矩阵乘法性能优化实验

一、实验配置

【平台】：Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz、Windows10 专业版

【IDE】：Microsoft Visual Studio 2015 Enterprise

二、优化性能展示

数据规模较小时，加速比不稳定，当数据规模增大到 2000 维以上时，可达到接近 30 倍的加速比。表 1 中展示的是从 1024 维到 10240 维的矩阵乘法运算的时间记录。

表 1　不同矩阵规模下的加速比

矩阵规模	老师的程序运行时间 T1（秒）	我的程序运行时间 T2（秒）	加速比（T1/T2）
1024	4.359	0.137	31.82
2048	34.67	1.08	32.10
3072	118.697	4.546	26.11
4096	276.356	8.935	30.93
5120	536.452	18.73	28.64
6144	962.734	27.905	34.50
7168	1471.89	49.167	29.94
8192	2288.99	76.873	29.78
9216	3226.3	102.493	31.48
10240	4289.64	128.812	33.30

注：由于运行时间太久，表中加速比只是一次运行的结果，没有多次执行求平均。

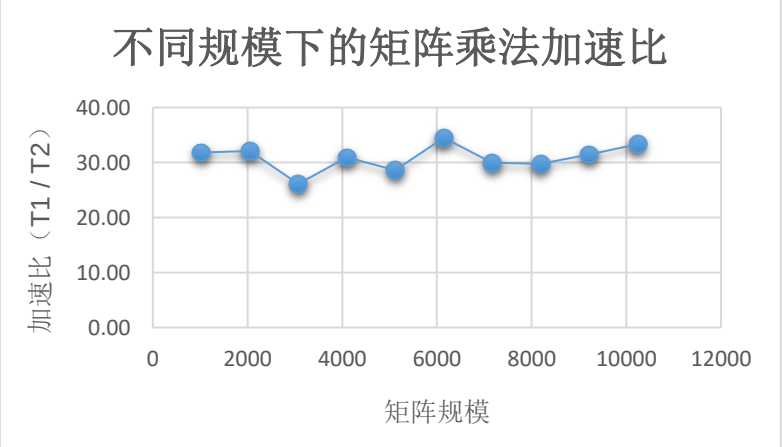

图 1　不同规模下的矩阵乘法加速比

从上图 1 中可以看出，矩阵规模 1024 维以上的时候，加速比在 30 附近徘徊。

三、具体实现用到的技术

主要技术是访存优化，即提高处理器读取数据的性能。

使用“CPU-Z”软件可以查看电脑的一些参数，如下图 2 所示。我的电脑是 Intel 酷睿 i5 系列的四代 CPU，支持 AVX 指令集，L1 缓存 256KB（8 路），L2 缓存 1MB（8 路），L3 缓存 6MB（8 路），线程数为 4。下边将利用这些参数来对访存进行优化。

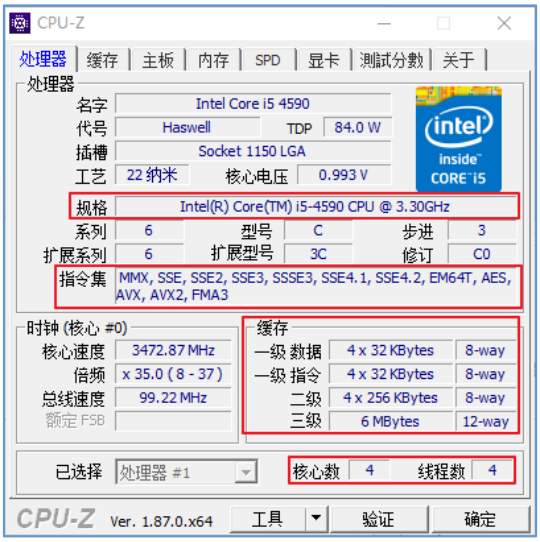

图 2　查看 CPU 参数

1. 利用 Cache（先分块再折叠）

这里主要是利用时间局部性和空间局部性。由于缓存空间有限，所以先对矩阵做划分，分成若干块。之后再对分割的块做折叠，即将矩阵块拷贝一份到连续空间，从而将矩阵块折叠成一位数组。

