南開大學
Nankai University

计算机学院

并行程序设计作业报告

SIMD 编程

姓名：徐文斌

学号：2010234

专业：计算机科学与技术

2022 年 4 月 12 日

目录

1 普通高斯消去算法

1.1 实验介绍

高斯消元法（Gaussian Elimination）是数学上线性代数中的一个算法，可以把矩阵转化为行阶梯形矩阵。高斯消元法可用来为线性方程组求解，求出矩阵的秩，以及求出可逆方阵的逆矩阵。

本实验将在 ARM 平台上利用 NEON 进行普通高斯算法的 SIMD 优化实验，从而对 SIMD 编程有更加深刻的认识与理解。

1.2 程序设计思路

1.2.1 算法分析

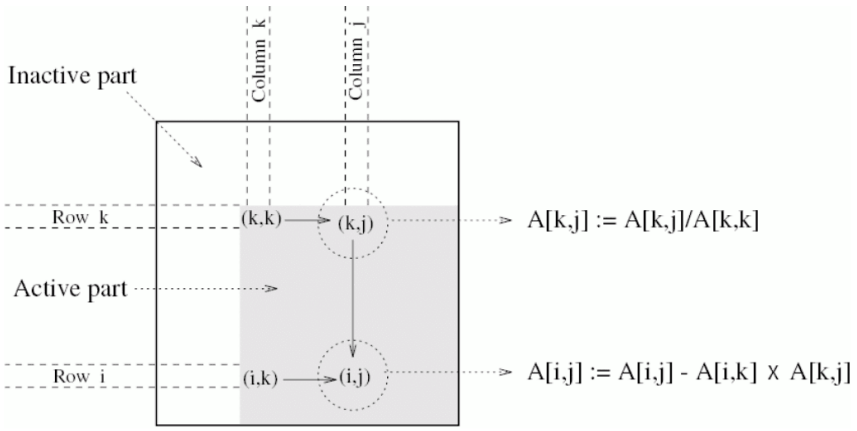

图 1.1: 高斯消去算法示意图

高斯消去的计算模式如图1.1所示，在第 k 步时，对第 k 行从 (k, k) 开始进行除法操作，并且将后续的 k + 1 至 N 行进行减去第 k 行的操作，串行算法如下面伪代码所示。

```
1  procedure LU (A)
2  begin
3      for k := 1 to n do
4          for j := k + 1 to n do
5              A[k, j ] := A[k, j] / A[k, k];
6          endfor;
7          A[k, k] := 1.0;
8          for i := k + 1 to n do
9              for j := k + 1 to n do
10                 A[i, j ] := A[i, j ] - A[i, k] * A[k, j];
11             endfor;
12             A[i, k] := 0;
13         endfor;
14     endfor;
15 end LU
```

由高斯消去伪代码容易推出该算法时间复杂度为 $O(n^3)$。观察高斯消去算法，注意到伪代码第 4，5 行第一个内嵌循环中的 A[k, j] := A[k, j]/A[k, k] 以及伪代码第 8，9，10 行双层 for 循环中的 A[i, j] := A[i, j] − A[i, k]×A[k, j] 都是可以进行向量化的循环。可以通过 SIMD 扩展指令对这两步进行并行优化。

1.2.2 算法设计

平凡算法

```
1  //平凡算法
2  for (int k = 0; k < n; k++) {
3      float ele = A[k][k];
4      for (int j = k + 1; j < n; j++)
5          A[k][j] = A [k][j] / ele;
6      A[k][k] = 1.0;
7      for (int i = k + 1; i < n; i++) {
8          for (int j = k + 1; j < n; j++)
9              A[i][j] = A[i][j] - A[i][k] * A[k][j];
10         A[i][k] = 0;
11     }
12 }
```

伪代码中第 4、5 行未对齐优化算法（以下简称优化算法 1）

```
1  //优化算法 1
2  for (int k = 0; k < n; k++) {
3      float ele = A[k][k];
4      float32x4_t v1 = vmovq_n_f32(ele);
5      float32x4_t v0;
6      int l;
7      for (l = k + 1; l <= n - 4; l += 4) {
8          v0 = vld1q_f32(A[k] + l);
9          v0 = vdivq_f32(v0, v1);
10         vst1q_f32(A[k] + l, v0);
11     }
12     for (l; l < n; l++)
13         A[k][l] = A[k][l] / ele;
14     A[k][k] = 1.0;
15     for (int i = k + 1; i < n; i++) {
16         for (int j = k + 1; j < n; j++)
17             A[i][j] = A[i][j] - A[i][k] * A[k][j];
18         A[i][k] = 0;
```

```
    }
}
```

