easy_lab2

1. 矩阵乘法优化实现

定义宏方便后续代码的编写

```cpp
#define A(i,j) matrix1[i][j]
#define B(i,j) matrix2[i][j]
#define C(i,j) result_matrix[i][j]
```

1.1 单线程优化

1.1.1 Blocking（分块）

Blocking 是在总的数据量无法放进 Cache 的情况下，把总数据分成一个小块去访问，让每个 Tile 都可以在 Cache 中。具体的做法就是把一个内层循环分解成 outer loop * inner loop。然后把 outer loop 移到更外层去，从而确保 inner loop 的数据访问一定能在 Cache 中。

经过测试，在后续多线程优化的情况下只对 i 进行分块效率比较优秀，故只对 A 矩阵进行分块。块大小比较优秀的是 16 。

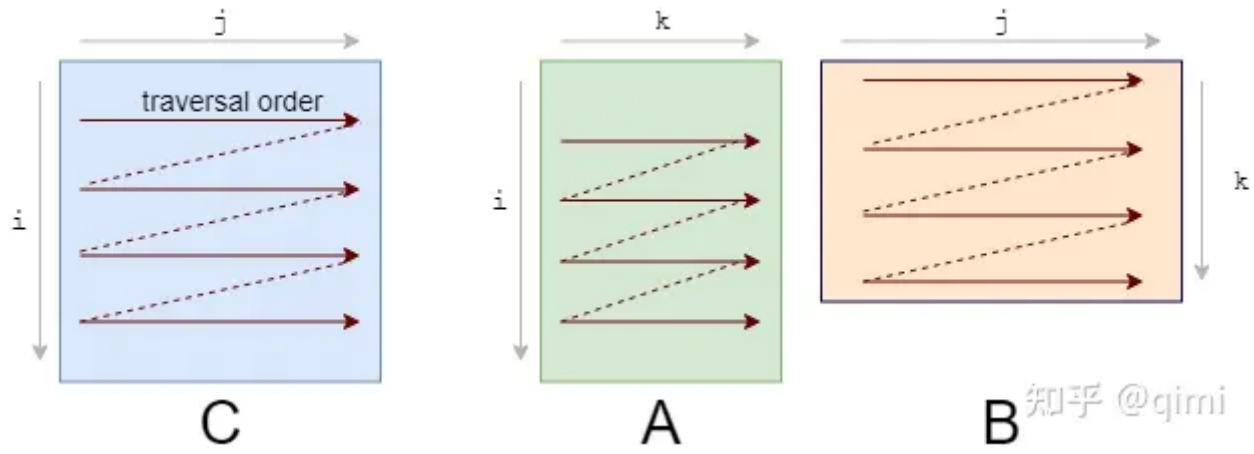
```cpp
for(int i_o=start_row;i_o<end_row;i_o+=16){

    for (int i = i_o; i< i_o+16 && i < end_row; i++) {
    //其他部分
    }
```

1.1.2 Loop Permutation（调整分块的计算顺序）

代码示例如下：

```cpp
for (int i = 0; i < M; i++)
  for (int k = 0; k < K; k++)
    for (int j = 0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j]
```

循环重排之后，对于 A，B，C 的空间访问局部性都很得到了保证。



1.1.3 1×4 的优化

1. 索引`j`每8个浮点操作（4个乘法+加法）只需要更新一次；
2. 每个元素 `A(i,j)` 只需从内存中取出 1 次，之后都在 L2 cache 中，直到后面的数填满了 L2 cache。

```cpp
for (int i = i_o; i< i_o+16 && i < end_row; i++)
        for (int k = 0; k < M; k+=4) {
            double s= A(i,k);
            double s1 = A(i,k+1);
            double s2 = A(i,k+2);
            double s3 = A(i,k+3);

            for (int j = 0; j < P; j++) {
                C(i,j) += s * B(k,j)+s1 * B(k+1,j)+s2
*B(k+2,j)+s3*B(k+3,j);
            }

        }
```

