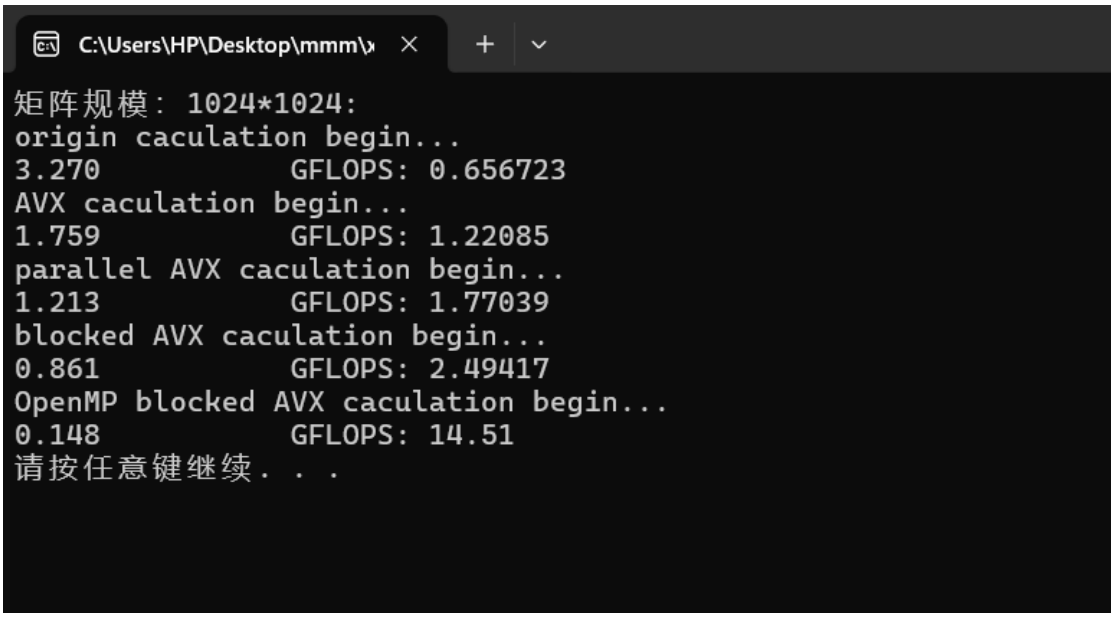
矩阵乘法优化作业

学号：2210737　　姓名：阿斯雅

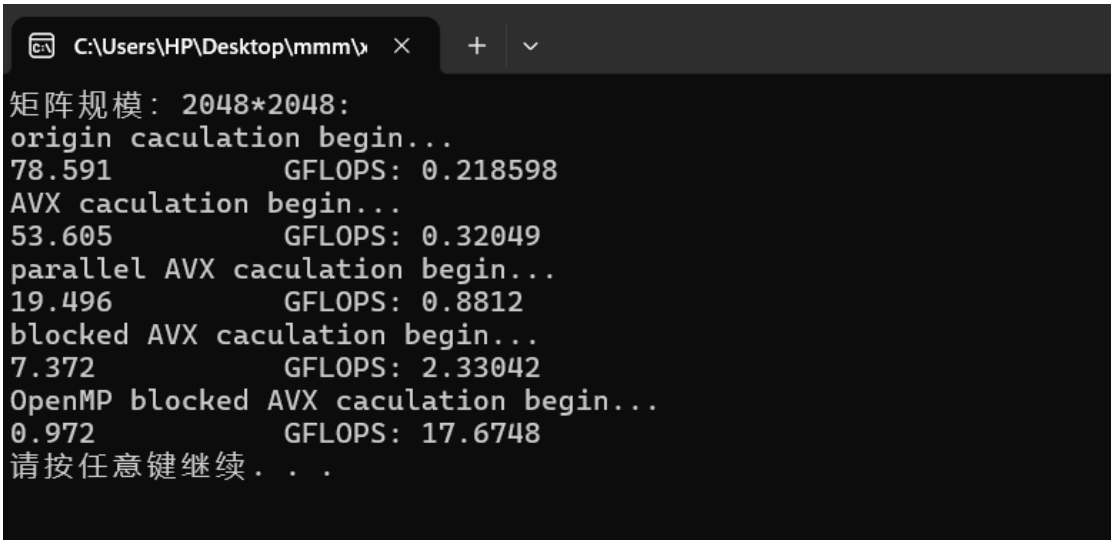
一、 在电脑上 VS 运行源代码

```
void main()
{
    REAL_T* A, * B, * C;
    clock_t start, stop;
    int n = 1024;
    cout << "矩阵规模：" << n << "*" << n << ":" << endl;
    A = new REAL_T[n * n];
    B = new REAL_T[n * n];
    C = new REAL_T[n * n];
    initMatrix(n, A, B, C);

    cout << "origin caculation begin...\n";
    start = clock();
    dgemm(n, A, B, C);
    stop = clock();
    cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) % CLOCKS_PER_SEC << "\t\t";
    printFlops(n, n, n, start, stop);

    initMatrix(n, A, B, C);
    cout << "AVX caculation begin...\n";
    start = clock();
    avx_dgemm(n, A, B, C);
    stop = clock();
    cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) % CLOCKS_PER_SEC << "\t\t";
    printFlops(n, n, n, start, stop);

    initMatrix(n, A, B, C);
    cout << "parallel AVX caculation begin...\n";
    start = clock();
    pavx_dgemm(n, A, B, C);
    stop = clock();
    cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) % CLOCKS_PER_SEC << "\t\t";
    printFlops(n, n, n, start, stop);

    initMatrix(n, A, B, C);
    cout << "blocked AVX caculation begin...\n";
    start = clock();
    block_gemm(n, A, B, C);
    stop = clock();
    cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) % CLOCKS_PER_SEC << "\t\t";
    printFlops(n, n, n, start, stop);
    initMatrix(n, A, B, C);
    cout << "OpenMP blocked AVX caculation begin...\n";
    start = clock();
    omp_gemm(n, A, B, C);
    stop = clock();
    cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) % CLOCKS_PER_SEC << "\t\t";
    printFlops(n, n, n, start, stop);
    system("pause");
}
```