伪代码中 8、9、10 行未对齐优化算法（以下简称优化算法 2）

```
//优化算法 2
for (int k = 0; k < n; k++) {
    float ele = A[k][k];
    for (int j = k + 1; j < n; j++)
        A[k][j] = A [k][j] / ele;
    A[k][k] = 1.0;
    for (int i = k + 1; i < n; i++) {
        float32x4_t v1 = vmovq_n_f32(A[i][k]);
        float32x4_t v0, v2;
        int j;
        for (j = k + 1; j <= n - 4; j += 4) {
            v2 = vld1q_f32(A[k] + j);
            v0 = vld1q_f32(A[i] + j);
            v2 = vmulq_f32(v1, v2);
            v0 = vsubq_f32(v0, v2);
            vst1q_f32(A[i] + j, v0);
        }
        for (j; j < n; j++)
            A[i][j] = A[i][j] - A[i][k] * A[k][j];
        A[i][k] = 0;
    }
}
```

优化算法 1 与优化算法 2 相结合的未对齐优化算法（以下简称优化算法 3）

```
//优化算法 3
for (int k = 0; k < n; k++) {
    float ele = A[k][k];
    float32x4_t v1 = vmovq_n_f32(ele);
    float32x4_t v0;
    int l;
    for (l = k + 1; l <= n - 4; l += 4) {
        v0 = vld1q_f32(A[k] + l);
        v0 = vdivq_f32(v0, v1);
        vst1q_f32(A[k] + l, v0);
    }
    for (l; l < n; l++)
```

```
            A[k][l] = A[k][l] / ele;
        A[k][k] = 1.0;
        for (int i = k + 1; i < n; i++) {
            v1 = vmovq_n_f32(A[i][k]);
            float32x4_t v2;
            int j;
            for (j = k + 1; j <= n - 4; j += 4) {
                v2 = vld1q_f32(A[k] + j);
                v0 = vld1q_f32(A[i] + j);
                v2 = vmulq_f32(v1, v2);
                v0 = vsubq_f32(v0, v2);
                vst1q_f32(A[i] + j, v0);
            }
            for (j; j < n; j++)
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            A[i][k] = 0;
        }
    }
```

优化算法 3 对齐版本（以下简称优化算法 4）

```
    //优化算法 4
    for (int k = 0; k < n; k++) {
        float ele = A[k][k];
        long long addr;
        float32x4_t v1 = vmovq_n_f32(ele);
        float32x4_t v0;
        int l;
        for (l = k + 1; l < n; l++) {
            if (l % 4 == 0) //只需在初始化数组时使用 aligned_alloc 函数确保每行首地址 16
            //字节对齐，当 l 为 4 的倍数时，地址则对齐为 16 的倍数
                break;
            A[k][l] = A [k][l] / ele;
        }
        for (l; l <= n - 4; l += 4) {
            v0 = vld1q_f32(A[k] + l);
            v0 = vdivq_f32(v0, v1);
            vst1q_f32(A[k] + l, v0);
        }
        for (l; l < n; l++)
            A[k][l] = A[k][l] / ele;
        A[k][k] = 1.0;
```

```
22      for (int i = k + 1; i < n; i++) {
23          v1 = vmovq_n_f32(A[i][k]);
24          float32x4_t v2;
25          int j;
26          for (j = k + 1; j < n; j++) {
27              if (j % 4 == 0)//对齐原理同上
28                  break;
29              A[i][j] = A[i][j] - A[i][k] * A[k][j];
30          }
31          for (j; j <= n - 4; j += 4) {
32              v2 = vld1q_f32(A[k] + j);
33              v0 = vld1q_f32(A[i] + j);
34              v2 = vmulq_f32(v1, v2);
35              v0 = vsubq_f32(v0, v2);
36              vst1q_f32(A[i] + j, v0);
37          }
38          for (j; j < n; j++)
39              A[i][j] = A[i][j] - A[i][k] * A[k][j];
40          A[i][k] = 0;
41      }
42  }
```

