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：刘　　刚

报告人：林宪亮　学号：2022150130　班级：国际班

实验时间：2024 年 6 月 10 日至 6 月 11 日

实验报告提交时间：2024 年 6 月 11 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？

4、尝试测量你的 x86 机器 TLB 有多大？（选做）

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
```

1、分析 Cache 访存模式对系统性能的影响

（1）代码优化：

原始代码 A 的部分代码：

```
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];
        }
    }
```

我对这个部分进行了优化，这里利用 Cache 的局部性原理进行优化，我先分析代码 A 的这个代码块的计算过程。

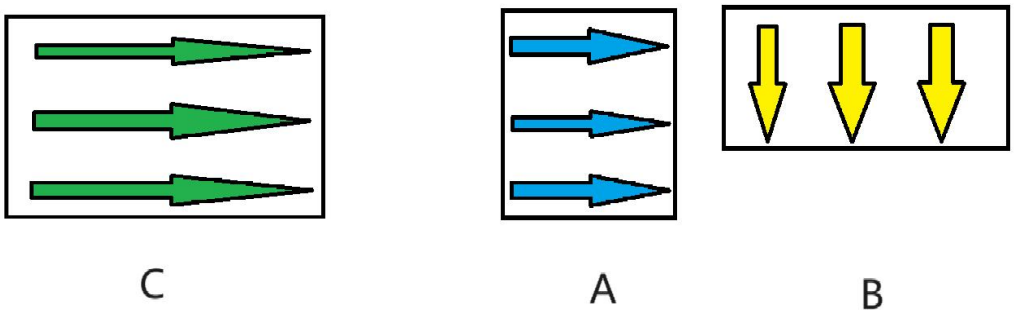

图 1 原始代码计算过程

如图为代码 A 中三个矩阵的访问方式。其中，矩阵 A 和矩阵 C 的访问模式都可以很好地利用 Cache 的局部性，因为它们是按行优先的方式进行访问。这意味着数据在内存中的存储顺序与访问顺序一致，能够有效地提升 Cache 命中率，从而提高程序执行效率。

然而，对于矩阵 B，它的访问模式是按列优先进行的。这样的访问方式并不能很好地利用 Cache 的局部性，因为列优先访问导致频繁跨行读取数据，Cache 命中率较低，绝大部分数据需要从主存中读取。这种情况下，Cache 的优势无法发挥，导致整体运行效率大幅下降。

为了优化矩阵 B 的访问效率，可以对其进行转置。通过转置矩阵 B，在进行矩阵乘法时，我们可以将原来的列优先访问转变为行优先访问。这样一来，矩阵 B 的访问模式就与矩阵 A 和矩阵 C 一致，充分利用 Cache 的局部性。具体来说，转置后的矩阵 B 在内存中的存储顺序将与访问顺序匹配，显著提高 Cache 的命中率，减少对主存的依赖，从而提升整体计算效率。

优化后的核心代码如下：