1.1.4 Write cache for blocks（为写操作数据块创建连续内存缓存）

因为分块后，程序将结果逐块写入 C，访问模式不是顺序的。 所以我们可以使用一个顺序缓存数组来保存块结果并在所有块的结果准备好时写入 C。

```cpp
 for(int j=0;j<P;j++)
        {
            temp[j]=C(i,j);
```

```
        }
        for (int k = 0; k < M; k+=4) {
            double s= A(i,k);
            double s1 = A(i,k+1);
            double s2 = A(i,k+2);
            double s3 = A(i,k+3);

            for (int j = 0; j < P; j++) {
                temp[j] += s * B(k,j)+s1 * B(k+1,j)+s2
*B(k+2,j)+s3*B(k+3,j);
            }
        }

        for(int j=0;j<P;j++)
        {
            C(i,j)=temp[j];
        }
```

完整代码如下：

```Cpp
void matrix_multiply_thread(double matrix1[N][M], double
matrix2[M][P], double result_matrix[N][P], int
start_row, int end_row) {
   for(int i_o=start_row;i_o<end_row;i_o+=16){
    for (int i = i_o; i< i_o+16 && i < end_row; i++) {
        double temp[3200];
        for(int j=0;j<P;j++)
        {
            temp[j]=C(i,j);
        }
        for (int k = 0; k < M; k+=4) {
            double s= A(i,k);
            double s1 = A(i,k+1);
            double s2 = A(i,k+2);
            double s3 = A(i,k+3);

            for (int j = 0; j < P; j++) {
                temp[j] += s * B(k,j)+s1 * B(k+1,j)+s2
*B(k+2,j)+s3*B(k+3,j);
            }
```

```
            }

            for(int j=0;j<P;j++)
            {
                C(i,j)=temp[j];
            }
        }
    }

    }
```

1.2 多线程优化

使用 C++ 11自带的 thread 线程库可以很方便的完成多线程代码的优化

```Cpp
void matrix_multiplication(double matrix1[N][M], double
matrix2[M][P], double result_matrix[N][P])
{

    const int num_threads = 64;

    std::vector<std::thread> threads;

    int rows_per_thread = N / num_threads;

    for (int i = 0; i < num_threads; i++) {
        int start_row = i * rows_per_thread;

        int end_row = (i == num_threads - 1) ? N : (i +
1) * rows_per_thread;


threads.emplace_back(std::thread(matrix_multiply_thread,
matrix1, matrix2, result_matrix, start_row, end_row));
    }

    for (auto& th : threads) th.join();
}
```

1.3 指令集优化

x86平台的 SIMD 指令优化主要有 SSE/AVX 扩展指令集，以此实现向量化计算。其中，avx512指令中 `__m512d` 可以保存8个双精度浮点共计512位数据。

同时，FMA 指令集（Fused-Multiply-Add，即融合乘加运算）`__m512 _mm512_fmadd_pd (__m512d a, __m512d b, __m512d c)` 结合了乘法和加法运算，可以通过单一指令完成乘加运算，以此提高了计算效率。

优化代码：

```cpp
        for (int i_o = start_row; i_o < end_row; i_o += 16)
        {
            for (int i = i_o; i < i_o + 16 && i < end_row; i++)
            {
                double temp[P] = {0};

                for (int k = 0; k < M; k++)
                {
                    __m512d vecA = _mm512_set1_pd(A(i, k));

                    for (int j = 0; j < P; j += 8)
                    {
                        __m512d vecC = _mm512_loadu_pd(&temp[j]);
                        __m512d vecB = _mm512_loadu_pd(&B(k, j));

                        vecC = _mm512_fmadd_pd(vecA, vecB, vecC);

                        _mm512_storeu_pd(&temp[j], vecC);
                    }
                }

                for (int j = 0; j < P; j++)
                {
                    C(i, j) = temp[j];
```

```
                    }
                }
            }
        }
```

性能对比如下：

	未使用指令集		使用指令集	
矩阵大小	运行时间(us微秒)	GFlops	运行时间(us微秒)	GFlops
(512*512) x (512*512)	18642.476600	14.399131	11418.388400	23.509049
(1024*1024) x (1024*1024)	89957.867600	23.872105	49433.972600	43.441454
(2048*2048) x (2048*2048)	566555.673200	30.323356	340279.420000	50.487535
(2560*2560) x (2560*2560)	1140927.590200	29.409782	706092.245400	47.521315
(3072*3072) x (3072*3072)	--	--	1228115.044200	47.212237

