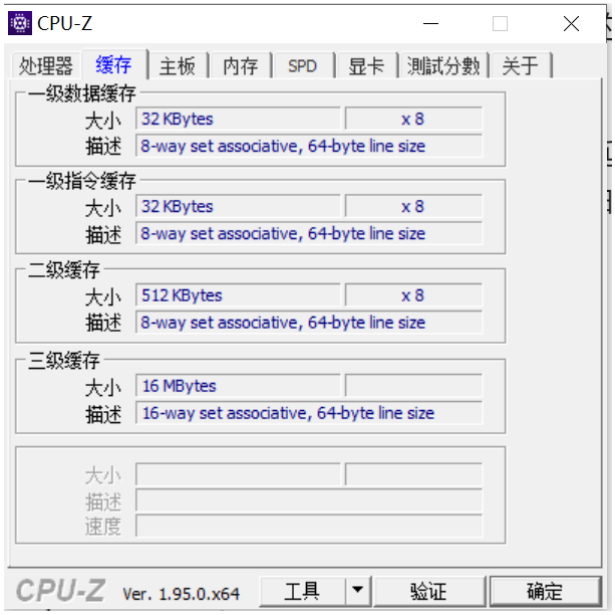
体系结构 第三次实验

Cache结构测量

在实验前，打开CPU-Z。



可见其一级数据缓存大小为32KB,二级缓存大小为512KB。一级数据缓存跟二级缓存的块大小均为64B。一级数据缓存跟二级缓存的相连度均为8。

测量Cache容量

题目分析

该题主要需要设计一种方法去测量一级数据缓存和二级缓存的大小。测量Cache容量最简单的方法就是不断从内存中读取一块连续的数据，然后观察平均访问速度。平均访问速度会在`L1 Cache Size`(一级数据缓存)和`L2 Cache Size`(二级缓存)中有一个激增。

设计思路

设计思路是：设置一个数组`array[ARRAY_SIZE]`,然后设置一个访问长度`size`，不断的以两倍方式增大`size`，去访问数组`array[size]`。同时因为单次访问速度较快，用时较短，不能较好的表现实验结果，所以对于每一个`size`，设置重复遍历次数`test_num`。当测试时对于`size`的访问延迟突然增加时，表明`size`已经大于缓存的大小。则$\frac{size}{2}$即为缓存大小。

由于在实验环境中无法做到像题目分析这么良好的实验现象，需要有一定容错，则通过测量`4~256K`大小的`array[size]`,寻找较大的访问时延突变，来确定`L1 Cache Size`，然后通过访问`64K~1024K`大小的`array[size]`来确定`L2 Cache Size`。

同时，询问助教后得知对于数组`array[size]`若顺序遍历，则其会被编译器优化掉，所以要设置一个`shift`值，来跳跃遍历。即对于`array[k]`的数组下标`k`，其更新方式为`k = k + shift`。

关于`shift`值和`test_num`的值的设定，需要针对机器感性调参而定。

关键实现

代码以及注释

```
int L1_DCache_Size() {
    cout << "L1_Data_Cache_Test" << endl;
    //add your own code
    clock_t start,finish;
    double time;
    double pre_time = 0;
    double maxtime = 0;
    int L1_size = 0;
    int shift = 128;
    for(int size = 4;size < 256;size = size * 2){//从 4K 到 256K去遍历size
        int size_kb = size * 1024;//将size 转为以kb为单位
        start = clock();
        for(int num = 0;num <= test_num;num ++){//对于每一个size,重复test_num次
            for(int i = 0; i < size_kb; i += shift){
                array[i] &= i;
            }
        }
        finish = clock();
        time = (double)(finish - start)/ size;
        //更新最大访问延迟跟size
        if((pre_time != 0)&&((time - pre_time) >= 0)){
            if(maxtime < (time - pre_time)){
                maxtime = time - pre_time;
                L1_size = size / 2;
            }
        }
        pre_time = time;
        cout<<"Test_Array_Size = "<<size << "KB    ";
        cout << "Average access time = " << time  << "ms"<< endl;
    }
    cout << "L1_Data_Cache_Size is  " << L1_size  << "KB" << endl;
    cout << "****************************************" << endl;
    return L1_size;
}
```

