矩阵乘法实验报告

姓名：田晋宇 学号：2212039 班级：李涛班

实验要求

- 个人PC电脑实验要求如下：
  - 使用个人电脑完成，不仅限于visual studio、vscode等。
  - 在完成矩阵乘法优化后，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。如PC电脑有Nvidia显卡，建议尝试CUDA代码。
  - 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。
  - 在作业中总结优化过程中遇到的问题和解决方式。
- 在Taishan服务器上使用vim+gcc编程环境，要求如下：
  - 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，登录成功后可自行修改密码。
  - 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。
  - 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。
  - 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。

实验步骤

- **个人PC电脑**
  - **代码部分**

```
#include<iostream>
#include<time.h>
#include<cstdlib>
//#include<x86intrin.h>
#include<immintrin.h>

using namespace std;
#define REAL_T double

//计算并输出计算时间和每秒浮点运算次数
void printFlops(int A_height, int B_width, int B_height, clock_t start,
clock_t stop ){

    cout<<"Time:\t"<<(stop - start)/CLOCKS_PER_SEC<<"."<<(stop -
start)%CLOCKS_PER_SEC<<" s" << endl;

    REAL_T flops = ( 2.0 * A_height * B_width * B_height ) / 1E9 /((stop -
 start)/(CLOCKS_PER_SEC * 1.0));
    cout << "Performance:\t" << flops << " FLOPS" << endl << endl;
}
```

```
// 随机生成浮点数构造原始矩阵
void initMatrix( int n, REAL_T *A, REAL_T *B, REAL_T *C ){
    for( int i = 0; i < n; ++i )
        for( int j = 0; j < n; ++j ){
            A[i+j*n] = rand() / REAL_T(RAND_MAX);
            B[i+j*n] = rand() / REAL_T(RAND_MAX);
            C[i+j*n] = 0;
        }
}

// 拷贝矩阵
void copyMatrix(int n, REAL_T *S_A, REAL_T *S_B, REAL_T *S_C, REAL_T *D_A,
REAL_T *D_B, REAL_T *D_C){
    memcpy( D_A, S_A, n * n * sizeof(REAL_T) );
    memcpy( D_B, S_B, n * n * sizeof(REAL_T) );
    memcpy( D_C, S_C, n * n * sizeof(REAL_T) );
}

//用随机生成数初始化矩阵
void initMatrix(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            A[i + j * n] = rand() / REAL_T(RAND_MAX);
            B[i + j * n] = rand() / REAL_T(RAND_MAX);
            C[i + j * n] = 0;
        }
}

//定义算法
void dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            REAL_T cij = C[i + j * n];
            for (int k = 0; k < n; k++) {
                cij += A[i + k * n] * B[k + j * n];
            }
            C[i + j * n] = cij;
        }
}

//子字并行优化方法
void avx_dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; i += 4)
        for (int j = 0; j < n; ++j) {
            __m256d cij = _mm256_load_pd(C + i + j * n);
            for (int k = 0; k < n; k++) {
                //cij += A[i+k*n] * B[k+j*n];
                cij = _mm256_add_pd(
                    cij,
                    _mm256_mul_pd(_mm256_load_pd(A + i + k * n),
_mm256_load_pd(B + i + k * n))
                );
            }
```

