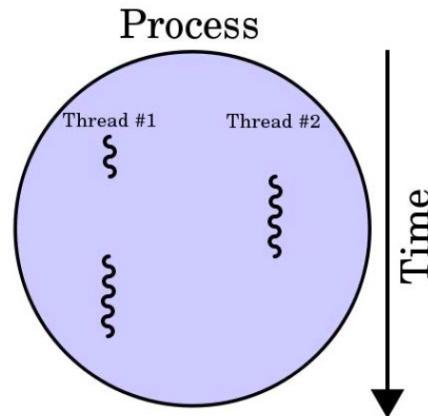
内存数据的组织

- 静态空间 (static area): 存储全局和静态变量，生存周期与程序一致
- 栈 (stack): 存储局部变量，后进先出，生存周期为变量所属的函数体/代码区
- 堆 (heap): 存储动态分配的变量，用户需要负责释放，与栈数据相比更灵活，分配开销更大，一般不如栈数据缓存友好



进程与线程

- 进程 (process): 指的是一个程序的执行者，拥有独立的程序、静态空间、栈空间、堆空间
- 线程 (thread): 一个进程可有一到多个线程，线程的栈空间相互独立，其他数据空间共享



线程的上下文 (context): 线程能访问的所有数据

虚拟内存与TLB

- 虚拟内存 (virtual memory): 内存空间不够用时，操作系统将部分数据以页 (page) 为单位换出 (swap out) 到硬盘，当该数据被需要时，将数据以页为单位从硬盘换入
- 这种换入换出是一个动态过程，每个程序执行时系统都会建立一个换页表 (page table)，保存逻辑页与实际内存页的对应关系
- 很多时候，程序访问的内存仅仅集中在少数内存页，可以通过“大页”方式提高性能
- Translation Look-aside Buffer (TLB): 是处理器拥有的用于保存常用页面信息的一种特殊缓存
- TLB一般采用关联映射和LRU的替换策略，可以保存64 到 512 个页面映射关系
- TLB命中率是衡量程序性能的一个指标，数据局部性好的程序往往访问的页面也较为集中，“大页”策略可以提高TLB命中率

从单核到多核

● 为什么采用多核?

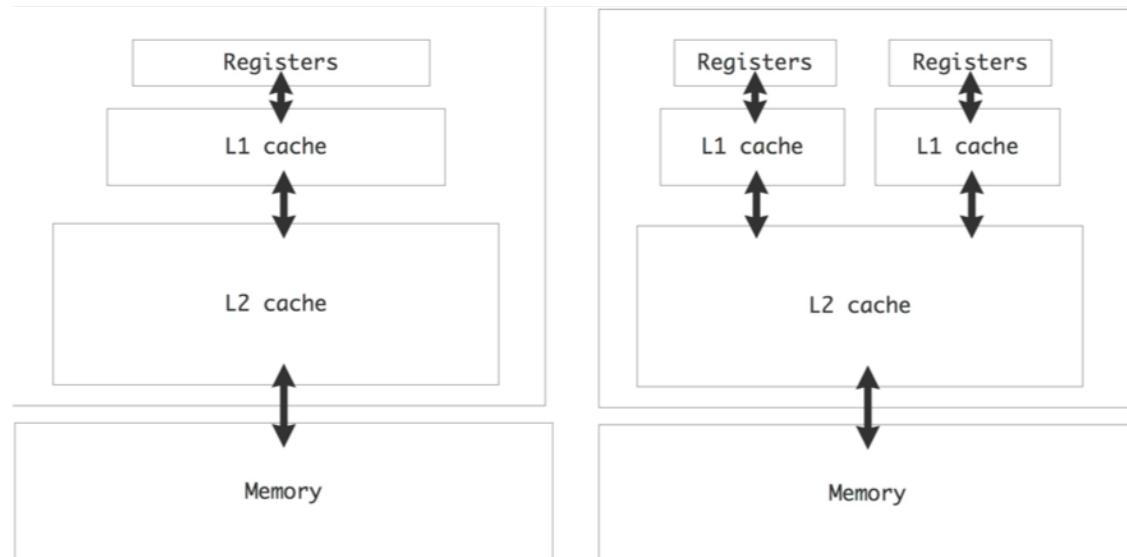
- 原因1：更节能，两个主频减半的核比一个核更节能(节省3/4)
- 原因2：指令/线程级并行(多发射、超标量、乱序执行、超线程...)不足以支撑更长的流水线

● 从单核到多核

- 理论性能继续稳步增长
- 将并发度转移给程序员

● 多核处理器的缓存

- 寄存器和L1私有
- L2、L3、...、内存共享



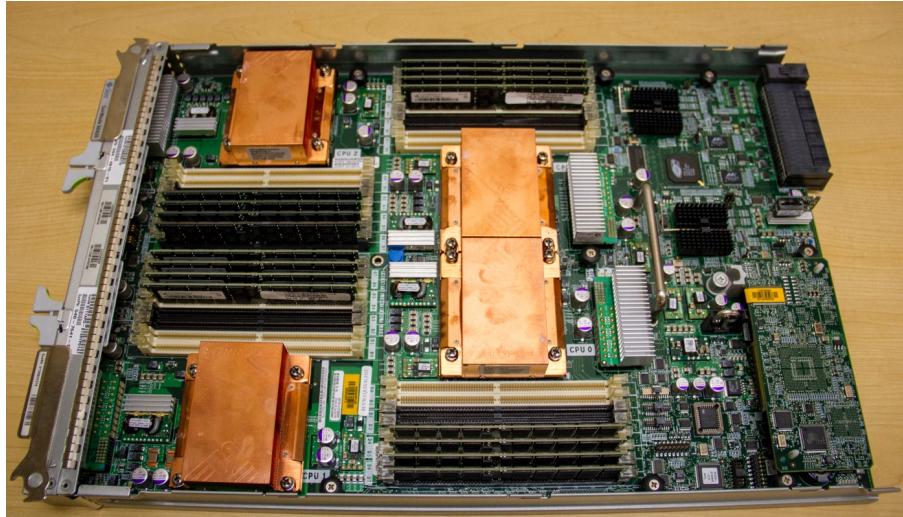
单核CPU的缓存设计

双核CPU的缓存设计

几个概念

- 核心 (core): 具备独立的逻辑单元、计算单元和寄存器的处理单元
- 插槽 (socket): 具有一个或多个核心的芯片 (chip) 通过引脚 (pin) 插入主板 (motherboard)
- 节点 (node): 具有一个或者多个插槽的主板构成的计算机，有时也会把一个节点在逻辑上分为多个节点
- 协处理器 (co-processor) : 通过PCI-Express通道连接在主板的一个或多个加速卡 (accelerator)，辅助CPU进行计算加速
- 网络 (network): 连接多个计算节点的连线，以后讲

计算节点举例



TACC Ranger超级计算机的计算节点
每个节点有4个CPU插槽



TACC Stampede超级计算机的计算节点
每个节点有2个CPU插槽和一个Intel Xeon Phi
加速卡

缓存一致性的意义

- 回忆：缓存的无效失效 (invalidation miss)
 - 由于另一个处理器核心改写了数据，导致数据在本缓存中的拷贝无效
 - (思考：还有哪几种缓存失效？)
- 缓存一致性 (cache coherence): 确保不同缓存中包含的同一数据的拷贝的准确性
- 归责
 - (1) 同一数据不同计算节点内存中的拷贝的准确性是由用户通过算法和编程维护的
 - (2) 同一数据在计算节点内不同核心的缓存中的拷贝的准确性是由系统维护的
- 也就是说，用户不用担心由于缓存的不一致性而导致出现错误，但是系统为了维护缓存一致性需要额外付出开销

缓存一致性的实现机制

处理器硬件底层提供了机制自动确保缓存一致性，主要有两种机制

- (1) 监听 (snooping)

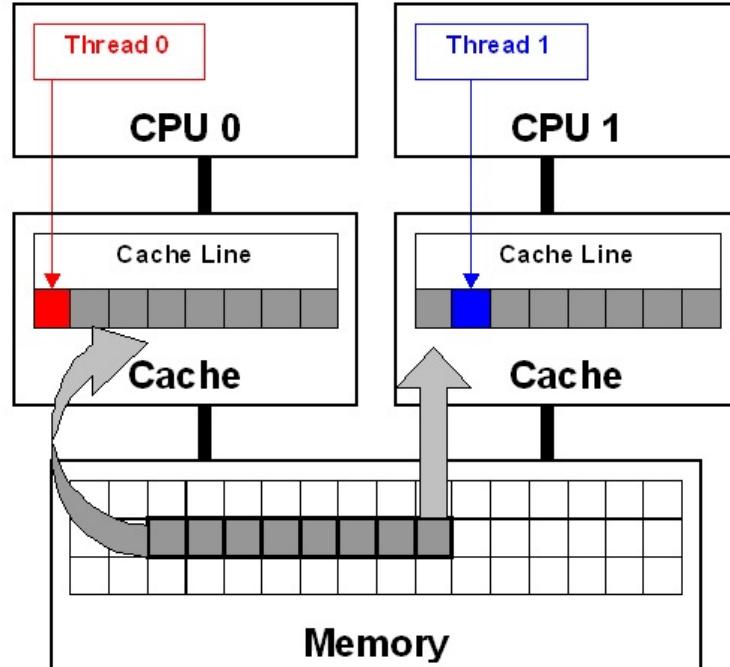
- 不断探测其他核心是否更改了数据，如果更改，就将自己缓存中的对应数据标记为无效
- 获取数据时，把数据请求发送给所有核心的缓存，只要有缓存含有有效数据就返回该数据，否则从内存中获取

- (2) 标签目录 (tag directory)

- 建立一个数据表格，保存每个缓存线都在哪些缓存中有效
- 获取数据时，从含有有效数据的缓存中获取，如果都没有，就从内存中获取
- 标签目录可以集中维护，也可以分摊给不同核心协同维护（例如Intel Xeon Phi）

伪共享

- 伪共享 (false sharing): 不同核心同时改写同一个缓存线的不同数据，系统为了确保缓存一致性，会将缓存线在不同核心间反复移动，导致性能下降



- 缓存颠簸 (cache thrashing/bouncing): 缓存线在不同核心间反复移动的现象

伪共享

◎ 举例

```
local_results = new double[num_threads];  
#pragma omp parallel  
{  
    int thread_num = omp_get_thread_num();           不同核心访  
    for (int i=my_lo; i<my_hi; i++)                问共享数组  
        local_results[thread_num] = ... f(i) ...      的相邻位置  
    }  
    global_result = g(local_results)
```

◎ 编写程序时如何避免伪共享

- (1) 数据私有化
- (2) 共享数据的访问地址保持“间隔” (Q: 怎么理解?)

