

《计算机组成原理》矩阵乘法实验报告

实验名称：矩阵乘法 班级：李涛老师 学生姓名：陆皓喆 学号：2211044

指导老师：董前琨 实验地点：实验楼A306 实验时间：2024.04.21

一、实验要求

1.1 个人PC电脑实验

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。

1.2 Taishan服务器实验

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。

二、个人PC电脑的实验

2.1 源代码

```
#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;
        }
}

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);
        }
}

#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++) {
                //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) )
    );*/
    __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]);
    }
}

#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]);
        }
    }

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 = 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 << "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);

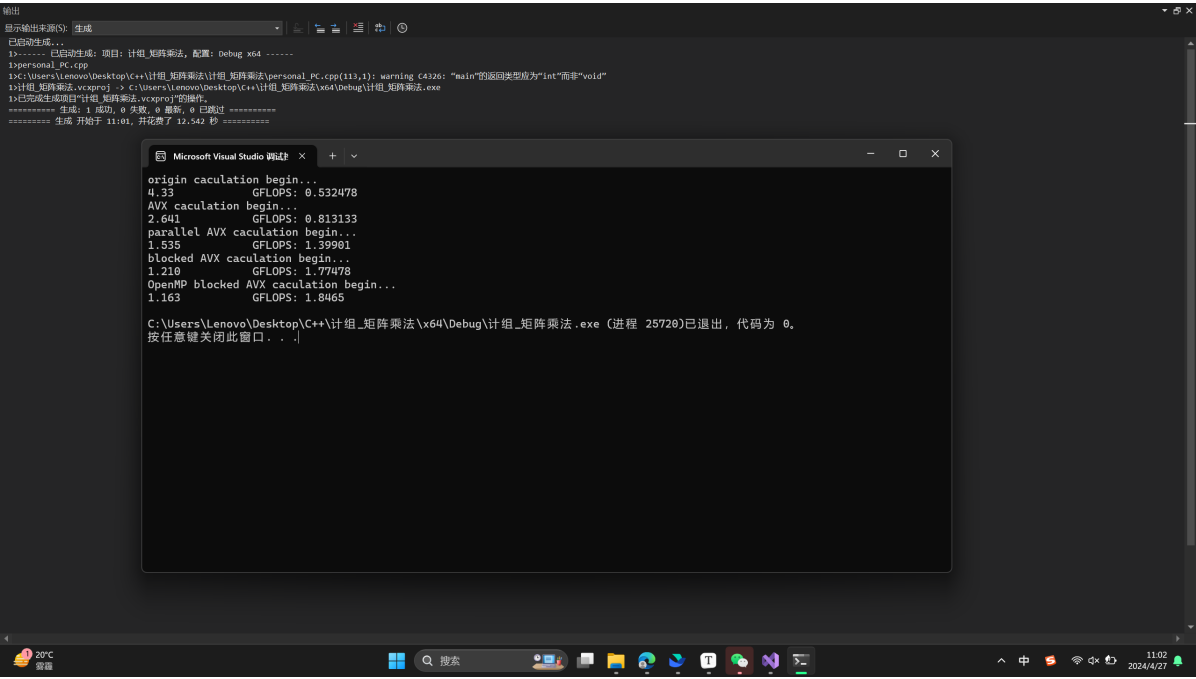
initMatrix(n, A, B, C);
cout << "blocked AVX caculation begin...\n";
start = clock();
block_gemm(n, A, B, C);
stop = clock();
cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
printFlops(n, n, n, start, stop);

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);
}

```

2.2 编译器运行

根据老师给出的源代码文件，我们将代码放入编译器中进行运行。下面是矩阵规模为 1024×1024 的时候的运行结果。



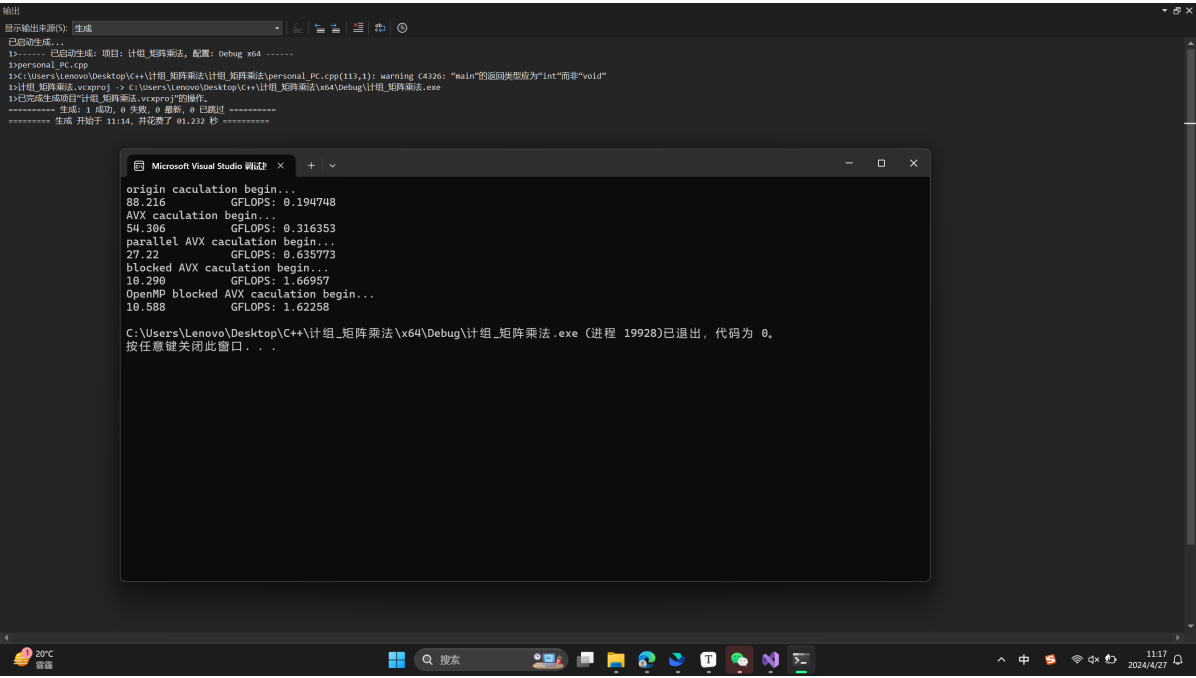
```
输出
显示输出来源(S): 生成
已启动生成...
1>----- 已启动生成: 项目: 计组_矩阵乘法, 配置: Debug x64 -----
1>personal_PC.cpp
1>C:\Users\Lenovo\Desktop\C++\计组_矩阵乘法\personal_PC.cpp(113,1): warning C4326: "main"的返回类型应为"int"而非"void"
1>计组_矩阵乘法.vcxproj -> C:\Users\Lenovo\Desktop\C++\计组_矩阵乘法\Debug\计组_矩阵乘法.exe
1>已生成项目"计组_矩阵乘法.vcxproj"的操作。
***** 生成: 1 成功, 0 失败, 0 警告, 0 已跳过 *****
***** 生成 开始于 11:01, 并花费了 12.542 秒 *****

Microsoft Visual Studio 调试
origin caculation begin...
41.33      GFLOPS: 0.532478
AVX caculation begin...
2.641      GFLOPS: 0.813133
parallel AVX caculation begin...
1.535      GFLOPS: 1.39901
blocked AVX caculation begin...
1.210      GFLOPS: 1.77478
OpenMP blocked AVX caculation begin...
1.163      GFLOPS: 1.8465
C:\Users\Lenovo\Desktop\C++\计组_矩阵乘法\x64\Debug\计组_矩阵乘法.exe (进程 25728)已退出, 代码为 0。
按任意键关闭此窗口...

20°C
晴
11:02
2024/4/27
```

同理，我们再进行实验，得到 2048×2048 、 3072×3072 、 4096×4096 的矩阵规模的时间统计。

2048×2048 :

