组成原理矩阵乘法作业2实验报告

学号：2111460 姓名：张洋 指导老师：李涛老师、董前琨老师

**一、实验要求**

1、在Taishan服务器上完成，使用Putty等远程软件在校内登录使用，服务器IP：222.30.62.23，端口22，用户名stu+学号，默认密码123456。

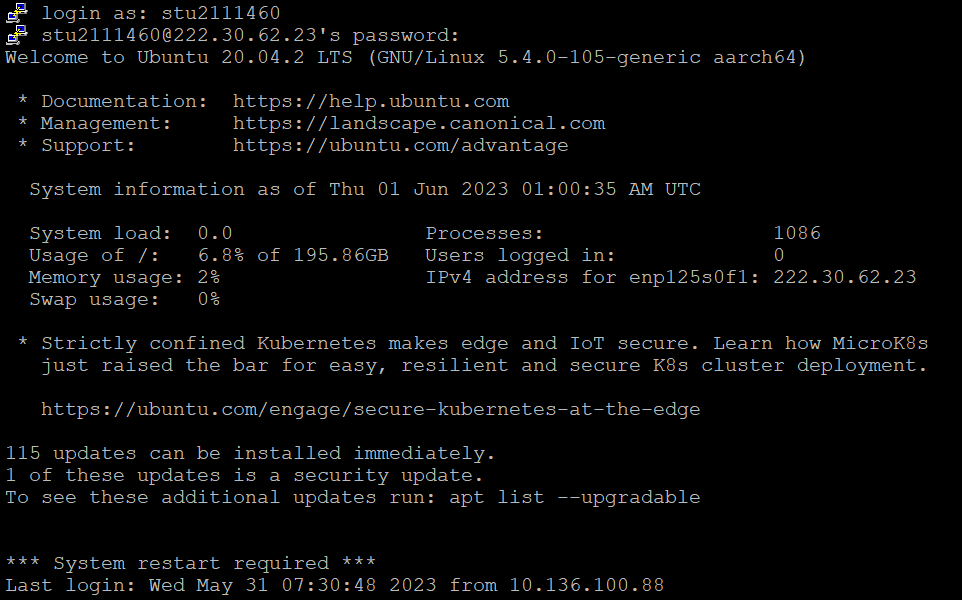
2、在完成矩阵乘法优化后（使用AVX库进行子字优化在Taishan服务器上的软件包环境不好配置，可以不进行此层次优化操作，注意原始代码需要调整），测试矩阵规模在1024~4096，或更大维度上，至少进行4个矩阵规模维度的测试。

3、在作业中需总结出不同层次，不同规模下的矩阵乘法优化对比，对比指标包括计算耗时、运行性能、加速比等。

4、在作业中需对比Taishan服务器和自己个人电脑上程序运行时间等相关指标，分析一下不同电脑上的运行差异的原因，总结在优化过程中遇到的问题和解决方式。

**二、实验步骤**

**1、登录泰山服务器**

****

**2、优化前的矩阵乘法**

|  |
| --- |
| #include <iostream>  #include <chrono>  using namespace std;  using namespace std::chrono;  void matrix\_multiply(int\*\* A, int\*\* B, int\*\* C, int N) {  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  C[i][j] = 0;  for (int k = 0; k < N; k++) {  C[i][j] += A[i][k] \* B[k][j];  }  }  }  }  int main() {  int N = 1024;  for (int n = 1; n <= 4; n++) {  N = N \* n;  cout << "矩阵规模：" << N << "\*" << N << endl;  int\*\* A = new int\* [N];  int\*\* B = new int\* [N];  int\*\* C = new int\* [N];  for (int i = 0; i < N; i++) {  A[i] = new int[N];  B[i] = new int[N];  C[i] = new int[N];  }  // initialize matrices A and B  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  A[i][j] = i + j;  B[i][j] = i - j;  }  }  auto start = high\_resolution\_clock::now();  matrix\_multiply(A, B, C, N);  auto stop = high\_resolution\_clock::now();  auto duration = duration\_cast<microseconds>(stop - start);  cout << "Time taken by function: " << duration.count() / 1000000 << " seconds" << endl;  cout << "Floating point operations per second: " << (double)N \* N \* N / duration.count() << endl;  for (int i = 0; i < N; i++) {  delete[] A[i];  delete[] B[i];  delete[] C[i];  }  delete[] A;  delete[] B;  delete[] C;  N = 1024;  }  } |