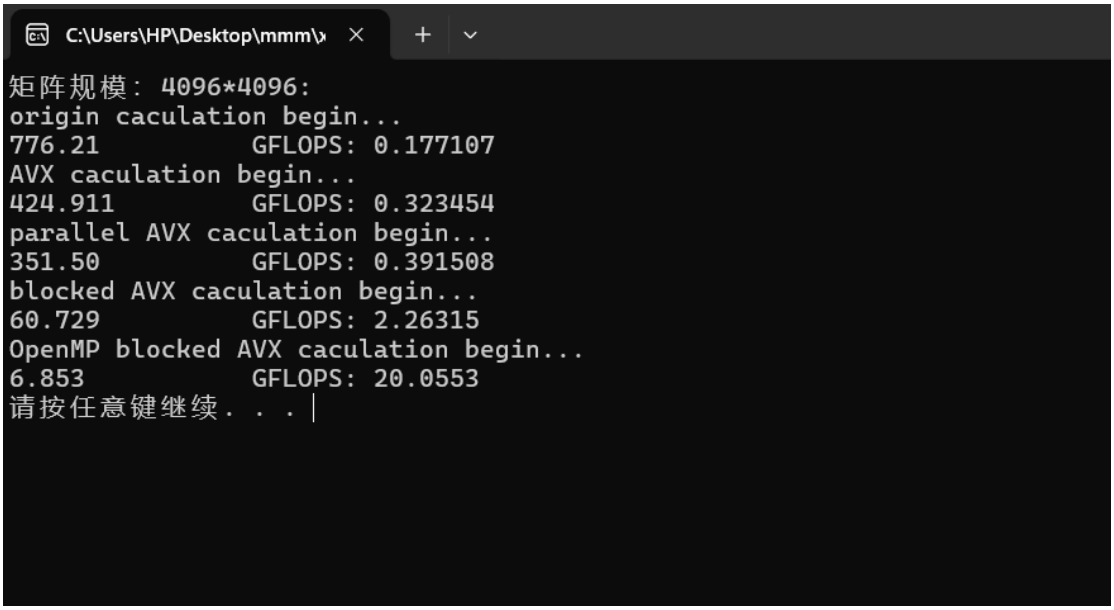
1.1、矩阵规模：1024*1024



1.2、矩阵规模：2048*2048



1.3、矩阵规模：4096*4096



二、 在泰山服务器上运行代码

2.1、修改源代码

因为泰山服务器上没有相应的 AVX 库，所以我们需要修改源代码，也就是要把子字并行这一层给去掉，然后再在泰山服务器上跑。

2.2、代码

```
void printFlops(int A_height, int B_width, int B_height, clock_t start, clock_t stop) {
    real_t flops = (2.0 * A_height * B_width * B_height) / 1E9 / ((stop - start) / (CLOCKS_PER_SEC * 1.0));
    printf("%f", flops);
    printf("\n");

}
void initmartix(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] = (i + j + (i * j) % 100) % 100;
            B[i + j * n] = ((i - j) * (i - j) + (i * j) % 200) % 100;
            C[i + j * n] = 0;
        }
    }
    return;
}
#define UNROLL (4)

#define BLOCKSIZE (32)
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;
        }
    }
    return;
}

void pavx_dgemm_normal(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) {
            for (int x = 0; x < UNROLL; ++x) {
                int idx = i + x * 4; // 计算当前元素的索引
                real_t cij[4] = { 0 }; // 初始化累加器数组

                for (int k = 0; k < n; k++) {
                    real_t a_val = A[idx + k * n];
                    real_t b_val = B[k + j * n]; // 注意这里不是AVX广播，而是直接取值
                    for (int yy = 0; yy < 4; ++yy) { // 模拟AVX的四个元素操作
                        int offset = (yy == 0) ? 0 : yy; // 对于非第一个元素，需要偏移
                        cij[yy] += a_val * b_val; // 累加结果

                        // 如果不是第一个元素，则加载下一个A的值（假设它们是连续的）
                        if (yy < 3) {
                            a_val = A[idx + (4 + offset) + k * n];
                        }
                    }
                }
                // 将结果存储回C矩阵
                for (int yy = 0; yy < 4; ++yy) {
                    C[idx + yy + j * n] = cij[yy];
                }
            }
        }
    }
}
```

```
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 < n; i += UNROLL * 4) { // 确保不越界
        for (int j = sj; j < sj + BLOCKSIZE && j < n; ++j) {
            real_t cij[UNROLL * 4]; // 假设我们要存储UNROLL个四字节块
            for (int x = 0; x < UNROLL; ++x) {
                for (int y = 0; y < 4; ++y) {
                    int idx = i + 4 * x + y;
                    if (idx < n) { // 确保索引不越界
