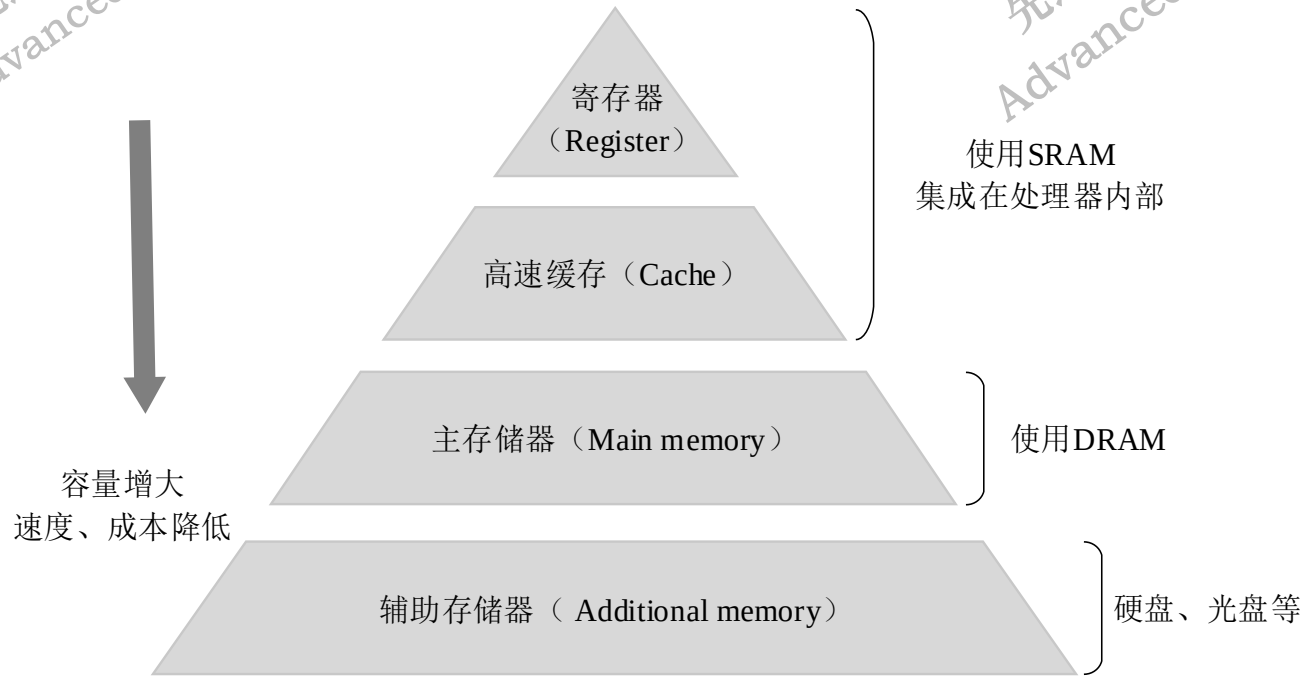
编译论坛

访存优化 I

嘉宾：王梦园

访存优化

为了平衡成本、保持存储容量及访存速度，现代计算机的存储系统中一般采用多种不同的存储器件结合即多级层次存储结构。如下图所示，最上层的寄存器在处理器芯片内直接参与运算，它的速度最快、平均每位的价格最高、容量最小。高速缓存存储器、主存储器、辅助存储器这三个级别的存储容量依次增大，但存储速度、价格成本及处理器访问的频度依次降低。



目录

寄存器优化

- 寄存器分配
- 寄存器重用

寄存器分配-减少全局变量

全局变量的有效范围在整个程序内，且全局变量会独占一个寄存器，导致过程内可分配寄存器的数量减少，因此编码时应尽量减少全局变量的使用。

```
float x=5.5642;//一个随机的float值
int function(float *a,int N){
int i;
float phi=2.541, delta, alpha;
delta = x * x;
alpha = x / 2;
for (i = 0; i < N; i++)
    a[i] = x * phi;
return 0;
}
```

```
int function(float* a, int N){
int i;
float x=5.5642;//一个随机的float值
float phi = 2.541, delta, alpha,temp;
delta = x * x;
alpha = x / 2;
for (i = 0; i < N; i++)
   a[i] = x * phi;
return 0;
}
```

寄存器分配-直接读取寄存器

一般情况下，编译时主要对标量进行分配寄存器，因此在编写程序时应该尽量将数组变为标量，这样可以直接读取寄存器中的数据，从而避免每次都从缓存中加载数据，减少部分程序中数据读写耗费的时间。

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

```
for (i = 0; i < n; i++) {
    x = a[i];//将数组a[i]的读取结果替换为x
    y = b[i];//将数组b[i]的读取结果替换为y
    c[i] = x + y;
    d[i] = x - y;
}
```

对优化前后示例进行测试，结果显示，标量替换前耗时0.96秒，替换优化后耗时0.88秒。

寄存器分配-直接读取寄存器

除此之外，此方法还可以减少内存写的次数。对代码进行测试，设置输入的数据规模为10000*10000，结果显示优化前耗时0.35s，优化后耗时0.28s，可见对内存写操作的减少会提升程序的性能。

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i] = a[i]+ b[i][j];//每次迭代都需要访问数组a
```

```
int sum;//使用sum将数组a的数据保存在寄存器中
for (i = 0; i < n; i++) {
    sum = a[i];//这样就可以减少对内存的写操作
    for (j = 0; j < n; j++) {
        sum = sum + b[i][j];
    }
    a[i] = sum;
}
```

寄存器分配-直接读取寄存器

优化人员除了需要考虑寄存器合理分配的问题之外，还需要防止寄存器溢出。当所需寄存器的数量大于可分配寄存器数量时，就会出现寄存器溢出，可能会抵消前期调优积累的性能优势。

```
int a[N], b[N], c[N];
int i;
int main() {
    for (i = 0; i < N; i++)
        a[i] = b[i] * c[i];
}
```

```
STORE R1，0(SP) #寄存器溢出
LOOP:
        LOAD R1，b[i]
        LOAD R2，c[i]
        MUL R1， R2， R1
        STORE R1，a[i]
LOOP END
LOAD R1，0(SP) #寄存器溢出
```

寄存器重用-寄存器重用

当数据从缓存加载到寄存器后，应该尽可能地将后续还要使用的数据保留在寄存器，以避免该数据再次从缓存读取，即寄存器重用可以有效地减少内存访问。

```
void func(double* A, double* B, int NI, int NJ) {
  for (int ii = 0; ii < NI; ii += Ti) {
    for (int j = 0; j < NJ; ++j) {
      for (int i = ii; i < min(ii + Ti, NI); ++i) {
        A[i] += B[j * NI + i]; // S
      }
    }
  }
}
```

