
南開大學
Nankai University

计算机学院

并行程序设计报告

# SIMD 并行加速的高斯消去算法

姓名：赵元鸣 苏胤华

学号：2211757 2213893

专业：计算机科学与技术

2024 年 4 月 28 日

# 目录

# 1 普通高斯消元算法的并行优化

## 1.1 实验环境

### 1.1.1 x86 平台的主机环境如下：

Windows 10 64 位操作系统
CPU 型号 i7-12800h，8 核 16 线程，主频 3.7GHz
显卡：NVIDIA MX550 移动端 GPU，CUDA 核心：2560，显存：2GB

### 1.1.2 ARM 指令集的服务器平台则采用华为鲲鹏服务器 + 毕升编译器的设置

ARM 平台上编译选项使用 clang++ -g -armv8-a 对 cpp 文件进行编译

## 1.2 SIMD 的优化探究

### 1.2.1 平凡代码

普通高斯消去平凡代码

```
for (int k = 0; k < N; k++) {
        for (int j = k + 1; j < N; j++) {
            m[k][j] = m[k][j] / m[k][k];
        }
        m[k][k] = 1.0;
        for (int i = k + 1; i < N; i++) {
            for (int j = k + 1; j < N; j++) {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
            m[i][k] = 0;
        }
    }
```

SIMD（Single Instruction, Multiple Data）是一项并行计算技术，其核心概念是在单条指令下同时处理多个数据元素。通过将多个相同类型的数据打包成向量，然后利用一条指令对整个向量执行操作，从而实现高效的并行计算。在普通高斯消元算法中，可以利用 SIMD 进行优化的部分包括：

1. 在除法部分，可以一次性取多个浮点数进行除法运算，以提高计算效率，特别是针对首项元素（即对角线元素）的除法操作；

2. 在消去部分，当对某一行进行消去操作时，同样可以一次性取多个浮点数进行消去，以加速计算过程。

SIMD 的优化并不复杂，主要工作是将原本逐个进行除法和消元的部分替换为加载向量、向量除法/消去以及向量保存的流程。这样的优化可以有效地利用硬件的并行计算能力，提高算法的执行效率。

### 1.2.2 优化代码

优化代码设计如下：

NEON 优化算法

```
for(int k = 0; k < N; k++){
        float32x4_t vt = vdupq_n_f32(m[k][k]);
        int j = 0;
        for(j = k+1; j+4 <= N; j+=4){
                float32x4_t va = vld1q_f32(&m[k][j]);
                va = vdivq_f32(va, vt);
                vst1q_f32(&m[k][j], va);
        }
        for( ;j < N; j++){
                m[k][j] = m[k][j]/m[k][k];
        }
        m[k][k] = 1.0;
        for(int i = k+1; i < N; i++){
                float32x4_t vaik = vdupq_n_f32(m[i][k]);
                for(j = k+1; j+4 <= N; j+=4){
                        float32x4_t vakj = vld1q_f32(&m[k][j]);
                        float32x4_t vaij = vld1q_f32(&m[i][j]);
                        float32x4_t vx = vmulq_f32(vakj, vaik);
                        vaij = vsubq_f32(vaij , vx);
                        vst1q_f32(&m[i][j] , vaij);
        }
        for( ; j < N; j++){
                m[i][j] = m[i][j]-m[i][k]*m[k][j];
        }
        m[i][k] = 0;

    }
}
```

sse 优化算法

```
for (int k = 0; k < N; k++) {
        __m128 t1 = _mm_set1_ps(m[k][k]);
        int j = 0;
        for (j = k + 1; j + 4 <= N; j += 4) {
            __m128 t2 = _mm_loadu_ps(&m[k][j]);    //未对齐，用loadu和storeu指令
            t2 = _mm_div_ps(t2, t1);
            _mm_storeu_ps(&m[k][j], t2);
        }
        for (; j < N; j++) {
            m[k][j] = m[k][j] / m[k][k];
        }
        m[k][k] = 1.0;
        for (int i = k + 1; i < N; i++) {
```

```
            __m128 vik = _mm_set1_ps(m[i][k]);
            for (j = k + 1; j + 4 <= N; j += 4) {
                __m128 vkj = _mm_loadu_ps(&m[k][j]);
                __m128 vij = _mm_loadu_ps(&m[i][j]);
                __m128 vx = _mm_mul_ps(vik, vkj);
                vij = _mm_sub_ps(vij, vx);
                _mm_storeu_ps(&m[i][j], vij);
            }
            for (; j < N; j++) {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
            m[i][k] = 0;
        }
    }
```