TLB的一致性问题

● (回顾) 什么是TLB

- TLB是处理器拥有的用于保存常用页面信息(逻辑页–物理页映射关系)的一种特殊缓存
- 任何一次内存的读或写都会触发TLB访问，因此TLB的性能十分重要

● 一旦某个处理器核心修改了页面映射表，与之相关的所有核心的TLB都需要修改

● 由此可见，TLB也存在数据一致性的问题

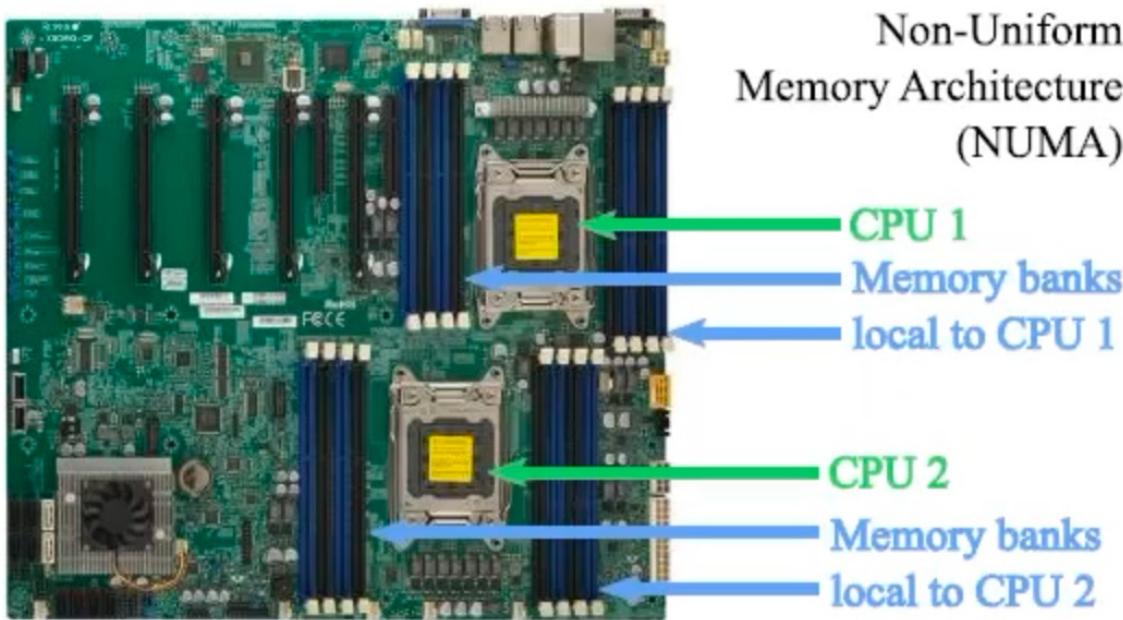
- 缓存一致性：不同核心所拥有的缓存之间的数据一致性，由硬件维护
- TLB一致性：不同核心所拥有的TLB之间的页面映射的一致性，一般由OS维护(如x86)

● OS维护TLB一致性又可大致分为两种情况

- 同一个socket内不同核心之间的TLB一致性，一般通过local flush实现
- 同一个node内不同socket之间的TLB一致性，一般会触发TLB shootdown——OS发送一个特殊的进程级中断，直到该中断请求成功完成，因此代价较高

NUMA节点的内存特性

- NUMA系统可以理解为具有“多条马路”的冯氏计算机
- 在同一个socket内，有多个memory bank，就好比“每条马路有多个车道”



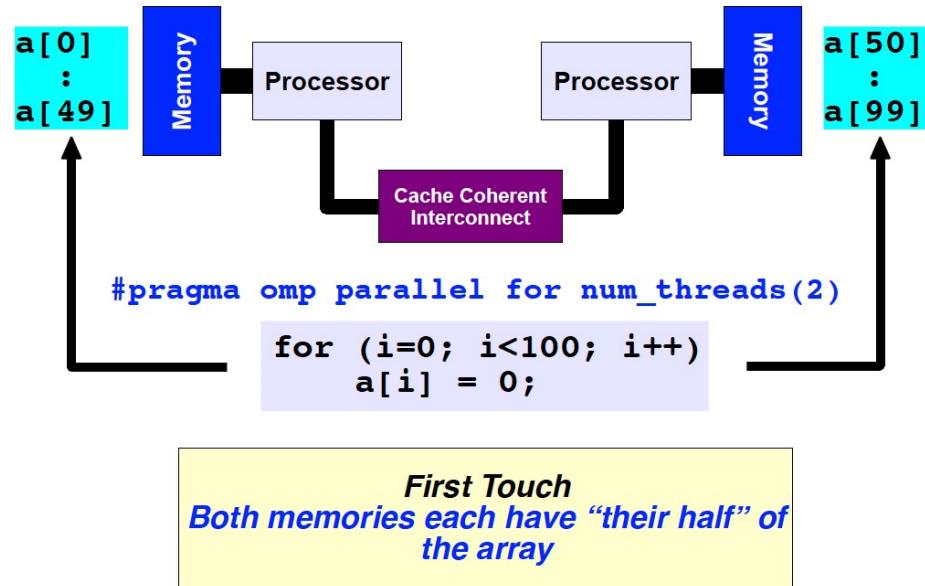
NUMA节点的first touch原则

- First touch原则：动态分配的内存数据直到第一次被使用时才真正被分配
- 例如，下述OpenMP并行程序中，array会被分配在主线程所在的socket，导致后续的多线程计算访存性能下降

```
double *array = (double*)malloc(N*sizeof(double));
for (int i=0; i<N; i++)
    array[i] = 1;
#pragma omp parallel for
for (int i=0; i<N; i++)
    .... lots of work on array[i] ...
```

NUMA节点的first touch原则

- 解决方法：动态分配内存后，以多线程并行方式赋初值(不管有无必要)，然后再使用



- 此外，一些OS还支持next touch原则，采用不同并行策略访问同一片内存时有效

计算密度的进一步讨论

● 为了充分利用多级缓存，一个重要的途径就是数据复用 (data reuse)

- 一方面，好的编程技巧能够增大数据复用的机会
- 另一方面，算法或程序本身的计算密度也是能否数据复用的一个前提

● 一般而言，计算密度越大，数据复用机会也越大，也越有可能隐藏访存开销

● 几个例子

- $\forall i: x_i \leftarrow x_i + y_i$ 向量加法：每次迭代有1次计算、3次访存（2读+1写），AI = 1/3
- $\forall i: x_i \leftarrow a x_i + y_i$ axpy：每次迭代有2次计算、3次访存（2读+1写），AI = 2/3
- $\forall i: s \leftarrow s + x_i \cdot y_i$ 向量内积：每次迭代有2次计算、2次访存（2读+0写），AI = 1
- $\forall i,j: c_{ij} = \sum_k a_{ik} b_{kj}$ 矩阵乘：共有 $2n^3$ 次计算（1乘1加）和 $3n^2$ 次访存（2读1写），AI = 2n/3

数据的局部性

- 除了一些特殊处理器 (例如Cell处理器、申威众核处理器、及某些GPU), 大多数系统不支持对缓存和和寄存器显式地编程控制
- 程序所表现出的数据局部性 (data locality)是提升缓存和和寄存器命中的重要判据
- 数据局部性大致分两类
 - 时间局部性 (temporal locality)
 - 空间局部性 (spatial locality)
- 此外, 在多核系统还有核局部性 (core locality), 单个核心在内存中访问的数据的相邻程度, 如果核局部性差, 不同核心访问的数据地址相互交错, 为了保持缓存一致性, 会额外占用内存带宽 (回顾: 缓存颠簸)

时间局部性

- 数据的时间局部性：被用过的数据短时间内还会被使用
- 依据：变量的重复利用 (SW), LRU缓存替换策略 (HW)
- 例如：

```
for (loop=0; loop<10; loop++) {  
    for (i=0; i<N; i++) {  
        ... = ... x[i] ...  
    }  
}
```



```
for (i=0; i<N; i++) {  
    for (loop=0; loop<10; loop++) {  
        ... = ... x[i] ...  
    }  
}
```

通过交换两层循环的顺序， $x[i]$ 具有内层循环不变性 (invariant)，因此，访问 $x[i]$ 的时间局部性得到提升， $x[i]$ 有很大机会进入寄存器

空间局部性

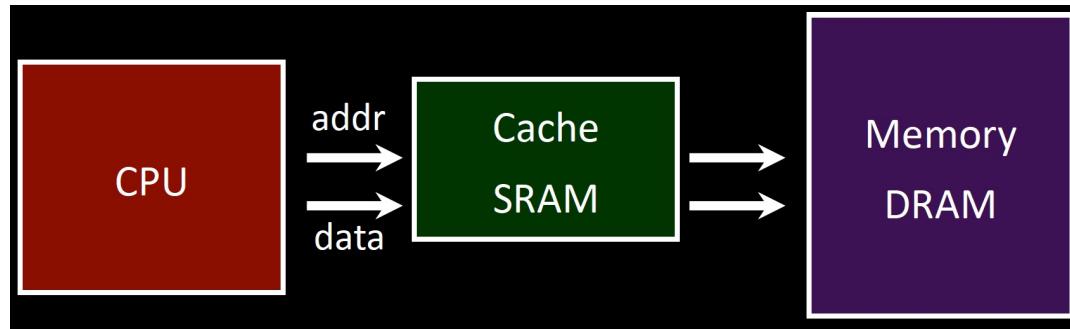
- 数据的空间局部性：被访问的数据的内存空间地址的临近程度
- 依据：数组、结构体等数据结构 (SW)，缓存线、TLB (HW)
- 例如：不同跨步情况下的数组遍历问题

```
for (i=0; i<N*s; i+=s) {  
    ... x[i] ...  
}
```

如果跨步为1，缓存线的利用率可以达到100%，否则，随着跨步增大，缓存线的利用率也不断下降，最低达到 $1/L$ ，其中L是缓存线大小

数据在缓存中的“写”与“读”