```
void func_unroll4(double* A, double* B, int NI, int NJ) {
  for (int ii = 0; ii < NI; ii += Ti) {
    for (int j = 0; j < NJ; j += 4) {
      for (int i = ii; i < min(ii + Ti, NI); ++i) {
        A[i] += B[j * NI + i];
        A[i] += B[(j + 1) * NI + i];
        A[i] += B[(j + 2) * NI + i];
        A[i] += B[(j + 3) * NI + i];
      }
    }
  }
}
```

AdvancedCompiler

Tel: 13839830713

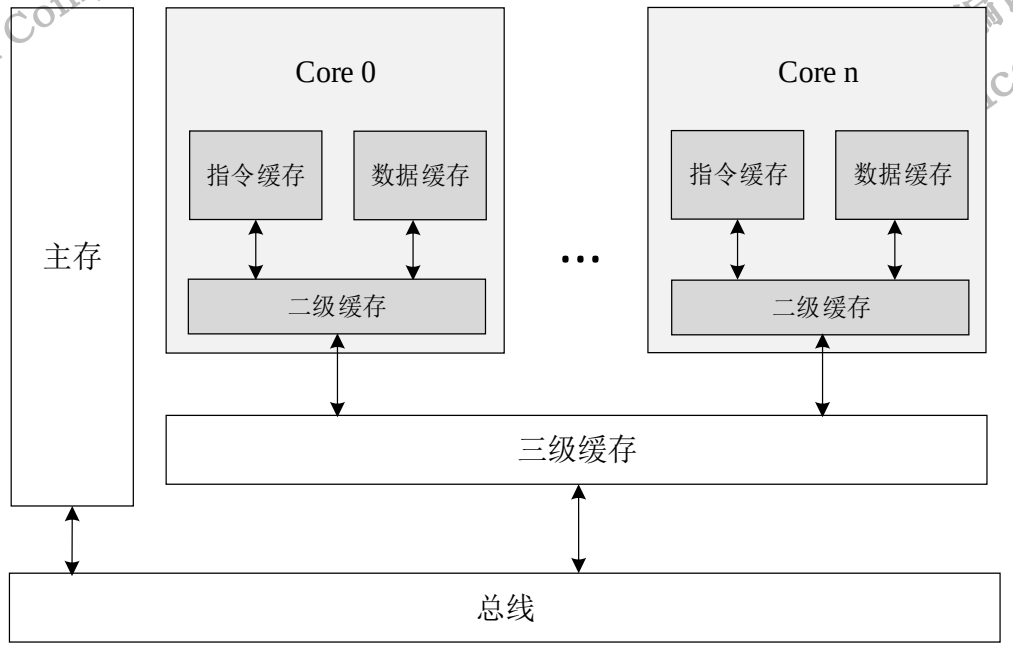
编译论坛

访存优化 II

嘉宾：王梦园

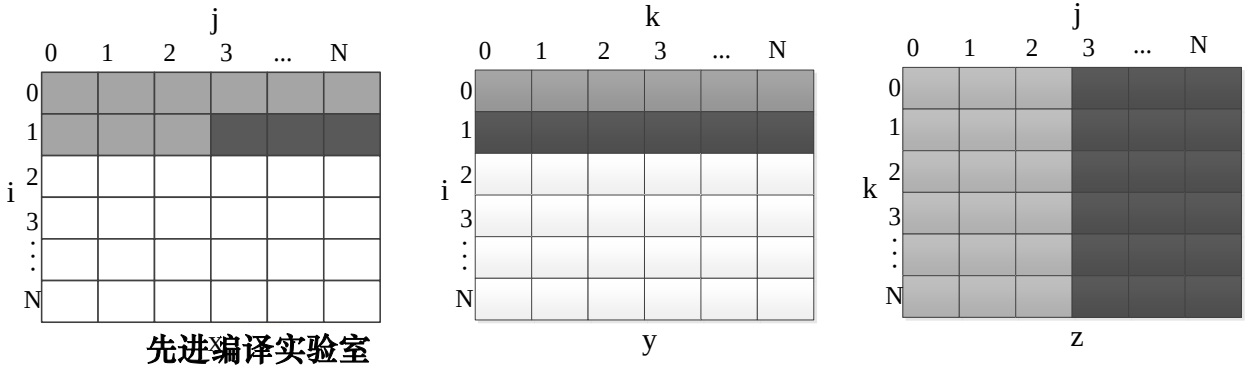
缓存优化

处理器缓存内部分为一级缓存和二级缓存，以及在多核处理器中常见的三级缓存，其中一级缓存又分为指令缓存和数据缓存两部分。这种结构缓和了处理器与主存之间速度不匹配的矛盾，使存储速度、存储容量与处理器功耗之间的关系相对平衡。



缓存分块

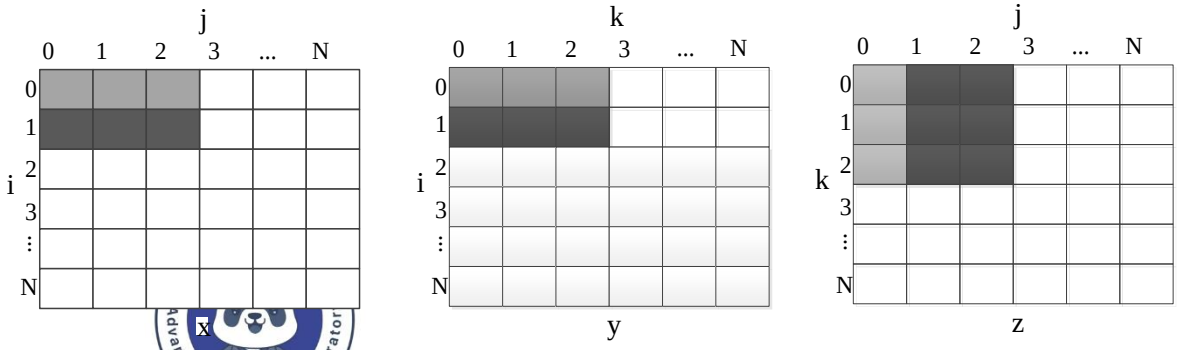
以矩阵乘代码为例具体说明缓存分块技术的使用方法。这段代码中两个内部循环读取了数组z的全部N*N个元素，以及数组y的某一行中的N个元素，所产生的N个结果被写入数组x的某一行。下图为i=1时，原始循环中三个数组的访问情况，其中黑色表示最近被访问过，灰色表示早些时候被访问过，而白色表示尚未被访问。




