Easy lab 实验报告

实验内容

矩阵乘法是一个有着广泛应用的基础算法，比如神经网络的核心计算任务就是矩阵乘法，因此如何提升矩阵乘法的效率是一个值得深入思考的问题。

本次Lab中我们就来运用课上的知识来代码实现加速矩阵乘法，并且将最终的实现代码提交至评测平台，顺利通过平台上的测试。

加速的方式有很多，你提交至评测系统的代码实现仅限于提高cache命中率、多线程两个方面的加速。

此外，鼓励大家自行探索更多的加速方法（见bonus）。

我的实现

本次lab的主要实现代码集中在 `multiply.cpp` 中。

头文件引入

```
1 #include "multiply.h"
2 #include <cstdio>
3 #include <thread>
4 #include <immintrin.h>
5 #include <algorithm>
```

- `#include "multiply.h"`：包含矩阵乘法程序的头文件，包含矩阵大小（`N`、`M`、`P`）常量的声明。
- `#include <cstdio>`：包含标准输入输出头文件。
- `#include <thread>`：包含C++线程库，用于多线程支持。
- `#include <immintrin.h>`：包含Intel内部函数的头文件，提供AVX-512指令集支持。
- `#include <algorithm>`：包含C++标准算法库。

定义宏和结构体

```
1 #define BLOCK_SIZE 64*64
```

```
2
3  typedef struct {
4      int thread_id;
5      double (*matrix1)[M];
6      double (*matrix2)[P];
7      double (*result_matrix)[P];
8  } ThreadData;
```

- `#define BLOCK_SIZE 32*32`：定义一个宏，表示在矩阵乘法中使用的块大小。

  > BLOCK_SIZE需要根据目标系统的cache大小来动态调整（特别是level-1 cache），我在进行过多次尝试（包括8*8，16*16，32*32，64*64，128*128等）后，发现64*64的GFLOPS表现最佳。

- `typedef struct {...} ThreadData;`：定义一个结构体，用于存储线程数据，包括线程ID和矩阵指针。

AVX-512优化的块矩阵乘法函数

```
1  void block_matrix_multiply_avx512(double (*matrix1)[M], double (*matrix2)[P],
   double (*result_matrix)[P], int row_start, int row_end, int col_start, int
   col_end,int mid_start,int mid_end) {
2      for (int i = row_start; i < row_end; i++) {
3          double *temp_rm = result_matrix[i];
4          for (int k = mid_start; k < mid_end; k++) {
5              double temp_m1 = matrix1[i][k];
6              double *temp_m2 = matrix2[k];
7
8              int j = col_start;
9              int no_need = (col_end - j) % 8;
10             for(int next = 0; next < no_need; next++, j++)
11                 temp_rm[j] += temp_m1 * temp_m2[j];
12
13             for (; j < col_end; j += 8) {
14                 __m512d m1 = _mm512_set1_pd(temp_m1);
15                 __m512d m2 = _mm512_loadu_pd(temp_m2 + j);
16                 __m512d rm = _mm512_loadu_pd(temp_rm + j);
17                 __m512d a = _mm512_mul_pd(m1, m2);
18                 __m512d b = _mm512_add_pd(a, rm);
19                 _mm512_storeu_pd(temp_rm + j, b);
20             }
21         }
22     }
23 }
```

- `void block_matrix_multiply_avx512(...)`：这个函数实现了使用AVX-512优化的**块状矩阵乘法**。它逐行遍历，使用AVX-512指令进行向量化计算。

多线程执行函数

```
void* block_matrix_multiply(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    int NUM_THREADS = std::thread::hardware_concurrency();
    int rows_per_thread = (N + NUM_THREADS - 1) / NUM_THREADS;
    int start_row = data->thread_id * rows_per_thread;
    int end_row = std::min(start_row + rows_per_thread, N);

    for (int i = start_row; i < end_row; i += BLOCK_SIZE) {
        int block_i_end = std::min(i + BLOCK_SIZE, end_row);
        for (int j = 0; j < P; j += BLOCK_SIZE) {
            int block_j_end = std::min(j + BLOCK_SIZE, P);
            for (int k = 0; k < M; k += BLOCK_SIZE) {
                int block_k_end = std::min(k + BLOCK_SIZE, M);
                block_matrix_multiply_avx512(data->matrix1, data->matrix2, data-
            }
        }
    }

    pthread_exit(NULL);
}
```

- `void* block_matrix_multiply(void* arg)`：这个函数用于在多线程环境下调用，**它根据线程ID计算每个线程负责的行范围，并调用上述的块矩阵乘法函数。**

主矩阵乘法函数

```
void matrix_multiplication(double matrix1[N][M], double matrix2[M][P], double re

    // Initialize the result matrix to zero
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            result_matrix[i][j] = 0.0;
        }
    }

    int NUM_THREADS = std::thread::hardware_concurrency();
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
```

```
13
14      for (int i = 0; i < NUM_THREADS; i++) {
15          thread_data[i].thread_id = i;
16          thread_data[i].matrix1 = matrix1;
17          thread_data[i].matrix2 = matrix2;
18          thread_data[i].result_matrix = result_matrix;
19          pthread_create(&threads[i], NULL, block_matrix_multiply, (void*)&thread_
20      }
21
22      for (int i = 0; i < NUM_THREADS; i++) {
23          pthread_join(threads[i], NULL);
24      }
25  }
```

- `void matrix_multiplication(...)`：这个函数用于**初始化结果矩阵，创建多个线程**。每个线程将调用 `block_matrix_multiply` 函数来执行矩阵乘法的一部分。
- 其使用 `pthread_create` 创建了多个线程，并在所有线程完成后使用 `pthread_join` 等待它们结束。

