
南開大學
Nankai University

# 南　开　大　学

## 计 算 机 学 院

并行程序设计第二次实验报告

---

# SIMD编程(以高斯消去法为例)

---

姓名：丁彦添

学号：1911406

年级：2019级

专业：计算机科学与技术

指导教师：王刚

2021 年 5 月 4 日

# 摘要

关键字：**SIMD** 并行化 高斯消去法 向量化

# 目录

# 一、 实验过程

## （一） 问题描述

实现高斯消去法（LU分解），对比实现SSE等算法加速计算过程。讨论不同算法策略对实验的影响。

高斯消去法，是线性代数中的一个算法，主要是将矩阵中各行各列进行线性变化，将方阵化为主对角线全为1的三角方阵，便于后续分解运算。高斯消去法可用来求解线性方程组，并可以求出矩阵的秩，以及求出可逆方阵的逆矩阵。本次实验实现高斯消去法并探讨对其进行并行化优化。具体的伪代码如下：

**Algorithm 1** 朴素的高斯消去算法

**Input:** 方阵A

**Output:** 高斯消元后上三角方阵A

```
1:  function GaussianElimination(matrix A)
2:      for k = 1; k < n; i++ do
3:          for j = k;j < n;j++ do
4:              A[k, j] ← A[k, j]/A[k, k]
5:          end for
6:          for i = k + 1;i < n;i++ do
7:              for j = k + 1;j < n;j++ do
8:                  A[i, j] ← A[i, j] − A[i, k] ∗ A[k, j]
9:              end for
10:             A[i, k] ← 0
11:         end for
12:     end for
13: end function
```

观察高斯消去法的伪代码，可以得知该算法最多嵌套了对于n的三重循环，高斯消去法的时间复杂度为 $O(n^3)$。

## （二） 算法设计与实现

由高斯消元法伪代码，很容易写出朴素的高斯消去法算法代码

朴素的高斯消去法算法代码

```
void ord(float **matrix) {
        for (int k = 0; k < N; k++) {
        //optimizable part one
                float tmp = matrix[k][k];
                matrix[k][k] = 1.0;
                for (int j = k + 1; j < N; j++) {
                        matrix[k][j] /= tmp;
                }
        //optimizable part two
                for (int i = k + 1; i < N; i++) {
                        float tmp = matrix[i][k];
                        for (int j = k + 1; j < N; j++) {
                                matrix[i][j] -= tmp * matrix[k][j];
                        }
                        matrix[i][k] = 0;
                }
        }
}
```

观察高斯消去算法，注意到代码第6行第一个内嵌循环中的matrix[k][j] = matrix[k][j]/tmp; 以及代码第10，11，12行双层for循环中的matrix[i][j] = matrix[i][j] - tmp * matrix[k][j]都是可以进行向量化的循环。可以通过SSE/AVX对这两步进行并行优化。值得注意的是，使用SSE/AVX向量并行化处理理论上优化的仅仅是常数时间复杂度，而且还会产生数据打包等等方面的开销。比如使用SSE四路并行计算，理论上并行优化部分加速比为4倍，实际上可能远远达不到。

下面将对比朴素算法、仅对除法做SSE并行化处理、仅对减法做SSE并行化处理、既对除法做SSE并行化处理又对减法做SSE并行化处理这四种进行测试，多次进行对结果求均值以减小误差。测量相关数据，算出相关优化效率，加速比等等。实验的数据规模N将从32到2048，多次测试。

