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：罗　　胜

报告人：詹耿羽　学号：2023193026　班级：　数计

实验时间：2025 年　6　月　1 日至　6 月　20　日

实验报告提交时间：2025 年　6 月　13　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？

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
```

实验内容

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。

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
```

下面的代码实现了矩阵 a[]与矩阵 b[]相乘，结果存在矩阵 c[]中，与上述代码相同:

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

代码 A 对矩阵乘法的实现是：遍历矩阵的每一行和每一列，求出结果矩阵对应位置的元素．在空间局部性上，a[]每次访问的步长为 1，空间局部性良好；b[]每次访问的步长为 size，size 较大时空间局部性较差，访问耗时长，

优化： 优化 b[]访问的空间局部性，使其每次访问的步长为 1. 具体地，遍历 a[]的每个元素，将每个元素的贡献累加到 c[]中，代码如下，注意清空 c[]:

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

(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

（5） 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。

```
int test(int elems, int stride) {
    long i, sx2 = stride * 2, sx3 = stride * 3, sx4 = stride * 4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems;
    long limit = length - sx4;

    /*Combine 4 elements at a time*/
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i + stride];
        acc2 = acc2 + data[i + sx2];
        acc3 = acc3 + data[i + sx3];
    }

    /*Finish any remaining elements*/
    for (; i < length; i += stride)
        acc0 = acc0 + data[i];
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

上述代码为教材中的参考代码，其中的 test()函数模拟计算机访问内存的过程.

用 elems * size(long) / stride 可得 test()函数访问的内存空间的大小，则只需记录调用

test()函数的过程中消耗的时间即可.

为精确测量时间, 将测量的精度调整到时间周期的级别, 即记录 test()函数的调用过程花费的时钟周期, 用时钟周期 / 电脑频率得到程序的运行时间.

```
/* run - Run test(elems, stride) and return read throughput
*             "size" is in bytes, "stride" is in array elements,
*             and Mhz is CPU clock frequency in Mhz.
*/
double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(long);

    test(elems, stride);    // 预热缓存
    cycles = fcy2(test, elems, stride, 0);    // Call test(elems, stride)
    return (size / stride) / (cycles / Mhz);    // Convert cycles to MB/s
}
```

```
cycles = fcy2(test, elems, stride, 0);    // Call test(elems, stride)
```

上述代码中的 fcy2()函数是通过在 C 语言中内嵌汇编实现的记录 test()函数的调用过程花费的时钟周期的函数, 通过汇编的接口记录开始运行时的时间戳, 运行完后再记录时间戳, 两时间戳相减即得运行的时间周期.

记录时间戳的函数:

(1) access_counter()函数通过 C 语言中嵌入汇编来获取当前程序运行到现在的时钟周期的时间戳, 结果保存在 hi 和 lo 两元素中. 汇编内部通过 rdtsc 命令实现, 返回当前程序运行到现在的时间周期, 将时间周期的高位保存到 edx 寄存器, 低位保存到 eax 寄存器.

```
void access_counter(unsigned *hi, unsigned *lo) {
    /* Get cycle counter */
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
        : "=r"(*hi), "=r"(*lo)
        : /* No input */
        : "%edx", "%eax");
}
```

(2) start_counter()函数返回当前的时间戳, 对应开始时的时间戳. get_counter()函数得到当前的时间戳并作差, 得到花费的时间周期.