```
//矩阵转置
for(int i=0;i<size;i++){
   for(int j=0;j<size;j++){
          b[i*size+j] = c[j*size+i];
   }
}
//矩阵相乘
for(int i=0;i<size;i++){
       for(int j=0;j<size;j++){
           c[i*size+j] = 0;
           for(int k=0;k<size;k++)
              c[i*size+j] +=a[i*size+k]*b[j*size＋k];
       }
}
```

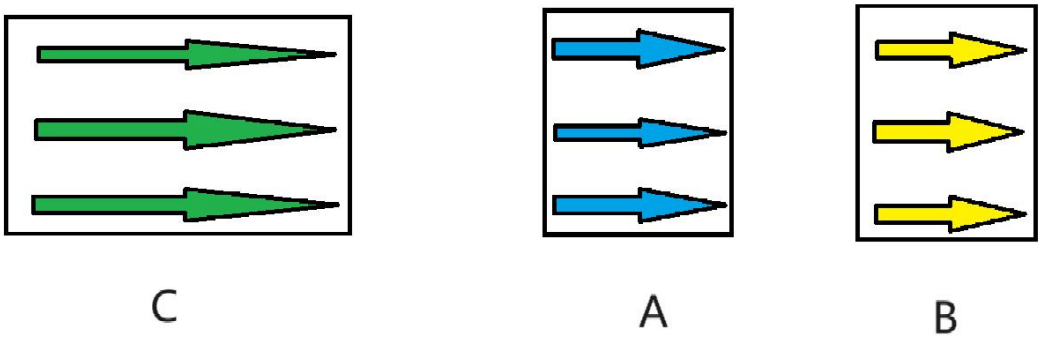

图 2 优化代码计算过程

如图，A，B，C 三个矩阵均是按照行优先的顺序来访问，充分利用了 Cache 的局部性，提高了算法的效率。

不同规模矩阵下的测试结果与比较分析见四、实验结果及分析。

四、实验结果及分析

1、分析 Cache 访存模式对系统性能的影响

表 1、普通矩阵乘法与及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	0.004272	0.633982	5.273233	16.312314	36.312311	75.314124	125.974385
优化算法执行时间	0.004894	0.684122	5.313114	16.123531	34.823113	65.723812	103.948512
加速比 speedup	0.8729	0.9266	0.9924	1.012	1.042	1.1459	1.2147

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。

分析原因：

首先，我们可以分析一般规律：当矩阵大小增加时，加速比通常会呈上升趋势。这是因为随着矩阵规模的增大，优化算法在减少内存访问延迟和提高 Cache 利用率方面的优势愈加明显。当矩阵大小达到一定程度时，优化算法的耗时显著低于未优化算法的耗时。

然而，当矩阵较小时，由于没有达到 Cache 的容量限制，优化算法的优势并不显著。尤其是由于矩阵转置本身也需要时间，这可能导致优化算法的总执行时间反而比未优化算法更长。在这种情况下，优化效果并不明显，加速比的提升也较为有限。

特别是在矩阵大小从 2500 增大到 3000 时，我们观察到加速比的增长趋势并不明显。对此，我的分析是这可能与 Cache 的大小有关。具体来说，当前的矩阵大小可能相对较小，尚未充分利用优化算法在大矩阵上的优势。Cache 的大小限制了优化效果的体现，当矩阵规模

较小时，内存访问并未达到 Cache 的瓶颈，优化算法无法显著减少内存访问延迟。

于是我增大了矩阵大小进行实验，结果如下：

表 2、普通矩阵乘法与及优化后矩阵乘法之间的性能对比（更大规模）

矩阵大小	3500	4000	4500
一般算法执行时间	235.124112	329.812457	499.523463
优化算法执行时间	174.125722	240.153423	354.156234
加速比 speedup	1.3503	1.3733	1.4104

可以看到当矩阵大小增大时，优化效果会更加明显，加速比呈增长的趋势。

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

（1） 实验原理；

高速缓存器（Cache）通常分为 L1、L2 和 L3 三个层次。层次越低，访问周期越短，读取数据的效率就越高。在这个实验中，我将设计一个方案，通过测量数据读取时间来分析层次结构、容量以及 L1 Cache 行有多少。

首先，我们知道从 L1 到 L3，Cache 的大小逐渐增大。因此，我可以通过不断扩大数据规模，测量不同数据规模下的读取时间来推断各层 Cache 的容量。基于 Cache 的局部性原理，我将通过设计一系列实验来测量不同数据规模下的读取时间，从而确定各层 Cache 的大小和 L1 cache line。

Intel Core i7 高速缓存层次结构如下图所示：

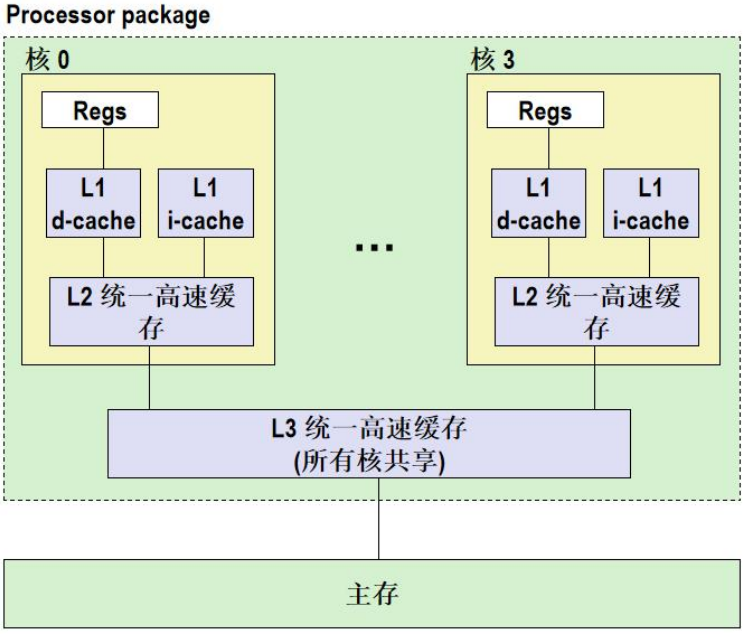

图 3 Intel Core i7 高速缓存层次结构

已知参数如下：

L1 i-cache 和 d-cache：32 KB，8 路组相联，访问时间：4 个周期

L2 统一高速缓存：256 KB，8 路组相联，访问时间：10 个周期

L3 统一高速缓存：8 MB，16 路组相联，访问时间：40-75 个周期

（2） 测量方案及代码；

• Cache 的层次结构、容量

我们要通过实验测量 Cache 层次结构及其大小，利用 Cache 在不同层级上的数据读取效率不同以及本身的容量差异来实现。具体来说，当数组的大小超过某一层级的 Cache 容量时，Cache miss 会增加，导致访问时间显著增大。通过测量数组访问时间，我们可以确定不同 Cache 层级的容量。

通过一个数组来控制其大小进行测量。当数组大小超出某一层级时，Cache miss 增加，时间显著增大。具体的测量代码如下：