- 对除法做并行化处理

  使用SSE四路并行，将单精度浮点整形数据以四个为一组进行打包计算除法，计算完成后再存入原矩阵相应的位置。将余下的部分单独串行化处理即可。

对除法做并行化处理部分

```
__m128 t1, t2, t3, t4;
for (int k = 0; k < N; k++) {
    //PART ONE
    float tmp[4] = {matrix[k][k], matrix[k][k], matrix[k][k], matrix[k][k]};
    t1 = _mm_loadu_ps(tmp);
    for (int j = N - 4; j >= k; j -= 4) {
        t2 = _mm_loadu_ps(matrix[k] + j);
        t3 = _mm_div_ps(t2, t1);
        _mm_storeu_ps(matrix[k] + j, t3);
```

```
    }
    if (k & 3 != (N & 3)) {
        for (int j = k; (j & 3) != (N & 3); j++) {
            matrix[k][j] /= tmp[0];
        }
    }
    for (int j = (N & 3) - 1; j >= 0; j--) {
        matrix[k][j] /= tmp[0];
    }
    //PART TWO...
}
```

- 对减法做并行化处理

  使用SSE四路并行，将单精度浮点数四个为一组进行打包，先计算乘法，再计算减法，计算完成后再存入原矩阵相应的位置。余下不能并行化的部分单独串行化处理即可。

对减法做并行化处理部分

```
//PART ONE...

//PART TWO
for (int i = k + 1; i < N; i++) {
    float tmp[4] = {matrix[i][k], matrix[i][k], matrix[i][k], matrix[i][k]};
    t1 = _mm_loadu_ps(tmp);
    for (int j = N - 4; j > k; j -= 4) {
        t2 = _mm_loadu_ps(matrix[i] + j);
        t3 = _mm_loadu_ps(matrix[k] + j);
        t4 = _mm_sub_ps(t2, _mm_mul_ps(t1, t3));
        _mm_storeu_ps(matrix[i] + j, t4);
    }

    for (int j = k + 1; (j & 3) != (N & 3); j++) {
        matrix[i][j] -= matrix[i][k] * matrix[k][j];
    }
    matrix[i][k] = 0;
}
```

在这一部分实验中，将任何编译器优化选项O2关闭进行实验。

为了探究向量长度对向量并行化的影响，将在下一步实验中使用AVX对朴素算法的可并行部分都使用向量并行化处理。本地编译选项O2关闭，增加-march=native为了编译AVX指令。和上一部分实验一样，使用多组不同的数据规模进行实验，测量相关数据，多次测试求平均值，以减小误差。实验结果分别与朴素算法和上一部分采用SSE优化算法进行比对。具体AVX代码如下：

AVX并行化处理

```
void AVX_optimize_both(float **matrix){
    __m256 t1,t2,t3,t4;
    for (int k = 0; k < N; k++) {
```

```
        //PART ONE
        float tmp[8] = {matrix[k][k], matrix[k][k], matrix[k][k], matrix[k][k],
        matrix[k][k], matrix[k][k], matrix[k][k], matrix[k][k]};
        t1 = _mm256_loadu_ps(tmp);
        for (int j = N - 8; j >= k; j -= 8) {
            t2 = _mm256_loadu_ps(matrix[k] + j);
            t3 = _mm256_div_ps(t2, t1);
            _mm256_storeu_ps(matrix[k] + j, t3);
        }

        if (k & 7 != (N & 7)) {
            for (int j = k; (j & 7) != (N & 7); j++) {
                matrix[k][j] /= tmp[0];
            }
        }

        for (int j = (N & 7) - 1; j >= 0; j--) {
            matrix[k][j] /= tmp[0];
        }
        //PART TWO
        for (int i = k + 1; i < N; i++) {
            float tmp[8] = {matrix[i][k], matrix[i][k], matrix[i][k], matrix[
            i][k],
            matrix[i][k], matrix[i][k], matrix[i][k], matrix[i][k]};
            t1 = _mm256_loadu_ps(tmp);
            for (int j = N - 8; j > k; j -= 8) {
                t2 = _mm256_loadu_ps(matrix[i] + j);
                t3 = _mm256_loadu_ps(matrix[k] + j);
                t4 = _mm256_sub_ps(t2, _mm256_mul_ps(t1, t3));
                _mm256_storeu_ps(matrix[i] + j, t4);
            }

            for (int j = k + 1; (j & 7) != (N & 7); j++) {
                matrix[i][j] -= matrix[i][k] * matrix[k][j];
            }
            matrix[i][k] = 0;
        }
    }
}
```

