南開大學
Nankai University

计算机学院

并行程序设计实验报告

Gauss 消元的 SIMD 并行优化研究

姓名：刘心源

学号：2112614

专业：计算机科学与技术

2023 年 4 月 14 日

目录

1 问题描述

在进行科学计算的过程中，经常会需要对多元线性方程组进行求解。一种有效的方法就是 Gauss 消去法。通过对线性方程组的线性矩阵进行自上而下的逐行消去，将其变为主对角线元素均为 1 的上三角矩阵。

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1\ n-1} & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2\ n-1} & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1\ 1} & a_{n-1\ 2} & \cdots & a_{n-1\ n-1} & a_{n-1\ n} \\ a_{n\ 1} & a_{n\ 2} & \cdots & a_{n\ n-1} & a_{n\ n} \end{bmatrix} => \begin{bmatrix} 1 & a'_{12} & \cdots & a'_{1\ n-1} & a'_{1n} \\ 0 & 1 & \cdots & a'_{2\ n-1} & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & a'_{n-1\ n} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

图 1.1: Gauss 消去法的数学表示

考虑在整个消去过程中，如1.2所示，主要包含两个过程：

1. 对于当前的消元行，应该全部除以行首的元素，使得该行转化为行首元素为 1 的行
2. 对于之后的每一行，用该行减去当前的消元行，得到本次的消元结果

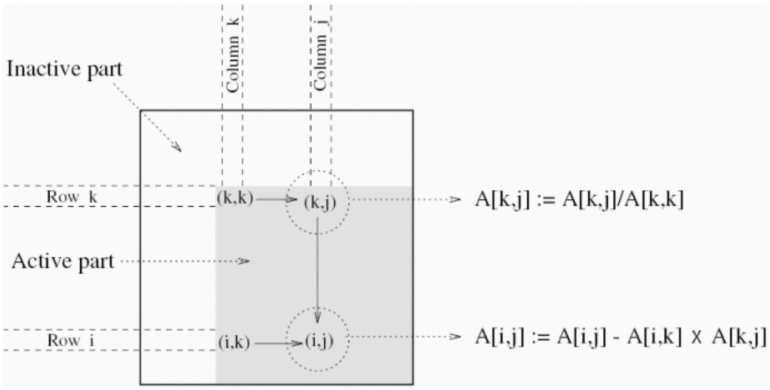


图 1.2: Gauss 消去法的逻辑表示

2 实验环境

	ARM	x86
CPU 型号	华为鲲鹏 920 处理器	Intel Core i7-1165G7
CPU 主频	2.60GHz	2.80GHz
L1 Cache	64KB	48KB
L2 Cache	512KB	1.25MB
L3 Cache	48MB	12MB
指令集	Neon	SSE、AVX、AVX-512

表 1: 实验环境

3 实验设计

3.1 数据规模设置

虽然就本题而言，运行时间与问题规模的变化趋势并不是我们的研究重点，但是为了保证测试的准确性，避免极端情况的出现，我们通过函数设置生成测试用例。

```
void Initialize(int n){
    for (int i = 0; i < n; i++){
        a[i][i] = 1.0;// 对角线初始化为 1
        for (int j = 0; j < i; j++) a[i][j] = 0;// 下三角初始化为 0
        // 上三角初始化为随机数
        for (int j = i + 1; j < n; j++) a[i][j] = rand(); // 上三角初始化为随机数
    }
    for (int k = 0; k < n; k++)
        for (int i = k + 1; i < n; i++)
            for (int j = 0; j < n; j++)
                a[i][j] += a[k][j];// 最终每一行的值是上一行与这一行之和
}
```

在进行测试的时候，我选取 2^n 数据规模的矩阵,n 的取值范围为 1~10。为了避免测试时间的偶然性，对于运行时间多次测量取平均值。

```
for (int n = 2; n <= N; n *= 2){
    Initialize(n);
    count = 1;
    /* 为了避免在数据规模较大时运行次数过多造成程序总运行时间过大
    * 设置不同数据规模下不同的重复测试次数
    */
    if (n <= 30) cycle = 1000;
    else if (n <= 70) cycle = 100;
    else if (n <= 300) cycle = 50;
    else cycle = 10;
    while (count < cycle){
        Gauss_Normal(n);
        count++;
    }
}
```