命令行输入：

g++ -o matrix0 matrix0.cpp

./matrix0

**3、分块矩阵乘法（因为源代码中优化部分都用到了AVX库，调用时会报错，所以这部分和下一部分中的代码全都是重新编写的）**

|  |
| --- |
| #include <iostream>  #include <chrono>  using namespace std;  using namespace std::chrono;  const int BLOCK\_SIZE = 32;  void matrix\_multiply(int\*\* A, int\*\* B, int\*\* C, int N) {  for (int i = 0; i < N; i += BLOCK\_SIZE) {  for (int j = 0; j < N; j += BLOCK\_SIZE) {  for (int k = 0; k < N; k += BLOCK\_SIZE) {  for (int ii = i; ii < min(i + BLOCK\_SIZE, N); ii++) {  for (int jj = j; jj < min(j + BLOCK\_SIZE, N); jj++) {  for (int kk = k; kk < min(k + BLOCK\_SIZE, N); kk++) {  C[ii][jj] += A[ii][kk] \* B[kk][jj];  }  }  }  }  }  }  }  int main() {  int N = 1024;  for (int n = 1; n <= 4; n++) {  N = N \* n;  cout << "矩阵规模：" << N << "\*" << N << endl;  int\*\* A = new int\* [N];  int\*\* B = new int\* [N];  int\*\* C = new int\* [N];  for (int i = 0; i < N; i++) {  A[i] = new int[N];  B[i] = new int[N];  C[i] = new int[N];  }  // initialize matrices A and B  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  A[i][j] = i + j;  B[i][j] = i - j;  }  }  auto start = high\_resolution\_clock::now();  matrix\_multiply(A, B, C, N);  auto stop = high\_resolution\_clock::now();  auto duration = duration\_cast<microseconds>(stop - start);  cout << "Time taken by function: " << duration.count() << " microseconds" << endl;  cout << "Floating point operations per second: " << (double)N \* N \* N / duration.count() << endl;  for (int i = 0; i < N; i++) {  delete[] A[i];  delete[] B[i];  delete[] C[i];  }  delete[] A;  delete[] B;  delete[] C;  N = 1024;  }  } |

命令行输入：

g++ -o matrix\_multiply matrix\_multiply.cpp

./matrix\_multiply

**4、并行计算的矩阵乘法**

|  |
| --- |
| vector<vector<int>> parallelMatrixMultiplication(const vector<vector<int>>& matrix1, const vector<vector<int>>& matrix2) {  int rows1 = matrix1.size();  int cols1 = matrix1[0].size();  int rows2 = matrix2.size();  int cols2 = matrix2[0].size();  if (cols1 != rows2) {  cout << "Error: The number of columns in matrix1 should be equal to the number of rows in matrix2." << endl;  return {};  }  vector<vector<int>> result(rows1, vector<int>(cols2, 0));  const int numThreads = thread::hardware\_concurrency();  vector<thread> threads;  // 创建线程  for (int t = 0; t < numThreads; t++) {  threads.emplace\_back([&, t]() {  for (int i = t; i < rows1; i += numThreads) {  for (int j = 0; j < cols2; j++) {  for (int k = 0; k < cols1; k++) {  result[i][j] += matrix1[i][k] \* matrix2[k][j];  }  }  }  });  }  // 等待线程完成  for (auto& thread : threads) {  thread.join();  }  return result;  } |