1.3 实验环境说明

实验平台为鲲鹏服务器的 ARM 平台，利用 ARM 平台的 g++ 编译器进行程序的 SIMD 并行优化。

1.4 实验方案设计

矩阵规模与程序的运行时间直接相关。因此，从矩阵规模为 32×32 开始每次将 n 扩大为原来的 2 倍直至矩阵规模为 2048×2048，分别测试五个算法的运行时间，同时结合 perf 性能评测工具对程序的性能进行更加细致的分析。

1.5 实验结果呈现及分析

根据控制变量原则，我们首先对未对齐的前四个算法的性能进行分析，以讨论优化程序不同部分对程序性能的影响。然后将优化算法 3 和优化算法 4 进行对比分析，以讨论对齐和不对齐对 SIMD 并行编程程序性能优化的影响。

1.5.1 并行加速程序不同部分对程序性能影响分析

根据实验方案进行实验，首先，我们对不进行地址对齐的前四个算法性能进行分析。结果如表1所示。

利用测试结果计算各个优化算法的加速比，如表2所示。

算法\规模	32	64	128	256	512	1024	2048
平凡算法	0.0876422	0.700277	5.54199	44.208	358.435	3013.24	25026
优化算法 1	0.0869498	0.695199	5.53052	44.176	361.927	2963.69	24825.8
优化算法 2	0.0630006	0.462591	3.56502	28.0859	228.285	1884.98	15271.1
优化算法 3	0.062619	0.461182	3.56673	28.1321	226.686	1847.58	14764.8

表 1: 普通高斯消去未对齐算法程序运行时间测试结果（单位:ms）

算法\规模	32	64	128	256	512	1024	2048
优化算法 1	1.008	1.007	1.002	1.0007	0.9904	1.017	1.008
优化算法 2	1.391	1.514	1.554	1.574	1.570	1.598	1.639
优化算法 3	1.400	1.518	1.554	1.571	1.581	1.631	1.695

表 2: 普通高斯消去未对齐各个优化算法加速比

从表1不难看出，每当 n 扩大到原来的两倍，算法的执行时间大致变为原来的 8 倍，符合 $O(n^3)$ 的变化趋势。

根据上述数据继续对算法进行大致分析，我们可以得到三点信息：1. 优化算法 1 并没有起到很好的优化效果，其与平凡算法相比只有微小的优化，有时甚至效率低于平凡算法。2. 优化算法 2 有着较为明显的优化效果。3. 优化算法 3 的优化效果与优化算法 2 的优化效果相近，只有微小的提升。

联系阿姆达尔定律，分析可能的原因。不难发现，优化算法 1 只是优化了高斯消去算法的 $O(n^2)$ 的部分，即只优化了高斯消去算法中的一个二重循环。然而，我们在算法分析部分已经提出，高斯消去算法的时间复杂度为 $O(n^3)$，因此，优化算法 1 的优化相对于对整个问题的求解中进行的总的计算次数来说十分微小，故而优化算法 1 未能得到较好的加速比。这也解释了优化算法 3 和优化算法 2 的优化效果相近这一事实。

同理，优化算法 2 和 3 对高斯消去算法中的 $O(n^3)$ 部分，即三重循环部分进行优化，抓住了限制程序性能的关键瓶颈。优化算法 2 可以大大地减少高斯消去算法的执行计算的指令条数，因此具有较为可观的加速比。

为了对我们的分析进行验证，利用 linux 下的 perf 程序评测工具对程序运行中各个算法所执行的指令数及消耗的时钟周期进行分析，结果如表3所示。可以看到，平凡算法和优化算法 1 的指令数和时钟周期数均相近。当问题达到一定规模后，优化算法 2 和优化算法 3 执行的指令数大致为平凡算法的 1/3，执行消耗周期数也明显少于平凡算法。由此可以证明我们的分析是正确的。我也从而学习到要抓住程序的重点对其优化，要首先考虑优化程序中最耗时的部分，对于次要部分的优化并不能对加速程序运行起到很好的效果。

算法\规模		32	64	128	256	512	1024
平凡算法	instructions	591879	4797759	37855867	302157257	2416617721	19330658417
	cycles	228458	1807966	14409370	114189906	923955377	7765188769
优化算法 1	instructions	579242	4601624	37385092	301923750	2413828243	19315811521
	cycles	222234	1805046	14182910	114140107	933588274	7637028650
优化算法 2	instructions	261251	1857794	13541711	105077871	821036340	6502940389
	cycles	162397	1203342	9248223	72258854	588409462	4856939907
优化算法 3	instructions	250331	1759041	13504787	103443326	818246862	6495516941
	cycles	161317	1173482	9243910	72856447	584395755	4758355200