```
void start_counter() {
    access_counter(&cyc_hi, &cyc_lo);
}
```

```
double get_counter() {
    unsigned ncyc_hi, ncyc_lo;
    unsigned hi, lo, borrow;
    double result;
    /* Get cycle counter */
    access_counter(&ncyc_hi, &ncyc_lo);
    /* Do double precision subtraction */
    lo = ncyc_lo - cyc_lo;
    borrow = lo > ncyc_lo;
    hi = ncyc_hi - cyc_hi - borrow;
    result = (double)hi * (1 << 30) * 4 + lo;
    if (result < 0) {
        fprintf(stderr, "Error: Cycle counter returning negative value: %.0f\n", result);
    }
    return result;
}
```

(3) 为防止单次测量不精确，采用多次测量取最小值作为运行的时钟周期. 具体地，每次循环前先清理缓存，再调用 test()函数进行缓存热身，再开始正式测量，将测量的结果放到一个数组中进行处理，该过程在 add_sample 中实现. 最后判断结果数组中的记录结果是否在超过测试次数前足够稳定，若是则返回对应结果. 判断是否稳定的标准：多次测量，取最小的 5 个值，若 5 个值的最大值与最小值之比小于一个常数，就认为结果已足够稳定；否则认为结果不够稳定，重新测量.

```
do {
        double cyc;
        if (clear_cache)
            clear();
        start_comp_counter_tod();
        f(param1, param2);
        cyc = get_comp_counter_tod();
        add_sample(cyc, k);
    } while (!has_converged(k, epsilon, maxsamples) && samplecount < maxsamples);
```

(4) 得到时钟周期后还需将初始周期转化为对应的运行时间，这可通过 CPU 的时钟周期频率计算. 在 linux 系统中，可在/proc/cpuinfo 文件中得到，通过字符串处理， 得到 cpu 频率的关键字后返回即可，代码如下:

```
static char buf[MAXBUF];
    FILE *fp = fopen("/procuinfo", "r");
    cpu_ghz = 0.0;
```