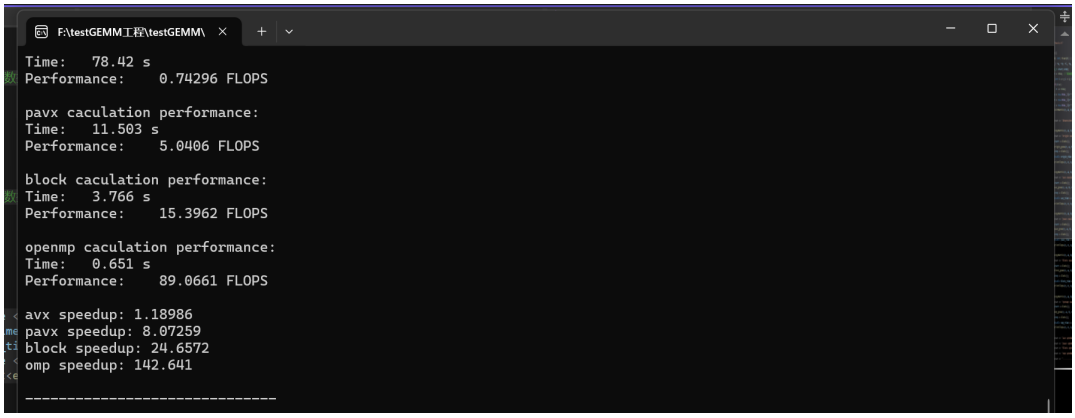
```
            _mm256_store_pd(C + i + j * n, cij);
        }
}

//指令并行的优化方法
#define UNROLL (4)
void pavx_dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; i += 4 * UNROLL)
        for (int j = 0; j < n; ++j) {
            __m256d cij[4];
            for (int x = 0; x < UNROLL; ++x)
                cij[x] = _mm256_load_pd(C + i + j * n);

            for (int k = 0; k < n; k++) {
                __m256d b = _mm256_broadcast_sd(B + k + j * n);
                for (int x = 0; x < UNROLL; ++x)
                    cij[x] = _mm256_add_pd(
                        cij[x],
                        _mm256_mul_pd(_mm256_load_pd(A + i + 4 * x + k * n),
b));
            }
            for (int x = 0; x < UNROLL; ++x)
                _mm256_store_pd(C + i + x * 4 + j * n, cij[x]);
        }
}

//分块的优化方法
#define BLOCKSIZE (32)
void do_block(int n, int si, int sj, int sk, REAL_T* A, REAL_T* B, REAL_T*
C) {
    for (int i = si; i < si + BLOCKSIZE; i += UNROLL * 4)
        for (int j = sj; j < sj + BLOCKSIZE; ++j) {
            __m256d c[4];
            for (int x = 0; x < UNROLL; ++x)
                c[x] = _mm256_load_pd(C + i + 4 * x + j * n);

            for (int k = sk; k < sk + BLOCKSIZE; ++k) {
                __m256d b = b = _mm256_broadcast_sd(B + k + j * n);
                for (int x = 0; x < UNROLL; ++x)
                    c[x] = _mm256_add_pd(
                        c[x],
                        _mm256_mul_pd(_mm256_load_pd(A + i + 4 * x + k * n),
b));
            }

            for (int x = 0; x < UNROLL; ++x)
                _mm256_store_pd(C + i + x * 4 + j * n, c[x]);
        }
}

//多处理器并行的优化方法
void block_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int sj = 0; sj < n; sj += BLOCKSIZE)
        for (int si = 0; si < n; si += BLOCKSIZE)
            for (int sk = 0; sk < n; sk += BLOCKSIZE)
```