                        int offset = j * n + idx;
                        cij[x * 4 + y] = C[offset]; // 加载C矩阵的值
                    }
                }
            }

            for (int k = sk; k < sk + BLOCKSIZE && k < n; ++k) {
                real_t bk = B[k + j * n]; // 获取B矩阵的当前值
                for (int x = 0; x < UNROLL; ++x) {
                    for (int y = 0; y < 4; ++y) {
                        int idx = i + 4 * x + y;
                        if (idx < n) { // 确保索引不越界
                            real_t ak = A[idx + k * n]; // 获取A矩阵的当前值
                            cij[x * 4 + y] += ak * bk; // 累加乘积到cij
                        }
                    }
                }
            }
            for (int x = 0; x < UNROLL; ++x) {
                for (int y = 0; y < 4; ++y) {
                    int idx = i + 4 * x + y;
                    if (idx < n) { // 确保索引不越界
                        int offset = j * n + idx;
                        C[offset] = cij[x * 4 + y]; // 将结果存回C矩阵
                    }
                }
            }
        }
    }
}

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);

}
```

```
void main() {
    real_t* A, * B, * C;
    clock_t start, stop;
    int n = 4096;
    A = (real_t*)malloc(sizeof(double) * n * n);
    B = (real_t*)malloc(sizeof(double) * n * n);
    C = (real_t*)malloc(sizeof(double) * n * n);
    initmartix(n, A, B, C);
    start = clock();
    dgemm(n, A, B, C);
    stop = clock();
    printf("originel_gemm:\n");
    printf("%ld", (stop - start) / CLOCKS_PER_SEC);
    printf(".");
    printf("%ld", (stop - start) % CLOCKS_PER_SEC);
    printf("      ");
    printFlops(n, n, n, start, stop);
    for (int i = 0; i < 5; i++) {
        printf("%lf", C[i]);
        printf("  ");
    }
    printf("\n");
    initmartix(n, A, B, C);
    start = clock();
    dgemm_unrolled(n, A, B, C);
    stop = clock();
    printf("rolled_gemm:\n");
    printf("%ld", (stop - start) / CLOCKS_PER_SEC);
    printf(".");
    printf("%ld", (stop - start) % CLOCKS_PER_SEC);
    printf("      ");
    printFlops(n, n, n, start, stop);
    for (int i = 0; i < 5; i++) {
        printf("%lf", C[i]);
        printf("  ");
    }
```