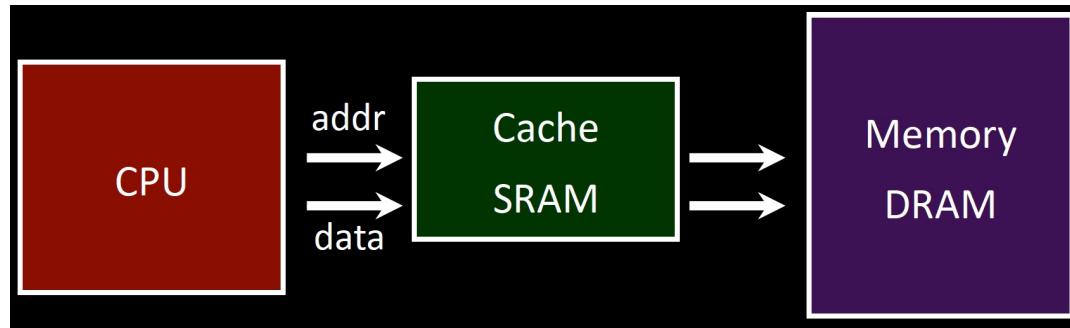
- 数据在缓存中的“写”(read/store)操作与“读”(write/load)操作有很大不同
- 因为写操作代价高，并且逻辑更复杂，涉及到把数据写到哪里的问题！



- 如果缓存中有该数据
 - 非写 (no write): 把对应的缓存线标记为无效，只写到内存
 - 写穿 (write-through): 改写缓存以及内存中的对应数据
 - 写回 (write-back): 只写到缓存，当且仅当这之后该缓存被清出时写入内存

数据在缓存中的“写”与“读”

- 数据在缓存中的“写”(read/store)操作与“读”(write/load)操作有很大不同
- 因为写操作代价高，并且逻辑更复杂，涉及到把数据写到哪里的问题！



- 如果缓存中没有该数据
 - 写分配 (write-allocate): 为该数据分配缓存线，可以写穿，也可以写回
 - 非写分配 (no-write-allocate): 忽略缓存，直接写入内存

数据局部性案例分析

- 矩阵乘

$$\forall_{i,j}: c_{ij} = \sum_k a_{ik} b_{kj}$$

- 三重循环，不同循环嵌套方式具有不同的时间和空间局部性，从而性能不同

```
for i=1..n
    for j=1..n
        for k=1..n
            c[i, j] += a[i, k] * b[k, j]
```

```
for i=1..n
    for k=1..n
        for j=1..n
            c[i, j] += a[i, k] * b[k, j]
```

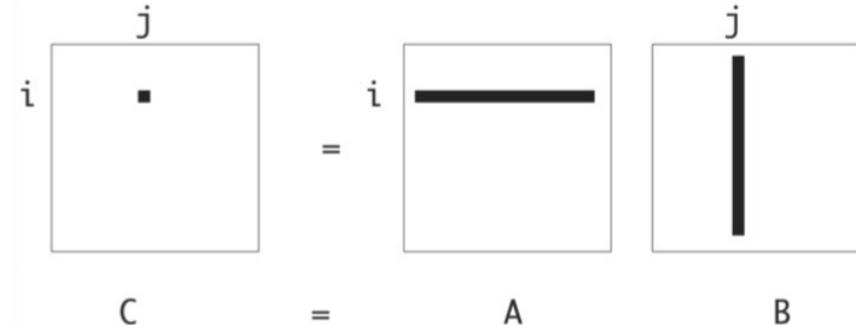
下面的分析中，假定二维数组采用C语言的行优先 (row-major) 存储

数据局部性案例分析

◎ 矩阵乘

$$\forall_{i,j}: c_{ij} = \sum_k a_{ik} b_{kj}.$$

```
for i=1..n  
  for j=1..n  
    for k=1..n  
      c[i, j] += a[i, k] * b[k, j]
```



◎ 方式一：i-j-k 模式

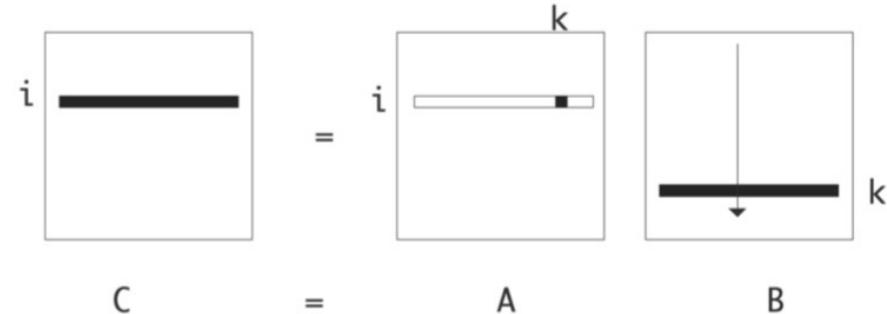
- $c[i, j]$: 在最内层循环不变，时间局部性高，对寄存器友好，每个元素只需1次访存
- $a[i, k]$: 按行遍历，空间局部性高，对缓存线友好，每个元素需 n/L 次访存
- $b[k, j]$: 按列遍历，时间和空间局部性都很差，每个元素需 n 次访存

数据局部性案例分析

◎ 矩阵乘

$$\forall_{i,j}: c_{ij} = \sum_k a_{ik} b_{kj}.$$

```
for i=1..n  
  for k=1..n  
    for j=1..n  
      c[i,j] += a[i,k]*b[k,j]
```



◎ 方式二：i-k-j 模式

- $c[i,j]$: 虽然是按行遍历，但因为是写操作，无法充分利用缓存线，代价较大
- $a[i,k]$: 按行遍历，时间、空间局部性都高，对缓存线和寄存器友好，每个元素1次访存
- $b[k,j]$: 按行遍历，空间局部性高，对缓存线友好，每个元素需 n/L 次访存

如何编写高性能程序

- 选择一门编程语言，并利用好该语言的一些重要特性

- C/C++、Fortran、Java、Python、Julia、Go、Matlab(?)、...

- 调用高性能库函数

- Intel MKL、NVIDIA CUDA Toolkit、AMD ACML、...

- 使用专用软件

- 高性能应用软件：PETSc、LAMPS、VASP、OpenFOAM、...
 - 领域编程语言 (domain specific language, DSL)
 - 自动性能调优 (autotuning)

- 了解编译器能力的“边界”，帮助编译器提升程序性能 【今天讨论】

- ILP、Register、Cache、TLB、内存、...

循环展开

- 为了更好地利用指令集流水线，通常采用循环展开 (loop unrolling) 的手段
- 例如，向量内积计算

```
for (i=0; i<N; i++)  
    s += a[i]*b[i]
```

- 通过循环展开并引入临时变量，消去变量s的读写依赖，增大流水线利用效率

```
for (i = 0; i < N/2-1; i++) {  
    sum1 += a[2*i] * b[2*i];  
    sum2 += a[2*i+1] * b[2*i+1];  
}
```

注：这里利用了加法的交换律和结合律

循环展开

● 在此基础上，去掉数组指标运算中的乘法

```
for (i = 0; i < N/2-1; i++) {  
    sum1 += *(a + 0) * *(b + 0);  
    sum2 += *(a + 1) * *(b + 1);  
  
    a += 2; b += 2;  
}
```

● 进一步，分离乘法和加法运算

```
for (i = 0; i < N/2-1; i++) {  
    sum1 += temp1;  
    temp1 = *(a + 0) * *(b + 0);  
  
    sum2 += temp2;  
    temp2 = *(a + 1) * *(b + 1);  
  
    a += 2; b += 2;  
}  
s = temp1 + temp2;
```

循环展开

- 上述是循环展开2次的结果，循环展开的次数可以继续增加
 - 潜在收益：增大可流水的运算个数，从而提升流水线利用率
 - 潜在问题：耗费寄存器资源，引发寄存器漫溢 (register spilling)
- 由此可见，循环展开的次数不是越多越好，而是适度为宜
- 此外，循环展开的程序可能会需要在循环前和/或循环后做一些额外运算
- 如下是随着循环展开次数，程序的耗时 (单位：节拍) 变化情况

1	2	3	4	5	6
6794	507	340	359	334	528

- 好消息是，大部分编译器提供了自动循环展开的功能

循环合并与循环分离

- 循环合并 (loop fusion): 合并相邻循环，有助于提升流水线和缓存利用率

```
for (i = 0; i < n; i++) {  
    a[i] = b[i] + 1;  
}  
for (i = 0; i < n; i++) {  
    c[i] = a[i] / 2;  
}  
for (i = 0; i < n; i++) {  
    d[i] = 1 / c[i];  
}
```

Before

```
for (i = 0; i < n; i++) {  
    a[i] = b[i] + 1;  
    c[i] = a[i] / 2;  
    d[i] = 1 / c[i];  
}
```

After

循环合并与循环分离

- 循环分离 (loop fission): 分裂相邻循环，有助于减轻寄存器和缓存压力

```
for (i = 0; i < n; i++) {  
    a[i] = b[i] + 1;  
    c[i] = a[i] / 2;  
    d[i] = 1 / c[i];  
}
```

Before

```
for (i = 0; i < n; i++) {  
    a[i] = b[i] + 1;  
}  
for (i = 0; i < n; i++) {  
    c[i] = a[i] / 2;  
}  
for (i = 0; i < n; i++) {  
    d[i] = 1 / c[i];  
}
```

After

循环合并与循环分离

合并 or 分离？

- 依赖于具体的硬件，依赖于具体的程序

- 如果循环体较小，且硬件缓存较大，应更多考虑循环合并
- 反之，如果循环体较大，且硬件寄存器资源稀缺，应更多考虑循环分离

- 例如，某处理器仅有容量较小的一级缓存

- 如果某循环体非常大，例如有上千行、数百个变量等等，可能需要考虑循环分离
- 如果连续有两个较小的循环体，且通过循环合并确实能够减少某些变量的访问次数或者提升数据局部性，此时可考虑循环合并

声明寄存器变量

◎ 例如，考虑如下程序

```
for (i=0; i<m; i++) {  
    for (j=0; j<n; j++) {  
        y[i] = y[i]+a[i][j]*x[j];  
    }  
}
```