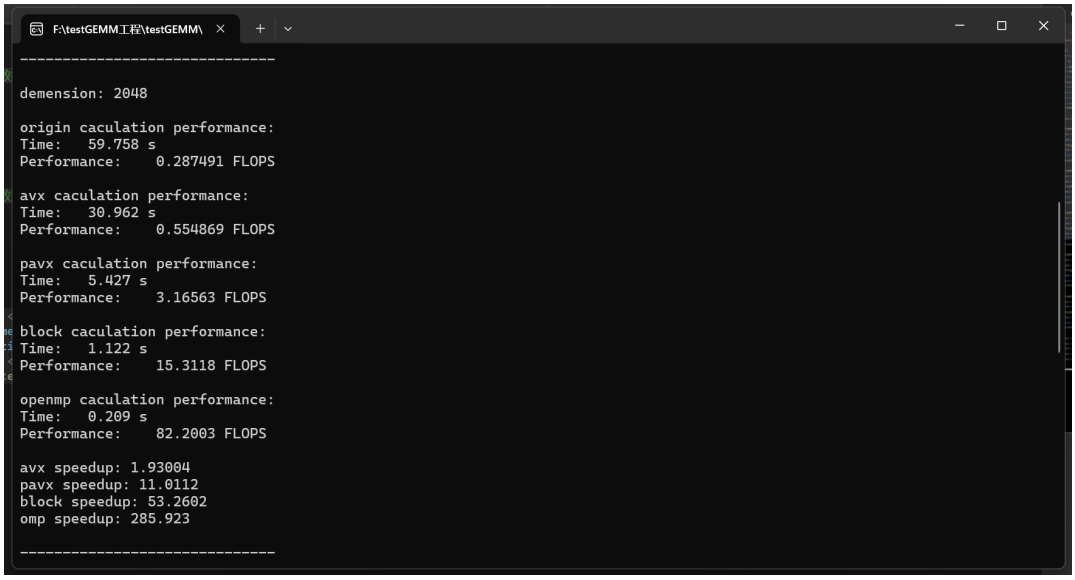
```
                do_block(n, si, sj, sk, A, B, C);
}

void omp_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
#pragma omp parallel for
    for (int sj = 0; sj < n; sj += BLOCKSIZE)
        for (int si = 0; si < n; si += BLOCKSIZE)
            for (int sk = 0; sk < n; sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}
int main(){
    srand( int( time(0) ) );
    REAL_T *A, *B, *C, *a, *b, *c;
    clock_t start,stop;
    int n = 1024; // 矩阵规模
    for (int i = 1;i < 5; i++) {
        if(i!=1)
          n += 1024;
        A = new REAL_T[n * n];  a = new REAL_T[n * n];
        B = new REAL_T[n * n];  b = new REAL_T[n * n];
        C = new REAL_T[n * n];  c = new REAL_T[n * n];
        initMatrix(n, A, B, C); //构造原始矩阵

        cout << "demension: " << n << endl<<endl;

        copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
        cout << "origin caculation performance: \n";
        start = clock();
        origin_gemm(n, a, b, c);
        stop = clock();
        double origin_time = stop - start;
        printFlops(n, n, n, start, stop);

        copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
        cout << "avx caculation performance: \n";
        start = clock();
        avx_gemm(n, a, b, c);
        stop = clock();
        double avx_time = stop - start;
        printFlops(n, n, n, start, stop);

        copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
        cout << "pavx caculation performance: \n";
        start = clock();
        pavx_gemm(n, a, b, c);
        stop = clock();
        double pavx_time = stop - start;
        printFlops(n, n, n, start, stop);

        copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
        cout << "block caculation performance: \n";
        start = clock();
        block_gemm(n, a, b, c);
        stop = clock();
        double block_time = stop - start;
```

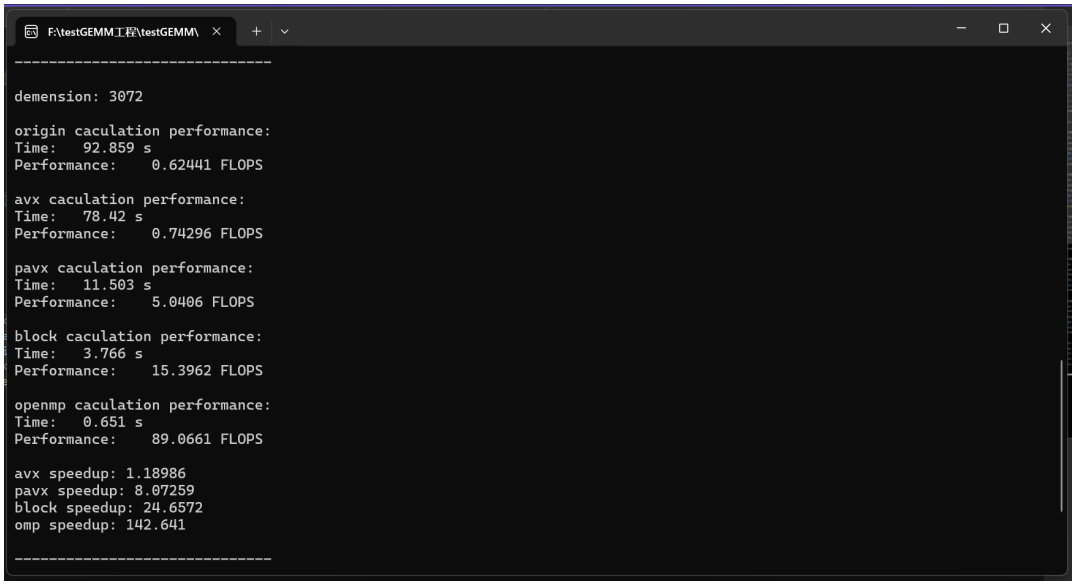
```
        printFlops(n, n, n, start, stop);

        copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
        cout << "openmp caculation performance: \n";
        start = clock();
        omp_gemm(n, a, b, c);
        stop = clock();
        double omp_time = stop - start;
        printFlops(n, n, n, start, stop);

        cout << "avx speedup: " << origin_time / avx_time << endl;
        cout << "pavx speedup: " << origin_time / pavx_time << endl;
        cout << "block speedup: " << origin_time / block_time << endl;
        cout << "omp speedup: " << origin_time / omp_time << endl<<endl;
        cout << "------------------------------" << endl<<endl;
    }
}
```