困难和思考

在第一次的easy lab实验中，经过多次尝试我完成了一个用户态线程切换的基本程序，对于操作系统的线程管理有了更加深刻的认识。

而在本次的easy lab2中，我尝试通过矩阵分块（blocking）和第一次lab学习到的多线程（multi-threading）方法来加速AI领域最常见的运算：矩阵运算操作。

作为人工智能专业的学生，我认为本次lab给我带来的收益是无法比拟的，也是其他的课程所无法提供的，能让我在掌握比较高层次和抽象的神经网络算法的同时对于底层的运算机制有所了解，进一步了解目前深度学习框架的优化方向和GPU的在大量简单并行运算上的巨大优势。

在这里，我想首先感谢徐梦炜老师和这个lab的负责TA杜嘉骏学长，是你们的努力才让北邮操作系统课程和这个lab都变得越来越好！🌺🌺🌺

在做这次lab时，群里有不少同学反应本次lab的门槛过高，即使在用上多线程和矩阵分块技术之后仍然远远达不到设定的标准。当然，我在一开始做的时候也遇到了这个问题，具体来说在初步实现了多线程和矩阵分块后我的GFLOPS只达到了7.4左右，并且与最基本的方法（GFLOPS=0.3）相比较的话，最主要的提升还是来自于多线程操作，因为服务器端有80个逻辑cpu，因此可以创建80个thread来同步进行计算，大大提高了计算速度。

但是，相比于SIMD（单指令多数据流）指令集架构来说，这些提升都显得有点微小了。由于在群里有同学反映SSE3指令集在使用后性能不降反升的情况，因此我在一开始选择了另外一位同学提到的AVX-512指令集，它是SIMD的一个分支，特别擅长与浮点数的加速并行计算。在使用了AVX512指令集架构

后，计算性能直接翻了一倍！终于达到了实验要求，由此可见在硬件底层进行优化的威力，也促使我更加有动力了解现有最先进的加速矩阵乘法都是如何实现的（如numpy）。

在实现lab代码和学习相关知识期间，通过和助教的沟通，我也对代码profiling和cache miss有了更加实际的认识，增强了我在未来编码过程中的实践能力。

另外，我发现了一个和本lab十分相关的在线电子书，由深度学习领域大牛Mu Li编写，给我的lab提供了很大帮助并且对于AI学习也很有帮助：3.2. Matrix Multiplication — Dive into Deep Learning Compiler 0.1 documentation

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。

使用SSE3指令集的代码如下（替换`block_matrix_multiply_avx512`）

```
#include <pmmintrin.h> // SSE3指令集头文件

void block_matrix_multiply_sse3(double (*matrix1)[M], double (*matrix2)[P], doub
    for (int i = row_start; i < row_end; i++) {
        double *temp_rm = result_matrix[i];
        for (int k = mid_start; k < mid_end; k++) {
            double temp_m1 = matrix1[i][k];
            double *temp_m2 = matrix2[k];

            int j = col_start;
            int no_need = (col_end - j) % 2;
            for(int next = 0; next < no_need; next++, j++)
                temp_rm[j] += temp_m1 * temp_m2[j];

            for (; j < col_end; j += 2) {
                __m128d m1 = _mm_set1_pd(temp_m1);
                __m128d m2 = _mm_loadu_pd(temp_m2 + j);
                __m128d rm = _mm_loadu_pd(temp_rm + j);
                __m128d a = _mm_mul_pd(m1, m2);
                __m128d b = _mm_add_pd(a, rm);
                _mm_storeu_pd(temp_rm + j, b);
            }
        }
    }
}
```

在使用了SSE3指令集之后，性能在本地相对于AVX-512指令集来说提升相差不多。在计算512*512矩阵乘法时GFLOPS维持在20左右。

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？
    - 线程开销：创建和管理线程本身有一定的开销。如果任务本身计算量不大，这些开销可能会超过并行计算带来的性能提升。
    - 资源竞争和同步：多线程运行时可能会发生资源竞争，如对共享数据的访问需要同步，这可能导致等待和阻塞，降低效率。
    - 内存访问模式：不合理的内存访问模式可能导致缓存效率低下，特别是在多线程环境下。
    - 硬件限制：硬件资源（如CPU核心数量）的限制也可能导致多线程性能不如预期。

    在本次lab中，我发现了多线程带来的计算速度提升并不与线程的数量成正比，并且随着线程数量的增加其带来的提升逐渐减少。

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？

    矩阵乘法可能会导致频繁的内存访问。如果矩阵过大而无法完全装入内存，可能会发生内存缺页。我们可以通过优化数据访问模式（如分块或重排矩阵）来减少内存缺页，总是确保矩阵的一部分在使用时保持在物理内存中，而不是频繁地从硬盘交换。

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？

    在使用GPU进行矩阵运算时，其主要特点与CPU相比在于高度并行化的计算能力。

    GPU含有大量的计算核心，能够同时处理成千上万的运算任务，这使得它在处理可以并行化的大规模数据计算（如矩阵运算）时特别有效。相比之下，CPU通常拥有较少的核心，每个核心的计算能力较强，更适合处理复杂的逻辑和顺序计算任务。

    因此，GPU在矩阵运算上远远快于CPU主要是由于其架构优势，即能够同时执行大量相同或类似的计算操作，有效利用并行处理大规模数据的能力。这种架构特点让GPU在处理矩阵乘法这类可高度并行化的任务时表现出色，大幅提升了计算效率。