$y[i]$ 具有很好的时间局部性，通过变量替换并声明register变量，提示编译器

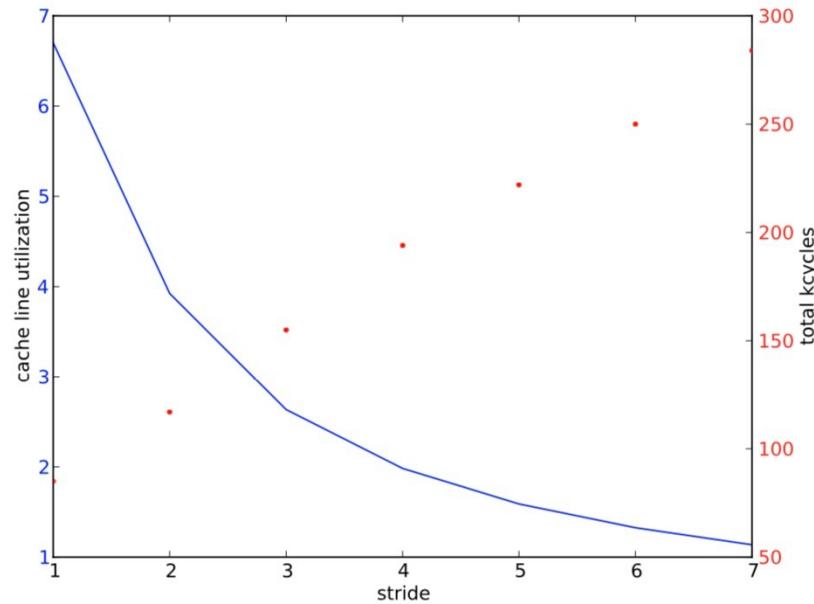
```
register double s;  
for (i=0; i<m; i++) {  
    s = 0.;  
    for (j=0; j<n; j++) {  
        s = s+a[i][j]*x[j];  
    }  
    y[i] = s;  
}
```

注意：register仅具有提示作用
如果想显式控制寄存器：写汇编

缓存线的充分利用

- 如果对连续的内存地址进行跨步 (stride)访问，会降低缓存线的利用率
- 例如，如下的数组遍历操作，随着步长增加，缓存线利用率下降

```
for (i=0, n=0; i<L1WORDS; i++, n+=stride)
    array[n] = 2.3*array[n]+1.2;
```

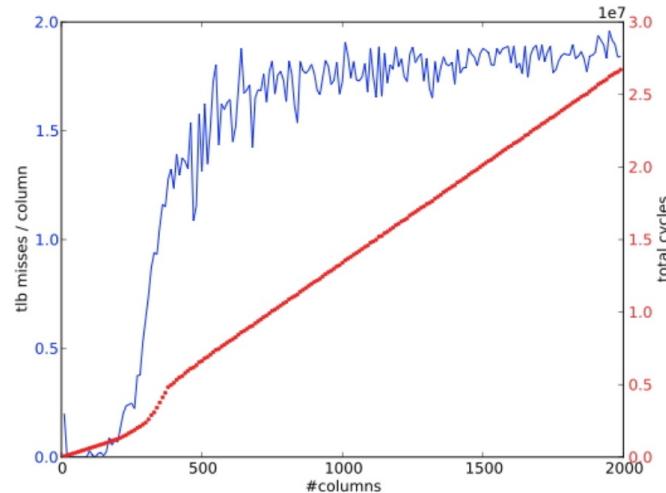


TLB的充分利用

- 访问大数组时，需充分考虑TLB命中率的影响，例如：方式一

```
#define INDEX(i,j,m,n) i+j*m  
array = (double*) malloc(m*n*sizeof(double));  
  
/* traversal #1 */  
for (j=0; j<n; j++)  
    for (i=0; i<m; i++)  
        array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
```

- Array is stored with columns contiguous
- Loop traverses the columns:
- No big jumps through memory
- (max: 2000 columns, 3000 cycles)

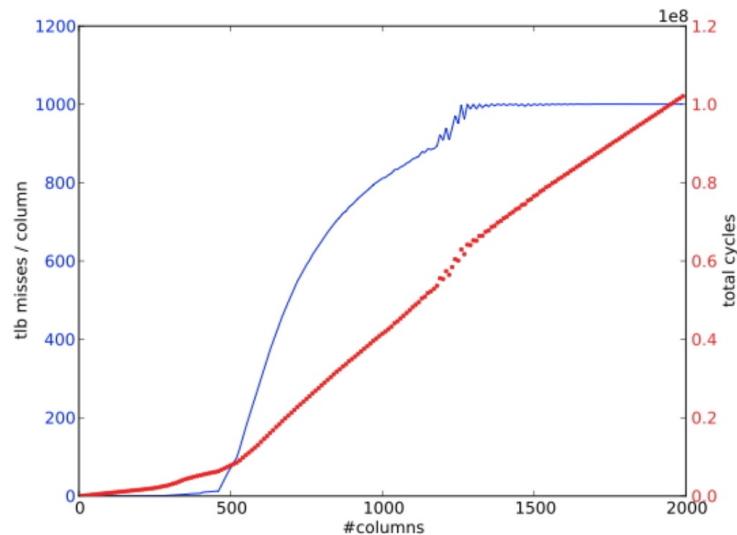


TLB的充分利用

- 访问大数组时，需充分考虑TLB命中率的影响，例如：方式二

```
#define INDEX(i,j,m,n) i+j*m  
array = (double*) malloc(m*n*sizeof(double));  
  
/* traversal #2 */  
for (i=0; i<m; i++)  
    for (j=0; j<n; j++)  
        array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
```

- Traversal by columns:
- Every next column is n words away
- If n more than page size: TLB misses
- (max: 2000 columns, 10Mcycles, 300 times slower)



回顾：缓存的冲突失效和映射策略

- 缓存冲突失效 (conflict miss): 两个不同的数据项被映射到了同一个缓存位置——与缓存映射策略相关
- 缓存的映射 (mapping): 缓存容量小于内存，需要确定内存中的数据地址与缓存中的地址对应关系，即映射策略，具体策略有
 - 直接 (direct) 映射
 - 全关联 (fully associative) 映射
 - k路关联 (k -way associative) 映射

回顾：缓存映射策略比较

- 例如：某缓存有12个缓存线，左图是直接映射(类似round-robin)，把数据地址按12取模后映射到不同的缓存线，同一缓存线中的数据相互之间有缓存冲突的风险

{0, 12, 24, ... }
{1, 13, 25, ... }
{2, 14, 26, ... }
{3, 15, 27, ... }
{4, 16, 28, ... }
{5, 17, 29, ... }
{6, 18, 30, ... }
{7, 19, 31, ... }
{8, 20, 32, ... }
{9, 21, 33, ... }
{10, 22, 34, ... }
{11, 23, 35, ... }

{0, 12, 24, ... } {4, 16, 28, ... }
{8, 20, 32, ... }
{1, 13, 25, ... } {5, 17, 29, ... }
{9, 21, 33, ... }
{2, 14, 26, ... } {6, 18, 30, ... }
{10, 22, 34, ... }
{3, 15, 27, ... } {7, 19, 31, ... }
{11, 23, 35, ... }

右图是3路关联映射，缓存线以3个一组与内存数据映射，可以避免同组缓存冲突小于3个的情况。

全关联映射则可以将任意内存数据映射给任意缓存线，能够最大程度降低缓存冲突失效，但造价高昂

避免缓存冲突失效

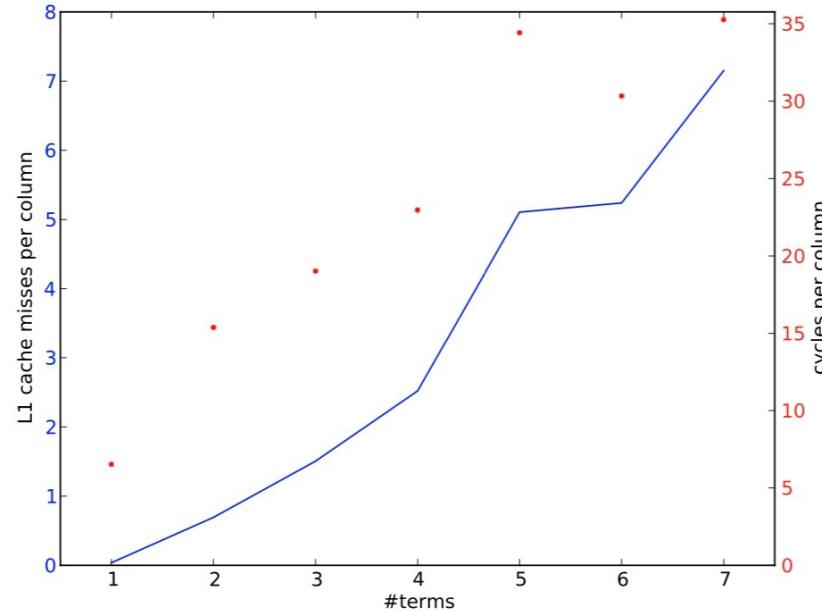
- 缓存大小通常是2的幂量级，而很多算法中数据规模也经常是2的幂，例如FFT等
- 此时，将可能出现严重的缓存冲突失效
- 例如：某系统64KB二路关联L1缓存，缓存线64B，进行如下运算

$$\forall_j: y_j = y_j + \sum_{i=1}^m x_{i,j}$$

➤ 如果向量大小为4096个双精度浮点数
 $= 4K * 8B = 32KB$

当 $m > 1$ 时会导致缓存冲突失效

每次循环缓存失效应量大致为 $m - 1$

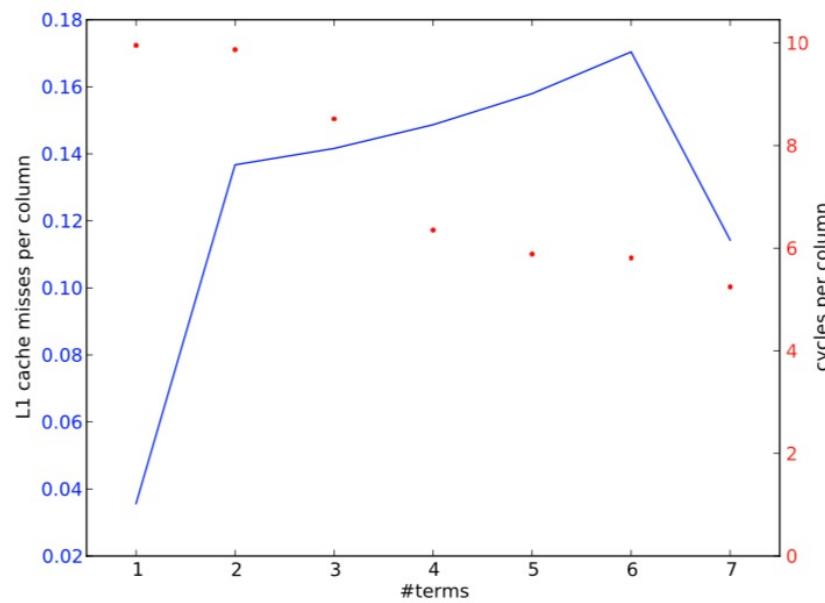


避免缓存冲突失效

- 缓存大小通常是2的幂量级，而很多算法中数据规模也经常是2的幂，例如FFT等
- 此时，将可能出现严重的缓存冲突失效
- 例如：某系统64KB二路关联L1缓存，缓存线64B，进行如下运算

$$\forall_j: y_j = y_j + \sum_{i=1}^m x_{i,j}$$

- 如果适当增加向量大小，例如调整为4096+8个双精度浮点数
- 同一个j, y和x的元素不对应一组缓存线
- $m > 1$ 时，每次循环缓存失效率大致为 $1/8$



几点讨论

● 刚才的例子里

- 如果本来向量大小就是4096个双精度浮点数怎么办?