```
    if (!fp) {
        fprintf(stderr, "Can't open /procuinfo to get clock information\n");
        cpu_ghz = 1.0;
        return cpu_ghz * 1000.0;
    }
    while (fgets(buf, MAXBUF, fp)) {
        if (strstr(buf, "cpu MHz")) {
            double cpu_mhz = 0.0;
            sscanf(buf, "cpu MHz\t: %lf", &cpu_mhz);
            cpu_ghz = cpu_mhz / 1000.0;
            break;
        }
    }
```

(5) 根据上述信息可得运行 test()函数所需的时间，进而得到计算机吞吐量和 memory mountain.

（6） 运行代码获得相应的测试数据.

将运行结果输出到文本，用 excel 的三维图标可视化.

（7） 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.

（8） 根据测试数据来详细分析 L1 Cache 行有多少.

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	**0.004**	**0.507**	**4.743**	**19.199**	**37.134**	**112.479**	**185.441**
优化算法执行时间	**0.003**	**0.355**	**2.839**	**9.604**	**17.712**	**44.429**	**76.510**
加速比 speedup	**1.333**	**1.428**	**1.671**	**1.999**	**2.097**	**2.532**	**2.424**

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：

(1) 上述结果用图表表示如下:

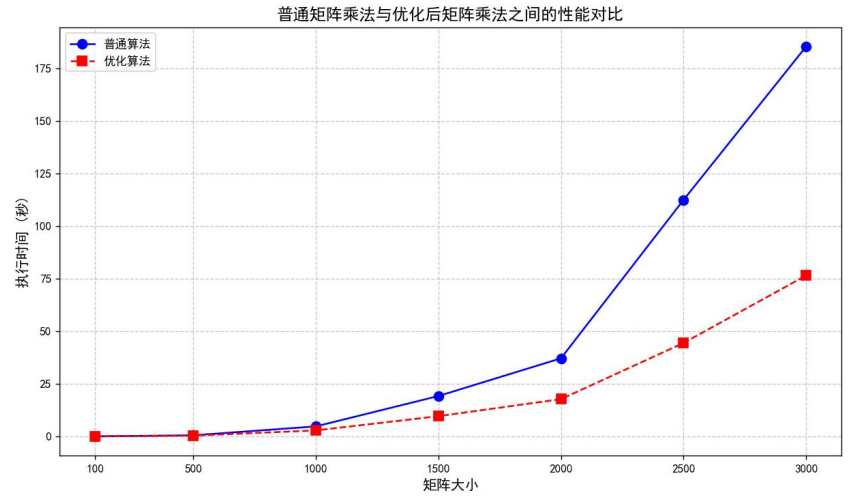

图 1.1: 一般算法与优化算法的执行时间对比

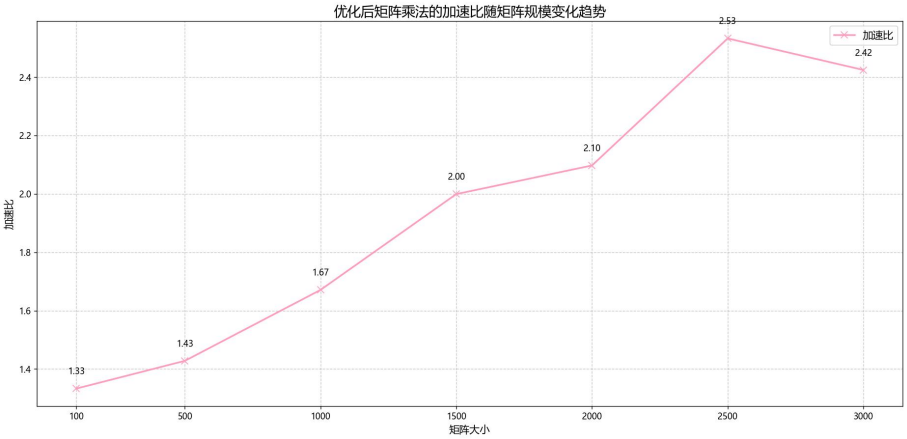

图 1.2: 加速比随矩阵大小的变化关系

(2) 分析原因：

① 在本研究中，针对所有实验数据规模的分析表明，相较于传统算法，优化算法在时间成本上显著降低，展现出更高的运行效率。这一结果揭示了优化算法在改善算法空间局部性方面的有效性，并进一步证实了空间局部性优化对于提升算法运行效率的积极作用。

② 加速比随数据规模的增大而增大, 因为数据规模较大, .即 size 增大时, 一般算法访问 b[]的步长增大, 空间局部性的影响因一般算法是 $O(n^3)$的时间复杂度而被放大.

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:

参考书本，根据局部性原理，可以知道，读吞吐量可以体现读取某部分内存空间时存储系统的性能。根据之前所学过的知识，我们知道，Cache 一般有三级，在读取速度上，L1>L2>L3>主存，当数据存放在不同位置的时候，数据的读取时间是会发现变化的，而且速度差距也是比较大的，即读吞吐量会发生阶跃性的改变，并且对于 L1,L2,L3 和主存来说读

吞吐量是逐渐变小的。因此我们可以对不同数据集进行不断读写操作，然后测量程序的读吞吐量，当读吞吐量发生显著变化的时候，即可以推测 Cache 的层次结构以及容量的大致区间。

(2) 测量方案及代码:

使用课本所给代码，通过调用 run 程序，通过传入的不同参数运行程序，分别查看不同 size 的读吞吐量以及 stride 的读吞吐量的对应关系，记录 size 与读吞吐量对应的关系和 stride 的读吞吐量的相应关系，分析两个关系，从而推测 Cache 的层次结构的容量和 L1 Cache 行的大小。

code/mem/mountain/mountain.c