为了探究在并行化是对齐策略和非对齐策略对性能的影响，将本地编译选项O2关闭，和上一部分实验一样，使用多组不同的数据规模进行实验，测量相关数据，多次测试求平均值，以减小误差。使用SSE对齐策略并行化的代码如下：

采用对齐策略优化

```
void SSE_optimize_both_align(float **matrix){
        __m128 t1, t2, t3, t4;
```

```
        int mod;
        float temp;
        for(int k = 0; k < N; k ++){
        t1 = _mm_set1_ps(matrix[k][k]);
        mod = 4 - (k & 3);
        temp = matrix[k][k];
        for(int j = k; j < k + mod; j ++)
            matrix[k][j] /= temp;
        for(int j = k + mod; j < N; j += 4){
            t2 = _mm_load_ps(matrix[k] + j);
            t2 = _mm_div_ps(t2, t1);
            _mm_store_ps(matrix[k] + j, t2);
        }
        for(int i = k + 1; i < N; i++){
            t1 = _mm_set1_ps(matrix[i][k]);
            temp = matrix[i][k];
            for(int j = k + 1; j < k + mod; j ++){
                matrix[i][j] -= temp * matrix[k][j];
            }
            for(int j = k + mod; j < N; j += 4){
                                t2 = _mm_load_ps(matrix[i] + j);
                                t3 = _mm_load_ps(matrix[k] + j);
                                t4 = _mm_sub_ps(t2, _mm_mul_ps(t1, t3));
                                _mm_store_ps(matrix[i] + j, t4);
            }
            matrix[i][k] = 0;
        }
        }
        //output(matrix);
}
```

## （三） 本次实验测试平台及环境

- CPU:AMD Ryzen 5 3500H with Radeon Vega Mobile Gfx 2.10GHz
- 操作系统:Windows 10
- Codeblocks GCC 关闭默认O2优化

## （四） 实验及结果分析

对于第一个实验，得到以下实验结果：

| 数据规模\测试算法 | 无优化 | 仅优化除法 | 仅优化减法 | 同时优化 |
|---|---|---|---|---|
| N = 128 | 4.16 | 4.34 | 3.30 | 1.78 |
| N = 256 | 32.43 | 32.28 | 24.75 | 13.57 |
| N = 512 | 326.56 | 312.68 | 232.14 | 114.41 |
| N = 1024 | 2579.03 | 2418.89 | 1457.08 | 848.41 |
| N = 2048 | 21054.41 | 19413.72 | 12384.95 | 6971.42 |

表 1: 使用四路并行优化高斯消去法性能测试结果（部分）(单位:ms)

根据运行时间，可求出如下加速比：

| 数据规模\测试算法 | 优化除法:无优化 | 减法优化:无优化 | 同时优化:无优化 |
|---|---|---|---|
| N = 128 | 0.99 | 1.26 | 2.34 |
| N = 256 | 1.00 | 1.31 | 2.39 |
| N = 512 | 1.04 | 1.41 | 2.85 |
| N = 1024 | 1.07 | 1.77 | 3.04 |
| N = 2048 | 1.08 | 1.70 | 3.02 |

