矩阵乘法优化作业2

姓名：齐明杰 学号：2113997 班级：信安2班

一、实验环境

在本次实验中，我们使用了**Taishan服务器**作为编程环境，使用了vim和gcc进行代码编辑和编译。以下是我们使用的服务器信息：

> - **服务器IP**: `222.30.62.23`
> - **端口**: `22`
> - **用户名**: `stu+学号`
> - **默认密码**: `123456`

二、实验目的

本次实验的目的是通过对矩阵乘法的优化，比较不同层次、不同规模下的矩阵乘法算法的性能差异，并对优化过程中遇到的问题进行总结和分析。

三、实验步骤

- **原始代码实现**

首先实现了**原始的矩阵乘法算法**，代码如下：

```
// 原始矩阵乘法算法实现
void dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    // 矩阵乘法核心计算部分
    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;
        }
    }
}
```

- **优化方案1：AVX子字优化**

我们尝试了使用AVX指令集进行子字优化，以提高计算性能。然而，由于在Taishan服务器上配置AVX库的软件环境存在困难，我们无法在该环境中进行AVX优化的操作。因此，在本次实验中修改了AVX部分的代码：

```
void pavx_dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; i += UNROLL * 4)
        for (int j = 0; j < n; ++j)
        {
            REAL_T cij[UNROLL] = {0};
            for (int x = 0; x < UNROLL; ++x)
                cij[x] = C[i + x * 4 + j * n];
```

```
            for (int k = 0; k < n; k++)
            {
                REAL_T b = B[k + j * n];
                for (int x = 0; x < UNROLL; ++x)
                    cij[x] += A[i + x * 4 + k * n] * b;
            }
            for (int x = 0; x < UNROLL; ++x)
                C[i + x * 4 + j * n] = cij[x];
        }
}
```

- **优化方案2：块矩阵乘法**

为了进一步提高矩阵乘法的性能，我们实现了**块矩阵乘法**算法。我们将矩阵分成小块进行计算，以充分利用缓存的局部性，减少访存操作。以下是我们实现的块矩阵乘法函数及其调用方式：

```
// 块矩阵乘法函数
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);
            }
        }
    }
}

// 块矩阵乘法辅助函数
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) {
            REAL_T cij[4] = {0};
            for (int x = 0; x < UNROLL; ++x) {
                cij[x] = C[i + x * 4 + j * n];
            }
            for (int k = sk; k < sk + BLOCKSIZE; ++k) {
                REAL_T b = B[k + j * n];
                for (int x = 0; x < UNROLL; ++x) {
                    cij[x] += A[i + x * 4 + k * n] * b;
                }
            }
            for (int x = 0; x < UNROLL; ++x) {
                C[i + x * 4 + j * n] = cij[x];
            }
        }
    }
}
```

