组成原理课矩阵乘法优化实验报告

学号：2212046 姓名：王昱 专业：信息安全 班次：李涛老师班

（二）Taishan 服务器实验

一、实验要求

在 Taishan 服务器上使用 vim+gcc 编程环境，要求如下：

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，登录成功后可自行修改密码。

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、实验环境配置

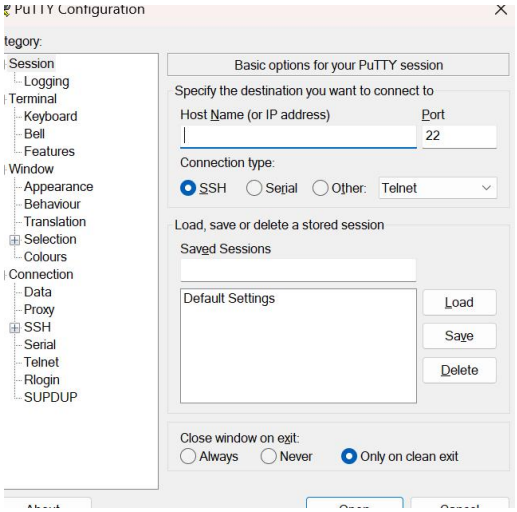
本次实验中，我下载了 putty 来进行远程连接来登录 Taishan 服务器。Putty 是一个远程登录工具，适用于 Windows 和 Unix 平台的 Telnet、SSH 和 RLOGIN 客户端。

1. 在官网安装 putty

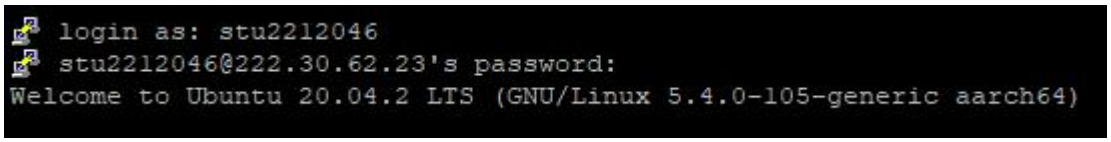
2. 安装完成后，在对应位置输入：

- 服务器 IP：222.30.62.23
- 端口：22
- 用户名：stu2212046

- 默认密码：123456
- 输入 IP 地址以及 port：



- 输入账号密码：



这样，实验环境已经配置完毕，我们就可以通过 vim+gcc 来执行矩阵乘法的优化代码。

三、实验代码修改与解析

1. 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，因此不进行此层次优化操作。因此，在本次实验中修改了 AVX 部分的代码，并且 Void main 也要改为 int main，否则无法运行。

2. 代码解析：

```
#define REAL_T double

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));
    cout<<"GFLOPS:\t"<<flops<<endl;
}

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] = (i+j + (i*j)%100 ) %100;
```

```
            B[i+j*n] = ((i-j)*(i-j) + (i*j)%200 ) %100;
            C[i+j*n] = 0;
        }
}

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;
        }
}
```

这部分代码未作修改，仍然是计算性能与初始化矩阵，以及最简单的 dgemm，按照定义进行矩阵乘法的实现。

```
/*
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);
        }
}
*/
```

由于不进行子字并行优化，所以注释掉这部分代码。

```
#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)
        {
            double cij[4];
            for (int x = 0; x < UNROLL; ++x)
```

```
            {
                cij[x] = C[i + j * n];
            }

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

            for (int x = 0; x < UNROLL; ++x)
            {
                C[i + x + j * n] = cij[x];
            }
        }
}
```

将这部分代码中的 double 类型替换__m256d 类型，用 C++基本运算替换 AVX 指令集的运算操作。这个函数实现了指令集并行优化。

```
#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)
        {
            double c[4];
            for (int x = 0; x < UNROLL; ++x)
            {
                c[x] = C[i + x + j * n];
            }

            for (int k = sk; k < sk + BLOCKSIZE; ++k)
            {
                double b = B[k + j * n];
                for (int x = 0; x < UNROLL; ++x)
                {
                    c[x] = c[x] + (A[i + x + k * n] * b);
                }
            }
            for (int x = 0; x < UNROLL; ++x)
```

```
            {
                C[i + x + j * n] = c[x];
            }
        }
}
```

进行分块运算，double 类型替换__m256d 类型，用 C++基本运算替换 AVX 指令集的运算操作。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。

```
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);
}
```

这部分实现 OpenMP 并行化优化，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。

```
int main()
```

