矩阵乘法优化作业 2

2112060-孙蕗-信息安全

一、 作业要求

【矩阵乘法优化作业 2】参考课程中讲解的矩阵乘法优化机制和原理，在 Taishan 服务器上使用 vim+gcc 编程环境，完成不同层次的矩阵乘法优化作业，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。

5、作业无固定模板，以附件形式提交，应为 word 或 pdf 文件，文件名为："学号_姓名_组成原理矩阵乘法作业 2.pdf"

二、 步骤

1. 登录 taishan 服务器

```
login as: stu2112060
stu2112060@222.30.62.23's password:
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-105-generic aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sat 03 Jun 2023 02:17:39 AM UTC

  System load:  0.03              Processes:                  1098
  Usage of /:   6.9% of 195.86GB  Users logged in:            2
  Memory usage: 2%                IPv4 address for enp125s0f1: 222.30.62.23
  Swap usage:   0%

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.

   https://ubuntu.com/engage/secure-kubernetes-at-the-edge

131 updates can be installed immediately.
10 of these updates are security updates.
To see these additional updates run: apt list --upgradable

New release '22.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


*** System restart required ***

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

stu2112060@parallel542-taishan200-1:~$
```

2. 在服务器上创建 my_program.cpp 文件

```
stu2112060@parallel542-taishan200-1:~$ vim my_program.cpp
        REAL_T *A, *B, *C;
        clock_t start,stop;
        int n = 1024;
        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();
```

按下 i 键进入插入模式，在插入模式下编写 C++代码。

C++代码：

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

using namespace std;
#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;
        }
}

#define UNROLL (4)

#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)
```

```
            for (int j = sj; j < sj + BLOCKSIZE; ++j) {
                REAL_T cij = C[i + j * n];
                for (int k = sk; k < sk + BLOCKSIZE; ++k) {
                    cij += A[i + k * n] * B[k + j * n];
                }
                C[i + j * n] = cij;
            }
    }

    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()
    {
        REAL_T* A, * B, * C;
        clock_t start, stop;
        int n = 1024;
        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 << "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);

        delete[] A;
        delete[] B;
        delete[] C;

        return 0;
    }
```

按 esc 退出插入模式，：wq 将代码保存到文件并退出 vim 编辑器

3. vim my_program.cpp 使用 g++编译器将 my_program.cpp 编译为可执行文件 my_program。

4. 使用命令./my_program 来运行该可执行文件并查看输出结果

N=1024

```
origin caculation begin...
18.58683                GFLOPS: 0.118917
blocked AVX caculation begin...
7.438177                GFLOPS: 0.288711
OpenMP blocked AVX caculation begin...
7.437245                GFLOPS: 0.288747
stu2112060@parallel542-taishan200-1:~$
```

N=2048

```
stu2112060@parallel542-taishan200-1:~$ ./my_program
origin caculation begin...
216.699656              GFLOPS: 0.0792796
blocked AVX caculation begin...
59.532431               GFLOPS: 0.28858
OpenMP blocked AVX caculation begin...
59.568130               GFLOPS: 0.288407
```

N=3072

```
stu2112060@parallel542-taishan200-1:~$ ./my_program
origin caculation begin...
804.739509              GFLOPS: 0.0720507
blocked AVX caculation begin...
202.783020              GFLOPS: 0.285932
OpenMP blocked AVX caculation begin...
203.577         GFLOPS: 0.285625
```

N=4096