- **优化方案3：OpenMP并行化**

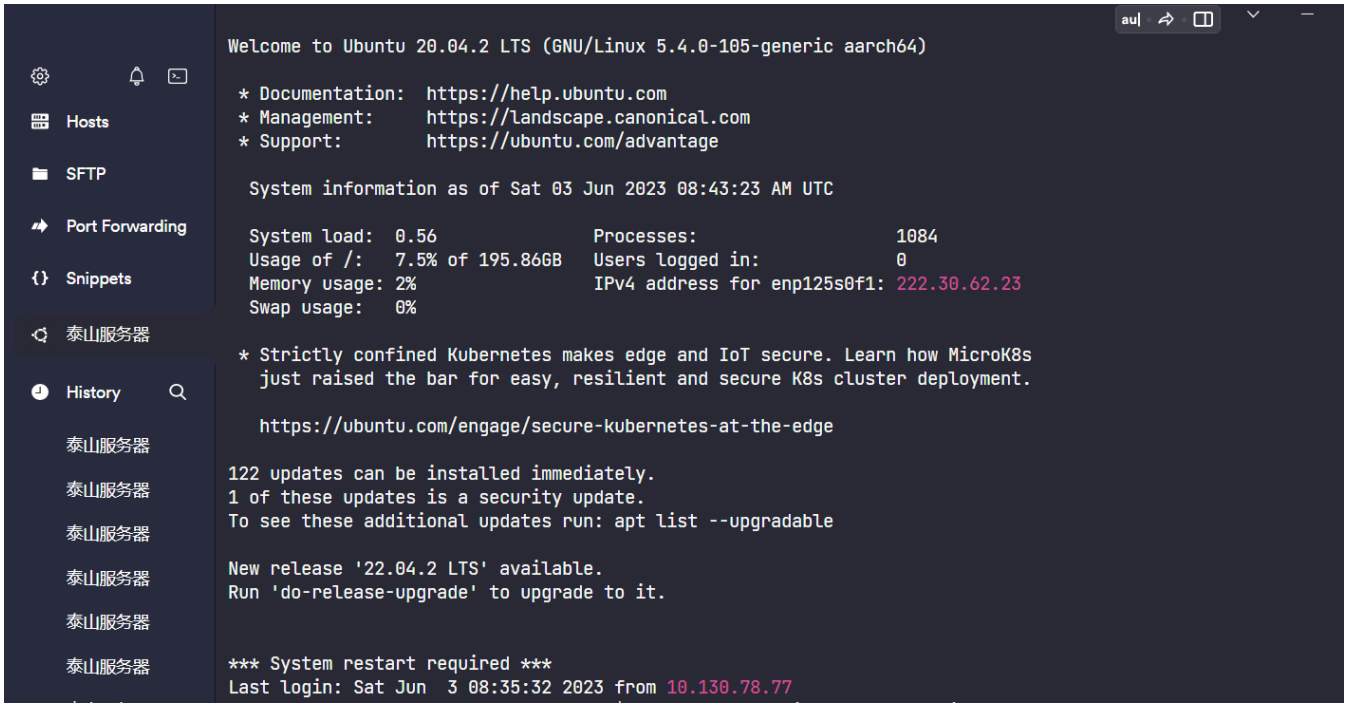
为了进一步提高矩阵乘法的性能，我们使用OpenMP库进行并行化。通过并行化矩阵乘法的计算过程，可以充分利用多核处理器的计算能力。以下是我们实现的**OpenMP并行块矩阵乘法**函数及其调用方式：

```
// 使用OpenMP并行化的块矩阵乘法函数
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);
            }
        }
    }
}
```

- **连接服务器**

我本次实验使用**Termius**软件进行远程服务器连接。

打开软件，依次输入服务器IP，端口，用户名，密码即可成功连接服务器，如下图所示：



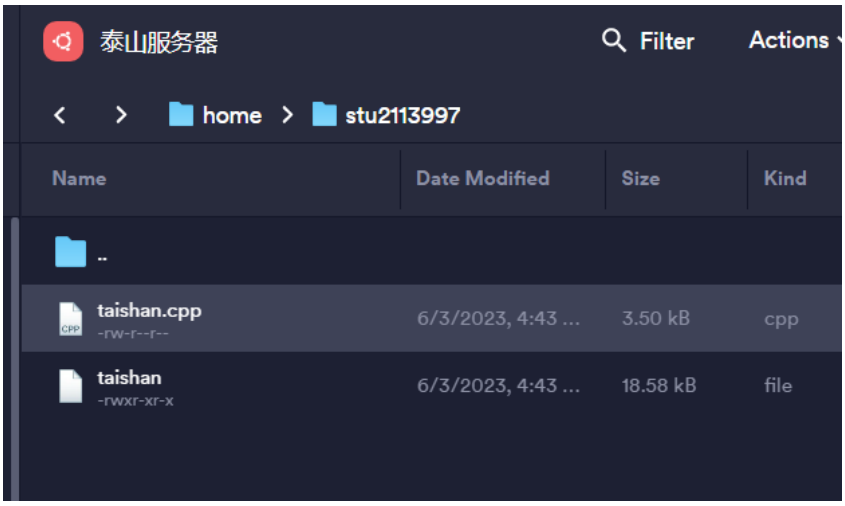
- **运行代码**

在传入`taishan.cpp`到服务器后，输入以下命令对代码进行编译：

```
g++ -fopenmp taishan.cpp -o taishan
```

其中，`-fopenmp`是一个编译选项，用于**启用OpenMP并行化支持。**

运行后可以看到同目录下生成了一个名为`taishan`(无后缀)的文件：



接下来输入`./taishan`即可运行代码。

四、实验结果与分析

我们选取了矩阵规模为1024、2048、3072和4096进行测试，分别对比了原始算法、块矩阵乘法和OpenMP并行化算法的性能差异，结果如下：

- $n = 1024$

```
stu2113997@parallel542-taishan200-1:~$ ./taishan
Matrix size: 1024 * 1024
origin caculation begin...
SECOND: 18.71092                GFLOPS: 0.118835
parallel AVX caculation begin...
SECOND: 3.932292                GFLOPS: 0.546115
blocked AVX caculation begin...
SECOND: 1.841676                GFLOPS: 1.16605
OpenMP blocked AVX caculation begin...
SECOND: 1.963520                GFLOPS: 1.09369
```