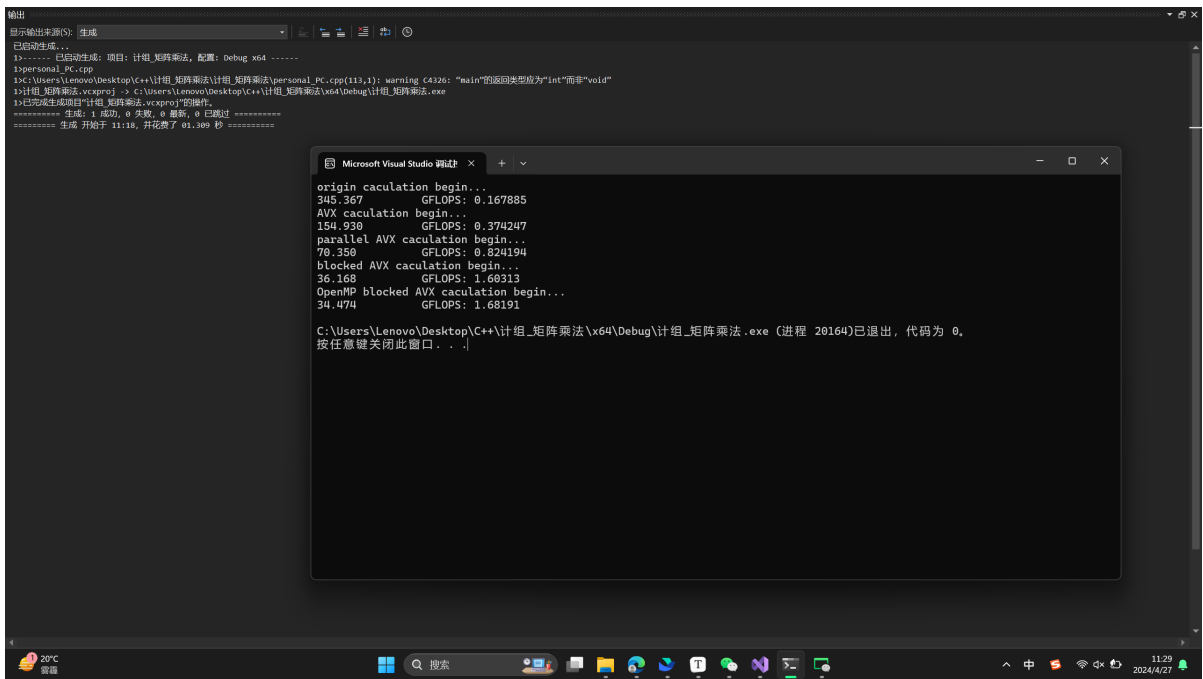


```
输出
显示输出来源(S): 生成
已启动生成...
1>----- 已启动生成: 项目: 计组_矩阵乘法, 配置: Debug x64 -----
1>personal_PC.cpp
1>C:\Users\Lenovo\Desktop\C++\计组_矩阵乘法\personal_PC.cpp(113,1): warning C4326: "main"的返回类型应为"int"而非"void"
1>计组_矩阵乘法.vcxproj -> C:\Users\Lenovo\Desktop\C++\计组_矩阵乘法\Debug\计组_矩阵乘法.exe
1>已生成项目"计组_矩阵乘法.vcxproj"的操作。
***** 生成: 1 成功, 0 失败, 0 警告, 0 已跳过 *****
***** 生成 开始于 11:14, 并花费了 01.232 秒 *****

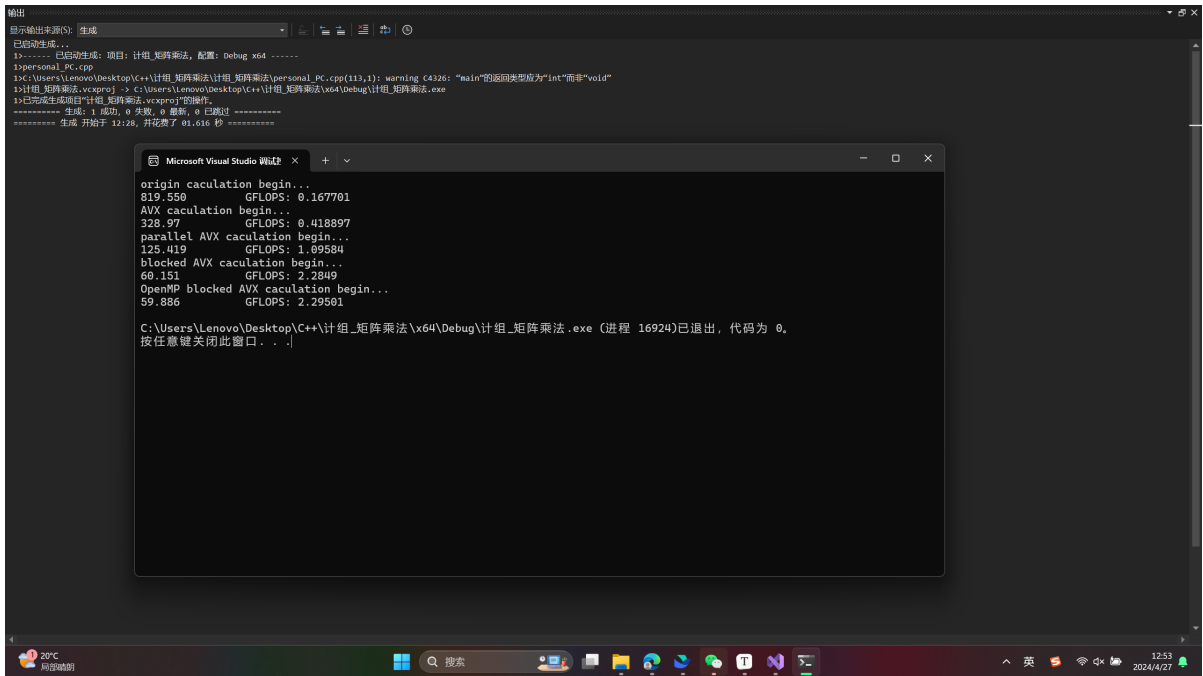
Microsoft Visual Studio 调试
origin caculation begin...
88.216     GFLOPS: 0.194748
AVX caculation begin...
54.306     GFLOPS: 0.316353
parallel AVX caculation begin...
27.22      GFLOPS: 0.635773
blocked AVX caculation begin...
10.290     GFLOPS: 1.66957
OpenMP blocked AVX caculation begin...
10.588     GFLOPS: 1.62258
C:\Users\Lenovo\Desktop\C++\计组_矩阵乘法\x64\Debug\计组_矩阵乘法.exe (进程 19928)已退出, 代码为 0。
按任意键关闭此窗口...

20°C
晴
11:17
2024/4/27
```

3072×3072 :



4096 × 4096:

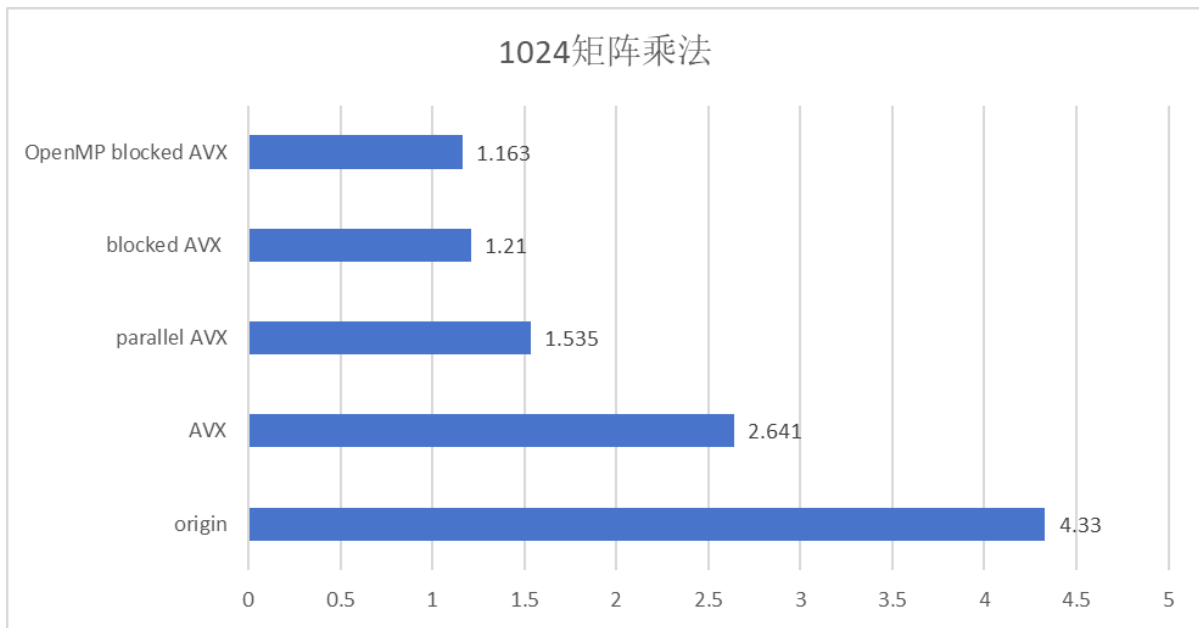


2.3 测试结果

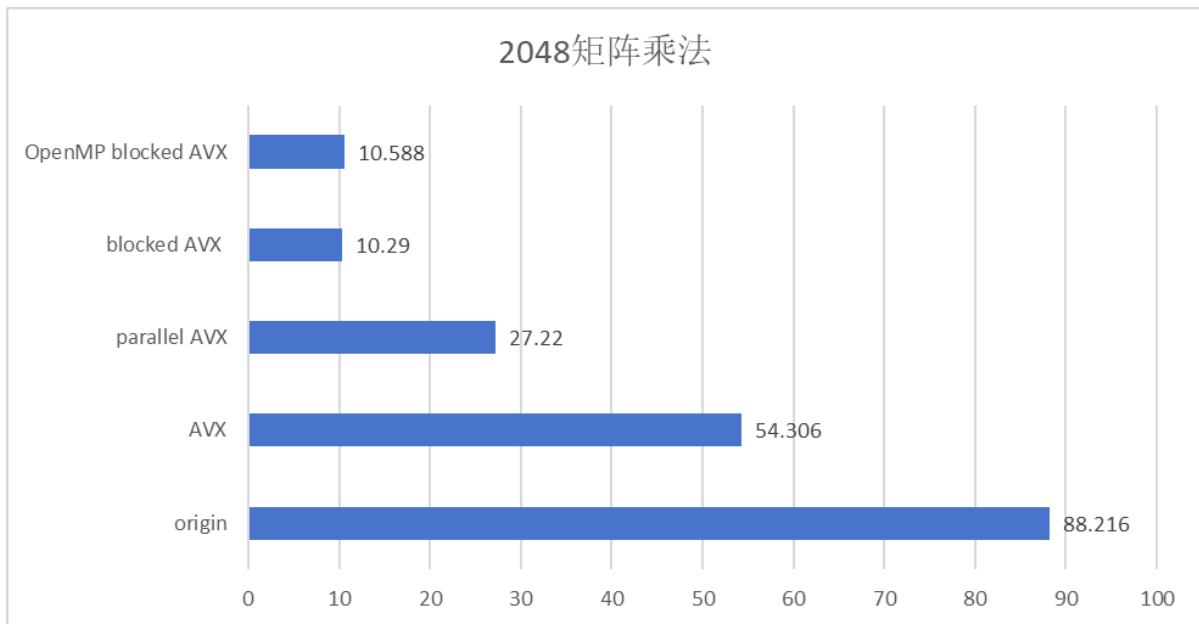
我们根据测试得到的时间做出统计, 给出以下的柱状图, 我们分为运行时间和 GFLOPS 这两个指标来评价我们的性能。