```
stu2112060@parallel542-taishan200-1:~$ ./my_program
origin caculation begin...
2089.389173             GFLOPS: 0.0657795
blocked AVX caculation begin...
480.375474              GFLOPS: 0.286107
OpenMP blocked AVX caculation begin...
481.191256              GFLOPS: 0.285622
```

	N=1024	N=2048	N=3072	N=4096
原始矩阵乘法执行时间	18.58683	216.699656	804.739509	2089.389173
原始矩阵乘法每秒执行的浮点运算次数（GFLOPS）	0.118917	0.0792796	0.0720507	0.0657795
使用分块计算的方式进行矩阵乘法优化的执行时间	7.438177	59.532431	202.783020	480.375474
使用分块计算的方式进行矩阵乘法优化的每秒执行的浮点运算次数（GFLOPS）	0.288711	0.28858	0.285932	0.286107
使用分块计算的方式进行矩阵乘法优化的加速比	2.4988	3.640	3.968	4.349
使用 OpenMP 进行并行计算的矩阵乘法优化的执行时间	7.437245	59.568130	203.577	481.191256
使用 OpenMP 进行并行计算的矩阵乘法优化的每秒执行的浮点运算次数（GFLOPS）	0.288747	0.288407	0.285625	0.285622
使用 OpenMP 进行并行计算的矩阵乘法优化的加速比	2.4991	3.638	3.953	4.342

三、 分析

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。

执行时间差异：在所有测试案例中，Taishan 服务器的执行时间明显长于个人电脑。例如，在 N=4096 的情况下，Taishan 服务器上的执行时间是个人电脑的约两倍左右。这可能

是由于服务器的硬件规格、处理器性能或者系统负载等因素导致的。

每秒执行的浮点运算次数（GFLOPS）差异：Taishan 服务器上的每秒执行的浮点运算次数比个人电脑上低得多。例如，在 N=1024 的情况下，Taishan 服务器的 GFLOPS 只有个人电脑的约 0.12 的一半左右。这表明个人电脑上的处理器性能可能更高，能够更快地执行浮点运算。

加速比差异：尽管 Taishan 服务器上的执行时间较长，但在使用分块计算和 OpenMP 进行并行计算的优化方式下，它实现了比个人电脑更高的加速比。例如，在 N=4096 的情况下，Taishan 服务器的加速比为 4.349，而个人电脑的加速比为 1.918。这表明在并行计算方面，Taishan 服务器的硬件和系统设置能够更好地利用多核处理器，从而获得更高的性能提升。

编译器优化：编译器的不同版本和优化级别可能会对程序的性能产生影响。不同的编译器可能有不同的优化策略，能够生成更高效的机器代码。Taishan 服务器可能使用了更先进的编译器版本或进行了更高级别的优化，从而提升了程序的性能。

系统负载：如果 Taishan 服务器上有其他运行中的任务或进程占用了计算资源，可能会影响程序的性能。个人电脑通常只有少量的后台任务，而 Taishan 服务器可能同时运行多个任务，因此在负载较高时，服务器的性能可能会受到影响。

在优化过程中，可能会遇到以下一些常见问题，以及相应的解决方式：

未充分利用并行计算：如果程序适用于并行计算，但未正确利用并行性，可能无法充分发挥多核处理器或并行处理指令集的优势。通过合理设计并实现并行算法、任务分配和同步机制，可以提高并行计算的效率。通过使用多线程或并行计算库（如 OpenMP）对循环进行并行化，可以充分利用多核处理器的计算能力。但在实施并行化时，需要考虑数据依赖性、线程同步和负载平衡等问题，以避免性能损失或并发问题。

编译器优化不充分：使用不同的编译器和编译标志可以对程序性能产生显著影响。编译器可能未能充分优化程序，生成效率较低的机器代码。通过调整编译器选项、启用特定的优化标志或手动优化代码，可以改善编译器的优化能力，提高程序性能。

不合理的代码结构和逻辑：代码结构和逻辑可能不够清晰和简洁，导致性能瓶颈或冗余计算。通过重构代码、消除不必要的分支和循环、减少函数调用等方式，可以优化代码结构和逻辑，提高性能。

资源限制和系统负载：优化过程中，可能会受到资源限制和系统负载的影响，导致无法充分利用计算资源。可以通过合理的资源管理和负载均衡，以及优化算法和调度策略，来充分利用可用资源，提高性能。