放在连续的地址空间，有利于计算机将整个矩阵块的内容都存储到缓存中，而每个矩阵块在计算时要反复使用，这样可以大大减少时间上的开销，也是利用到了时间局部性和空间局部性。由于分割后的矩阵块用完之后会及时释放掉并且矩阵块的规模不大，所以空间上的开销较小。

如图 3 所示，我在程序中对于规模超过 2048 维的矩阵，块大小设置为 512 Size，矩阵采用的是 4 字节的浮点数，故一个矩阵块占用 512*512*4=1MB 的存储空间，这部分主要利用 L3 的缓存空间。

```
int BlockSize;  // 矩阵分块大小，0代表不分块
// 经测试，512*512的分块速度较快，这个规模 512*512*4 = 1 MB
// 我的电脑是 "Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz"
// 其中 L1缓存为256KB，L2缓存为1MB，L1缓存为6MB
if (nCalcu >= 2048) { BlockSize = 512; }  // 超过2048的规模时，块大小设置为512
else if (nCalcu >= 768) { BlockSize = 256; }  // 规模在768到2048时，块大小设置为256
else if (nCalcu >= 256) { BlockSize = 64; }  // 规模在256到768时，块大小设置为64
else { BlockSize = 0; }  // 规模小于256时，不分块，直接用AVX算
```

图 3　矩阵分块大小的设计

对于矩阵块进行计算时，则是在 L3 的基础上利用 L1 缓存和 L2 缓存。每次计算时，使

用 AVX 指令同时读取 A 矩阵分块的 8 行数据和矩阵 B 的 1 列数据进行运算，这样有利于对于读取到的 B 矩阵分块的数据进行充分的数据复用，这部分数据在前边加载时被缓存在 L1，后续可节省读取时间，从而提升时间效率。

2. 多线程（使用四个线程的同时控制负载）

如前所述，我的电脑线程数为 4，所以我在该程序中设置了 4 线程。

为了使每个线程之间没有数据相关性，程序中对矩阵 B 的的列进行划分，之后传给不同的线程，这样相当于每个线程计算矩阵 C 的不同的列的值，四个线程之间没有数据冲突。

为了更好的在每个线程上对矩阵进行分块，这里对每个线程所分到矩阵 B 的列的规模进行了控制，在程序中体现在如图 4 所示的代码中。

```
// 对四个线程均衡负载（尽量使得四个线程分到的矩阵 B 的列数比较均衡），若改为其他线程数需从新设计
if ((nCalcu / BlockSize) % threadNum == 0) {
    start_1 = ((nCalcu / BlockSize) / threadNum)*BlockSize;
    start_2 = (2 * (nCalcu / BlockSize) / threadNum)*BlockSize;
    start_3 = (3 * (nCalcu / BlockSize) / threadNum)*BlockSize;
} else {
    start_1 = (((nCalcu / BlockSize) / threadNum) + 1)*BlockSize;
    start_2 = ((2 * (nCalcu / BlockSize) / threadNum) + 1)*BlockSize;
    start_3 = ((3 * (nCalcu / BlockSize) / threadNum) + 1)*BlockSize;
}

pool.push_back(thread(cacheIntrin, nCalcu, 0, start_1, mat_a, mat_b, mat_c, BlockSize));
pool.push_back(thread(cacheIntrin, nCalcu, start_1, start_2, mat_a, mat_b, mat_c, BlockSize));
pool.push_back(thread(cacheIntrin, nCalcu, start_2, start_3, mat_a, mat_b, mat_c, BlockSize));
pool.push_back(thread(cacheIntrin, nCalcu, start_3, nCalcu, mat_a, mat_b, mat_c, BlockSize));
pool[0].join();
pool[1].join();
pool[2].join();
pool[3].join();
```