2.3.1 运行时间

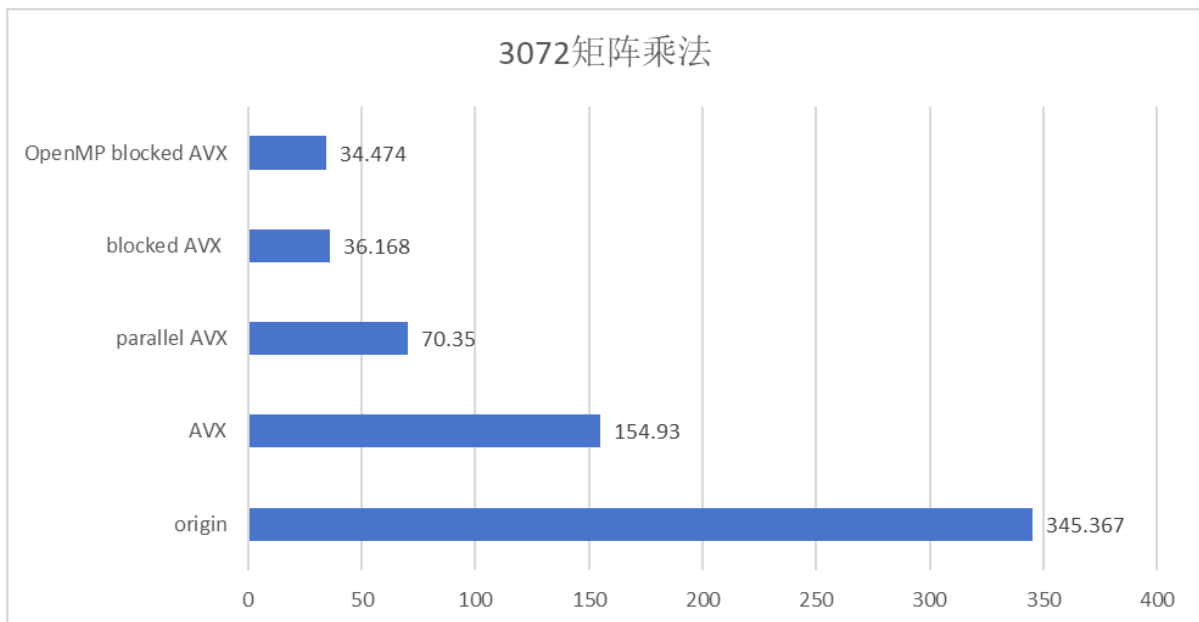
2.3.1.1 1024 × 1024



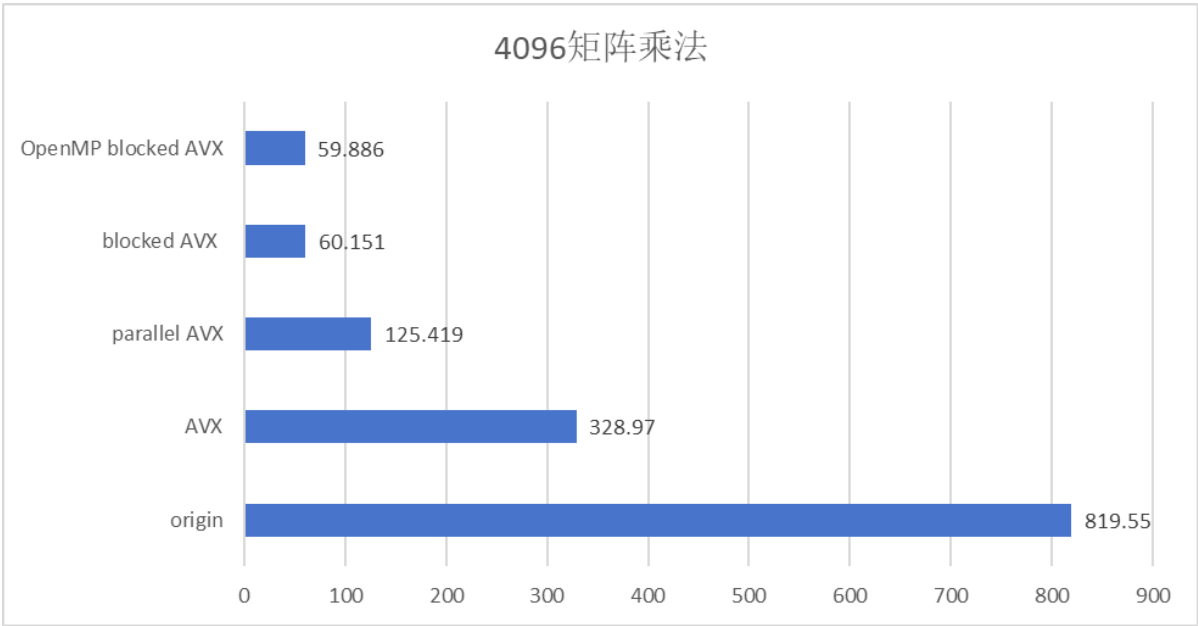
2.3.1.2 2048×2048



2.3.1.3 3072×3072

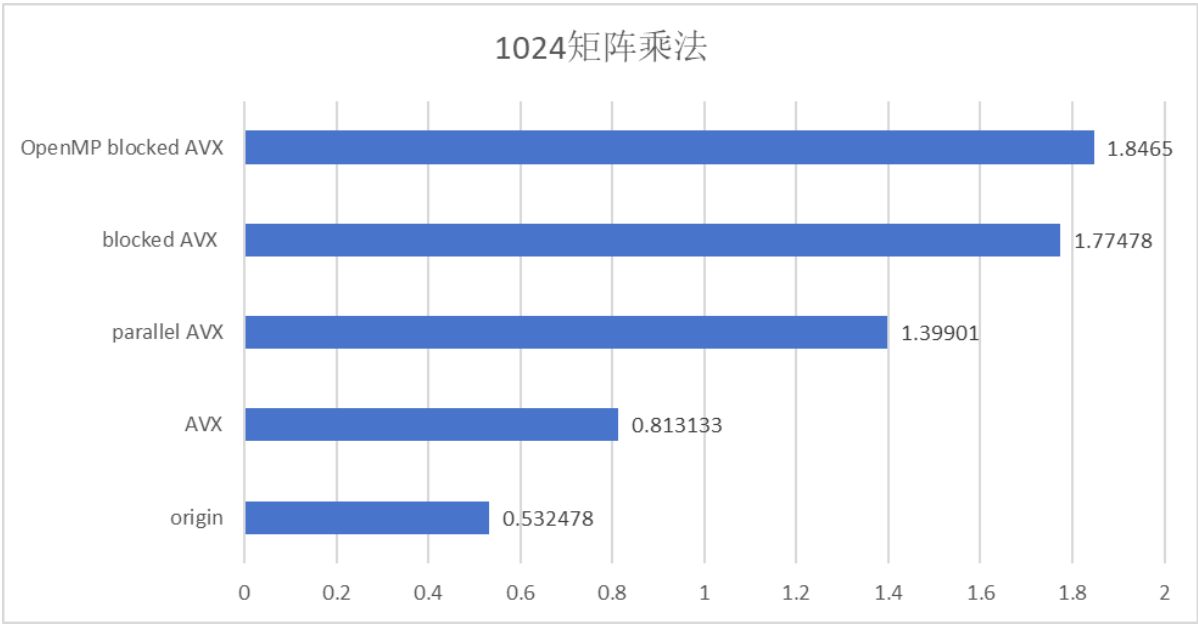


2.3.1.4 4096 × 4096

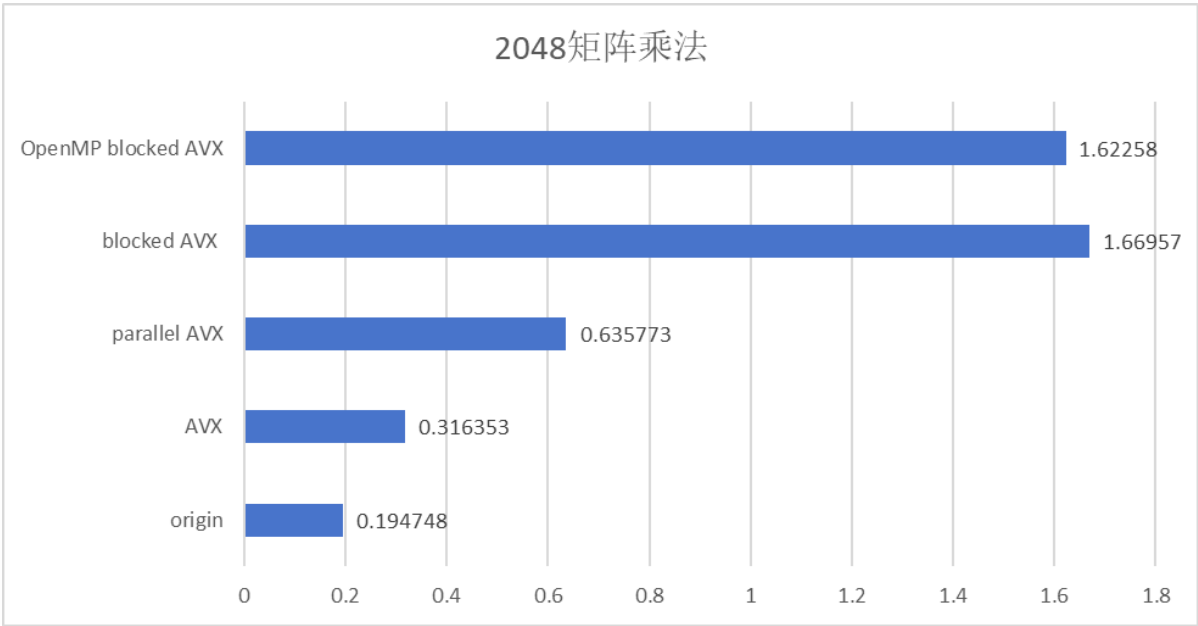


2.3.2 GFLOPS

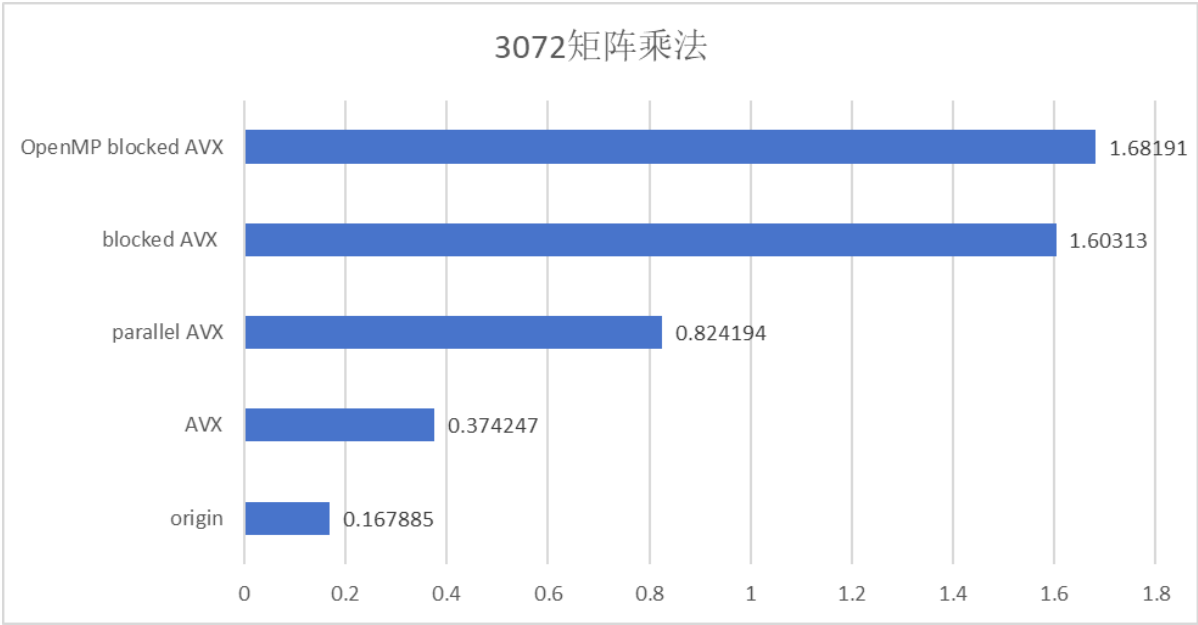
2.3.2.1 1024 × 1024



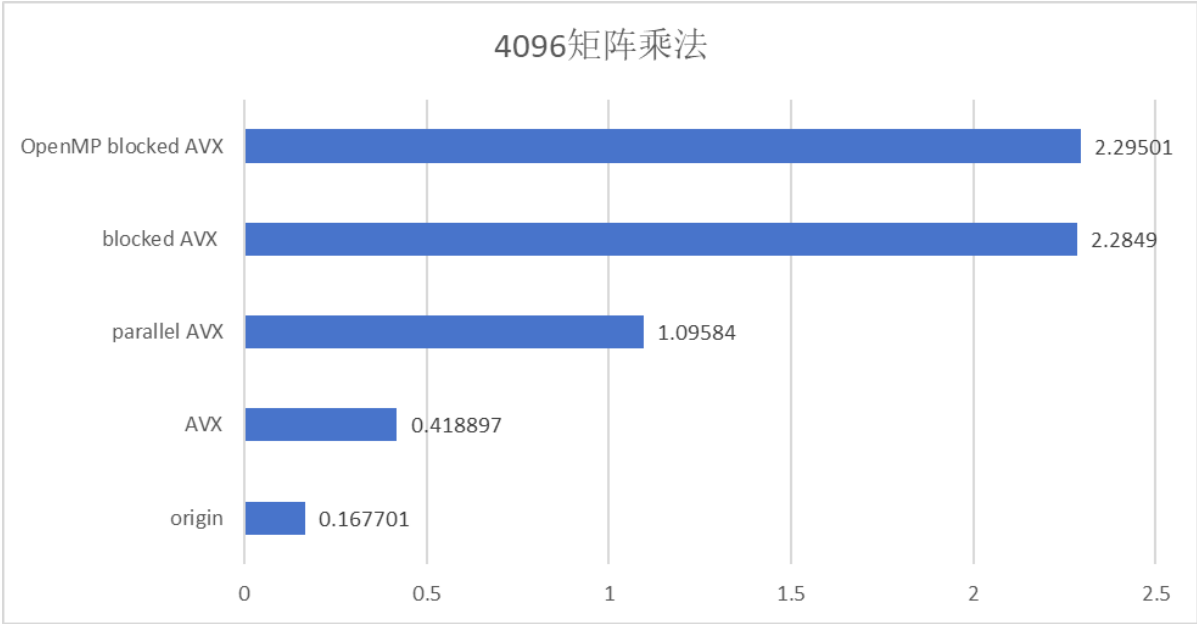
2.3.2.2 2048 × 2048



2.3.2.3 3072 × 3072



2.3.2.4 4096 × 4096



2.4 归纳与总结

通过以上的时间统计，我们发现了一些规律：

- 1. 从 blocked AVX 到 OpenMP blocked AVX 的优化，起到的作用比较小，都是优化了没几秒，甚至在2048矩阵乘法的时候性能还变差了，说明在1024 — 4096这个数量级的矩阵乘法中，这一步优化实际上没有起到很大的作用。
- 2. 我们发现，随着矩阵规模的增长，时间呈非线性增长，比如说，从1024到2048，理论上时间应该是乘上8，但是实际上时间却变大了22左右，与理论值还是有一定的差距的，说明一定还有一些别的影响因素在影响着这个矩阵计算。
- 3. 我们发现，在从 origin 到 AVX 的优化中，提升的效率是最高的，这说明我们从普通运算到子字并行的提升是最大的，从子字并行到指令集并行。

我们从计算耗时、运行性能以及加速比这三个方面来进行优化的评估。

计算耗时：

按照定义计算的矩阵乘法耗时最长，因为它需要执行三重循环来逐个元素进行计算；优化后的矩阵乘法采用分块矩阵乘法的方法，在矩阵乘法计算中减少了不必要的访存操作，从而提高了计算效率，耗时相对较短；并行计算矩阵乘法利用多线程进行计算，可以同时进行多个乘法运算，因此具有更快的计算速度，耗时最短。