`L2_Cache_Size`的代码跟`L1_DCache_Size`代码的区别为其`size`的访问区间不同。`L1_DCache_Size`的`size`的访问大小为`64~2048`。

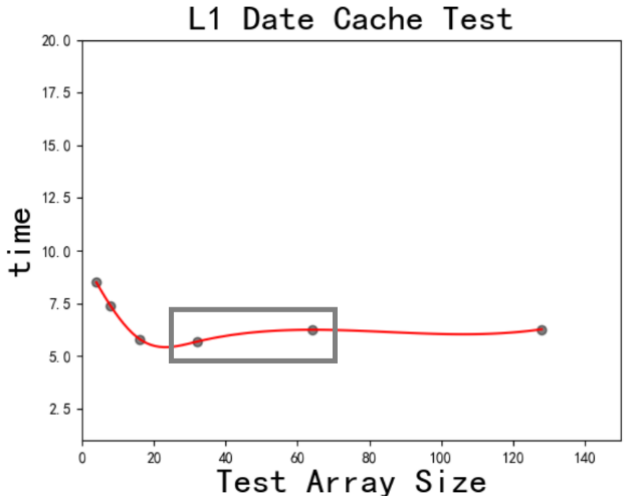
测试结果及分析

结果

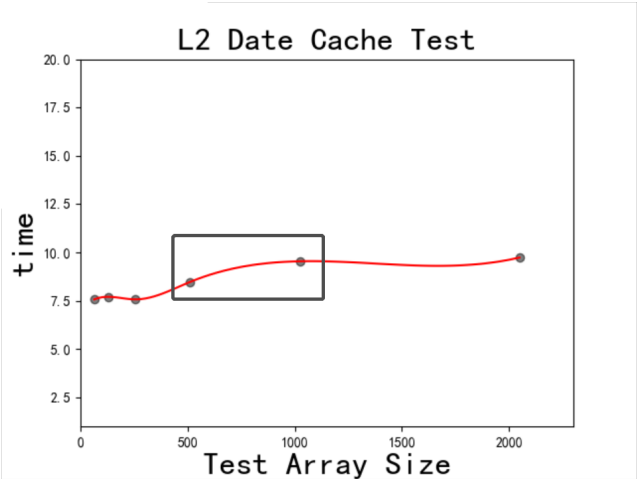
```
L1_Data_Cache_Test
Test_Array_Size = 4KB   Average access time = 8.5ms
Test_Array_Size = 8KB   Average access time = 7.375ms
Test_Array_Size = 16KB   Average access time = 5.8125ms
Test_Array_Size = 32KB   Average access time = 5.6875ms
Test_Array_Size = 64KB   Average access time = 6.25ms
Test_Array_Size = 128KB   Average access time = 6.27344ms
L1_Data_Cache_Size is  32KB
****************************************
L2_Data_Cache_Test
Test_Array_Size = 64KB   Average access time = 7.57812ms
Test_Array_Size = 128KB   Average access time = 7.71094ms
Test_Array_Size = 256KB   Average access time = 7.57812ms
Test_Array_Size = 512KB   Average access time = 8.48242ms
Test_Array_Size = 1024KB   Average access time = 9.54883ms
Test_Array_Size = 2048KB   Average access time = 9.74121ms
L2_Data_Cache_Size is  512KB
****************************************
```

Cache的访问时间折线图

如图，`size`为`64K`的时候，相较于`32K`有较快增长。



如图，在`size`为1024K的时候，较`512K`增长较多。



测量Cache块的大小

题目分析

对于数组的访问，如果是连续访问，Cache块的命中率很快，所以速度很快。但是如果设置一个跳跃长度`step`，不断按照间隔`step`去访问数组，就有可能每一次访问都会造成cache缺失，这样命中率较低，就需要较长时间，访问延迟增加。