表 2: 使用四路并行优化高斯消去法性能加速比（部分）

根据以上结果，绘制加速比统计图如下：

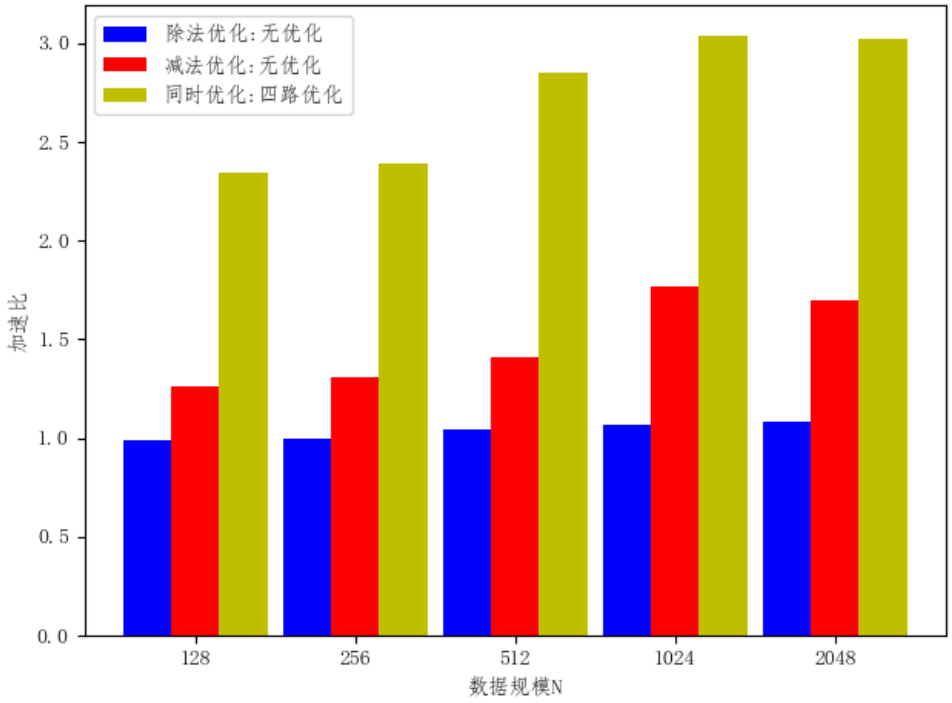


图 1: 使用SSE优化不同部分的加速比

结合算法运行时间和计算出的加速比可知：

1. 仅仅优化除法法部分，加速比比较小而且有时会小于1。结合时间复杂度，除法部分为$O(n^2)$，即使有优化也并不明显，容易被其它并行化开销抵消掉。

2. 仅仅优化减法（和乘法）部分，加速比在1.5左右但是并没有超过2.0。说明程序对减法部分进行并行化所得到的收益只比数据打包、存储等开销大，但结合到代码编写量的提升，性价比不高。减法部分的时间复杂度为$O(n^3)$，为耗时的主要部分，在数据规模较低时效果并不明显。

3. 同时优化除法部分和减法部分，加速比在数据规模较小的时候比较低，随着数据规模逐渐增大加速比可增加到3.0左右，说明用SSE对高斯消去法四路并行化的稳定性比较好。

对于第二个实验，得到以下实验结果：

| 数据规模\测试算法 | 无优化 | 四路优化 | 八路优化 |
|---|---|---|---|
| N = 128 | 3.99 | 2.10 | 1.85 |
| N = 256 | 30.96 | 9.67 | 14.49 |
| N = 512 | 233.54 | 101.11 | 61.78 |
| N = 1024 | 1973.78 | 774.03 | 480.24 |
| N = 2048 | 18121.13 | 6380.29 | 4185.20 |

表 3: 不同向量长度并行优化高斯消去法性能测试结果（部分）(单位:ms)

根据运行时间，可求出如下加速比：

| 数据规模\测试算法 | 四路:无优化 | 八路:无优化 | 八路:四路 |
|---|---|---|---|
| N = 128 | 1.96 | 2.16 | 1.10 |
| N = 256 | 2.00 | 3.20 | 1.60 |
| N = 512 | 2.31 | 3.78 | 1.63 |
| N = 1024 | 2.55 | 4.11 | 1.61 |
| N = 2048 | 2.84 | 4.33 | 1.52 |