```
void matrixmulti(float N, float**x, float** y, float** z){
    int i, j, k,r;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            r = 0;
            for (k = 0; k < N; k++) {
              r = r + y[i][k] * z[k][j];
            }
            x[i][j] = r;
        }
    }
}
```

缓存分块

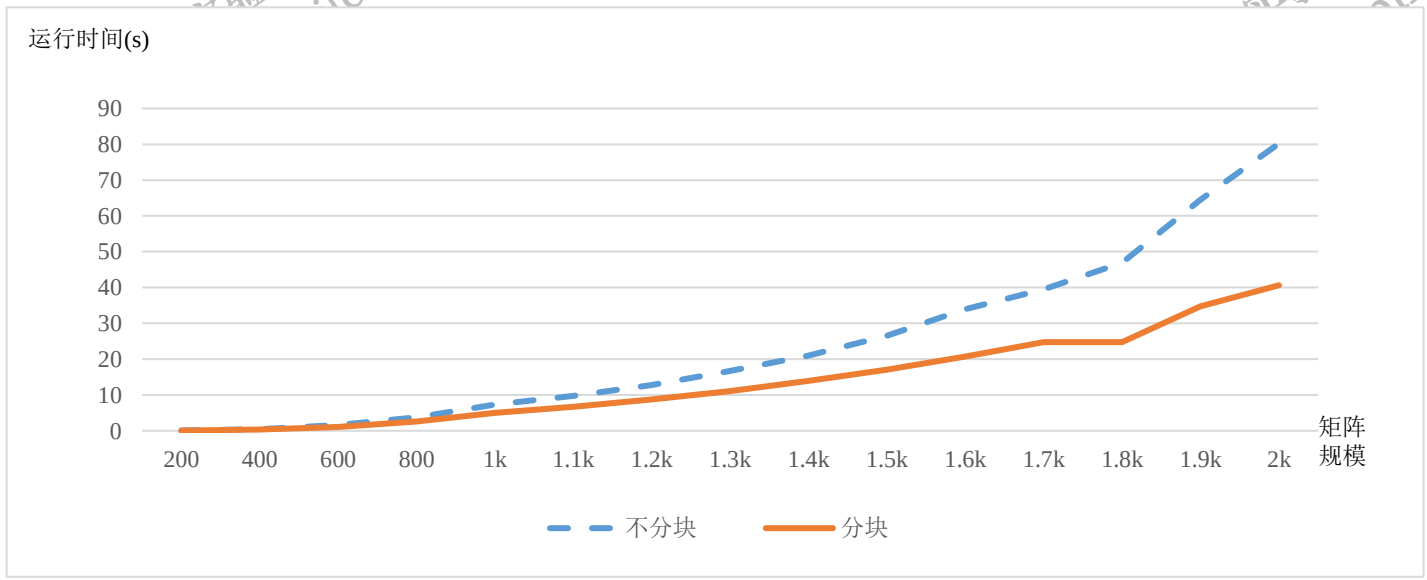
对原始循环中的i，j层循环分别进行分块，将原始的大矩阵分割为了若干个小矩阵，嵌套循环每次对一个小矩阵内的数据进行计算得出一个部分结果，下图说明了分块后三个数组的访问情况。




```
void matrixmulti_1(int N, int** x, int** y, int**
z,int S)
{//S为分块后小矩阵的长度
    int kk,jj,i, j, k, r;
    for (jj = 0; jj < N; jj = jj + S) {
        for (kk = 0; kk < N; kk = kk + S) {
            for (i = 0; i < N; i++) {
                for (j = jj; j < min(jj + S , N); j++) {
                    r = 0;
                    for (k = kk; k < min(kk + S , N); k++)
                        r = r + y[i][k] * z[k][j];
                        x[i][j] = x[i][j] + r;
                }
            }
        }
    }
}
```

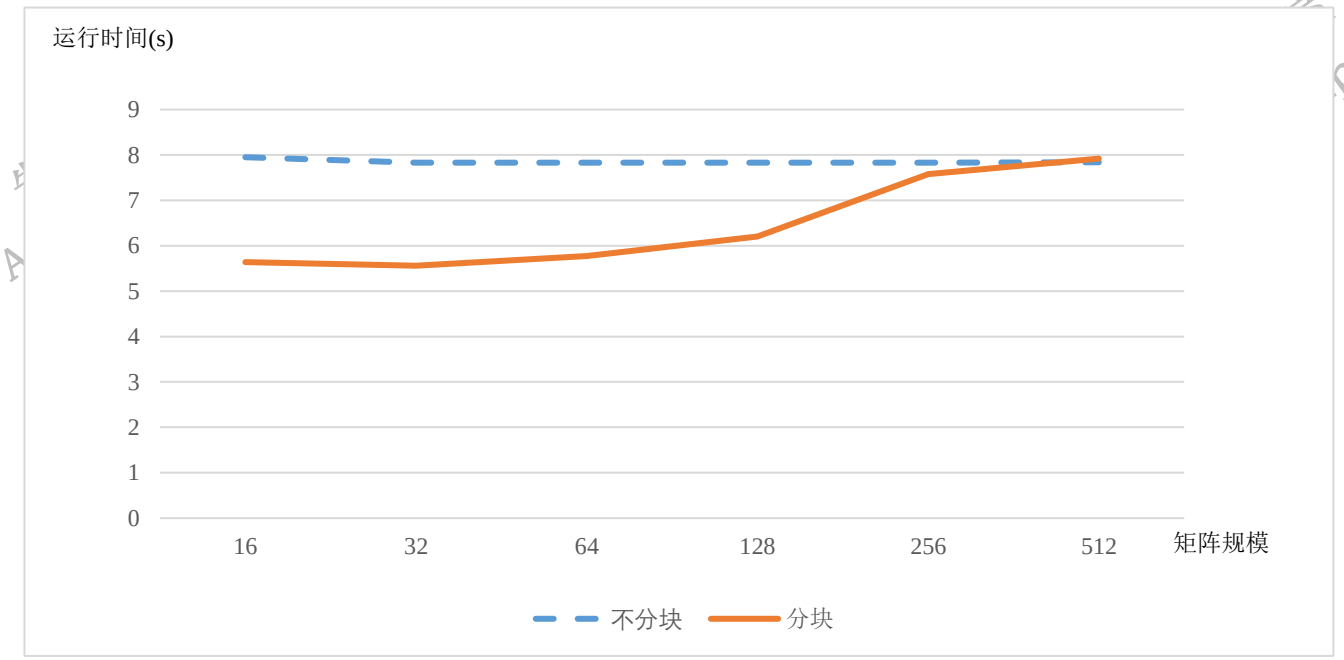
缓存分块

为进一步说明分析结果，针对上文中的矩阵乘示例使用x86架构平台进行测试。当固定分块大小为50时，随着矩阵规模的不断增大，分块后的矩阵乘程序相对于未分块程序的性能提升越来越明显，如下图所示。



缓存分块

当固定矩阵规模为1024时，此时三个矩阵并不能全部保存在缓存中。随着分块大小的逐渐提高，分块前后运行时间对比如图所示。



缓存分块

造成上述现象的部分原因为高速缓存不同的映射策略以及替换策略，地址映射就是主存地址与高速缓存地址的对应方式，常见的缓存地址映射策略包括直接相联、组相联和全相联。

全相联高速缓存的映射策略为内存中的每个数据块均能够被映射到缓存中的任意一个缓存行，但其请求数据时需要将其地址的标记位与所有缓存行标记位进行对比，花费代价较大。

直接相联的映射方式是内存中的每一个数据块都只能存放在缓存中的一个特定的缓存行，但每个主存块只有一个固定位置可存放，容易产生冲突，使缓存效率下降。

组相联映射实际上是直接映射和全相联映射的折中方案，映射方法是将主存和缓存都分组，组间采用直接映射，组内采用全相联映射。尽量避免了全相连和直接相连的缺点，适度兼顾二者的优点，缓存命中率较高，因而得到普遍采用。

缓存分块

结合上文中矩阵乘的例子，可以将数组按照读取效率从高到低分为三类：

一类数组

这类数组是最好的情况，其矩阵行的长度是缓存行大小的整数倍。当程序访问一个缓存行行大小的数据时，一类数组的这些数据会在一个缓存行中，缓存命中率高。

二类数组

第二类数组其矩阵行的长度不是缓存行的整数倍。这种情况下，程序运行时访存数据很有可能需要跨越两个缓存行，与一类数组相比缓存命中率会变低。

三类数组

第三类数组其矩阵行是整个缓存大小的整数倍。这种情况下，数组的每一列数据均映射到同一缓存行组，当进行数组转置时缓存的利用率非常低，会多次发生缓存不命中的情况，严重影响程序执行效率。

缓存分块

在矩阵乘计算或者其它类似程序中，这三类数组的计算效率差异很大。所以优化人员应结合缓存映射方式，尽可能避免三类数组或者二类数组的出现。例如在计算时对对矩阵的行列进行扩充，使其变为一类数组，从而获得性能的提升。以矩阵乘代码为例，对其进行不同规模下运行时间的测试，结果如下。

矩阵规模	256*256	251*251
程序耗时	126	161

