1. 题目分析

本次实验使用程序进行存储访问来估算 cache 相关参数，通过利用 cache 缺失时访问时间的突然增大来判断 cache 大小、块大小、相联度等参数。

矩阵乘法任务要求对矩阵乘法进行优化，基础的矩阵乘法实现中的访存局部性不好，cache 缺失率高，任务中对矩阵乘法进行优化使之具有更好的访存局部性，加快运算速度。

2. 设计思路

（1） cache 大小估算

通过对 cache 进行大量随机有意义的访问，并统计访存时间来判断是否出现了大量的 cache 缺失，在访存时间徒增的节点的前一个节点即为 cache 块大小。

（2） cache 块大小估算

通过刻意设计步长为 cache 块大小的 cache 访问来制造大量的 cache 缺失，通过不同步长对 cache 进行访存，当出现访存时间激增时即为 cache 块大小。

（3） cache 相联度估算

通过访问 1 个 2 倍 cache 大小的内存区域，并刻意持续访问映射到同一个 cache 组的不同 cache 块来制造 cache 缺失，通过设置不同的组数来对 cache 进行访问，当出现访存时间激增的前 1 个节点即为 cache 相联度。

（4） 矩阵乘法优化

原有的矩阵乘法 c[i][j] += a[i][k] × b[k][j]对矩阵 B 的访存局部性效果不好，现在通过迭代计算的方式来充分利用矩阵 B，即在计算矩阵 C 时，c[i][k]+=a[i][j]*b[j][k]，每次对矩阵 C 计算 1 行，并迭代计算，使得矩阵 B 的访问是连续进行的，从而优化矩阵乘法。

3. 关键实现

（1） Cache 大小计算函数

```
int Test_Cache_Size(int index, int *avg_time)
{
    int size;
    int data_size;
    char data;
    clock_t begin, end;
    size = index;
    for (int i = 0; i < 5; i++)
    {
        data_size = (1 << (size + 10));
        Clear_L1_Cache();
        Clear_L2_Cache();
        begin = clock();
        for (int j = 0; j < SIZE_TEST_TIMES; ++j)
        {
            data += array[(rand() * rand()) % data_size];
        }
        end = clock();
```

```
        avg_time[i] = end - begin;
        size++;
    }

    int temp, process_time;
    size = index;
    int cache_size_result = (1 << size);
    process_time = avg_time[1] - avg_time[0];
    for (int i = 0; i < 5; i++)
    {
        cout << "Test_Array_Size = " << (1 << size) << "KB, ";
        cout << "Average access time = " << avg_time[i] << "ms" << endl;
        if (i < 4)
        {
            temp = avg_time[i + 1] - avg_time[i];
            if (temp > process_time)
            {
                cache_size_result = (1 << size);
                process_time = temp;
            }
        }
        size++;
    }
    return cache_size_result;
}
```

（2） cache 块大小计算函数

```
int Test_Cache_Block(int index, int *avg_time)
{

    int size;
    int data_size;
    char data;
    clock_t begin, end;
    unsigned int temp;
    temp = index;
    for (int i = 0; i < 8; ++i)
    {
        begin = clock();
        for (int j = 0; j < temp; ++j)
        {
            for (int k = 0; k < ARRAY_SIZE; k += temp)
            {
                data += array[k];
            }
```

```
        }
        end = clock();
        avg_time[i] = end - begin;
        temp = temp << 1;
    }

    temp = index;
    int process_time = avg_time[1] - avg_time[0], temp_time;
    int cache_block_result;
    for (int i = 0; i < 8; ++i)
    {
        cout << "Block_Size = " << temp << " B, ";
        cout << "Average access time = " << avg_time[i] << "ms " <<
endl;
        if (i < 7)
        {
            temp_time = avg_time[i + 1] - avg_time[i];
            if (temp_time > process_time)
            {
                cache_block_result = (1 << size);
                process_time = temp_time;
            }
        }
        size++;
        temp = temp << 1;
    }
    return cache_block_result;
}
```