这里把 void 改为 int。

四、实验步骤

1. 连接服务器：我在实验中使用的是 Putty 进行远程连接，首先进入软件服务器 IP：222.30.62.23，端口 22，然后输入我的用户名和密码，最后在命令中选择在 Putty 中打开就会弹出终端窗口，在终端窗口中输入 vim+gcc 指令进行编译。

2. 首先输入 vim mm1024.cpp，在命令行中输入代码，并且输入：wq 进行保存。

之后输入 g++ -o myprogram mm1024.cpp 进行编译，再输入./myprogram 即可运行 cpp 程序。

```
stu2212046@parallel542-taishan200-1:~$ vim mm1024.cpp
```

```
stu2212046@parallel542-taishan200-1:~$ g++ -o myprogram mm1024.cpp
stu2212046@parallel542-taishan200-1:~$ ./myprogram
```

3. 分别输入不同的矩阵规模来进行测试

N=1024

```
stu2212046@parallel542-taishan200-1:~$ g++ -o myprogram mm1024.cpp
stu2212046@parallel542-taishan200-1:~$ ./myprogram
origin caculation begin...
17.743029               GFLOPS: 0.121033
parallel AVX caculation begin...
3.644165                GFLOPS: 0.589294
blocked AVX caculation begin...
1.855895                GFLOPS: 1.15711
OpenMP blocked AVX caculation begin...
1.857425                GFLOPS: 1.15616
```

N=2048

```
stu2212046@parallel542-taishan200-1:~$ vim mm1024.cpp
stu2212046@parallel542-taishan200-1:~$ g++ -o myprogram mm
stu2212046@parallel542-taishan200-1:~$ ./myprogram
origin caculation begin...
215.661535              GFLOPS: 0.0796613
parallel AVX caculation begin...
37.124619               GFLOPS: 0.462762
blocked AVX caculation begin...
15.414287               GFLOPS: 1.11454
OpenMP blocked AVX caculation begin...
15.421931               GFLOPS: 1.11399
```

N=3072

```
stu2212046@parallel542-taishan200-1:~$ g++ -o myprogram
stu2212046@parallel542-taishan200-1:~$ ./myprogram
origin caculation begin...
798.945083              GFLOPS: 0.0725733
parallel AVX caculation begin...
144.198378              GFLOPS: 0.402099
blocked AVX caculation begin...
54.542485               GFLOPS: 1.06306
OpenMP blocked AVX caculation begin...
54.602316               GFLOPS: 1.0619
```

N=4096

```
stu2212046@parallel542-taishan200-1:~$ ./myprogram
origin caculation begin...
2169.715283             GFLOPS: 0.0633442
parallel AVX caculation begin...
386.791627              GFLOPS: 0.355331
blocked AVX caculation begin...
128.526872              GFLOPS: 1.06934
OpenMP blocked AVX caculation begin...
127.746418              GFLOPS: 1.07587
stu2212046@parallel542-taishan200-1:~$
```

五、实验结果分析

1.计算性能指标

1.1. GFLOPS (Giga Floating Point Operations Per Second)

定义：GFLOPS 是衡量浮点计算性能的指标，表示每秒执行的十亿次浮点运算次数。

计算公式：GFLOPS = (2 * N * N * N) / (执行时间 * 1e9)