我们可以输出结果矩阵 C 的前五个元素来查看优化方法计算的结果到底对不对。

2.3、在个人电脑上运行

矩阵规模：1024*1024

```
originel_gemm:
3.246     0.661578
2037776.000000  2091976.000000  2001176.000000  2175376.000000  2239176.000000
rolled_gemm:
3.172     0.677012
2037776.000000  2239176.000000  1980376.000000  1953476.000000  2239176.000000
blocked_gemm:
2.786     0.770813
2037776.000000  2091976.000000  2001176.000000  2175376.000000  2239176.000000
open_gemm:
0.340     6.316128
2037776.000000  2091976.000000  2001176.000000  2175376.000000  2239176.000000
请按任意键继续. . .
```

矩阵规模：2048*2048

```
originel_gemm:
70.938     0.242181
4100384.000000  4233788.000000  4009192.000000  4348596.000000  4481000.000000
rolled_gemm:
69.620     0.246766
4100384.000000  4481000.000000  3971404.000000  3913808.000000  4481000.000000
blocked_gemm:
26.822     0.640514
4100384.000000  4233788.000000  4009192.000000  4348596.000000  4481000.000000
open_gemm:
4.996     3.438725
4100384.000000  4233788.000000  4009192.000000  4348596.000000  4481000.000000
请按任意键继续. . .
```

矩阵规模：4096*4096

```
originel_gemm:
1474.968     0.093181
8299600.000000  8465720.000000  8037340.000000  8695460.000000  8964080.000000
rolled_gemm:
719.840     0.190930
8299600.000000  8964080.000000  7941200.000000  7824420.000000  8964080.000000
blocked_gemm:
348.210     0.394701
8299600.000000  8465720.000000  8037340.000000  8695460.000000  8964080.000000
open_gemm:
37.394     3.675428
8299600.000000  8465720.000000  8037340.000000  8695460.000000  8964080.000000
请按任意键继续. . .
```