表 3: 普通高斯消去未对齐各算法执行指令数及周期数

1.5.2 是否采取对齐策略对程序性能影响分析

可以看到，采取 SIMD 并行编程可以很好地提升程序地性能。接下来，我们对是否对齐对 SIMD 并行编程的影响进行分析，具体为对比分析优化算法 3 和优化算法 4。优化算法 3 和优化算 4 的运行时间及加速比如表4所示。

算法\规模		32	64	128	256	512	1024	2048
优化算法 3	时间	0.062619	0.461182	3.56673	28.1321	226.686	1847.58	14764.8
	加速比	1.400	1.518	1.554	1.571	1.581	1.631	1.695
优化算法 4	时间	0.0604395	0.442393	3.38704	26.6318	210.871	1676.98	13391.6
	加速比	1.450	1.583	1.636	1.660	1.700	1.797	1.869

表 4: 普通高斯消去优化算法 3、4 运行时间 (ms) 及加速比

我们发现在各个问题规模下，对齐的优化算法 4 的加速效果均好于优化算法 3。而且当问题规模较大时，优化算法 4 具有明显的加速比提升，在 1024 和 2048 的问题规模下，优化算法 4 相比于优化算法 3 的加速比有了近 10% 的提升。

显然，这种差异应该是地址对齐的 load 和 store 指令具有更短的时钟周期造成的。利用 perf 分析优化算法 3 和优化算法 4 在指令数和时钟周期数上的差异，结果如表5所示。

算法\规模		32	64	128	256	512	1024
优化算法 3	instructions	250331	1759041	13504787	103443326	818246862	6495516941
	cycles	161317	1173482	9243910	72856447	584395755	4758355200
优化算法 4	instructions	255397	1805649	13926511	103829008	820011474	6501602917
	cycles	154828	1130761	8766697	68365326	544056384	4322917810

表 5: 普通高斯消去优化算法 3、4 执行指令数及时钟周期

从表5中我们可以看到，在每个问题规模下，对齐的优化算法 4 所执行的指令数均大于优化算法 3 执行的指令数，这可能是由于为了对齐地址所进行的循环操作造成的；而优化算法 4 的时钟周期数均少于优化算法 3，由此可见对齐的 load 和 store 指令具有更短的时钟周期。

2 特殊高斯消去算法

2.1 实验介绍

特殊高斯消去算法源自布尔 Gröbner 基计算。该算法的特征为：

- 运算均为有限域 GF(2) 上的运算，即矩阵元素的值只可能是 0 或 1。其加法运算实际上为异或运算：0+0=0、0+1=1、1+0=1、1+1=0。由于异或运算的逆运算为自身，因此减法也是异或运算。乘法运算 0*0=0、0*1=0、1*0=0、1*1=0。因此，高斯消去过程中实际上只有异或运算——从一行中消去另一行的运算退化为减法。
- 矩阵行分为两类，“消元子”和“被消元行”，在输入时即给定。消元子是在消去过程中充当“减数”的行，不会充当“被减数”。所有消元子的首个非零元素（即首个 1，称为首项）的位置（可通过将消元子放置在特定行来令该元素位于矩阵对角线上）都不同，但不会涵盖所有对角线元素。被消元行在消去过程中充当“被减数”，但有可能恰好包含消元子中缺失的对角线 1 元素，此时它“升格”为消元子，补上此缺失的对角线 1 元素。

本实验将对特殊高斯消去算法进行实现，并利用 SIMD 编程对算法进行优化。实验将分别利用基于 ARM 平台 NEON 的 SIMD 编程和利用基于 x86 平台 SSE 的 SIMD 编程对特殊高斯消去算法程序进行优化，并比较两者优化效果的差异。其中 ARM 平台使用鲲鹏服务器，并使用 g++ 编译器编译程序；x86 平台使用 Intel DevCloud 环境，并使用 icpc 编译器编译程序。

