南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计期末实验报告

高斯消去法的并行优化

姓名：　姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2024 年 7 月 2 日

摘要

本报告是本学期的并行程序课程期末报告，针对高斯消去算法和 Gröbner 基计算高斯消去算法进行了 SIMD，Pthread，OpenMP，MPI，CUDA 等并行化优化。对于每个并行化优化问题，采取不同的优化结构、优化部分、优化进程/线程/block 大小等进行测试，并将多种优化方式结合起来进行进一步的讨论。通过计时的性能测试和 profiling 对优化程序进行进一步分析，并对运行结果加以总结。

关键字：并行；SIMD；Pthread；OpenMP；MPI；CUDA；高斯消去；Gröbner 基

目录

一、 高斯消去算法的并行优化

（一） 问题背景与重述

1. 问题背景

高斯消去法（Gauss–Jordan elimination）是求解线性方程组的经典算法，它在当代数学中有着重要的地位和价值，是线性代数课程教学的重要组成部分。高斯消去法除了用于线性方程组求解外，还可以用于行列式计算、求矩阵的逆，以及其他计算机和工程方面。

高斯消去法以著名德国数学家 Carl Friedrich Gauss(1777-1855) 命名。Gauss 被认为是历史上最重要的数学家之一，他在数学的众多分支，如数论、代数、分析、微分几何等以及统计学、物理学、天文学、大地测量学、地理学、电磁学、光学等领域都有重要的贡献。Gauss 还享有“数学王子”的美誉. 值得一提的是，这种解线性方程组的消元法最早出现在中国古代数学著作《九章算术》中，相关内容在大约公元前 150 年前就出现了。

2. 串行实现

普通高斯消去的流程如图1所示:

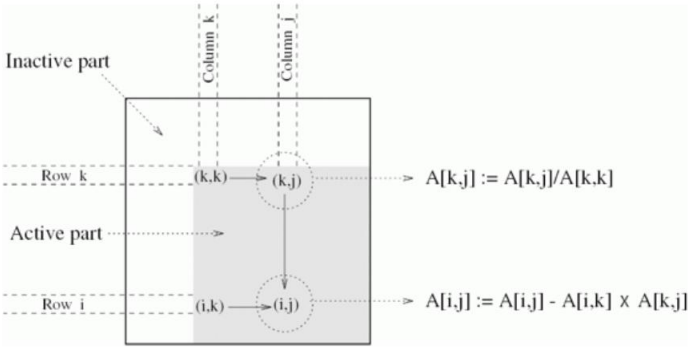


图 1: 普通高斯消去流程

该算法的主要流程是在第 k 步的时候，对第 k 行进行除法运算使得 (k,k) 位置元素变为 1，然后对后续行减去第 k 行的相应倍数，使得后续行第 k 列的元素均为 0，然后进行下一步循环。

在一般情况下，消元算法的思路如下述伪代码所示。

Input: 原始矩阵 A,A 的阶数 n
Output: 消元结果矩阵 A

```
1: function LU(A, n)
2:     for k = 1, 2, ··· , n do
3:         for j = k + 1, k + 2, ··· , n do
4:             A[k][j] ← A[k][j]/A[k][k]
5:         end for
6:         A[k][k] ← 1.0
7:         for i = k + 1, k + 2, ··· , n do
8:             for j = k + 1, k + 2, ··· , n do
9:                 A[i][j] ← A[i][j] − A[i][k] ∗ A[k][j]
10:            end for
```

```
            A[i][k] ← 0.0
        end for
    end for
end function
```

消去步骤结束后的结果如图2所示：

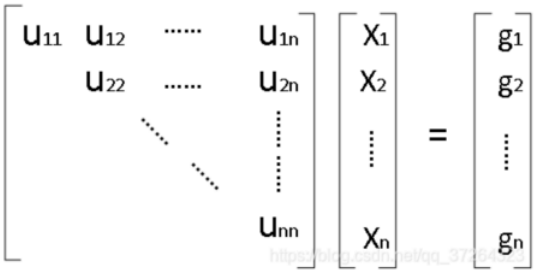


图 2: 消去过程结果演示

在消元步骤结束后，通过一个较为简单的回代步骤，即可完成各个方程组的求解。回代步骤思路如下所示。

Input: 消元结果矩阵 A，A 的阶数 n
Output: 结果数组 X

```
function GEN(A, n, X)
    for i = n, n − 1, ··· , 1 do
        X[i] ← A[i][n + 1]/A[i][i]
        for j = i − 1, i − 2, ··· , 1 do
            A[j][n + 1] ← A[j][n + 1] − X[i] ∗ A[j][i]
        end for
    end for
end function
```

3. 并行优化

在本实验中，主要针对消元部分进行并行化的优化。在高斯消元法化上三角矩阵的过程中，每一次清除一列元素时，每一行的元素需要进行一次数字运算，行与行之间没有数据依赖，可以实现并行化。此外，在 SIMD 向量化优化中，兼有对回代部分优化的对比。

在串行算法中，这部分的时间复杂度来到 $0(n^3)$，因此，如果能通过较好的划分方式对其进行优化，加速比是颇为理想的。

在对高斯消去算法的并行化优化求解中，我将分别通过 SIMD 向量化优化，Pthread 多线程优化，OpenMP 多线程优化，MPI 多进程优化以及通过 CUDA 发挥 GPU 性能的方式对该问题进行处理，在每一部分中，我考虑了多种实现方式，此外为追求更高效率，我也对多方法的结合也进行了讨论。

4. 核心问题

- 数据有效性

  如不对数据进行较好的初始化，可能会导致计算结果出现 nan 和 inf，这不但会大幅影响计算精度，也会对计时的可靠性造成影响。

- 计时方式

  考虑数据精度的划分，尽可能的在计算中统计到有效计算部分，能够更好的确定运行时间。在本实验中，采取了多种计算时间的方式。我主要设计的是通过 chrono 库完成计算时间。在 MPI 实验中，我通过 MPI 的自带的计时方式获得了更加精确的结果。

（二） 算法设计与编程实现

1. 平凡算法

为进行后续实验，提供两个版本的平凡算法，即仅包含消元的平凡算法，以及包含消元和回代的平凡算法。以下是 c++ 的实现方式。

在算法设计部分的“SIMD 向量化优化”小节中，给出了各优化算法分别是对哪一个平凡算法进行优化。对于后续多线程/多进程/GPU 优化，均为针对仅包含消元的平凡算法进行优化。

在后续性能测试与分析中，第1和第2小节是基于包含消元和回代的平凡算法进行优化，以更全面的认识 SIMD 划分所造成的影响。其余小节是基于仅包含消元的平凡算法进行优化，以简化问题结构，在更多情况下探讨问题。

包含消元和回代的平凡算法

```
void common(int n){
    for(int k=0;k<n;k++) {
        for (int i=k+1;i<n;i++){
            float factor=A[i][k]/A[k][k];
                for (int j=k+1;j<n;j++) A[i][j]-=factor*A[k][j];
                b[i]-=factor*b[k];
        }
    }
    x[n-1]=b[n-1]/A[n-1][n-1];
    for(int i=n-2;i>=0;i--){
        float sum=b[i];
        for(int j=i+1;j<n;j++) sum-=A[i][j]*x[j];
        x[i]=sum/A[i][i];
    }
}
```

仅包含消元的平凡算法

```
void common(int n) {
    for (int k=0;k<n;++k) {
        for (int j=k+1;j<n;++j)
            A[k][j]/=A[k][k];
        A[k][k]=1.0;
        for (int i=k+1;i<n;++i) {
            for (int j=k+1;j<n;++j) A[i][j]-=A[i][k]*A[k][j];
            A[i][k]=0;
        }
    }
}
```

2. 计时、验证等函数实现

a) 计时函数

- chrono 计时

  在本实验中主要通过以下两种类型的 chrono 实现测量。

  high_resolution_clock：旨在提供最高的分辨率，但可能不是单调的。

  steady_clock：提供单调时间，不受系统时钟调整的影响，但可能不是最高分辨率的。

  在便于进行重复实验或者单次实验足够长的情况下，我尽可能的选用 steady_clock，因为它足够可靠。然而，在单次运行时间足够短或者实验之间需要进行初始化的背景下，我使用 high_resolution_clock，以便提供更细粒度的分析。

- MPI 计时

  通过 MPI 自带的 MPI_Wtime() 获取当前时间，通过开始时间和结束时间相减确定运行时间。MPI_Wtime() 的计时一般可以达到微妙级别，因此在 MPI 相关编程的实验中，均使用 MPI 计时。

b) 验证方式

通过保存平凡算法的计算结果并且优化算法对比输出的形式，确认优化算法实际上的有效性。考虑到浮点计算中不同的计算顺序可能会造成计算结果的误差（即不具有分配律），因此直接通过打印的形式直接进行比对，可以较好的确认优化算法的正确性。

c) 矩阵 reset

为避免 inf/nan 的出现，增加计算的可靠性，我选择的初始化方式为先按照消元的结果矩阵初始化，然后从上到下累加。这样初始化可以确保结果矩阵是符合我们设定的数据范围的。

Input: 待初始化的矩阵 A
Output: 初始化完成的矩阵 A

```
function RESET(n)
    seed ← time(NULL)
    srand(seed)
    for i = 0 to n − 1 do
        for j = 0 to i − 1 do
            A[i][j] ← 0
        end for
        A[i][i] ← 1.0
        for j = i + 1 to n − 1 do
            A[i][j] ← (rand() mod 1000 + 1)/10.0
        end for
    end for
    for k = 0 to n − 1 do
        for i = k + 1 to n − 1 do
            for j = 0 to n − 1 do
                A[i][j] ← A[i][j] + A[k][j]
            end for
```

```
18:         end for
19:     end for
20: end function
```

对于 MPI 环境,需要添加 barrier 来保证同步。对于 GPU 环境,定义了函数 initialize_matrix 进行初始化并解决冲突问题。相关代码可以由GitHub 仓库中查看。

3. SIMD 向量化优化

SIMD 向量化优化将主要分为适用于 x86 平台的 SSE/AVX 向量化优化, 和适用于 ARM 平台的 NEON 向量化优化。

a) NEON 向量化优化

在 NEON 向量化优化中，考虑不同的划分方式。

- NEON 优化 1:（对仅包含消元的平凡算法）内层（对应代码第 7 行）的循环优化。

  优化后的内层循环如下图所示。

```
1: for i = 0 to n − 1 do
2:     for k = 0 to i − 1 do
3:         aik ← vmovq(A[i][k])
4:         j ← k + 1
5:         while j ≤ n − 4 do
6:             akj ← vld1q(A[k] + j)
7:             multi ← aik × akj
8:             aij ← vld1q(A[i] + j)
9:             aij ← vsubq(aij, multi)
10:            vst1q(A[i] + j, aij)
11:            j ← j + 4
12:        end while
13:        for j = j to n − 1 do
14:            A[i][j] ← A[i][j] − A[i][k] × A[k][j]          ▷ 处理剩余元素
15:        end for
16:        A[i][k] ← 0
17:    end for
18: end for
```

- NEON 优化 2:（对包含消元和回代的平凡算法）对消元的内层循环进行优化（对应代码第 5 行）。

  相关语法与上述优化方式相同，不再列出。

- NEON 优化 3:（对包含消元和回代的平凡算法）消元部分外层（对应代码 3-7 行）直接进行展开，核心代码展示如下。

```
1: for i = k + 1 to n − 4 step 4 do
2:     factor4 ← vld1q(A[i] + k)
3:     vsetq(factor4, 1, A[i + 1][k])
4:     vsetq(factor4, 2, A[i + 2][k])
```

```
5:    vsetq(factor4, 3, A[i + 3][k])
6:    akk ← vmovq(A[k][k])
7:    factor4 ← vdivq(factor4, akk)
8:    j ← n − 1
9:    while j ≥ k and (j − k) mod 4 ≠ 0 do
10:       A[i][j] ← A[i][j] − vgetq(factor4, 0) × A[k][j]
11:       j ← j − 1
12:   end while
13:   factor ← vmovq(vgetq(factor4, 0))
14:   for j = k + 1 to n − 4 step 4 do
15:       cal ← vld1q(A[k] + j)
16:       sum4 ← vmovq(cal, factor)
17:       cal ← vld1q(A[i] + j)
18:       cal ← vsubq(cal, sum4)
19:       vst1q(A[i] + j, cal)
20:   end for
21:   b[i] ← b[i] − vgetq(factor4, 0) × b[k]
22:   ...                                   ▷ 按相同方式处理后面 3 行
23: end for
24: ...                                     ▷ 处理剩余元素（非 4 的倍数部分）
```