```
void test_cache(int size) {
    int n = size / sizeof(int);  // 一个 int 占 4 个字节
    int* array = new int[n];

    // 初始化数组
    for(int i = 0; i < n; i++) {
        array[i] = 1;
    }

    uniform_int_distribution<> dis(0, n - 1);
    int test_times = 1000000 * 10;  // 循环次数，保证测量时间足够长

    vector<int> random_index;
    random_device rd;
    mt19937 gen(rd());

    // 生成随机索引
    for(int i = 0; i < test_times; i++) {
        int index = dis(gen);
        random_index.push_back(index);
    }

    int sum = 0;
    int begin = clock();  // 记录开始时间

    // 进行数据访问
```

```
        for(int i = 0; i < test_times; i++) {
            sum += array[random_index[i]];
        }

        int end = clock();  // 记录结束时间

        if(size >= 1024 * 1024) {
            cout << size / (1024 * 1024) << "MB ";
        } else {
            cout << size / 1024 << "KB ";
        }

        cout << "Time: " << end - begin << "ms" << endl;

        delete[] array;
    }

    // 进行多次测量
        int sizes[] = {32 * 1024, 256 * 1024, 512 * 1024, 1 * 1024 * 1024, 2 * 1024
* 1024, 4 * 1024 * 1024, 8 * 1024 * 1024, 16 * 1024 * 1024};
        for(int size : sizes) {
            test_cache(size);
        }
```

代码说明：

数组初始化：创建一个大小为 size / sizeof(int) 的数组，并初始化每个元素为 1。

随机索引生成：使用 uniform_int_distribution 生成范围在 0 到 n-1 的随机数，并存储在 random_index 向量中，以确保访问内存位置的随机性。

时间测量：使用 clock() 函数记录开始和结束时间，通过 sum += array[random_index[i]] 来模拟大量内存访问。循环次数设为一个较大的常数 test_times，以保证测量时间不为 0，并且精度较高。

输出结果：根据数组大小以 KB 或 MB 为单位输出测量结果。通过观察时间的显著变化点来确定不同的 Cache 层级。

• L1 Cache 行：

为了测量 L1 Cache line 的大小，我们可以利用 Cache 的局部性原理。Cache line 是 Cache 中的最小传输单位，每次从主存中读取的数据块就是一个 Cache line 的大小。因此，在读取数组数据时，通过跳着读（即每次跳过一个固定的间隔 gap），我们可以测量访问时

间的变化，从而确定 Cache line 的大小。

下面是测量 Cache line 大小的具体代码：