运行性能：

按照定义计算的矩阵乘法的性能较低，因为它使用了三重循环的嵌套，导致计算复杂度较高；优化的矩阵乘法通过采用分块矩阵乘法的优化方法，减少了不必要的访存操作，提高了运行性能；并行计算矩阵乘法利用多线程实现并行计算，充分利用多核处理器的计算能力，因此具有更好的运行性能。

加速比：

优化的矩阵乘法和并行计算矩阵乘法相对于按照定义计算的矩阵乘法都能够取得较好的加速比；优化的矩阵乘法通过减少不必要的访存操作和利用分块矩阵乘法的优化策略，加速比相对较高；并行计算矩阵乘法通过利用多线程并行计算的特点，能够进一步提高计算速度，加速比最高。

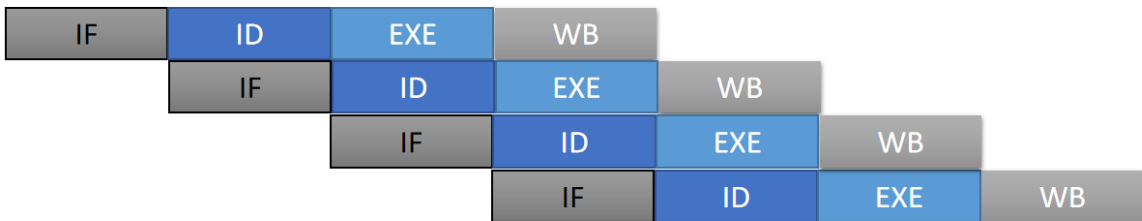
2.5 遇到的问题及解决方法

首先，我们根据大一时学习的C++编程的思想，简单的用三层嵌套的 `for` 循环来实现这个矩阵乘法，但是这样计算的话，时间复杂度会非常高，所耗时较长，性能也很差；于是我们考虑使用分块矩阵乘法等优化方法来减少不必要的计算和访存操作，从而提高计算效率。

$$\begin{aligned} c_{11} &= c_{11} + a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} + \cdots + a_{18} \cdot b_{81} \\ c_{21} &= c_{21} + a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31} + \cdots + a_{28} \cdot b_{81} \\ c_{31} &= c_{31} + a_{31} \cdot b_{11} + a_{32} \cdot b_{21} + a_{33} \cdot b_{31} + \cdots + a_{38} \cdot b_{81} \\ c_{41} &= c_{41} + a_{41} \cdot b_{11} + a_{42} \cdot b_{21} + a_{44} \cdot b_{41} + \cdots + a_{48} \cdot b_{81} \end{aligned}$$