```
1  long data[MAXELEMS];      /* The global array we'll be traversing */
2
3  /* test - Iterate over first "elems" elements of array "data" with
4   *        stride of "stride", using 4 x 4 loop unrolling.
5   */
6  int test(int elems, int stride)
7  {
8      long i, sx2 = stride*2, sx3 = stride*3, sx4 = stride*4;
9      long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
10     long length = elems;
11     long limit = length - sx4;
12
13     /* Combine 4 elements at a time */
14     for (i = 0; i < limit; i += sx4) {
15         acc0 = acc0 + data[i];
16         acc1 = acc1 + data[i+stride];
17         acc2 = acc2 + data[i+sx2];
18         acc3 = acc3 + data[i+sx3];
19     }
20
21     /* Finish any remaining elements */
22     for (; i < length; i+=stride) {
23         acc0 = acc0 + data[i];
24     }
25     return ((acc0 + acc1) + (acc2 + acc3));
26 }
27
28 /* run - Run test(elems, stride) and return read throughput (MB/s).
29  *       "size" is in bytes, "stride" is in array elements, and Mhz is
30  *       CPU clock frequency in Mhz.
31  */
32 double run(int size, int stride, double Mhz)
33 {
34     double cycles;
35     int elems = size / sizeof(double);
36
37     test(elems, stride);                    /* Warm up the cache */
38     cycles = fcyc2(test, elems, stride, 0);  /* Call test(elems,stride) */
39     return (size / stride) / (cycles / Mhz); /* Convert cycles to MB/s */
40 }
```

code/mem/mountain/mountain.c

图 2.1: 课本的参考代码

(3) 测试结果:

① 程序的测试内存取 16 kB 到 128 MB, 步长取 1 到 15, 得到下表所示的数据:

Clock frequency is approx. 3688.9 MHz															
Memory mountain (MB/sec)															
	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15
128m	14200	7460	5003	3810	3143	2621	1952	1769	1552	1420	1329	1326	1282	1213	1171
64m	13382	7324	4987	3876	3126	2693	2374	2107	1951	1516	1385	1324	1236	1181	1180
32m	14393	7815	5232	4107	3321	2671	2256	1836	1751	1638	1552	1455	1394	1169	1158
16m	14636	8328	5863	4488	3812	3141	2732	2523	2404	2472	1563	1614	1650	1597	1435
8m	15213	12934	10469	7465	5345	4720	4181	3700	3859	3873	3862	3843	3981	4123	4084
4m	21938	16722	13669	11364	9592	8541	7500	6641	6341	6038	5740	5438	5246	5067	4965
2m	22965	17664	14528	12125	10146	8719	7619	6752	6666	6294	5966	5673	5458	5279	5163