2.4、在泰山服务器上分别运行 1024,2048 和 4096

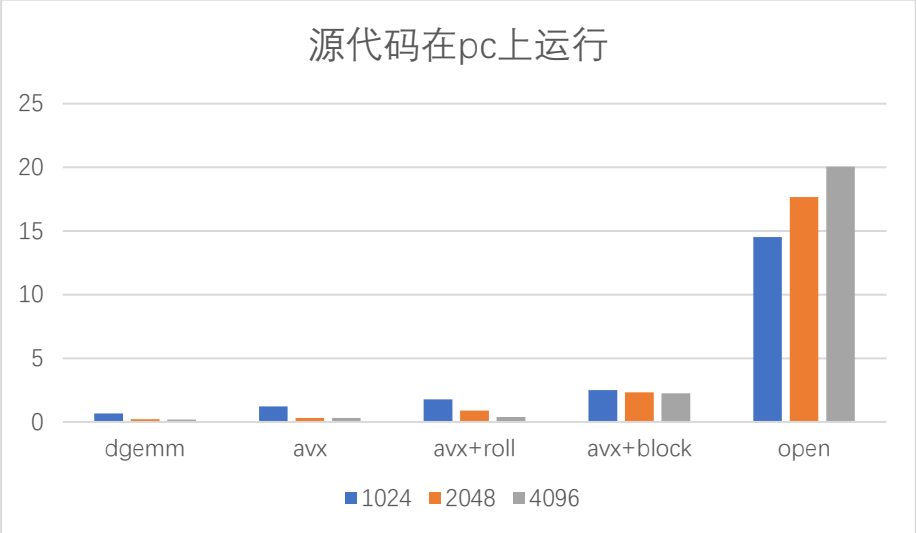
```
originel_gemm:
18.777630     0.114364
2037776.000000  2091976.000000  2001176.000000  2175376.000000  2239176.000000
rolled_gemm:
17.340083     0.123845
2037776.000000  2239176.000000  1980376.000000  1953476.000000  2239176.000000
blocked_gemm:
10.495381     0.204612
2037776.000000  2091976.000000  2001176.000000  2175376.000000  2239176.000000
open_gemm:
10.485735     0.204800
2037776.000000  2091976.000000  2001176.000000  2175376.000000  2239176.000000
```

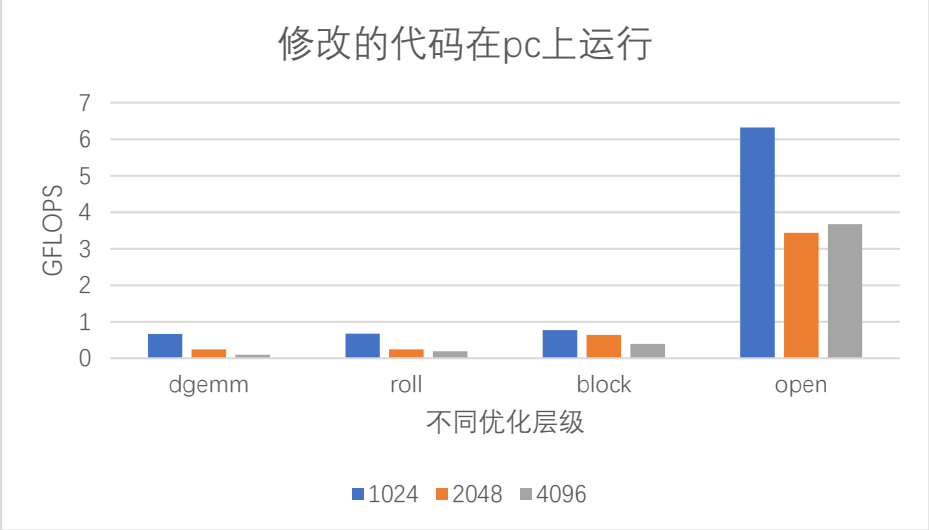
```
originel_gemm:
219.166248     0.078387
4100384.000000  4233788.000000  4009192.000000  4348596.000000  4481000.000000
rolled_gemm:
174.744492     0.098314
4100384.000000  4481000.000000  3971404.000000  3913808.000000  4481000.000000
blocked_gemm:
84.187013     0.204068
4100384.000000  4233788.000000  4009192.000000  4348596.000000  4481000.000000
open_gemm:
84.184600     0.204074
4100384.000000  4233788.000000  4009192.000000  4348596.000000  4481000.000000
```