- NEON 优化 4：（对包含消元和回代的平凡算法）回代部分进行 NEON 向量化优化，即平凡算法 12 行，语法与上述优化方式类似，不再列出。
- NEON 优化 5：（对包含消元和回代的平凡算法）消元的内层循环和回代部分分别进行 NEON 向量化优化，即优化 2 和优化 4 的结合版本。

b) SSE 向量化优化

- SSE 优化 1：（对仅包含消元的平凡算法）内层（对应代码第 7 行）的循环优化。
- SSE 优化 2：（对包含消元和回代的平凡算法）对消元的内层循环进行优化（对应代码第 5 行）。

实现思路与 NEON 类似，但使用了支持 x86 平台的 SSE 语法。

c) AVX 向量化优化

AVX 优化：（对包含消元和回代的平凡算法）对消元的内层循环进行 AVX 优化（对应代码第 5 行）。实现方式与 SSE 优化 2 类似，但 AVX 作为 SSE 的升级版本，可以一次计算八个数据，效率更高。

4. Pthread 多线程优化

针对平凡算法，进行 Pthread 多线程优化，实现了以下 4 种优化方式。

- Pthread 优化 1：动态线程

为第二个内层循环（即更新其他行的操作，对应平凡算法的 6-10 行）创建动态线程，每次外层循环迭代时，根据当前的主元行来动态地创建一组线程。每个线程将当前主元行下方的某一行进行消元操作。

```
1: struct ThreadParam { int k, t_id, n; }
2: function ThreadFunc(param)
3:     k ← param.k
4:     t_id ← param.t_id
5:     n ← param.n
6:     i ← k + t_id + 1
7:     for j = k + 1 to n do
8:         A[i][j] −= A[i][k] ∗ A[k][j]
9:     end for
10:    A[i][k] ← 0
11:    exit thread
12: end function
13: function Active(n)
14:    for k = 0 to n do
15:        for j = k + 1 to n do
16:            A[k][j] /= A[k][k]
17:        end for
18:        A[k][k] ← 1.0
19:        handles ← create threads(n − k − 1)
20:        params ← create thread params(n − k − 1)
21:        for t_id = 0 to n - k - 1 do
22:            params[t_id] ← ThreadParam(k, t_id, n)
23:            create thread(ThreadFunc, params[t_id])
24:        end for
25:        join threads(handles)
26:    end for
27: end function
```

- Pthread 优化 2：信号量同步——为第二个内层循环创建静态线程，采用信号量同步的方式

  在程序开始时创建一组固定数量的线程，并在整个算法执行过程中重复使用这些线程。以信号量同步的方式保证在更新某一行之前，上面的所有行均已经完成消元。

```
1: struct ThreadParamA { int t_id, n; }
2: function ThreadFuncA(param)
3:     t_id ← param.t_id
4:     n ← param.n
5:     for k = 0 to n do
6:         wait sem_workerstart[t_id]
7:         for i = k + 1 + t_id to n step NUM_THREADS do
8:             for j = k + 1 to n do
9:                 A[i][j] −= A[i][k] ∗ A[k][j]
10:            end for
```

```
            A[i][k] ← 0
        end for
        signal sem_main
        wait sem_workerend[t_id]
    end for
    exit thread
end function
function STAT(n)
    init semaphores
    handles ← create threads(NUM_THREADS)
    params ← create thread params(NUM_THREADS)
    for t_id = 0 to NUM_THREADS do
        params[t_id] ← ThreadParamA(t_id, n)
        create thread(ThreadFuncA, params[t_id])
    end for
    for k = 0 to n do
        for j = k + 1 to n do
            A[k][j]/ = A[k][k]
        end for
        A[k][k] ← 1.0
        signal sem_workerstart
        wait sem_main
        signal sem_workerend
    end for
    join threads(handles)
    destroy semaphores
end function
```

- Pthread 优化 3：三重循环——在上一方法的基础上，将三层循环全部纳入线程函数

  主线程进行除法操作，其他线程在收到主线程除法完成的信号进行减法操作。在收到其他线程完成减法的信号后，主线程再进行下一轮除法操作，保证了程序的可靠性。

```
struct ThreadParamS3 { int t_id, n; }
function THREADFUNCS3(param)
    t_id = param.t_id
    n = param.n
    for k = 0 to n do
        if t_id == 0 then
            for j = k + 1 to n do
                A[k][j]/ = A[k][k]
            end for
            A[k][k] = 1.0
            signal sem_Divsion
        else
            wait sem_Divsion[t_id - 1]
        end if
```

```
        for i = k + 1 + t_id to n step NUM_THREADS do
            for j = k + 1 to n do
                A[i][j] -= A[i][k] * A[k][j]
            end for
            A[i][k] ← 0
        end for
        if t_id == 0 then
            wait sem_leader
            signal sem_Elimination
        else
            signal sem_leader
            wait sem_Elimination[t_id - 1]
        end if
    end for
    exit thread
end function
function STAT3(n)
    init semaphores
    handles ← create threads(NUM_THREADS)
    params ← create thread params(NUM_THREADS)
    for t_id = 0 to NUM_THREADS do
        params[t_id] ← ThreadParamS3(t_id, n)
        create thread(ThreadFuncS3, params[t_id])
    end for
    join threads(handles)
    destroy semaphores
end function
```

- Pthread 优化 4：barrier 同步——在上一方法的基础上，不再使用信号量同步，改为使用 barrier 同步

  建立两个同步点，分别在除法和减法完成后，保证程序的并发性和正确性。

```
struct ThreadParamB { int t_id, n; }
function THREADFUNCB(param)
    t_id = param.t_id
    n = param.n
    for k = 0 to n do
        if t_id == 0 then
            for j = k + 1 to n do
                A[k][j] /= A[k][k]
            end for
            A[k][k] = 1.0
        end if
        wait barrier_Divsion
        for i = k + 1 + t_id to n step NUM_THREADS do
            for j = k + 1 to n do
```

```
            A[i][j] -= A[i][k] * A[k][j]
        end for
        A[i][k] = 0
    end for
    wait barrier_Elimination
  end for
  exit thread
end function
function StatB(n)
  init barriers
  handles = create threads(NUM_THREADS)
  params = create thread params(NUM_THREADS)
  for t_id = 0 to NUM_THREADS do
    params[t_id] = ThreadParamB(t_id, n)
    create thread(ThreadFuncB, params[t_id])
  end for
  join threads(handles)
  destroy barriers
end function
```

5. OpenMP 多线程优化

- OpenMP 优化 1：在内层循环中以行来划分进程

行划分版本

```
void mp(int n){
    int i,j,k;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k)
    for (k = 0; k < n; ++k) {
        #pragma omp single
        {
            for (j = k + 1; j < n; ++j) A[k][j] /= A[k][k];
            A[k][k] = 1.0;
        }
        #pragma omp for
        for (i = k + 1; i < n; ++i) {
            for (j = k + 1; j < n; ++j) A[i][j] -= A[i][k] * A[k][j];
            A[i][k] = 0;
        }
    }
}
```

- OpenMP 优化 2：在内层循环中以列来划分进程

列划分版本

```
void mp2(int n){
    int i,j,k;
```

```
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k)
    for (k = 0; k < n; ++k) {
        #pragma omp single
        {
            for (j = k + 1; j < n; ++j) A[k][j] /= A[k][k];
            A[k][k] = 1.0;
        }
        #pragma omp for
        for (j = k + 1; j < n; ++j) {
            for (i = k + 1; i < n; ++i) A[i][j] -= A[i][k] * A[k][j];
        }
        for (i = k + 1; i < n; ++i)
            A[i][k] = 0;
    }
}
```

猜测：行划分性能或许会更好一些，因为这样可以增加行主存储下 cache 命中率。

6. MPI 多进程优化

- MPI 优化 1：一维行划分

  注：该部分除法 $\frac{n}{m}$ 均为整数除法，等效于 $\frac{n-n\%m}{m}$。

  在 MPI 一维行划分算法中，主要实现方式是把每一行分配给每一个进程，假设矩阵规模为 n*n，进程数量为 m(从第 0 个到第 $m-1$ 个)，则对于进程 $i(i!=m-1)$, 分配 $\frac{n}{m}i$ 至 $\frac{n}{m}(i+1)-1$ 行。

  对于进程 $m-1$，要将从 $\frac{n}{m}(m-1)$ 到最后的所有行分配，以保证所有行都被处理。

MPI 一维行划分算法 1

```
void mpi(int n) {
    int per = n / mpi_size;
    int r1 = mpi_rank * per;
    int r2 = (mpi_rank == mpi_size - 1) ? (n - 1) : ((mpi_rank+1) * per - 1);
    for (int k = 0; k < n; k++) {
        if (r1 <= k && k <= r2) {
            for (int j = k + 1; j < n; j++)
                A[k][j] /= A[k][k];
            A[k][k] = 1.0;
            for (int j = 0; j < mpi_size; j++) {
                if(j!=mpi_rank)
                MPI_Send(&A[k][0], n, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
            }
        } else {
            MPI_Status status;
            MPI_Recv(&A[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
        }
        for (int i = max(r1,k+1); i <= r2; i++) {
```

```
            for (int j = k + 1; j < n; j++)
                A[i][j] -= A[k][j] * A[i][k];
            A[i][k] = 0;
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

- MPI 优化 2：更均匀的一维行划分

  考虑一种负载更加均衡的方式，以余数的对应关系给各进程分配行，使得每个进程的负载更加均衡。

  对于进程 i，所有除以进程数量余数为 i 的行将分配给它。

MPI 一维行划分算法 2

```
void mpi_2(int n) {
    for (int k = 0; k < n; k++) {
        if (k%mpi_size==mpi_rank) {
            for (int j = k + 1; j < n; j++)
                A[k][j] /= A[k][k];
            A[k][k] = 1.0;
            for (int j = 0; j < mpi_size; j++) {
                if(j!=mpi_rank)
                MPI_Send(&A[k][0], n, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
            }
        } else {
            MPI_Status status;
            MPI_Recv(&A[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
        }
        for (int i = k+1; i <n; i++) {
            if(i%mpi_size==mpi_rank){
                for (int j = k + 1; j < n; j++)
                    A[i][j] -= A[k][j] * A[i][k];
                A[i][k] = 0;
            }
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

- MPI 优化 3：流水线划分优化

  对于一维行划分方式 1 进行流水线划分优化，每一个进行计算的进程至传递给后一个进程和 0 号进程（为保证输出结果正确），非进行计算且在进行计算之后的进程则从前一个进程处接受，并传递给后一个进程（如果不是最后一个进程）。

MPI 流水线优化算法

```
void mpi_pipe(int n) {
    //分配进程，同行划分1的2-4行
    for (int k = 0; k < n; k++) {
        if (r1 <= k && k <= r2) {
            //除法，同行划分1的6-9行
            if (mpi_rank != mpi_size - 1)
                MPI_Send(&A[k][0], n, MPI_FLOAT, mpi_rank + 1, 0,
                    MPI_COMM_WORLD);
            if (mpi_rank != 0)
                MPI_Send(&A[k][0], n, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
        } else if(k<r1){
            MPI_Status status;
            MPI_Recv(&A[k][0], n, MPI_FLOAT, mpi_rank - 1, 0, MPI_COMM_WORLD,
                &status);
            if (mpi_rank != mpi_size - 1) {
                MPI_Send(&A[k][0], n, MPI_FLOAT, mpi_rank + 1, 0,
                    MPI_COMM_WORLD);
            }
        }else if(!mpi_rank){
            MPI_Status status;
            MPI_Recv(&A[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
        }
        //减法，同行划分1的18-22行
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

- MPI 优化 4/5：非阻塞通信实现

  对于两种一维行划分方式的阻塞通信改为非阻塞通信，以降低等待时间，通过 wait 来保证算法正确性。对于优化 1 的非阻塞通信对应优化 4，对于优化 2 的非阻塞通信对应优化 5。

MPI 非阻塞通信