设计思路

设计思路是设置一个跳跃长度`step`，不断增大`step`。当`step`增大到一定程度，可能会造成每次访问都缺失，造成访问时延增加。同时因为单次访问速度较快，用时较短，不能较好的表现实验结果，所以对于每一个`step`，设置重复遍历次数`test_num`。并且不用遍历整个`array[ARRAY_SIZE]`，设置`test_size`，遍历`array[test_size]`即可。

对于一级数据缓存，`array[test_size]`的大小应在`[0,L1_DCache_Size]`之间，对于二级缓存`array[test_size]`的大小应控制在`[L1_DCache_Size,L2_Cache_Size]`之间。

关键实现

代码及注释

```
int L1_DCache_Block() {
    cout << "L1_DCache_Block_Test" << endl;
    //add your own code
    Clear_L1_Cache();
    double time;
    double pre_time = 0;
    double maxtime = 0;
    int L1_size = 0;
    clock_t start,finish;
    for(int step_size = 8;step_size < 256;step_size = step_size *2){//设置不同的
step
        start = clock();
        for(int num = 0;num < test_num;num ++){//重复遍历
            for(int i = 0,test_size = 0;test_size < 1024;i +=
step_size,test_size ++){//test_size控制访问数组的大小
                array[i] &= test_size;
            }
        }
        finish = clock();
        time = (double)(finish - start) * 1000 / CLOCKS_PER_SEC ;
        if((pre_time != 0)&&((time - pre_time) >= 0)){
            if(maxtime < (time - pre_time)){
                maxtime = time - pre_time;
                L1_size = step_size;
            }
        }
        pre_time = time;
        cout<<"Test_Step_Size = "<<step_size << "B    ";
        cout << "Average access time = " << time  << "ms"<< endl;
    }
    cout << "L1_DCache_Block_Size is  " << L1_size  << "B" << endl;
    cout << "****************************************" << endl;
}
```

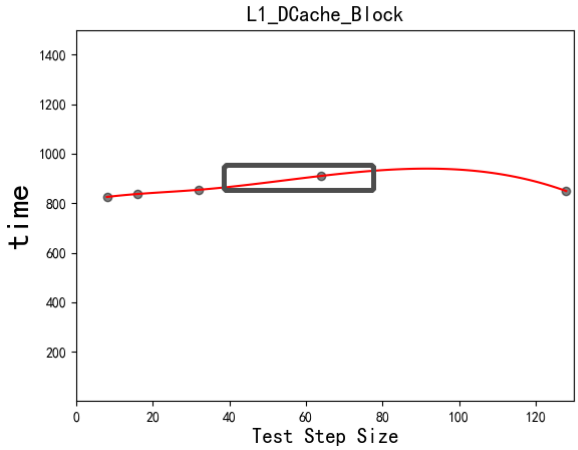
测试结果及分析

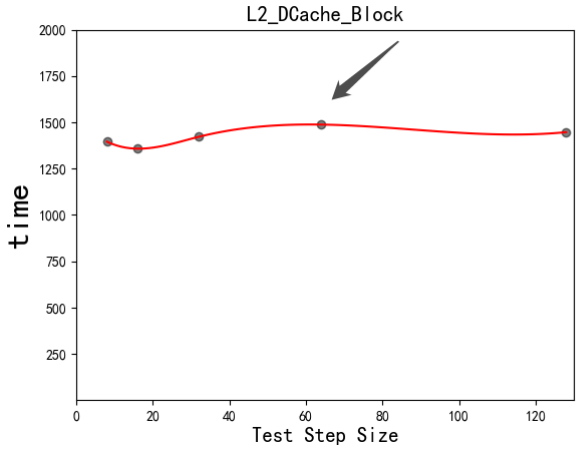
结果

```
(pytorch) PS C:\Users\shen_\Desktop\lab3_student\src> a.exe
L1_DCache_Block_Test
Test_Step_Size = 8B   Average access time = 825ms
Test_Step_Size = 16B   Average access time = 837ms
Test_Step_Size = 32B   Average access time = 854ms
Test_Step_Size = 64B   Average access time = 910ms
Test_Step_Size = 128B   Average access time = 849ms
L1_DCache_Block_Size is  64B
****************************************
L2_Cache_Block_Test
Test_Step_Size = 8B   Average access time = 1396ms
Test_Step_Size = 16B   Average access time = 1358ms
Test_Step_Size = 32B   Average access time = 1422ms
Test_Step_Size = 64B   Average access time = 1488ms
Test_Step_Size = 128B   Average access time = 1447ms
L2_Cache_Block_Size is  64B
****************************************
```