可以通过补零 (zero padding) 的方式调整向量大小

- 为什么实际缓存失效率比理论预测略低?

与处理器的复杂设计相关, 例如预取的影响、读写开销不均的影响等等

● 实际应用中, 经常遇到嵌套多层循环, 如果可以交换循环次序, 应采用什么原则?

- 例如, 对二维数据 (i, j) 索引进行遍历:

- 如果是C语言, 建议 j -循环在内
- 如果是Fortran语言, 建议 i -循环在内

- 如果讨论不同的并行计算模型

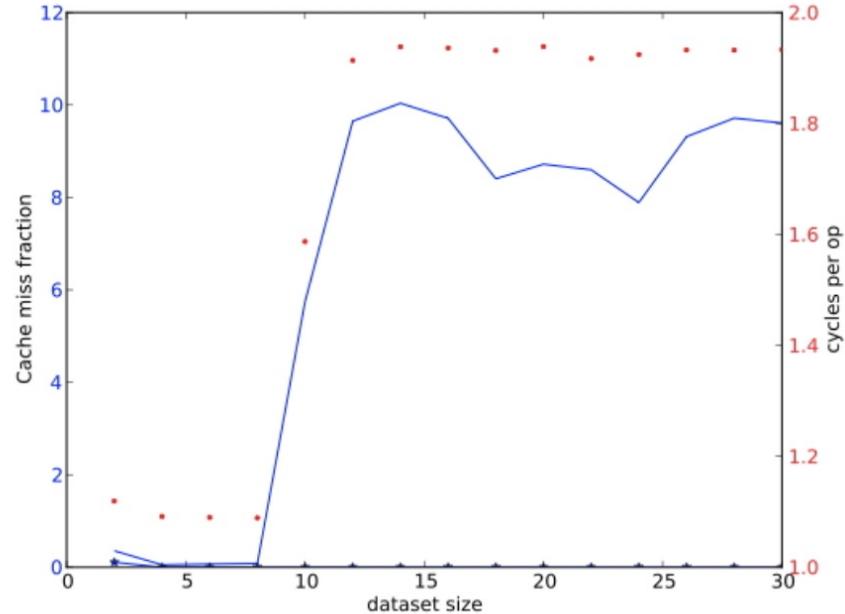
- 如果是OpenMP, 一般希望外层循环较长, 较短的内层循环有时有助于向量化
- 如果是CUDA, 一般希望内层循环较长, 并且不包含分支和判断

缓存分块

- ◎ 例如，如下一个数组遍历程序，数组长度size对性能影响

- 如果数据规模小于L1缓存大小，L1缓存命中率高，大幅提升性能
- 如果数据规模增大，超出L1缓存大小，性能则进一步受L2缓存性能影响

```
for (i=0; i<NRUNS; i++)
    for (j=0; j<size; j++)
        array[j] = 2.3*array[j]+1.2;
```



缓存分块

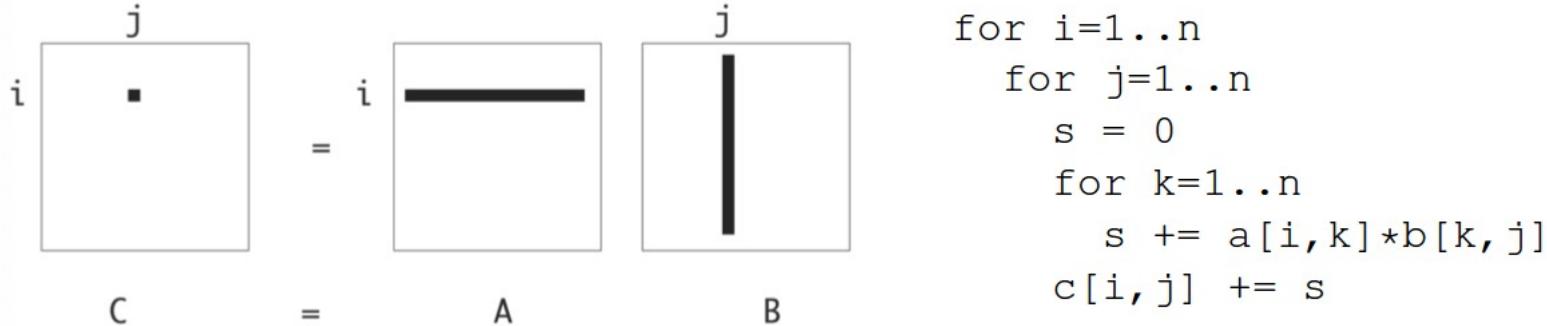
- 缓存分块 (cache blocking): 把数据切分成能够放入缓存的块，提升缓存利用率
- 例如，对刚才的例子，可以增加一重循环并引入缓存分块提升性能

```
for (b=0; b<size/l1size; b++) {  
    blockstart = 0;  
    for (i=0; i<NRUNS; i++) {  
        for (j=0; j<l1size; j++)  
            array [blockstart+j] = 2.3*array [blockstart+j]+1.2;  
    }  
    blockstart += l1size;  
}
```

- 缓存分块的一个重要手段是循环分块 (loop tiling)，即把循环分为两层，内层遍历数据块内，外层遍历所有数据块，从而提升数据局部性和缓存命中率
- 注意：大部分时候，编译器不会自动做循环分块

一个例子：矩阵乘

- ◎ 矩阵乘，若采用 $i-j-k$ 模式， $c[i, j]$ 可以放入寄存器，但是 $a[i, k]$ 不行



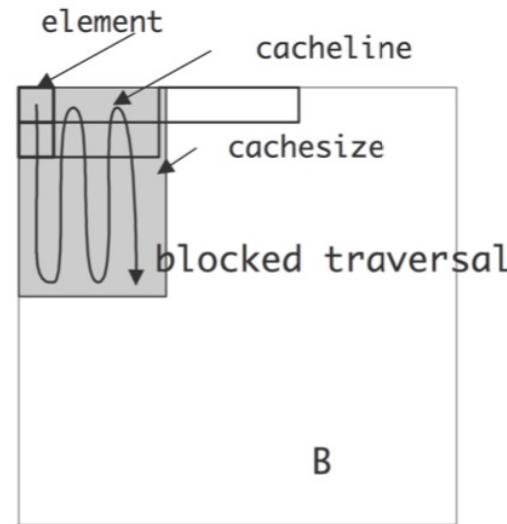
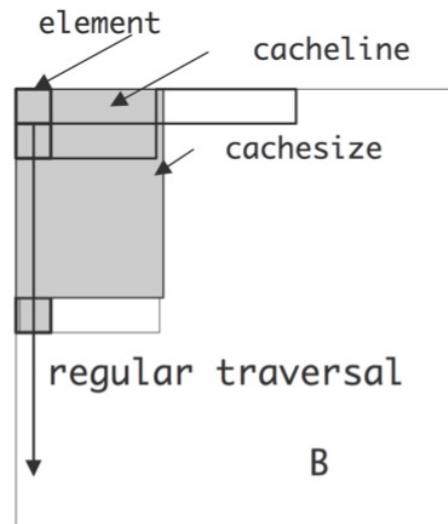
- ◎ 此时，如果对 k 循环采用循环分块，可以有效提升 $a[i, k]$ 的时间局部性

```
for kk=1..n/b
    for i=1..n
        for j=1..n
            s = 0
            for k=(kk-1)*bs+1..kk*bs
                s += a[i,k]*b[k,j]
            c[i,j] += s
```

另一个例子：矩阵转置

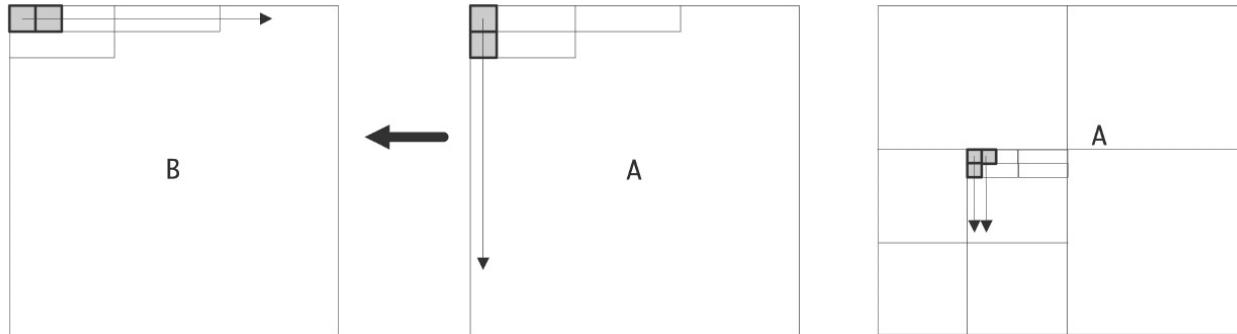
◎ 举例：矩阵转置，通过循环分块充分利用L2缓存

```
for (int ii=0; ii<N; ii+=blocksize)
    for (int jj=0; jj<N; jj+=blocksize)
        for (int i=ii*blocksize; i<MIN(N, (ii+1)*blocksize); i++)
            for (int j=jj*blocksize; j<MIN(N, (jj+1)*blocksize); j++)
                A[i][j] += B[j][i];
```



