



南開大學

Nankai University

计算机学院
并行程序设计实验报告

并行期末大作业

姓名：张天歌

学号：2211123

专业：计算机科学与技术

2024 年 7 月 2 日

目录

1 问题重述	3
1.1 普通高斯消元	3
1.2 特殊高斯消元	4
1.3 研究历史和现状	4
1.4 正确性验证说明	5
1.5 实验用时测量说明	5
2 实验平台以及编译选项	5
2.1 实验平台	5
2.2 编译选项	6
2.2.1 ARM 平台上编译选项	6
2.2.2 Code Blocks 上进行 SIMD 和 pthreadOpenMP 实验	6
3 普通高斯消元的并行优化	6
3.1 SIMD 优化探究	6
3.1.1 实验设计	7
3.1.2 ARM 平台——NEON 指令集	7
3.1.3 X86 平台——SSE、AVX、AVX512 指令集	10
3.1.4 SIMD 其他优化尝试	11
3.2 Pthread 优化探究	12
3.2.1 实验设计	12
3.2.2 划分方式和同步机制	12
3.2.3 ARM 平台与 x86 平台对比	15
3.3 OpenMP 优化探究	18
3.3.1 负载均衡任务划分方式	18
3.3.2 尝试改变 chunksize	19
3.3.3 改变线程数目	20
3.3.4 OpenMP 与 SIMD 结合	21
3.4 Pthread 和 OpenMP 对比和其他优化	21
3.4.1 Pthread 和 OpenMP 性能差异	21
3.4.2 cache 优化和按列划分	22
3.5 MPI 优化探究	23
3.5.1 实验设计	23
3.5.2 不同进程数的探究	24
3.5.3 数据划分方式的探究	25
3.5.4 x86 平台与 ARM 平台对比	26
3.5.5 非阻塞通信 MPI 的探索:	28
3.5.6 MPI 流水线形式的改进	29
3.5.7 MPI、多线程、SIMD 的结合	30
3.6 CPU 优化的最终加速方法	31
3.7 GPU 计算的优化探究	32

3.7.1	CUDA 编程优化	32
3.7.2	OneAPI 平台上的 GPU 加速尝试	34
4	特殊高斯消元的并行化	35
4.1	SIMD 优化	35
4.1.1	X86 平台——SSE、AVX、AVX512 指令集	35
4.1.2	实验结果	35
4.1.3	优化编译选项	36
4.2	openmp 和 pthread 优化	37
4.2.1	pthread 优化思路	37
4.2.2	OpenMP 优化思路	37
4.2.3	实验结果	37
4.3	MPI 优化	39
4.3.1	算法设计	39
4.3.2	实验结果	39
5	总结与反思	40
5.1	期末新的工作	40
5.2	最终成果总结	41
5.3	反思	41

Abstract

1 问题重述

1.1 普通高斯消元

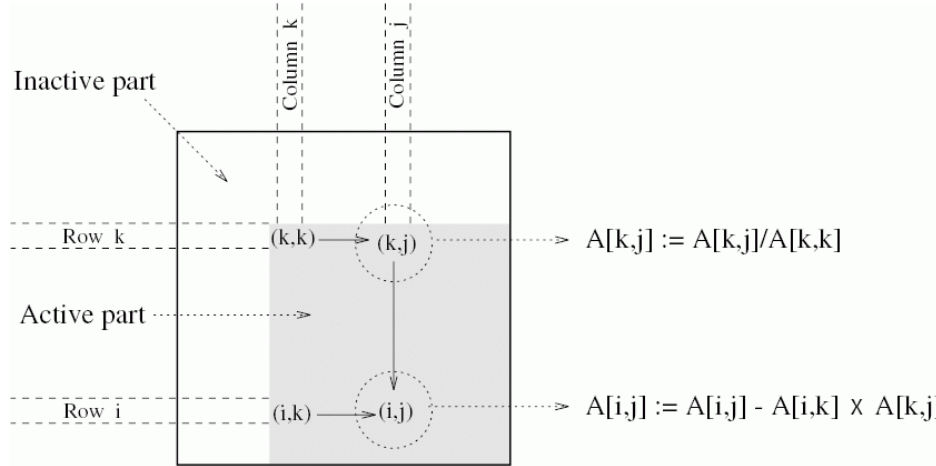


图 1.1: 高斯消去法示意图

高斯消去的计算模式如图1.1所示，主要分为消去过程和回代过程。在消去过程中进行第 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。串行代码的算法如下面伪代码（算法 1）所示：

Algorithm 1 Gaussian Elimination 普通高斯消元

```

1: procedure GAUSSIAN_ELIMINATION( $A, B$ )
2:    $n \leftarrow \text{size}(A)$  ▷ 消元过程
3:   for  $k$  from 1 to  $n$  do
4:     for  $i$  from  $k + 1$  to  $n$  do
5:        $\text{factor} \leftarrow \frac{A[i, k]}{A[k, k]}$ 
6:       for  $j$  from  $k + 1$  to  $n$  do
7:          $A[i, j] \leftarrow A[i, j] - \text{factor} \times A[k, j]$ 
8:       end for
9:        $B[i] \leftarrow B[i] - \text{factor} \times B[k]$ 
10:    end for
11:  end for
12: end procedure

```

在分析算法性能时，我们发现该算法包含了三个嵌套的线性时间复杂度循环，导致整体的时间复杂度提升至 $O(n^3)$ 。串行算法的每次计算步骤都需等待前一步骤完成，这限制了算法的执行速度，并且没有充分利用现代多核处理器的并行处理能力。对于处理大规模矩阵，串行算法不仅需要大量存储空间来保存中间结果，而且在计算过程中频繁访问存储器，进一步降低了效率。

通过深入研究高斯消去法，我们发现该算法的两个关键步骤——即第一部分除法和第二部分减法——具有并行化的潜力。具体来说：

1. 第一个循环中的除法步骤，即 $A_{kj} = A_{kj} / A_{kk}$ ，可以同时为多个元素执行除法操作，实现并行化。

2. 第二个嵌套循环中的减法步骤，即 $A_{ij} = A_{ij} - A_{ik} \times A_{kj}$ ，可以同时多个向量进行加减消去操作，同样可以实现并行化。

本学期的研究工作将重点放在这两个步骤的并行优化上，以期提高算法的执行效率。

1.2 特殊高斯消元

特殊高斯消元算法在有限域 GF(2) 上进行，主要涉及两个操作：消元子和被消元行的操作。该算法的流程是：

1. 将文件中的消元子和被消元行读入内存。
2. 对每个被消元行，检查其首项，如有对应消元子，则将其减去（异或）对应消元子，重复此过程直至其变为空行（全 0 向量）或首项在范围内但无对应消元子或该行。
3. 如果某个被消元行变为空行，则将其丢弃，不再参与后续消去计算；如其首项被覆盖，但没有对应消元子，则将它“升格”为消元子，在后续消去计算中将以消元子身份而不再以被消元行的身份参与。
4. 重复第二步和第三步的过程，直至所有被消元行都处理完毕。

其串行算法的代码如下所示，其中 $R[i]$ 代表首项为 i 的消元子， $E[i]$ 代表第 i 个被消元行， $lp(E[i])$ 代表被消元行第 i 行的首项。

Algorithm 2 串行特殊高斯消元算法

```

1: for  $i$  from 0 to  $m - 1$  do
2:   while  $E[i] \neq 0$  do
3:      $x \leftarrow lp(E[i])$ 
4:     if  $R[x] \neq \text{NULL}$  then
5:        $E[i] \leftarrow E[i] - R[x]$ 
6:     else
7:        $R[x] \leftarrow E[i]$ 
8:     end if
9:     break
10:  end while
11: end for
12: return  $E$ 

```

对该算法的并行化优势在于能够同时对多个数据点执行消元操作，这可以通过 SIMD 技术来实现。但是，我们必须注意保持消元操作的顺序性，以确保在将某一行提升为消元子之后，所有依赖于该消元子的后续消元行都能正确地进行消元。

1.3 研究历史和现状

高斯消元法最早由德国数学家 Carl Friedrich Gauss 提出，但其思想在中国古代数学著作《九章算术》中已有体现。该方法通过一系列初等行变换，将线性方程组转化为上三角方程组，再通过回代求解。随着计算机的出现，高斯消元法的计算效率得到了显著提升，尤其是在并行计算技术的帮助下，对于大规模线性方程组的求解变得更加高效。

研究者们对高斯消元法进行了多种改进和优化，以适应不同的计算环境和提高算法性能。例如，部分选主元和全选主元的高斯消元法被提出以改善数值稳定性；稀疏矩阵的高斯消元法针对稀疏特性进

行了特殊处理，减少了计算量；结合迭代法的高斯消元法则旨在提高对大规模问题的求解效率。此外，随着 GPU 等并行计算资源的发展，基于 GPU 的高斯消元法实现了对大规模线性方程组的快速求解。

1.4 正确性验证说明

在进行高斯消元算法的并行化程序开发时，确保结果的准确性是至关重要的。由于高斯消元过程中的数据具有强烈的时序依赖性，验证最终输出矩阵的正确性可以为我们提供算法正确执行的强有力证据。在实际操作中，用具有确定性特征的矩阵，如对角矩阵、上三角矩阵或下三角矩阵，验证算法的正确性；如果测试程序输出正确，此后重复测试过程中，仅比对输出矩阵各元素均值，若无误，则认为并行化程序正确性得到检验，数据可以采信。

1.5 实验用时测量说明

在时间性能的测量上,x86 平台与 ARM 平台分别选用高精度计时函数 QueryPerformanceCounter() 与 time.h 来计时，并通过重复多次取平均值的策略计算单次程序/函数运行时间。且由于精确计时函数开销巨大，实验中统一在循环体外进行计时，忽略循环条件判断等微小开销。

2 实验平台以及编译选项

2.1 实验平台

本次实验中，ARM 基于华为鲲鹏云平台进行，x86 平台中的实验基于 Code::Block 进行。平台本机处理器信息如下图2.2所示：


Processor					
Name	Intel Core i5 12500H				
Code Name	Alder Lake	Max TDP	45.0 W		
Package	Socket 1744 FCBGA				
Technology	10 nm	Core VID	1.273 V		
Specification	12th Gen Intel®Core™i5-12500H				
Family	6	Model	A	Stepping	3
Ext. Family	6	Ext. Model	9A	Revision	L0
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, FMA3, SHA				
Clocks (P-Core #0)					
Core Speed	4489.02 MHz				
Multiplier	x 45.0 (4.0 - 31.0)				
Bus Speed	99.76 MHz				
Rated FSB					
Cache					
L1 Data	4 x 48 KB + 8 x 32 KB				
L1 Inst.	4 x 32 KB + 8 x 64 KB				
Level 2	4 x 1.25 MB + 2 x 2 MB				
Level 3	18 MBytes				
Selection					
	Socket #1		Cores		4P + 8E
			Threads		16

图 2.2: x86 处理器信息

选项	信息
内核版本	Linux master 4.14.0-115.el7a.0.1.aarch64
发行版版本	OpenEuler(CentOS)
编译器	HUAWEI 毕升编译器 (clang 12.0.0)
CPU 型号	鲲鹏 920 服务器版
性能分析工具	Perf
CPU 核心数	96 核 96 线程
CPU 主频	2.6GHz
L1 一级缓存	96 64KB 数据缓存 96 64KB 指令缓存
L2 二级缓存	96 512KB
L3 三级缓存	48MB
内存	20GB

表 1: 鲲鹏服务器的 CPU 相关参数

图 2.3: arm 处理器信息

2.2 编译选项

2.2.1 ARM 平台上编译选项

- clang++ -g -armv8-a 对 cpp 文件进行编译
- pthread 编程时添加 -lpthread
- OpenMP 编程添加 -fopenmp
- MPI 编程使用 mpic++ 进行编译

2.2.2 Code Blocks 上进行 SIMD 和 pthreadOpenMP 实验

- Have g++ follow the C++17 GNU C++ language standard
- Target x86(32bit)
- Optimize even more(for speed) [-O2]
- Intel Pentium Prescott(MMX,SSE,SSE2,SSE3)
- 执行 AVX 指令集时需添加 Other Compilers Options: -march=native
- 执行 OpenMP 时需添加: Other Compiler options: -march = native-fopenmp