```
//send代码替换如下
MPI_Request request;
MPI_Isend(&A[k][0], n, MPI_FLOAT, j, 0, MPI_COMM_WORLD, &request);
//recv代码替换如下
MPI_Status status;
MPI_Request request;
MPI_Irecv(&A[k][0], n, MPI_FLOAT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &request
    );
MPI_Wait(&request, &status);
```

7. CUDA GPU 优化

利用 CUDA 的编程范式对高斯消元法进行改进，在 GPU 上进行高斯消去算法，并与 CPU 上进行的平凡算法实现进行对比。高斯消去的 CUDA 计算主体框架如下所示。

CUDA 高斯消去

```
//主函数
void cud(float* data_D, int n) {
    dim3 grid((n + BLOCK_SIZE - 1) / BLOCK_SIZE);
    dim3 block(BLOCK_SIZE);
    cudaError_t ret;
    for (int k = 0; k < n; k++) {
        division_kernel <<<grid, block>>>(data_D, k, n);
        cudaDeviceSynchronize();
        ret = cudaGetLastError();
        if (ret != cudaSuccess) {
            printf("division_kernel failed, %s\n", cudaGetErrorString(ret));
        }
        eliminate_kernel <<<grid, block>>>(data_D, k, n);
        cudaDeviceSynchronize();
        ret = cudaGetLastError();
        if (ret != cudaSuccess) {
            printf("eliminate_kernel failed, %s\n", cudaGetErrorString(ret));
        }
    }
}
//除法部分实现
__global__ void division_kernel(float* data, int k, int n) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n) {
        float element = data[k * n + k];
        float temp = data[k * n + tid];
        data[k * n + tid] = temp / element;
    }
}
//减法部分实现
__global__ void eliminate_kernel(float* data, int k, int n) {
    int tx = blockDim.x * blockIdx.x + threadIdx.x;
    if (tx == 0) data[k * n + k] = 1.0f;
    int row = k + 1 + blockIdx.x;
    while (row < n) {
        int tid = threadIdx.x;
        while (k + 1 + tid < n) {
            int col = k + 1 + tid;
            float temp_1 = data[row * n + col];
            float temp_2 = data[row * n + k];
            float temp_3 = data[k * n + col];
            data[row * n + col] = temp_1 - temp_2 * temp_3;
            tid = tid + blockDim.x;
        }
        __syncthreads();
        if (threadIdx.x == 0) data[row * n + k] = 0;
        row += gridDim.x;
```