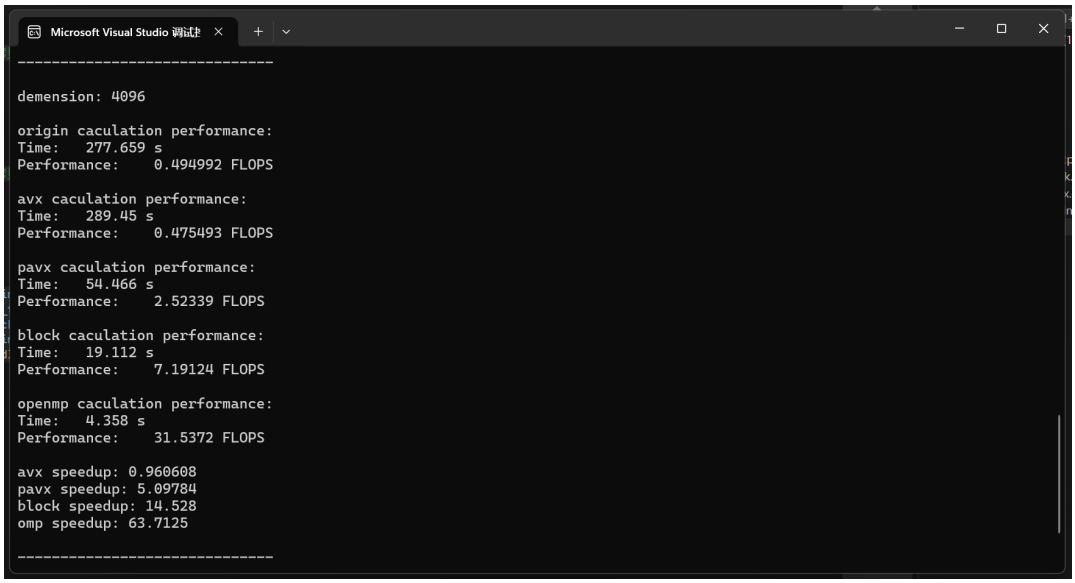
- **运行结果**

  在代码中分别对四个维度的矩阵进行了测试，得到的结果如下：









- **结果分析**
  - **计算耗时**

    按照定义的矩阵乘耗时最多，由于需要对每个元素进行遍历，多重循环后将得到的结果合并，复杂度较高；几种并行优化的方法大大提高了多核cpu的利用率，耗时有了明显的缩短。
  - **运行性能**

    此处的性能与计算耗时成反比，与计算耗时分析相同，不过多赘述。
  - **加速比**

    几种优化方法是层层递进的关系，通过原始定义方法的计算耗时与优化后的计算耗时作比来得到加速比，从一开始的只有一左右的加速比，到最后可以达到即使甚至一百的加速比，性能有了大幅提升。
- **总结**

  1. **按照定义的GEMM**

     原始的计算方法运用了多层嵌套导致计算的时空复杂度较高，当维度达到4096时，时间甚至达到半小时，对cpu的多核算力是一种极大地浪费，因此我们考虑通过并行的方式来进行优化，减少不必要的计算和访存操作，提高cpu的利用率，从而提高计算效率。
  2. **子字并行的GEMM**

利用AVX指令集中的256位向量化指令来并行化矩阵乘法运算。每次同时处理4个数据元素，提高了计算效率。

3. **指令并行的GEMM**

   在AVX指令集优化的基础上，进一步利用指令级并行性，通过展开循环和使用广播加载等技术来提高性能。原始的是多发射乱序执行。循环展开，复制4份后，gcc编译器优化时会将后面的三次循环进行指令级并行，B[i] [j]可以在四次循环中反复使用，因此broadcast一份即可。