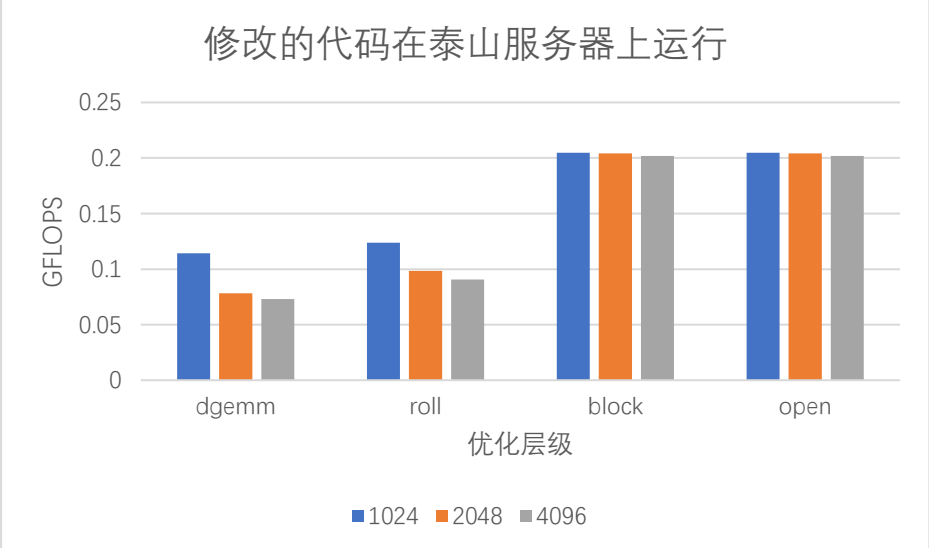
```
stu2210737@parallel542-taishan200-1:~$ ./asy1
originel_gemm:
1880.563708     0.073084
8299600.000000  8465720.000000  8037340.000000  8695460.000000  8964080.000000
rolled_gemm:
1517.96465     0.090593
8299600.000000  8964080.000000  7941200.000000  7824420.000000  8964080.000000
blocked_gemm:
680.722820     0.201901
8299600.000000  8465720.000000  8037340.000000  8695460.000000  8964080.000000
open_gemm:
680.674950     0.201916
8299600.000000  8465720.000000  8037340.000000  8695460.000000  8964080.000000
```

三、 分析比对

3.0、可视化数据


源代码在pc上运行
25
20
15
10
5
0
dgemm
avx
avx+roll
avx+block
open
1024
2048
4096


修改的代码在pc上运行
GFLOPS
7
6
5
4
3
2
1
0
dgemm
roll
block
open
不同优化层级
1024
2048
4096


修改的代码在泰山服务器上运行
GFLOPS
0.25
0.2
0.15
0.1
0.05
0
dgemm
roll
block
open
优化层级
1024
2048
4096

3.1、源代码和修改之后的代码比对

从 GFLOPS 和所用时间上的结果中，我们明显观察到我们修改后的代码在矩阵乘法运算性能方面不如源代码。这一现象的根本原因在于，源代码通过 AVX 库实现了子字并行，进而在此基础上利用循环展开或分块的方法来提升性能。然而，我们修改后的代码并没有充分利用子字并行这一优化层级，而是直接采用了分块或循环展开的方式，由此导致了性能差异。

尽管“分块”或“循环展开”等技术本身可以提升性能，但它们通常与向量化（如 AVX）结合使用，以达到最佳效果。分块技术　通常用于处理大型矩阵，将其分解为较小的子矩阵，以更有效地利用缓存并减少内存访问延迟。而循环展开则通过减少循环开销（如分支预测错误和循环控制指令）来提高性能。

因此，为了充分发挥代码的性能潜力，我们应该在性能优化过程中考虑到这些因素，并采取综合性的优化策略，而不仅仅是依赖于单一的优化手段。这样才能确保代码在性能和效率方面取得最佳表现。