```
48      }
49  }
```

（三） 性能测试与分析

1. SIMD:NEON 各向量化方式对比

本部分在 ARM 平台下进行，ARM 平台环境配置：华为鲲鹏服务器，G++ 编译器，O2 优化，-march=armv8-a。

- 平凡算法依托：包含消元和回代的平凡算法
- NEON 组：NEON 优化 2
- NEON01 组：NEON 优化 3
- NEON2 组：NEON 优化 4
- NEON02 组：NEON 优化 5

在测试过程中，保证每组计算至少经过了 5 次计算取得平均值，对于矩阵规模较小的实验组，适当的扩大计算次数以便取得较为精确的结果。测试结果如表1所示，各算法优化比如图3所示。

矩阵大小 n	16	32	64	128	256	500	750	1000	1500	2000
平凡	0.0015	0.0102	0.082	0.656	5.33	43.8	190.2	386.8	1320.2	3017.2
NEON	0.0010	0.0051	0.032	0.230	1.87	15.4	58.2	135.4	454.6	1026.2
NEON01	0.0017	0.0061	0.032	0.246	1.93	15.4	56	134.2	452.2	1025.6
NEON2	0.0015	0.0102	0.078	0.639	5.26	43.6	160.4	389.4	1345.8	3286.8
NEON02	0.0012	0.0051	0.029	0.213	1.86	15	55.2	133.4	448.4	1019.8

表 1: 各部分优化算法平均运行时间 (单位:ms)

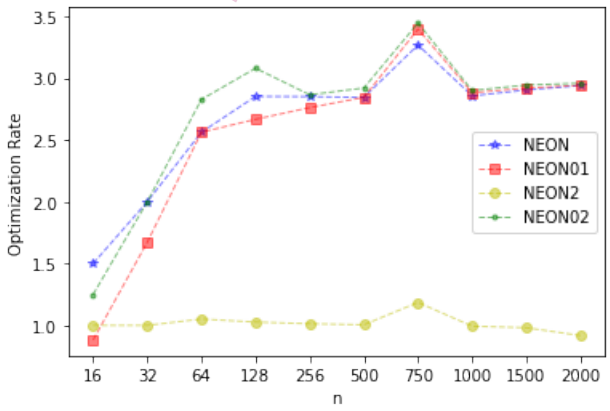


图 3: 各算法优化比

整体来说，NEON 各优化算法在数据规模较小的情况下优化效率不大，但随着数据规模的增加，其优化效率也逐渐凸显出来。

通过图3可以看出，NEON2 优化算法的优化比稳定在 1.0 左右，即使在 n 较大的时候也存在优化比小于 1 的情况。这种情况是由于优化的部分并不是核心的 n^3 循环，而调用 NEON 库的耗能过大导致的。然而这并不是说这部分不需要优化，对比 NEON02 和 NEON、NEON01 的优化比，可以看到 NEON02 的优化比能够稳定略高于另外两种算法。

NEON01 算法的优化比仅仅略高于 NEON 算法，不如 NEON02 算法。猜测这是因为外层循环要读取的数据不是连续存储的，使用向量优化效果并不显著。

2. SIMD:NEON/SSE/AVX 性能比较

x86 平台环境配置：个人物理机，x86-64 平台，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，G++ 编译器，C++ 17 GNU，O2 优化，-march=native。

ARM 平台环境配置：华为鲲鹏服务器，G++ 编译器，02 优化，-march=native。

在本部分中主要对包含消元和回代的算法进行讨论。

- 平凡算法依托：包含消元和回代的平凡算法，角标标识为运行环境
- NEON 组：NEON 优化 2
- SSE 组：SSE 优化 2
- AVX 组：AVX 优化

在测试过程中，保证每组计算至少经过了 5 次计算取得平均值，对于矩阵规模较小的实验组，适当的扩大计算次数以便取得较为精确的结果。测试结果如表18所示，各算法优化比如图4所示。

矩阵大小 n	16	32	64	128	256	500	750	1000	1500	2000
平凡 $_{x86}$	0.0044	0.0327	0.229	1.819	14.5	103.6	356.8	824	2845	6695
SSE	0.0041	0.0317	0.180	1.328	9.87	75	237.6	561.8	1951	4648
AVX	0.0049	0.0204	0.135	0.770	5.47	38.8	128.6	305.4	1079	2602
平凡 $_{ARM}$	0.0015	0.0102	0.081	0.639	5.20	51.6	184.6	411.2	1334.6	3062
NEON	0.0010	0.0051	0.032	0.230	1.93	18	65.2	145.4	499.2	1027

表 2: 各类型算法平均运行时间 (单位:ms)

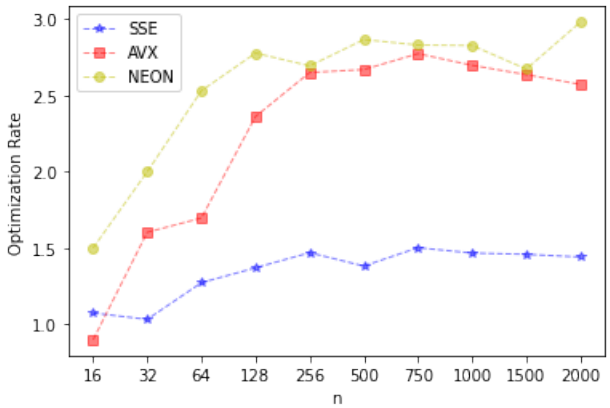


图 4: 各算法优化比

在 x86 环境 (环境说明同上节，-march=native)，将先前的十个测试点一起测试，检测 SSE、AVX 优化算法的效率，结果如表3所示。

	平凡算法	SSE 优化算法	AVX 优化算法
CPU Time/s	11.495	6.304	5.160
Instructions Retired	1.52e11	3.36e10	1.78e10
CPI rate	0.303	0.728	1.083

表 3: Vtune 分析结果

此外，通过对1、2两节的对比分析，我们可以发现在 ARM 环境下，-march=native 和-march=armv8-a 性能差距不大，仅在矩阵大小较大的情况下-march=armv8-a 性能略好。

通过图4可以看出，和上一部分我们的发现相同，在 n 较小的时候，SIMD 各优化算法优势并不明显，甚至在部分情况下出现优化比小于 1（即优化算法运行时间比平凡算法更长的情况)。但随着 n 的增大，大多优化算法均有着明显的优化表现。

此外，我们注意到 SSE 的 4 路并行算法在 n 较大的时候，优化比稳定在约 1.3-1.4 之间，优化效果比较一般，AVX 的 8 路并行算法在 n 较大的时候，优化比稳定在约 2.5-2.7 之间，优化效果尚可，但考虑到其消耗并不如 NEON 并行算法。NEON 的 4 路并行算法在 n 较大的时候取得了接近 3.0 的优化比，优化效果非常可观。当然，这部分差异也有可能是处理器能效的差别造成的。

通过 Vtune 对 SSE/AVX 的分析，我们可以看出 SSE 和 AVX 优化算法对于指令执行数量有非常大的优化，SSE 相比平凡算法优化了 4 倍，AVX 相比 SSE 算法再优化了接近 2 倍。然而，这却导致了 CPI rate 的快速上升，这也与上一部分所得优化效率没有那么高相对应。

3. Pthread: 各优化方式在不同线程数量中的对比分析

在矩阵规模上，我们选择 16,32,64,128,256,500,750,1000,1500,2000 十个矩阵规模。考虑到在 n 规模较小的时候运行时间较短，并进行多次实验取平均值，保证每组实验至少为重复 5 次取平均值，在规模较小的时候适当增加重复次数以便统计精确性。

- 平凡算法依托仅包含消元的平凡算法，下述各组角标标识为线程数。
- active 组：Pthread 优化 1——动态线程
- stat 组：Pthread 优化 2——信号量同步
- stat3 组：Pthread 优化 3——三重循环
- ststb 组：Pthread 优化 4——barrier 同步

a) x86 平台

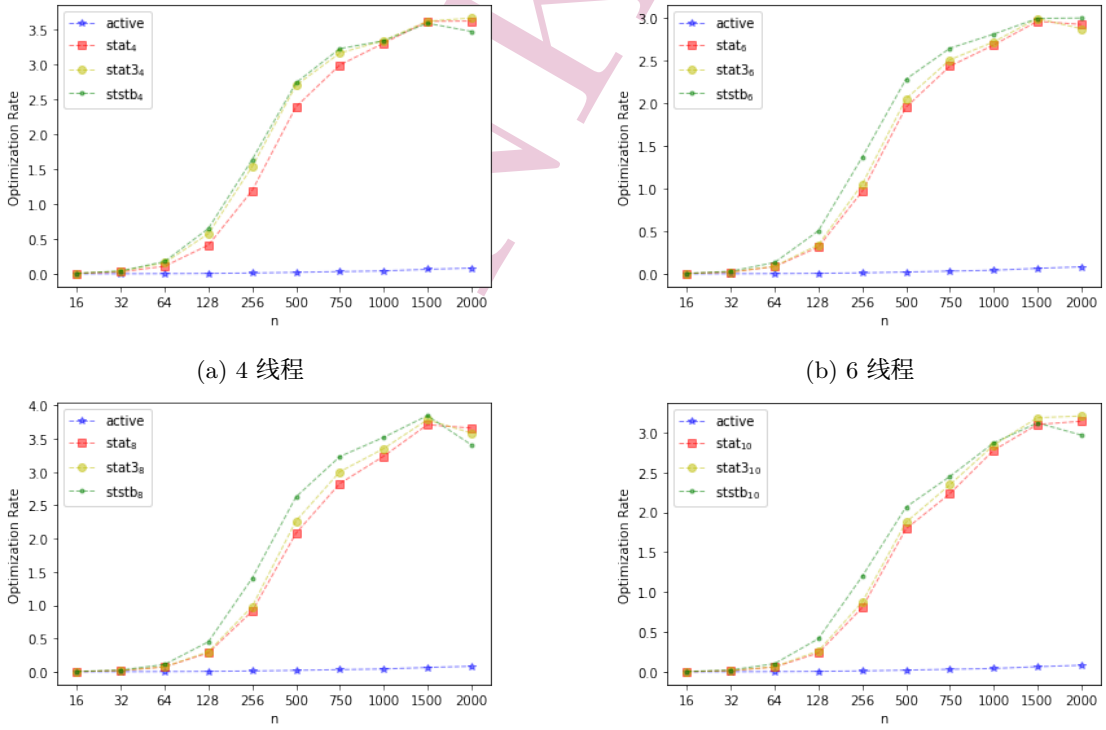
考虑到本人物理机与华为鲲鹏服务器都是 8 核的架构，首先我们以 4 线程，6 线程，8 线程，10 线程在本人物理机中进行测试，结果如表4所示。(其中，由于线程数量对平凡算法和动态线程算法不造成影响，且需要消耗极长的运行时间，这两个实验组仅测试一次)

x86 平台环境配置：个人物理机，x86-64，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，G++ 编译器，C++ 17 GNU。

与平凡算法相比较，得到加速比如图5所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.004	0.035	0.266	2.115	16.67	121.8	410.2	971.2	3388	7947
active	6.375	24.67	96.43	375.9	1490	5701	12826	23235	52868	96561
$stat_4$	0.766	1.294	2.455	5.148	14.00	51.00	137.4	294.6	938.6	2199
$stat3_4$	0.597	0.936	1.693	3.689	10.87	45.20	130.0	291.2	938.2	2173
$ststb_4$	0.558	0.855	1.5164	3.262	10.20	44.60	127.4	291.6	945.6	2293
$stat_6$	1.055	1.708	3.189	6.738	17.27	62.40	168.6	362.2	1145	2719
$stat3_6$	0.993	1.639	2.992	6.213	15.87	59.40	163.6	357.4	1132	2767
$ststb_6$	0.769	1.162	2.004	4.197	12.20	53.40	155.2	345.8	1132	2651
$stat_8$	1.263	2.046	3.693	7.475	18.33	58.60	145.2	300.2	913.4	2172
$stat3_8$	1.204	1.945	3.541	7.049	17.00	53.80	136.6	290.0	895.2	2218
$ststb_8$	0.941	1.434	2.426	4.705	11.87	46.40	127.0	276.0	881.2	2332
$stat_{10}$	1.552	2.496	4.471	8.967	20.73	68.00	183.8	349.8	1093	2531
$stat3_{10}$	1.479	2.381	4.168	8.082	19.07	65.00	175.0	342.0	1064	2479
$ststb_{10}$	1.107	1.578	2.602	5.098	13.93	59.00	167.6	338.6	1086	2678

表 4: x86 平台测试结果 (单位:ms)



(a) 4 线程

(b) 6 线程

(c) 8 线程

(d) 10 线程

图 5: x86 平台加速比

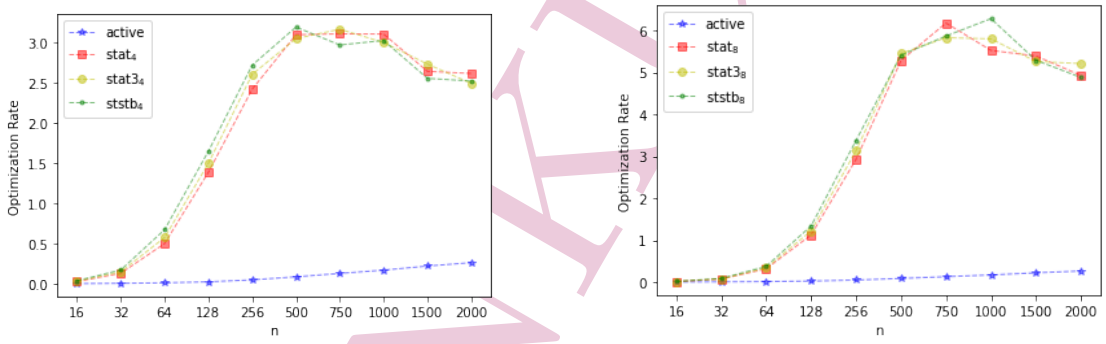
b) ARM 平台

ARM 平台环境配置：华为鲲鹏服务器，G++ 编译器。

对 ARM 平台进行测试，分别选择 4 线程和 8 线程进行，结果如表5所示。与平凡算法相比较，得到加速比如图6所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.010	0.081	0.652	5.164	41.00	311.8	1055	2538	8556	20000
active	2.718	11.33	49.20	202.2	826.2	3548	8109	14829	38553	75624
$stat_4$	0.352	0.615	1.299	3.721	16.93	100.6	339.2	816.6	3236	7641
$stat3_4$	0.297	0.538	1.131	3.443	15.73	102.2	333.0	845.2	3129	8018
$ststb_4$	0.257	0.468	0.967	3.131	15.07	97.60	355.0	838.2	3346	7926
$stat_8$	0.674	1.128	2.053	4.590	14.00	59.00	170.8	459.4	1587	4058
$stat3_8$	0.608	1.040	1.934	4.229	13.07	57.20	181.0	437.6	1627	3838
$ststb_8$	0.578	0.931	1.738	3.852	12.13	57.60	179.8	404.0	1618	4098

表 5: ARM 平台测试结果 (单位:ms)



(a) 4 线程 (b) 8 线程

图 6: ARM 平台加速比

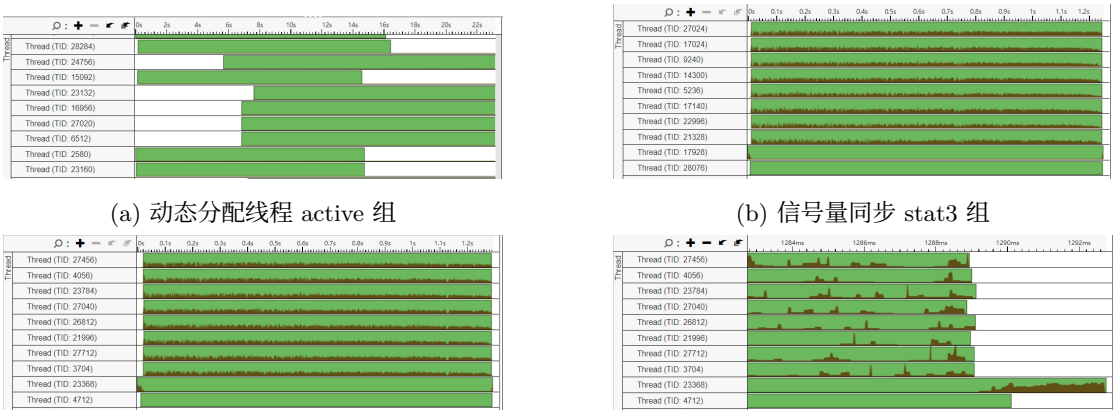
c) profiling

将各优化算法使用 Vtune 进行 profiling，结果如图7所示，其中 active 组创建线程数量极大，仅进行部分展示，其余进行全部展示。stat 组与 stat3 组结果没有明显差别，选择一个进行展示。最后一张子图展示 barrier 同步 statb 组的结尾细节，可以看到每个线程虽然总体结束时间相近，但仍有先后之分。

d) 总结

该部分所有结果均通过 check 函数检查正确性。

从性能测试结果可以看出，无论是 x86 平台还是 ARM 平台，动态分配进程的表现都不尽如人意。其分配的开销远远大于消元进行的开销（即使是在矩阵规模达到 2000 依然如此）。三种静态分配内存的表现效果都较为理想，且随着矩阵规模增大效果越发明显。在矩阵规模较小的时候，因为线程分配的开销相较于运算开销来说比较显著，所以优化比小于 1。在矩阵规模较大



(a) 动态分配线程 active 组

(b) 信号量同步 stat3 组

(c) barrier 同步 statb 组

(d) barrier 结尾细节展示

图 7: Vtune 分析结果

后，加速比逐渐趋于稳定。甚至在矩阵规模过大的情况下，或许受到访存开销的影响和平台波动影响，x86 平台和 ARM 平台甚至有了一点加速比下降的势头。

从线程角度对比来看，x86 平台上，8 线程表现最好，略好于 4 线程，而 6 线程和 10 线程相较来说效果不够理想。猜测是对于 8 核处理器来说 4 线程和 8 线程更便于分配。但 8 线程并没有高出 4 线程太多，猜测是设备 CPU 资源有限，也无法将全部电脑计算资源用于该程序的运行导致。

然而，ARM 平台上，8 线程的加速比远远高于 4 线程，最多时可以达到 6 倍以上的加速比。体现了华为鲲鹏服务器的性能之高，以及 pthread 并行化在 ARM 平台上的效果较为理想。

通过 profiling 结果（见图7）可以看出，静态分配线程各组线程的持续时间几乎从头到尾。从 d 图可以看到各被分配的子线程虽然结束时间相近，但也是存在不确定的先后顺序，这也与我们所期望的结果相同。动态分配的线程数量众多，持续时间也各不相同，这也体现了为什么动态分配线程造成如此大的开销以至于程序性能反向优化。

4. OpenMP: 各优化方式在不同线程数量中的对比分析

测试将围绕 x86 平台和 ARM 平台两个平台进行。在矩阵规模上，我们选择 16,32,64,128,256,500,750,1000,1500,2000 十个矩阵规模。考虑到在 n 规模较小的时候运行时间较短，并进行多次实验取平均值，保证每组实验至少为重复 5 次取平均值，在规模较小的时候适当增加重复次数以便统计精确性。在线程数量上，分别采用 4 线程和 8 线程进行测试。

- 平凡算法依托仅包含消元的平凡算法，下述各组角标标识为线程数。
- mp 组：OpenMP 优化 1——行划分
- mp2 组：OpenMP 优化 2——列划分

a) x86 平台

x86 平台环境配置：个人物理机，x86-64，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，G++ 编译器，C++ 17 GNU。

分别在 4 线程和 8 线程下测试 OpenMP 优化情况，测试结果如表6所示。将各进程下优化算法的运行时间与平凡算法比较，各优化方式优化比如图8所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.004	0.035	0.266	2.115	16.67	121.8	410.2	971.2	3388	7947
mp_4	1.320	2.586	5.107	9.869	21.60	63.00	192.0	319.8	1101	2460
$mp2_4$	1.318	2.609	5.049	10.44	26.13	93.60	313.6	703.6	3085	8228
mp_8	2.566	5.036	10.05	19.74	37.20	87.60	179.0	346.8	988.8	2180
$mp2_8$	2.569	5.042	9.988	19.03	38.47	103.0	280.8	714.2	2652	6529

表 6: x86 平台测试结果 (单位:ms)

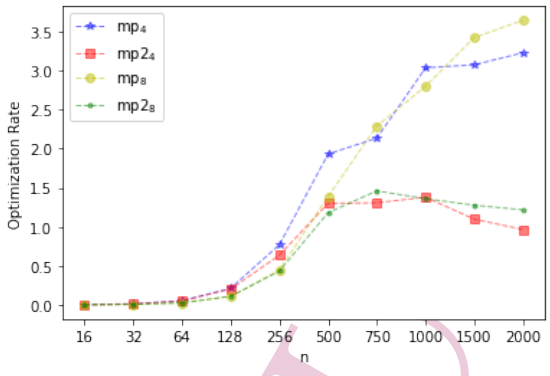


图 8: x86 平台优化比

b) ARM 平台

ARM 平台环境配置：华为鲲鹏服务器，G++ 编译器。

分别在 4 线程和 8 线程下测试 OpenMP 优化情况，测试结果如表7所示。将各进程下优化算法的运行时间与平凡算法比较，各优化方式优化比如图9所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.010	0.081	0.652	5.164	41.00	311.8	1055	2538	8556	20000
mp_4	0.025	0.066	0.418	2.000	11.80	80.40	267.8	636.4	2162	5112
$mp2_4$	0.035	0.191	2.135	11.72	48.33	149.0	358.0	910.4	2923	7055
mp_8	0.052	0.248	0.496	1.672	7.000	42.6	139.2	322.4	1077	2525
$mp2_8$	0.090	0.342	1.533	8.984	44.80	133.6	334.6	631.4	1825	4386

表 7: ARM 平台测试结果 (单位:ms)

c) 总结

从线程分配方式来看，在 x86 平台上，在矩阵规模较小的时候按行分配和按列分配的优化比差距不大，但随着矩阵规模的增大，按行分配和按列分配则有着显著的优化差异。甚至在矩阵规模达到 1500 以上的时候，按列分配甚至造成了对程序的负优化。（因为我们的串行算法也是按行访问的）

在 ARM 平台上，虽然按列分配在矩阵规模较大的时候优化仍然很显著，但是与相同线程的按行分配方案相比，优化比仅仅只有一半多。

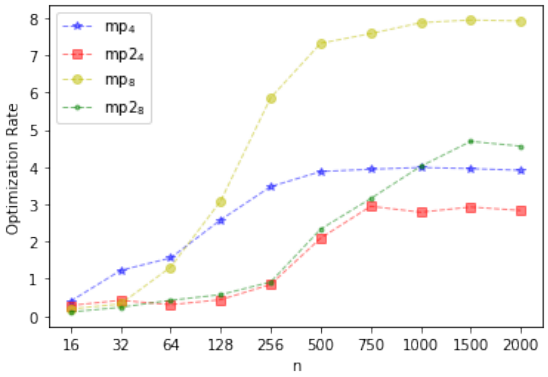


图 9: ARM 平台优化比

这是因为数组存储是行主存储的，访问同一列的元素的开销远远大于访问同一行元素的开销，因此列主访问的并行算法确实存在差于行主存储的串行算法的可能。随着矩阵规模的增大，cache 中单次存储的行数也在变少，从而造成了负优化。

从总体趋势来看，在矩阵规模较小的时候，由于分配线程的开销远高于计算的开销，所以优化比小于 1。随着矩阵规模的增加，优化比逐渐提升。最终，在矩阵规模足够大的时候趋于稳定.

从平台对比来看，x86 平台的 8 线程仍然是稍稍高于 4 线程，在矩阵规模较大后，行主优化比均在 3-4 之间，列主优化比趋于 1 左右（这说明优化结果已经很差了）。

ARM 平台整体优化比较高且增长迅速，行主 4 线程优化比接近 4，8 线程优化比接近 8。这是一个非常大的优化幅度。即使是较为笨拙的列主方法，也获得了 4 线程优化比约为 2.5，8 线程优化比超过 4 的优异成绩。

与 `Pthread` 编程对比　在 x86 平台上，比较两平台上表现最为优异的算法，无论是 4 线程还是 8 线程，pthread 编程和 OpenMP 编程的性能差异不大，均为缓慢增加，然后最后到达 3.5-4.0 的顶峰。在 ARM 平台上,OpenMP 编程的结果相较于 pthread 编程有着明显的优势，pthread 编程的优化比往往只能达到 3（4 线程），6（8 线程），而 OpenMP 编程的优化比则可以达到接近线程数量的完美数据，这正是我们所期望的结果。从编程难度上，OpenMP 的编程代码量和难度远远小于 pthread 编程，且不需要对原代码进行任何修改，更便于开发和维护。

5. Pthread/OpenMP with SIMD: 综合优化对比

在 8 线程下，分别对 NEON 向量化优化（在 ARM 平台）和 SSE 向量化优化（在 x86 平台）与 Pthread/OpenMP 的结合版本进行测试，选取梯度上升的十个数据规模，并保证每组测试数据进行了至少 5 次循环，且较小规模的数据进行更多次数循环，并与平凡算法和未进行向量化优化算法进行对比。

- 平凡算法：仅包含消元的平凡算法
- stat3 组：Pthread 优化 3——三重循环
- stat3sse 组（仅 x86）：Pthread 优化 3+SSE 优化 1
- stat3NEON 组（仅 ARM）：Pthread 优化 3+NEON 优化 1
- mp 组：OpenMP 优化 1——行划分
- mpsse 组（仅 x86）：OpenMP 优化 1+SSE 优化 1
- mpNEON 组（仅 ARM）：OpenMP 优化 1+NEON 优化 1

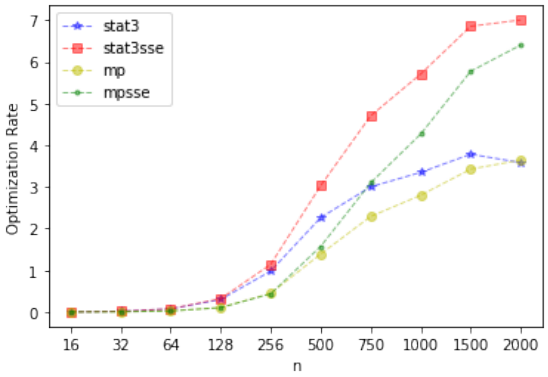


图 10: x86 平台优化比

a) x86 平台

x86 平台环境配置：个人物理机，x86-64，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，G++ 编译器，C++ 17 GNU。

在 x86 平台下对 8 线程进行测试，测试结果如表8所示。将优化算法性能与平凡算法比较，优化比如图10所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.004	0.035	0.266	2.115	16.67	121.8	410.2	971.2	3388	7947
stat3	1.204	1.945	3.541	7.049	17.00	53.80	136.6	290.0	895.2	2218
stat3sse	1.213	1.955	3.459	6.607	14.60	40.20	87.20	170.2	494.4	1135
mp	2.566	5.036	10.05	19.74	37.20	87.60	179.0	346.8	988.8	2180
mpsse	2.555	5.057	10.07	20.12	38.20	77.40	132.4	227.6	586.8	1241

表 8: x86 平台测试结果 (单位:ms)

b) ARM 平台

ARM 平台环境配置：华为鲲鹏服务器，G++ 编译器。

在 ARM 平台上对 8 线程进行测试，测试结果如表9所示。将优化算法性能与平凡算法比较，优化比如图11所示。

size	16	32	64	128	256	500	750	1000	1500	2000