3 普通高斯消元的并行优化

3.1 SIMD 优化探究

SIMD (Single Instruction, Multiple Data) 是一种并行计算的技术, 它在一条指令下同时处理多个数据元素。SIMD 的基本思想是将多个相同类型的数据元素打包成一个向量, 然后通过一条指令对整个向量进行操作, 以实现高效的并行计算。算法中可以利用到 SIMD 的地方有:

- 除法部分可以一次性取多个浮点数同时除以首项元素 (即对角线元素);
- 消去部分对某一行进行消去时, 可以一次性取多个浮点数进行消去;

3.1.1 实验设计

基于 SIMD Intrinsics 函数对普通高斯消元进行向量化的伪代码（算法 3），可以逐句翻译为 Neon/SSE/AVX512 高斯消元函数，完成 ARM 和 X86 平台的基本实验，补齐测试样例生成和时间测量辅助工具，分别进行对齐和不对齐的 SIMD 访存操作。SIMD 优化的代码可见 [SIMD 优化代码](#)。

Algorithm 3 SIMD Intrinsic 版本的普通高斯消元

Require: 系数矩阵 $A[n, n]$

Ensure: 上三角矩阵 $A[n, n]$

```

1: for k = 0 to n-1 do
2:   vt ← dupTo4Float(A[k, k])
3:   for j = k + 1; j + 4 ≤ n; j += 4 do
4:     va ← load4FloatFrom(&A[k, j])           ▷ 将四个单精度浮点数从内存加载到向量寄存器
5:     va ← va/vt                                ▷ 这里是向量对位相除
6:     store4FloatTo(&A[k, j], va)             ▷ 将四个单精度浮点数从向量寄存器存储到内存
7:   end for
8:   for j in remaining indices do
9:     A[k, j] ← A[k, j]/A[k, k]                ▷ 该行结尾处有几个元素还未计算
10:  end for
11:  A[k, k] ← 1.0
12:  for i = k + 1 to n-1 do
13:    vaik ← dupToVector4(A[i, k])
14:    for doj = k + 1; j + 4 ≤ n; j += 4
15:      vakj ← load4FloatFrom(&A[k, j])
16:      vaij ← load4FloatFrom(&A[i, j])
17:      vx ← vakj × vaik                        ▷ 这里应该是向量对位相乘
18:      vaij ← vaij - vx
19:      store4FloatTo(&A[i, j], vaij)
20:    end for
21:    for j in remaining indices do
22:      A[i, j] ← A[i, j] - A[k, j] × A[i, k]    ▷ 修正了这里的计算
23:    end for
24:    A[i, k] ← 0
25:  end for
26: end for

```

3.1.2 ARM 平台——NEON 指令集

(1) 非对齐算法：对于规模小的程序，运行时以 1 秒为限进行循环，循环结束后除以循环次数来获取平均值；对于规模大的程序，则采用重复运行 5 次来取平均值。结果如下表1所示：

耗时比的变化揭示了算法性能在特定问题规模下的异常表现（如图3.4），并通过工具分析找到了性能提升和下降的具体原因。

数据规模	串行用时/s	并行用时/s	加速比
100	0.00247	0.00161	1.513
200	0.0194	0.0127	1.532
300	0.0669	0.0425	1.574
400	0.1547	0.1005	1.540
500	0.3069	0.2001	1.534
600	0.5499	0.3494	1.573
700	0.6676	0.5533	1.207
800	1.2539	0.8266	1.517
900	1.7980	1.1682	1.539
1000	2.4794	1.6187	1.532
2000	20.028	12.949	1.547
3000	53.197	46.430	1.146
4000	176.631	121.125	1.458

表 1: NEON 非对齐性能测试结果

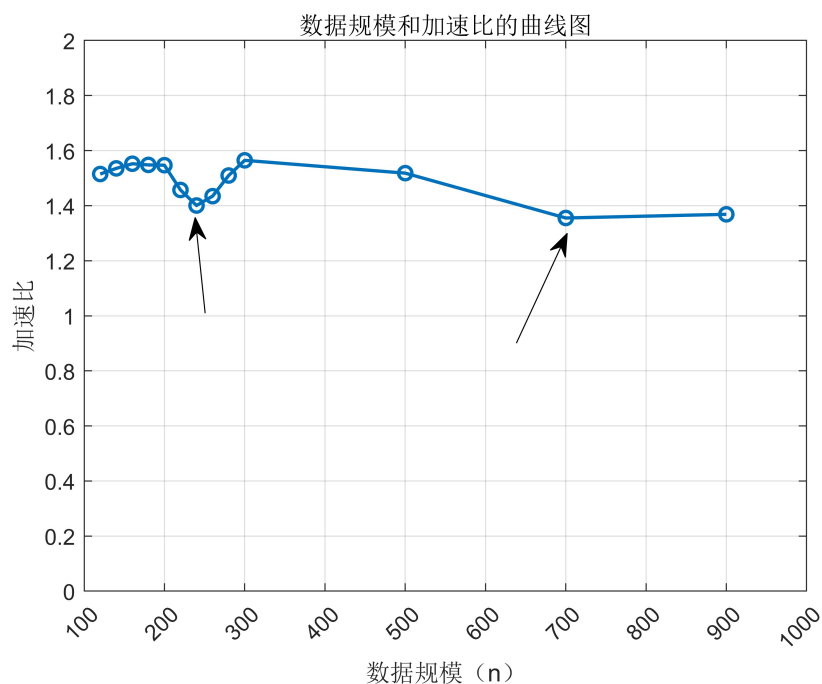


图 3.4: NEON 未对齐并行化

(2) 对齐算法：对齐实验原理：需要对上三角矩阵 $m1$ 进行对齐处理，对每一行的前几个元素进行单独处理，直到找到该行的“主元”（主元是行中第一个非零元素，它在上三角矩阵中尤为重要）。列数倍数对齐指的是继续处理矩阵直到当前处理的元素的列数是 4 的倍数，如果处理到的列数不是 4 的倍数，可能需要对行中的元素进行交换或移动，以确保列数符合对齐要求。原理如下图3.5表示：

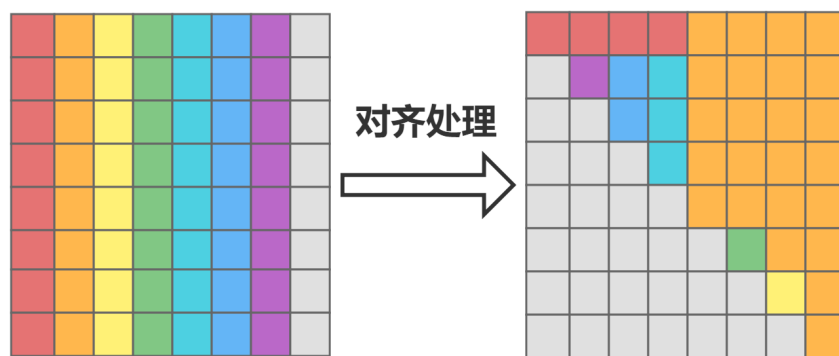


图 3.5: 对代码进行对齐处理

对代码进行修改后，编译和运行的操作同上，结果如下表2和下图3.6所示：

数据规模	串行用时/s	并行用时/s	加速比
100	0.00244	0.00129	1.898
200	0.0194	0.0099	1.947
300	0.0654	0.0333	1.967
400	0.157	0.0797	1.973
500	0.314	0.159	1.969
600	0.547	0.276	1.979
700	0.869	0.441	1.968
800	1.303	0.660	1.972
900	1.852	0.938	1.975
1000	2.530	1.280	1.976
2000	19.839	9.957	1.992
3000	70.74	36.255	1.951
4000	202.753	109.174	1.857

表 2: NEON 对齐性能测试结果

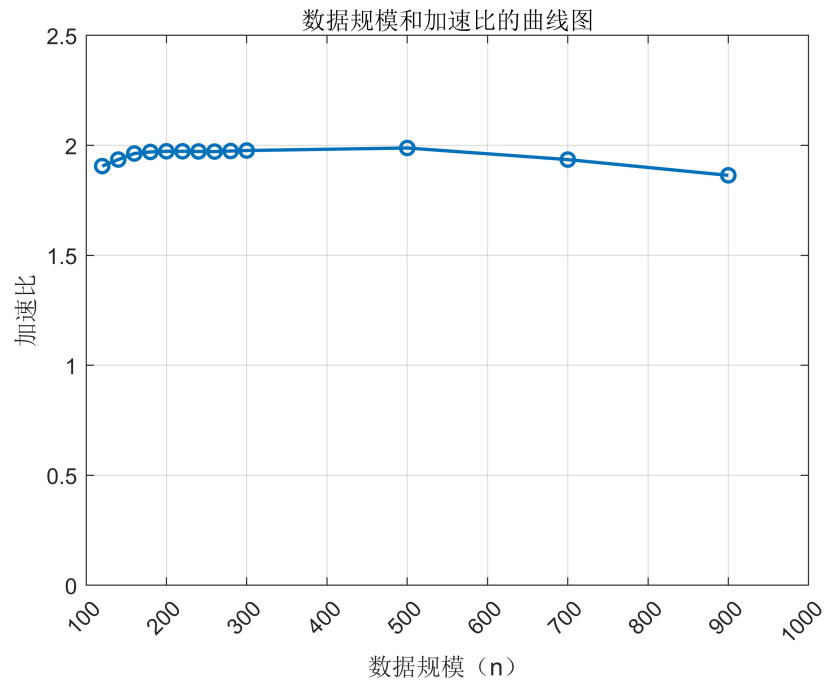


图 3.6: NEON 对齐后并行化

总结：两条曲线走势大致相同，但是发现在相同规模下，**对齐后的加速比相比于未对齐的平均多出 0.4 秒**，观察数据发现串行耗时与未对齐时几乎相同，而并行耗时得到了缩短，因此对齐的作用在于更好的优化了 NEON 指令集的并行化，而未对其他因素有明显影响。

3.1.3 X86 平台——SSE、AVX、AVX512 指令集

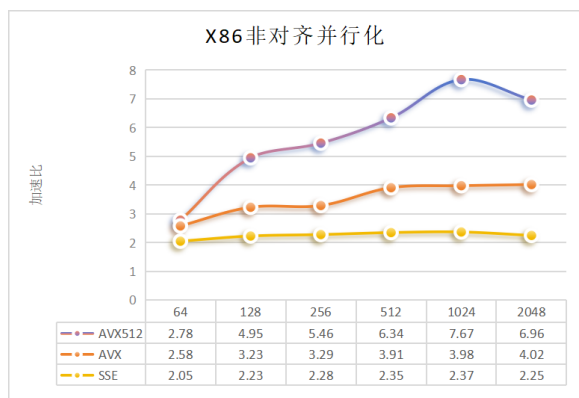
x86 平台的非对齐和对齐实验结果如下所示：

数据规模	串行用时/s	SSE 用时/s	AVX 用时	AVX512 用时/s	串行/SSE	串行/AVX	串行/AVX512
64	0.00040	0.00019	0.000151	0.000156	2.099	2.667	2.579
128	0.00327	0.0014	0.000983	0.000799	2.279	3.327	4.094
256	0.026	0.011	0.00713	0.00476	2.354	3.643	5.459
512	0.206	0.0854	0.0541	0.032	2.416	3.815	6.435
1024	1.676	0.705	0.429	0.237	2.378	3.901	7.064
2048	13.373	5.563	3.364	1.926	2.404	3.976	6.942

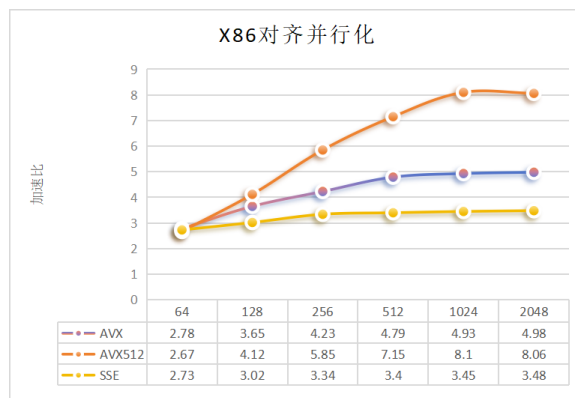
表 3: x86 非对齐性能测试结果

数据规模	串行用时/s	SSE 用时/s	AVX 用时	AVX512 用时/s	串行/SSE	串行/AVX	串行/AVX512
64	0.000401	0.000151	0.000144	0.000157	2.651	2.78	2.55
128	0.00329	0.0011	0.000887	0.000810	2.999	3.747	4.055
256	0.0258	0.00792	0.006	0.00446	3.257	4.3	5.687
512	0.207	0.0625	0.0437	0.0288	3.312	4.73	7.086
1024	1.672	0.501	0.338	0.203	3.337	4.95	8.06
2048	13.213	3.878	2.562	1.744	3.406	4.97	8.02

