深 圳 大 学 实 验 报 告

课程名称：计算机系统(2)

实验项目名称：Cache 实验

学院：计算机与软件学院

专业：计算机与软件学院所有专业

指导教师：冯 禹 洪

报告人：郑杨 学号：2020151002 班级：腾班

实验时间：2022 年 6 月 23 日至 6 月 23 日

实验报告提交时间：2022 年 6 月 23 日

教务处制

一、实验目的：

1. 加强对 Cache 工作原理的理解；
2. 体验程序中访存模式变化是如何影响 cahce 效率进而影响程序性能的过程；
3. 学习在 X86 真实机器上通过调整程序访存模式来探测多级 cache 结构以及 TLB 的大小。

二、实验环境

X86 真实机器

三、实验内容和步骤

1、分析 Cache 访存模式对系统性能的影响

（1） 给出一个矩阵乘法的普通代码 A，设法优化该代码，从而提高性能。

（2） 改变矩阵大小，记录相关数据，并分析原因。

2、编写代码来测量 x86 机器上（非虚拟机）的 Cache 层次结构和容量

（1） 设计一个方案，用于测量 x86 机器上的 Cache 层次结构，并设计出相应的代码；

（2） 运行你的代码获得相应的测试数据；

（3） 根据测试数据来详细分析你所用的 x86 机器有**几级 Cache**，**各自容量**是多大？

（4） 根据测试数据来详细分析 **L1 Cache 行**有多少？

3、选做：尝试测量你的 x86 机器 TLB 有多大？

代码 A：

```
#include <sys/time.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    float *a,*b,*c, temp;
    long int i, j, k, size, m;
    struct timeval time1,time2;

    if(argc<2) {
        printf("\n\tUsage:%s <Row of square matrix>\n",argv[0]);
        exit(-1);
    } //if

    size = atoi(argv[1]);
    m = size*size;
    a = (float*)malloc(sizeof(float)*m);
    b = (float*)malloc(sizeof(float)*m);
    c = (float*)malloc(sizeof(float)*m);
```

```
    for(i=0;i<size;i++) {
        for(j=0;j<size;j++) {
            a[i*size+j] = (float)(rand()%1000/100.0);
            b[i*size+j] = (float)(rand()%1000/100.0);
        }
    }

    gettimeofday(&time1,NULL);
    for(i=0;i<size;i++) {
        for(j=0;j<size;j++)  {
            c[i*size+j] = 0;
            for (k=0;k<size;k++)
                c[i*size+j] += a[i*size+k]*b[k*size+j];
        }
    }
    gettimeofday(&time2,NULL);

    time2.tv_sec-=time1.tv_sec;
    time2.tv_usec-=time1.tv_usec;
    if (time2.tv_usec<0L) {
        time2.tv_usec+=1000000L;
        time2.tv_sec-=1;
    }

    printf("Executiontime=%ld.%06ld seconds\n",time2.tv_sec,time2.tv_usec);
        return(0);
}//main
```

四、实验结果及分析

1、分析 Cache 访存模式对系统性能的影响

1.1 分析优化前代码

优化前代码中与矩阵乘法有关的代码段如图 1 所示，可以清晰的看出，该代码段根据矩阵乘法的定义简单实现了矩阵乘法。

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

图 1：优化前矩阵乘法

模拟一下矩阵乘法的过程可以知道（如图 2），矩阵 b 在乘法过程中，是按列遍历矩阵中元素的，这将导致不良好的空间局部性。因为，在 x86 机器的内存中，二维数组是按

行存储的，也就是说，数组同一列间的上下相邻元素在内存中的实际存储距离为$size * w$个字节（w是实际数据类型所占字节数）。于是遍历一列需要跨越的空间较大，不具有良好的空间局部性。

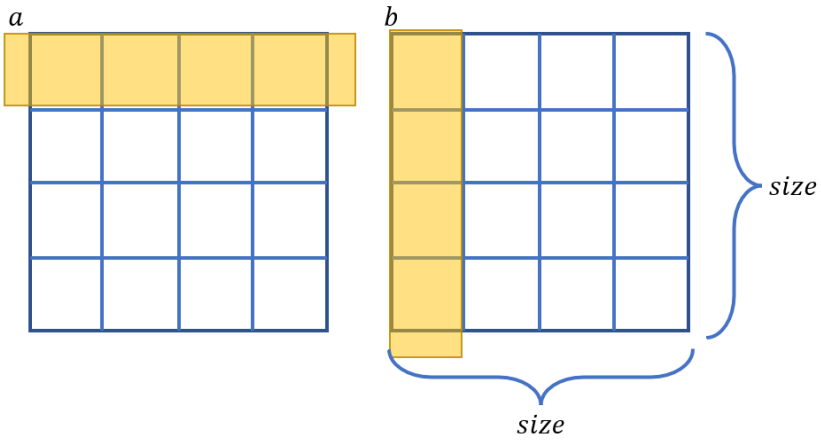

图 2：矩阵乘法过程

由于缓存是以块作为存储单位的，且每一块对应了内存中的一段地址连续的内容，假设每一块对应内存中连续四个字节的内容，如图 3 所示，那么不命中就会访问内存并且一次性取出四个字节的内容。