avx 优化算法

```
for (int k = 0; k < N; k++) {
        __m256 t1 = _mm256_set1_ps(m[k][k]);
        int j = 0;
        for (j = k + 1; j + 8 <= N; j += 8) {
            __m256 t2 = _mm256_loadu_ps(&m[k][j]);
            t2 = _mm256_div_ps(t2, t1);
            _mm256_storeu_ps(&m[k][j], t2);
        }
        for (; j < N; j++) {
            m[k][j] = m[k][j] / m[k][k];
        }
        m[k][k] = 1.0;
        for (int i = k + 1; i < N; i++) {
            __m256 vik = _mm256_set1_ps(m[i][k]);
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 vkj = _mm256_loadu_ps(&m[k][j]);
                __m256 vij = _mm256_loadu_ps(&m[i][j]);
                __m256 vx = _mm256_mul_ps(vik, vkj);
                vij = _mm256_sub_ps(vij, vx);
                _mm256_storeu_ps(&m[i][j], vij);
            }
            for (; j < N; j++) {
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            }
            m[i][k] = 0;
        }
    }
```

## 1.3 优化结果

在 ARM 平台和 x86 平台上，在不同大小的矩阵下经过多次实验后得到的最终结果如下：

| 指令集 | 向量类型 | dupTo4FLoat | load4Float | 向量乘法 | 向量减法 |
|---|---|---|---|---|---|
| SSE | _m128 | _mm_set1_ps | _mm_loadu_ps | _mm_storeu_ps | _mm_mul_ps |
| AVX | _m256 | _mm256_set1_ps | _mm256_loadu_ps | _mm256_storeu_ps | _mm256_mul_ps |
| NEON | float32x4_t | vdupq_n_f32 | vld1q_f32 | vst1q_f32 | vmulq_f32 |

| | 500 | 1000 | 2000 | 3000 | 4000 |
|---|---|---|---|---|---|
| 平凡算法平均耗时（s） | 0.3564 | 2.9845 | 23.6564 | 78.4589 | 182.4256 |
| NEON 优化平均耗时（s） | 0.2865 | 2.4656 | 18.7854 | 62.6548 | 147.9854 |
| 加速比 | 1.1120 | 1.1894 | 1.1798 | 1.2456 | 1.2356 |

表 1: ARM 平台上 NEON 指令集 SIMD 优化效果

| | 500 | 1000 | 2000 | 3000 | 4000 |
|---|---|---|---|---|---|
| 平凡算法平均耗时（ms） | 27.2 | 211.0 | 189.0 | 4241.0 | 9252.3 |
| SSE 优化平均耗时（ms） | 9.7 | 85.2 | 938.0 | 2252.3 | 4984.0 |
| SSE 加速比 | 2.90 | 2.60 | 1.97 | 1.96 | 1.87 |
| AVX 优化平均耗时（ms） | 6.3 | 48.3 | 653.6 | 1726.0 | 3966.2 |
| AVX 加速比 | 5.65 | 4.65 | 3.00 | 2.90 | 2.60 |

表 2: x86 平台上 SSE 和 AVX 指令的 SIMD 优化效果

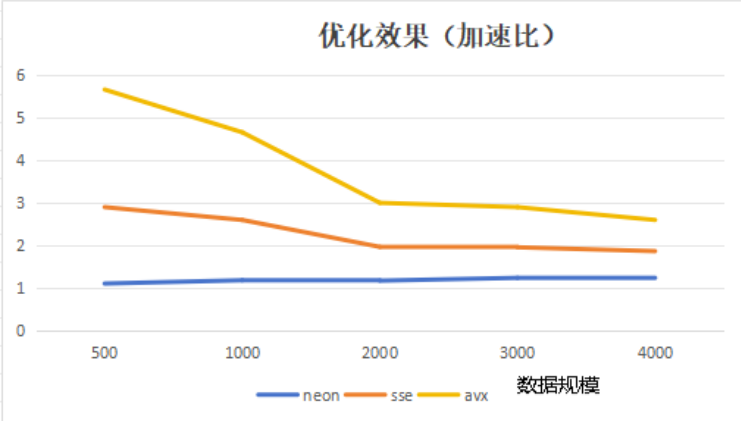



图 1.1: 优化效果

利用 NEON 指令集成功实现了速度提升，在数据规模较小时，由于并行部分的计算占比不大，所以提升不明显，加速比仅有 1.0970，然而随着数据规模提升，执行时间上的差异愈发可见，优化效果不断变得显著，当数据规模来到 3000*3000 以上时，耗时基本只有平凡算法的 80，加速比来到 1.25。不过由于整个算法中有其他环节的耗时，以及 NEON 指令集自身的执行开销，导致加速比会低于 NEON 的加速比 4。