表 4: x86 对齐性能测试结果



(a) x86 非对齐并行化



(b) x86 对齐并行化

图 3.7: x86 对齐与非对齐对比

数据对齐后，性能提升在各种问题规模下都有所增加，特别是在大规模数据下，优化效果更为显著，SSE 指令集的性能提升约为 0.9 倍，AVX 指令集的性能提升约为 1 倍，AVX512 指令集的性能提升约为 1 倍。数据对齐主要通过优化并行算法的执行来提升性能，而对串行算法的耗时和其他因素如 cache 规模等没有显著影响。

3.1.4 SIMD 其他优化尝试

1. ARM 与 X86 平台对比

综合 SIMD 实验数据，普通高斯消元法在 ARM 平台下和 X86 平台下存在诸多差异，大致有如下几点：

- 指令集差异：X86 平台下的各指令集 AVX, AVX512, SSE 的并行化效果优于 ARM 平台的 NEON 指令集
- 缓存影响：X86 平台下的算法受各级 cache 的影响较为明显，而 ARM 平台下仅受 L3 cache 的影响。
- 对齐 VS 不对齐：对齐后 X86 和 ARM 平台下的并行效果都得到提升，从绝对数值上看 X86 平台下提升更大，从相对比例而言 ARM 平台下提高更多

2. 编译选项的优化

在 ARM 平台下，串行算法的运行速度是 $O_0 < O_1 < O_2 < O_3$ ，并行算法则是 $O_0 < O_1 < O_2 = O_3$ ，这是由于低级的编译参数对串并行算法都有优化，且并行优化更多，而高级编译参数仅对串行算法起优化作用。在 X86 平台下，串行算法的运行速度是 $O_0 < O_1 = O_2 < O_3$ ，并行算法则是 $O_0 < O_1 < O_2 = O_3$ 。总结可以发现在两个平台下优化编译选项都能提升运行速度，且低级优化选项对于并行算法提升更多，高级优化选项对串行算法提升更多。具体耗时比变化可见于下图3.8，其中 arm 平台下为 $n=500$ ，X86 平台下 $n=128$ ，其他规模下的变化与此类似：

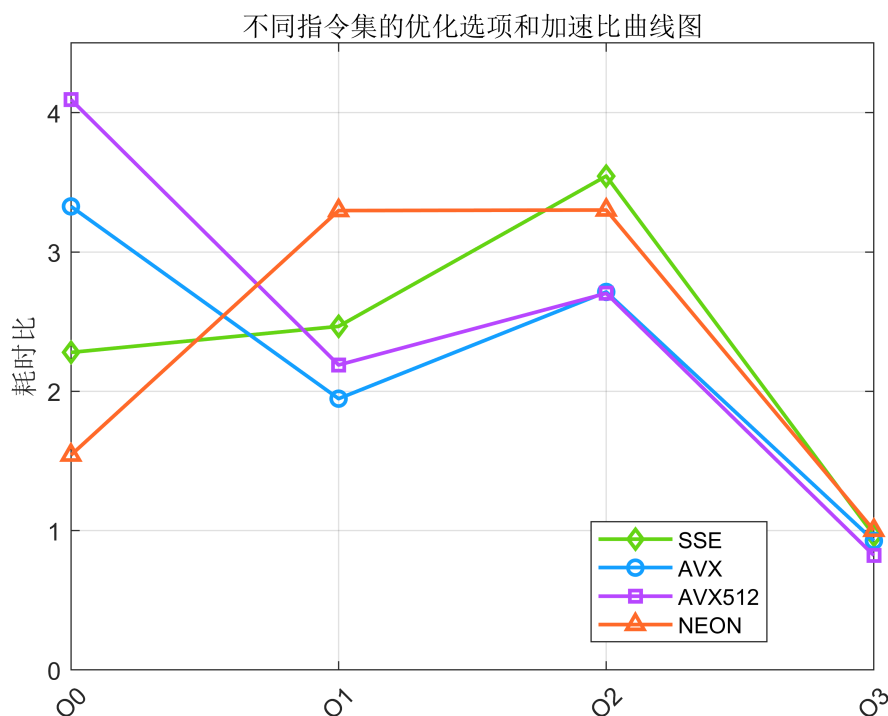


图 3.8: 不同指令集的优化选项和加速比曲线图

3.2 Pthread 优化探究

3.2.1 实验设计

POSIX Threads 简称 Pthreads, POSIX 是"Portable Operating System Interface" (可移植操作系统接口) 的缩写, 在 C++/C 程序中 pthread 能让进程在运行时可以分叉为多个线程执行, 并在程序的某个位置同步和合并, 得到最终结果。

对于两个关键的步骤: 除法 $\text{factor} := A[k, j]/A[k, k]$ 和消去 $A[i, j] := A[i, j] - \text{factor} \times A[k, j]$, 可以考虑使用不同的同步机制 (如信号量和 barrier) 和线程管理策略 (动态线程和静态线程) 并行化, 并结合 SIMD 编程, 讨论并行优化加速比随问题规模和线程数量的变化情况。Pthread 优化代码在这里。

1. 讨论同步机制和线程管理策略时, 固定程序线程数目为 8。实验中分别使用动态分配线程、静态分配线程 + 信号量 `sem_t` 同步、静态 `sem_t` 同步 + 三重循环整合, 以及静态 barrier 同步的 pthread 优化方法。
2. 结合 SIMD 指令集, 分析 x86 的 AVX 和 arm 的 neon 指令集对静态分配线程的性能影响, 实现三重纳入 + neon/AVX、静态 barrier+neon/AVX、静态信号量同步 + neon/AVX 的性能优化对比。
3. 线程数目: 在线程分配中, 线程太少, 会使得加速效果不明显, 而线程太多, 导致管理线程、分配任务的开销又增大, 使得其加速效果下降, 因此找到合适的线程数目极为重要。固定一种线程管理方式和同步机制, 分析 4、6、8、10、12、16 不同线程数对于 pthread 性能的影响。

3.2.2 划分方式和同步机制

1. 动态线程版本

通过每轮消去步骤前动态创建和销毁线程，实现高斯消元的并行计算。

分别采取 500、1000、1500、2000 数据规模，对于规模小的程序，运行时以 1 秒为限进行循环，循环结束后除以循环次数来获取平均值；对于规模大的程序，则采用重复运行 5 次来取平均值。结果如下图所示：