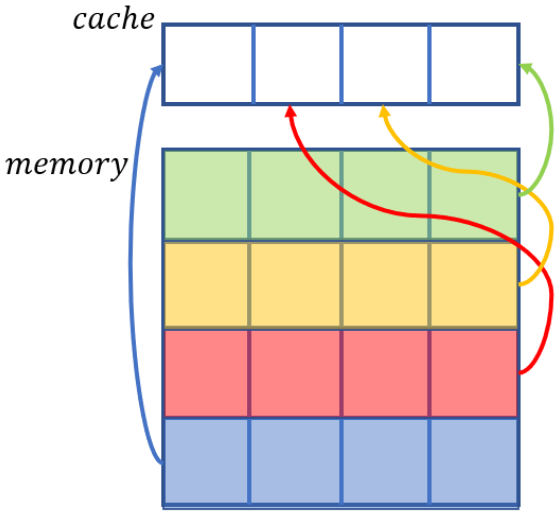

图 3：*cache*缓存机制

现假设矩阵如图 2 所示且每个矩阵元素占一个字节，*cache*结构如图 3 所示，结合矩阵乘法过程，我们来看看*cache*的变化过程。对于矩阵a而言，由于每一次乘法获得结果矩阵元素的操作都是遍历矩阵a的某一行所有元素，且*cache*在不命中时会从内存中取出矩阵a的一整行，于是每一行的遍历的不命中只会在某行的第一个元素产生（在这个例子里），如图 4 所示。访问a第一行的第一个元素会导致*cache*的冷不命中，会使得*cache*找到内存中对应的元素并把该元素对应的块复制到*cache*中，之后，第一行中所有的访问都会在*cache*中找到对应的内容，即命中，如图 5 所示。

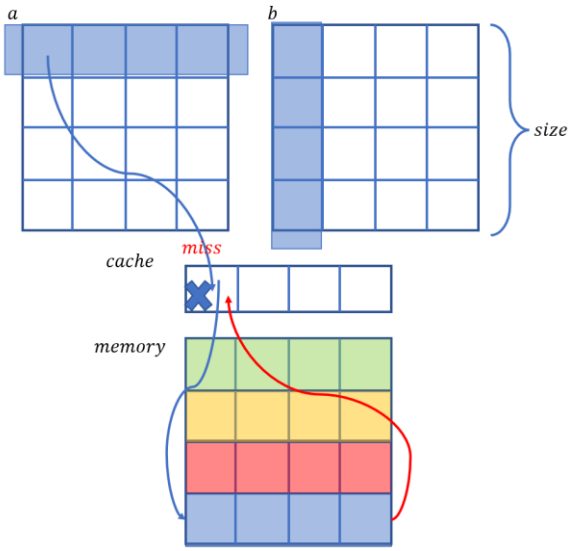

图 4：运行过程中*cache*变化（1）

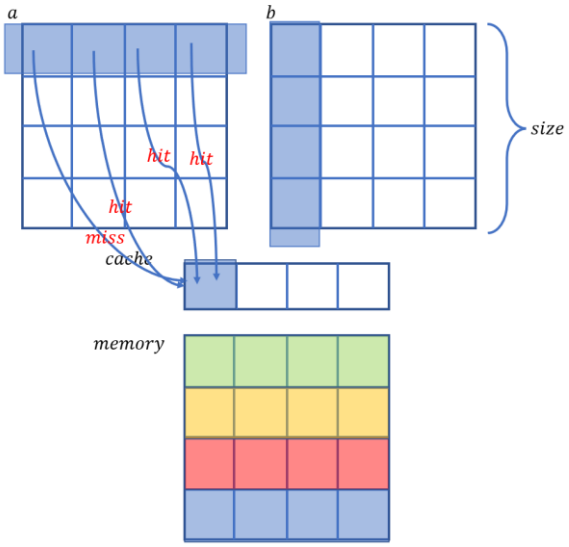

图 5：运行过程中*cache*变化（2）

从上述内容可以得出矩阵*a*的访问具有良好的局部性，于是*cache*的不命中次数就会大大减少，也就是说访问内存的次数会减少，提高了运行效率。再看矩阵*b*的访问，由于是按列访问，于是某一列每一个元素的地址变换较大，容易跳出*cache*中的同一块，如图 6 所示，这样子的访问方式将导致四次*cache*的不命中，故效率较低下。