3.2 内存地址对齐

考虑到高斯消元计算过程中，对数据进行向量化存取的时候，第 k 步消去的起始元素是变化的，从而导致距 16 字节边界的偏移是变化。如果取数据的原始内存不是按照存取规模的大小内存对齐的，就

会导致在每一次数据向量化的时候，由于内存不对齐，需要进行两次的内存访问，然后将得到的数据再次进行拼接，返回给向量寄存器。CPU 在访问内存时，通常会按照一定的块大小（比如 4 字节或 8 字节）进行访问，如果数据未对齐，则需要额外的操作将其调整为正确的位置，这将浪费时间和资源，从而导致性能下降。

对于 Gauss 消元算法，我们通常需要处理大量的矩阵运算，这些运算需要访问大量的内存。如果我们将矩阵数据按照未对齐的方式存储，CPU 就需要额外的时间来访问和处理这些数据，这将导致程序的运行速度变慢。而如果我们将矩阵数据按照特定字节的倍数对齐，CPU 就可以更快地访问这些数据，从而提高程序的性能。

为了研究内存对齐对于程序耗时的影响，我们对 SSE 和 Neon 两种并行算法进行对齐处理。由于 C++ 中数组的初始地址一般为 16 字节对齐，所以只需要确保每次加载数据 A[i:i+3] 中 i 为 4 的倍数即可。为了方便代码的编写，我们测试的数据规模大部分为 16 的整倍数，这样仅需要首行地址对齐即可，简化了代码编写。内存对齐操作如下：

```
//针对第一部分的内存对齐
 while((k*n+j)%4!=0)
 {
    e[k][j]=e[k][j]*1.0/e[k][k];
    j++;
 }
 //针对第二部分的内存对齐
 while((k*n+j)%4!=0){//对齐操作
    e[i][j]=e[i][j]-e[k][j]*e[i][k];
    j++;
}
```

在计算过程之前检查所有数据数组的首地址是否对齐，若对齐则不做任何处理，如果未对齐则对未对齐的元素进行串行化操作，在 SSE 算法代码中加入对齐操作后，可以使用类似 _mm_load_ps 的指令进行 Gauss 并行算法的编写。Neon 算法内存对齐操作与 SSE 算法类似。对齐与未对齐的实验结果将在下面进行具体分析。

4 实验结果分析

4.1 x86 平台

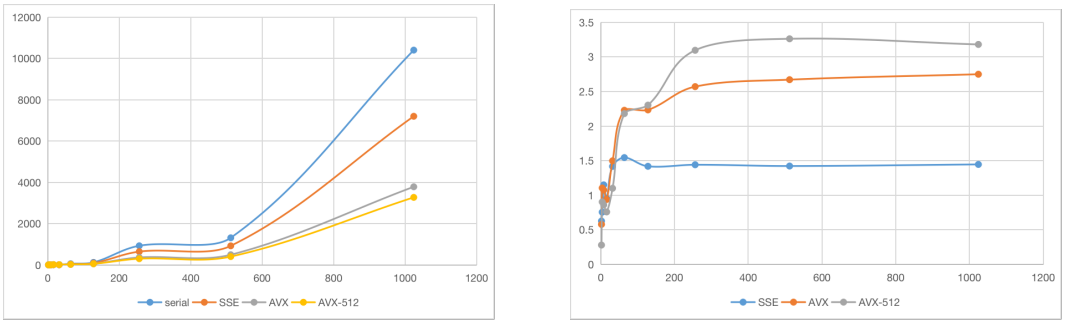
4.1.1 向量化的实现及其加速比对比

在 x86 平台上主要尝试 SSE、AVX、AVX512 这三种并行指令集架构，编写了 Gauss 消去的串行与采用这三种并行架构的并行算法（三种算法均未进行内存对齐操作）。根据所得的实验数据表2，对比实验效果，并计算出各种优化算法的加速比，如图4.3所示。