1024k	23822	18135	14969	12566	10547	9399	8207	7298	6902	6539	6203	5880	5660	5489	5406
512k	24619	18940	15880	13318	11212	9647	8417	7474	7105	6794	6526	6259	6153	6261	6261
256k	26530	21988	19346	16835	14621	12718	11125	9893	9750	9740	9494	9688	9700	9910	10401
128k	27617	24740	23729	21788	18726	16319	13881	12671	13614	13574	14133	14078	13505	14005	14051
64k	28815	26175	25582	24729	22138	20105	18081	15723	16083	16267	15475	16485	15168	17372	22573
32k	30189	29656	29452	28946	28372	28615	27941	27775	27743	27096	29531	27366	27023	26807	29403
16k	30159	29569	29027	28295	28236	27974	26975	29282	29707	30212	29217	28610	30182	22716	30987

表 2.1: 内存、步长与传输效率间的关系

可视化如下:

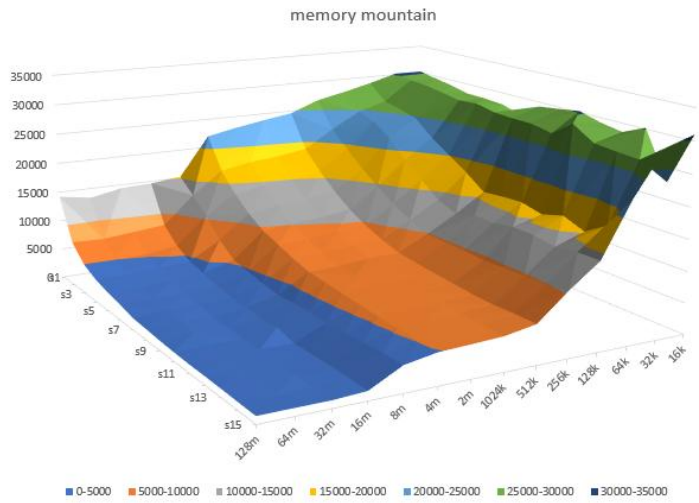

图 2.2: 内存、步长与传输效率间的关系的可视化

② 程序的测试内存取 16 kB 到 128 MB, 步长取 2^0 到 2^11, 得到下表所示的数据:

Clock	frequency	is	approx.	3647.8	MHz							
Memory	mountain	(MB/sec)										
	s1	s2	s4	s8	s16	s32	s64	s128	s256	s512	s1024	s2048
128m	14399	7533	2626	1711	1063	981	583	786	687	838	783	689
64m	14388	7656	3694	1908	1095	765	482	467	445	447	425	404
32m	13734	7667	4099	2220	1473	1250	816	709	662	702	691	670
16m	13935	8102	4632	2524	1669	1710	1062	1047	979	995	997	2305
8m	20692	14786	7590	4325	2998	3327	2409	2681	2531	1472	3712	3955
4m	21788	18381	12240	6801	4483	3536	3483	3721	3908	4253	4271	4883
2m	22923	18769	12462	6925	4552	3642	3596	3894	4026	4296	4716	4457
1024k	22711	18727	12462	6905	4555	3623	3602	3913	4029	4764	5382	9529
512k	23555	19659	13222	7316	4880	4126	4190	4465	5329	5642	4693	22234
256k	25013	21695	17573	10693	7774	6658	6885	8210	9778	10612	18677	10150
128k	25546	22711	19621	12277	9327	7981	8120	10672	13340	19455	11673	7296
64k	25700	23401	21345	12598	11388	10319	10734	19869	21223	11673	3242	3433
32k	27980	27669	26921	25672	27669	27068	27466	16101	11117	7296	3648	2084
16k	27746	27618	25850	27876	25940	29182	14591	10150	7296	4169	2084	652

表 2.2: 内存、步长与传输效率间的关系

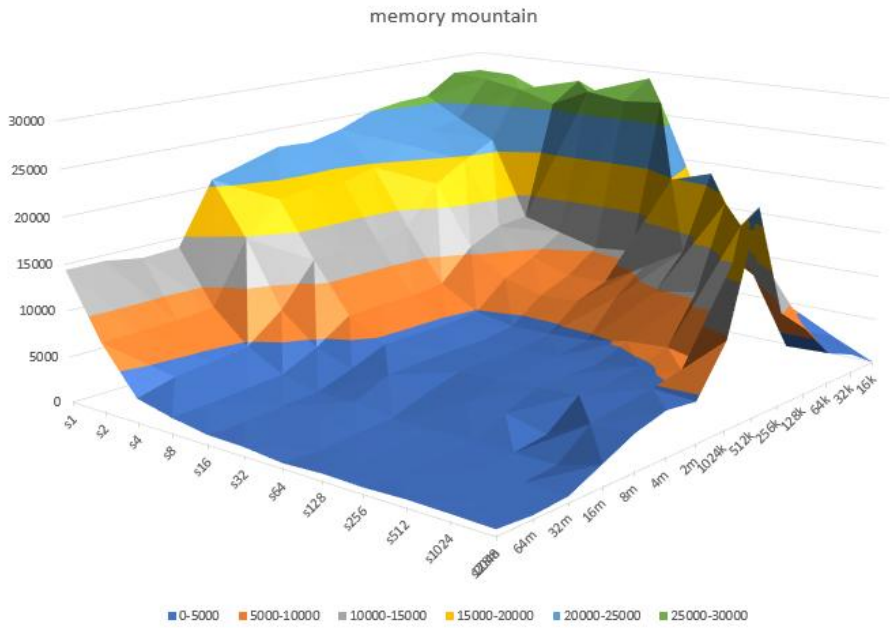

图 2.3: 内存、步长与传输效率间的关系的可视化

(4) 分析过程:

① 结果一: 程序根据不同的访问大小被分为 4 个山脊, 分别对应 L1、L2、L3 级缓存和主存. 每条山脊在步长为 8 或 9 时趋于平缓, 与教材中的存储器山模型大致相同.

② 结果二: 步长较大时图 2.4 与图 2.2 有较大差异, 这是因为步长过长时, 对 size 较小的情况, 访问的元素较少, 时间开销主要在初始化上, 使得时间偏高. size 逐渐增大时该现象逐渐减缓直至消失, 验证了算法的正确性.

(5) 验证实验结果:

① 用命令 getconf -a|grep CACHE 查看机器的不同级别的缓存, 结果如下图所示:

```
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

图 2.4: 机器的各级缓存

② 由上图: L1 缓存分为两部分, 分别对应指令缓存和数据缓存. 对数据缓存部分, 三级缓存的大小分别为 32KB、256KB、8MB. 图 2.2 和图 2.4 中四个山脊的分界线也分别在 32KB、256KB、8MB 处.

(6) 根据测试数据分析使用的 x86 机器由几级 Cache, 容量分别为多大.

① 图象中有四个山脊, 分别对应 L1、L2、L3 级缓存和主存, 故机器有 3 级 Cache.

② 各 Cache 的容量为山脊的分界线, 分别为 32KB、256KB、8MB.

(7) 根据测试数据分析 L1 Cache 的行数.

① 步长增大时, 计算机的吞吐量减小, 这与程序的空间局部性有关. 步长大于缓存的一行的 block 的字节数时, 吞吐量趋于稳定.

② 由图象知: 步长大于 8 时吞吐量基本趋于稳定, 则缓存的一个 block 可容纳 8 个 long 类型的元素, 而 long 类型在 x86-64 系统中为 8 Bytes, 则一个 block 的大小为 8 * 8 = 64 Bytes.

③ **行数 = 空间 / 块大小 = 32 kB / 64 B = 512 行.**

五、实验结论与心得体会

通过这次实验，我对空间局部性和存储器结构有了更深刻的理解。我掌握了缓存的一般结构，以及在通用系统中缓存分级的情况，并学会了如何通过吞吐量来反映不同级别缓存的大小和块大小关系。

首先，我认识到空间局部性对编写性能优化代码至关重要。为了打造具有良好空间局部性的代码，我们需要深入了解缓存的结构，并掌握步长大小对程序性能的影响。通过矩阵乘法的优化实验，我更加透彻地理解了这一概念，明白了即使在相同的时间复杂度和算法下，不同的空间局部性也会导致显著的性能差异。

其次，在存储器结构方面，缓存的大小虽然是一个底层概念，难以直观展现，但我意识到各级缓存的大小与计算机的吞吐量密切相关。通过调整步长和访问数组的大小，我们可以推导出存储器中各级缓存的大小关系，直观地认识到不同级别存储器的存在。

此外，通过调整不同的步长大小，我还能观察存储器之间传输的块大小，从而确定每一级缓存的行数。这让我感到非常神奇，因为我们竟然可以通过运行程序来获取底层硬件的信息，这进一步加深了我对硬件对程序性能影响的认识。

对于程序而言，若后续访问的空间地址紧邻先前访问的地址，则表明该程序具有良好的空间局部性。利用这一特性，我们可以优化程序，增强其空间局部性，减少空间地址访问时间，从而提高程序执行效率，降低时间复杂度。这对我今后的学习和实践都具有极大的帮助。

通过本次实验，我进一步深化了对 Cache 的理解，认识到空间局部性对程序性能的重要影响，并学会了如何利用空间局部性进行程序优化。同时，我也掌握了测量各级缓存的方法。这对我未来的代码编写具有极大的帮助，因为我意识到在编写代码时必须考虑空间局部性以提升代码性能。

指导教师批阅意见：

成绩评定：

指导教师签字： 2022 年　　月　日
备注：