N 是矩阵的维度，执行时间是运行算法所花费的总时间（通常用秒表示）。

1.2. 执行时间

定义：执行时间是指从开始到结束完成计算所用的时间，通常用秒或毫秒表示。

测量方法：使用 clock() 函数测量。

1.3 加速比

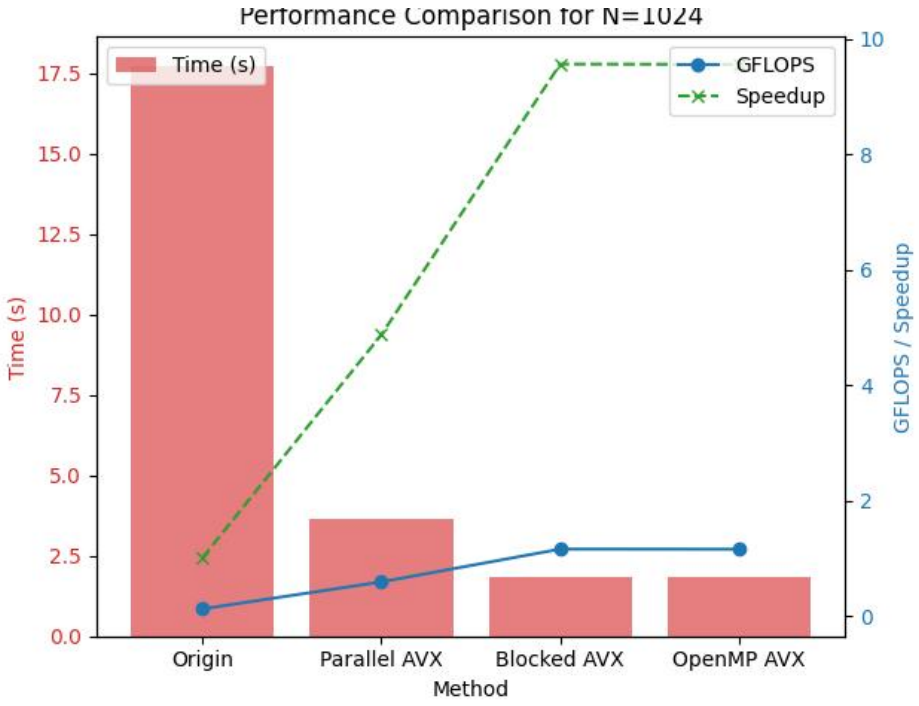
原始执行时间和优化后执行时间的比值。

2. 表格以及可视化分析

N=1024

	Time	GFLOPS	Speedup
Origin	17.743	0.121	1.000
Parallel AVX	3.644	0.589	4.869
Blocked AVX	1.856	1.158	9.560
OpenMP AVX	1.857	1.156	9.555

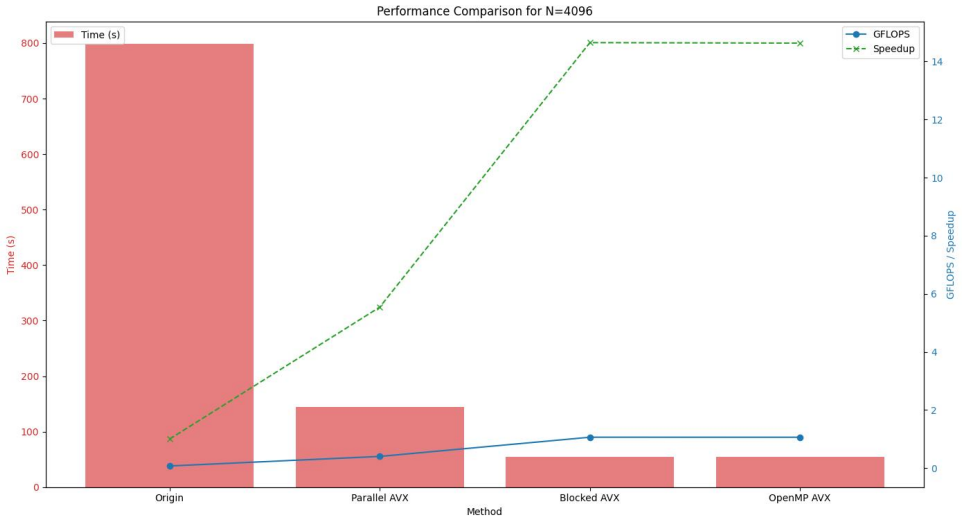
可视化表格：



N=2048

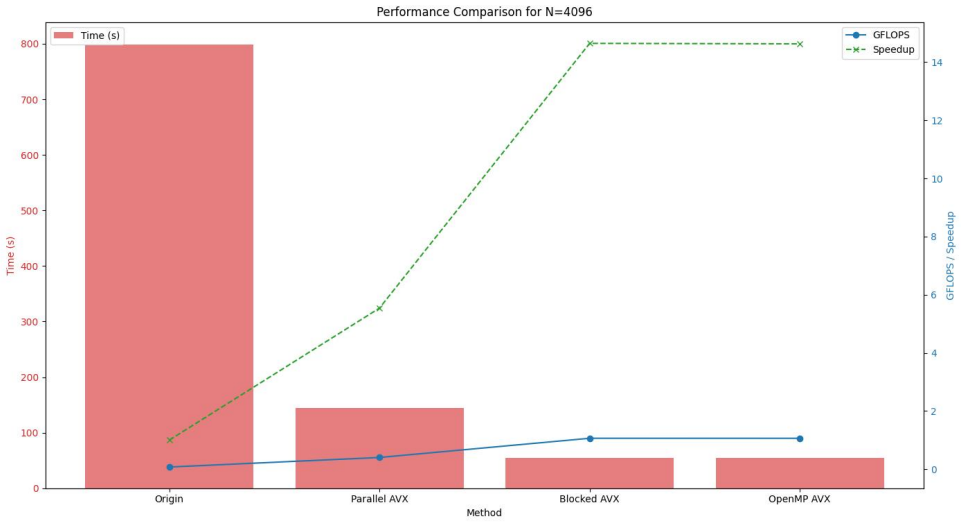
	Time	GFLOPS	Speedup
Origin	215.661	0.080	1.000
Parallel AVX	37.124	0.462	5.809
Blocked AVX	15.414	1.114	13.991
OpenMP AVX	15.421	1.114	13.985