common	0.010	0.081	0.652	5.164	41.00	311.8	1055	2538	8556	20000
stat3	0.608	1.040	1.934	4.229	13.07	57.20	181.0	437.6	1627	3838
stat3NEON	0.613	1.040	1.934	4.016	10.93	43.00	117.0	285.2	869.0	2221
mp	0.052	0.248	0.496	1.672	7.000	42.6	139.2	322.4	1077	2525
mpNEON	0.0996	0.220	0.475	1.262	5.067	28.60	89.40	207.8	735.4	1941

表 9: ARM 平台测试结果 (单位:ms)

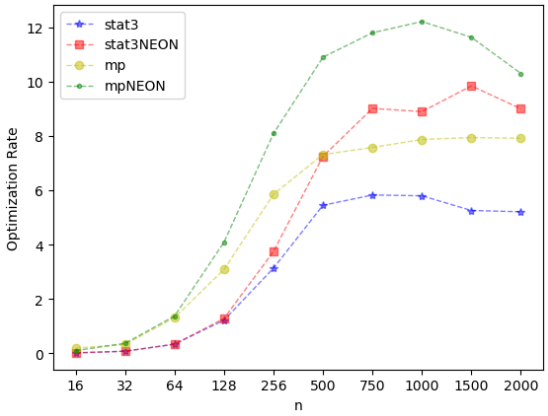


图 11: ARM 平台优化比

c) 总结

在本部分实验中，我们可以看到在 x86 平台上，SSE 优化算法较平凡算法获得了 6-7 的优化比，相较于未进行 SSE 优化的 Pthread/OpenMP 算法也有两倍多的优化，效果与单独进行 SSE 优化类似，相当可观。

在 ARM 平台上，NEON+stat3 优化算法相较于平凡算法获得了 9 倍左右的优化比，NEON+mp 获得了 10 倍以上的优化比，甚至在最高点接近 12 倍优化比。这一数值是相当可观的。不过若是将 NEON 优化后和未进行 NEON 优化之前的对比，NEON+stat3 相较于 stat3 大约有 2 倍优化，NEON+mp 相较于 mp 优化不足 1.5 倍。这其实远低于仅仅使用 NEON 可以获得的优化比。这可能是因为程序优化已经较为完善所导致，当然，这样的优化比其实仍然是非常可观的。

6. MPI: 各优化方式在不同进程数中的对比分析

在这一部分中，分别在 x86 平台下和 ARM 平台下，开辟 4 进程和 8 进程两种方式，对 MPI 优化程序进行性能测试，并于对应进程下的平凡算法进行比较。

- 平凡算法依托仅包含消元的平凡算法，下述各组角标标识为 MPI 进程数。
- mpi 组：MPI 优化 1——一维行划分
- mpi_2 组：MPI 优化 2——更均匀的一维行划分
- mpi_pipe 组：MPI 优化 3——流水线划分优化
- mpi_i 组：MPI 优化 4——1 的非阻塞通信版本
- mpi_i2 组：MPI 优化 5——2 的非阻塞通信版本

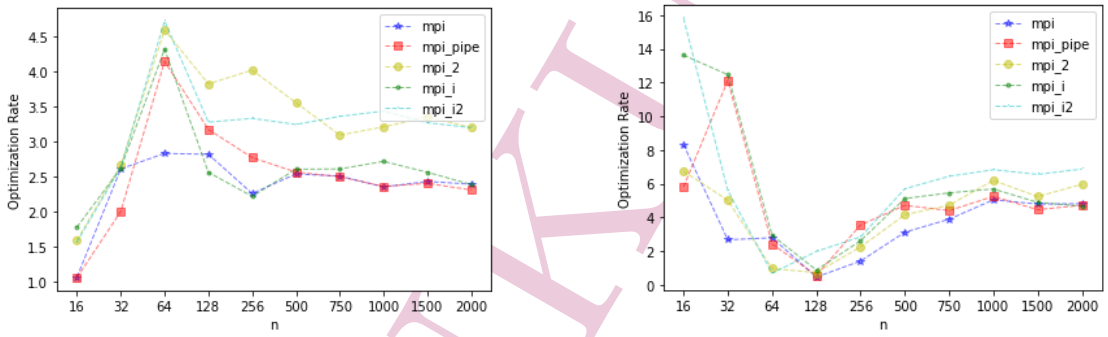
a) x86 平台

x86 平台环境配置：Ubuntu 22.04.4 LTS 虚拟机，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，mpic++ 编译器。

在 x86 平台下，对 4 进程和 8 进程的 MPI 优化程序进行性能测试，结果如表10所示。对比同进程下与平凡算法相比较，结果如图12所示，左图为 4 进程的对比结果，右图为 8 进程的对比结果。

	16	32	64	128	256	500	750	1000	1500	2000
$common_4$	0.0174	0.0912	0.510	5.05	25.1	175	585	1368	4475	10504
mpi_4	0.0163	0.0349	0.180	1.79	11.1	68.9	233	581	1839	4388
mpi_pipe_4	0.0164	0.0455	0.123	1.59	9.05	68.2	234	579	1860	4543
mpi_2_4	0.0109	0.0342	0.111	1.32	6.24	49.2	189	426	1334	3279
mpi_i_4	0.00977	0.0348	0.118	1.97	11.3	67.1	224	503	1745	4394
mpi_i2_4	0.0111	0.0346	0.108	1.54	7.53	53.9	174	398	1368	3283
$common_8$	2.262	1.297	0.584	5.91	35.1	260	869	2148	7185	17006
mpi_8	0.272	0.488	0.210	13.2	25.1	83.7	224	426	1502	3539
mpi_pipe_8	0.390	0.107	0.242	10.9	9.93	55.2	198	410	1615	3607
mpi_2_8	0.336	0.259	0.611	8.38	15.7	62.8	184	347	1374	2857
mpi_i_8	0.166	0.104	0.199	6.85	13.5	50.9	160	378	1472	3647
mpi_i2_8	0.143	0.232	0.863	3.01	12.4	45.7	135	315	1097	2479

表 10: x86 平台性能测试结果 (下标表示进程数量)(单位:ms)



(a) 4 进程 (b) 8 进程

图 12: 各进程 x86 平台加速比

b) ARM 平台

ARM 平台环境配置：华为鲲鹏服务器，mpic++ 编译器。

在 ARM 平台下，对 4 进程和 8 进程的 mpi 优化程序进行性能测试，结果如表11所示。对比同进程下与平凡算法相比较，结果如图13所示，左图为 4 进程的对比结果，右图为 8 进程的对比结果。

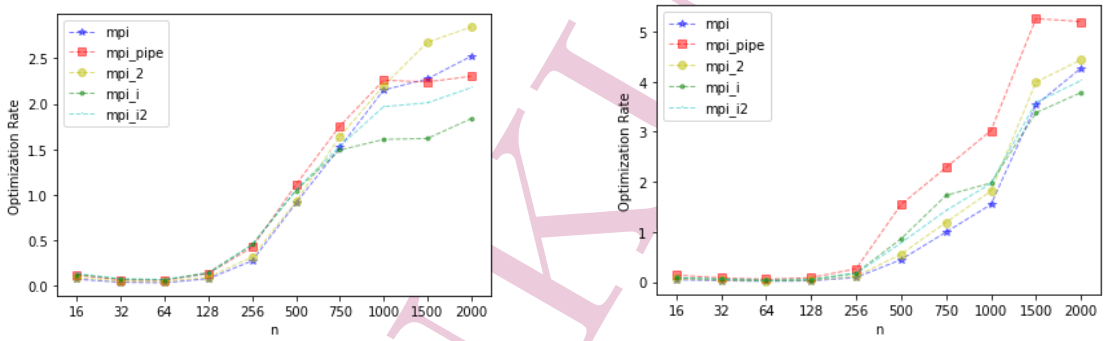
c) 总结

各 mpi 优化算法都展示出了较好的优化性能。8 进程的优化性能往往好于 4 进程。在 x86 平台下，流水线优化算法 (mpi_pipe) 并未能展现出相比于普通 mpi 算法的优势，第二种更为均衡的行划分方式的性能相比于第一种有着较为明显的性能提升，两种划分方式的非阻塞通信效率均高于阻塞通信。

这体现了更加均衡的划分方式有利于提升程序性能，非阻塞通信在不必要的阶段不进行等待也能够提升程序性能，流水线优化算法未能取得较好的成绩，可能是因为尤其是在第一种划分方式下，排队等待接收的开销相比于单一进程进行消元的开销较为显著，可能也于机器性能有关。

	16	32	64	128	256	500	750	1000	1500	2000
$common_4$	0.0978	0.103	0.179	0.777	5.73	56.4	199	482	1421	3431
mpi_4	1.36	2.72	5.41	9.98	20.8	62.2	131	224	625	1360
mpi_pipe_4	0.827	1.60	3.04	5.73	13.5	50.8	114	213	634	1490
mpi_2_4	1.07	2.11	3.96	8.30	18.1	60.9	122	219	531	1206
mpi_i_4	0.793	1.42	2.68	5.33	12.69	53.9	134	300	878	1867
mpi_i2_4	0.707	1.36	2.60	5.81	12.7	53.5	130	245	707	1575
$common_8$	0.146	0.258	0.293	0.887	5.85	60.3	206	483	2009	4184
mpi_8	3.07	9.40	17.3	35.1	64.7	135	207	312	568	980
mpi_pipe_8	1.06	3.15	5.19	10.1	22.2	39.0	90.3	160	382	804
mpi_2_8	2.38	7.36	13.1	27.4	52.6	109	173	266	503	943
mpi_i_8	1.71	4.47	8.24	17.5	32.7	69.9	119	244	594	1106
mpi_i2_8	1.62	4.64	8.72	18.8	35.1	77.3	145	245	560	1040

表 11: ARM 平台性能测试结果 (下标表示进程数量)(单位:ms)



(a) 4 进程 (b) 8 进程

图 13: 各进程 ARM 平台加速比

在 ARM 平台上，反而是阻塞通信组好于非阻塞通信组，这可能是由于 ARM 平台上由于数据传输足够快，对于非阻塞通信的判定开销反而高于阻塞的等待开销导致。pipe 组虽然在 4 进程上没有取得太好的成绩，但是在 8 进程上成绩非常显著，说明在进程较多的环境下，流水线优化对于程序性能的提升还是非常有必要的，

同时，我们会发现无论是在 x86 平台上还是 ARM 平台上，没有使用优化的平凡算法因为 barrier 的缘故造成了更多的运行时间，这绝对不是多次实验取最大值的结果。这在 x86 平台上尤为显著，这可能是因为个人设备的性能不足导致，在 ARM 平台上虽然问题依然存在，但没有特别显著，这得益于鲲鹏服务器优异的性能。

7. MPI with SIMD/Pthread/OpenMP: 综合优化对比

在各进程下，分别对 SIMD 向量化优化（x86 的 SSE 和 ARM 的 NEON），Pthread（8 线程），OpenMP（8 线程）与 MPI 的结合版本进行测试，选取梯度上升的十个数据规模，并保证每组测试数据进行了至少 5 次循环，且较小规模的数据进行更多次数循环，并与平凡算法和未进行向量化优化算法进行对比。

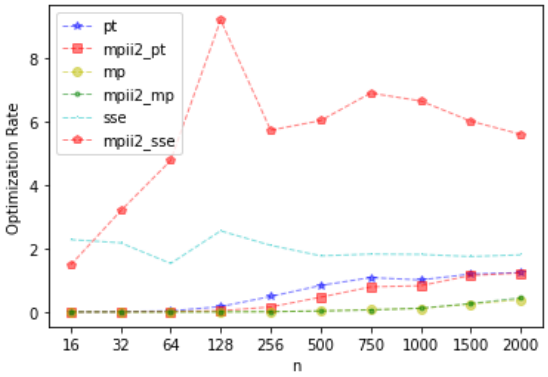


图 14: x86 平台优化比

- 平凡算法依托仅包含消元的平凡算法，下述各组角标标识为 MPI 进程数。
- pt 组：Pthread 优化 3——三重循环
- mpii2_pt 组：Pthread 优化 3+MPI 优化 5
- mp 组：OpenMP 优化 1——行划分
- mpii2_mp 组：OpenMP 优化 1+MPI 优化 5
- sse 组（仅 x86）：SSE 优化 1
- mpii2_sse 组（仅 x86）：SSE 优化 1+MPI 优化 5
- neon 组（仅 ARM）：NEON 优化 1
- mpii2_neon 组（仅 ARM）：NEON 优化 1+MPI 优化 5

a) x86 平台

x86 平台环境配置：Ubuntu 22.04.4 LTS 虚拟机，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，mpic++ 编译器。

在 x86 平台下，对 4 进程的 mpi 优化程序进行性能测试，其中 pthread 和 openmp 的线程数均为 8，结果如表12所示。将各优化算法与平凡算法相比较，得到的优化比如图14所示。

	16	32	64	128	256	500	750	1000	1500	2000
$common_4$	0.017	0.091	0.510	5.05	25.1	175	585	1368	4475	10504
pt_4	5.61	8.06	15.7	29.7	51.6	210	540	1357	3746	8471
$mpii2\_pt_4$	27.1	19.5	38.8	129	160	377	742	1654	3894	8554
mp_4	173	345	686	1384	2575	5100	8240	12303	17900	27198
$mpii2\_mp_4$	191	361	719	1471	2756	5519	8990	11564	16970	23697
sse_4	0.008	0.042	0.334	1.98	11.97	99.2	321	755	2565	5838
$mpii2\_sse_4$	0.012	0.029	0.107	0.550	4.39	29.1	84.9	206	745	1878

表 12: x86 平台性能测试结果 (4 进程)(单位:ms)

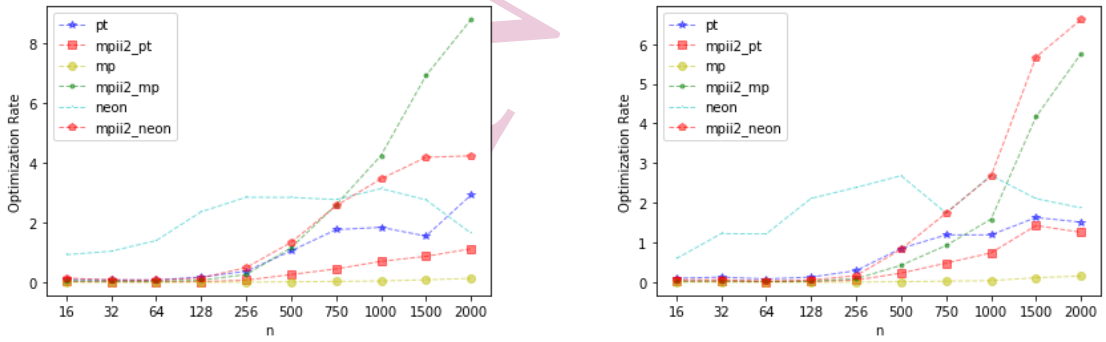
b) ARM 平台

ARM 平台环境配置：华为鲲鹏服务器，mpic++ 编译器。

在 ARM 平台下，对 4 进程和 8 进程的 mpi 优化程序进行性能测试，其中 pthread 和 openmp 的线程数均为 8，结果如表13所示。将各优化算法在同进程下与平凡算法相比较，结果如图15所示，左图为 4 进程的对比结果，右图为 8 进程的对比结果。

	16	32	64	128	256	500	750	1000	1500	2000
$common_4$	0.0978	0.103	0.179	0.777	5.73	56.4	199	482	1421	3431
pt_4	1.16	1.40	2.48	4.54	15.9	54.0	114	263	923	1179
$mpii2\_pt_4$	5.12	10.0	19.5	41.6	87.4	222	451	694	1648	3081
mp_4	0.641	0.450	0.804	1.35	4.48	12.7	29.3	59.9	188	444
$mpii2\_mp_4$	2.26	3.27	6.83	11.5	22.8	49.2	77.2	114	206	391
$neon_4$	0.107	0.099	0.129	0.331	2.02	19.9	72.3	154	516	2093
$mpii2\_neon_4$	0.718	1.36	2.69	5.53	11.8	42.5	78.2	140	341	813
$common_8$	0.146	0.258	0.293	0.887	5.85	60.3	206	483	2009	4184
pt_8	1.50	2.04	3.65	6.95	20.1	70.9	173	406	1233	2767
$mpii2\_pt_8$	6.05	14.4	27.1	55.7	114	265	430	655	1408	3329
mp_8	1.25	0.844	1.31	2.53	5.45	12.5	34.0	66.0	260	647
$mpii2\_mp_8$	5.34	8.42	17.2	34.7	68.0	142	223	305	482	727
$neon_8$	0.247	0.211	0.241	0.420	2.45	22.5	118	179	954	2234
$mpii2\_neon_8$	2.40	4.49	9.56	16.7	35.8	72.9	119	181	354	632

表 13: ARM 平台性能测试结果 (下标表示进程数量)(单位:ms)



(a) 4 进程 (b) 8 进程

图 15: 各进程 ARM 平台加速比

8. CUDA GPU: 不同 block size 与不同数据规模的对比

本部分在 x86 平台下进行，x86 平台环境配置：个人物理机，x86-64，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，NVIDIA GeForce GTX 1650，CUDA 12.4，Visual Studio 2022 编译环境。

针对不同的 block size 与不同的数据规模进行测试，每项测试至少重复 10 次 (观察到在次

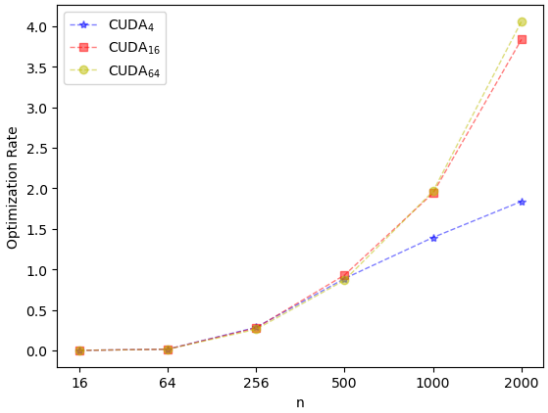


图 16: CUDA 各 block size 加速比

数较少的时候波动性极大，除了最后两组外至少重复 50 次），在数据规模较小的情况提高循环次数以增加精确度。

- 平凡算法：仅包含消元的平凡算法
- CUDA 组：CUDA GPU 优化，下标代表 block size

在进行下述测试之前，对 2000 规模的数据下，在 block size=4 和 block size=64 下各进行了 3 次测试，测试结果表明 block size 对平凡算法性能无明显影响。故以下用 block size=4 下的平凡算法测试结果代指全部平凡算法测试结果。

最终测试结果如表14所示。将优化算法与平凡算法对比，得到加速比如图16所示。

	16	64	256	500	1000	2000
common	0.004	0.253	15.52	112.5	903.6	7342
$CUDA_4$	2.073	15.23	54.31	126.7	648.0	3991
$CUDA_{16}$	2.153	15.34	55.67	120.8	465.3	1914
$CUDA_{64}$	2.215	16.20	59.12	129.6	460.3	1809

表 14: CUDA 性能测试结果 (单位:ms)

从图16中可以看出，block size=16 和 block size=64 的差距并不明显（这大概是受限于硬件环境），相对于 block size=4 有较大优化。在矩阵规模较小的时候，往往 CUDA 算法造成的是负优化，但随着矩阵规模的增大优化比不断增大。在 n=2000 的时候，GPU 优化比已经达到 4 左右，且还有较大的增长势头。受限于内存环境，难以进行更大规模的测试，但可以相信随着规模增大，至少在一定规模内还能够达到更好的性能优化。

二、 Gröbner 基计算高斯消去算法的并行优化

(一) 问题背景与重述

1. Gröbner 基的历史与核心思想

Gröbner 基的概念由 B. Buchberger 于 1965 年在他的博士论文《Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem nulldimensionalen Polynomideal》(计

算零维多项式理想的剩余类环的基的算法）中引入，论文的导师是著名的代数几何学家 W. Gröbner。Buchberger 在论文中设计了计算多元多项式理想的 Gröbner 基算法（称为 Buchberger 算法），并提出了优化该算法的若干准则（称为 Buchberger 第一、第二准则）。

Gröbner 基是由多元多项式理想的特殊生成元构成的集合，它具有非常良好的性质。通过计算 Gröbner 基很多有关多项式理想的基本问题都可以算法化求解。前面提到的理想成员判定问题也可以通过使用 Gröbner 基的性质获得解决：一个多元多项式属于给定生成元的多项式理想当且仅当它对该理想的 Gröbner 基的范式为 0。又譬如，字典序 Gröbner 基关于其中出现的变元具有一定的层级结构，即所谓的消元性质。基于这种性质，Gröbner 基方法又可以用来研究各种消元问题，如多项式方程组求解、参数曲线与曲面的隐式化等。

2. Gröbner 基在高斯消去中的应用

Gröbner 基计算方法相比于普通高斯消去方法进行了以下几点改进：

1. Gröbner 基中的运算均为有限域 GF(2) 上的运算，也就是矩阵元素的值只可能是 0 或 1。在该条件下，加法运算的实际执行方式为异或运算，即：0+0=0，0+1=1，1+0=1，1+1=0。同理由于异或运算的逆运算为自身，因此减法运算也是异或运算，即在高斯消去中实际上仅仅存在异或运算，将从一行中对另一行消去的运算退化为减法。

2. 矩阵行分为两类，“消元子”和“被消元行”。“消元子”在消去过程中充当减数，不会充当被减数。所有“消元子”的首个非零元素的位置都不同。“被消元行”在消元过程中充当被减数，但若其包含“消元子”中缺少的对角线 1 元素，也可升格成“消元子”。

该方法能够较好的降低算法的时间复杂度，在实际问题操作中的计算过程如下：

1. 分批次将” 消元子” 和” 被消元行” 读入内存。

2. 对每个读入的” 被消元行” 检查首项，如有对应” 消元子”，则将其与” 消元子” 进行异或操作，重复此操作直到无法进行，即发生 3/4 所述情况。

3. 若“被消元行”变成空行，将其丢弃。

4. 若“被消元行”首项无对应“消元子”，则将其升格为“消元子”。

5. 重复，直至所有批次处理完毕。

在实际执行该问题时，需要对数据存储和访问，计算进行较为合理的规划，同时也要结合输入数据格式进行优化调整。此外，需要以合理的方式对程序进行并行化优化。

3. 串行实现

该消元方式的串行实现主要分为三个步骤。

- 从文档中读取消元子和被消元行，并以合理的方式进行数据转储。
- 按照 Gröbner 思想处理每个被消元行。
- 将消元结果转编为原本的数据格式，存入消元结果文档中。

为节约内存空间，采取用 int 整型的每一位表示矩阵的每一列的形式。根据题目的 io 要求定义输入函数及输出函数如下。

4. 并行优化

将实际消元过程作为平凡算法进行优化，分别进行 SIMD/Pthread/OpenMP/MPI 优化。将优化后的程序与原来的平凡算法进行比较，分析性能。

（二） 算法设计与编程实现

1. 性能测量与 I/O 框架

a) 读函数：数据读取和存储

数据读取和存储框架如下所示。

定义常量: $xsize \leftarrow 130$, $xlen \leftarrow \lceil xsize/32 \rceil + 1$, $maxrow \leftarrow 10$
初始化矩阵: $row[maxrow][xlen]$, $xyz[xsize][xlen]$
初始化标记数组: $is[xsize]$
$nrow \leftarrow 0$
function READ
    打开文件" 消元子.txt" 用于读写
    while 从文件中读取一行 **do**
        初始化字符串流 iss
        读取行中第一个整数 $value$，设置 $nrow \leftarrow value$ 并标记 $is[value]$ 为真
        while 从 iss 中继续读取整数 $value$ **do**
            更新 $xyz[nrow][value/32]$ 的第 $value\%32$ 位为 1
        end while
    end while
    关闭文件
    打开文件" 被消元行.txt" 用于读写
    重置 $nrow \leftarrow 0$
    while 从文件中读取一行 **do**
        初始化字符串流 iss
        while 从 iss 中读取整数 $value$ **do**
            更新 $row[nrow][value/32]$ 的第 $value\%32$ 位为 1
        end while
        $nrow \leftarrow nrow + 1$
    end while
    关闭文件
end function

b) 写函数：消元结果的保存

消元结果的保存方式如下所示。

打开文件: 打开" 消元结果.txt" 用于追加
写入文件开头标记
for i 从 $xsize - 1$ 递减到 1 **do**
    if $is[i]$ **then**
        for j from $xlen - 1$ down to 0 **do**
            for u from 31 down to 0 **do**