可以发现 SSE 和 AVX 指令集都有很明显的加速效果，二者在 x86 平台的优化效果良好，可以用在最终的高斯消元算法并行加速上。

在 x86 平台上分别尝试了 128 位向量的 SSE 指令集和 256 位的 AVX 指令集，前者可以一次性处理 4 个浮点数，和后者可以一次性处理 8 个浮点数。

在优化结果上，二者都表示出：数据规模小时加速比比较大，然而随着数据规模增长，会逐渐下降并趋于平缓。分析可能是因为内存可能没有对齐导致多余的访存操作，以及数据规模增大后优化前后都需要进行巨大运算等因素造成的。

# 2 特殊高斯消去法

## 2.1 算法设计

### 2.1.1 平凡算法设计

特殊高斯消去算法来自于一个实际的密码学问题–Gronber 基的计算，他与普通高斯消去算法有以下的区别：首先，运算均为有限域 GF(2) 上的运算，即矩阵元素只可能是 0 或者 1，所以加法、减法运算实际上为异或运算，普通高斯消去中从一行减去另一行的操作退化为减法; 输入分为两部分，消元子与被消元行，与普通高斯消去对比，消元子相当于减数，被消元子充当被减数，消元子具有首个非零元素位于对角线的特征，同时输入的所有消元子首项不会覆盖所有对角线元素，而被消元子在消元过程中首项变化后不存在于所有消元子中，被消元子就会升格为消元子。

特殊消元法的串行（平凡）算法如下：1. 逐批次读取消元子与被消元子进入内存，重复执行以下步骤（分批次是因为矩阵规模可能十分庞大，内存无法同时存储）2. 检查当前批次每个被消元子，检查首项，如果有对应消元子，将其减去（异或）对应消元子，重复直到出现以下两种情况之一：变为空行、首项无对应消元子并且在范围内。3，对二中的两种情况分别做以下处理：第一种情况则将该被消元子丢弃，第二种情况则将被消元子加入消元子。4. 重复上述过程直到所有批次处理完毕，此时消元子与被消元行共同组成结果矩阵（可能存在许多空行）

伪代码

**Algorithm 1** 特殊高斯消元法

**Input:** 消元子与被消元行

**Output:** 消元子与被消元行组成的结果矩阵

**function** SpecialGauss($DivRowbeDivRow$)
  $for\ i = 1\ to\ n\ do$
    $while\ E_i\ != 0\ do$
      $if\ R_{lp_{(E_i)}} != NULL\ then$
        $E_i = E_i - R_{lp_{(E_i)}}$
      $else\ R_{lp_{(E_i)}} = E_i$
      $end\ if$
    $end\ while$
  $end\ for$
  **return** $E$
**end function**

### 2.1.2 SSE、AVX 优化算法

首先对平凡算法进行分析，特殊高斯消去法主要的运算部分有：消元子与被消元子提取，消元子与被消元子异或运算，被消元子升格。所以主要可以并行的部分在于异或运算，在此基础上引入 SSE、AVX 进行优化。

以下是 SSE 与 AVX 指令集的基本介绍：SIMD 编程思想是将要运算的数据打包为向量同时进行运算，SSE 与 AVX 是 SIMD 的两种指令集，其中 SSE 支持 8 个 128 位的向量寄存器，AVX 支持 16 个 256 位的向量寄存器，这两种指令集具有相似的指令命名方式，即指令名称中蕴含数据位宽、精度、向量长度、对齐、高低位等信息，指令类型分为移动、算数、逻辑、比较、洗牌等类型。总的来说 AVX 指令集可以看作 SSE 指令集的扩展，但是也要求更高的性能消耗。

根据平凡算法的特点与 SIMD 的特点，本次实验主要对消元子与被消元子的异或运算进行并行优化，对于 SSE 指令集，每次循环对四个 32 位数做运算，对于 AVX 指令集，每次循环对八个 32 位数进行运算。

## 2.2 编程实现

### 2.2.1 平凡算法

根据需求，代码分为以下四个部分：计时、消元子与被消元子数据读取、异或运算，消元子更新。

计时代码

```
void ord()
{
    double head,tail,freq,head1,tail1,timess=0; // timers
    init(N);
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq );
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    (Other code...)
    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    cout << "\nordCol:" <<(tail-head)*1000.0 / freq<< "ms" << endl;
}
```

