Memory Hierarchy & Cache Memories

1 存储技术

1.1 随机存取存储器（RAM）

该部件用于同 CPU 进行交互，亦称作主存。它属于易失性存储器，即使用中需要保持通电以使数据不丢失。

RAM 的制造技术主要分为两种：DRAM（动态）和 SRAM（静态）。两者对比如下：

	DRAM	SRAM
位信息表示方式	电容电荷	晶体管状态（6 根晶体管表示 1 位）
计算效率	低	高
存储密度（单位面积分布信息量）	高	低
是否需要定期刷新	是（由于电容漏电）	否
主要应用	缓存	主存

DRAM 有如下提高性能的手段：

- SDRAM：同步 DRAM。普通的异步 DRAM 随时响应控制输入；而 SDRAM 在响应之前首先等待时钟信号，以与系统总线的时钟主频同步，于是可以通过流水线模式操作指令，提升性能。
- DDR SDRAM：双倍数据率 SDRAM。它在时钟的上升沿和下降沿均进行数据传输，使传输速度达到时钟主频的 2 倍。
- DDR2、DDR3：通过将预读取位数分别上升至 4 位和 8 位，进一步提升效率。

1.2 非易失性存储器（NVM）

该类存储器一旦写入数据后便无法改变或删除，故可在断电后仍然保存信息。若需要对这类存储器实现写操作，则需要将整片区域的数据擦除之后再行覆写，因而读写效率不对称。

ROM（只读存储器，预编码，用于存储引导计算机的指令）即属于此类存储器。

闪存技术即 EEPROM，在 ROM 的基础上增加了可电子抹除、可复写的功能。大量应用于容量较小的移动存储设备如 U 盘、MP3 播放器等之中。

1.3 外存

1.3.1 硬盘

传统外存主要使用硬盘（HDD）存储数据。

硬盘的存储介质是坚硬的旋转盘片，读写装置是磁头。

硬盘由数个盘片（platter）组成，每个盘片包括两面（surface）。每个表面由轨道（track）组成，它们是同心圆。每个轨道进而被区分为多个扇区（sector）。外圈轨道的扇区数目多于内部扇区。

读写操作时，通过磁盘的旋转运动，及读写头沿磁盘半径方向的运动来定位。这两种机械运动是磁盘读写操作的主要耗时部分。

磁盘存取时间为 ms 量级，而 RAM 是 ns 量级，两者之间存在百万量级的差距。

1.3.2 固态硬盘

固态硬盘（SSD）使用闪存技术代替磁盘作为存储技术，即通过电磁运动替代了传统磁盘的机械运动。这极大提升了存取效率，同时也降低了工作温度和工作噪音、提升了抗损坏能力。

但同时，固态硬盘单位容量的造价较为昂贵，这限制了固态硬盘应用的容量。而且固态硬盘拥有写入次数的限制，有一定的使用寿命；而且数据删除后即难以恢复，但传统硬盘可以做到这一点。

固态混合硬盘（SSHD）同时运用了上述两种技术。

1.4 未来发展趋势

一种新型的 NVM 技术，通过熔化晶体进行写操作（以其冷却后的状态表示 0 或 1）、通过测试电阻进行读操作。该技术可使得读操作效率大幅提升，以至于主存和外存可以进行合并，以取代 DRAM。

但该技术同时进一步拉大了读写操作效率的差距。故在其投入应用后，相应的程序优化策略将会以尽量减少写操作为主导。

2 存储器层次结构

2.1 层次结构概述及应用价值

存储器层次结构将各存储器分为不同层级，每次取用下层内容时同时将其复制到上层，以便此后在上层直接取用。在存储器层次结构中，高层次存储器使用存取速度快而体积小的介质、进行小量数据操作；低层次则使用速度慢、容量大的介质，单次存取量大，以此将存取开销平摊至较多的信息当中。