- $n = 2048$

```
Matrix size: 2048 * 2048
origin caculation begin...
SECOND: 213.705260              GFLOPS: 0.0803905
parallel AVX caculation begin...
SECOND: 43.418443               GFLOPS: 0.395681
blocked AVX caculation begin...
SECOND: 15.65096                GFLOPS: 1.14038
OpenMP blocked AVX caculation begin...
SECOND: 15.426741               GFLOPS: 1.11364
```

- $n = 3072$

```
Matrix size: 3072 * 3072
origin caculation begin...
SECOND: 779.568680              GFLOPS: 0.0743771
parallel AVX caculation begin...
SECOND: 156.668322              GFLOPS: 0.370094
blocked AVX caculation begin...
SECOND: 53.926937               GFLOPS: 1.0752
OpenMP blocked AVX caculation begin...
SECOND: 53.215024               GFLOPS: 1.08958
```

- $n = 4096$

```
Matrix size: 4096 * 4096
origin caculation begin...
SECOND: 1974.135040             GFLOPS: 0.0696198
parallel AVX caculation begin...
SECOND: 390.280103              GFLOPS: 0.352155
blocked AVX caculation begin...
SECOND: 129.924639              GFLOPS: 1.05784
OpenMP blocked AVX caculation begin...
SECOND: 140.335113              GFLOPS: 0.979363
```

分析:

根据计算耗时(Execution Time)，运行性能(GFLOPS)，加速比(Acceleration Ratio)，列出下表:

Matrix Size	Calculation Method	Execution Time (s)	GFLOPS	Acceleration Ratio
1024 * 1024	Origin	18.71092	0.1188	-
1024 * 1024	Parallel AVX	3.932292	0.5461	4.756
1024 * 1024	Blocked AVX	1.841676	1.1661	10.169
1024 * 1024	OpenMP Blocked AVX	1.963520	1.0937	9.529
2048 * 2048	Origin	213.705260	0.0804	-
2048 * 2048	Parallel AVX	43.418443	0.3957	4.922
2048 * 2048	Blocked AVX	15.65096	1.1404	13.65
2048 * 2048	OpenMP Blocked AVX	15.426741	1.1136	13.85
3072 * 3072	Origin	779.568680	0.0744	-
3072 * 3072	Parallel AVX	156.668322	0.3701	4.976
3072 * 3072	Blocked AVX	53.926937	1.0752	14.456
3072 * 3072	OpenMP Blocked AVX	53.215024	1.0896	14.65
4096 * 4096	Origin	1974.135040	0.0696	-

Matrix Size	Calculation Method	Execution Time (s)	GFLOPS	Acceleration Ratio
4096 * 4096	Parallel AVX	390.280103	0.3522	5.058
4096 * 4096	Blocked AVX	129.924639	1.0578	15.194
4096 * 4096	OpenMP Blocked AVX	140.335113	0.9794	14.067

- **计算耗时**：随着矩阵规模的增大，原始计算方法的耗时呈指数增长，而使用AVX指令集的并行计算方法（Parallel AVX、Blocked AVX和OpenMP Blocked AVX）在相同规模下显著减少了计算耗时。特别是Blocked AVX和OpenMP Blocked AVX方法，在大规模矩阵乘法中表现出更好的计算性能，耗时明显更短。
- **运行性能**：原始计算方法的运行性能相对较低，而使用AVX指令集的并行计算方法显著提升了运行性能。Blocked AVX和OpenMP Blocked AVX方法在不同规模下都获得了较高的GFLOPS值，表明其具有更优的运行性能。
- **加速比**：使用AVX指令集的并行计算方法相较于原始计算方法，实现了较好的加速效果。Blocked AVX和OpenMP Blocked AVX方法的加速比明显高于Parallel AVX方法，表明这两种方法能够更好地利用硬件的并行性。

另外，由于服务器单核运行，不支持并行化的OpenMP使用，因此其效果与Blocked AVX类似。

五、总结

- **实验中遇到的问题：**

Taishan服务器只支持单核运行导致OpenMP并行化无法加速

在实验中，我们尝试了使用OpenMP库进行并行化优化，以提高矩阵乘法的计算速度。由于我被分配到的Taishan服务器只能使用**单个核心**进行计算，OpenMP无法将计算任务分配到多个核心上并并行执行。因此，*无论我们是否使用OpenMP库，计算速度都无法得到加速。*