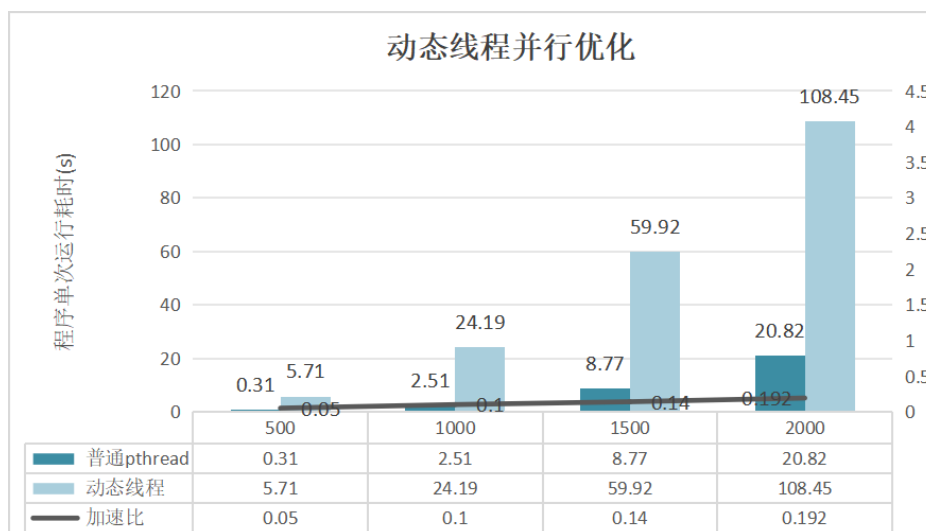


图 3.9: 动态线程版本的优化效果

2. 静态线程 + 信号量同步版本

固定分配 8 个线程，利用静态创建的线程池和sem_t信号量进行同步，以减少线程创建销毁开销，并实现高斯消元的并行处理。

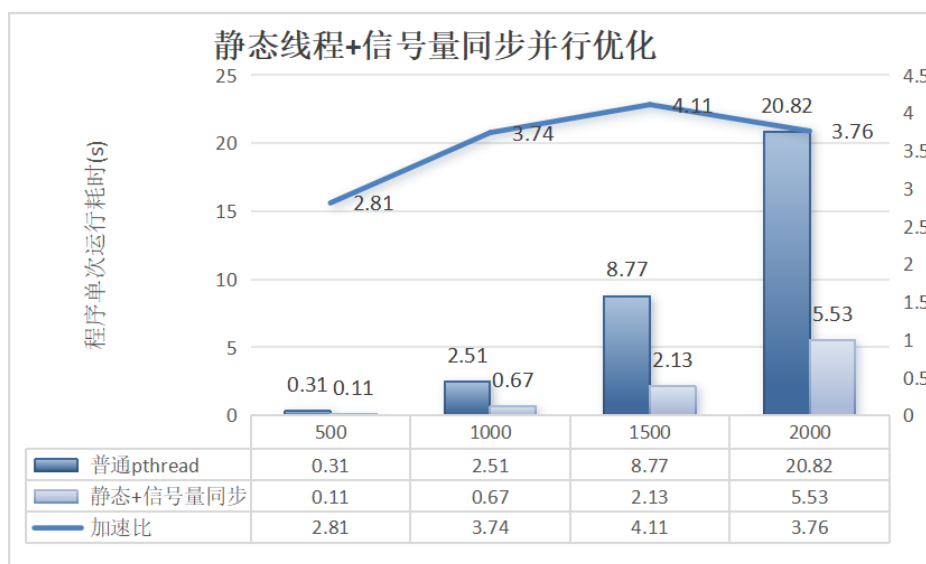


图 3.10: 静态线程 + 信号量同步版本的优化效果

3. 静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数

将三重循环整体抽取出来构成线程函数，并使用静态线程池和信号量同步，全程只有一次线程创建和销毁开销，以提高计算效率和同步灵活性。

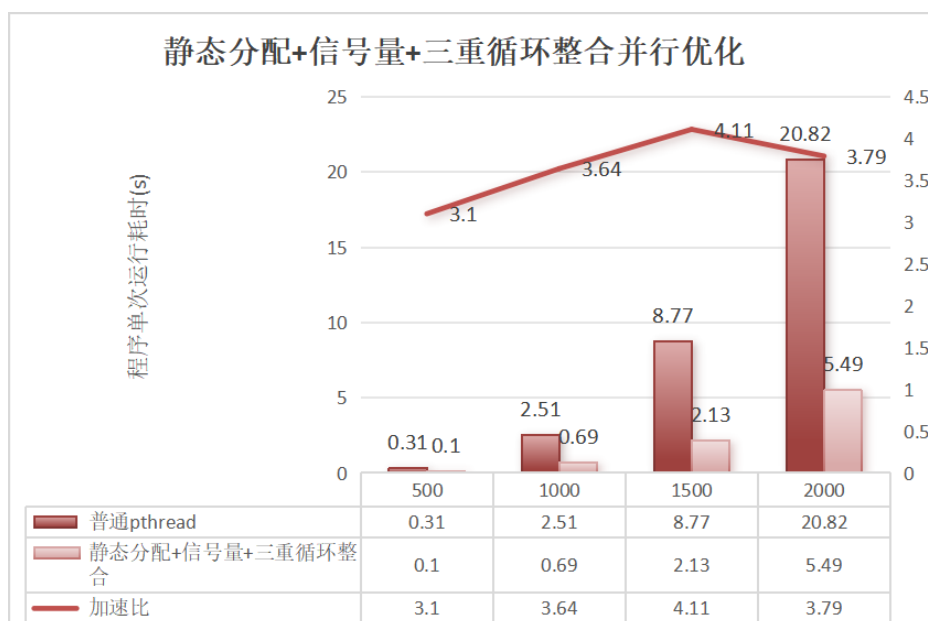


图 3.11: 纳入三重循环的静态线程信号量同步版本的优化效果

4. 静态线程 + barrier 同步

采用静态线程池和 barrier 同步机制简化线程间的协调，实现高斯消元算法的高效并行执行。

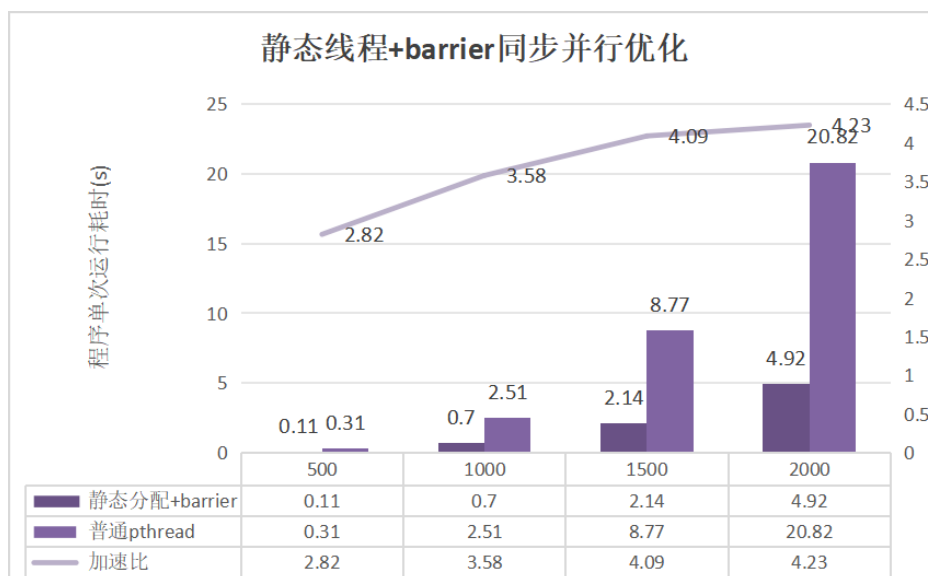


图 3.12: 静态线程 + barrier 同步的优化效果

5. 对比总结

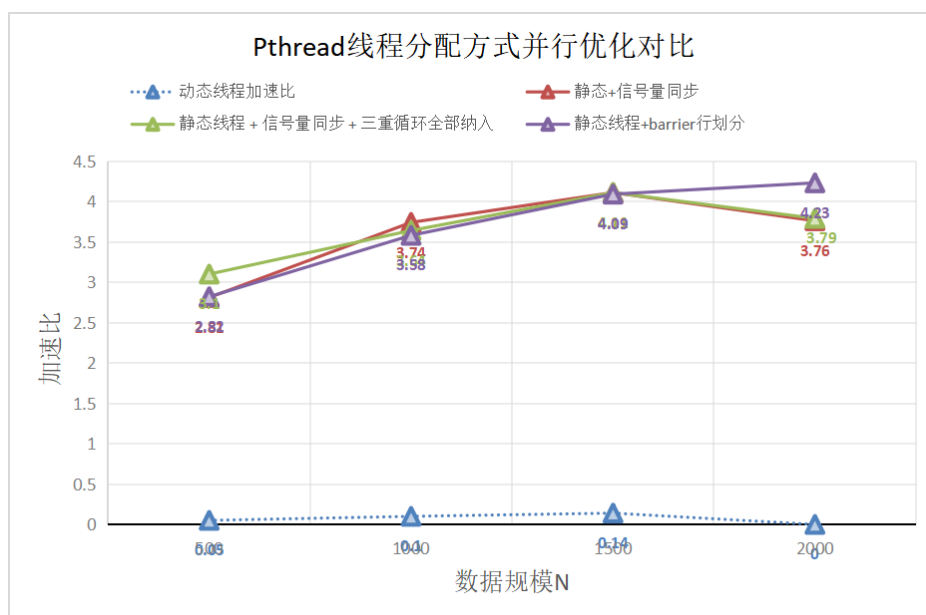


图 3.13: Pthread 线程管理对比

- 负载不均衡：线程之间工作负载分配不均，导致一些线程空闲等待；指导书中说明 CPU 核心数有限，过多的线程数可能会因为上下文切换和资源竞争而导致效率降低。
- 加速比：所有方法的加速比都随着数据规模的增加而提高，但增长速度和最终值各有不同。三重循环纳入 + 静态信号量同步并行优化在大部分数据规模下显示出最高的加速比。
- 线程管理：动态线程方法可能会因为线程创建和销毁的开销而在小规模数据上表现不佳。静态线程方法通过减少这些开销提高了效率。
- 同步机制：使用屏障和信号量作为同步机制的方法在大规模数据上显示出较好的性能。

3.2.3 ARM 平台与 x86 平台对比

1. 分配方式 + 同步机制

(1) x86 运行结果：

在小数据规模下，动态线程可能因为线程管理开销较大而性能不佳；中等数据规模下，静态信号量方案开始展现出较好的性能，但可能仍受同步开销的限制。大规模数据规模下，静态屏障方案和静态信号量 + 三重循环方案表现最佳，因为它们能够有效地管理线程同步和数据局部性。

(2) x86 和 ARM 对比图：

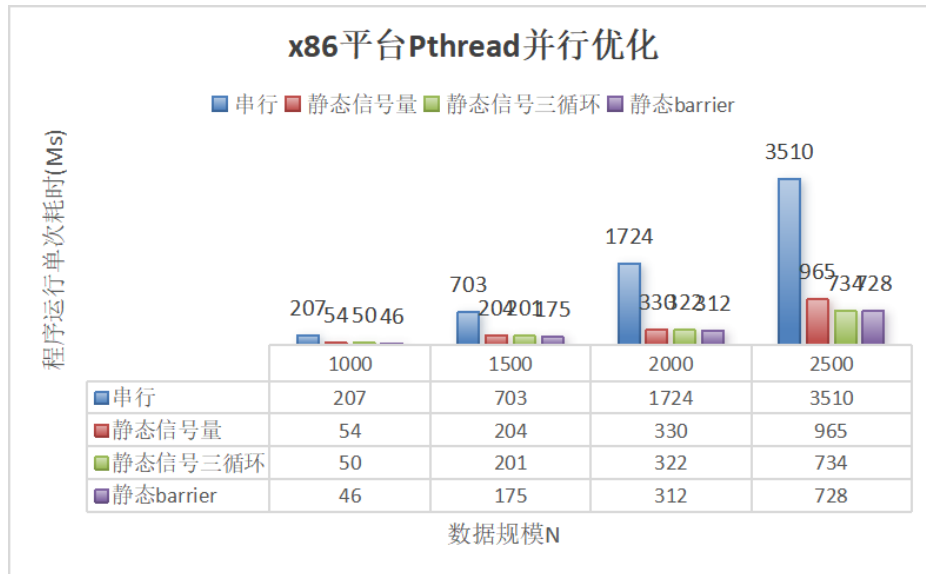


图 3.14: x86 平台 Pthread 并行优化

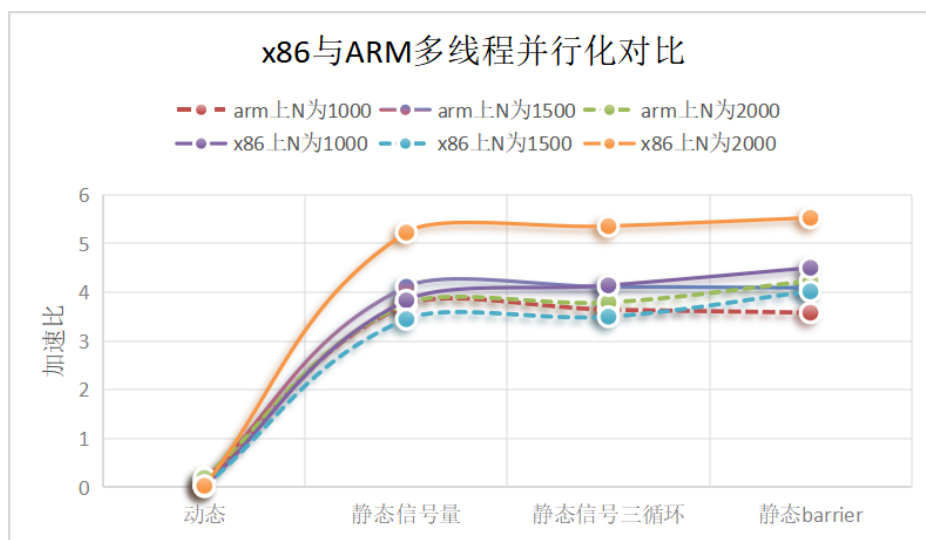


图 3.15: x86 与 ARM 多线程并行化对比

用 perf 对 x86 程序进行 profiling, 详细数据如表5:

	L1-loads	L1-misses	miss 的概率	L1-load-stores	cache 命中率	CPI
串行	36310296866	639794535	0.017620196	35670502331	0.982379804	0.396296002
动态分配	7822398470	773305938	0.098857907	7049092532	0.901142093	6.285635291
静态信号量	7302221068	716645591	0.098140769	6585575477	0.901859231	6.744104566
静态信号三循环	7269205001	723051561	0.099467763	6546153440	0.900532237	6.928883775
静态 barrier	7215566552	421017005	0.058348433	6794549547	0.941651567	6.791952618

表 5: x86 特殊高斯消元的对齐并行化

结论:

- 线程同步机制的选择：这对性能有显著影响。静态同步机制（如信号量和 barrier）可能会增加线程间的协调开销，但可以减少线程间的竞争。
- 线程管理开销：多线程执行需要额外的线程调度和管理开销，这可能抵消了并行带来的一些好处，cache miss 概率增高，cache hit 概率变低。

2. 线程数目对比

在 x86 平台下，使用静态线程和 barrier 信号同步，在 1000, 2000, 2500 规模下，讨论不同线程数目对 pThread 并行化的影响。

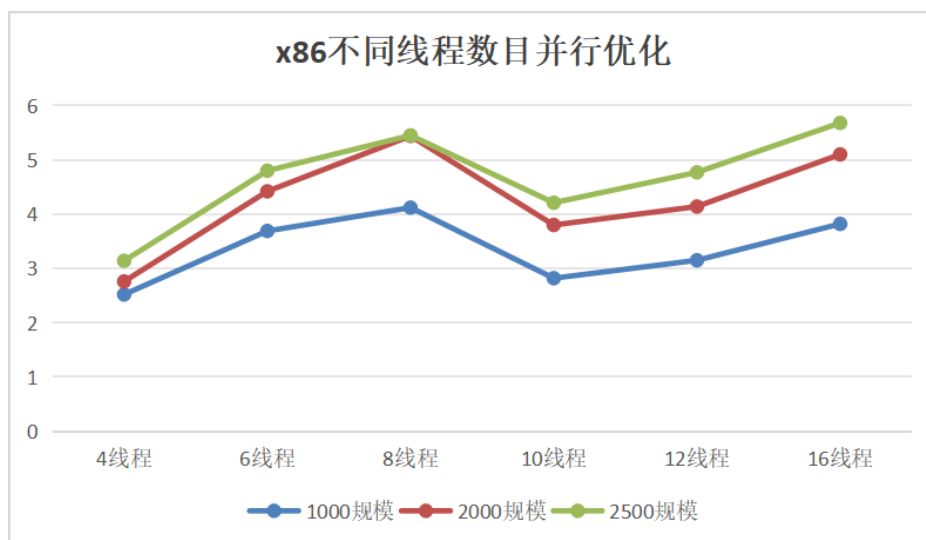
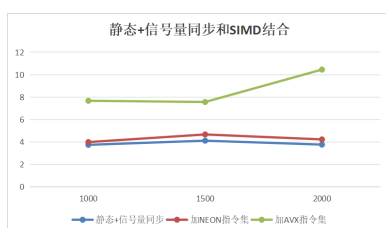