表 4: 不同向量长度并行优化高斯消去法性能加速比（部分）

根据加速比，绘制统计图如下：

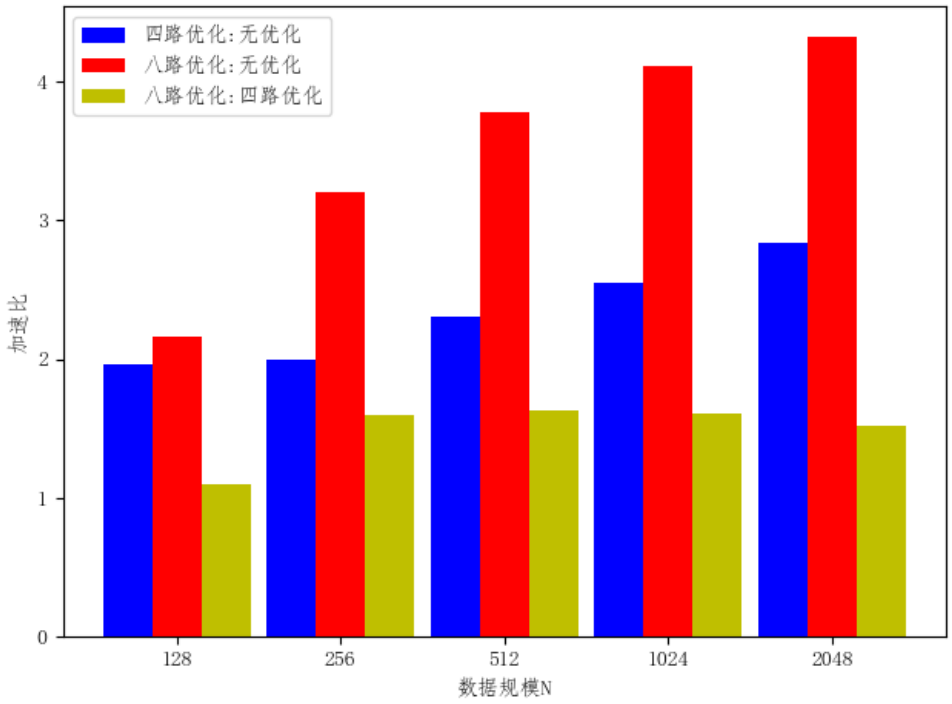


图 2: 不同向量长度并行优化高斯消去法性能加速比

结合第二部分实验的实验结果，可以发现八路并行化和四路并行化最终都对高斯消去法有加速作用其中四路并行化对朴素串行算法加速比保持在2.0到3.0之间，数据规模过低是甚至有可能低于2.0，八路并行化对朴素串行加速比在数据规模小的时候加速效果并不明显，随着数据规模的增大能增大到4.0以上。而八路并行化对四路并行的加速比在保持在1.6左右。

八路并行化在理论上和四路并行化的加速比上限为2.0，但是在本次实验中却加速比低于2.0。观察八路并行化的代码，可以发现八路并行化只有在向量计算时才能相对四路并行化加速，而对数据的打包产生的开销、单独串行化处理不能被打包的数据产生的代价却比四路并行化高。后者会抵消掉加速的效果。

对于第三个对比对齐策略和不对齐策略的实验，经过测试，得到以下实验结果：

| 数据规模\测试算法 | 无优化 | 对齐优化 | 不对齐优化 |
|---|---|---|---|
| N = 128 | 4.10 | 2.30 | 2.41 |
| N = 256 | 33.25 | 16.96 | 16.62 |
| N = 512 | 263.84 | 114.22 | 125.64 |
| N = 1024 | 2102.61 | 778.75 | 702.33 |
| N = 2048 | 19022.13 | 6383.45 | 5889.20 |

表 5: 数据对齐策略和数据不对齐策略性能测试结果（部分）(单位:ms)

根据测得的运行时间，可求出加速比如下： 根据加速比，绘制统计图如下：

| 数据规模\测试算法 | 对齐:无优化 | 不对齐:无优化 | 不对齐:对齐 |
|---|---|---|---|
| N = 128 | 1.78 | 1.70 | 0.96 |
| N = 256 | 1.96 | 2.00 | 1.02 |
| N = 512 | 2.31 | 2.10 | 0.91 |
| N = 1024 | 2.70 | 2.99 | 1.10 |
| N = 2048 | 2.98 | 3.23 | 1.08 |