缓存感知与缓存无关编程

- 缓存感知 (cache aware) 编程：根据缓存大小，有意识地进行数据分块，提升数据局部性，从而提高缓存命中率
- 然而，寄存器和缓存具有多级特性，每一层级又有不同参数，如何自动利用？
- 缓存无关 (cache oblivious) 编程：通过递归、分治等策略对数据分块，自动适配各种不同缓存配置，提升每一层缓存的命中率
- 例如，矩阵转置，对矩阵的行列递归二分遍历，只要确保最小分块可放入寄存器



注：缓存无关编程
虽然可以提升缓存
命中率，但往往不
能实现最优性能

第三个例子：矩阵向量乘

$$\forall_{i,j} : y_i \leftarrow a_{ij} \cdot x_j$$

- 参考实现： $3mn$ 次读， mn 次写， $2mn$ 次计算

```
for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
        y[i] = y[i]+a[i][j]*x[j];
    }
}
```

第三个例子：矩阵向量乘

$$\forall_{i,j} : y_i \leftarrow a_{ij} \cdot x_j$$

- 优化1：重用 $y[i]$ ， $2mn$ 次读， m 次写， $2mn$ 次计算

```
for (i=0; i<m; i++) {
    s = 0.;
    for (j=0; j<n; j++) {
        s = s+a[i][j]*x[j];
    }
    y[i] = s;
}
```

第三个例子：矩阵向量乘

$$\forall_{i,j} : y_i \leftarrow a_{ij} \cdot x_j$$

- 优化2：重用 $x[j]$ ， $2mn+n$ 次读， mn 次写， $2mn$ 次计算

```
for (j=0; j<n; j++) {
    t = x[j];
    for (i=0; i<m; i++) {
        y[i] = y[i]+a[i][j]*t;
    }
}
```

第三个例子：矩阵向量乘

$$\forall_{i,j} : y_i \leftarrow a_{ij} \cdot x_j$$

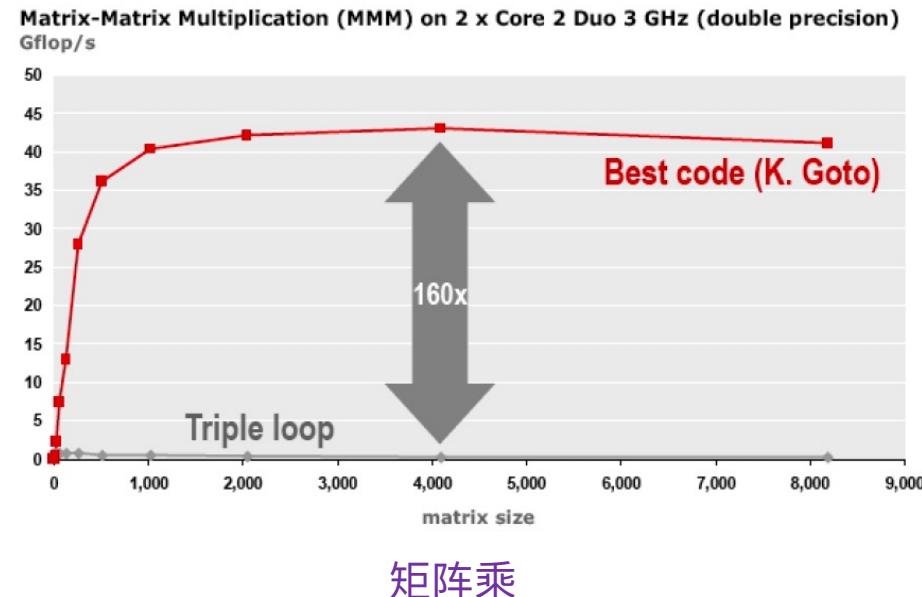
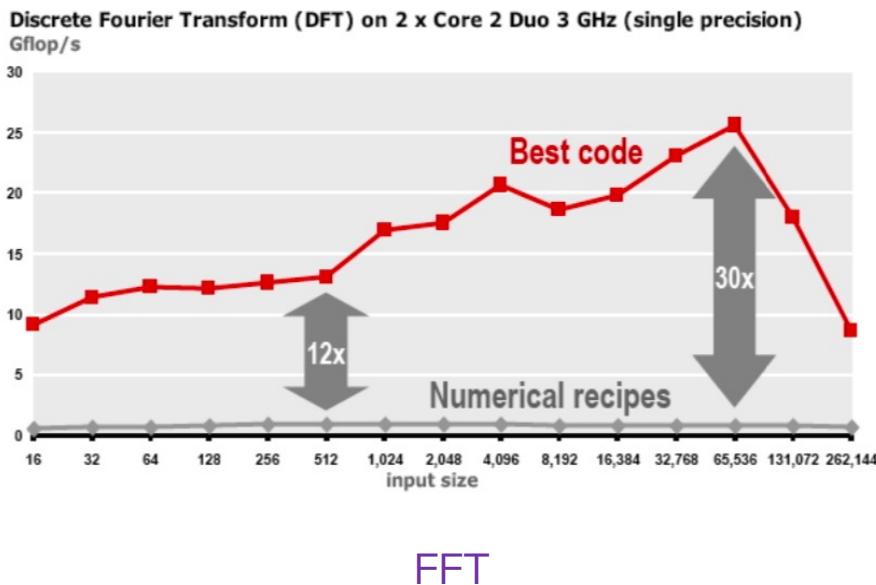
- ◎ 优化3：循环分块(展开), $3mn/2$ 次读, m 次写, $2mn$ 次计算

```
for (i=0; i<m; i+=2) {
    s1 = 0.; s2 = 0.;
    for (j=0; j<n; j++) {
        s1 = s1+a[i][j]*x[j];
        s2 = s2+a[i+1][j]*x[j]
    }
    y[i] = s1; y[i+1] = s2;
}
```

性能的极致优化

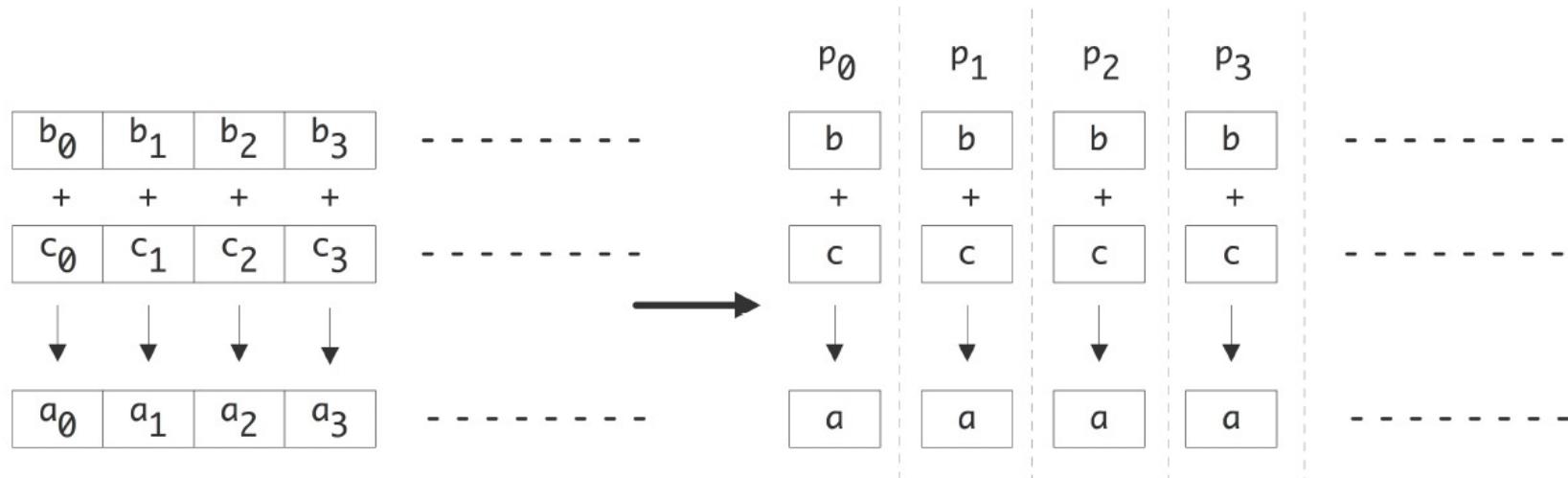
◎ 经过极致优化的代码相对于参考实现可以达数十上百倍的加速

- 这并不是依赖编译器就可实现的优化，需要编程算法中充分考虑设计
- 由于系统的复杂性，通过自动调优自适应选取最优参数是一个研究领域



并行计算举例

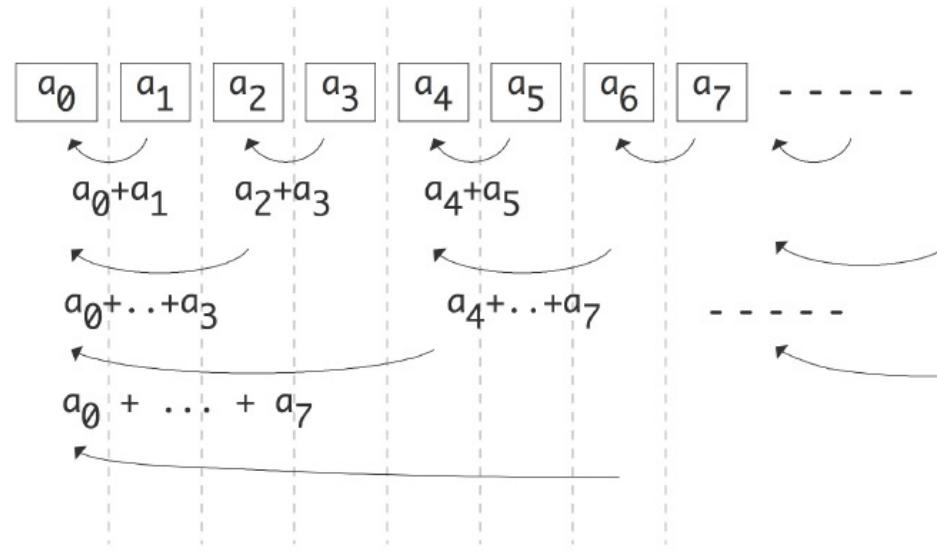
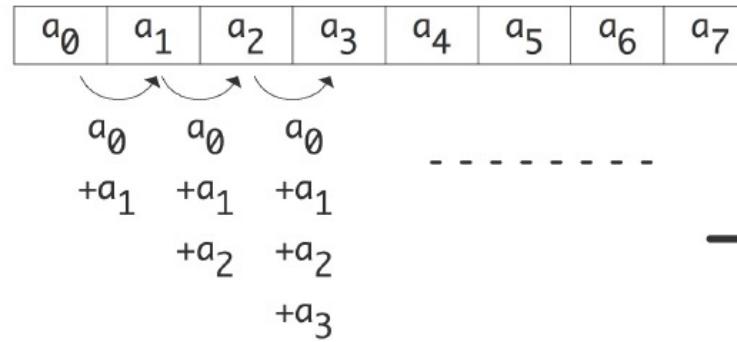
◎ 例子1: 向量加法