4. **分块的GEMM**

   将矩阵分成小块，分块处理可以减少缓存未命中带来的性能损失。通过在每个块上应用AVX指令集和指令级并行优化，提高了计算效率。

5. **多处理器并行的GEMM**

   使用OpenMP库中的指令来实现并行化。通过在循环上添加`#pragma omp parallel for`指令，将循环中的迭代任务分配给多个线程并行处理，从而加快矩阵乘法的计算速度。

- **感想**
  - 对于并行的优化方法有了一定的认识。
  - 对于一个C++的各个组成文件以及作用有了更深刻的了解。
  - 使用OpenMP多核处理器并行优化矩阵乘法代码时，对于项目不同的配置可能会影响程序的性能和行为。

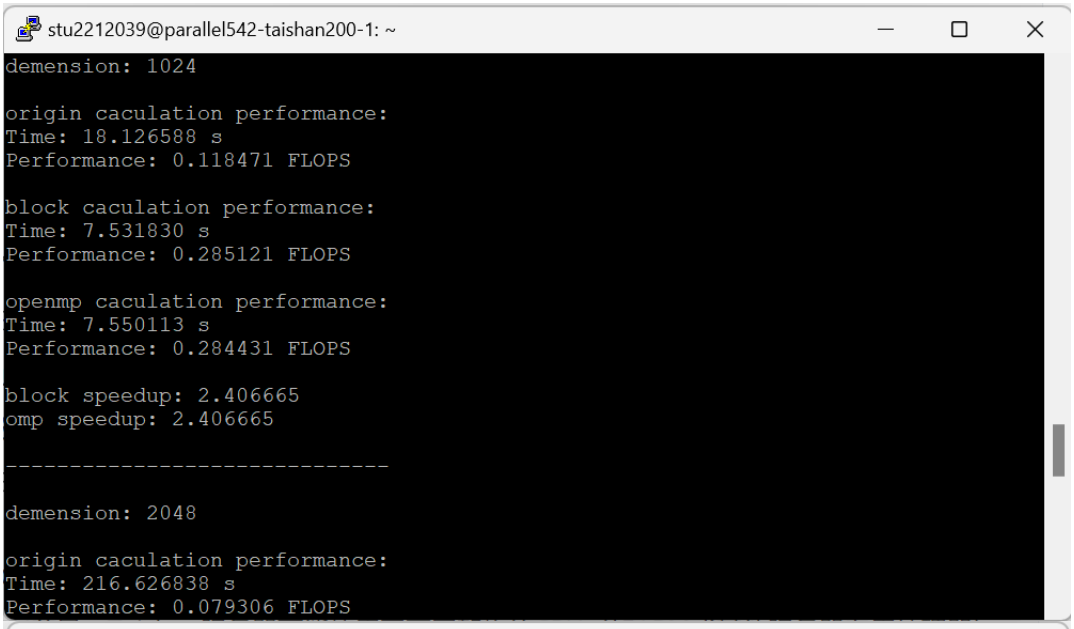
- **Taishan服务器**
  - 代码部分

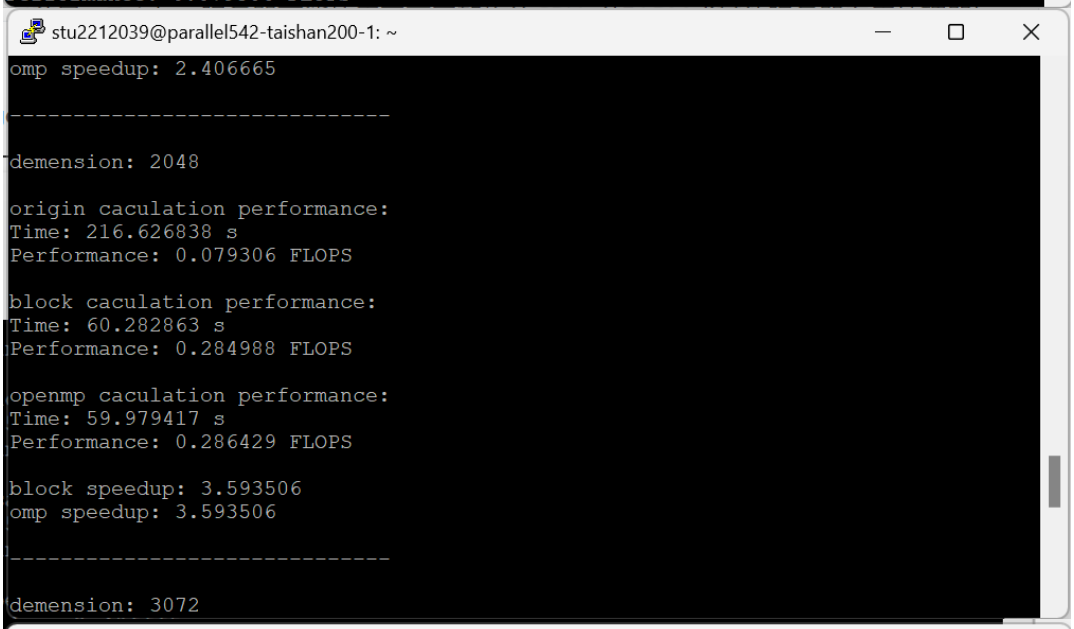
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>
#include<omp.h>