表 6: 数据对齐策略和数据不对齐策略性能加速比（部分）

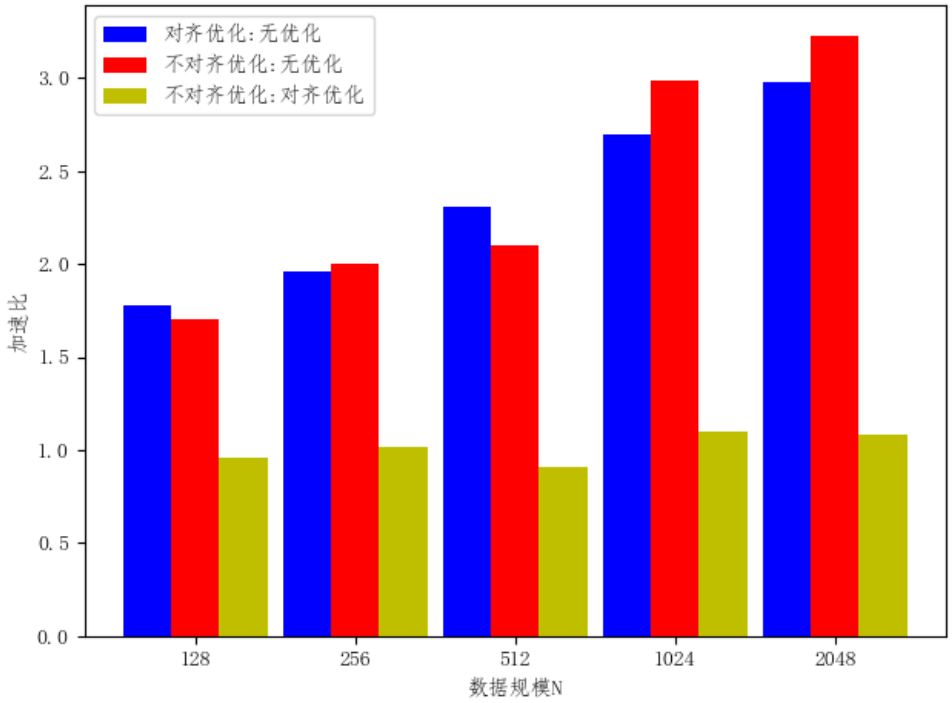


图 3: 对齐策略和不对齐策略的性能加速比

结合第三部分实验的实验结果，得出结论在总体来看，采用对齐策略和不对齐策略大致上性能相当。当数据规模较小时，在误差范围内可认为采用对齐策略性能和不对齐性能相等。在数据规模较大时，认为不对齐策略的性能比对齐策略的性能稍微好一点但不明显。

# 二、 总结

- 本次实验完成了在朴素的高斯消去算法的基础上使用SSE/AVX进行优化，讨论了对朴素串行算法的不同部分进行并行化后对程序性能的影响，得出单独对单独一个部分尤其是时间复杂度较低的除法部分进行向量并行化效果并不理想，需要同时进行并行化才能较好的得到加速的效果。
- 本次实验还研究探讨了使用SSE的128位对算法进行四路向量化并行和使用AVX的256位对算法进行八路向量并行化优化后的性能对比，发现使用八路向量最终能够达到4倍的加速效果。
- 本次实验也探究了在SSE向量并行化的时候采用数据对齐策略和数据不对齐策略的性能对比，发现在数据规模较小时差异并不明显，随着数据规模的增大，不对齐策略的性能小幅度优于对齐策略的性能。
- 分析时间复杂度，原朴素的串行算法的时间复杂度为 $O(n^3)$ 。本实验SIMD并行化对算法进行的优化本质上是对算法进行常数优化，最终时间复杂度还是 $O(n^3)$ 。
- 本次实验需要改进的地方：由于时间紧促，原定的在多平台上实现并行算法的性能比较没能完成。