```
void test_line(int* array, int gap, int size) {
    int begin = clock();
    int n = size / sizeof(int);
    int sum = 0;
    for (int j = 0; j < gap; j++) {
        for (int i = 0; i < n; i += gap) {
            sum += array[i];
        }
    }
    int end = clock();
    cout << gap * 4 << "B " << end - begin << "ms" << endl;
}


    const int size = 1024 * 1024;  // 1MB
    int* array = new int[size / sizeof(int)];

    // 初始化数组
    for (int i = 0; i < size / sizeof(int); i++) {
        array[i] = 1;
    }

    // 测量不同 gap 下的时间
    for (int gap = 1; gap <= 128; gap *= 2) {
        test_line(array, gap, size);
    }
```

代码说明：

数组初始化：

创建一个大小为 size / sizeof(int) 的数组，并初始化每个元素为 1。

测量函数：

test_line(int* array, int gap, int size)：使用 clock() 函数记录开始和结束时间。

gap 表示每次跳过的元素个数，因为我们使用的是 int 类型，每个 int 占 4 个字节，因此 gap 表示跳过的字节数乘以 4。

通过嵌套循环实现跳着读，每次读取数组时跳过 gap 个元素。

主函数：

初始化一个 1MB 大小的数组。

通过循环调用 test_line 函数，测量不同 gap 下的访问时间。

原理解释：

当 gap 小于 Cache line 的大小时，访问 array[i] 后，接下来的数据可能还在同一个 Cache line 中，读取速度较快。

当 gap 等于或超过 Cache line 的大小时，访问 array[i] 后，接下来的数据不在同一个 Cache line 中，导致 Cache miss 频率增加，访问时间显著增加。

通过记录不同 gap 下的访问时间，找到时间急剧上升的临界点。这个临界点的 gap 乘以 4 就是 Cache line 的大小（单位：字节）。

（3） 测试结果；

运行上述所展示的代码，所得到的实验结果如下表所示：

表 3、不同规模下的时间（ms）