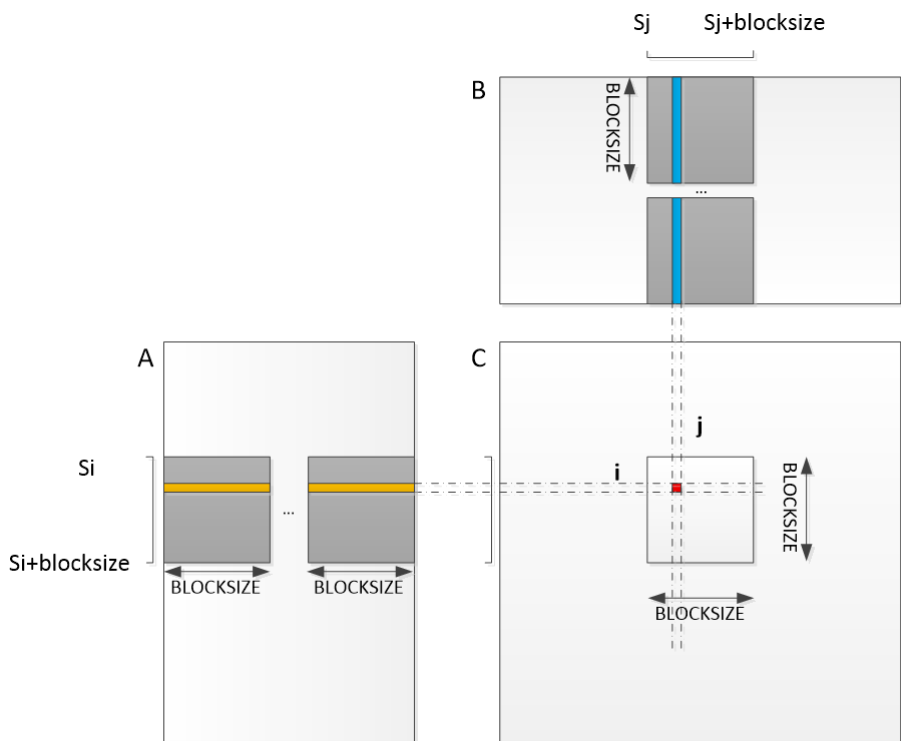
上图就是我们使用子字并行的计算原理，可以发现在这一步我们所提升的性能与加速都是最好的，因为我们同时计算了四个数，相当于并行了4核计算，相当于将速度提升了4倍左右，但是实际上可能提升仅仅只有两倍左右。

接下来我们就使用理论课上讲授的流水线并行，指令集并行的 `GEMM` 来进行优化。



我们使用 `unroll` 循环展开次数，控制指令依次发射执行理论上至少有1倍的加速比，对照前面的结果，确实加速比来到了一倍左右。

然后我们进一步优化，利用理论课第五章的内容，使用 `cache` 命中来加速，在一个子矩阵（块）被 `cache` 替换出去之前，最大限度的对其进行数据访问利用时间局部性，提高 `cache` 命中率，注意到此处，我们需要通过实验和测试来选择合适的分块大小，以达到最佳性能。



最后，我们利用并行程序设计的思想，将该程序使用4个处理器利用4线程并行来进行加速，但是发现该部分提升的时间和性能并不是特别多，并行计算可能会引发数据竞争和同步问题，导致结果错误或性能下降。所以我们可以使用线程同步机制，如互斥锁（`mutex`）、信号量（`semaphore`）等来解决数据竞争和同步问题。确保每个线程访问共享资源时的互斥和同步操作，以保证正确的计算结果和高效的并行计算。

经过优化发现性能仍然不高，甚至有几组数据的 `openmp` 优化的效率还不如前面的分块运算，所以我们初步认为前面的 `cache` 优化和流水线优化已经能够很好的优化该计算了。

三、Taishan服务器的实验

3.1 源代码

```
#include <iostream>
#include <ctime>
#include <omp.h>

#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));
    std::cout << "GFLOPS:\t" << flops << std::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;
        }
}

// 1. 原始GEMM
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;
        }
}

// 2. 指令集并行优化 (ILP)
void ILP_dgemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    // 将内层循环拆分为多个小循环，以提高指令级并行性
    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 += 4) { // 假设处理器支持4个并行乘法
            cij += A[i + k * n] * B[k + j * n];
            cij += A[i + (k + 1) * n] * B[(k + 1) + j * n];
            cij += A[i + (k + 2) * n] * B[(k + 2) + j * n];
            cij += A[i + (k + 3) * n] * B[(k + 3) + j * n];
        }
        C[i + j * n] = cij;
    }
}

```

// 3. 考虑Cache的分块优化

```

void block_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    int BLOCK_SIZE = 64; // 块大小
    for (int bi = 0; bi < n; bi += BLOCK_SIZE)
        for (int bj = 0; bj < n; bj += BLOCK_SIZE)
            for (int bk = 0; bk < n; bk += BLOCK_SIZE)
                for (int i = bi; i < std::min(n, bi + BLOCK_SIZE); ++i)
                    for (int j = bj; j < std::min(n, bj + BLOCK_SIZE); ++j) {
                        REAL_T cij = C[i + j * n];
                        for (int k = bk; k < std::min(n, bk + BLOCK_SIZE); ++k) {
                            cij += A[i + k * n] * B[k + j * n];
                        }
                        C[i + j * n] = cij;
                    }
}

```

```
#define BLOCKSIZE 64
```

```

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;
        }
}

```

// 4. 多处理器并行的优化

```

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; // 1024-4096修改

    A = new REAL_T[n * n];
    B = new REAL_T[n * n];
    C = new REAL_T[n * n];
    initMatrix(n, A, B, C);
}

```

```

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

initMatrix(n, A, B, C);
std::cout << "ILP calculation begin...\n";
start = clock();
ILP_dgemm(n, A, B, C);
stop = clock();
std::cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
printFlops(n, n, n, start, stop);

initMatrix(n, A, B, C);
std::cout << "blocked calculation begin...\n";
start = clock();
block_gemm(n, A, B, C);
stop = clock();
std::cout << (stop - start) / CLOCKS_PER_SEC << "." << (stop - start) %
CLOCKS_PER_SEC << "\t\t";
printFlops(n, n, n, start, stop);

initMatrix(n, A, B, C);
std::cout << "OpenMP blocked calculation begin...\n";
start = clock();
omp_gemm(n, A, B, C);
stop = clock();
std::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;
}

```

3.2 Taishan服务器运行

首先我们根据老师给出的服务器端口来连接系统，输入账号 `stu2211044` 和密码 `123456` 就可以进入系统了。

```

stu2211044@parallel542-taishan200-1: ~
login as: stu2211044
stu2211044@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 Sun 28 Apr 2024 02:09:04 AM UTC

System load:  2.0           Processes:            1103
Usage of /:   7.2% of 195.86GB Users logged in:        3
Memory usage: 3%           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

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

New release '22.04.3 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.

stu2211044@parallel542-taishan200-1:~$

```

输入命令行 `gcc -v` 和 `g++ -v` 来检验系统是否安装编译环境，发现输出了对应的版本号9.4.0，说明系统中存在我们所需要的编译环境。

```

stu2211044@parallel542-taishan200-1:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/aarch64-linux-gnu/9/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.4.0-1ubuntu1~20.04.2' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-l
er --with-gcc-major-version-only --program-suffix=-9 --program-prefix=aarch64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib
/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-ob
nable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --enable-fix-cortex-a53-84
h64-linux-gnu --host=aarch64-linux-gnu --target=aarch64-linux-gnu
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)
stu2211044@parallel542-taishan200-1:~$ g
g: command not found
stu2211044@parallel542-taishan200-1:~$ g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib/gcc/aarch64-linux-gnu/9/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.4.0-1ubuntu1~20.04.2' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-l
er --with-gcc-major-version-only --program-suffix=-9 --program-prefix=aarch64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib
/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-ob
nable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --enable-fix-cortex-a53-84
h64-linux-gnu --host=aarch64-linux-gnu --target=aarch64-linux-gnu
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)
stu2211044@parallel542-taishan200-1:~$

```

我们在本地创建自己的目录，获取到了我们的代码存放的位置：`/home/stu2211044/code`，接下来我们配置我们的编译环境，结果如下所示：可以看到我们已经完成了自己的编译环境的搭建！

然后我们通过 `vim` 和 `g++` 来进行代码的编辑与编译运行，就可以得到最后的结果了，我们修改 `n` 的值，分别计算出 `1024 - 4096` 的不同结果。

第一个文件 `final.cpp` 跑出来的结果如下所示：

```

stu2211044@parallel542-taishan200-1:~$ g++ final.cpp -o final
stu2211044@parallel542-taishan200-1:~$ ./final
origin calculation begin...
18.158558          GFLOPS: 0.118263
ILP calculation begin...
16.755988          GFLOPS: 0.128162
blocked calculation begin...
10.541150          GFLOPS: 0.203724
OpenMP blocked calculation begin...
7.952775           GFLOPS: 0.270029
stu2211044@parallel542-taishan200-1:~$

```

第二个文件 `final2.cpp` 跑出来的结果如下所示:

```

stu2211044@parallel542-taishan200-1:~$ vim final2.cpp
stu2211044@parallel542-taishan200-1:~$ g++ final2.cpp -o final2
stu2211044@parallel542-taishan200-1:~$ ./final2
origin calculation begin...
218.639354         GFLOPS: 0.0785763
ILP calculation begin...
188.214997         GFLOPS: 0.0912779
blocked calculation begin...
85.698506          GFLOPS: 0.200469
OpenMP blocked calculation begin...
63.52740           GFLOPS: 0.272468
stu2211044@parallel542-taishan200-1:~$