2.2 程序设计思路

2.2.1 算法分析

首先考虑算法的执行流程，我们需要循环遍历每一个消元行对其进行消元。对于每一个消元行的消元，消元过程首先寻找是否有消元子与该行具有相同的首项，若存在满足条件的消元子，则用相应的消元子对该消元行进行消去，即将消元子的每一位和消元行进行异或操作，之后判断消元行是否消元为 0，若仍未被消元为 0，则重复寻找消元子继续进行消元；若没有满足条件的消元子，则该消元行“升格”为消元子，参与后续消元行的消元过程，并结束该消元行的消元过程。

伪代码如下：

```
//R: 所有消元子构成的集合
//R[i]: 首项为 i 的消元子
//E: 所有被消元行构成的数组
//E[i]: 第 i 个被消元行
//lp(E[i]): 被消元行第 i 行的首项
for i := 0 to m - 1 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]
            break
        end if
    end while
end for
return E
```

2.2.2 算法设计

考虑实现算法使用的数据结构。首先是数据的存储，为了节约内存，我们使用位图的形式存储矩阵，并将消元行和消元子存储在同一个二维矩阵 A 之中，矩阵的前 NUME 行为消元行，后 NUMR 行为消元子行。同时，我们还需要记录每一行的首项值，定义一位数组 lp 进行值得存储。最后，为了快速地查找与某一消元行具有相同首项的消元子，我们使用 *unordered_map* 数据结构，该数据结构维护了一个哈希表，可以实现 key 到 value 的快速查找，将变量名定义为 lpE2R。

接下来，考虑算法的可优化部分，根据算法分析部分易得，我们可以对消元子对消元行进行消去的循环进行 SIMD 编程优化。

具体算法如下：

平凡算法

```
//平凡算法
for (int i = 0; i < NUME; i++) {
    while (lp[i] > -1) { //lp[i] 为-1 则表示该行被消为零
        if (!(lpE2R.find(lp[i]) == lpE2R.end())) { //寻找符合的消元子
            int rowR = lpE2R[lp[i]];
            bool lpHasChanged = false;  //该 bool 变量用于判断本消元行是否消元为零
            int p = totalCols - 1;
            for (p; p >= 0; p--) {
                A[i][p] ^= A[rowR][p]; //消元过程
            }
            for (p = totalCols - 1; p >= 3; p -= 4) {
            //此循环及接下来的循环为更新该消元行的首项值
            //此处为了加快速度使用循环展开
                int x = ((A[i][p] | A[i][p - 1]) | (A[i][p - 2] | A[i][p - 3]));
                if (x == 0)
                    continue;
                break;
            }
            for (p; p >= 0; p--) {
                if (A[i][p] != 0) { //消元行的首项值一定在第一个不为零的 32 个 bit 中
                    lpHasChanged = true;
                    for (int k = 31; k >= 0; k--) {
                        if ((A[i][p] & (1 << k)) != 0) {
                            lp[i] = p * 32 + k;
                            break;
                        }
                    }
                    break;
                }
            }
            if (lpHasChanged == false) //判断该消元行是否被消元为零
                lp[i] = -1;
            }
        else {
            lpE2R[lp[i]] = i;   //找不到合适的消元子，该消元行成为消元子
            break;
        }
    }
}
```

NEON 优化算法

```
//NEON 优化算法
for (int i = 0; i < NUME; i++) {
    while (lp[i] > -1) {
        if (!(lpE2R.find(lp[i]) == lpE2R.end())) {
            int rowR = lpE2R[lp[i]];
            bool lpHasChanged = false;
            int32x4_t v0, v1;
            int p = 0;
            for (p; p <= totalCols - 4; p += 4) {
                v0 = vld1q_s32(A[i] + p);
                v1 = vld1q_s32(A[rowR] + p);
                v0 = veorq_s32(v0, v1);
                vst1q_s32(A[i] + p, v0);
            }
            for (p; p < totalCols; p++)
                A[i][p] ^= A[rowR][p];
            for (p = totalCols - 1; p >= 3; p -= 4) {
                int x = ((A[i][p] | A[i][p - 1]) | (A[i][p - 2] | A[i][p - 3]));
                if (x == 0)
                    continue;
                break;
            }
            for (p; p >= 0; p--) {
                if (A[i][p] != 0) {
                    lpHasChanged = true;
                    for (int k = 31; k >= 0; k--) {
                        if ((A[i][p] & (1 << k)) != 0) {
                            lp[i] = p * 32 + k;
                            break;
                        }
                    }
                    break;
                }
            }
            if (lpHasChanged == false)
                lp[i] = -1;
        }
        else {
            lpE2R[lp[i]] = i;
            break;
```

```
        }
    }
}
```