存储器层次结构，可以便于对有良好局部性的代码减少执行开销。局部性分为两种：

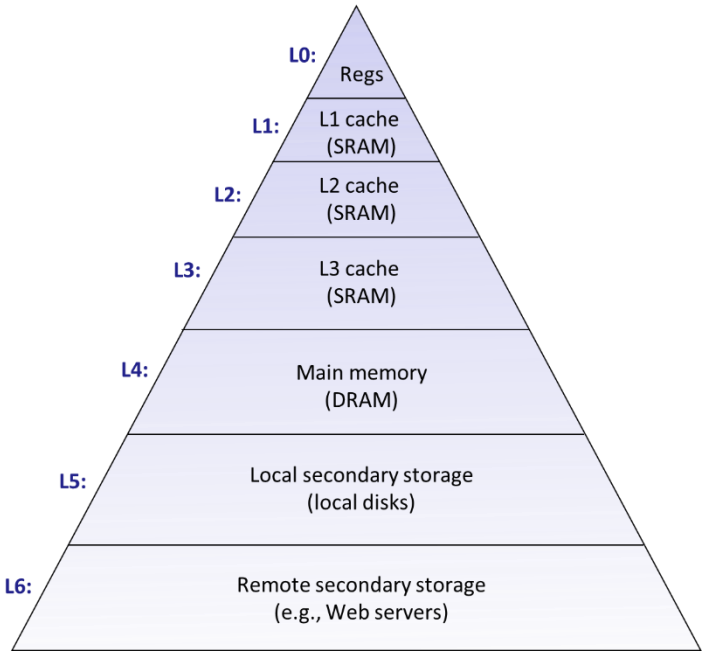
- 时间（temporal）局部性：往往在短时间内使用同一数据；
- 空间（spatial）局部性：地址重用，往往使用地址相近/连续存放的数据。

2.2 缓存的基本概念

缓存是集成在 CPU 上的存储设备，分为 L1，L2 和 L3 共三个级别。它们用于存储近期曾使用到的主存中的信息。在存储器层次结构中，它们位于寄存器（L0）之下、主存之上，如下页图所示。

当下 CPU 通常采用三级缓存设计，缓存使用 SRAM 技术制造。CPU 里绝大部分的体积用于 cache，只有很小一部分（小于 5%）是运算单元。CPU 的核心集成了 L1 和 L2 缓存，

L3 缓存在同一 socket 内的核心之间共享。各 socket 之间共享主存。并行计算中，为便于线程之间通信，在一个 socket 内部进行运算。这可以通过 CPU affinity（CPU 绑定）实现。



图：存储器层次结构图

©Randy Bryant, CMU

在层次结构中，位于较下层的存储器被区分为不同的“块”，每个块有各自不同的编号。其上层存储器缓存其中的部分块。这两层之间数据即以块为单位传输。

CPU 在使用数据时，首先在上层请求所需的块。若上层缓存存有该块，则为 hit（命中）；反之为 miss（不命中）。若读取数据时命中，则不需要继续访问下层存储器，否则需要从下层存储器中取出所需内容，并将该块复制到上层缓存中。这可能会挤占上层缓存中已存有内容的区域。

块的存取策略都是硬件级别可见的。

缓存不命中分为以下种类：

- Cold miss：上层缓存尚未被使用，其中放置的内容同本程序没有关系。义即，缓存是“空的”。
- Conflict miss：大部分缓存都规定，下层的一定编号（地址段落）的块，需要放在该缓存确定的区域。当缓存仍有空余，但目标区域已满，则发生此类不命中。
- Capacity miss：缓存已满，则发生此类不命中。

2.3 缓存的组织架构

一般而言，缓存被分为 $S=2^s$ 个集合（set），每个 set 被分为 $E=2^e$ 个 line，每个 line 中包括：1 个 valid bit，1 个 t 位的 tag，以及 1 个 $B=2^b$ 字节的 block。block 即是缓存主存内容的区域。

当访问主存中的地址时，缓存以如下方式解析该地址：t 位 tag + s 位 set 编号 + b 位 block offset。缓存在相应编号的 set 中并行（硬件层面的并行）寻找 valid bit 设为 1、且 tag 与地址中的 tag 相对应的 line。若找到，即为命中。若未找到，即为不命中。

使用中间位作为 set 索引的原因在于：这样可以使得地址邻近的块分属于不同的 set，于是对空间局部性强的代码便减少了 conflict miss 次数。

现代存储器的缓存价格通常有如下两种方式：

- Direct-mapped Cache：E = 1。它的缺点是：在上层，只能映射到指定 set 的相应位置，即使其他位置有空位也不能放置，造成 conflict miss。但这样每次读 cache 的速度比较快。
- E-way Set Associative Cache：E != 1。速度慢（O(E)时间复杂度），但是减少了 conflict miss。

在读操作时，发生不命中，则将相应地址所属块（tag 及 set 编号均相同即为一块）的内容，放置于该 set 的某一 line 中。若该 set 尚有未使用的 line，则放置于该 line，且将该 line 的 valid bit 设为 1。否则，需要将某个已放置内容的 line 内容替换为本块的内容。