访问时间折线图

如图当`step`为`64B`的时候，访问时延变大。





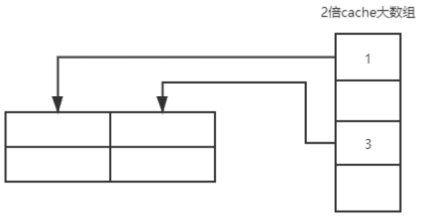
测量Cache容量

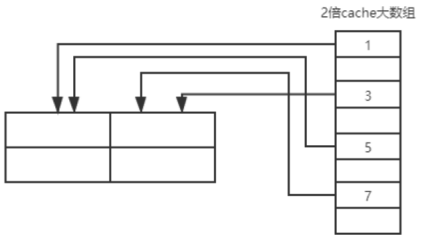
题目分析

设计一种测量相连度的方法。

设计思路

可以通过访问一个两倍于`Cache`大小的数组来实现。





若Cache为2路组相联结构，当分为4块时，不会发生不命中。但是将其分为8块时，会有Cache被不断替换出去，使得访存时间增加。

则可以假设Cache的相联路为`way`，不断的增大变量`way`进行测试，对于每一个`way`，可按照之前的思想，将数组`array[2*L_Cache_Size]`分为`group = 2*way`组，然后依次访问第1、3、…组，统计访问时延。同时因为单次访问速度较快，用时较短，不能较好的表现实验结果，所以对于每一个`way`，设置重复遍历次数`test_num`。当对某一个`way`出现访存时延增加时，前一次就是相联度的值。

关键实现

代码及注释

```
int L1_DCache_Way_Count() {
    cout << "L1_DCache_Way_Count" << endl;
    //add your own code
    //用一个两倍于Cache大小的数组来实现
    Clear_L1_Cache();
    int array_size = 2 * L1_cache_size;
    double time;
    double pre_time = 0,maxtime = 0;
    int L1_way = 0;
    int shift = 64;
    for(int way = 2;way < 32;way = way * 2){//遍历可能的way
        int group = way *2;//分组
        clock_t start = clock();
        int length = array_size / group;
        for(int num = 0;num < test_num;num ++){
            for(int i = 0;i < group / 2;i ++){
                int begin = 2*i*length;//访问1，3，5，7...
                for(int k = begin;k < begin + length;k +=shift){
                    array[k] &= k;
                }
            }
        }
        clock_t finish = clock();
        time = (double)(finish - start) * 1000 / CLOCKS_PER_SEC ;
        if((pre_time != 0)&&((time - pre_time) >= 0)){
            if(maxtime < (time - pre_time)){
                maxtime = time - pre_time;
```

```
                L1_way = way / 2;
            }
        }
        pre_time = time;
        cout<<"way = "<<way << "\t";
        cout << "Average access time = " << time  << "ms"<< endl;

    }
    cout << "L1_DCache_Way_Count is  " << L1_way << endl;
    cout << "****************************************" << endl;
    return L1_way;
}
```