数据并行 (data parallelism): 对一批数据进行相同操作，操作的数据往往存储于向量、矩阵、图或者某种计算网格上

并行计算举例

◎ 例子2: 向量规约 (reduction)



加速比与并行效率

- 这里，我们用“处理器”表示并行计算中所使用的最基本的处理单元
- 定义如下两个基本概念
 - 加速比 (speedup): 同一个任务，单处理器执行时间与 p 个处理器执行时间之比, $S_p = T_1/T_p$
 - 并行效率 (parallel efficiency): 加速比与处理器个数的比值, $E_p = S_p/p$
- 额外开销 (overhead) 指的是从串行到并行引起的额外工作: $T_o = pT_p - T_1$
- 一个并行算法称为代价最优 (cost-optimal), 指的是: $T_o = O(T_1)$
- embarrassingly/conveniently parallel computing: 专门指 $T_o = 0$ 的问题, 此类问题可实现线性加速 (linear speedup)
- 一般而言, 加速比不超过 p , 并行效率不超过100%, 否则称为超线性加速(superlinear speedup)

可扩展性

- 阿姆达尔与古斯塔法森定律中的假定可能与实际应用之间存在出入，但是两种扩展问题规模的方式非常实用
- 可扩展性 (scalability)

➤ 强可扩展性 (strong scalability): 阿姆达尔意义下的加速比

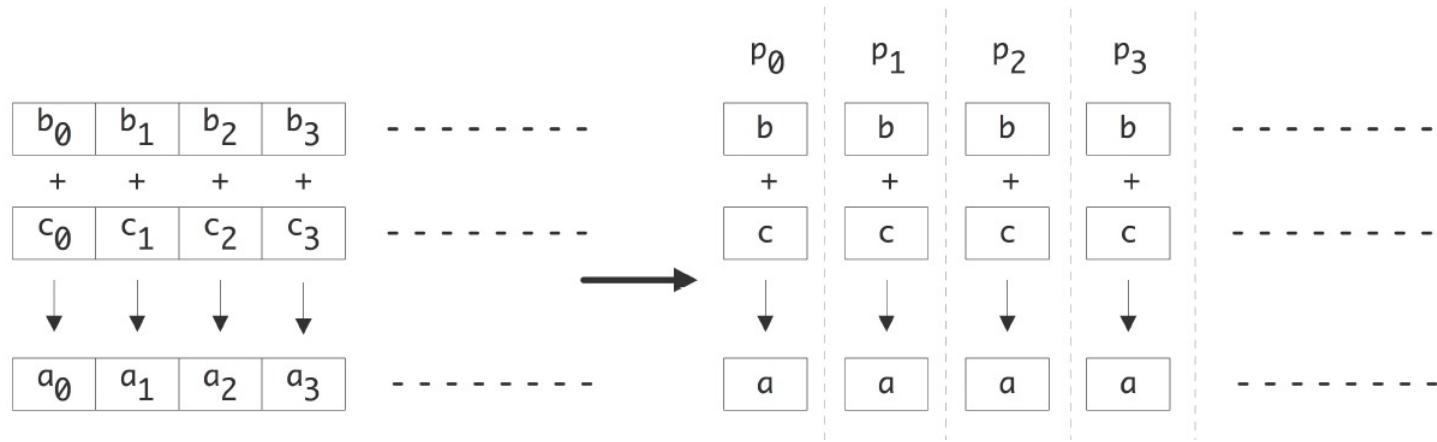
$$\left. \begin{array}{l} N \equiv \text{constant} \\ P \rightarrow \infty \end{array} \right\} \Rightarrow E_P \approx \text{constant}$$

➤ 弱可扩展性 (weak scalability): 古斯塔法森意义下的加速比

$$\left. \begin{array}{l} N \rightarrow \infty \\ P \rightarrow \infty \\ M = N/P \equiv \text{constant} \end{array} \right\} \Rightarrow E_P \approx \text{constant}$$

几个例子

◎ 例子1: 向量加法



◎ 使用 n 个处理器，加速 n 倍，平行效率 1

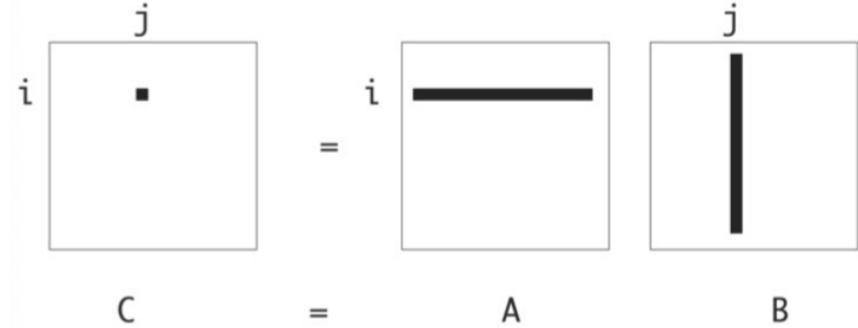
几个例子

◎ 例子2: 矩阵乘

$$\forall_{i,j}: c_{ij} = \sum_k a_{ik} b_{kj}$$

共需 $2n^3$ 次计算

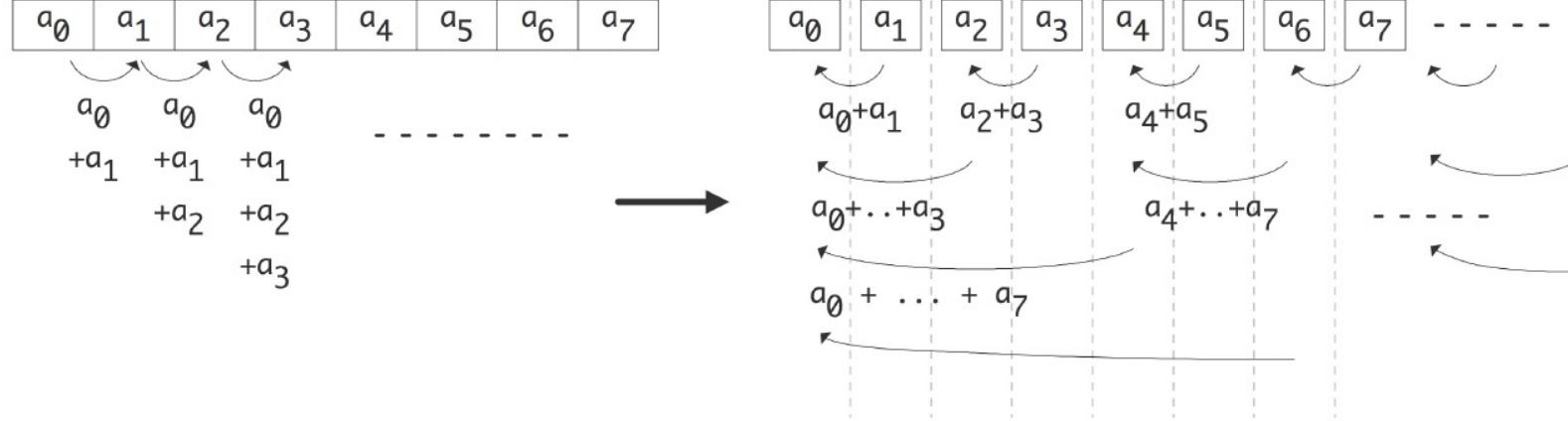
```
for i=1..n
  for j=1..n
    for k=1..n
      c[i, j] += a[i, k] * b[k, j]
```



- 使用 n^2 个处理器，每个处理器计算 C 的一个元素，各需 $2n$ 次计算，并行效率 1
- 注意，上述仅是从计算层面进行的分析，从访存来看，数据读取的重复率为多少？

几个例子

◎ 例子3: 向量规约



- ◎ 使用 $n/2$ 个处理器，共 n 次计算，需 $\log_2 n$ 步完成，并行效率约为 $\frac{1}{\log_2 n}$

算法的并行度vs程序的并行度

- 有些问题从串行代码来看似乎不能并行，但从算法角度经过变换实际上可并行
- 比如：

```
for i in [1:N]:  
    x[0,i] = some_function_of(i)  
    x[i,0] = some_function_of(i)
```

```
for i in [1:N]:  
    for j in [1:N]:  
        x[i,j] = x[i-1,j]+x[i,j-1]
```

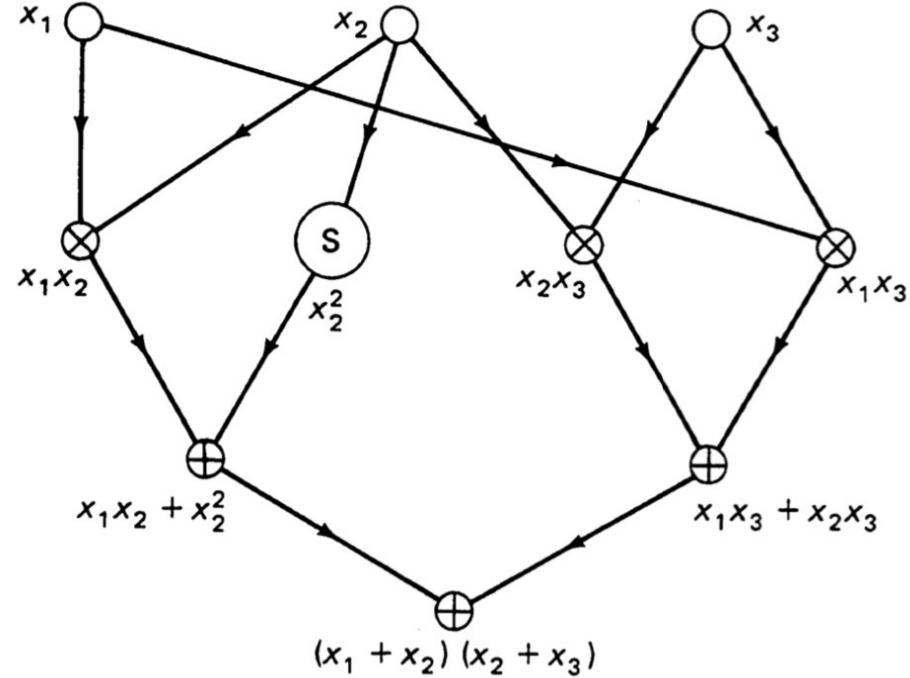
Answer the following questions about the double i, j loop:

1. Are the iterations of the inner loop independent, that is, could they be executed simultaneously?
2. Are the iterations of the outer loop independent?
3. If $x[1, 1]$ is known, show that $x[2, 1]$ and $x[1, 2]$ can be computed independently.
4. Does this give you an idea for a parallelization strategy?