2. 实验中遇到的困难

1. AMD 处理器没有 AVX 512 指令集，只能借同学的 Intel 的电脑Debug
2. 在分片的时候因为没有处理好经常会发生数组越界的问题
3. 在前期单线程优化的过程中，不知道是什么原因在本地环境中测试 GFlops 的提升十分不明显，挫败感很强
4. 一开始不知道怎么同时进行多线程和分片，在 Chatgpt 的帮助下最终还是顺利完成了
5. C++17 标准下不能使用寄存器，参考 GEMM wiki 的代码实现十分不方便

3. 思考

O2 和 O3 优化都十分厉害，在进行了多线程和单线程优化的情况下速度还是被 O2 和 O3 优化爆杀，在未来的学习中，可以参考 O2 和 O3 优化下编译器做了什么事情，进一步优化代码。

4. Bonus

1. 在 Intel 的 SSE3指令集中，包含了一个神奇的指令：可以在一个时钟周期内完成两次浮点数的乘法和两次浮点数的加法。在提升了 cache 命中率和采用多线程计算

后，这个指令可以很好的帮助我们进一步地提升矩阵乘法的性能，并且使用非常简单，可以参考[1]、[2]、[3]。请给出使用 SSE3指令集前后的性能对比和代码。
见 1.3 指令集优化指令集优化部分

2. 为什么有时候多线程性能反而不如单线程？在什么情况下会导致这样的情况？
主要原因包括：

1、**线程管理开销**：创建和销毁线程需要时间和资源。如果任务本身非常轻量，线程管理的开销可能会超过并行执行带来的收益。

2、**上下文切换**：当线程数量超过处理器核心数时，系统需要频繁进行线程上下文切换，这会产生额外开销。

3、**资源竞争和同步**：多线程环境下，线程间可能需要访问和修改共享资源，导致必须使用锁或其他同步机制，这会增加等待时间。

4、**内存访问模式**：不合理的内存访问模式可能导致缓存未命中和内存带宽问题，尤其是在数据集较大时。

5、**假共享**：当多个线程频繁访问和修改相邻的内存位置时，可能会导致缓存行无效化，增加内存访问延迟。

3. 矩阵乘法是否会出现频繁的内存缺页？如何解决这样的问题？
矩阵乘法在处理大型矩阵时可能会出现内存缺页，尤其是当矩阵大小超过物理内存容量时。解决方法包括：

1、**数据分块**：将矩阵分割成小块，每次只处理一部分数据，以保证数据能够适合于缓存。

2、**优化内存访问顺序**：按照内存访问的局部性原理，调整算法以优化数据访问模式。

3、**使用交换空间**：如果物理内存不足，可以使用交换空间，但这会显著降低性能。

4. 尝试使用 GPU 进行矩阵运算，CPU 和 GPU 运算各有什么特点？为什么 GPU 矩阵运算远远快于 CPU?
CPU 特点：

- **通用性强**：适用于各种类型的计算任务。
- **核心数较少**：但每个核心性能更强，适合复杂的逻辑处理。
- **高频率、高缓存**：适合处理需要大量逻辑判断和少量数据的任务。

GPU 特点：

- **专用于并行计算**：拥有大量计算核心，适合同时执行大量相同的计算任务。
- **核心数多但频率较低**：每个核心不如 CPU 强，但数量众多。
- **适合大数据量的计算**：如图形处理、科学计算，以及矩阵运算。

为什么 GPU 矩阵运算远快于 CPU?

- **并行处理能力强**：GPU 有成百上千个核心，可以同时处理大量数据，非常适合执行矩阵运算这样的并行任务。
- **内存带宽更大**：GPU 拥有更高的内存带宽，可快速处理大规模数据集。
- **计算优化**：许多 GPU 针对浮点运算和向量运算进行了优化。