```
void matrixmulti(float N, float**x, float** y,
float** z)
{
    int i, j, k,r;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            r = 0;
            for (k = 0; k < N; k++) {
                r = r + y[i][k] * z[k][j];
            }
            x[i][j] = r;
        }
    }
}
```

缓存分块

在此例子中，若将数据由251列扩展到256列，扩充的部分使用0进行补齐，这样矩阵行的长度就是缓存行的整数倍，即从二类数组变成了一类数组，可以增加缓存命中率。

在进行行列扩充优化后，实际的有效数据量依然为251*251，但优化后矩阵乘耗时为100ms，加速比达到了1.6倍左右。

```
for (i = 0; i < n; i++) {
   a[i] = (float*)realloc(a[i], sizeof(float) *
Cache_len);
   b[i] = (float*)realloc(b[i], sizeof(float) *
Cache_len);
}
for (i = 0; i < n; i++) {
   for (j = n; j < Cache_len; j++) {
      a[i][j] = 0;
      b[i][j] = 0;
   }
}
for (i = 0; i < n; i++){
   for (j = 0; j < n; j++){
      c[i][j] = 0;
      for (k = 0; k < n; k++)
         c[i][j] += a[i][k] * b[k][j];
   }
}
```

AdvancedCompiler

Tel: 13839830713

编译论坛

访存优化 III

嘉宾：王梦园

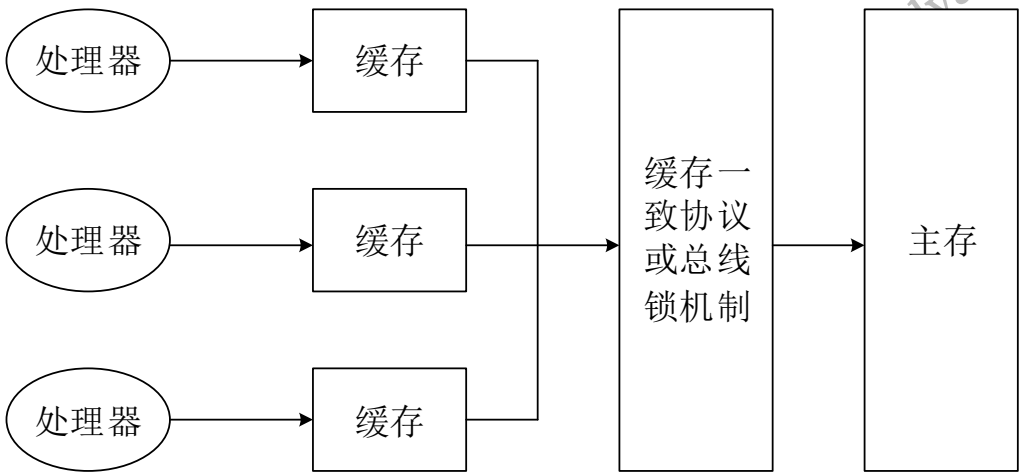
目录

缓存优化

- 减少伪共享
- 数据预取

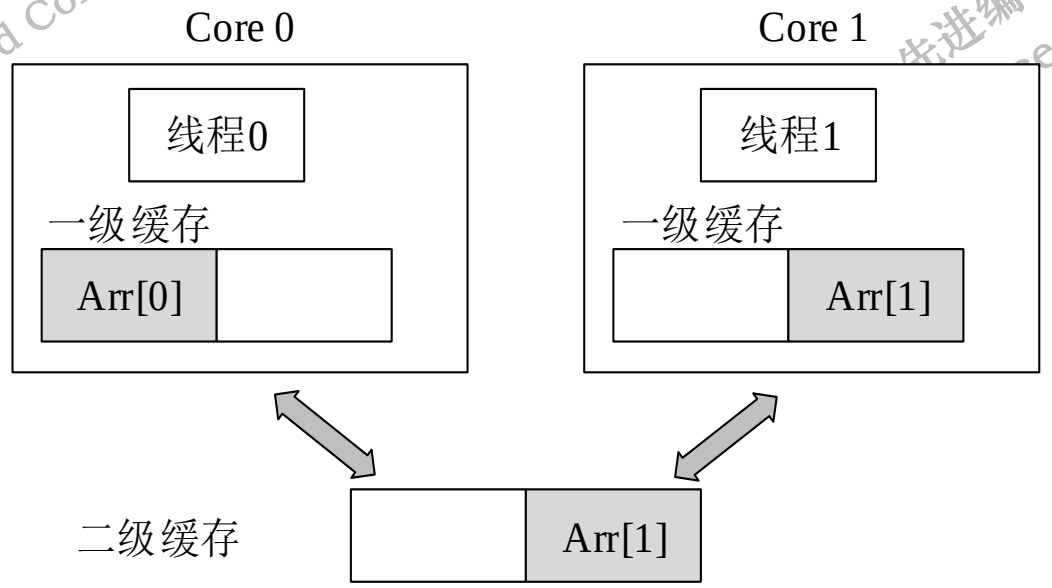
减少伪共享

为解决数据一致性的问题，需要处理器各个核心访问缓存时都遵循缓存一致性协议。即当某个处理器核心修改缓存行数据时，其它的处理器核通过监听机制获悉并使其共享缓存行失效，此时该处理器核心将修改后的缓存行写回到主内存中。若其它处理器核需要此缓存行共享数据，则从主内存中重新加载，并放入缓存，这样可以保证读取数据的正确性，如图所示。



减少伪共享

在多核CPU系统中，由于要操作的不同变量处于同一缓存行，某核心更新缓存行中数据并将其写回缓存，同时其它核心会使该缓存行失效，使用时需要从内存中重新加载，这种情况就是缓存行的伪共享问题，伪共享会导致大量的缓存冲突，应尽量避免。



减少伪共享

多线程编程时往往不可避免的要遇到数据共享，编程时应该注意如下原则：

尽量少的使用共享数据，可以将不同线程操作的数据分配在不同的缓存行中，或者进行缓存行填充，避免多个线程共享同一缓存行内的数据。

在多个核心共享同一缓存行数据时，如果不进行对缓存的写入，是不会发生伪共享问题的，所以应当尽量少的修改数据。例如上述的例子中，若核心0与核心1频繁地对Arr[0]以及Arr[1]进行修改，代价是非常高的。

减少伪共享

在内核版本4.1.0及以上的Linux系统中，perf工具包含了一些对程序中伪共享的分析功能，称为perf c2c。perf c2c可以收集高速缓存的性能数据，之后基于采样数据生成报告，以此代码为例说明如何使用perf c2c进行伪共享分析。