3.2、个人电脑与泰山服务器比对

经过测试，我们发现在执行相同规模的矩阵乘法运算时，个人电脑与泰山服务器之间表现出了显著的性能差异。具体来

说，个人电脑在处理这些任务时往往比泰山服务器更为迅速。为了深入解析这种差异，我们需要从多个方面进行综合考量。

首先，我们不得不考虑的是 CPU 的性能差异。个人电脑所使用的 CPU，尤其是现代笔记本电脑中的高性能处理器，往往集成了最新的制程技术、更高的时钟频率和更多的缓存资源。这些特点使得个人电脑在处理矩阵乘法等计算密集型任务时能够展现。

然而，泰山服务器虽然在处理大规模数据集或并发任务时可能具有优势，但针对单一的矩阵乘法运算，其 CPU 性能可能并不如个人电脑的处理器。这可能是由于服务器 CPU 在设计时更注重的是稳定性和可扩展性，而非单一任务的性能表现。

其次，除了 CPU 性能的差异外，内存带宽、磁盘 I/O 速度以及网络延迟等因素也可能对矩阵乘法运算的性能产生影响。服务器通常配备有更大的内存和更快的磁盘，但这些优势在单个矩阵乘法运算中可能并不明显，因为该任务主要依赖于 CPU 的计算能力。

此外，我们还需要考虑到软件环境和配置的差异。个人电脑和服务器可能运行着不同的操作系统、编译器版本和库文件，这些因素都可能对程序的性能产生影响。因此，在比较两者之间的性能时，我们需要确保软件环境和配置的一致性，以便更准确地评估硬件性能的差异。

总结来说，个人电脑和泰山服务器在执行矩阵乘法运算时

表现出的性能差异主要源于 CPU 性能的差距。个人电脑通常配备有高性能的处理器，能够更高效地处理这类计算密集型任务。然而，在评估这种差异时，我们也还需要考虑到其他因素如内存带宽、磁盘 I/O 速度以及软件环境和配置的影响。

四、 总结

通过本次矩阵乘法优化实验，我本人获得了宝贵的实践经验，不仅在服务器上编写了代码并成功编译执行，还深入了解了 vim、gcc 等命令语句的用法。在此之前，我对于在服务器上直接进行代码编写和编译的过程并不熟悉，但通过自己动手操作和上网查阅资料，我逐渐掌握了这些基本工具的使用。

在编译过程中，我也遇到了一个困难，就是泰山服务器并不支持 C++的输入输出流（如 iostream），这就要求我使用 C 语言的标准输入输出函数（如 printf 和 scanf）。这一发现不仅让我意识到跨平台开发时可能遇到的兼容性问题，也让我更加熟悉和依赖 C 语言的基础 I/O 功能。

更重要的是，在阅读源代码和修改代码的过程中，我深入了解了并行、循环展开和分块等优化方法的基本原理。这些优化技术都与计算机的底层设计和 CPU 的设计理念紧密相关，旨在最大限度地利用 CPU 资源，从而提高程序的执行速度和性能。

首先，并行化是一种通过将计算任务分配给多个处理器或处理器核心来同时执行的方法。它能够有效利用多核 CPU 的并行处理能

力，显著提高程序的运行效率。

其次，循环展开是通过减少循环开销来提高性能的一种技术。它通过直接复制循环体内的代码，减少循环控制指令的执行次数，从而降低循环开销，提高程序执行速度。

最后，分块策略则是一种处理大型矩阵时常用的优化方法。它将大型矩阵划分为较小的子矩阵，并分别对这些子矩阵进行处理。这样做可以更有效地利用缓存资源，减少内存访问的延迟，从而提高程序的性能。

总而言之，通过这次实验，我不仅掌握了在服务器上编写和编译代码的基本技能，还深入了解了并行、循环展开和分块等优化方法的基本原理。这些经验对于我在未来学习计算机组成原理课具有重要的指导意义。