在数据规模较小的时候，由图表可得采用 SIMD 编程的三种并行算法基本与串行算法耗时相近或是更加耗时的情况，这是由于 SIMD 指令集的优势主要在于对于大量数据的并行处理，如果数据规模较小，使用 SIMD 指令集的优势会被弱化，反而会因为 SIMD 指令集使用的额外寄存器、数据对齐等操作产生额外开销。

N	serial	SSE	AVX	AVX-512
2	0.0612	0.0978	0.106	0.22
4	0.3333	0.4418	0.3018	0.3694
8	1.3007	1.1318	1.2103	1.5133
16	10.3817	10.8794	11.0741	13.7029
32	9.4381	6.6617	6.304	8.5808
64	54.553	35.3325	24.4773	25.0193
128	120.576	85.0293	53.9594	52.3476
256	925.823	642.736	360.11	298.923
512	1311.7	922.545	490.935	401.93
1024	10408.6	7201.78	3784.48	3271.39

表 2: x86 平台不同数据规模下并行优化效果



(a) 不同算法执行随问题规模变化

(b) 不同算法的加速比

图 4.3: 数据趋势图

当数据规模较大时 (>64) 时，由图表可知通过数据对比可以发现，由于采取的指令集不同，随着向量寄存器所能容纳的数据增多，并行算法的优化效果也在不断提升。这主要是由于采用向量化的优化，能够一次性处理多条数据，当数据规模和数据维度足够大的时候，各种并行算法的理论加速比应该逼近其向量寄存器一次性能够处理的数据的数量。但是与理论值仍然有差距，如表3所示。根据理论，SSE 指令集使用 128 位向量寄存器的指令集，可以一次处理 4 个单精度浮点数或者 2 个双精度浮点数，其加速比通常可以达到 2 倍左右。AVX 指令集加速比通常可以达到 3 倍左右。AVX-512 指令集加速比通常可以达到 4 倍。与上面类似地，我们推测加速比受到数组首地址未对齐、问题规模、内存带宽等影响。进行内存对齐操作造成运行时间上的额外开销，导致加速比并没有预期的那么高。在后面的部分我们对于对齐操作的影响进行具体分析。

加速比	理论值	测量值
SSE	2	1.4
AVX	3	2.7
AVX-512	4	3.2

表 3: 三种 SIMD 指令集算法加速比对比

实验在 Windows 环境的 Unbuntu20.04 虚拟机上使用 perf 对这几种算法的 CPU 硬件事件进行进一步的分析，其具体数据如下表4所示。性能测试中取数据规模为 n=1024，重复执行 10 次的结果。

通过对比指令数可以发现，三种并行优化算法在指令数上均远低于串行算法。并且其指令数与串行算法的比值也基本接近加速比，因此可以得出并行算法的优化效果主要体现在能够减少指令数上。其

中 AVX-512 的测量结果与预期有一点偏差，初步判断是因为性能测量过程中使用的笔记本 CPU 仅支持 AVX-512F 指令集，AVX512F 是 AVX512 的基本子集，能够支持基本的向量化计算，因此在本次实验中能够支持 AVX512 指令集架构的实验。但是在 CPU 上运行使用 AVX-512F 指令集的代码，就会导致部分代码执行效率下降。因此指令数和优化效果并未达到预期的效果。

algorithm	cycles	instructions	IPC
normal	48244181110	156453874193	3.24
SSE_unaligned	12739254547	26765019723	2.1
AVX_unaligned	9768192439	15356847677	1.57
AVX-512_unaligned	26610822499	51902089227	1.95