`L2_Cache_Way_Count`代码与其基本相同，不做重复叙述。

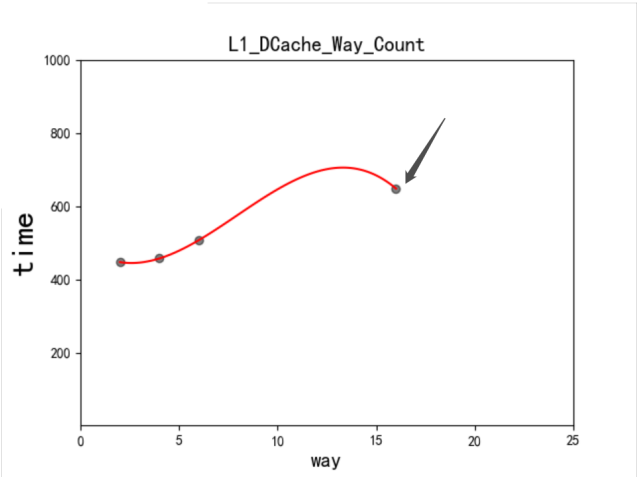
测试结果及分析

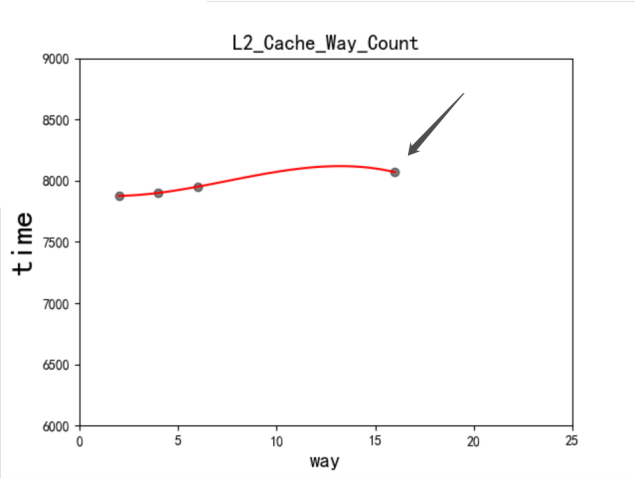
实验结果

```
(pytorch) PS C:\Users\shen_\Desktop\lab3_student\src> a.exe
L1_DCache_Way_Count
way = 2 Average access time = 447ms
way = 4 Average access time = 457ms
way = 8 Average access time = 507ms
way = 16        Average access time = 648ms
L1_DCache_Way_Count is  8
****************************************
L2_Cache_Way_Count
way = 2 Average access time = 7876ms
way = 4 Average access time = 7900ms
way = 8 Average access time = 7951ms
way = 16        Average access time = 8069ms
L2_DCache_Way_Count is  8
****************************************
Press any key to continue . . .
```

访问时间折线图

如图，当`way = 16`时，访问时延增加明显。





矩阵乘法优化

空间局部性

题目分析

调整基本算对矩阵的访问顺序，即可进行连续访问，减少所需时间。

设计思路

同题目分析，即调换循环的顺序即可。

同时利用`register int`代替`int`，这样将原来存在堆里的值存在寄存器里，增加速度。

关键实现

```
//=========================================================
    for (register int i = 0; i < 1000; i++) {
        for (register int k = 0; k < 1000; k++) {
            const int s = a[i][k];
            for (register int j = 0; j < 1000; j++) {
                d[i][j] += s * b[k][j];
            }
        }
    }
//=========================================================
```

实验结果

```
(pytorch) PS C:\Users\shen_\Desktop\lab3_student\src> g++ -O0 matrix_mul.cpp
(pytorch) PS C:\Users\shen_\Desktop\lab3_student\src> a.exe
time spent for original method : 2896 ms
time spent for new method : 1744 ms
```

问题与解决

感觉这次试验有一点小小的问题，就是在测`L2`的`Block`数据跟其他数据大小的时候，不可避免的会用到`L1`，没有办法完全的只对`L2`进行测试。只能说`L1`占比较小，对`L2`的结果影响较小。

还有测`cache size`和`block`的时候的`shift`值完全靠感性调参，有一点为了答案而凑答案。