```
7:                 if (xyz[i][j]) 的第 u 位为 1 then
8:                     写入文件 u + 32 × j 后跟一个空格
9:                 end if
10:             end for
11:         end for
12:         写入文件换行符
13:     end if
14: end for
15: 关闭文件
```

c) 计时与验证

计时函数构建思路与普通高斯消去类似，可以于上一部分查看。
通过对比预设的消元结果与实际输出的消元结果，可以得到验证。

2. 平凡算法

根据对特殊高斯消元法的分析，得到平凡算法如下，后续的并行优化均围绕这一部分展开。

平凡算法

```
1  void common(){
2      for(int i=0;i<nrow;i++){
3          bool isend=false;
4          for(int j=xlen-1;j>=0 && !isend;j--){
5              while(true){
6                  if(!row[i][j])break;int f;
7                  for(int u=31;u>=0;u--){
8                      if((row[i][j])&(1<<u)){f=u;break;}
9                  }
10                 f+=32*j;
11                 if(!is[f]){
12                     for(int k=0;k<xlen;k++) xyz[f][k]=row[i][k];
13                     is[f]=true;
14                     isend=true;
15                 }
16                 else
17                     for(int k=0;k<xlen;k++) row[i][k]^=xyz[f][k];
18             }
19         }
20     }
21 }
```

3. SIMD 向量化优化

a) SSE 优化

主要针对平凡算法的第 12 行和 17 行两个内层循环进行优化如下。

SSE 优化算法