选取被替换的 line（victim），有不同的策略。较常用的策略为 LRU，即替换最近一次使用时间最早的 line。

对于写操作的命中与不命中，有两种处理方式：

- write-through + no-write-allocate：命中时，逐层写下层相应位置的内容；不命中时，直接写主存中对应地址、对缓存不做修改；
- write-back + write-allocate：命中时，只写本层，直到该层被替换时再写下层；不命中时，将主存相应位置的 tag 放入，再只写本层。这种策略需要每个缓存 line 另加一个 dirty bit，表示该块的内容与主存内容不同。

现代处理器多采用第二套策略。第一套策略略方便于 miss 较多的情况；但总体而言第二套策略的性能更佳（利于局部性强的代码）。

2.4 衡量缓存性能

衡量 CPU 缓存性能有如下参数：

- Miss 率：miss 次数与存取次数之比。使用 miss 率而不用 hit 率的原因是：miss 率 3%（hit 97%）和 miss 率 1%（hit 99%）的情况，耗时钟周期数相差几乎一倍。使用 miss 率可以直观地反映出这种倍率。
- Hit 时间：从某层缓存取用数据至处理器所需的时钟周期数。
- Miss 处罚：发生一次 miss 增加的时钟周期数。由于随时代发展，缓存块的平均大小会上升，因而每次 miss 后刷新缓存的消耗更大，所以这一指标反而会增加，因此现代程序优化多是提升访问效率（提高程序局部性）而不是计算效率。