（3） cache 相联度计算函数

```
int Test_Cache_Way_Count(int array_size, int index, int *avg_time)
{
    int temp;
    int array_jump;
    char data;
    clock_t begin, end, temp_time;
    int process_time, way_count_result;
    temp = index;
    for (int i = 0; i < 5; ++i)
    {
        array_jump = array_size / temp;
        begin = clock();
        for (int j = 0; j < WAY_TEST_TIMES; ++j)
        {
```

```
            for (int k = 0; k < temp; k += 2)
            {
                memset(&array[k * array_jump], 0, array_jump);
            }
        }
        end = clock();
        avg_time[i] = end - begin;
        temp = temp << 1;
    }

    temp = index;
    process_time = avg_time[1] - avg_time[0];
    way_count_result = temp / 2;
    for (int i = 0; i < 5; ++i)
    {
        cout << "Way_Count = " << temp / 2 << ", ";
        cout << "Average access time = " << avg_time[i] << "ms " <<
endl;
        if (i < 4)
        {
            temp_time = avg_time[i + 1] - avg_time[i];
            if (temp_time > process_time)
            {
                way_count_result = temp / 2;
                process_time = temp_time;
            }
        }
        temp = temp << 1;
    }
    return way_count_result;
}
```

（4） 矩阵乘优化