大小	8KB	64KB	128KB	512KB	1MB	2MB	4MB
时间	22	21	23	31	39	44	47
大小	6MB	8MB	12MB	16MB	32MB	64MB	
时间	49	52	62	133	236	247	

这是测量 cache 层级的结果。

表 4、不同规模下的时间（ms）

大小	4B	8B	16B	32B	64B	128B	256B
时间	102	107	118	156	276	507	522

这是测量 cache line 的结果。

（4） 分析过程；

根据所得的结果将数据绘制成曲线：

• Cache 层级测量曲线：

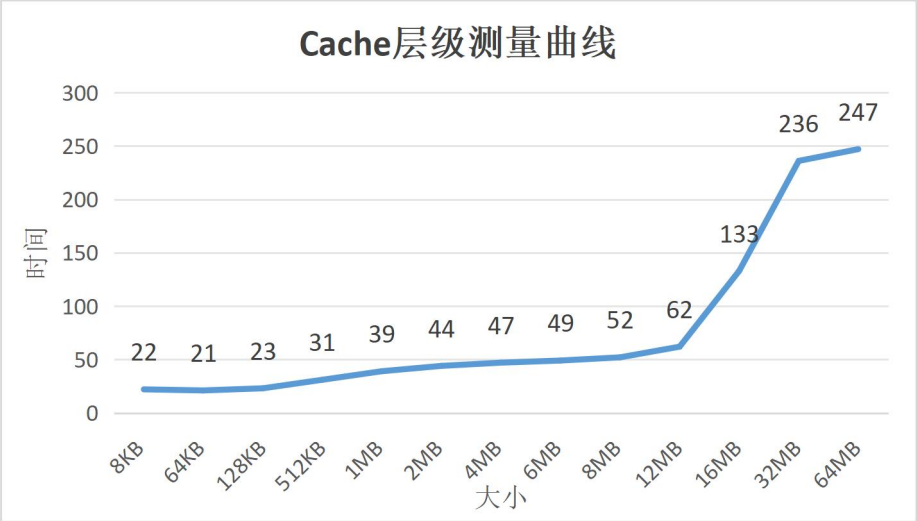

图 4 Cache 层级测量曲线

在进行 Cache 层级及其大小的测量时，我们通过实验数据观察到了不同层级 Cache 的变化点。尽管矩阵大小并未在横坐标上连续显示，我只标注了一些关键的数据点，这些点对应于显著的性能变化。

根据实验结果，Cache 访问时间的变化趋势如下：

L1 Cache：在 16KB 到 512KB 之间，访问时间变化不大，表明这一范围内的数据访问主要命中 L1 Cache。

L2 Cache：在 512KB 到 4MB 之间，访问时间开始显著增加，表明这一范围的数据访问主要命中 L2 Cache。

L3 Cache：在 4MB 到 16MB 之间，访问时间进一步增加，表明这一范围的数据访问主要命中 L3 Cache。

所以测得各层 Cache 的大小如下：

L1 Cache：512KB　L2 Cache：4MB　L3 Cache：16MB

• Cache line 的测量曲线

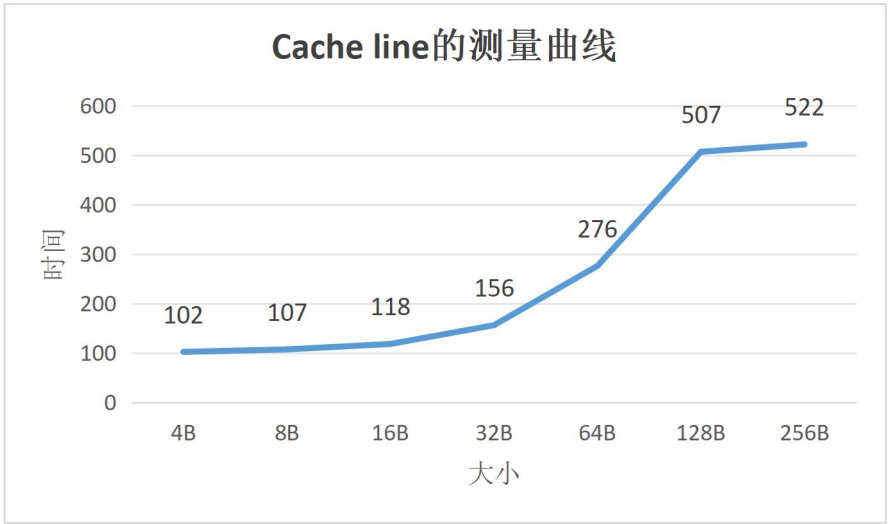

图 5 Cache line 的测量曲线

这里可以看出，从 64B 开始，时间急剧上升，因此 64B 即为我们所找的临界点。所以 L1 cache line 大小为 64B。

（5） 验证实验结果

打开任务管理器，可以看到各个 cache 层级的大小，与我的实验得到的结果一致。

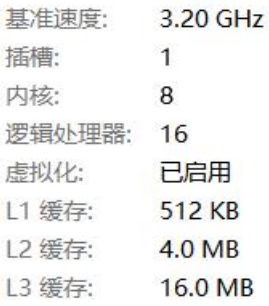

图 6 cache 层级大小

通过命令行操作也可以查看 Cache line 的大小，为 64B，与实验结果一致。

3、尝试测量你的 x86 机器 TLB 有多大？（选做）

为了测量 x86 机器上的翻译后备缓冲区（TLB）的大小，我们可以通过访问不同大小的数据集合，并观察访问时间的变化来确定。TLB 是一种用于加速虚拟地址到物理地址转换的缓存，每当内存访问发生时，CPU 会首先检查 TLB 是否包含所需的地址映射。如果找到则称为 TLB 命中，否则发生 TLB miss，系统需要从页表中查找并加载映射到 TLB 中。

代码如下：