```

第三个文件 `final3.cpp` 跑出来的结果如下所示:

```

stu2211044@parallel542-taishan200-1:~$ vim final3.cpp
stu2211044@parallel542-taishan200-1:~$ g++ final3.cpp -o final3
stu2211044@parallel542-taishan200-1:~$ ./final3
origin calculation begin...
795.192457         GFLOPS: 0.0729158
ILP calculation begin...
679.714979         GFLOPS: 0.0853035
blocked calculation begin...
322.484691         GFLOPS: 0.179798
OpenMP blocked calculation begin...
219.707196         GFLOPS: 0.263906
stu2211044@parallel542-taishan200-1:~$

```

第四个文件 `final4.cpp` 跑出来的结果如下所示:

```

stu2211044@parallel542-taishan200-1:~$ cat nohup.out
origin calculation begin...
1939.677475        GFLOPS: 0.0708566
ILP calculation begin...
1680.863441        GFLOPS: 0.0817669
blocked calculation begin...
907.900103         GFLOPS: 0.151381
OpenMP blocked calculation begin...
564.333315         GFLOPS: 0.243542
stu2211044@parallel542-taishan200-1:~$

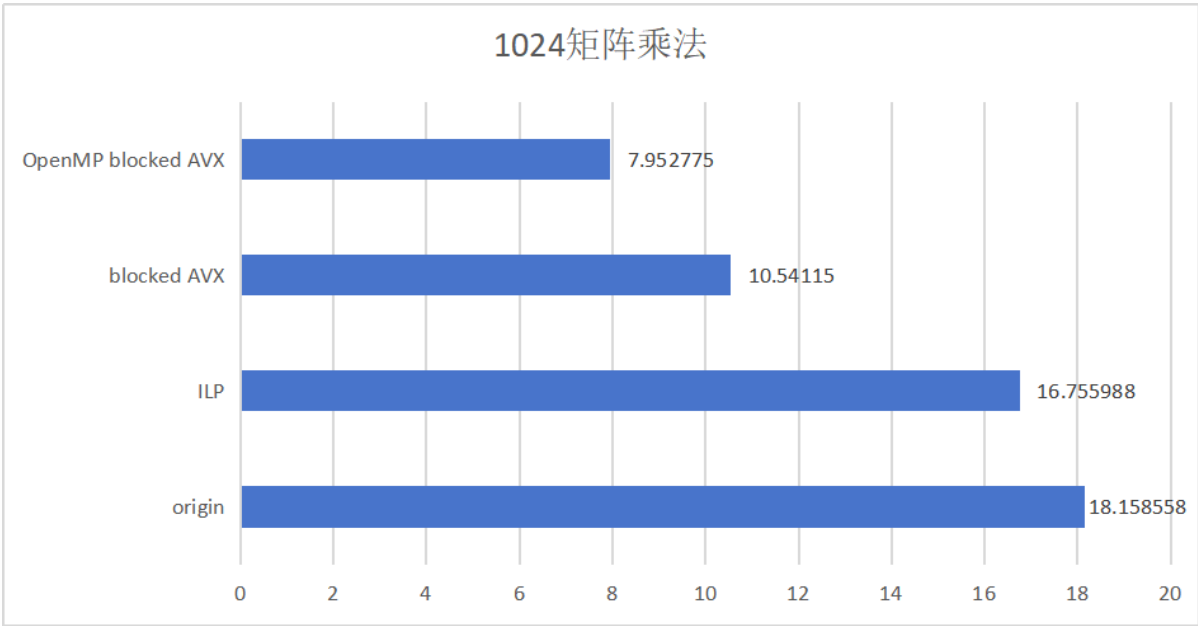
```