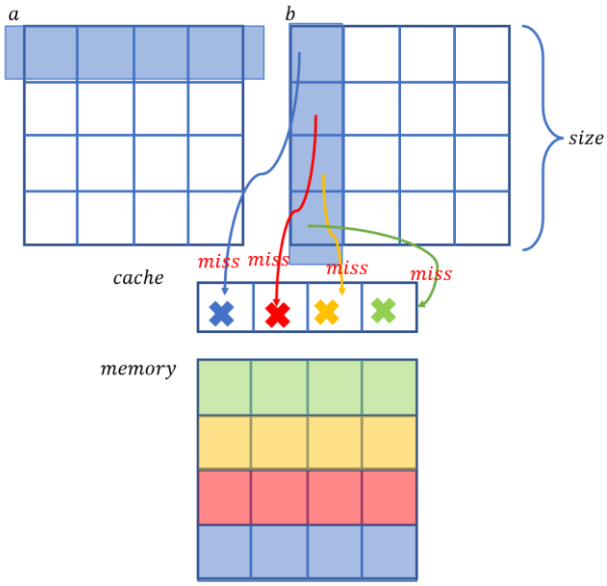

图 6：运行过程中*cache*变化（3）

1.2 第一种优化思路

由于矩阵a按行遍历方式的局部性较好，故想到能否把矩阵b也转化为按行遍历的方式。由于在矩阵乘法中，右边的矩阵每次需要遍历每一列，那我们把它转置之后，每一次就变成遍历某一行了。于是，在生成矩阵b时先把矩阵b转置一下，如图 7 所示。

```
for(i=0;i<size;i++) {
    for(j=0;j<size;j++) {
        a[i*size+j] = (float)(rand()%1000/100.0);
        b[j*size+i] = (float)(rand()%1000/100.0);
    }
}
```

图 7：将矩阵转置

再把矩阵乘法代码段中枚举矩阵b某一列所有元素的代码改为枚举矩阵b某一行的所有元素，如图 8 所示。

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

图 8：第一种优化后的矩阵乘法

1.3 优化前与第一种优化代码的性能比对

表 1：普通矩阵乘法与及第一种优化后矩阵乘法之间的性能对比

矩阵大小	100	500	1000	1500	2000	2500	3000
一般算法执行时间(s)	0.004	0.408	3.249	13.091	36.501	96.310	178.205
优化算法执行时间(s)	0.004	0.401	3.166	10.783	25.200	50.696	85.151
加速比 Speedup	1.065	1.019	1.026	1.214	1.448	1.900	2.093

加速比定义：加速比=优化前系统耗时/优化后系统耗时；
所谓加速比，就是优化前的耗时与优化后耗时的比值。加速比越高，表明优化效果越明显。
结果分析：

随着矩阵规模的增大，加速比也逐渐增大，因为矩阵规模增大的时候，优化前代码的*cache*不命中率会增多，优化之后不命中率大大降低，故加速比增大。同时，由于优化前矩阵b是遍历列的，步长较大，故效率较低。

1.4 第二种优化思路

根据矩阵乘法的特点，我们可以交换优化前代码中内层循环的顺序，如图 9 所示，

```
for(i=0;i<size;i++)
    for(k=0;k<size;k++)
        for (j=0;j<size;j++)
            c[i*size+j] += a[i*size+k]*b[k*size+j];
```

图 9：第二种优化方案

对于目标矩阵每个元素的计算，并没有一次性计算出来，而是拆成多次加和得成，这样由于最内层的循环只有j会变化，故b矩阵的访问的空间局部性良好。

1.5 优化前与第二种优化代码的性能比对

表 2：普通矩阵乘法与及第二种优化后矩阵乘法之间的性能对比

矩阵大小	100	500	1000	1500	2000	2500	3000
一般算法执行时间(s)	0.004	0.408	3.249	13.091	36.501	96.310	178.205
优化算法执行时间(s)	0.004	0.377	2.985	10.190	23.940	46.839	80.711
加速比 Speedup	0.998	1.082	1.088	1.285	1.525	2.056	2.208