#define REAL_T double
#define UNROLL 4
#define BLOCK_SIZE 32

//计算并输出计算时间和每秒浮点运算次数
void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t stop) {
    printf("Time: %ld.%ld s\n", (stop - start) / CLOCKS_PER_SEC, (stop - start) % CLOCKS_PER_SEC);

    REAL_T flops = (2.0 * A_height * B_width * B_height) / 1E9 / ((stop - start) / (CLOCKS_PER_SEC * 1.0));
    printf("Performance: %f FLOPS\n\n", flops);
}


// 随机生成浮点数构造原始矩阵
void initMatrix(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            A[i + j * n] = rand() / (REAL_T)RAND_MAX;
```

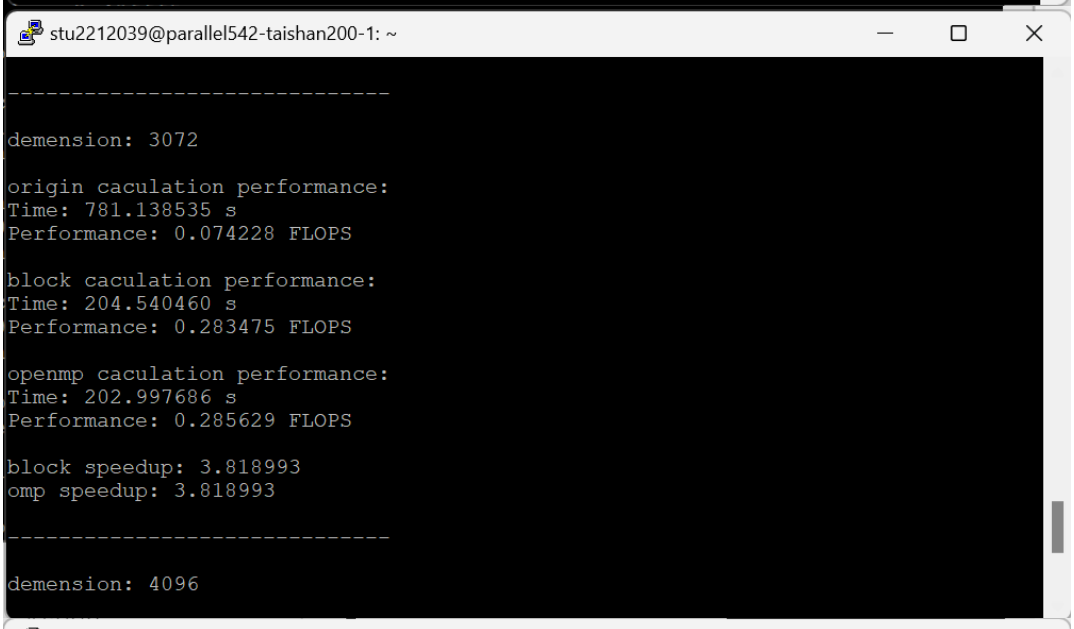
```
            B[i + j * n] = rand() / (REAL_T)RAND_MAX;
            C[i + j * n] = 0;
        }
}