因为此处运行时间过长, 所以我将它放到了后台运行, 最后使用命令行 `cat nohup.out` 来查看最后的结果。

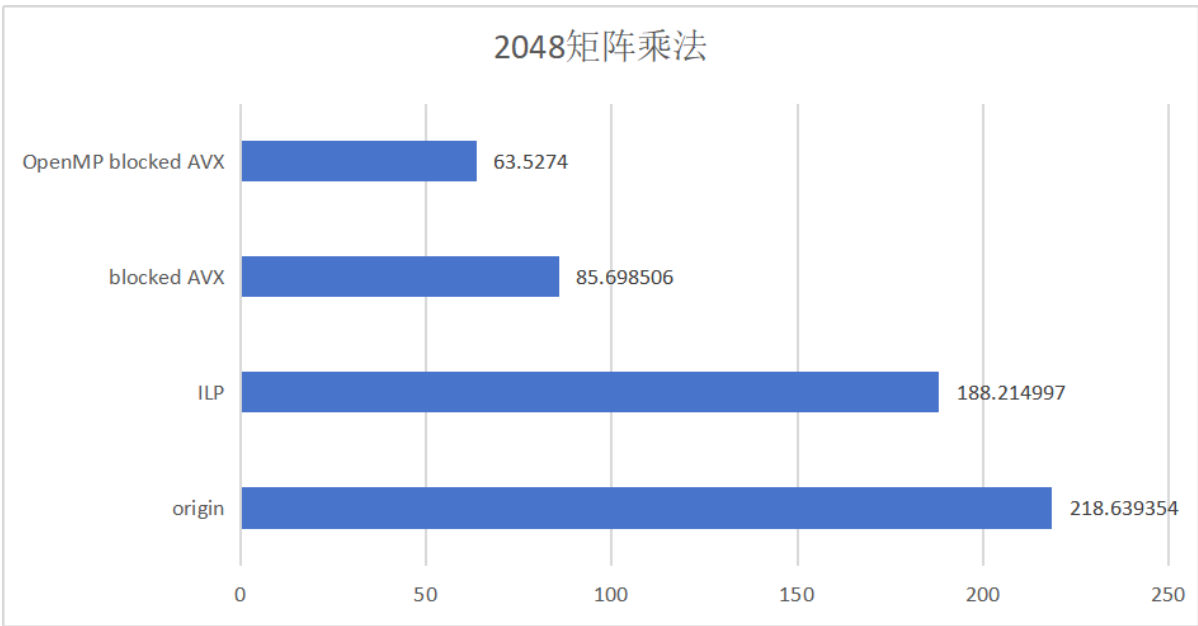
3.3 测试结果

3.3.1 运行时间

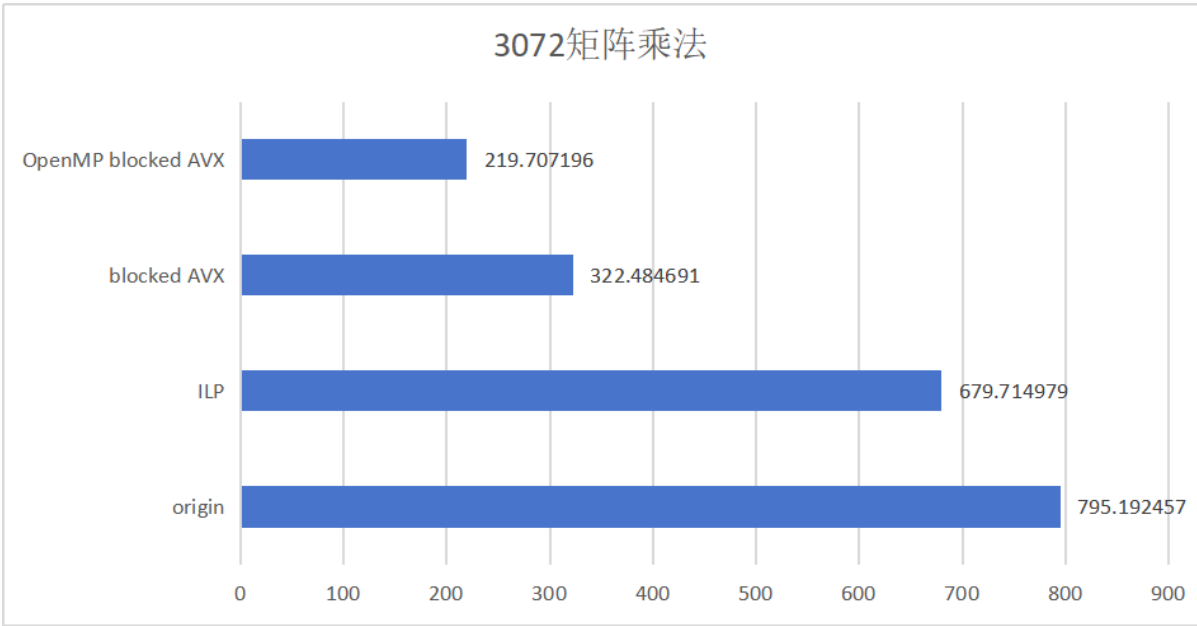
3.3.1.1 1024×1024



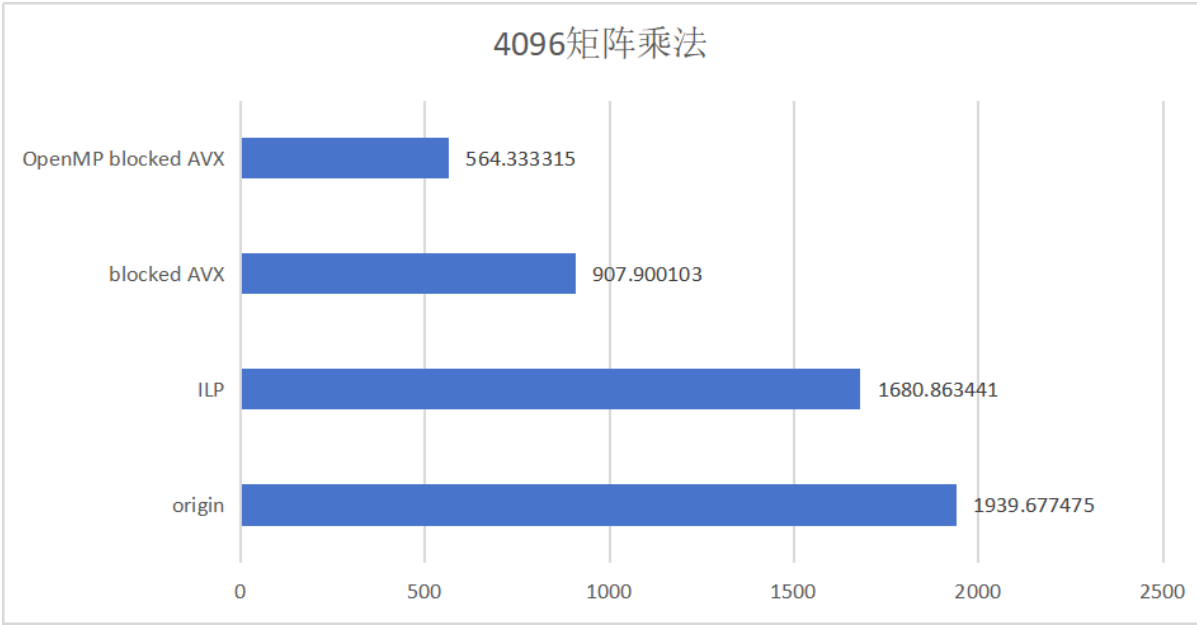
3.3.1.2 2048×2048



3.3.1.3 3072 × 3072

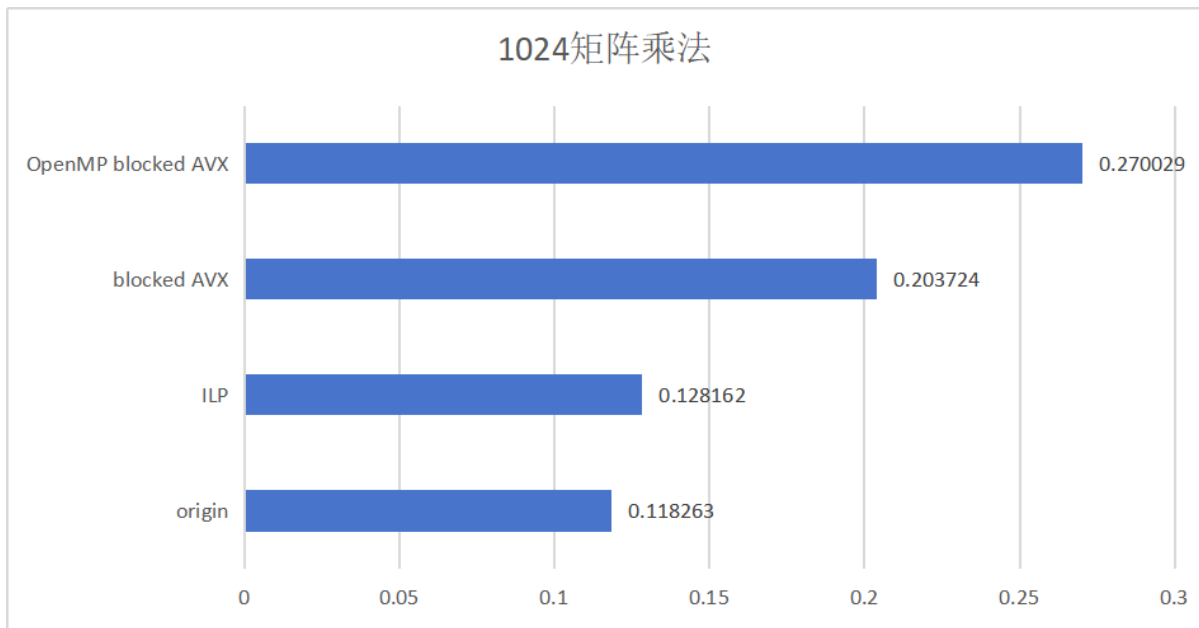


3.3.1.4 4096 × 4096

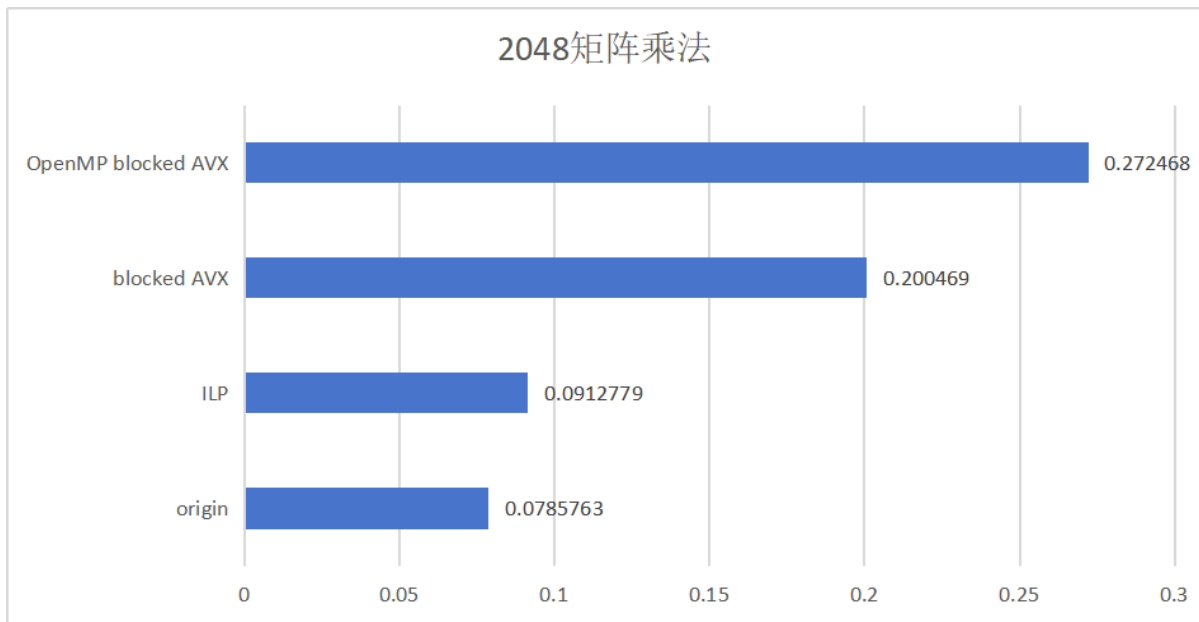


3.3.2 GFLOPS

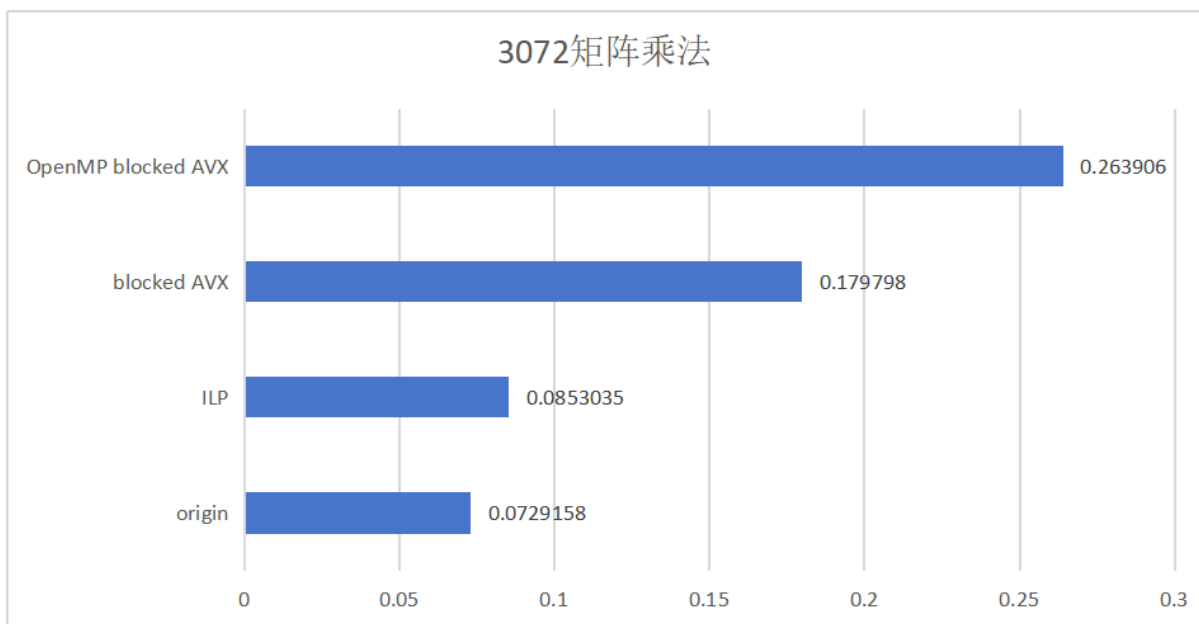
3.3.2.1 1024 × 1024



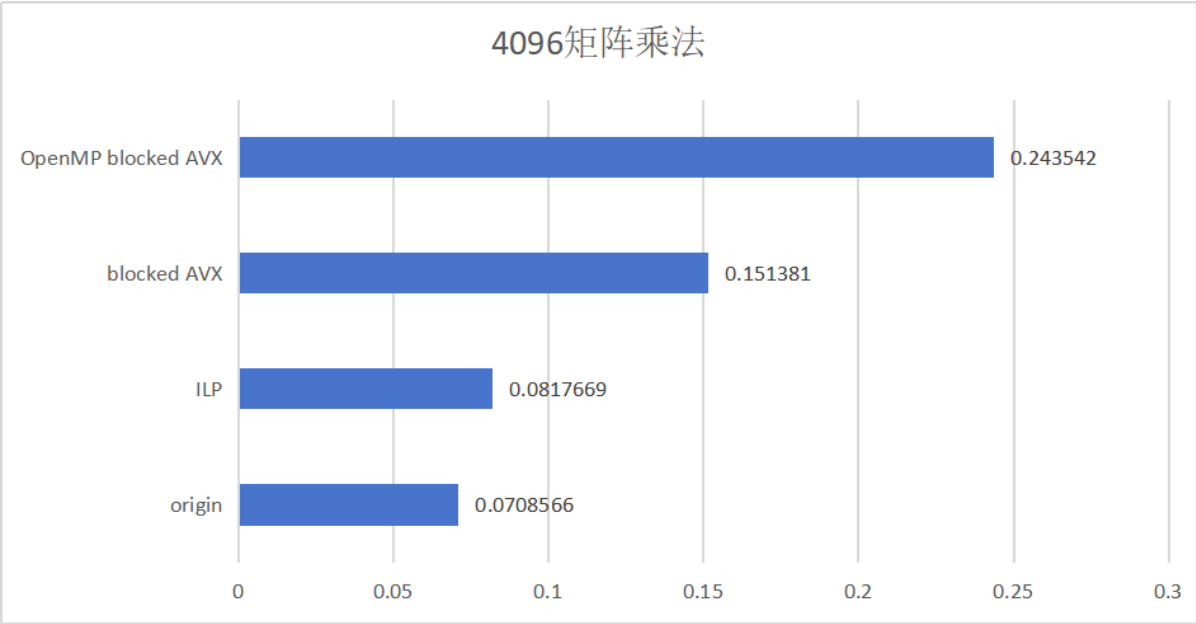
3.3.2.2 2048 × 2048



3.3.2.3 3072 × 3072



3.3.2.4 4096 × 4096



3.4 归纳与总结

通过以上的时间统计，我们发现了一些规律：

- 1. 我们发现使用 taishan 服务器运行得出的结果并没有比在 PC 电脑上得出的结果要好，反而性能还差于 PC 电脑。
- 2. 在性能的替身方面，也与 PC 电脑端差距较大：PC 端提升较快的是第一级到第二级，而 taishan 服务器提升较快的是第二级到第三级，有一定的差距。
- 3. 在 openMP 的使用方面，taishan 服务器的表现要明显优于 PC 端，这是因为在服务器上运行并行程序会大大加速程序的速度，而在个人 PC 上就不一定会加速很多了。

我们从计算耗时、运行性能以及加速比这三个方面来进行优化的评估。

计算耗时：

按照定义计算的矩阵乘法耗时最长，因为它需要执行三重循环来逐个元素进行计算；优化后的矩阵乘法采用分块矩阵乘法的方法，在矩阵乘法计算中减少了不必要的访存操作，从而提高了计算效率，耗时相对较短；并行计算矩阵乘法利用多线程进行计算，可以同时进行多个乘法运算，因此具有更快的计算速度，耗时最短。

运行性能：

按照定义计算的矩阵乘法的性能较低，因为它使用了三重循环的嵌套，导致计算复杂度较高；优化的矩阵乘法通过采用分块矩阵乘法的优化方法，减少了不必要的访存操作，提高了运行性能；并行计算矩阵乘法利用多线程实现并行计算，充分利用多核处理器的计算能力，因此具有更好的运行性能。

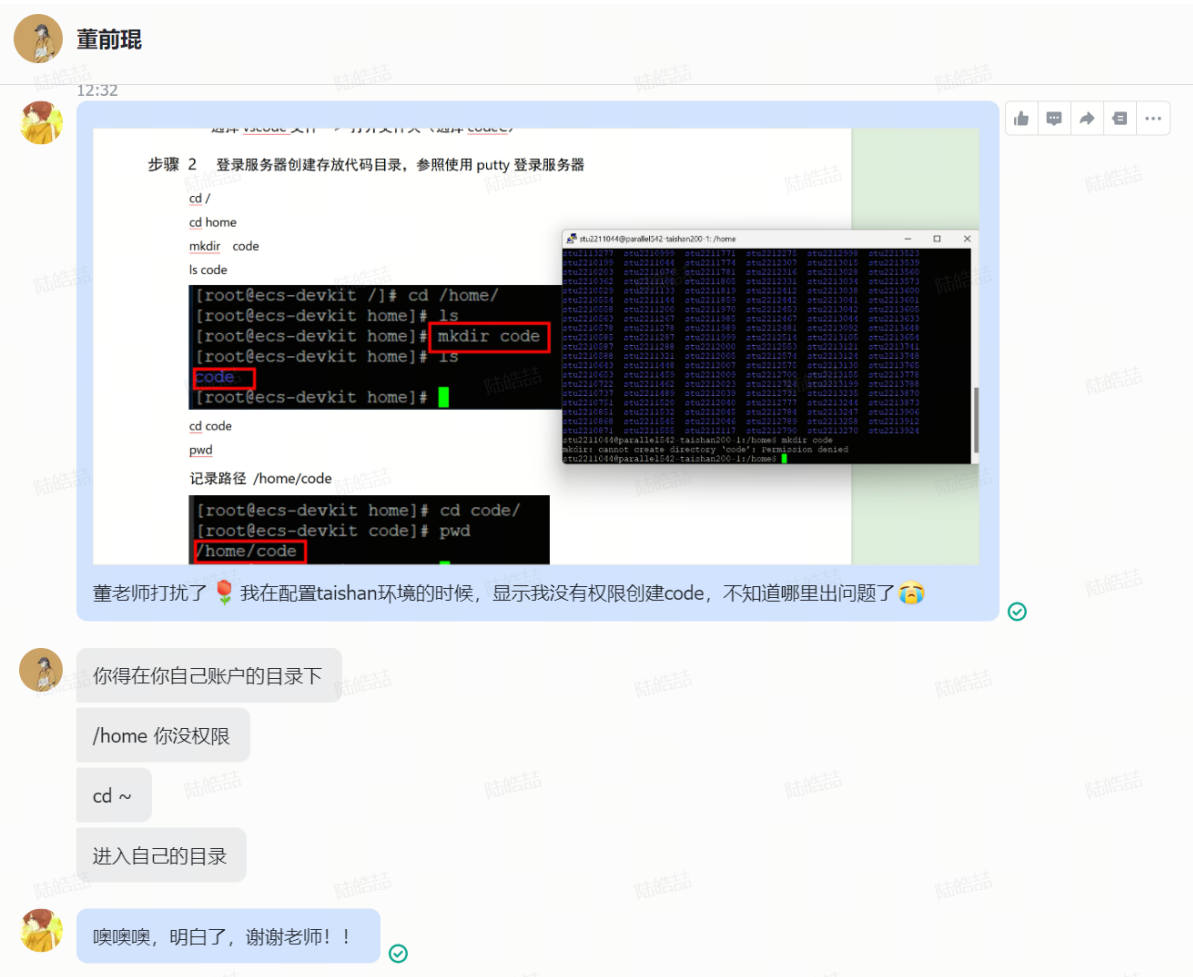