加速比定义：加速比=优化前系统耗时/优化后系统耗时；
所谓加速比，就是优化前的耗时与优化后耗时的比值。加速比越高，表明优化效果越明显。
结果分析：

可以看出，这种优化方法与第一种优化方法一样有效，原理与第一种方法差不多。

2、测量分析出 Cache 的层次结构、容量以及 L1 Cache 行有多少？

2.1 实验原理

通过测量存储系统在不同大小的数据与不同读取字长下读取数据的速率（称为读吞吐量），画出存储器山，反映$cache$的层次结构。如果一个程序在s秒的时间段内读n个字节，那么这段时间内的读吞吐量为n/s，通常以兆字节每秒（MB/s）为单位。

于是我们需要编写代码，通过指定步长和数据大小，通过一个循环发出一系列读请求，这样测量出的读吞吐量就能让我们看到对于这个读序列来说的存储系统的性能。个人的理解是，不同的存储器层级速度不同，在固定数据大小时，不同的步长对应的不同的读吞吐量可以帮助我们区分不同的存储器层级

2.2 测量方案及代码

代码老师已给出，现简单分析一下这段代码，代码中包括两个主要函数，test()函数（如图 10 所示）和 run()函数（如图 11 所示）。

```
void test(int elems, int stride) /* The test function */
{
    int i;
    double result = 0.0;
    volatile double sink;

    for (i = 0; i < elems; i += stride) {
    result += data[i];
    }
    sink = result; /* So compiler doesn't optimize away the loop */
}
```

图 10：代码分析（1）

test()函数通过指定的步长 stride，在指定的数组 data 中扫描 elems 个元素，并对元素进行累加。

```
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(double);

    test(elems, stride);                    /* warm up the cache */       //line:mem:warmup
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */ //line:mem:fcyc
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */  //line:mem:bwcompute
}
/* $end mountainfuns */
```

图 11：代码分析（2）

run()函数中调用了 test 函数并返回测量得到的吞吐量，其中还调用了一个外部函数 fcyc2()，该函数以参数 elems 调用 test 函数，并估计 test 函数的运行时间（以 CPU 周期为单位）。该函数的参数 size 和 stride 允许我们控制产生的读序列的时间和空间局部性程度。size 值越小，得到的数据集越小，时间局部性越好;stride 值越小，访问数组元素的步长越小，空间局部性越好。

2.3 测试结果

在本地上运行上述代码，将会输出一个表示读吞吐量与步长和数据集大小关系的二维表格，将该表格在三维空间中画出来，得到的图如图 12 所示。

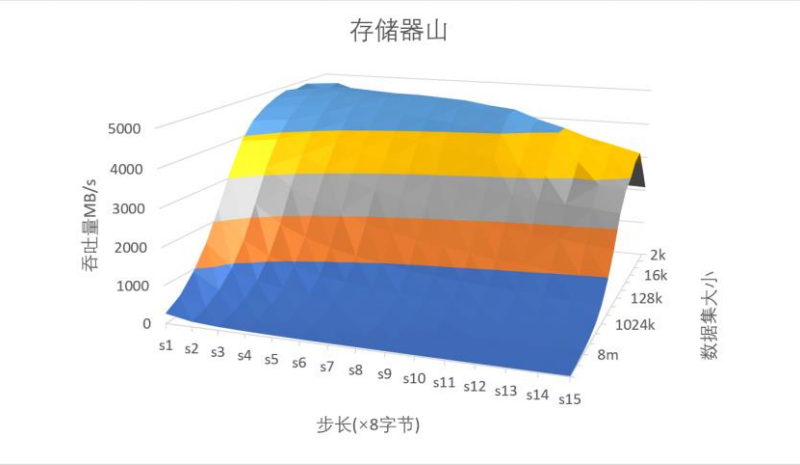

图 12：存储器山

2.4 分析过程

固定步长为 8，观察读吞吐量随着数据集大小的变化趋势，如图 13 所示。

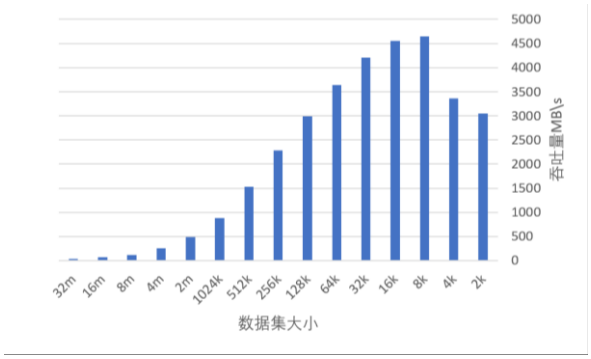

图 13：读吞吐量随数据集大小的变化

由于高速缓存的大小和时间局部性对读吞吐量有影响，可以看到，各个*cache*层次的划分如图 14 所示。大小最大为 32KB 的数据集可以放入 L1 缓存中，故读都是由 L1 缓存完成的，吞吐量保持于峰值；大小最大为 256KB 的数据集可以放入 L2 缓存中；最大为 4m 的数据集可以放入 L3 缓存中；其他更大的数据集的读则由主存来完成。