```
void test_tlb(int* array, int size, int stride) {
    int sum = 0;
    int n = size / sizeof(int);
    int iterations = 1000000;

    int begin = clock();

    for (int i = 0; i < iterations; i++) {
        for (int j = 0; j < n; j += stride) {
            sum += array[j];
        }
    }

    int end = clock();
    cout << "Size: " << size / 1024 << " KB, Stride: " << stride * 4 << " B, Time:
" << end - begin << " ms" << endl;
}

int main() {
    const int max_size = 128 * 1024 * 1024;  // 128 MB
```

```
    int* array = new int[max_size / sizeof(int)];

    // 初始化数组
    for (int i = 0; i < max_size / sizeof(int); i++) {
        array[i] = 1;
    }

    // 测量不同大小和步长下的时间
    for (int size = 4 * 1024; size <= max_size; size *= 2) {
        for (int stride = 1; stride <= 2048; stride *= 2) {
            test_tlb(array, size, stride);
        }
    }

    delete[] array;
    return 0;
}
```

代码说明：

数组初始化：创建一个大小为 128MB 的数组，并初始化每个元素为 1。

测量函数：test_tlb(int* array, int size, int stride)：stride 表示每次跳过的元素个数，每个 int 占 4 个字节。通过嵌套循环进行数据访问，内层循环按 stride 访问数组，外层循环控制总的访问次数。使用 clock() 函数记录开始和结束时间，计算访问时间。

主函数：通过循环调用 test_tlb 函数，测量不同大小和步长下的访问时间。

测试结果：

Size: 4 KB, Stride: 4 B, Time: 994 ms

Size: 8 KB, Stride: 4 B, Time: 1453 ms

Size: 16 KB, Stride: 4 B, Time: 2055 ms

Size: 32 KB, Stride: 4 B, Time: 3468 ms

Size: 64 KB, Stride: 4 B, Time: 5593 ms

Size: 128 KB, Stride: 4 B, Time: 8166 ms

Size: 256 KB, Stride: 4 B, Time: 14034 ms

Size: 512 KB, Stride: 4 B, Time: 44606 ms

根据测试数据，可以看出 size 从 256kb 到 512kb 的时候，时间会有一个突变，急剧上升，所以可以推测 TLB 的大小约为 512KB。

五、实验结论

通过本次实验，我了解了 Cache 访存模式对系统性能的影响，并探究 CPU Cache 的层级结构以及 Cache line 的大小对系统性能的影响。
我通过对普通矩阵乘法和优化后矩阵乘法的性能对比来分析 Cache 访存模式对系统性能的影响。同时，我们设计了实验方案和代码，通过测量数据访问时间来确定 CPU Cache 的层级结构和 Cache line 的大小。

Cache 访存模式对系统性能的影响：
• 随着矩阵大小增加，优化算法的加速比逐渐增加，优化效果更为明显。
•在小规模矩阵计算时，优化效果并不明显，这主要是由于转置等操作的开销和未达到 Cache 容量限制所致。
CPU Cache 的层级结构和 Cache line 大小：
• 经过实验测量，确定了 CPU Cache 的层级结构和大小：L1 Cache 为 512KB，L2 Cache 为 4MB，L3 Cache 为 16MB。
• 确定了 L1 Cache line 大小为 64B。

本次实验帮助我更好地理解了 Cache 对系统性能的重要性，以及如何通过优化算法和合理利用 Cache 来提高程序执行效率。同时，实验结果也验证了课堂所学的知识，并对未来的学习和实践具有重要意义。

指导教师批阅意见：

成绩评定：

指导教师签字：

2024 年　　月　日

备注：