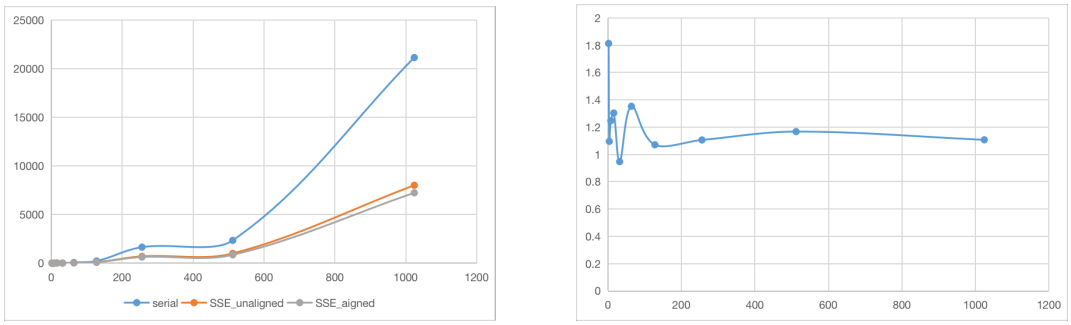
表 4: 不同算法所需的 cycles 和 instructions

4.1.2 对齐与不对齐算法策略的对比

在前面的研究中，我们已经发现内存对齐操作会对程序执行产生一些影响，以 SSE 指令集编程为例，测量对齐和未对齐的程序执行并进行对比。

N	serial	SSE_unaligned	SSE_aigned
2	0.0729	0.2372	0.1308
4	0.2752	0.6673	0.6087
8	1.2167	2.6523	3.438
16	18.3548	13.1319	10.0729
32	12.799	6.0039	6.3384
64	66.3679	39.0185	28.8377
128	220.065	96.5796	90.1879
256	1642.31	699.434	632.267
512	2341.54	999.919	856.548
1024	21139.9	8008.86	7731.91

表 5: SSE 指令集对齐与未对齐的执行时间测量



(a) 对齐与未对齐随问题规模变化 (b) 未对齐/对齐的加速比

图 4.4: 数据趋势图

可以从图4.4中直观地看到在进行内存对齐操作之后，程序的执行时间得到优化。因为 SSE 指令集要求数据按照 16 字节对齐。当数据按照 16 字节对齐时，可以使用 SSE 指令集的 Load 和 Store 指令来高效地加载和存储数据，从而提高程序的性能。此外，数据对齐还可以减少 SSE 指令集的处理时间，提高程序的效率。

在数据规模 n<16 时，内存对齐得到的优化并不明显，有可能耗时比未对齐的更大。当数据规模较大时，内存对齐操作的加速比稳定在 1.15 左右，对比之前有一定的提升。为了证实我们的结论，我们在 Windows 环境的 Unbuntu20.04 虚拟机上使用 perf 对这两种算法的 CPU 硬件事件进行进一步的分析，结果如表6。

algorithm	cycles	instructions	IPC
unaligned	17251242038	35653436146	2.07
aligned	16120734303	35640950780	2.21

表 6: 对齐与未对齐算法的硬件事件测量

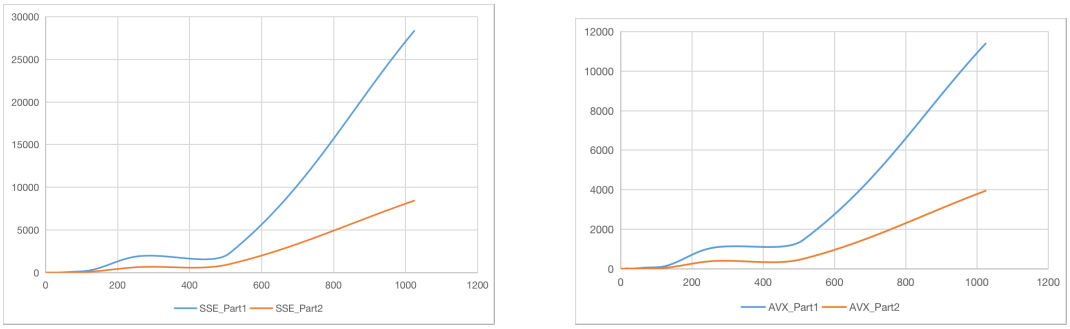
由表6可以看出进行内存对齐的算法的指令数比内存未对齐的算法的指令数要少，说明内存对齐的算法由于不涉及到数据的拼接，因此其所需要的指令数会略低于内存不对齐的算法，这也是其性能要略高于内存未对齐的算法的原因。

4.1.3 对于不同部分的优化效果

在 Gauss 消元的串行算法中，有两个循环可以进行向量化，分别对应算法中的前向消元和消元后的矩阵上三角化。第一个循环对应的是前向消元的过程，第二个循环对应的是上三角化的过程。为了对比两个循环在进行向量化之后哪一个效果更好，我们对于程序耗时进行测量。