```
//14-15行优化如下
int k=0;
for(;k<=xlen-4;k+=4) {
    __m128i temp = _mm_loadu_si128((__m128i *) (row[i] + k));
    _mm_storeu_si128((__m128i *) (xyz[f] + k), temp);
}
for(;k<xlen;k++)xyz[f][k]=row[i][k];
//20-21行优化如下
int k=0;
for(;k<=xlen-4;k+=4) {
    __m128i tx = _mm_loadu_si128((__m128i *) (xyz[f] + k));
    __m128i tr = _mm_loadu_si128((__m128i *) (row[i] + k));
    tr= _mm_xor_si128(tx, tr);
    _mm_storeu_si128((__m128i *) (row[i] + k), tr);
}
for(;k<xlen;k++)row[i][k]^=xyz[f][k];
```

b) NEON 优化

与 SSE 优化算法类似，针对平凡算法的第 12 行和 17 行两个内层循环进行优化如下。将 SSE 优化语法替换为 NEON 语法即可，不再列出。

4. Pthread 多线程优化

在这一部分，我将对内层函数实现 pthread 多线程优化，通过静态分配线程的方式，完成信号量的建立，线程之间通信以及最后线程的销毁。

Pthread 优化算法

```
sem_t sem_main; sem_t sem_workerstart[NUM_THREADS];
sem_t sem_workerend[NUM_THREADS];
struct threadParam_t { int t_id; int f; int* current_row;};
void* threadFunc(void* param) {
    threadParam_t* p = (threadParam_t*)param; int t_id = p->t_id;
    while (true) {
        sem_wait(&sem_workerstart[t_id]);
        if (p->f == -1) break;
        for (int k = t_id; k < xlen; k += NUM_THREADS) {
            if (!is[p->f]) xyz[p->f][k] = p->current_row[k];
            else p->current_row[k] ^= xyz[p->f][k];
        }
        sem_post(&sem_main);
        sem_wait(&sem_workerend[t_id]);
    }
    pthread_exit(NULL);
}
void stat() {
    sem_init(&sem_main, 0, 0);
```

```
    for (int i = 0; i < NUM_THREADS; ++i) {
        sem_init(&sem_workerstart[i], 0, 0);
        sem_init(&sem_workerend[i], 0, 0);
    }
    pthread_t handles[NUM_THREADS];
    threadParam_t params[NUM_THREADS];
    for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
        params[t_id].t_id = t_id; params[t_id].f = -1;
        params[t_id].current_row = nullptr;
        pthread_create(&handles[t_id], NULL, threadFunc, &params[t_id]);
    }
    for (int i = 0; i < nrow; ++i) {
        bool isend = false;
        for (int j = xlen - 1; j >= 0 && !isend; --j) {
            while (true) {
                if (!row[i][j]) break;
                int f = -1;
                for (int u = 31; u >= 0; --u)
                    { if ((row[i][j]) & (1 << u)) {f = u; break;} }
                f += 32 * j;
                if (!is[f]) {
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
                        params[t_id].f = f;
                        params[t_id].current_row = row[i];
                        sem_post(&sem_workerstart[t_id]);
                    }
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
                        sem_wait(&sem_main);
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
                        sem_post(&sem_workerend[t_id]);
                    is[f] = true; isend = true;
                } else {
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id) {
                        params[t_id].f = f;
                        params[t_id].current_row = row[i];
                        sem_post(&sem_workerstart[t_id]);
                    }
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
                        sem_wait(&sem_main);
                    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
                        sem_post(&sem_workerend[t_id]);
                }
            }
        }
    }
    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
        { params[t_id].f = -1; sem_post(&sem_workerstart[t_id]);}
    for (int t_id = 0; t_id < NUM_THREADS; ++t_id)
```