对于计时代码，为了能更好的对比 SSE 优化、AVX 优化对平凡串行算法的加速比，在主函数中计算全部过程运行的时间，同时还要记录文件读取过程的时间，总时间减去读取时间作为最后的衡量标准计算加速比，

同时，注意到 SSE 与 AVX 向量中每个数据长度都是 32 位，而消元子和被消元子只有 1 或者 0 并且后续进行运算都是异或形式，考虑不把单独的 0、1 作为元素处理，而是将 32 个 0 或 1 元素作为一个数存入向量的一个位置，这样做既可以在一次读取中读入更多的数据，也可以充分利用 SSE 与 AVX 指令集的运算，以 SSE 为例，对比两种存储方式，优化方式可以做到一次计算 128 个元素，是 01 元素单独存储的 32 倍。

数据读取

```
void store()
{
  int index = pos / 32;
  int offset = pos % 32;
  Beixiaoyuan[row][index] = Beixiaoyuan[row][index] | (1 << offset);
}
```

如上述代码所示，每次读取记录三个变量作为下一个要存储的位置：row、index、offset，分别表示行号、第几个 32 位数、当前存入的 0 或 1 是 32 位对应的第几位。而数据的读取方式则是使用 C++ 中

的输入输出流库，先使用 getline 获取 txt 文件中的一行，再使用重定义的运算符 »，可以从一个字符串中读取到下一个空格前的内容并自动识别数据类型，在本次实验中对应的就是 32 位 int 数。

### 2.2.2 SIMD 并行算法算法

SSE 优化

```
void parallel()
{
___m256i vij = _mm256_loadu_si256((___m256i*) & Beixiaoyuanzi[i][j]);
  ___m256i vj = _mm256_loadu_si256((___m256i*) & Xiaoyuanzi[j]);
  ___m256i vx = _mm256_xor_si256(vij, vj);
  _mm256_storeu_si256((___m256i*) & Beixiaoyuanzi[i][j], vx);
}
```

AVX 优化

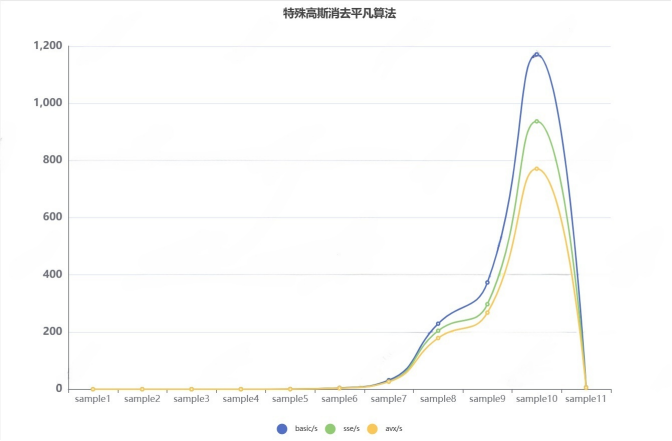
```
void parallel()
{
        ___m128i vij = _mm_loadu_si128((___m128i*)&Beixiaoyuanzi[i][j]);
        ___m128i vj = _mm_loadu_si128((___m128i*)&Xiaoyuanzi[j]);
        ___m128i vx = _mm_xor_si128(vij, vj);
        _mm_storeu_si128((___m128i*)&Beixiaoyuanzi[i][j], vx);
}
```

## 2.3 性能测试

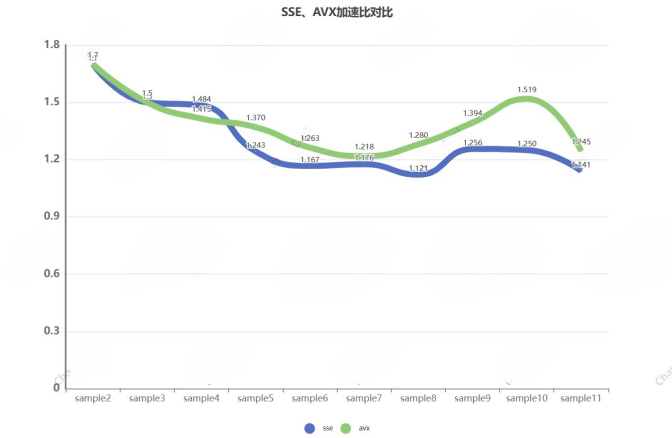
本次特殊高斯消去的 SIMD 编程选择在 windowsX86 架构下的 codeBlock 进行，环境配置如下：General：C++17GNU、X86(32bit);Optimization：-O2；CPU：Intel Pentium 4 Prescott，在该环境下对程序进行测试，对每一个样例循环运行五次后取运行的平均时长，得到如下运行结果。