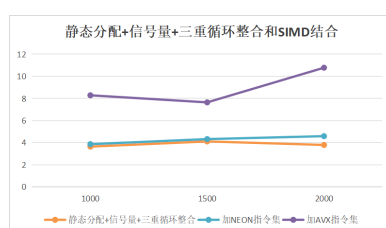
图 3.16: x86 不同线程数目并行优化

根据图3.16所示，在不同平台对线程数目因素的检测，加速比曲线趋势大致相同，又因为八线程具有独特的性能表现优势，故比较 ARM 平台八线程、不同规模的静态 + barrier 同步即可。在 2000 数据规模八线程，ARM 的加速比为 4.23，低于 x86 的 5.42；在 1000 数据规模八线程，ARM 加速比为 3.58，低于 x86 的 4.05。可以得出结论，x86 性能更加优越，更适合做并行优化，程序性能更优。

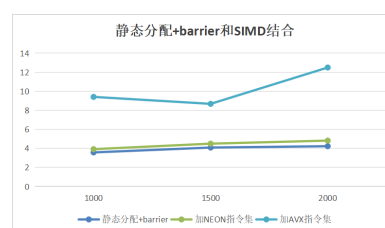
3. 结合 SIMD



(a) 静态 + 信号量同步和 SIMD 结合



(b) 静态分配 + 信号量 + 三重循环整合和 SIMD 结合



(c) 静态分配 + barrier 和 SIMD 结合

图 3.17: 不同并行优化算法的执行时间与准确率对比

将三种划分方式做横向比较，三者最高加速比在 2000 规模达到 10 以上，可以证明集合 SIMD 对 pthread 多线程优化有积极作用。静态分配 + barrier 结合 SIMD 的效果最好，在 2000 规模可达 12.49，比静态 + 信号量同步的 10.45 高，比三重循环整合的 10.77 高。

单独做纵向比较，对比 AVX 指令集的提高一倍加速比，NEON 指令集对程序优化很不明显，证明了之前 SIMD 实验中 NEON 只可达到一点多的优化效果，AVX 性能远高于 NEON。

3.3 OpenMP 优化探究

在 OpenMP 中，编程范式较为一致简单，因此在两个平台可以实现同步对比，将任务划分方式、静态算法优化、不同线程数对比、结合 SIMD 两两比较分析，在单平台内分析算法优化，在多平台比较架构差异。OpenMP 优化思想与 pthread 相同，都是对行消去部分进行并行优化，不同的是 OpenMP 只用几行简单的编译语句即可转换成并行程序，不用像 pthread 一样麻烦地管理线程。

3.3.1 负载均衡任务划分方式

OpenMP 使用简单，比较容易探究多线程并行优化中任务划分方式带来的影响。对于 for 循环的多线程并行，大致有 static (chunksize)、dynamic(chunksize)、guided(chunksize) 三种方式，它们的区别在：

1. `#pragma omp for schedule` 默认采用 static 划分，static 指每个线程划分 chunksize 个任务量，如果 chunksize 没有人为指定，则默认为

$$\frac{N}{\text{NUM_THREADS}}$$

N 是任务总量。

2. dynamic 方式可以动态分配任务量，每当一个工作线程空出来时会为其分配 chunksize 的工作量，chunksize 为 3。这样的方法比较灵活，但是可能会出现各线程之间负载不均衡的情况，甚至会出现有的线程没有参与工作。
3. guided 方式与 dynamic 相近，不过每次分配的任务量不同，开始比较大，之后逐渐减小至 chunksize 不再减小，chunksize 为 3。这种方式一开始多分配一些任务量，避免了多次的任务划分，也是一种很好的划分方式。

OpenMP 优化代码仓库在这里，实验结果如下：

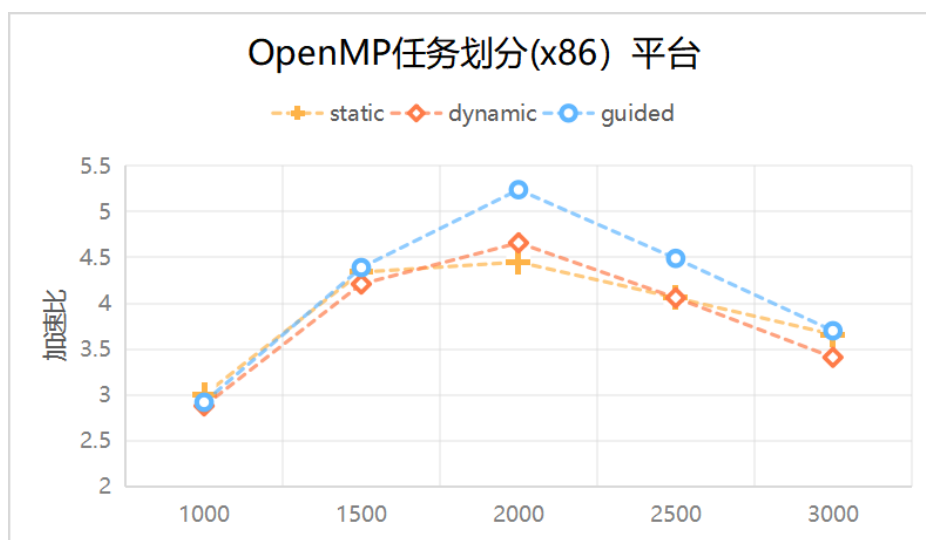


图 3.18: OpenMP 任务分配 (x86) 平台

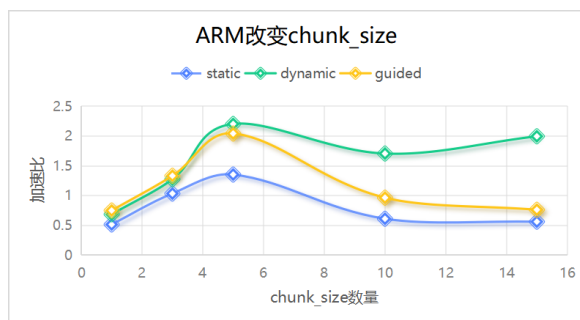
- dynamic 方式的优化效果在三种方式中表现较差，分析是因为 dynamic 方式需要不断分配空闲线程，导致效果不佳，甚至可能会出现有的线程没有参与工作，或线程任务量分配不均的情况，不能较好的利用线程资源。
- guided 划分方式表现最好，在所有数据规模中，guided 总是达到最高的加速比。这与 guided 划分方式有关，**guided 相比 dynamic 更加灵活**，最初为每个线程分配较多任务量，然后指数下降到 chunksize，能够有效的均衡线程之间的负载。
- 用 perf 测量指令数、周期数，计算 CPI，可以更细粒度地分析出 static 划分是一种更平稳、有效的划分方法。

	指令数	周期数	CPI
static	8826664293	6032198764	0.683406388
dynamic	8798756157	6593784243	0.749399588
guided	8792903322	6224766053	0.707930683

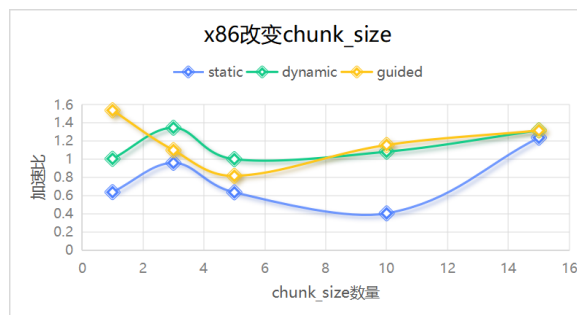
表 6: 对 OpenMP 静态、动态、guided 进行 profiling

3.3.2 尝试改变 chunksize

在任务执行中某个线程结束了先前的任务之后，又会从任务队列中取出新的任务，而并不依赖于线程号与行号的某种映射关系。改变 chunksize 个数分别为：1、3、5、10、15，在二线程规模为 1024 下，观察循环调度的多线程管理是否对负载均衡有帮助，找到合适的 chunksize。



(a) ARM 改变 chunksize



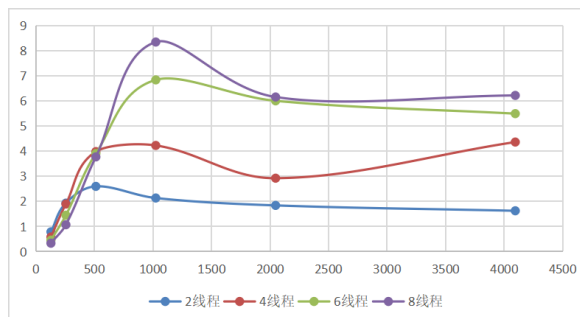
(b) x86 改变 chunksize

图 3.19: arm 与 x86 对比 chunksize 变化

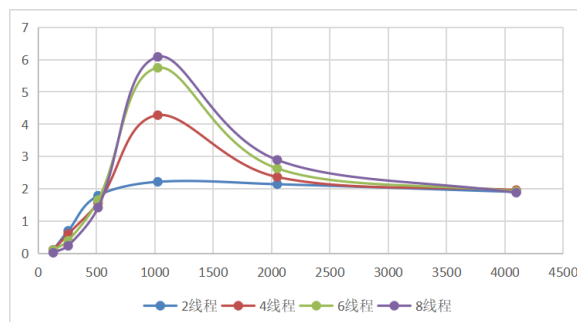
- 较小的 chunksize 可以帮助平衡负载，避免某些线程过早完成它们的迭代而闲置，而其他线程还在处理它们的迭代，例如在 x86，chunksize 为 1 时，guided 分配方式加速比较大。
- 然而，过小的 chunksize 也会增加线程之间的同步开销，x86 平台适合的 chunksize=3，arm 平台适合的 chunksize=5。
- 本次比较在数据规模较小情况下完成，不同划分方式的性能可能与预期结果有出入，误差不大，且只需增大数据规模可看到更高的加速比和明显的曲线走势。

3.3.3 改变线程数目

分别在 x86 和 arm 使用静态划分，在 128、256、512、1024、2048、4096 规模下，讨论不同线程数目对 pThread 并行化的影响。



(a) arm 不同线程数对比



(b) x86 不同线程数对比

图 3.20: arm 与 x86 对比线程数不同

OpenMP 和 Pthread 差不多，根据图3.20所示，观察到随着线程数从单线程增加到八线程，程序的性能呈现出先提升后下降的趋势，八线程性能最好。但当数据规模增加到一定程度后，在较大规模例如 2048 之后，性能增益开始减少，这可能是由于硬件资源（如 CPU 核心数、内存带宽）的限制。下面是平台对比：

- 加速比趋势：ARM 平台在较大规模下加速比趋于稳定，而 x86 平台在较小规模下加速比提升较为明显。

- 最佳线程数：对于 ARM，最佳线程数在 8 线程左右，1000 规模尤其明显。对于 x86，随着线程数的增加，加速比提升较为平稳，但 8 线程时加速比最高。
- 线程扩展性：ARM 平台在较小规模下线程扩展性较好，但在较大规模下加速比提升有限。x86 平台的线程扩展性较为一致，随着线程数增加，加速比稳步提升。

3.3.4 OpenMP 与 SIMD 结合

使用八线程，chunksize=3，在 x86 平台分别实现动态分配 +avx、动态分配 +sse，静态分配 +avx、静态分配 +sse，分析程序耗时和加速比，实现并行化最好效果。