```
int main(void)
{
    int sum[THRAED_NUM];
    #pragma omp parallel for
    for (int i = 0; i < THRAED_NUM; i++){
        for (int j = 0; j < N; j++){
            sum[i] += values[j] >> i;
        }
    }
    return 0;
}
```

减少伪共享

sum数组大小为64字节，在实验测试平台中恰好占用一个缓存行。当8个线程同时对该缓存行内的sum数组进行写时，程序将不可避免地发生伪共享现象。使用perf c2c对该程序的执行过程进行分析，在Linux中输入以下指令。

```
# gcc –fopenmp –g false_share_test.c –o false_share_test  //使用gcc编译器编译目标文件
# perf c2c record ./false_share_test   //通过采样，收集性能数据
# perf c2c report –stdio     //基于采样数据，生成报告
```

生成的报告如下所示，为更清晰的予以说明，对表中的输出结果适当进行了简化。

```
Total records                              :     65407
......
LLC Misses to Local DRAM                   :     36.9%
LLC Misses to Remote DRAM                  :     33.8%
LLC Misses to Remote 缓存 (HIT)            :     0.0%
LLC Misses to Remote 缓存 (HITM， Hit In The Modified)          :     29.2%
......
```

减少伪共享

为避免多个线程频繁访问同一缓存行，减少伪共享问题，修改后的程序如代码所示。修改后的程序避免了多个线程同时频繁地访问同一个缓存行，此时再次使用pref c2c分析，可以发现程序中的伪共享现象大幅度减少。

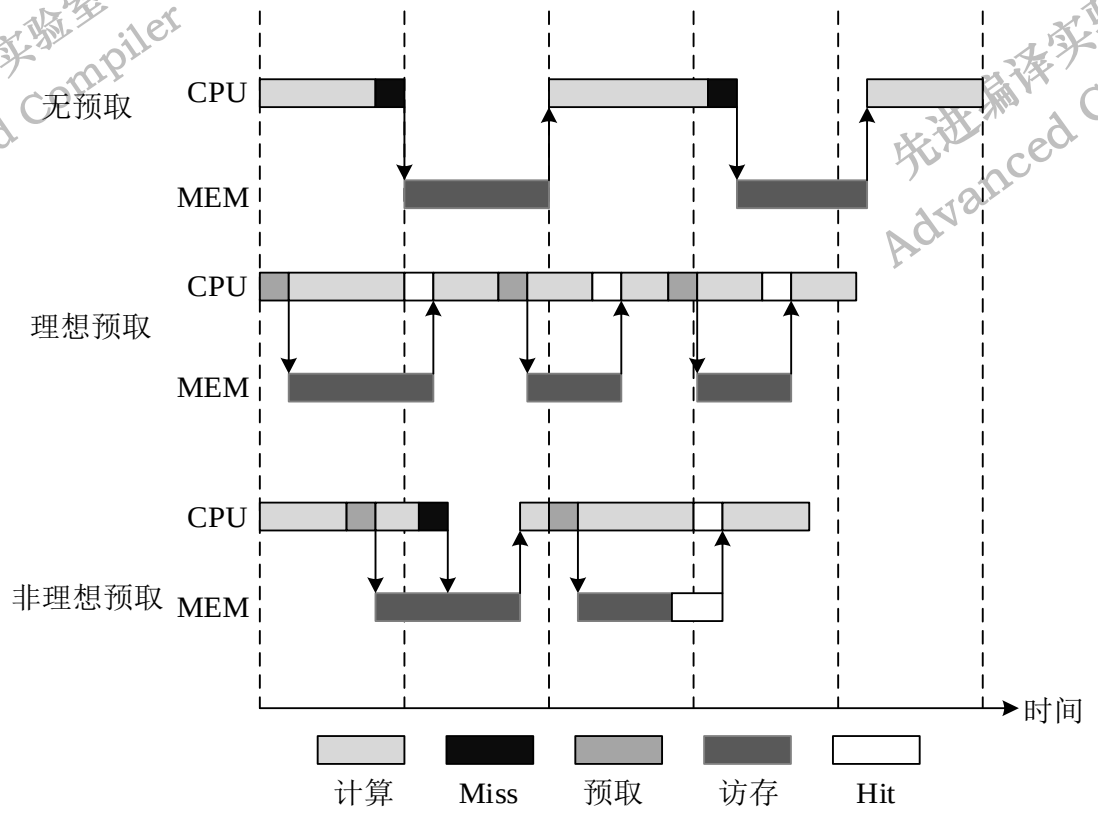
```
int main(void)
{
    int sum[THRAED_NUM];
    #pragma omp parallel for
    for (int i = 0;  i < THRAED_NUM;  i++){
        int local_sum;
        for (int j = 0;  j < N;  j++){
            local_sum += values[j] >> i;
        }
        sum[i] = local_sum;
    }
    return 0;
}
```

```
LLC Misses to Local DRAM        :      88.7%
LLC Misses to Remote DRAM       :       9.9%
LLC Misses to Remote 缓存 (HIT)    :       0.0%
LLC Misses to Remote 缓存 (HITM)   :       1.4%
......
```


先进编译实验室
Advanced Compiler

数据预取

数据预取将待处理器所需的数据提前加载到缓存中，避免缓存不命中的情况出现。按照数据预取的效果，可将数据预取分为无预取、理想预取和非理想预取三种，如图所示。



硬件预取与软件预取

预取的实现通常有软件预取和硬件预取两种方式。硬件预取是提前把指令和数据预取到缓存中的一种硬件机制，其不需要使用预取指令或额外的编写代码。但硬件预取必需建立在能动态进行程序执行分支预测的基础上，所以预取的范围很小，且会增加系统整体开销。

在绝大部分情况下，程序对内存的访问模式是随机的、不规则的。此时需要使用软件预取，通过插入的预取指令，提前把数据取入缓存，这种方法对系统开销的影响较小，也不会减慢访问缓存的速度，是一种灵活有效的数据预取方式。

硬件预取与软件预取

本节主要介绍软件预取对程序性能的影响，软件预取通过插入预取指令或者函数实现，使用预取函数__builtin_prefetch()，该函数原型为void__builtin_prefetch (const void*addr,rw,locality)，在测试用例合适位置插入预取指令，使用LLVM7-1版本的编译器对加入预取指令的程序进行编译，结果显示加入预取指令比未加入指令的执行时间快了13%。