//定义算法
void origin_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            REAL_T cij = C[i + j * n];
            for (int k = 0; k < n; k++) {
                cij += A[i + k * n] * B[k + j * n];
            }
            C[i + j * n] = cij;
        }
}

//多处理器并行的优化方法
void block_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    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 < i + BLOCK_SIZE; ++ii) {
                    for (int jj = j;jj < j + BLOCK_SIZE; ++jj) {
                        REAL_T cij = C[ii + jj * n];
                        for (int kk = k;kk < k + BLOCK_SIZE;++kk) {
                            cij += A[ii + kk * n] * B[kk + jj * n];
                        }
                        C[ii + jj * n] = cij;
                    }
                }
            }
        }
    }
}

void omp_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
#pragma omp parallel for collapse(2) default(none) shared(A,B,C,n)
    for (int j = 0; j < n; j += BLOCK_SIZE) {
        for (int i = 0; i < n; i += BLOCK_SIZE) {
            for (int k = 0; k < n; k += BLOCK_SIZE) {
                for (int ii = i;ii < i + BLOCK_SIZE; ++ii) {
                    for (int jj = j;jj < j + BLOCK_SIZE; ++jj) {
                        REAL_T cij = C[ii + jj * n];
                        for (int kk = k;kk < k + BLOCK_SIZE;++kk) {
                            cij += A[ii + kk * n] * B[kk + jj * n];
                        }
                        C[ii + jj * n] = cij;
                    }
                }
            }
        }
    }
}
```

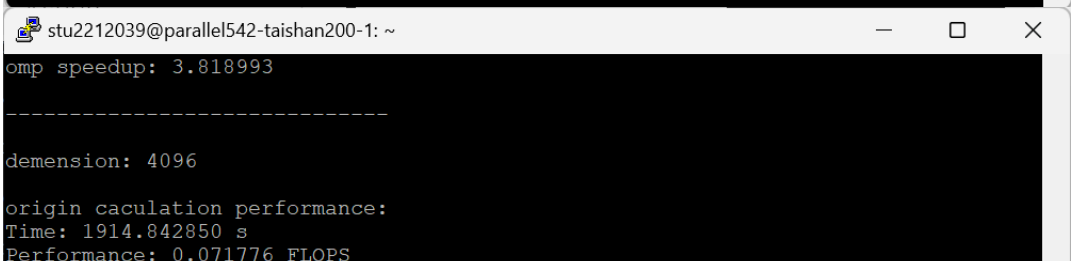
```
int main() {
    srand((unsigned int)time(NULL));
    REAL_T* A, * B, * C;
    clock_t start, stop;
    int n = 1024; // 矩阵规模
    for (int i = 1;i < 5; i++) {
        if (i != 1)
            n += 1024;
        A = (REAL_T*)malloc(n * n * sizeof(REAL_T));
        B = (REAL_T*)malloc(n * n * sizeof(REAL_T));
        C = (REAL_T*)malloc(n * n * sizeof(REAL_T));
        //构造原始矩阵
        printf("demension: %d\n\n", n);

        initMatrix(n, A, B, C); // 从原始矩阵拷贝数据
        printf("origin caculation performance:\n");
        start = clock();
        origin_gemm(n, A, B, C);
        stop = clock();
        double origin_time = stop - start;
        printFlops(n, n, n, start, stop);

        initMatrix(n, A, B, C); // 从原始矩阵拷贝数据
        printf("block caculation performance:\n");
        start = clock();
        block_gemm(n, A, B, C);
        stop = clock();
        double block_time = stop - start;
        printFlops(n, n, n, start, stop);

        initMatrix(n, A, B, C); // 从原始矩阵拷贝数据
        printf("openmp caculation performance:\n");
        start = clock();
        omp_gemm(n, A, B, C);
        stop = clock();
        double omp_time = stop - start;
        printFlops(n, n, n, start, stop);

        printf("block speedup: %f\n", origin_time / block_time);
        printf("omp speedup: %f\n\n", origin_time / block_time);
        printf("------------------------------\n\n");