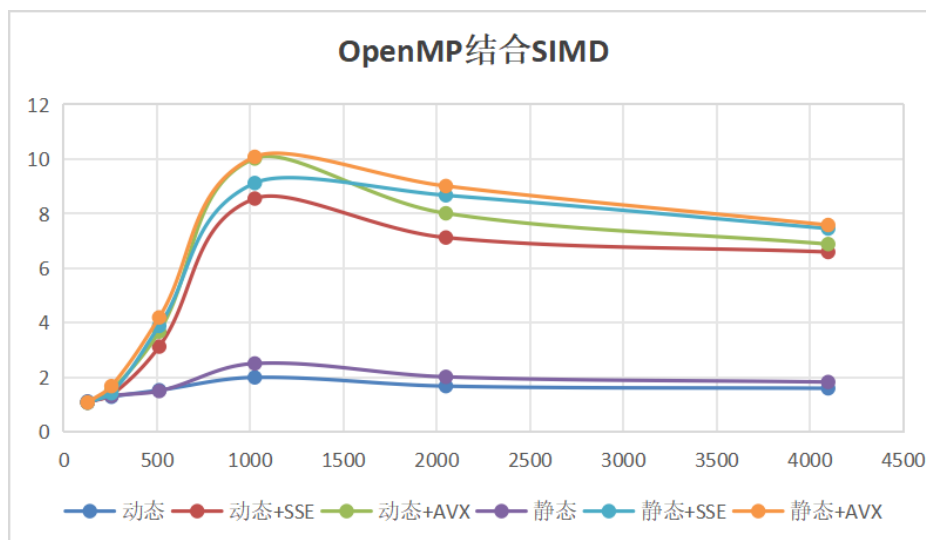


图 3.21: OpenMP 结合 SIMD

与 Pthread 实验同理，AVX 结合效果最好，SSE 次之，可以从 perf 数据中细致观察，静态 +AVX 的 CPI 最高 4.73，结合 SSE 的动态分配和静态分配的 CPI 仅为 1.4 和 1.96。

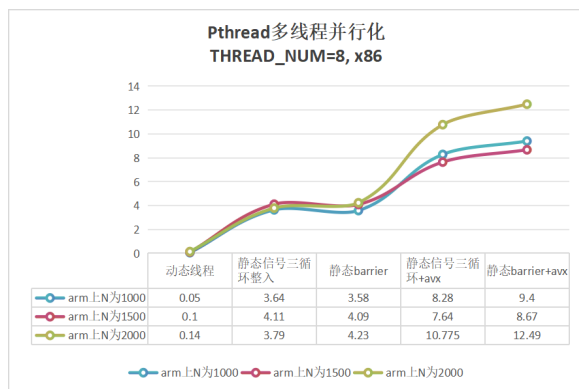
	指令数	周期数	CPI
串行	2973209222	1096552996	0.368811245
动态	8821796942	6373696096	0.722494083
动态 SSE	798060768	1119310306	1.40253769
动态 AVX	444581465	825302684	1.856358731
静态	8760436200	5282370667	0.602980325
静态 SSE	715644849	1399838921	1.956052535
静态 AVX	730587486	3462901448	4.73988607

表 7: OpenMP 结合 SIMD 的 profiling

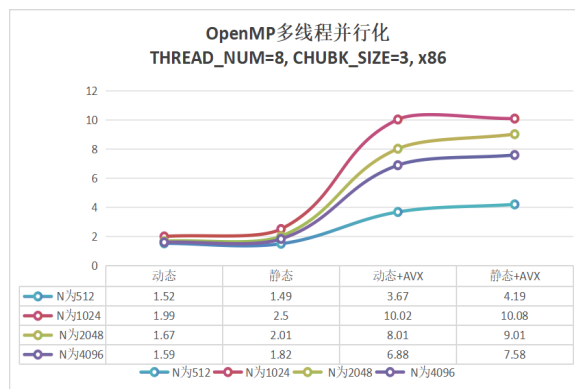
3.4 Pthread 和 OpenMP 对比和其他优化

3.4.1 Pthread 和 OpenMP 性能差异

综合上述 Pthread 和 OpenMP 的实验结果，我们观察整体趋势，并且找到性能最优的拐点。



(a) Pthread 优化



(b) OpenMP 优化

图 3.22: Pthread 与 OpenMP 对比

Pthread 最高性能: N=2000, 结合 avx, 八线程, 静态分配 +barrier 同步, 加速比 12.49

OpenMP 最高性能: N=1024, chunksize=3, 结合 AVX, 八线程, static 分配, 10.08

总体来说, Pthread 性能和 OpenMP 差不多, 在结合 barrier 同步机制达到了 12.49, 两者差异有以下几点:

- 第一, 性能开销: OpenMP 的运行时库针对并行循环和区域进行了优化, 可以减少线程创建和管理开销。pthread 的开销可能会更高, 特别是如果程序员没有有效地管理线程生命周期和同步的话。
- 第二, 可扩展性: OpenMP 在处理大规模并行问题时通常表现良好, 特别是在多核处理器上, 它可以自动地将工作分配给可用的核心。

3.4.2 cache 优化和按列划分

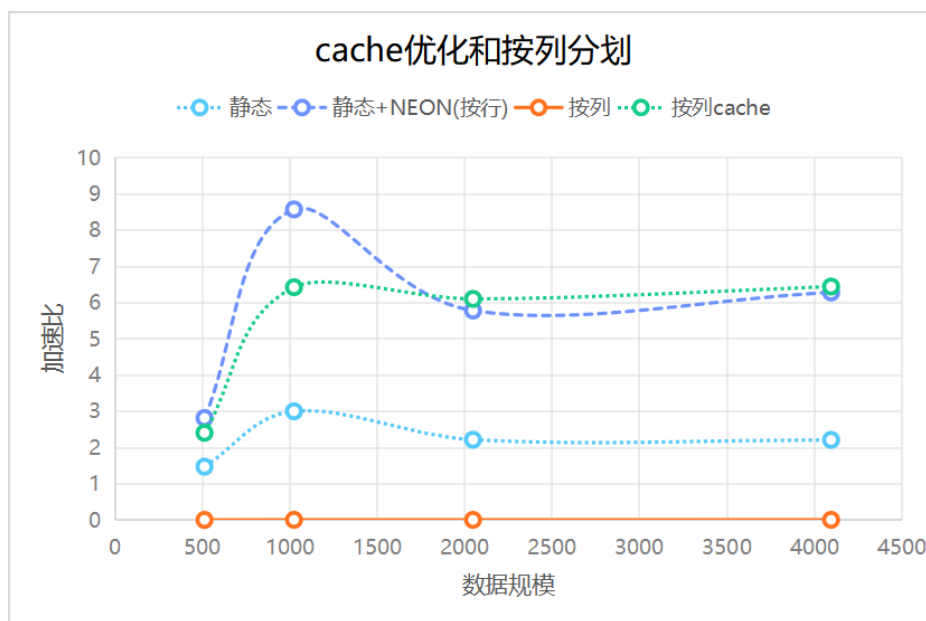


图 3.23: 按列划分和 cache 优化

按列划分减少不同线程间的同步需求和数据竞争,所以对比普通静态分配按行划分和按列划分,可以发现按列是非常低耗时耗时的方法,和普通串行算法相比,加速比接近 0。同时对按列划分进行 cache 优化算法,考虑到了缓存的层次结构和数据访问模式,这可以提高缓存命中率,因为同一列的数据更可能被加载到同一缓存行中,加速比到 6 点多。如图3.23

注意在与 SIMD 结合时, SIMD 只能将行内连续元素的运算打包进行向量化,即只能对最内层循环进行展开、向量化,故按行划分结合 neon 指令集,性能比按列划分 cache 优化要好,最高可接近 9。

3.5 MPI 优化探究

MPI 是一种用于编写并行程序的通信库接口,全称为 Message Passing Interface (消息传递接口),可以理解为是一种独立于语言的信息传递标准,其有着多种具体实现,本次则探究了 ARM 平台上的 MPICH 和 x86 平台上的 MS-MPI。

进程 (process) 是操作系统进行资源分配的最小单元,线程 (thread) 是操作系统进行运算调度的最小单元,进程可以包含多个线程,但每个线程只能属于一个进程。如图2.2所示,每个进程有独立的地址空间 (外部框表示),同一进程内不同线程 (曲线) 共享该进程的内存。

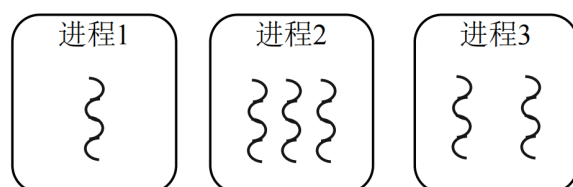


图 3.24: 进程与线程概念

3.5.1 实验设计

算法的主要思想包括以下几个步骤:

- 标准化对角线元素: 在每一步中, 首先将当前行 (第 k 行) 的对角线元素 (A_{kk}) 通过除法操作标准化为 1, 这样每步消元后, 对角线上的元素都是 1。
- 消元过程: 接着, 使用当前行去消去其下方所有行的对应列元素。这是通过将第 k 行乘以一个系数, 然后从下方的行中减去这个结果来实现的。
- 进程间通信: 在 MPI 环境中, 不同的进程可以并行地执行消元步骤。每个进程负责矩阵的一部分。当一个进程完成了其部分的消元后, 它会将结果传递给其他需要这些数据来继续消元的进程。
- 数据传递: 算法中使用了 `MPI_Send` 和 `MPI_Recv` 来在进程之间传递数据。`MPI_Send` 用于发送数据, 而 `MPI_Recv` 用于接收数据。在消元过程中, 如果一个进程需要其他进程的数据来完成消元, 它会使用 `MPI_Recv` 来接收数据。
- 并行处理: 算法通过在多个进程之间分配任务来实现并行处理, 这样可以显著减少总体的计算时间, 尤其是在处理大型矩阵时。

在这个算法中, 每当一个进程将某行向量消元并且对角线元素置 1 之后就会将其数据广播到整个进程域; 这样的方法有一个好处, 就是在最终消元之后不必再次将结果传回给 0 号线程, 在整个算法执行完后即可在 0 号线程中得到正确结果。MPI 优化的代码可见[MPI 优化代码](#)。

Algorithm 4 MPI 优化的普通高斯消元算法

```

1: function GAUSSIAN_ELIMINATION(Matrix A, begin, end, total)
2:   for  $k \leftarrow 1$  to  $n$  do
3:     if  $begin \leq k < end$  then
4:       for  $j \leftarrow k + 1$  to  $n$  do
5:          $A_{nj} \leftarrow \frac{A_{nj}}{A_{rk}}$  ▷ 标准化, 使对角线元素为 1
6:       end for
7:        $A_{nk} \leftarrow 1.0$ 
8:       for  $j \leftarrow 0$  to  $total$  do
9:         MPI_Send(&Ak0, N, MPI, j, ...) ▷ 将除法部分完成的行向量数据传递给各进程
10:      end for
11:     else
12:       MPI_Recv(&Ak0, N, MPI, src, ...) ▷ 注意, 数据源进程 Src 需根据 k 进行判断
13:     end if
14:     for  $i \leftarrow \max(k + 1, begin)$  to  $end$  do
15:       for  $j \leftarrow k + 1$  to  $n$  do
16:          $A_{ij} \leftarrow A_{ij} - A_{ik} \cdot A_{kj}$ 
17:       end for
18:        $A_{ik} \leftarrow 0$ 
19:     end for
20:   end for
21: end function

```

3.5.2 不同进程数的探究

在 arm 平台上进行 100, 500, 1000, 1500, 2000, 3000 规模的 MPI 基础算法不同线程数探索, 进程数分别为 4 进程、8 进程、12 进程。以串行算法的耗时与并行算法的耗时之比作为优化力度评估优化算法的影响。

进程数	100	500	1000	1500	2000	3000
平凡算法	0.025	0.31	2.58	8.74	22.1	74.57
4 进程	0.084	0.74	1.91	5.56	13.33	44.74
8 进程	0.2	0.96	2.41	4.6	8.59	30.32
12 进程	0.35	1.59	3.59	6.28	9.61	25.55