```
__builtin_prefetch(mul1, 0, 3);
__builtin_prefetch(mul2, 0, 0);
__builtin_prefetch(res,  1, 3);
nop;nop;nop;
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        for (k = 0; k < N; ++k)
            res[i][j] += mul1[i][k] * mul2[k][j];
```

AdvancedCompiler

Tel: 13839830713

编译论坛

访存优化 IV

嘉宾：王梦园

目录

内存优化

- 减少内存读写
- 数据对齐
- 直接内存访问
- 访存与计算重叠

减少内存读写

内存又可以叫做主存，是处理器能直接寻址的存储空间，由半导体器件制成。以DDR3内存来说，通常一次内存读写大约要200~400个时钟周期，访问内存速度非常慢，因此在程序优化的过程中应当优先充分使用寄存器而不是访存。大多数情况下，编译器能够很好地解决这个问题，但是在具有存储器别名情况或读写依赖情况下，需要优化人员手动处理。

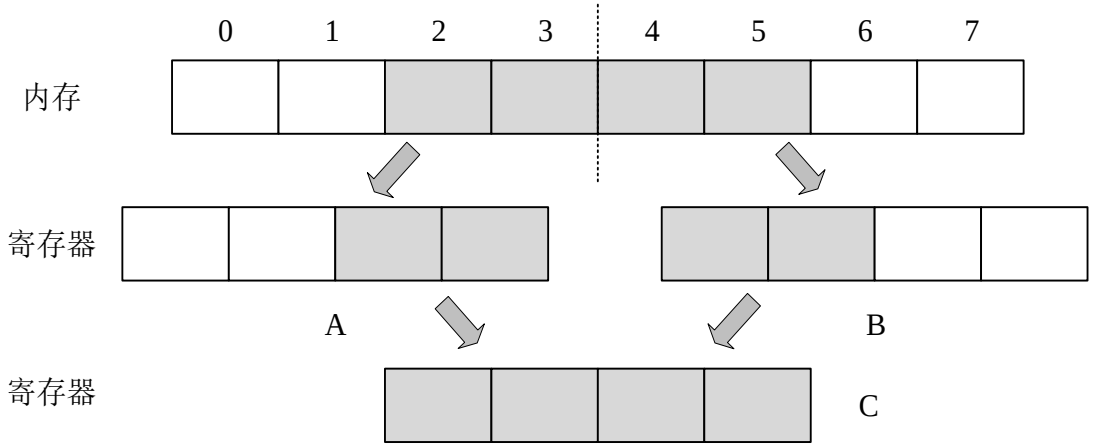
```
for (int i = 1; i < n; i++)
    a[i] += a[i - 1];
```

```
temp = a[0];
for (int i = 1; i < n; i++) {
//通过保存临时中间计算结果减少一些内存访问
    temp += a[i];
    a[i] = temp;
}
```

数据对齐

当处理器访问正确对齐的数据时，它的运行效率最高。当数据值没有正确对齐时，处理器需要产生一个异常条件或执行多次对齐的内存访问，以便读取完整的未对齐数据，导致运行效率降低。所以处理器提供的对齐的数据访问指令效率要远高于非对齐的数据访问指令。

在32位处理器中，一个int型变量占4个byte，假设这个变量i在内存中占据2、3、4、5这4个byte的位置，数据非对齐存储如图所示。

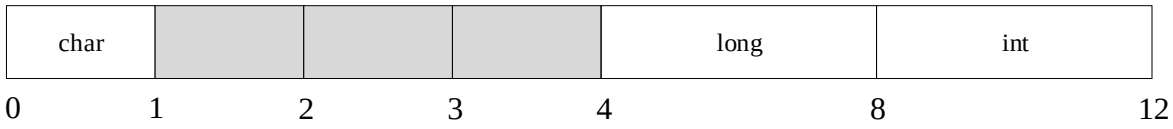


结构体对齐

结构体分配内存空间时采用的对齐规则为，变量的起始地址能够被其对齐值整除，结构体变量的对齐值为最宽的成员大小；结构体每个成员相对于起始地址的偏移能够被其自身对齐值整除，如果不能则在前一个成员后面补充字节；结构体总大小能够被最宽的成员的大小整除，如不能整除则在后面补充字节。

例如A结构体的各成员所占内存空间大小为sizeof（a）+sizeof（b）+sizeof（c），即1+4+4=9，而实际上结构体的大小为sizeof（A）=12。内存分配如图所示。

```
struct A {
    char a;
    long b;
    int c;
}
```



结构体对齐

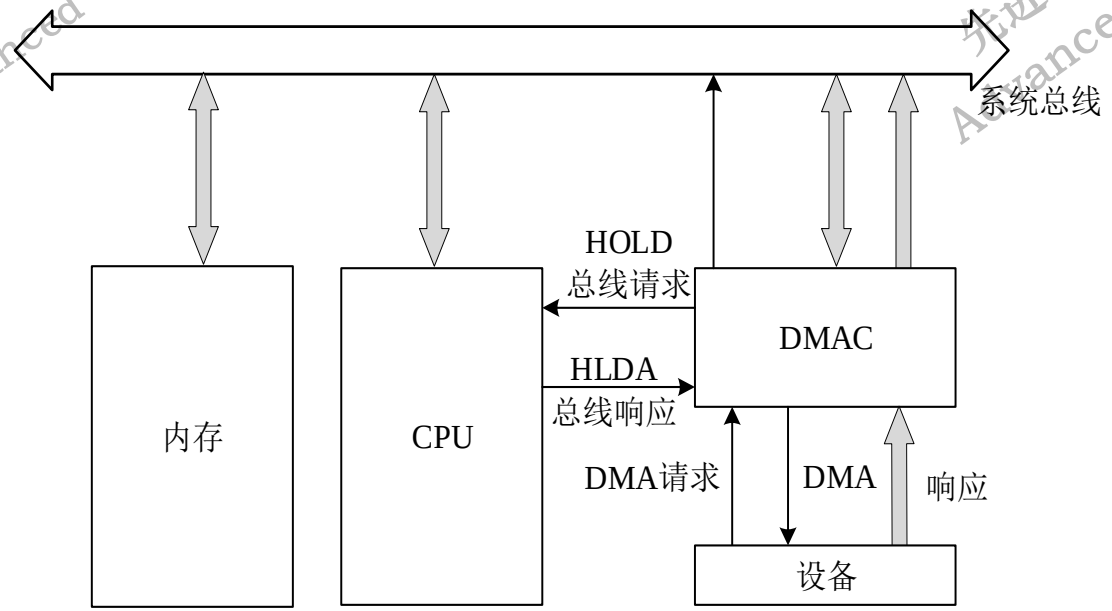
在定义结构体时，应按照成员大小从小到大或从大到小依次定义各成员。建议尽量大数据类型在前，小数据类型在后，一方面这样会节省一些空间，另一方面可以更好地满足处理器的对齐要求。

例如结构体A1需要分配12字节的内存空间，将结构体内成员顺序进行调整后得到A2结构体，只需要分配8个字节空间，处理器访问2次内存就可读完数据。

struct A1{ *char a;* *int b;* *char c;* *short d;* *};*	*struct A2 {* *char a;* *char c;* *short d;* *int b;* *};*

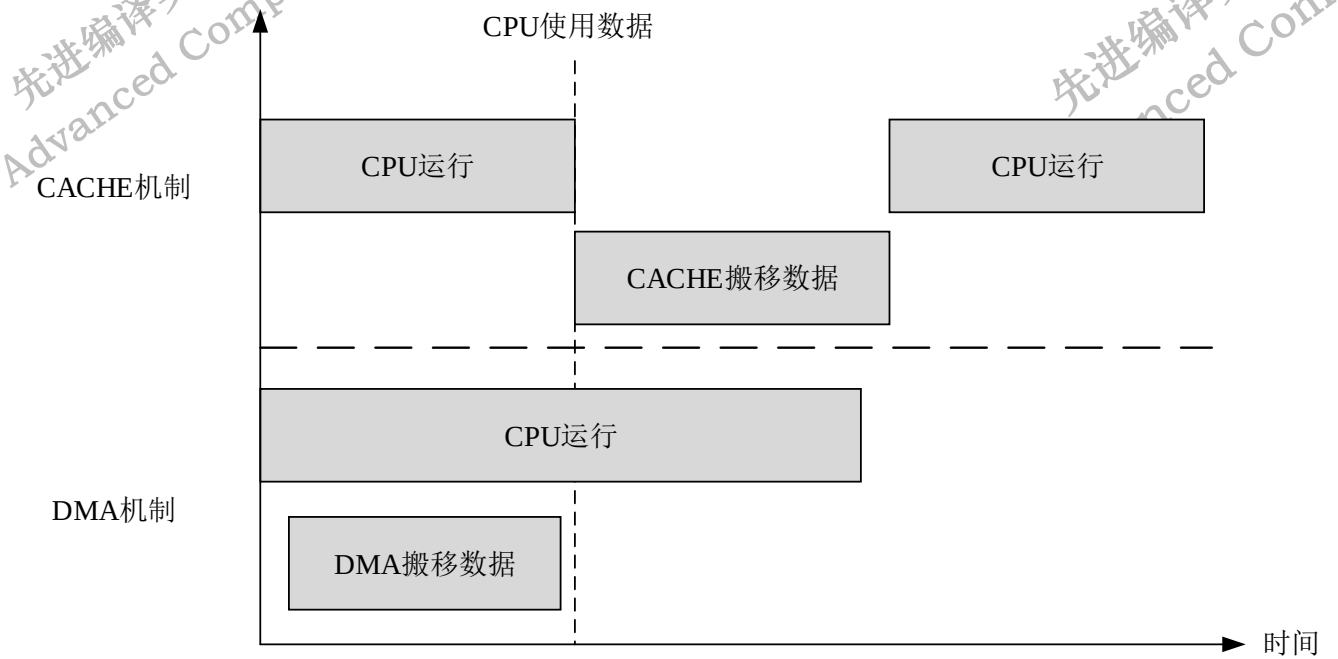
直接内存访问

直接内存访问（Direct Memory Access,DMA），是一种广泛应用的硬件机制，可以直接传输外围设备和主内存之间的数据，这样处理器可以直接使用这部分的数据，这种机制可以提升内设备数据传输的效率，DMA机制如图所示。



直接内存访问

DMA机制与上文介绍的缓存机制的不同之处在于，DMA在处理器需要数据时会提前将数据搬移到处理器内，而缓存机制则是在需要数据的时候才搬移数据，因此利用DMA机制，程序的执行时间比利用缓存机制的更短。



直接内存访问

以德州仪器C6678板卡为例，若数据存放在DDR中，需要读取到缓存中就可以使用EDMA机制实现。在循环中EDMA读数据需要采用EDMA _WAIT等待数据传输结束，伪代码如下。