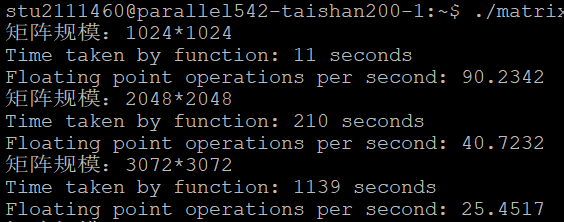
命令行输入：

g++ -o matrix\_multiply matrix\_multiply.cpp -O3 -march=native

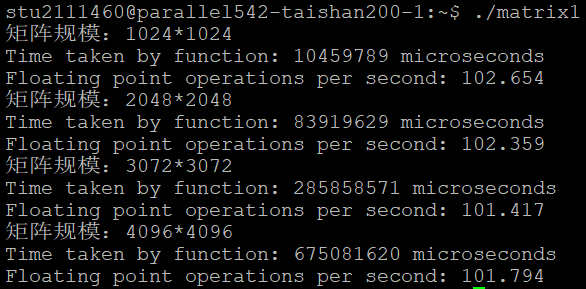
./matrix\_multiply

**三、运行结果**

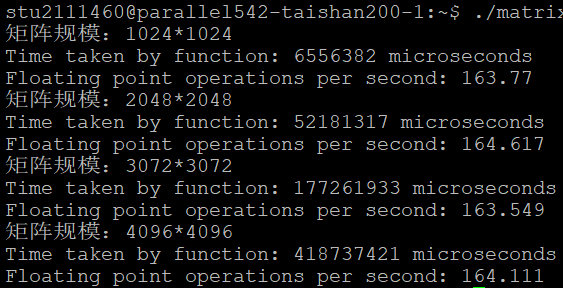
1、优化前的矩阵乘法



2、分块矩阵乘法



3、并行计算的矩阵乘法



**四、对比实验效果**

1、执行时间

2、加速比

3、运行性能

由运行截图可知，优化前的矩阵乘法计算耗时最长，运行性能最差。

执行时间的计算：按照优化前计算的矩阵乘法耗时最长，因为它需要执行三重循环来逐个元素进行计算。不同规模优化后的执行时间都有所下降，但是在测试的四种规模中，执行时间最快的优化方式都是多处理器并行的分块矩阵乘法。

加速比：优化的矩阵乘法和并行计算矩阵乘法相对于按照定义计算的矩阵乘法都能够取得较好的加速比。优化的矩阵乘法通过减少不必要的访存操作和利用分块矩阵乘法的优化策略，加速比相对较高。并行计算矩阵乘法通过利用多线程并行计算的特点，能够进一步提高计算速度，加速比最高。

运行性能：按照优化前计算的矩阵乘法的性能最低，因为它使用了三重循环的嵌套，导致计算复杂度较高。优化的矩阵乘法通过采用分块矩阵乘法的优化方法，减少了不必要的访存操作，提高了运行性能。并行计算矩阵乘法利用多线程实现并行计算，充分利用多核处理器的计算能力，因此具有更好的运行性能。

**五、与个人端比较差异分析**

经过实验，可以发现泰山服务器比个人端运行代码运行时间会更长，我总结了以下几点原因：

1.网络延迟：泰山服务器通常位于远程数据中心，而个人端通常位于本地网络中。因此，在使用泰山服务器时，需要通过网络传输数据和代码，这可能会导致网络延迟和传输速度变慢，从而影响代码的运行速度。

2.硬件配置：泰山服务器通常配备了多个CPU和大量内存，可以提供更高的计算能力和更快的数据处理速度。但是，如果代码无法充分利用这些硬件资源，那么使用泰山服务器可能不会比个人端更快。

3.软件环境：泰山服务器和个人端可能使用不同的操作系统和软件环境。如果代码依赖于特定的软件环境或库，那么在不同的环境中运行可能会导致性能差异。

4.代码优化：在泰山服务器上运行代码时，需要考虑代码的并行性和优化程度。如果代码无法充分利用多个CPU或内存，或者没有进行优化，那么在泰山服务器上运行可能不会比个人端更快。

因此，可以考虑优化代码、调整软件环境、减少网络传输等方式来提高性能。同时，也需要注意不同硬件和软件环境之间的差异，以便更好地利用泰山服务器的计算能力。