        free(A);
        free(B);
        free(C);
    }
}
```

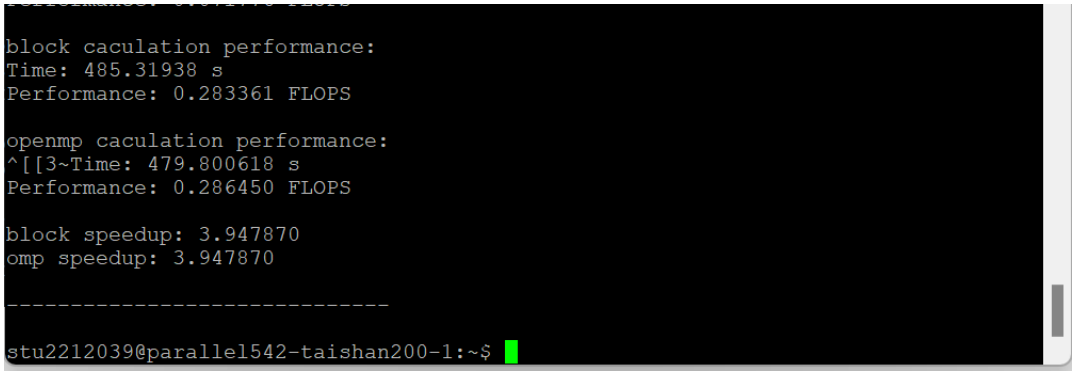
- **运行结果**


stu2212039@parallel542-taishan200-1: ~
demension: 1024
origin caculation performance:
Time: 18.126588 s
Performance: 0.118471 FLOPS
block caculation performance:
Time: 7.531830 s
Performance: 0.285121 FLOPS
openmp caculation performance:
Time: 7.550113 s
Performance: 0.284431 FLOPS
block speedup: 2.406665
omp speedup: 2.406665
demension: 2048
origin caculation performance:
Time: 216.626838 s
Performance: 0.079306 FLOPS


stu2212039@parallel542-taishan200-1: ~
omp speedup: 2.406665
demension: 2048
origin caculation performance:
Time: 216.626838 s
Performance: 0.079306 FLOPS
block caculation performance:
Time: 60.282863 s
Performance: 0.284988 FLOPS
openmp caculation performance:
Time: 59.979417 s
Performance: 0.286429 FLOPS
block speedup: 3.593506
omp speedup: 3.593506
demension: 3072


stu2212039@parallel542-taishan200-1: ~
demension: 3072
origin caculation performance:
Time: 781.138535 s
Performance: 0.074228 FLOPS
block caculation performance:
Time: 204.540460 s
Performance: 0.283475 FLOPS
openmp caculation performance:
Time: 202.997686 s
Performance: 0.285629 FLOPS
block speedup: 3.818993
omp speedup: 3.818993
demension: 4096


stu2212039@parallel542-taishan200-1: ~
omp speedup: 3.818993
demension: 4096
origin caculation performance:
Time: 1914.842850 s



- **总结**

  1. 开启了多处理器并行优化后并没有什么显著变化，分析原因可能有以下几点：
     - 在并行过程中运用了深层嵌套，限制了并行化的效果。
     - 负载不平衡，可能导致某些县城计算较快，而其他线程仍在工作。
     - 服务器上的硬件资源可能未被完全调用起来。
  2. 学会在远程服务器上使用vi编辑，gcc编译运行C++程序，为以后对远程服务器的使用打下了良好的基础。
  3. 个人PC与服务器产生差异的原因：
     - CPU差异：Taishan服务器可能使用了不同架构的CPU（如ARM或其他服务器专用处理器），这些处理器在单核性能上可能不如专为高性能计算优化的桌面CPU。这可能导致即便在多核利用率高的情况下，单核运算性能仍然低于个人电脑。
     - 内存带宽和延迟：服务器通常配置有更高容量且更注重错误纠正的内存，但这些内存在速度和延迟上可能不如专为高速计算设计的桌面内存。
     - 操作系统和优化：服务器操作系统可能更注重稳定性和安全性，而不是像个人电脑操作系统那样优化性能。此外，编译器和编译选项在服务器上可能未能充分利用其硬件特性。
     - 并行计算优化：虽然服务器具备强大的多核处理能力，但如果并行算法和线程管理未针对其具体架构优化，就不能发挥出最佳性能。
     - 代码通用性与特定性：代码没有针对Taishan服务器的特定硬件进行优化，没有使用特定的向量指令集或调整内存访问模式以适配其缓存架构。