N	SSE_Part1	SSE_Part2	N	AVX_Part1	AVX_Part2
2	0.0717	0.1037	**2**	0.0771	0.0625
4	0.3112	0.3569	**4**	0.4651	0.2041
8	5.1585	6.0775	**8**	1.3641	1.2719
16	13.1319	10.0729	**16**	11.8828	10.9433
32	12.2995	6.4689	**32**	9.6656	5.9
64	68.065	35.0072	**64**	50.1497	16.5574
128	306.587	115.606	**128**	147.961	52.7639
256	1908.39	640.434	**256**	1053.52	384.925
512	2303.6	1002.81	**512**	1454.98	499.838
1024	28333.8	8433.29	**1024**	11391.1	3943.42

表 7: SSE 与 AVX 算法的两部分分别进行优化的耗时



(a) SSE 两部分循环的优化效果 (b) AVX 两部分循环的优化效果

图 4.5: 优化效果对比趋势图

由表7和图4.6中可以知道，第二个循环消元后的矩阵上三角化的优化效果更好。推断是第二个循环需要对矩阵的多行元素进行类似向量内积的操作，因此可以进行较好的向量化。而对于第一个循环，

虽然也可以将每行的元素并行处理，但是由于每行的元素数相对较少，因此向量化的优化效果可能会有限。此外，第一个循环的计算量通常也比较小，因此其在整个算法中的优化效果相对较小。

为证明该推断，我们在 Unbuntu20.04 虚拟机上使用 perf 对这几种算法的 CPU 硬件事件进行进一步的分析，其具体数据如下表8所示。

Part	cycles	instructions	IPC
1	26146252978	94184979403	3.6
2	17253512432	35762182763	2.07

表 8: SSE 两部分循环的硬件事件采样

可以看出仅对第二个循环进行 SSE 优化让指令数明显降低，这是因为向量化技术可以将多个浮点数的操作并行化，从而减少循环次数和指令数。我们使用 SSE 指令集对每个浮点数进行 128 位的并行操作，那么在处理每一行的元素时，可以将每四个浮点数看做一个向量，并一次性处理四个浮点数。这样，处理每一行的指令数就可以减少到原来的四分之一。因此通过使用向量化技术，可以在不改变算法正确性的前提下，有效地减少指令数和计算时间，从而提高算法的执行效率。

4.2 ARM 平台

4.2.1 向量化的实现

在 ARM 平台上主要尝试 Neon 并行指令集架构，编写了 Gauss 消去的串行算法以及 Neon 指令集内存对齐与未对齐的算法，并且也使用不同的编译选项 (优化序列) 进行了编译优化。数据如下表9所示。

	N	normal	Neon_unaligned	Neon_aligned
未优化	**8**	1.311	1.5	1.335
	32	8.103	6.089	5.62
	64	64.583	44.301	40.97
	128	252.722	167.003	153.747
	256	2012.17	1304.85	1208.29
	512	3089.94	1931.12	1765.15
	1024	24635.3	15382.9	14006.8
O2 优化	**N**	**normal**	**Neon_unaligned**	**Neon_aligned**
	8	0.242	0.327	0.213
	32	1.306	0.678	0.513
	64	9.814	3.599	2.991
	128	37.759	12.278	9.822
	256	301.843	95.084	71.846
	512	509.62	160.107	139.849
	1024	4240.49	1324.12	1255.22
O3 优化	**N**	**normal**	**Neon_unaligned**	**Neon_aligned**
	8	0.197	0.153	0.189
	32	0.457	0.484	0.476
	64	3.141	2.73	2.936
	128	11.521	10.291	9.567
	256	98.74	89.905	71.193
	512	197.66	185.147	159.373
	1024	1639.64	1509.65	1388.64