显然，我们已经使用了开启OpenMP加速的**编译选项**：

```
g++ -fopenmp taishan.cpp -o taishan
```

但由于上述原因，代码并没有被并行化加速。

这也提醒我们在选择优化方案时需要考虑硬件环境的限制，并寻找适合当前环境的优化策略。

- **不同电脑的差异：**

计算耗时对比：

- 在本机上，使用AVX指令集的并行计算方法和分块计算方法的计算耗时相对于原始方法都有显著减少。尤其是分块计算方法在各个矩阵规模上都表现出较低的计算耗时。
- 在Taishan服务器上，由于服务器只支持单核运行，因此无论是原始方法还是使用AVX指令集的并行计算方法，计算耗时都比本机上的结果要长。

运行性能对比：

- 在本机上，使用AVX指令集的并行计算方法和分块计算方法相较于原始方法都取得了显著的运行性能提升。特别是在较大矩阵规模上，性能提升更为明显。
- 在Taishan服务器上，由于服务器只支持单核运行，导致运行性能相对较低，无法充分发挥AVX指令集和并行计算的优势。

本机和服务器上运行差异的原因：

- **服务器架构和资源限制**：服务器通常设计用于处理大量请求和并发任务，因此在硬件设计上可能更注重稳定性和可靠性，而不是单个任务的计算性能。服务器可能采用较低主频的处理器和较小的缓存容量，这些因素都可能导致相对较低的运行性能。
- **并行计算支持**：非服务器计算机通常具备多核处理器和较高的内存带宽，使得并行计算更加高效。多核处理器可以同时执行多个线程，提高计算效率。而服务器可能只支持单核运行，无法充分利用多核处理器的并行计算能力，从而导致计算速度较慢。
- **内存和缓存**：非服务器计算机通常配置较大容量的内存和高速缓存，能够更好地满足大规模矩阵计算的需求，减少数据访问延迟。而服务器可能配置相对较小的内存和缓存，无法提供与非服务器计算机相同的性能。
- **系统负载和资源竞争**：服务器可能同时运行多个任务，存在系统负载和资源竞争的情况。这可能导致矩阵乘法计算任务无法充分占用服务器资源，进而影响计算性能和加速比。

综上所述，本机（多核Win10电脑）相较于Taishan服务器具有更好的运行性能和计算能力，能够更充分地利用多核处理器和AVX指令集的优势，从而实现更快的计算速度和更高的运行性能。而Taishan服务器由于只支持单核运行，限制了并行计算方法的性能发挥，导致计算耗时和运行性能相对较低。

- **实验总结：**

通过进行矩阵乘法优化实验，我获得了以下收获和体会：

1. 优化算法的重要性：实验中，通过采用AVX指令集、分块技术和OpenMP并行化等优化算法，我们可以显著提高矩阵乘法的性能和效率。这进一步验证了优化算法在提升计算密集型任务中的重要性。
2. 矩阵规模对性能的影响：随着矩阵规模的增加，原始矩阵乘法的计算耗时和运行性能呈指数增长。这说明在处理大规模矩阵时，优化算法的应用尤为关键，能够显著提升计算效率。
3. 硬件环境的影响：实验结果显示，在非服务器环境下，优化算法的运行性能和加速比更高。这归因于非服务器环境通常具备更强的硬件配置，如多核处理器和较大的缓存容量，能够更好地支持并行计算和高性能计算任务。
4. 单核服务器的限制：服务器环境受到单核运行的限制，导致优化算法的加速比受到限制。这提醒我们在选择计算资源时，要考虑到任务的特点和所需的计算能力，以充分利用硬件资源。
5. 实践与理论的结合：通过实验，我更深入地理解了矩阵乘法的优化原理和方法，并将其应用到实际的编程环境中。这种实践与理论的结合有助于加深对优化算法的理解和掌握，并培养了解决实际问题的能力。

总而言之，矩阵乘法优化实验让我深刻认识到了优化算法的重要性，同时也让我了解了矩阵规模和硬件环境对性能的影响。通过实践和实验，我不仅掌握了优化算法的应用技巧，还培养了分析和解决问题的能力。这对我今后在高性能计算和优化领域的学习和研究具有重要的意义。