```
for(i=0；i<ProcCnt；i++){
   Coherence；//缓存、预取数据一致性操作
   Sync()；//同步
   if(iCoreID==Core0){
      EDMARead(pInBuf，EDMA_WAIT，EDMA_NTCC_L3)；
   }
   Sync()；
   Processing(pInBuf，pResultBuf)；
}
```

访存与计算重叠

本书前文中提到的数据预取和直接内存访问，目的都是为了数据在使用前能从内存调入缓存中，从而减少处理器停顿的访存优化方法。除了上述两种方法外，还可以在指令层次将访存与计算部分重叠，可以有效地解决访存的延迟。以德州仪器TI C66xx平台的汇编代码为例进行说明。

SUB R6,R7,R5	*SUB R6,R7,R5*
MUL R6,R7,R8	*||LOAD R1,a[i]*
LOAD R1,a[i]	*MUL R6,R7,R8*
LOAD R2,b[i]	*||LOAD R2,b[i]*
ADD R1,R2,R3	*ADD R1,R2,R3*
MUL R1,R2,R4	*MUL R1,R2,R4*

AdvancedCompiler

Tel: 13839830713

编译论坛

访存优化 V

嘉宾：王梦园

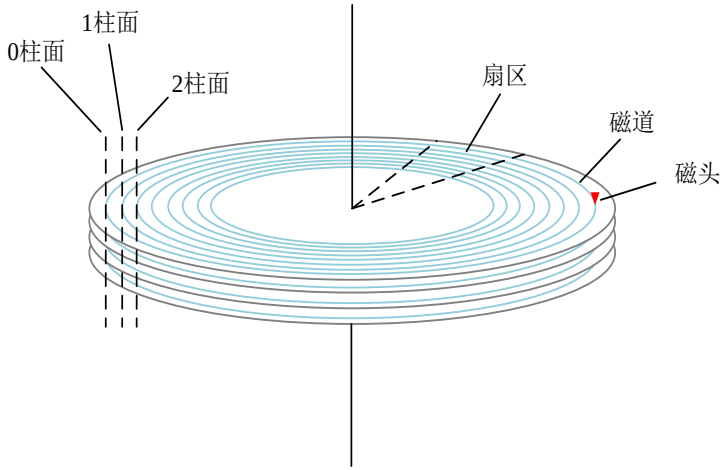
目录

磁盘优化

- 多线程操作
- 避免随机写
- 磁盘预读

磁盘优化

磁盘是由大小相同且同轴的圆形盘片组成，如下图所示。由于存储介质的特性，内存比磁盘的读写速度要快很多，但内存容量要远小于磁盘，程序的执行要调入内存后才能执行，所以内存和磁盘要经常进行I/O操作，而磁盘的I/O涉及机械操作，因此为了提高程序效率，要尽量减少磁盘的输入输出I/O。本节主要介绍针对磁盘的常用优化方法。



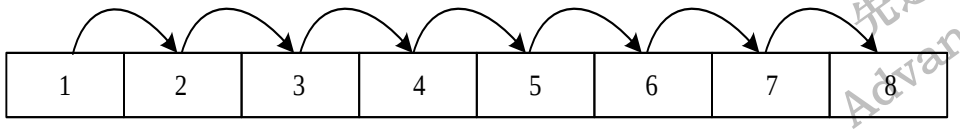
多线程操作

多线程随机读的处理速度可以达到单线程随机读的10倍以上，但同时会导致响应时间增大。结论表明增加线程数，可以有效的提升程序整体的I/O处理速度。但同时，也使得每个I/O请求的响应时间随之上升。下表中统计了随着线程数增加多次读数据平均耗时的变化。

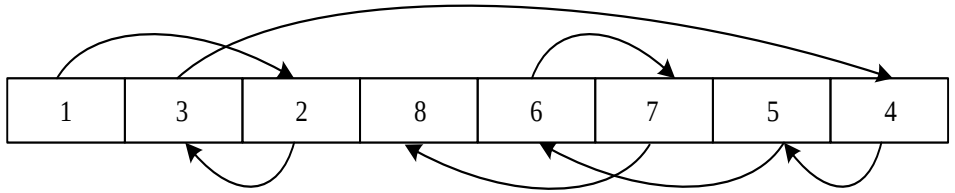
读线程数	读出100次耗时	读平均相应时间
1	1329575	13294
5	251769	12977
10	149201	15989
20	126753	25453
50	96596	48355

避免随机写

磁盘读取数据花费的读取时间，是由读取数据大小和磁盘密度、磁盘转速决定的固定值共同决定。若磁盘为顺序访问，即相邻两次I/O操作的逻辑块起始地址也是相邻的，此时磁头几乎不用换道，或者换道的时间很短，反之若为随机写会导致磁头不停地换道，造成效率的极大降低，如图所示。



顺序访问



随机访问

避免随机写

要想改进这种单线程随机写慢的问题，可以通过改变对磁盘的访问模式来减寻道时间和潜伏时间，即将完全随机写变成有序的跳跃随机写。具体操作是通过将数据在内存中缓存并进行排序，使得在写盘的时候不是完全随机的，而是使得磁盘磁头的移动只向一个方向，缩短磁盘的寻址时间。

磁盘预读

为减少磁盘的读入写出，磁盘可以采用数据预读的方式将所需的部分数据放入内存。当确定了要进行顺序预读时，需要决定合适的预读大小。为此，Linux采用了一个快速的窗口扩张过程，首次预读设置readahead_size = read_size * 2，即预读窗口的初始值是读大小的二倍，后续的预读窗口将逐次倍增，直到达到系统设定的最大预读大小。当然，预读大小不是越大越好，在很多情况下也需要同时考虑I/O延迟问题。 在进行预读时，可以使用Linux平台上的current窗口和ahead窗口来跟踪当前顺序流预读状态。

参考资料

AI框架发展白皮书（2022年）

https://syncedreview.com/2020/12/14/a-brief-history-of-deep-learning-frameworks/

先进编译实验室
王磊

2022年09月

AdvancedCompiler

Tel: 13839830713

编译论坛

访存优化 VI

嘉宾：王梦园

目录

数据布局

- 数组重组
- 数组转置
- 结构属性域调整
- 结构体拆分
- 结构体数组转换

数据重组

程序的核心循环中常常存在多个数组之间的运算，而这些数组在内存中并不是连续存放的，会导致程序的访存局部性较差。此时可以使用数组重组的方式将多个数组合并成为结构体数组，该结构体的属性域为重组前各数组的元素，从而提高程序访问数据的局部性。