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

        }
```

```
    }
```

4. 测试结果与分析

L1、L2cache 大小测试结果如图 1、图 2 所示。

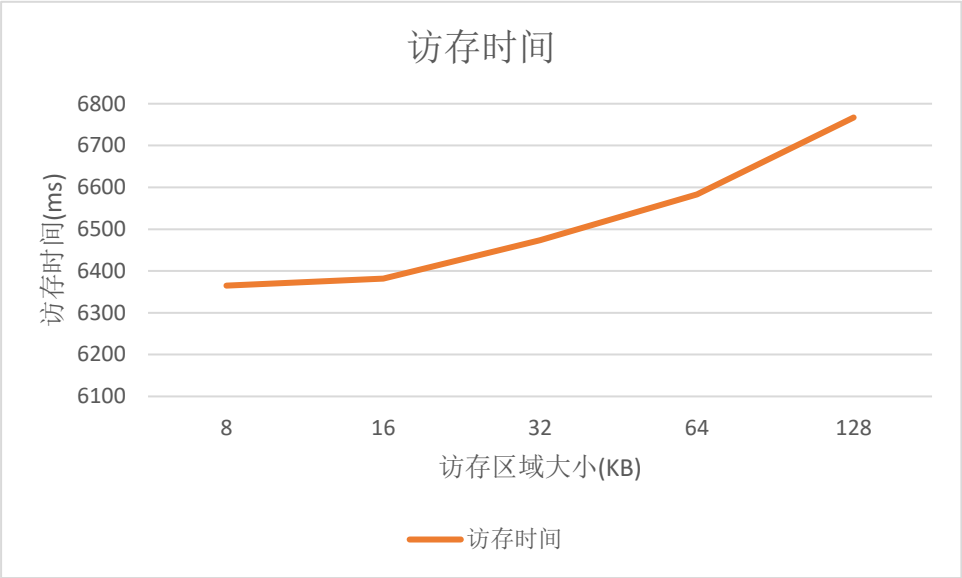

图 1 L1 cache 大小测试结果

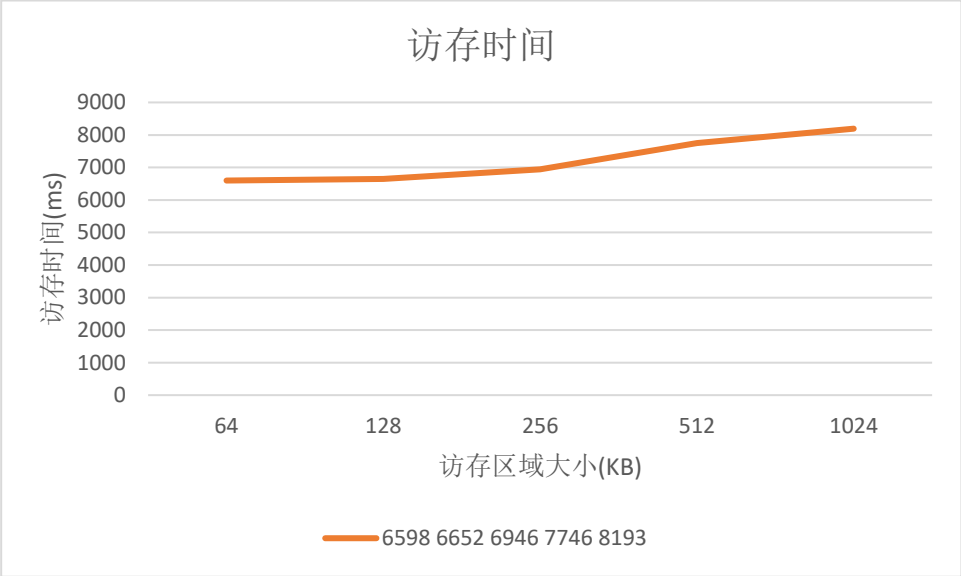

图 2 L2 cache 大小测试结果

L1、L2cache 块大小测试结果如图 3、图 4 所示。

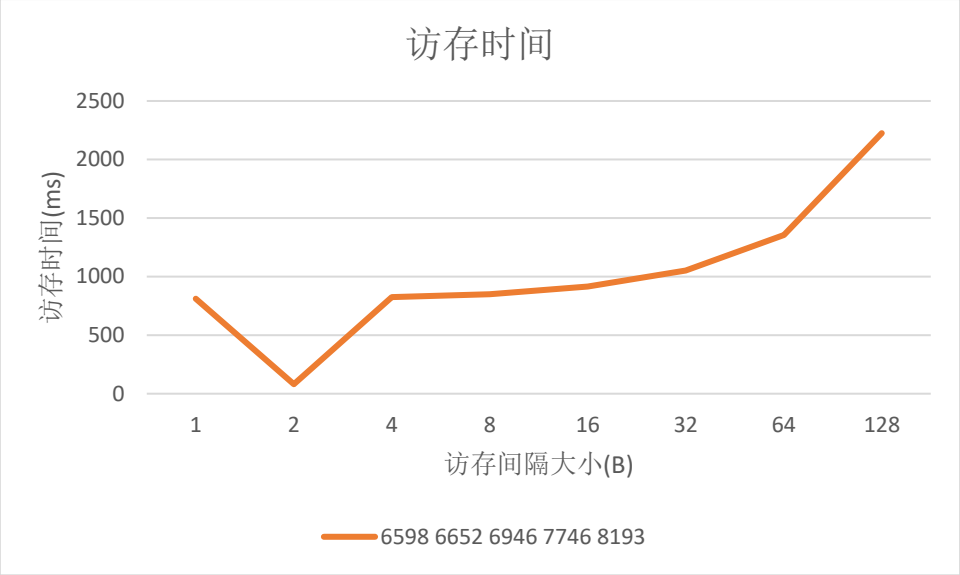

图 3 L1 cache 块大小测试结果

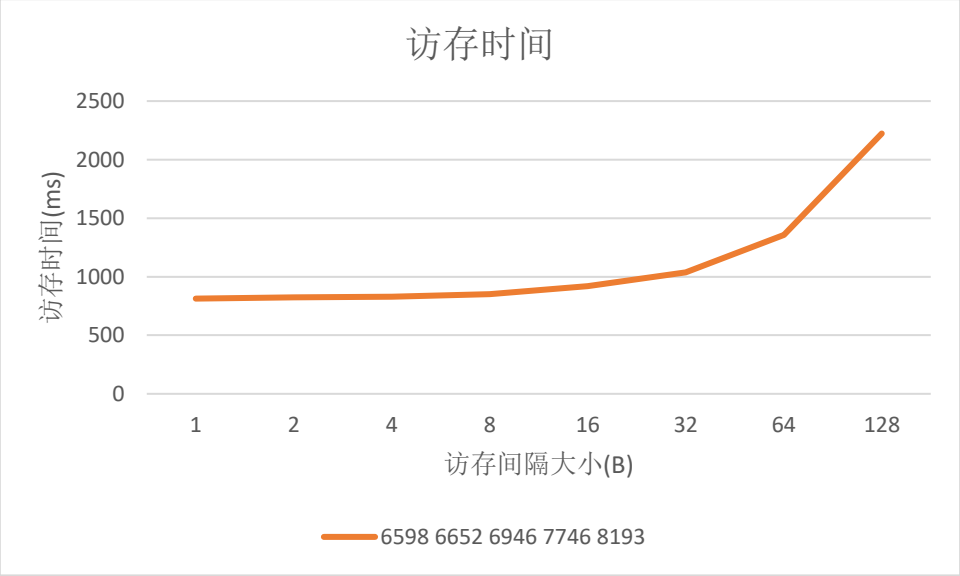

图 4 L2 cache 块大小测试结果

L1、L2cache 相联度测试结果如图 5、图 6 所示。

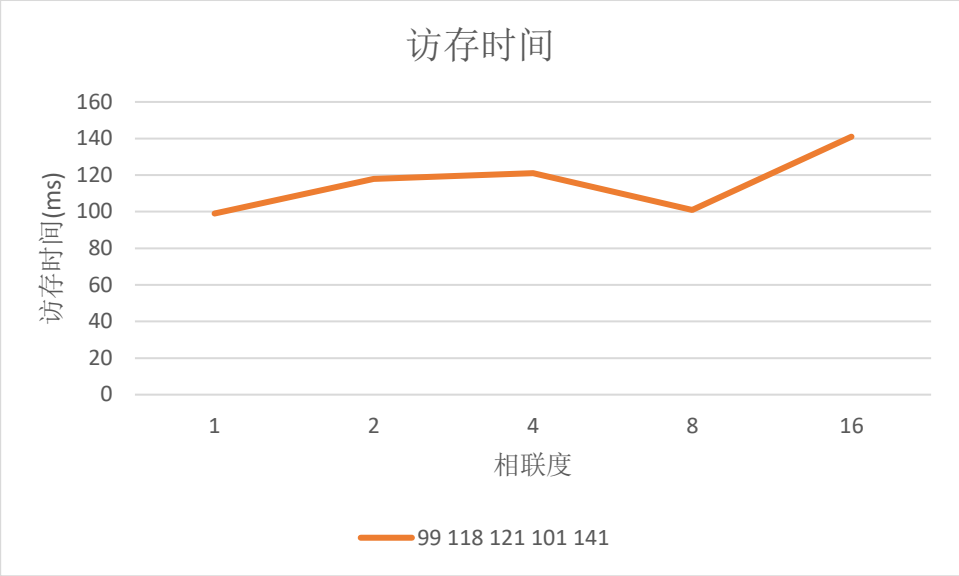

图 5 L1 cache 相联度测试结果

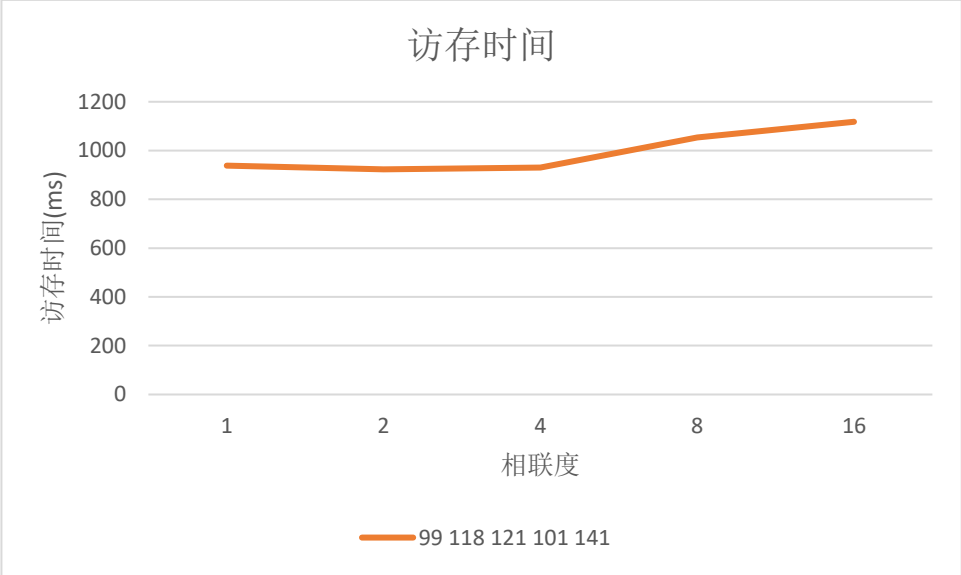

图 6 L2 cache 相联度测试结果

优化后矩阵乘法效果如图 7 所示

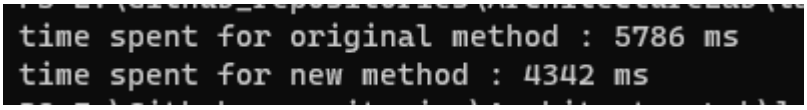

图 7 优化后矩阵乘法输出