加速比：

优化的矩阵乘法和并行计算矩阵乘法相对于按照定义计算的矩阵乘法都能够取得较好的加速比；优化的矩阵乘法通过减少不必要的访存操作和利用分块矩阵乘法的优化策略，加速比相对较高；并行计算矩阵乘法通过利用多线程并行计算的特点，能够进一步提高计算速度，加速比最高。在服务器上的表现是明显要优于个人电脑的。

3.5 遇到的问题及解决方法

遇到的问题有挺多的，比如说刚开始的配 taishan 环境，以及后面的运行得出的结果，都遇到了很多的困难，下面我一一列举：

首先就是对于环境的配置，我先根据教程打开 putty，然后输入 ip 地址，就进入了一个命令行输入的一个界面。我是第一次自己编写 linux 命令行语句来操作，所以并不是很熟悉，我也在网上查了很多教程，但是还是无法进入我的 home 界面，然后我就询问了老师，老师也给了我简洁清晰的回答。



最后根据老师的说明，我成功的找到了我的 home 地址。接下来我就准备将修改好的代码进行运行，但是一时间不知道如何将代码导入到我的服务器端。我在网上查询了很多资料，有的说需要下载一个 fftp 和 xshe11 来传输文件，有的说不用下载，直接传输就可以了。最后，我仔细阅读参考资料，最后使用在服务器自己利用 vim 语句建立一个文件，再使用 g++ 来进行编译的方法，最后解决了这个问题。这次实验，让我初步了解了 linux 语句的使用，也扩展了我的知识面。

四、PC与taishan服务器的差异

原先我觉得，个人 PC 端跑出来的结果一定会差于 taishan 服务器端口，但是事实上并不是这样的。我发现在所有的测试点上，PC 基本都是要优于 taishan 服务器的，不管是在时长上，还是在性能方面，taishan 服务器的性能居然上限就只有 0.27 左右，但是 PC 端却可以达到 2 以上，这也颠覆了我的认知。

在提升加速比这一块，我发现了：个人 PC 在第一级的提升是最快的，但是在并行这一级的提升接近 0，有的甚至还倒退了；taishan 服务器的话，在第二级到第三级的流水线是提升最快的一级，但是在第一级的提升就要明显弱于 PC，我猜测是其巨大的算力导致无法进行四核运算（不太确定），但是在并行这一级，我们的服务器提升是巨大的，几乎有了接近一倍的提升，这也与我们的 PC 不太相同。

五、总结感想

通过此次实验，我了解了个人 PC 与 taishan 服务器的不同，深入了解了矩阵乘法的逐级优化的原理与上手实操，了解了 linux 命令行的语句编写，更深入熟悉了性能，时间，加速比等基础概念，了解到了两者的差异与共同之处，学会了使用 putty 来进行 SSH 的客户端连接等等。这次作业很好的将理论课上所学习的流水线、cache 缓存加速、并程序加速等知识点与实验结合在了一起，是一次很好的锻炼，希望以后有更多这样的实验。