图 4　对不同线程划分不同的数据块

3. 使用 AVX256 指令（一次加载八个浮点数）

程序中使用 AVX256 指令来提高访存效率，因为 AVX256 指令可以同时操作 8 个 4 字节的浮点数，这将原来需要 8 次访存的时间节省到了 1 次。

4. 边缘处理、矩阵转置、循环展开

边缘填充：为了在 AVX 指令访问、矩阵分块时方便处理，程序中对矩阵边缘填充零，使得矩阵的 Size 可以整除 8。

矩阵转置：为了使矩阵相乘运算更方便快捷并且更好地使用 AVX 指令，在 B 矩阵分块折叠之前先进行了转置操作，如下图 5 所示。

```
float *packB = new float[kMicroSize*jMicroSize];  // 矩阵 B 块大小为 kMicroSize * jMicroSize
for (_row = 0; _row < kMicroSize; _row++) {  // Pack matrix_B : Transpose and Vectorization
    float *mat_b_row = mat_b[k + _row];
    for (_col = 0; _col < jMicroSize; _col++) {  // 矩阵 B 的块的左上角坐标为 (k, j)
        packB[_col * kMicroSize + _row] = mat_b_row[j + _col];
    }
}
```

图 5　矩阵 B 的分块先转置后折叠

循环展开：将原来的循环展开，在矩阵块相乘时，同时对 A 矩阵分块的 8 行数据进行乘法操作，减少循环次数，从而减少汇编级别的指令跳转预测。如下图 6 所示，这是 for 循环首尾处代码对应的汇编指令。VS2015 中可以查看汇编代码，可以看到 01001788 处的代码每次都要跳转回到 010013C4 处对变量进行加法操作，之后在 010013DF 处的 jge 指令

会再次进行判断是否需要跳转。循环展开则能够减少这种由于跳转带来的流水线断流现象。

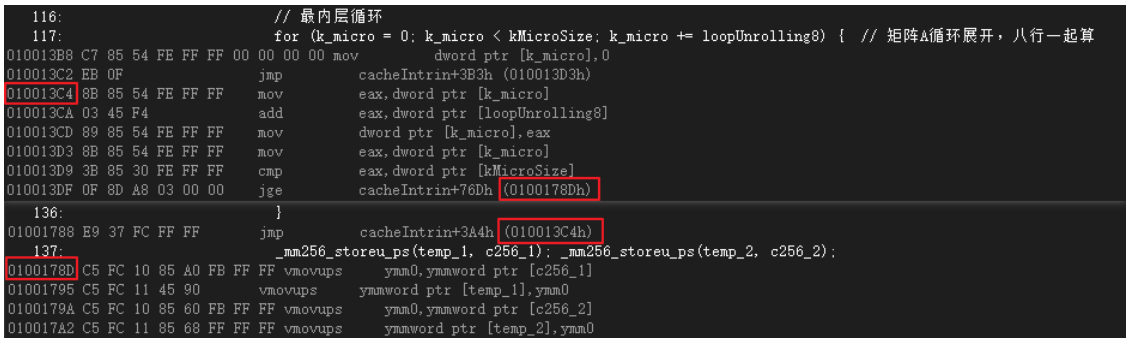

图 6　For 循环首尾处的汇编指令

四、总结

内存的访存速度和 CPU 的计算速度差距很大，这导致对于存取操作较多的运算会非常慢。这里的矩阵乘法优化，主要思路就是针对矩阵 A 和矩阵 B 的访存进行优化。

计算机的设计者为了解决存储墙的问题，设置了多级存储部件。越靠近 CPU，存储部件的存取速度越快并且存储容量越小；越远离 CPU，存储部件的存取速度越慢并且存储容量越大。L1、L2 和 L3 这三级缓存是集成在芯片上的，容量小速度快。若能更好的利用这三级缓存，则可以大大提升程序的运行速度。