| 单位/s | sample1 | sample2 | sample3 | sample4 | sample5 | sample6 | sample7 | sample8 | sample9 |
|---|---|---|---|---|---|---|---|---|---|
| basic | 0 | 0.0034 | 0.003 | 0.092 | 0.363 | 4.529 | 31.586 | 229.099 | 373.281 |
| sse | 0 | 0.002 | 0.002 | 0.062 | 0.292 | 3.882 | 26.851 | 204.331 | 297.202 |
| avx | 0 | 0.002 | 0.002 | 0.065 | 0.265 | 3.587 | 25.931 | 178.939 | 267.758 |

| sample10 | sample11 |
|---|---|
| 1170.736 | 6.073 |
| 936.724 | 5.324 |
| 770.964 | 4.879 |




为了直观体会到 SSE 与 AVX 的加速效果，同时也为了能够清晰地对比 SSE 与 AVX 对程序的加速程度的区别，用优化算法的运行时间除去平凡算法运行时间得到加速比，结果如下。




## 2.4 Profiling

对比平凡、SSE、AVX 三算法运行的时间可以明显的发现 AVX 优于 SSE，而 SSE 优于平凡算法，对于矩阵规模较小的 sample，各种算法运行的时间基本相同，没有太大的参考价值，而当矩阵规模逐渐扩大，并行算法的优势就逐步开始体现，可以发现 SSE 的加速比较为稳定大约保持在 1.1 到 1.2 之间，而 AVX 加速比在各个 sample 中加速比都优于 SSE，可以肯定性能是更好的，但是在各个 sample 的加速比波动也略大，在 1.2 到 1.5 之间，其中 AVX 算法在 sample7 中表现最差，和 SSE 算法最为接近，而在 sample10 中表现出最好的加速比，通过观察给出的数据发现，AVX 加速比的变化趋势和被消元子与问题规模大小的比例是很一致的，也就是说初步可以认为，对相同的问题输入规模，需要处理的元素越多 AVX 相对于 SSE 和串行算法的优势就更加明显，这一点在 sample11 也可以得到进一步验证，观察 sample11 发现 sapmple11 尽管规模最大，但却是一个非常稀疏的矩阵，被消元行仅有 756，印证了在 sample11 运行结果上 SSE 和 AVX 性能差距较小的事实。

通过分析代码，发现在做并行化处理过程中实际上存在数据前后依赖的问题，在对被消元子进行消元时，由于同时进行多个消元子的消元，每个消元子在完成消元之后都可能升格为消元子，这就导致前一个被消元子升格后可能会对下一个被消元子做消元操作，这种情况不会对程序运行时间产生较大影响，但是会对结果的正确性产生影响，如果要精确的判断某个被消元子升格后再同一次并行操作中是否有其他作用可能就需要不同并行之间的通信操作，可以在之后的实验当中进行优化（在本次实验当中按照顺序进行处理）。

同时在程序运行过程当中发现如果采用一次读取所有数据的方式，会花费大量时间用于读取数据，考虑采取分批次处理的方式进行，由于时间关系目前只有基本的思路，考虑在后续的实验当中验证。第一步是数据划分，即将数据根据规模大小划分为若干组，每组先分别处理（也就是采取 SIMD 做并行的部分），把每组的结果都保存在 map 的数据结构当中，同时在这一步当中每组分别处理的过程也可以进行并行，第二步是结果合并，依次合并，每次对两组结果进行合并具体过程和先前一步类似，但是由于大部分数据都应该已经经过消元，因此这一步效率不会太低。

### 2.5 结果分析

在本次实验当中，我对特殊高斯消元法进行了实现，并进行了 SIMD 的并行化编程，通过结果来看，和预期基本保持一致，SSE 和 AVX 都实现了一定程度的优化，但是虽然加速比趋势符合预期，加速比本身的大小并不是非常理想，从这个方面来说还需要进一步的程序优化。

## 3 其他事项

### 3.1 源代码查阅

本次实验的源代码储存在了 github 的 https://github.com/zhaoyuanmingzhendeshuai/parallel-programming，如果有需要，请读者自行取用。

### 3.2 小组分工

本次实验由赵元鸣进行普通高斯消去法的优化研究，由苏胤华进行特殊高斯消去法的优化研究。