CS205 C/C++ Programming Report Of Project 4

Name: 杨博乔
SID: 12112805

Part 1. analysis

因为是基于project3的，本次project进行了以下优化（上次没实现的

1. 使用 size_t 而非 int 进行位置的读取&存储
2. 矩阵硬拷贝使用了memcpy函数而非手动遍历
3. 处理了一些可能由我自己导致的内存泄漏的地方
4. 限制用户必须全部使用指针

本次题目需求：首先写一个较劣的朴素乘法，接下来使用**复杂度更优的算法**，**SIMD和openMP等降低常数的手段**进行优化，并在一些矩阵大小下与**openblas**库比较速度。

本次测试基于intel的x86指令集，运行于内存16G的arch linux上。多数时候使用了<sys/time.h>的gettimeofday，因为可以读取微秒级别差异，但是对于多线程失效。

该project实现了任意大小的矩阵乘法，但本文为了便于解释，假设为**n阶方阵**，希望读者能更好的理解。

对于优化部分：

裸的矩阵乘

```
for(int i=0;i<n;++i)
    for(int j=0;j<n;++j)
        for(int k=0;k<n;++k)
            C[i][j]+=A[i][k]*B[k][j];
```

显然对于缓存命中这是不好的，因为内存不连续。为了加速，就要尽可能使内存访问连续，即不要跳来跳去。原理是探针去内存中取值时，会尝试把附近的元素也扔进cache,然后会优先在cache中寻找再去访问内存。 因此，让c和b的读更连续是好的，这一点我将在接下来的转置优化部分提到。 变换循环顺序为i,k,j是最快的，伪代码如下

mul_ikj

```
for(int i=0;i<n;++i)
    for(int k=0;k<n;++k)
        s=A[i][k];
        for(int j=0;j<n;++j)
            C[i][j]+=s*B[k][j];
```

容易发现它尽可能连续了。 定性分析：假设对于连续内存，访问时不跳，记**跳跃数J**为探针发现缓存里没有去内存找的次数 则对于以下循环顺序

```
ikj: T(n*n)
kij: T(2*n*n)
ijk: T(n*n*n+n*n-n)
jik: T(n*n*n+n*n+n)
kji: T(2n*n*n)
jki: T(2n*n*n+n*n)
```

对于测试1024阶方阵，开O3，ikj的速度(852.3ms)较jki的速度(18005.2ms)提升了约20倍，猜测是聪明的编译器帮助优化了慢的那个。

接下来是SIMD优化，这里进行了矩阵转置，使用了 **__mm256** 来进行八位浮点数的一次向量化，并单个的处理了后续部分，此外，该算法先将b转置再乘，常数更小了。 因为转置是n方的嘛，n上了千这玩意的浪费就可以忽略不计了，此外，‘写’操作不需要考虑缓存命中，所以只要连续读就可以啦。 曾经的伪代码：

```
c[i][j]=a[i][]*b[][j];
```

连续的b访问飞快，而经过转置的b则对于原乘法可表示为

```
c[i][j]=a[i][]*b[j][];
```

因此我们可以直接对这两个向量进行向量点乘扔给c,这个过程可以被SIMD并行，**请在支持AVX的CPU中使用，当然改成NEON也就稍微自己改下指令就行（逃**

这个过程中会出问题，就是并行的寄存器跑不完，后面单独处理一下就可以了（详见part2）

然后是基于上一部分简单的循环展开

令人疑惑的是，对于1024k阶方阵，对于编译器优化开关，-O比上一版本快了10.3%，而 -O3 却比上一个版本（也是O3）慢了0.12%，具体时间结果在part3

###然后是omp展开循环到多线程 这个导致我gettimeofday坏了，跑出来负数了（逃），只能手动秒表，因此没有小数据，这个和带着openblas的都没有4k以下的小数据。

最后是strassen优化，考虑分治但实际作用不大，因为有进出栈过程容易常数暴毙（实际上也是，使用了block的方法几乎没有对于上一种的优化，哪怕开了O3）

这个是对于算法复杂度的优化，对于n阶方阵，其时间复杂度为O(pow(n,2.8))，stl库说得好，数据小可以尝试复杂度更小的暴力，因此经过试验，对于128以下的分治出来的矩阵，直接走寻址优化返回，（因为多线程会导致错误结果，写保护又失去了多线程的意义。

Part 2. code

坏了要写不完了

```
struct Matrix *mul_plain(const struct Matrix *a, const struct Matrix *b,
struct Matrix *ans);          //ans=a*b,plain_version
struct Matrix *mul_ikj(const struct Matrix *a, const struct Matrix *b,
struct Matrix *ans);          //寻址优化
struct Matrix *mul_avx(const struct Matrix *a, struct Matrix *b, struct
Matrix *ans);          //avx指令集
//调用此后的功能请确保矩阵大小是2的整数次方（其实能整除八就行）
//可以在 matrix.c中调整defined ROLL实现其他大小循环展开
struct Matrix *mul_unroll(const struct Matrix *a, struct Matrix *b, struct
Matrix *ans);      //unroll
struct Matrix *mul_omp(const struct Matrix *a, struct Matrix *b, struct
Matrix *ans);          //omp
struct Matrix *mul_strassen(const struct Matrix *a, struct Matrix *b,
struct Matrix *ans, size_t size);    //strassen,会丢精度，不是omp的问题
```

Part 3. Result

```
规模，strassen，openblas

4k*4k 7.2 0.65

8k*8k 57.7 4.32

16k*16k 460.8 33.7

32k*32k 5068.8 269.5
```

Part 4. Difficulties & Solutions

1. strassen的精度损失来源于多次加减法，如果是无损数据类型就可以了，比如int

2. openblas好快，我优化了200倍，这玩意还比我快接近十倍，绝望qwq