```
        pthread_join(handles[t_id], NULL);
    for (int i = 0; i < NUM_THREADS; ++i)
        {sem_destroy(&sem_workerstart[i]);sem_destroy(&sem_workerend[i]);}
    sem_destroy(&sem_main);
}
```

5. OpenMP 多线程优化

在这一部分中，由于串行算法的依赖性导致并行化优化没那么容易进行，我尝试了多种在外层循环开辟 openmp 的方法，但最终未能如愿实现，程序均以各种形式的失败告终。然而，我仍然准备了最简单的在内层开辟 openmp 线程的算法，在此部分体现，虽然它可能会造成程序的负优化，但仍然可以带来一些启示。

OpenMP 优化算法

```
void mp(){
    for(int i=0;i<nrow;i++){
        bool isend=false;
        for(int j=xlen-1;j>=0 && !isend;j--){
            while(true){
                if(!row[i][j])break; int f;
                for(int u=31;u>=0;u--){if((row[i][j])&(1<<u)){f=u;break;}}
                f+=32*j;
                if(!is[f]){
#pragma omp parallel for num_threads(NUM_THREADS)
                    for(int k=0;k<xlen;k++) xyz[f][k]=row[i][k];
                    is[f]=true;
                    isend=true;
                }
                else{
#pragma omp parallel for num_threads(NUM_THREADS)
                    for(int k=0;k<xlen;k++) row[i][k]^=xyz[f][k];
                }
            }
        }
    }
}
```

6. MPI 多进程优化

考虑这样一个方式，若可供分配 n 个进程，则将 n-1 个进程用于在他们各自的程序中消元他们被分配到的行，再将他们的消元结果发送给 0 号进程，0 号进程使用他们的消元结果进行进一步消元。这样的分配方式可以在保证消元结果正确性的前提下，最大限度的应用进程的负载，即使最后一部分工作仍然需要 0 号进程独立完成。

在这部分应用了 MPI 计时器重写的计时功能，通过 MPI_Wtime 获得精确的时间测量。以下是 mpi 的优化算法。

MPI 优化算法

```
1  void mpi() {
2      int rows_per_proc = nrow / (mpi_size - 1);
3      int start = (mpi_rank - 1) * rows_per_proc;
4      int end = (mpi_rank == mpi_size - 1) ? nrow : start + rows_per_proc;
5      if (mpi_rank != 0) {
6          for (int i = start; i < end; i++) {
7              bool isend = false;
8              for (int j = xlen - 1; j >= 0 && !isend; j--) {
9                  while (!isend) {//这里的判断条件进行了调整, 以使得消元结束后
                       立即停止, 平凡算法也进行同步调整
10                     if (!row[i][j]) break;
11                     int f = -1;
12                     for (int u = 31; u >= 0; u--) {
13                         if ((row[i][j]) & (1 << u)) {
14                             f = u + 32 * j;
15                             break;
16                         }
17                     }
18                     if (f == -1) break;
19                     if (!is[f]) {
20                         for (int k = 0; k < xlen; k++)
21                             xyz[f][k] = row[i][k];
22                         is[f] = true;
23                         isend = true;
24                     } else {
25                         for (int k = 0; k < xlen; k++)
26                             row[i][k] ^= xyz[f][k];
27                     }
28                 }
29             }
30             MPI_Send(row[i], xlen, MPI_INT, 0, 0, MPI_COMM_WORLD);
31         }
32     } else {
33         for (int i = 0; i < nrow; i++) {
34             MPI_Status status;
35             MPI_Recv(row[i], xlen, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                   MPI_COMM_WORLD, &status);
36             bool isend = false;
37             //消元, 同7-29行
38         }
39     }
40     MPI_Barrier(MPI_COMM_WORLD);
41 }
```

（三） 性能测试与分析

1. 样例数据集说明

在该部分性能测试中，选用不同数据规模的 7 个样例进行测试，详细数据规模如表15所示。

样例编号	1	2	3	4	5	6	7
矩阵列数	130	254	562	1011	2362	3799	8399
非零消元子数	22	106	170	539	1226	2759	6375
被消元行数	8	53	53	263	453	1953	4535

表 15: 样例数据集说明

2. SIMD:NEON/SSE 性能比较

在这一部分中，由于使用了 chrono 的 steady_clock，也为了测量优化算法对整体算法执行的影响，因此在这一小节统计的时间是一次消元完整的运行时间（即读、计算、写的全过程，与后面部分不同）。

对每组实验保证至少进行了 10 次重复实验，在数据规模较小的时候增加实验次数以保证数据有效性。

a) x86 平台：SSE 优化算法性能测试

x86 平台环境配置：个人物理机，x86-64 平台，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，G++ 编译器，C++ 17 GNU，O2 优化，-march=native。

样例编号	1	2	3	4	5	6	7
平凡 $_{x86}$	0.399	1.579	2.99	16.18	70.5	422.6	3976.7
SSE	0.386	1.502	2.77	15.58	64.55	343.7	2776.7

表 16: x86 平台算法平均运行时间 (单位:ms)

`profiling` 使用 Vtune 对 SSE 优化算法进行效率分析，结果如表17所示。（选用样例数据集 7，单次循环）

	平凡算法	SSE 优化算法
CPU Time/s	4.359	2.863
Instructions Retired	3.80e10	2.29e10
CPI rate	0.409	0.447

表 17: Vtune 分析结果

b) ARM 平台：NEON 优化算法性能测试

ARM 平台环境配置：华为鲲鹏服务器，G++ 编译器，O2 优化，-march=native。

样例编号	1	2	3	4	5	6	7
平凡 $_{ARM}$	0.1552	0.897	1.85	11.97	56.49	505.9	5188.4
NEON	0.1548	0.893	1.83	11.09	49.11	394.1	3453.2

表 18: ARM 平台算法平均运行时间 (单位:ms)

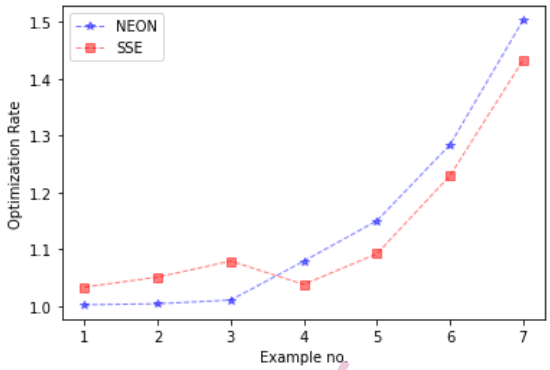


图 17: 各算法优化比

c) 总结

从图17中可以看出，随着矩阵列数和被消元行的增加，SSE 优化算法和 NEON 优化算法的优化比也在逐步增加，在矩阵有数千行下优化比可以达到接近 1.5。但在此情况下对于系统的空间和时间负载已经较大，故没有对更大的样例集进行试验。

随样例规模增大，实验时间快速增大，这与算法较大的时间复杂度有关。但各优化算法的优化比也在逐步增加。然而，优化比在数据规模较大的情况下，仍不够显著。这可能是因为在规模较大的时候，IO 的时间消耗也很大，而本实验在计算过程中计算了全流程的时间，却仅仅优化了中间的计算部分。同时，我们可以看到在各组实验中前期 x86 平台效率较低，后期 ARM 平台效率较低。这可能与当时电脑环境相关，也与鲲鹏服务器负载相关。然而，对于每一组实验的平凡算法运行时间和优化算法运行时间都是在一次执行中完成的，因此该优化比的计算仍然较为可靠。

根据表17可以看出，优化算法的 CPI rate 略高于平凡算法，但 Instructions Retired 完成的指令数量远低于平凡算法，从而得到更小的 CPU Time 结果。同时，减少的指令数并没有那么多也和先前猜测的 IO 开支对应。

3. Pthread/OpenMP with SIMD: 优化算法的协同对比

本部分在 x86 平台进行实验，x86 平台环境配置：个人物理机，x86-64 平台，Intel^R Core^{TM} i5-10200H CPU @ 2.40GHz × 8，windows 11 系统，G++ 编译器，C++ 17 GNU，O2 优化，-march=native。

各测试组说明如下。

- common 组：平凡算法
- sse 组：SSE 优化
- stat 组：Pthread 优化（静态分配线程信号量同步）
- mp 组：OpenMP 优化

- mpsse 组：OpenMP 优化 +SSE 优化

样例编号	1	2	3	4	5	6
common	<1	<1	<1	5.3	52.6	703.0
sse	<1	<1	<1	3.0	26.6	408.0
stat	3.1	91.5	83.0	2322.9	8995	100986
mp	3.0	108.2	104.3	2814.1	10730	122113
mpsse	3.0	109.0	104.0	2812.8	10747	122063

表 19: x86 平台测试结果（单位：ms）

测试结果如表19所示。我们可以看到我们设计的 pthread 优化算法和 OpenMp 优化算法实际上耗费了更多时间, 即使是使用了 SIMD 优化。对于 OpenMP，这个大概是由于动态分配线程的开销很大，而位运算的开销本来就很小。然而，pthread 编程也没有得到较为良好的结果。我认为是位运算计算速度本来就已经很快了，即使是在较大规模的循环下，信号量传输的速度也快于计算的速度。也有可能是我对并行划分还有优化空间，或者我的串行算法不适于并行划分。

同时，注意到 simd 组的优化比相较于上次测算的优化幅度会更大一些，这也说明内存读写与消元过程相比开销是不能够被忽略的，使用本次的计时方法，对于算法优化幅度的测算更为合理。

4. MPI: 优化算法在各进程下的对比分析

在各组实验中，测试 ARM 平台（因为该环境更加稳定，x86 平台下经测试波动较大）下，4 进程和 8 进程的平凡算法和 MPI 优化算法的表现。各组计算五次取得平均值，结果如表20所示。将优化算法与对应进程数量的平凡对比，得到优化比如图18所示。

样例编号	1	2	3	4	5	6	7
common_4	0.0525	0.105	0.114	2.45	15.7	377	4512
mpi_4	0.209	0.844	0.979	4.92	22.5	155	1622
common_8	0.0793	0.135	0.146	2.48	15.9	383	5124
mpi_8	0.263	1.06	1.096	5.93	18.3	140	1115

表 20: ARM 平台测试结果（单位：ms，下标标识进程数）

可以看出，MPI 优化算法在较大规模消元下，二次消元的优化效果非常显著。虽然在规模较小下并不显著，但在规模足够大的情况下，如第七组，在 4 进程中得到了将近 3 倍的加速比，在 8 进程中更是超过 4 倍。这得益于其更好的使用了更多的进程进行计算。

三、 总结与展望

在本学期的学习课程中，我逐步对 SIMD 编程，Pthread 编程，OpenMP 编程，MPI 编程，GPU 编程进行了学习，体会了各并行化优化的方向与思路。

在本次期末作业中，我将本学期所学内容应用到实际工作中，针对常见高斯消去算法以及 Gröbner 基计算高斯消去算法这个实际问题进行并行程序优化，将不同优化方法，不同进程/线程，不同编程方式，结合起来，探究他们的优势与不足。

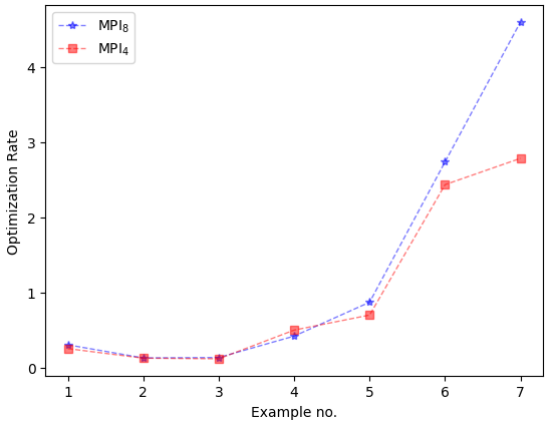


图 18: MPI 优化算法在各进程下优化比（下标标识进程数）

这次实验使我对这些并行架构有了进一步的了解，完成了将理论应用于实践的过程，很有收获。在之后的学习中，并行化的思想也将在实际工作中给我启迪，对并行编程结构的了解也将为我未来的工作助力。

相关代码，过程图片以及数据集文本文件等，可以从GitHub 仓库中查看。