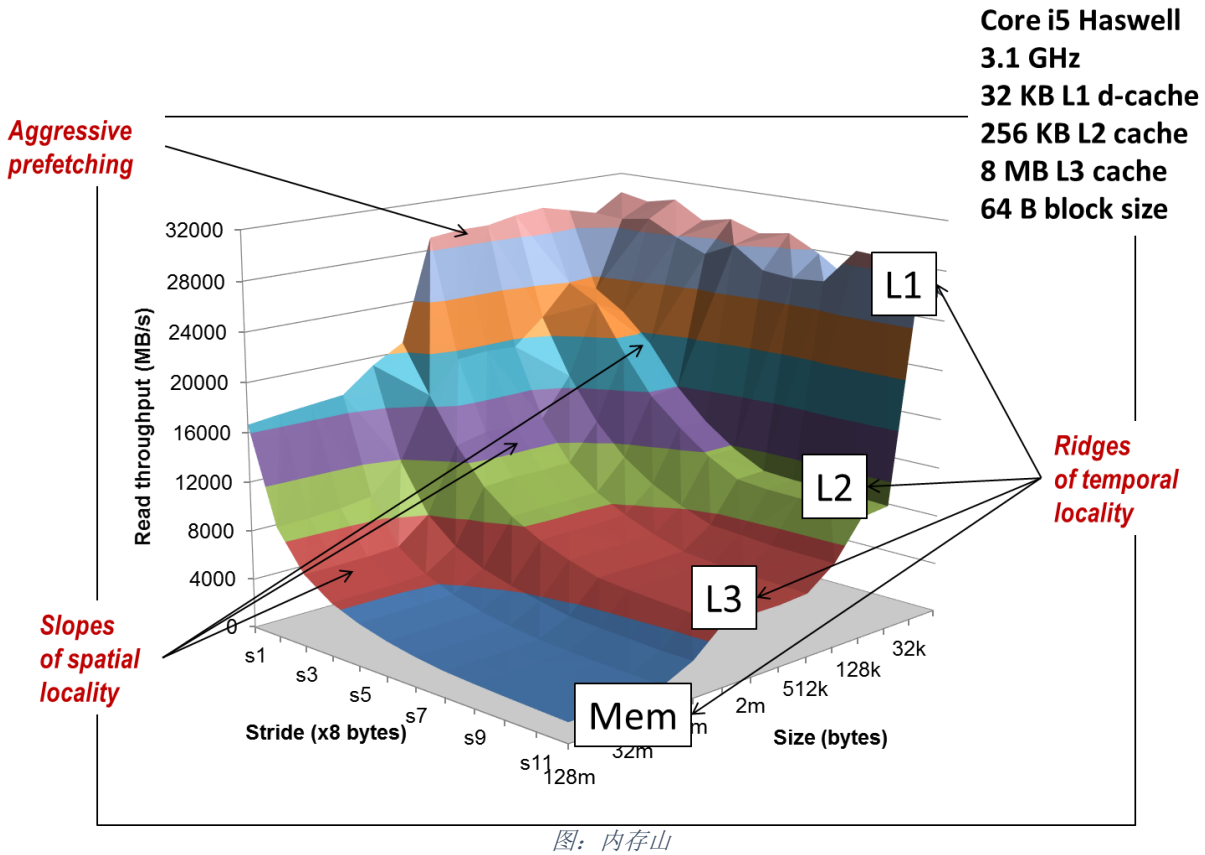
3 编写缓存友好的代码

3.1 基本原则

为编写缓存友好的代码，程序需要尽量减少最常访问的代码（即最内层循环）的 miss 率，因而需要尽量提升这些语句的局部性：往往访问最近访问过的地址（时间局部性）；尽量采取步长为 1 的遍历模式（空间局部性）。

3.2 内存山

衡量程序平均读写速度的指标为吞吐量（read throughput），由每秒从内存中读取的内容大小（MB/s）计算。



图：内存山

©Randy Bryant, CMU

对于按确定步长遍历一段内存区域的程序，吞吐量与步长、区域大小的关系图，呈现出上图的“山”状结构，是为内存山。

在区域大小变大至某些阈值时，吞吐量图示有明显的下降陡坡，这些阈值恰好和测试计算机的各级别缓存大小相应。这是因为：如果工作集合小到完全可以装到一级 cache 里，那么访问速度几乎就是该级 cache 的访问速度，所以在 cache 大小的边界处会出现一个急剧的下坡。

但在步长为 1 时程序受此影响很小。这是因为现代处理器会进行预读操作：根据读的规律预先覆盖冲突/满溢的缓存，减少单次 miss 开销。

吞吐量随步长增大也在缓慢下降。这是由于临近时间往往访问了不相邻的数据，造成了 miss 增多。

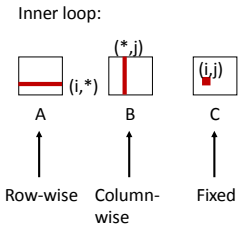
3.3 示例：矩阵乘法

本示例需要完成一个具有以下功能的函数：对给定的三个二维数组（大小给定），将前两个的矩阵乘法结果保存至第三个。

在普通的三层循环模式中，下示第二种具有最小的平均 miss 率。因为它对 A、B、C 访问的步长为最小。如图（假定缓存块大小为 8 个矩阵元素）：

Matrix Multiplication (`ijk`)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

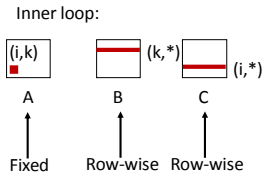


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

Matrix Multiplication (`kij`)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

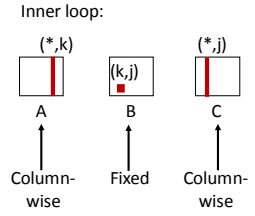


Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

Matrix Multiplication (`jki`)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

图：计算矩阵乘法三种模式

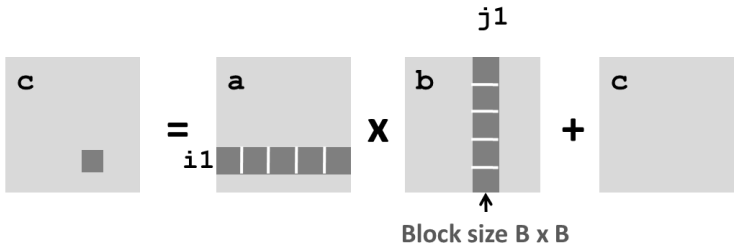
©Randy Bryant, CMU

但实际上，还存在一种效率更高的分块访问模式：如图所示，将三个矩阵分别分为一定大小的块，在外层（三层）循环中控制块、内层（三层）中控制对块的内部的访问。

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                  for (i1 = i; i1 < i+B; i1++)
                      for (j1 = j; j1 < j+B; j1++)
                          for (k1 = k; k1 < k+B; k1++)
                              c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



图：分块法计算矩阵乘法

©Randy Bryant, CMU

若块的长宽为 B，则在第三层循环中，每块内对 a 和 b 的访问产生 $B^2/8$ 次 miss；且包含对 2n/B 块的访问，程序共产生 miss 数目为 $n^3/(4B)$，而上述非分块法产生最小 miss 数为 $n^3/4$。保证 $3B^2$ 小于 L1 大小而取 B 的最大值，即可使得总 miss 率取得最小。