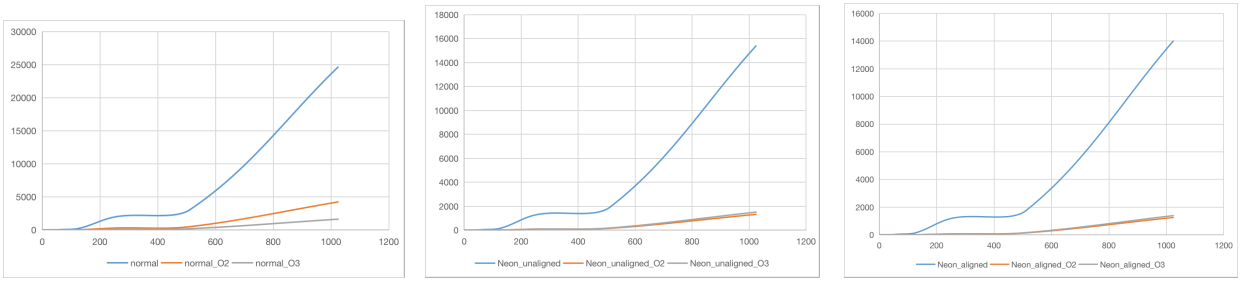
表 9: Neon 优化的多种测量数据

根据表9可看出，使用 Neon 指令集进行编程可以提高运算速度，Neon 指令集支持一次性处理多

个数据，因此在使用 Neon 指令集进行优化的高斯消元中，可以同时计算多个元素，从而提高运算速度。同时对比对齐与未对齐的算法，与在 x86 上的实验一样，在数据规模较小时差距不大，在数据规模较大时进行内存对齐优化之后，程序耗时得到一定程度的缩短，更加证明我们的猜想是正确的。

4.2.2 编译器的不同优化力度

在本次实验中我在华为鲲鹏服务器上对于 normal 算法和使用 Neon 指令集的算法进行了不同编译器优化的对比实验，如表9，我们对此进行探究。

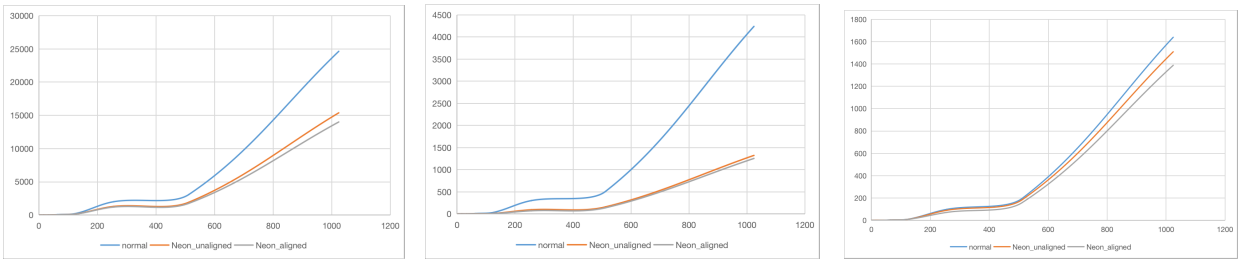


(a) normal 算法 (b) 未对齐的 Neon 算法 (c) 对齐的 Neon 算法

图 4.6: 不同编译器 (无编译器、-O2、-O3) 下算法耗时

绘出图像进行分析，如图4.6。可以清楚地看到，在进行编译器优化 (-O2、-O3) 后，每种算法的耗时都大幅度缩短，但是 normal 算法的-O2、-O3 优化有一定差距，但是在进行 Neon 优化之后-O2、-O3 编译器优化相差较小。

经过资料查询，得知-O2 优化序列会牺牲部分编译速度以进行更多的优化，是比-O1 更高级的编译优化序列.-O2 将执行几乎所有的不包含时间和空间折中的优化。与-O1 比较而言,-O2 以更多编译时间为代价，提高了生成代码的执行效率。-O2 主要应用一些比较基础的优化技术如函数内联、循环展开、常量传播、变量寄存器分配，以减少目标代码的执行时间和空间占用，适合对执行时间敏感的应用，例如实时系统、嵌入式系统等。-O3 优化序列表示编译器应用更加激进的优化技术，以进一步减少目标代码的执行时间和空间占用。这些优化技术包括指令调度、代码移动、数据流分析、循环变量优化等。优化级别较高，适合对执行时间极度敏感的应用，例如高性能计算、科学计算等。由于在进行 Neon 优化之后指令数已经得到优化，所以-O2、-O3 的优化效果相差不大。[1]