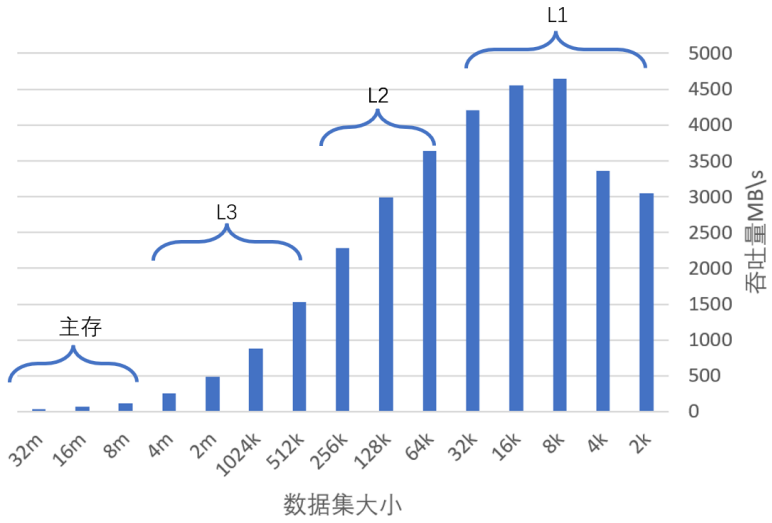

图 14：各层缓存容量

之后，固定数据集大小为 64KB，探讨读吞吐量随着数据集大小的变化趋势，如图 15 所示。

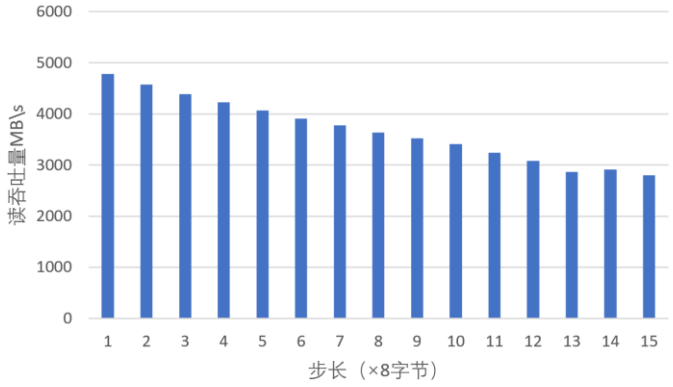

图 15：读吞吐量随步长大小的变化

随着步长从 1 个字增长到 12 个字，读吞吐量平稳地下降。因为在山的这个区域中，L1 中的读不命中会导致一个块从 L2 传送到 L1。后面在 L1 中这个块会有一定数量的命中，这个数量是取决于步长的。随着步长的增加，不命中率也会提高，吞吐量就下降了。当步长达到 12 个字之后，吞吐率趋于平缓，这是因为步长已经达到了这个系统 L1 的块的大小，于是每一个读请求在 L1 中都会不命中，必须经过 L2，故吞吐量是一个常数速率。于是 L1 的块长为 12 个字也就是 96 个字节。

五、实验结论与心得体会

实验结论：

对给出的矩阵乘法代码进行优化，提高了其性能，并进行了相关测试与结果分析。

对老师提供的计算吞吐量代码进行分析，运行代码获得了相应的测试数据并绘制存储山图。

对测试数据进行详细分析，并得出我使用的 x86 机器由 3 级别 Cache，容量分别为：

L1：32KB; L2：256KB;L3：4M

对测试数据详细分析，并得出 L1 Cache 行有 96 个字节。

心得体会：

经过本次实验，对时间局部性与空间局部性概念的理解更加深刻，学会使用空间局部性分析代码并对代码进行优化。通过测试对 Cache 的层次概念更加了解，学会分析 Cache 各层次的容量与 L1 Cache 行的大小。

指导教师批阅意见：   成绩评定：   指导教师签字：　冯禹洪 2022 年　　月　日
备注：