SSE 优化算法

```
//SSE 优化算法
for (int i = 0; i < NUME; i++) {
    while (lp[i] > -1) {
        if (!(lpE2R.find(lp[i]) == lpE2R.end())) {
            int rowR = lpE2R[lp[i]];
            bool lpHasChanged = false;
            __m128 v0, v1;
            int p = 0;
            for (p; p <= totalCols - 4; p += 4) {
                v0 = _mm_load_ps((float*)(A[i] + p));
                v1 = _mm_load_ps((float*)(A[rowR] + p));
                v0 = _mm_xor_ps(v0, v1);
                _mm_store_ps((float*)(A[i] + p), v0);
            }
            for (p; p < totalCols; p++)
                A[i][p] ^= A[rowR][p];
            for (p = totalCols - 1; p >= 3; p -= 4) {
                int x = ((A[i][p] | A[i][p - 1]) | (A[i][p - 2] | A[i][p - 3]));
                if (x == 0)
                    continue;
                break;
            }
            for (p; p >= 0; p--) {
                if (A[i][p] != 0) {
                    lpHasChanged = true;
                    for (int k = 31; k >= 0; k--) {
                        if ((A[i][p] & (1 << k)) != 0) {
                            lp[i] = p * 32 + k;
                            break;
                        }
                    }
                    break;
                }
            }
            if (lpHasChanged == false)
                lp[i] = -1;
```

```
            }
            else {
                lpE2R[lp[i]] = i;
                break;
            }
        }
    }
```

2.3 实验环境说明

实验分为 ARM 平台和 x86 平台。其中 ARM 平台使用鲲鹏服务器，并使用 g++ 编译器编译程序；x86 平台使用 Intel DevCloud 环境，并使用 icpc 编译器编译程序。

2.4 实验方案设计

实验计划在课程提供的测试数据的前七组数据集上进行测试，采集不同平台不同算法程序运行的时间，进行对比分析。

2.5 实验结果呈现及分析

根据实验方案进行实验，得到不同平台平法算法与优化算法的运行时间如表6所示。表7为两平台优化算法的加速比。

平台\数据集编号		1	2	3	4	5	6	7
鲲鹏	平凡算法	0.0191553	0.745921	0.908	26.811	152.731	2534.5	32054.9
	优化算法	0.0194575	0.761949	0.921733	26.3242	137.913	2021.87	21608.3
DevCloud	平凡算法	0.0035091	0.183244	0.200015	5.33096	18.837	193.661	2136.21
	优化算法	0.0015959	0.162156	0.192901	2.04701	8.85248	131.617	1381.48

表 6: 特殊高斯消去算法不同平台下平凡算法与优化算法运行时间（ms）

平台\数据集编号	1	2	3	4	5	6	7
鲲鹏	0.984469	0.978964	0.985101	1.01849	1.10744	1.25354	1.48345
DevCloud	2.19882	1.13005	1.03688	2.60427	2.12788	1.4714	1.54632

表 7: 特殊高斯消去算法不同平台下优化算法加速比

首先我们进行纵向对比，比较每个平台内部程序运行结果的特点。可以看到，在 ARM 平台中，当数据集规模较小时，优化算法并未取得较好的加速比，当数据规模较大时优化算法才显现出较好的加速比。这可能是由于规模较小时 NEON 的 SIMD 指令减少程序指令数带来的优化效果未能弥补具有更长时钟周期的 SIMD 指令对程序的影响。在 DevCloud 平台中，优化算法加速比变化趋势并不是十分明显，但是加速比均大于 1，且在某些数据规模下可以达到 2 以上的加速比，具有较好的加速效果。

接下来我们进行横向对比，比较两个平台之间程序运行结果的差异。首先，一个最明显的差异是 DevCloud 平台下程序的运行时间远小于 ARM 平台下程序的运行时间。在某些数据规模下，ARM 平台程序运行时间甚至达到了 DevCloud 平台程序运行时间的 10 倍以上。由此可见平台差异对程序运行时间的影响之大。此外，可以看到 DevCloud 平台的 SSE 优化算法具有更好的加速比。

3 实验源码项目 GitHub 链接

实验源码项目 GitHub 链接