```
for (int i = 0; i < n; i++)
sum[i] = a[i] + b[i] + c[i];
```

```
typedef struct {//提高数组局部性
    int a, b, c;
    int sum;
}arr_struct;
for (int i = 0; i < n; i++)
    arr[i].sum = arr[i].a + arr[i].b + arr[i].c;
```

数据转置

当循环访问数组中元素时，若最内层循环对数组的索引方式与内存中的存放方式不同，会导致数据的访问不连续，无法充分利用程序的空间局部性。针对这一问题，可以使用数组转置的方法对数组的数据布局进行变换，使得内层循环对数组的访问连续。

```
for (int i = 0; i<n; i++) {
    x[i][1] = x[i][1] + phi * y[i][1];
    x[i][2] = x[i][2] + phi * y[i][2];
}
```

```
for (int i = 0; i < n; i++) {
    x[1][i] = x[1][i] + phi * y[1][i];
    x[2][i] = x[2][i] + phi * y[2][i];
}
```

结构属性域调整

程序中访问结构体变量时，常被访问的可能是少量的属性域，如果改变这些结构的定义，使经常被访问的属性域组织在一起，能够有效地提高结构定义变量的空间局部性。示例如下。

```
typedef struct {
    float t_x, t_y, t_z;//x维的时间t_x、速度v_x和位移d_x在内存中并不是连续存放的
    float v_x, v_y, v_z;
    float d_x, d_y, d_z;
} motion;
```

```
typedef struct {
    float t_x, v_x, d_x;//对整个结构的属性域进行调整，改进数组在内存中存放的连续性
    float t_y, v_y, d_y;
    float t_z, v_z, d_z;
} motion_1;
```

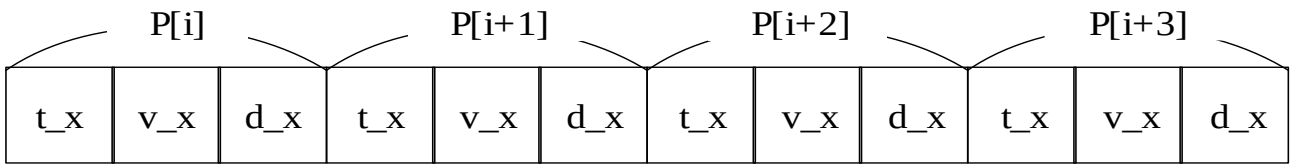
属性域调整后改善了内存中数据的连续性。经过测试，进行100000次循环迭代，结构属性域调整前耗时494us，调整后耗时395us。优化人员在编写程序时合理的运用此类技巧，可以使得程序的运行情况更好。

结构体拆分

除了结构体属性域调整能够改进数据的局部性，结构体拆分也能改进数据的局部性。上节列举的示例经过结构体属性域调整后，虽然同一次迭代内x维的时间t_x、速度v_x和位移d_x在内存中存放连续，但是相邻的迭代间P[i]和P[i+1]的数据在内存中还是不连续的，此时可以利用结构体拆分的方法进行改写。

```
typedef struct {//可以利用结构体拆分的方法进行改写
    float t_x, v_x, d_x;
} motion_x;
typedef struct {
    float t_y, v_y, d_y;
    float t_z, v_z, d_z;
} motion_yz;
```

将结构体拆分成三个结构体后，以motion_x为例，数据在内存中的布局如图示

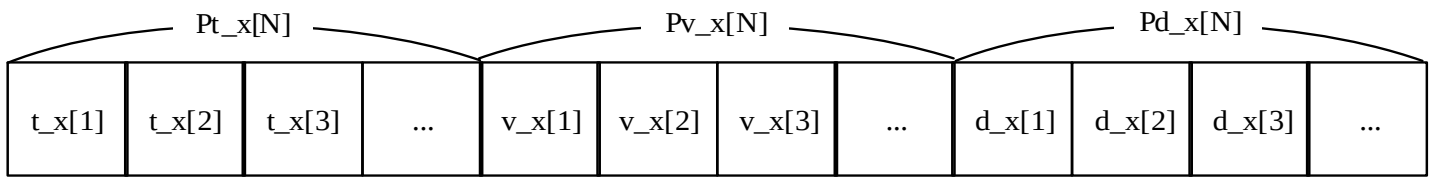


结构体拆分

继续以上述结构拆分的代码段定义的motion_x结构体为例，经过结构拆分之后相邻的迭代间P[i]和p[i+1]的数据已经连续，但t_x、v_x、d_x相邻迭代的数据在内存中依然不是连续的。此时可以使用结构体数组转为数组结构体的方法，将不连续的数据存放在数组结构体中，改进程序的数据布局情况。

```
typedef struct {
        float Pt_x[N];
        float Pv_x[N];
        float Pd_x[N];
} motion_x;
```

将结构体数组转换为数组结构体后，数据t_x、v_x和d_x相邻迭代的数据在内存中连续存放，如图所示。



总结

访存性能优化是程序性能优化中重要的组成部分。这部分内容从计算机多层次存储结构的基本概念出发，按照离处理器从近至远的顺序介绍了一些如何更好地利用多层次存储结构的程序优化方法，并说明了如何在编写程序时改善数据局部性，以提高程序的访存性能。

AdvancedCompiler

Tel: 13839830713