刘一郴

12112609

2022/11/27

**Report for Assignment 4**

**1.code**

<https://github.com/cunlidaniang/coding-workshop/tree/master/c%2B%2B%20workshop/project%204>

**2.设备配置**

Windows10 x64

手机屏幕截图

描述已自动生成

图示

中度可信度描述已自动生成

**3.优化过程**

**3.1 avx**

Matrix \* matmul\_improved(const Matrix \* const pA, const Matrix \* const pB)

一开始仅仅使用avx，对于1k\*1k大小的矩阵乘法，速度提升至7倍左右

文本

描述已自动生成

**3.2 avx + openMP**

Matrix \* matmul\_improved(const Matrix \* const pA, const Matrix \* const pB)

然后尝试在原有代码的基础上使用openMP,对于1k\*1k大小的矩阵效率升至31倍左右

然而对于8k\*8k大小矩阵乘法，耗间大幅度增长，是1k\*1k大小矩阵乘法时间的3000倍，远高于理论的512倍。原因可能是L2 cache容量较小，不断地访问L2 cache很快就填满，速度就拖累下来。

文本

描述已自动生成

文本

描述已自动生成

**3.3 Avx + openMP + Blocking**

Matrix \* matmul\_improved\_Final(const Matrix \* const pA, const Matrix \* const pB)；

inline void do\_block(float \* C, float \* A, float \* B, size\_t row1, size\_t col1, size\_t row2, size\_t col2, size\_t si, size\_t sj, size\_t sk , block\_size)；

针对以上L2cache问题，我们采用分块乘法的方式，把大矩阵分成8\*8小矩阵组成的分块矩阵。然后每次只进行8\*8的矩阵乘法，就减少了需要存在L2cache的数据的量，避免L2cache刷满问题。

这样对于1k\*1k的矩阵乘法耗时15s左右，耗时为朴素算法的63倍左右，8k\*8k的矩阵乘法耗时25s左右，是1k\*1k大小矩阵乘法耗时的1700倍左右，速度是不分块的4倍左右。

然而对于16k\*16k矩阵乘法，耗时400s左右，是8k\*8k耗时的16倍左右。

总结这样矩阵耗时基本为理论增长，但仍然有差距。

文本

描述已自动生成

文本

描述已自动生成

**3.4 尝试继续优化分块矩阵 8\*8->16\*16**

在原有代码的基础上，把分块大小从8\*8提升至16\*16.发现8k\*8k矩阵乘法耗时降为12s左右，为原来的1/2左右,16k\*16k矩阵乘法耗时降为14s左右，为原来的1/4左右。且8k\*8k到16k\*16k的耗时增长为11倍，越发贴合理论的8倍。

图片包含 图形用户界面

描述已自动生成

**3.5 继续尝试 16\*16->32\*32->64\*64**

在原有代码的基础上，把分块大小从16\*16提升至32\*32.

16\*16 32\*32 64\*64

8k\*8k 12s 9s 7s

16k\*16k 148s 102s 65s

32k\*32k /// 585s 629s

由此发现，提升分块大小能进一步提升效率，对于数据规模增长，较好贴合O（n^3），带来的收益是矩阵规模提升时性能不会下降。

文本

描述已自动生成 32\*32

文本

描述已自动生成 64\*64

**3.6 64k\*64k矩阵乘法**

由于64k\*64k过于巨大，程序直接killed, 但根据3.5提供的数据我们可以预测，64k\*64k矩阵乘法耗时约在5000s左右。

**4. openBLAS对比**

由于64k\*64k过于巨大，程序直接killed, 仅仅测试了16k\*16k矩阵乘法的openBLAS耗时为14s.

显然，相比openBLAS，该project中实现的最优算法为65s，与openBLASS的14s仍有4倍速的差距。