有向无环图

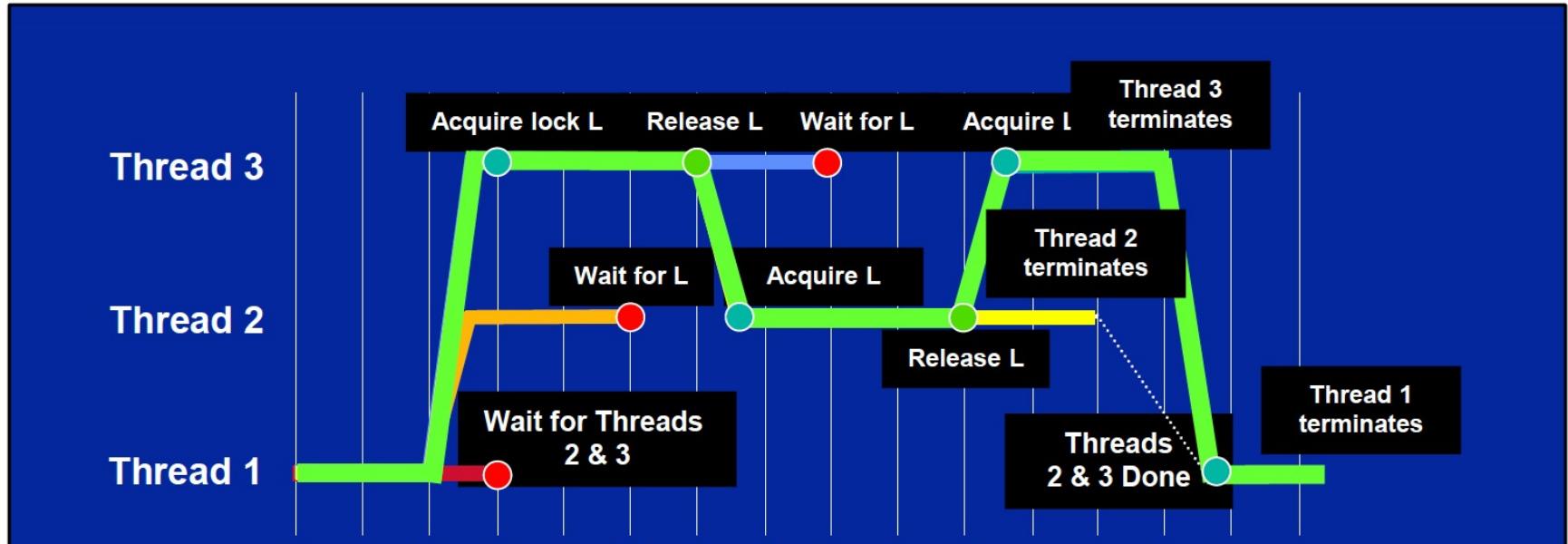
- 有向无环图 (Directed Acyclic Graph, DAG): 不含环路的有向图

- 节点: 表示任务
(可用节点个数表示任务量)
- 边: 表示任务之间的依赖关系
(从被依赖任务指向依赖任务)
- 例如: $(x_1+x_2)(x_2+x_3)$



关键路径

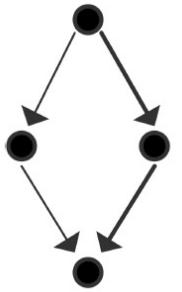
- 关键路径 (critical path): 并行算法中计算任务之间的最长依赖路径，可能不唯一



平均并行度

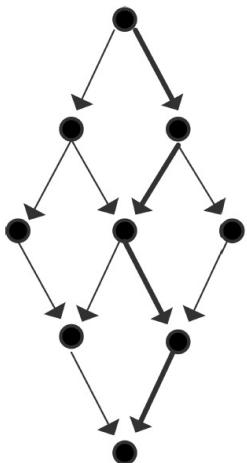
- 平均并行度定义为 T_1/T_∞ , 其中 $T_\infty = \lim_{p \rightarrow \infty} T_p$ 为关键路径的执行时间
- 进一步, 记 p_∞ 为使得 $T_p = T_\infty$ 的最小处理器数
- 思考1: 平均并行度最小可是多小? 它的DAG是什么样子的?
- 思考2: 平均并行度最大可是多大? 它的DAG又是什么样子呢?

利用DAG分析算法的几个例子



◎ 例子1:

$$\begin{aligned} T_1 &= 4, & T_\infty &= 3 \Rightarrow T_1/T_\infty = 4/3 \\ T_2 &= 3, & S_2 &= 4/3, & E_2 &= 2/3 \\ P_\infty &= 2 \end{aligned}$$

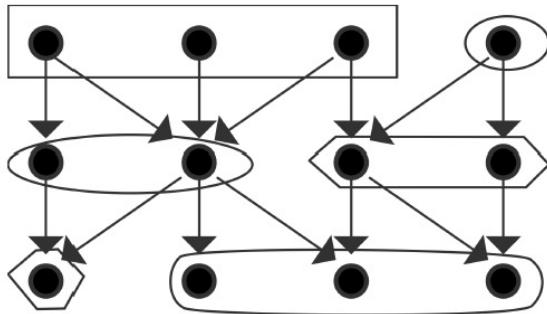


◎ 例子2:

$$\begin{aligned} T_1 &= 9, & T_\infty &= 5 \Rightarrow T_1/T_\infty = 9/5 \\ T_2 &= 6, & S_2 &= 3/2, & E_2 &= 3/4 \\ T_3 &= 5, & S_3 &= 9/5, & E_3 &= 3/5 \\ P_\infty &= 3 \end{aligned}$$

利用DAG分析算法的几个例子

● 例子3：



$$\begin{aligned}T_1 &= 12, \quad T_\infty = 4 \quad \Rightarrow T_1/T_\infty = 3 \\T_2 &= 6, \quad S_2 = 2, \quad E_2 = 1 \\T_3 &= 4, \quad S_3 = 3, \quad E_3 = 1 \\T_4 &= 3, \quad S_4 = 4, \quad E_4 = 1 \\P_\infty &= 4\end{aligned}$$

思考：书中这个例子的分析有错误，哪里错了？

布伦特定理

- 布伦特定理 (Brent's Theorem): 记 t 为关键路径长度, m 为总任务数, 则采用 p 个处理器的计算时间上限为: $t + \frac{m-t}{p}$

Proof. Divide the computation in steps, such that tasks in step $i + 1$ are independent of each other, and only dependent on step i . Let s_i be the number of tasks in step i , then the time for that step is $\lceil \frac{s_i}{p} \rceil$. Summing over i gives

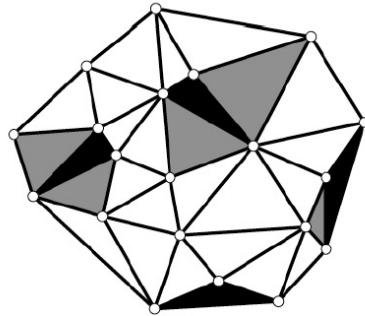
$$T_p = \sum_i^t \lceil \frac{s_i}{p} \rceil \leq \sum_i^t \frac{s_i + p - 1}{p} = t + \sum_i^t \frac{s_i - 1}{p} = t + \frac{m - t}{p}.$$

几种不同层次的并行

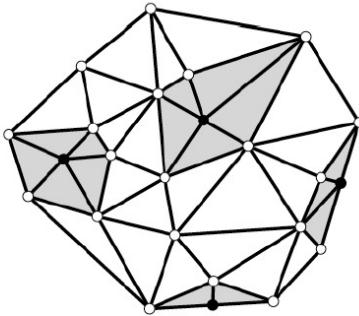
- 指令级并行 (instruction-level parallelism, ILP): 并行执行相互独立的指令，一般通过CPU硬件（如超标量技术）或者编译器实现
- 数据级并行 (data-level parallelism, DLP): 对一批数据进行相同操作，操作的数据往往存储于向量、矩阵、图或者某种计算网格上
- 任务级并行 (task-level parallelism, TLP): 按照计算任务/指令之间的依赖关系确定并行策略，有向无环图 (DAG)常常作为分析任务之间依赖关系和确定并行方案的重要工具

非规则数据并行

◎ 例子: Delaunay网格细化



(a) Unrefined Mesh



(b) Refined Mesh

```
1: Mesh mesh = /* read in initial mesh */
2: WorkList wl;
3: wl.add(mesh.badTriangles());
4: while (wl.size() != 0) {
5:     Triangle t = wl.get(); //get bad triangle
6:     if (t no longer in mesh) continue;
7:     Cavity c = new Cavity(t);
8:     c.expand();
9:     c.retriangulate();
10:    mesh.update(c);
11:    wl.add(c.badTriangles());
12: }
```

非规则数据并行 (amorphous data-parallelism): 任务之间的依赖关系复杂，并且受运行结果动态影响，一般需要基于工作列表遍历数据，数据之间的关系可以用图表示，计算的过程是对图上每个活跃节点的邻居节点进行计算，并修改该活跃节点的状态

并行计算的粒度

- 粒度 (granularity): 指的是每个处理器的工作集的大小
- 例1: 指令级并行的粒度为单个或者几个指令, 属于细粒度并行 (fine-grain parallelism)
- 例2: SIMD作为数据级并行的一个特例, 任务粒度也为单个或者几个指令, 因此也属于细粒度并行
- 例3: 对stencil计算采用区域分解的方法并行, 任务粒度为子区域, 属于粗粒度并行 (coarse-grain parallelism)
- 例4: 对于难以进行规则剖分的问题, 往往可以引入任务级并行, 将问题分解为若干中粒度 (medium-grain)的子任务, 通过任务调度掩盖任务之间的不均衡