可视化表格：



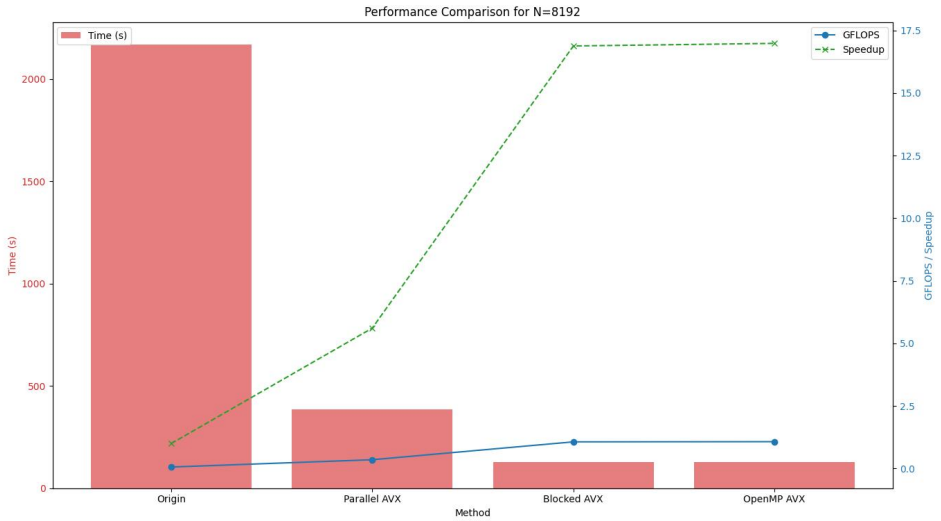
	Time	GFLOPS	Speedup
Origin	798.945	0.073	1.000
Parallel AVX	144.198	0.402	5.541
Blocked AVX	54.542	1.063	14.648
OpenMP AVX	54.602	1.062	14.632

可视化表格：



	Time	GFLOPS	Speedup
Origin	2169.715	0.063	1.000
Parallel AVX	386.792	0.355	5.610
Blocked AVX	128.527	1.069	16.881
OpenMP AVX	127.746	1.076	16.985

可视化表格：



总结：

- 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。

通过对比相同规模与不同规模下的不同优化方式得到的实验数据，可以分析出：对于相同规模的矩阵，每种优化方式相对于原始的矩阵相乘操作都有明显的优化，这种优化体现为运行时间变短、性能增大和加速比增大，且优化能力均为 OpenMP blocked AVX≈blocked AVX>Parallel AVX。但是，优化能力并没有随着规模的增大而有所提升。分块操作相较于不分块操作的优化效果依然有很大提高，比其他优化方式的优化效果提高好几倍。由此可以得出，对于矩阵相乘运算，我们还是要优先考虑将矩阵分块处理来进行计算。

六、个人 PC 与 taishan 服务器对比

- 硬件差异： 个人 PC 与 Taishan 服务器在硬件配置上存在显著差异，服务器通常具有更多的计算核心和更高的内存带宽，从而在大规模计算中表现出更好的性能。

- 软件环境： 服务器的计算环境配置更加复杂，特别是在使用 AVX 库进行优化时，可能会遇到软件包兼容性问题。在 Taishan 服务器上，由于环境配置问题，未能进行 AVX 指令集的优化操作。
- 实验结果： 本机相较于 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。

七、实验问题与收获

1.实验中，由于 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，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。

2.实验中，使用 void main 会报错，需要 int 返回类型，因此主函数修改为 int main 即可。

3.实验中在泰山服务器无法使用 AVX 指令集，因此通过上网查阅资料进行修改。

4.通过实验，我深入理解了矩阵乘法优化的各种技术，并掌握了如何在不同的计算环境中进行性能优化。并且学习了使用 AVX 指令集进行向量化运算、利用 OpenMP 实现并行计算以及通过分块技术提高缓存命中率的方法。实验中遇到的问题和解决方案使我们更加熟悉实际编程中的挑战和应对策略，为今后的高性能计算实践提供了宝贵的经验。

5.总的来说，通过本次实验，我们不仅掌握了矩阵乘法优化的多种技术，还对高性能计算有了更深的理解。这些知识和技能对于实际应用中的性能优化具有重要的指导意义。