(a) 未进行优化优化 (b) -O2 优化 (c) -O3 优化

图 4.7: 同一编译器下算法耗时

然后我们对于同一编译选项对于不同算法的优化效果进行对比。如图4.7。可以看到在未进行编译器优化时三种算法的性能相差很大，但是在进行-O2 优化后，对齐与未对齐的算法的性能几乎没有差距，在进行-O3 优化后，三种算法的性能都很相近。猜测-O3 的编译选项应该对 normal 算法进行了效

果较好的优化。为了验证我们的猜想所以我们使用 perf 对于不同编译器优化下的不同算法的指令数进行监测。其结果如下表10。

	normal	Neon_unaligned	Neon_aligned
O2	33974195869	709792632	4849537014
O3	7844758360	7045640022	5384658368

表 10: 不同编译器优化的硬件事件测量

由表10中数据我们可以看出，在进行-O3 优化之后，normal 算法的指令数相较于-O2 优化后缩减很多，与进行 Neon 优化过的算法相差很小，说明-O3 编译器优化效果主要为对原本的数据自动向量化。Gauss 消元中的矩阵运算包括矩阵相加、矩阵相减、矩阵乘法和矩阵消元等，这些操作可以被编译器自动向量化，以提高程序的执行效率。编译器会将多个标量操作转换为单个向量操作，以减少指令的执行次数。一般来说，编译器会根据程序的特性和编译选项来决定是否对代码进行向量化，例如优化级别、目标机器的指令集支持情况等。在 Gauss 消元中，由于程序的矩阵操作特别是化为上三角矩阵具有一定的规律和重复性，编译器会对此自动进行向量化，从而提高程序的执行效率。

在进行了-O3 优化后，编译器可能会应用更多的高级优化技术，例如代码移动、指令调度、循环变量优化等，从而使得目标代码更为紧凑、高效。因此，normal 算法在进行-O3 优化后指令数缩减很多，这也说明了-O3 优化可以获得更好的性能提升效果。

5 总结反思

5.1 实验总结

在本次 SIMD 指令集的并行实验中，我选择对 Gauss 消元算法进行并行化处理，主要针对算法的两部分进行向量化处理，即算法中的前向消元和消元后的矩阵上三角化。。在 x86 上，我们使用了 SSE，AVX，AVX-512 进行 SIMD 并行化处理，在 ARM 上使用了 NEON 指令集进行优化。通过对比几种指令集的优化效果，我们发现 AVX-512 相比其他指令集具有更好的加速效果，与理论上的分析吻合。同时，我们使用 perf 性能分析工具在 Ubuntu 环境下对其他的可能影响到优化效果和程序性能的操作进行分析。我们先对内存对齐与不对齐做了进一步研究，并且对比了不同指令集下内存对齐与内存不对齐算法的性能表现。通过使用 perf 和 VTune 等性能分析工具，进一步分析并行算法能够取得性能提升的深层原因，也能够对比发现内存对齐算法和内存不对齐算法之间产生性能差异的原因。然后对于可以优化的两部分进行对比研究。以 SSE 为例，我们对发现优化矩阵上三角化部分的加速效果明显好于前面的部分。在 ARM 上，我们对不同编译选项的优化效果进行了研究，并发现在进行不同操作之后编译器的优化效果不同，进一步了解了编译器优化的操作。在本次实验中让我在对于在多个实验平台和环境上的实验更加熟悉，使用性能分析工具熟练。本次实验的相关代码和文档已经上传刘大圆的 Github。

5.2 存在问题

在实验测量中发现在数据规模较小的时候，n=32 的程序耗时总是比 n=16 的程序耗时要短，一开始以为是数据规模太小的偶然性，但是经过查阅资料并未找到可以解释的原因，因此未对该问题进行深入研究。

参考文献

[1] 高国军, 任志磊, 张静宣, 李晓晨, and 江贺. 编译优化序列选择研究进展. 中国科学: 信息科学, 2019.