表 8: 不同进程数下 MPI 基础优化用时

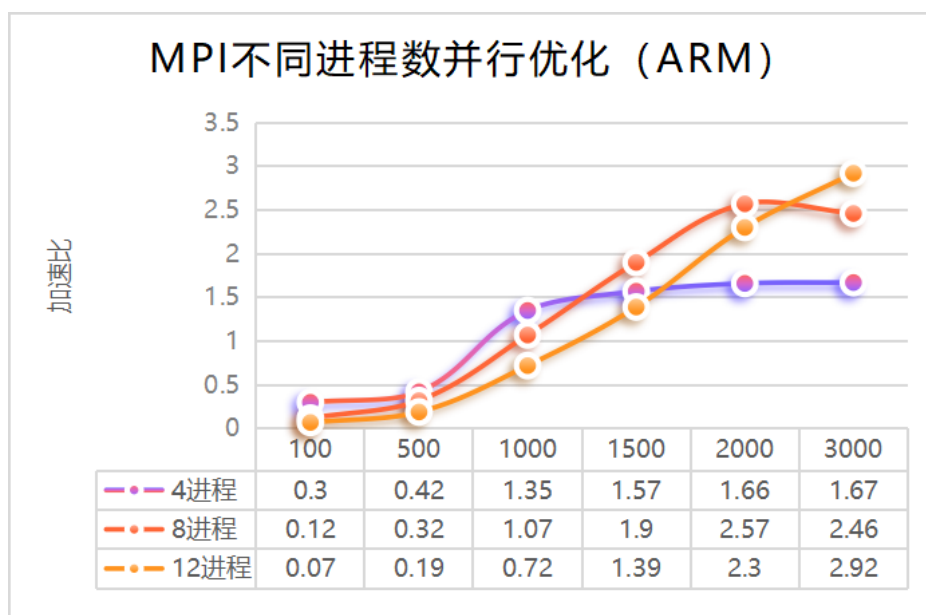


图 3.25: MPI 不同进程数并行优化加速比

从图像3.25和数据中，可以得出以下结论：

- 多进程 MPI 的初始负优化效果：**在数据规模较小（例如 100 和 500）时，多进程 MPI 可能会因为进程间通信和数据同步的开销而导致性能下降，即所谓的负优化效果。这可能是因为在小数据规模下，进程间通信和数据同步的时间占总执行时间的比例较大，从而抵消了并行计算带来的速度提升。
- 数据规模对优化效果的影响：**随着数据规模的增加（从 1000 开始），多进程 MPI 开始展现出优化效果。这是因为随着数据量的增加，每个进程可以处理更多的数据，从而减少了进程间通信和同步的相对成本。
- 进程数对优化效果的影响：**在数据规模较大时，更多的进程数可以带来更好的优化效果。例如，8 进程 MPI 在 1500 规模后开始超过 4 进程 MPI，而 12 进程 MPI 在 2000 规模时超过 4 进程，并且在 3000 规模时成为最优。这说明随着数据规模的增加，更多的进程可以更有效地并行处理数据，从而提高整体性能。

3.5.3 数据划分方式的探究

- 在上一节的测试中用的是块划分(Block Decomposition)方式,每个线程给予 N/MPI_Comm_size 个行向量的任务量,而当 $N \% MPI_Comm_size \neq 0$ 时,最末尾的一个进程会被赋予额外的任务量,最多可以是 $MPI_Comm_size - 1$ 个行向量,因此可以尝试其他的任务划分方式。
- 一种是利用循环划分(Cyclic Decomposition)的方法,进行等步长的划分,以 MPI_Comm_size 为间距为进程划分任务,可以做到即使末尾有多出来的任务,也能分配给前 $N \% MPI_Comm_size$ 个进程,从而达到更均衡的任务分配。但是这样的方式在相互传递信息和进行接收信息、进行消去部分的代码需要重新设计,略微复杂。
- 还有一种方式是在块划分的基础上,将末尾多余的行向量均分给前 $N \% MPI_Comm_size$ 个进程,这样的方式在进程数较多(即末尾余下数据越多)时能使各进程间负载更均衡,防止出现

最末尾进程的任务量明显高于其余其他进程的情况。

数据划分方式	1000	1500	2000	3000
平凡算法	2.58	8.74	22.1	74.57
普通块划分	2.04	5.89	14.19	49.16
任务更均衡的块划分	1.95	5.81	14.15	45.3
循环划分	1.76	5.82	12.97	43.71

表 9: MPI 不同数据划分方式耗时

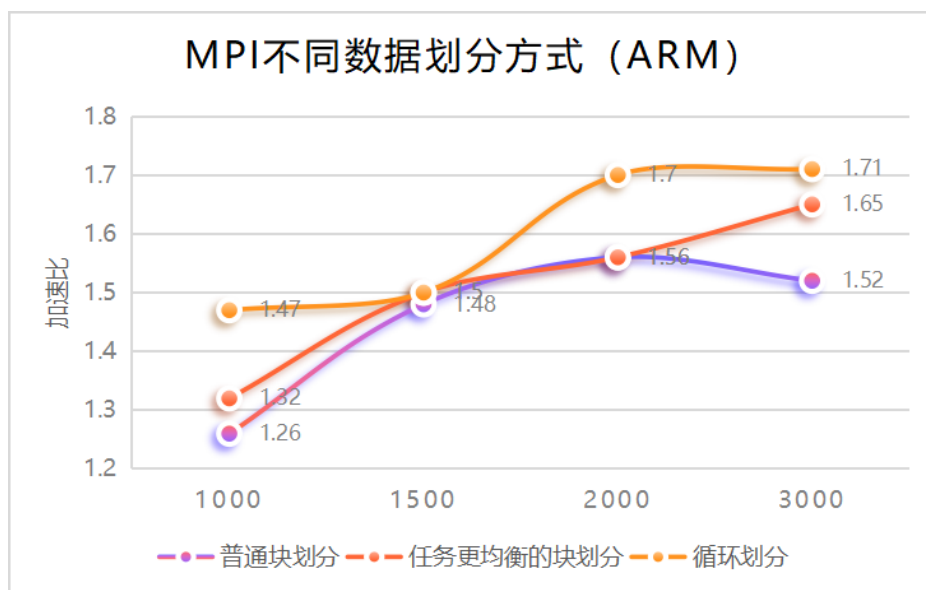


图 3.26: MPI 不同数据划分方式加速比

- **普通的块划分耗时最长**，这可能是因为这种划分方式没有考虑到数据的局部性和任务的均衡性，导致某些进程可能在等待其他进程完成，或者某些进程的工作负载比其他进程重。
- **负载均衡后的块划分与原始块划分的性能表现相近**，这是因为它们的总体结构相同，而且当数据规模较小时，二者差别不大，除非是单一一行行向量数据巨大（3000 规模），或进程数目较多时，二者差别才会明显。
- **循环划分的性能相对较好**，它是一种更为平衡的任务划分方式，循环划分通过循环地将数据分配给各个进程，可以更好地平衡各个进程的工作负载，从而提高整体性能。

3.5.4 x86 平台与 ARM 平台对比

本次在 x86 平台上的实验选用了 MS-MPI 这一种 MPI 接口的实现，MS-MPI 是 Microsoft Message Passing Interface 的简称，是微软公司开发的一种用于编写并行分布式计算程序的通信库。MS-MPI 主要用于在 Windows 平台上开发高性能计算应用程序，因此此次 MS-MPI 将搭配 Visual Studio 2019 进行 MPI 编程实验。

1. 不同进程数的探究：

x86 平台的 MPI 优化的思路与 ARM 平台的相同，代码上也大同小异，测试规模 100, 500, 1000, 1500, 2000, 3000，具体测试结果如下 (ms)：

进程数	100	500	1000	1500	2000	3000
平凡算法	0.5	140.5	1047.75	3518.75	8467.74	28751.34
4 进程	0.59	50.07	385.57	1309.24	2880.15	10906.46
6 进程	1.06	41.46	313.23	1061.31	2215.07	7858.7
8 进程	1.81	37.41	289.42	934.04	2006.88	6997.9

表 10: 不同进程数下 MPI 基础优化用时 (x86)

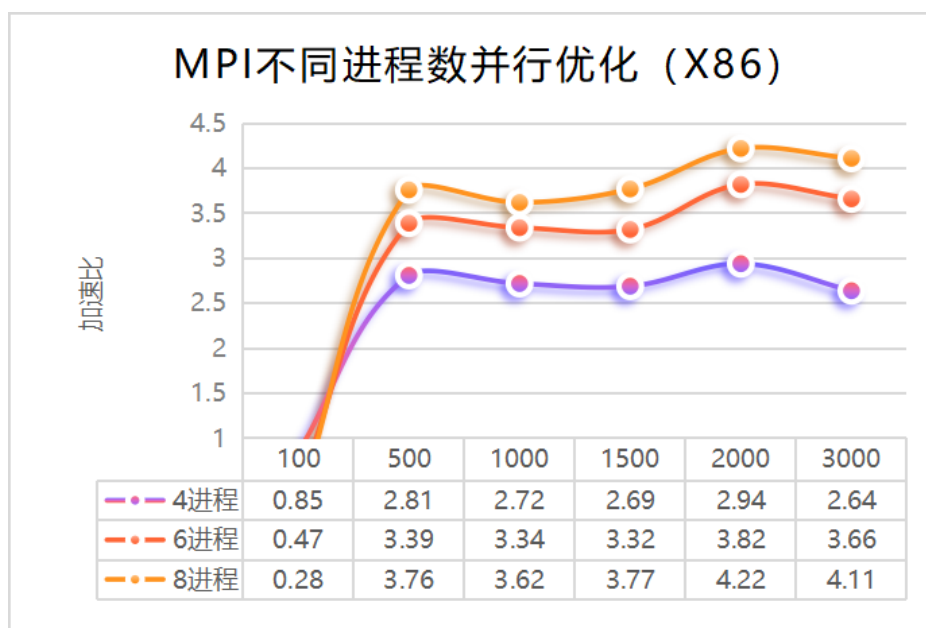


图 3.27: MPI 不同进程数并行优化 (x86)

和 arm 平台 (图3.25) 表现类似, MPI 有很好的加速效果, 特别是在 x86 平台上, 在数据规模较大时四进程时加速比稳定在 2.5 以上 (如图3.27); 而当增加进程数至 6 和 8 时, 加速比还有进一步的提升, 在数据规模较大 (3000) 时, 八进程的加速比超过了 4, 有着优异的表现。

2. 数据划分方式的探究:

这部分主要对循环划分和块划分以及均衡负载后的块划分这三种方式进行探究, 将在 x86 平台上, 四线程的设置下检测不同划分方式和编程策略的差别, 经过多次实验后取得数据如下 (单位: ms):

数据划分方式	500	1000	1500	2000	3000
块划分	2.81	2.71	2.69	2.89	2.67
任务更均衡的块划分	2.84	2.77	2.69	2.87	2.69
循环划分	2.77	3.43	3.63	3.73	3.6

表 11: MPI 不同数据划分方式耗时 (x86)

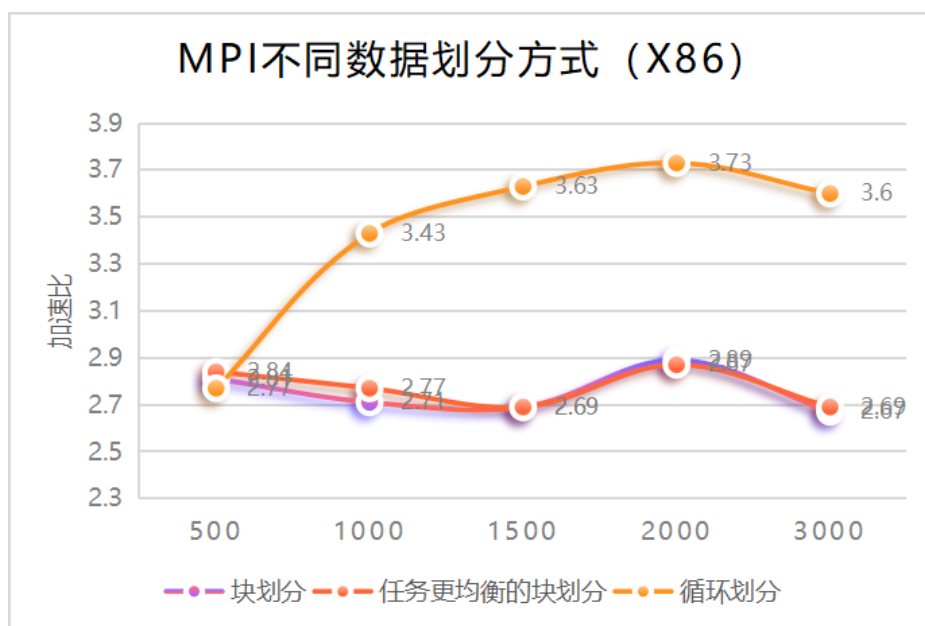


图 3.28: MPI 不同数据划分方式 (x86)

对比图3.28和图3.26:

- 负载均衡后的块划分的方式和普通块划分差别不大, 这是因为 4 进程时, 末尾的多余任务为小于 4 个行向量, 除非是每个行向量的计算量巨大, 否则这种方法和普通块划分差别微小。
- 循环划分是所有里面耗时最少的, 因为这种方式任务划分是十分均衡的, 加速比相对于块划分的提升是较为明显, 在 2000 规模时, 两种块划分的加速比在 2.89 左右, 而循环划分加速比在 3.73, 是块划分的 1.3 倍左右, 这一点比较与 arm 平台实验结果类似。
- x86 平台对于任务划分的优化性能更敏锐, 相比 arm 可以更高效地实现 MPI 的多进程并行化, 因为 arm 数据划分的最优性能加速比为 1.7, 而 x86 加速比可达 3.73, 是 arm 平台加速比的两倍多。

3.5.5 非阻塞通信 MPI 的探索:

非阻塞通信策略, 即发送进程将在消息被接收进程接收之前一直等待的通信方式, 直到接收进程成功接收消息后, 发送进程才能继续执行; 阻塞通信简单易用, 由于发送进程要一直等待接收进程的响应, 因此可以避免出现竞争条件。但是, 这会导致整个程序的效率降低, 因为发送进程必须等待接收进程处理完消息之后才能继续执行。

在高斯消元算法中, 每个进程发送的数据相互独立, 不会相互干扰, 因此适合采取非阻塞通信, 用 `MPI_Isend(void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request)` 进行数据的非阻塞传递。但要注意, 在这次实验的算法的数据接收一定要等到接收完毕后才能执行下一步, 所以要用 `MPI_Wait(&request, MPI_STATUS_IGNORE)` 进行阻塞。非阻塞版本的代码可以参见[非阻塞优化代码](#)。

在 x86 平台, 四进程, 块划分上运用非阻塞策略得到的结果如下 (单位: ms):

通信方式	500	1000	1500	2000	3000
平凡算法	140.5	1047.75	3518.75	8467.74	28751.34
阻塞版本 MPI	50.41	364.11	1283.72	3075.4	10745.62
非阻塞版本 MPI	45.86	298.95	1023.52	2872.43	10167.29

表 12: 改变通信方式 MPI 优化

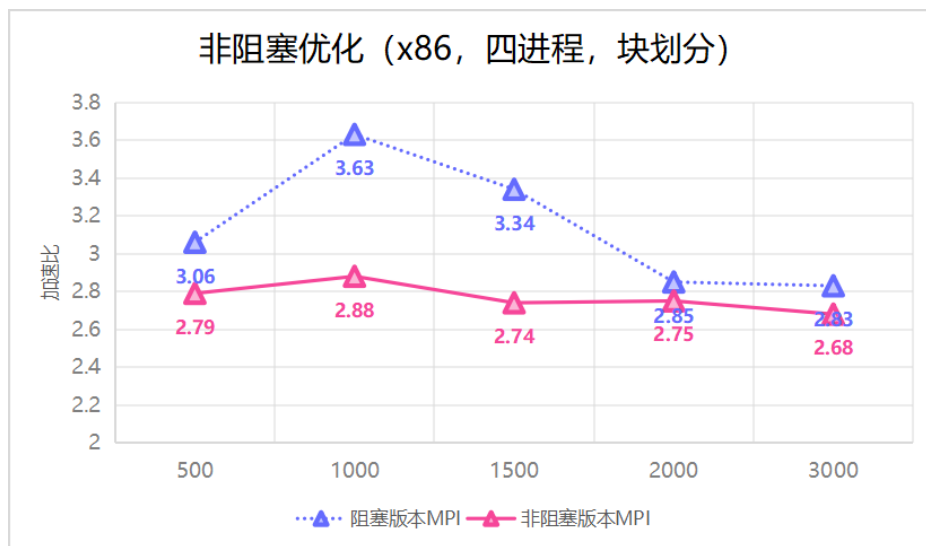


图 3.29: 非阻塞优化 (x86, 四进程, 块划分)

从实验结果来看，运用非阻塞技术是有效且合理的，最高可达 3.63 加速比（1000 数据规模），能切实提升 MPI 程序的速度和效率，虽然在更大数据规模中提升有限，但是仍然效果明显。后续实验设计如果在更复杂的环境条件下，需要更谨慎判断非阻塞是否会造成数据传送的混乱。

3.5.6 MPI 流水线形式的改进

本节为新加入的 mpi 改进实验：在先前的算法中，每个进程会将消元完毕的行向量传递给到整个进程域，然而在块划分中，实际上只有该进程之后的进程需要使用该行向量，因此只用将数据传递给相邻之后的进程。流水线一般操作，这样可以减少约一半的数据传递操作，最终在最后一个进程中得到最终结果。

在设计代码中，根据之前探究的非阻塞通信，只需要当前进程非阻塞地传递数据给之后的进程即可，这样可以避免每个进程接收数据后又传递的“搬运”过程。流水线改进版本的代码可以参见[流水线改进](#)。

流水线改进	1000	1500	2000	3000
ARM 普通 MPI (s)	1.94	5.85	13.42	46.52
ARM 流水线版 MPI (s)	1.52	4.13	9.2	30.63
x86 普通 MPI (ms)	389.63	1312.24	2873.25	10825.35
x86 流水线版 MPI (ms)	324.12	1101.77	2321.26	8208.36

表 13: MPI 流水线改进

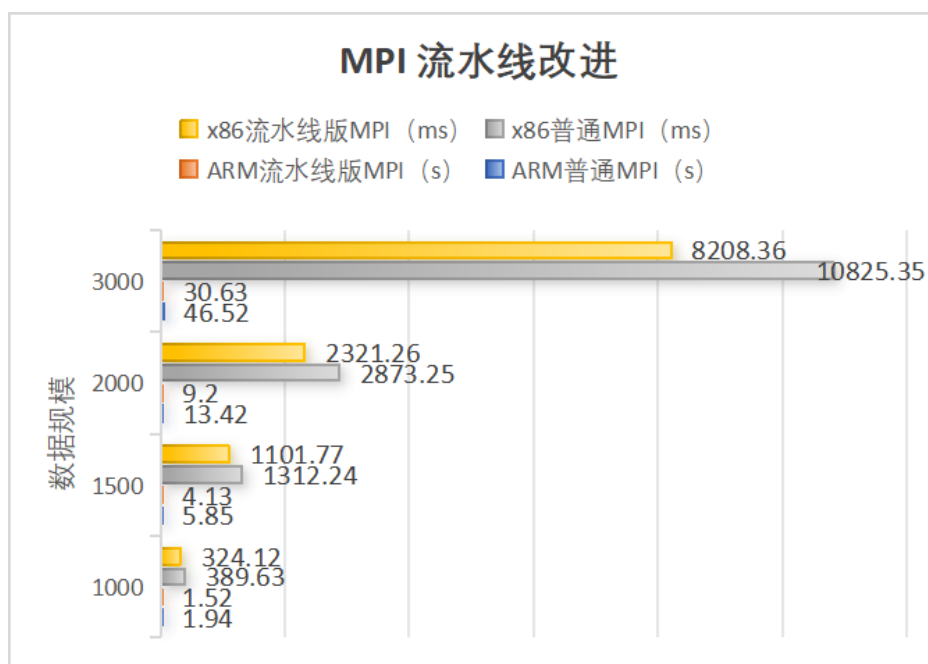


图 3.30: MPI 流水线改进

据实验结果可知，减少信号传递次数后的流水线版本 MPI，在速度上确实有提升，可以使得算法执行更快；但是提升幅度不算很高，推测可能是因为原先的运行中，数据传递占据的时间就不长，因此提升不明显，但是流水线改进依然有用。

3.5.7 MPI、多线程、SIMD 的结合

从 MPI 算法中可以发现，在执行行消去的部分时，每个进程要负责 N/total 个行向量的消去，此时可以结合 OpenMP 进行多线程优化，每个线程负责 $N/\text{total}/\text{threads}$ 个行向量，并且采用了任务分配更灵活均衡的 guided 方式，能够有效加快每个进程的执行速度。

而在除法部分，或是对单一行进行消去时，可以结合 SIMD 的技术进行优化，一次性对一行中多个元素同时除去对角线元素或同时进行行间的消去，也能达到提升执行速度的效果。

结合方式	500	1000	1500	2000	3000
普通 MPI	50.41	364.11	1283.72	3075.4	10745.62
非阻塞 MPI	45.86	298.95	1023.52	2872.43	10167.29
非阻塞 MPI+AVX	25.28	145.63	782.16	1920.57	5825.33
非阻塞 MPI+OpenMP	20.35	90.82	301.24	650.25	3524.42
非阻塞 MPI+OpenMP+AVX	18.88	53.54	160.19	361.9	1748.55

表 14: MPI 结合 OpenMP & SIMD (x86, 块划分)

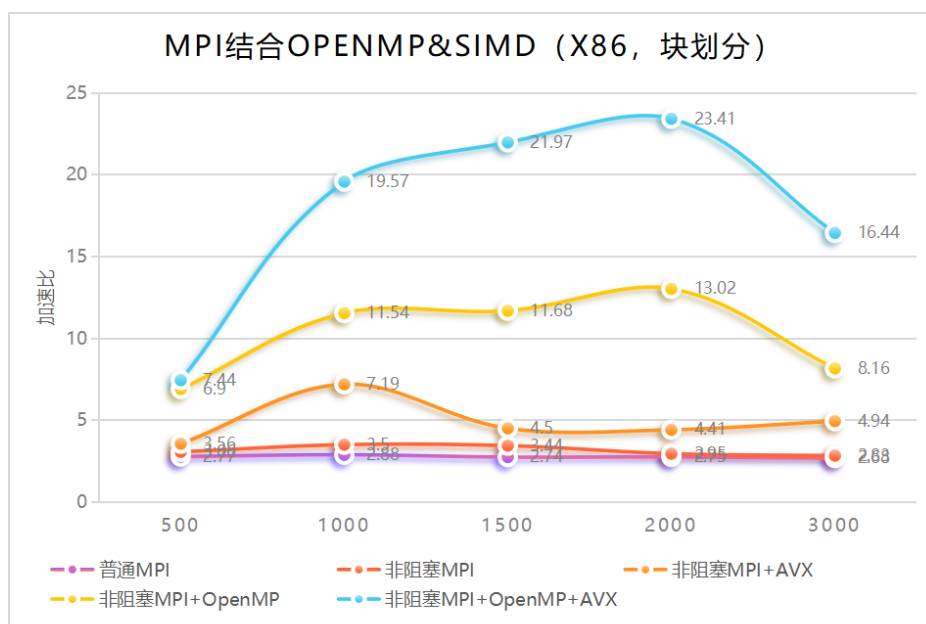


图 3.31: MPI 结合 OpenMP & SIMD 加速比

最终把三种优化手段结合，得到结论：

- 在本次结合了 MPI 的多进程优化后，高斯消元算法有多个进程分配任务，每个进程里又有多个线程进行并行计算，每个线程又会利用 AVX 指令集进行 SIMD 加速，最终达到的加速比最高可达 23.4。
- 当数据量较小（500）时加速比的提升无论是和 AVX、OpenMP 单独结合还是同时结合二者，加速效果都不明显，三种方式差距也不大，推测认为是因为数据量较小，无法完全体现加速优势。
- 当数据量增加时，加速比一度升高，而在 3000 规模时，可能由于进程间通信增多或计算量增大等，同时结合 SIMD 和 OpenMP 的加速比降低至 16.44。
- 依据上面非阻塞通信带来的优势，结合 AVX、OpenMP 后又带来了新的飞跃，这三者的有机结合使得算法大大加速。

3.6 CPU 优化的最终加速方法

在本学期的工作中，我综合了之前所有的研究成果，做出了最新的加速版本，包括以下几个方面：

- 利用 SIMD 技术进行加速，尤其是内存对齐后的 AVX 指令集，其性能表现最为出色；
- 应用多线程技术 OpenMP 进行加速，这种方法非常适合与多进程技术相结合使用；
- 实现了非阻塞通信的 MPI 版本，优化了流水线的执行效率。

这些优化措施的结合，预计将带来显著的性能提升。所有这些探索的成果和最终加速的代码都汇总在了“CPU 最终加速代码”中。

测试结果如下，单位为毫秒：

	500	1000	1500	2000	3000
平凡算法	140.5	1047.75	3518.75	8467.74	28751.34
最终加速	12.23	50.32	152.68	342.56	1212.08

表 15: CPU 最终加速结果

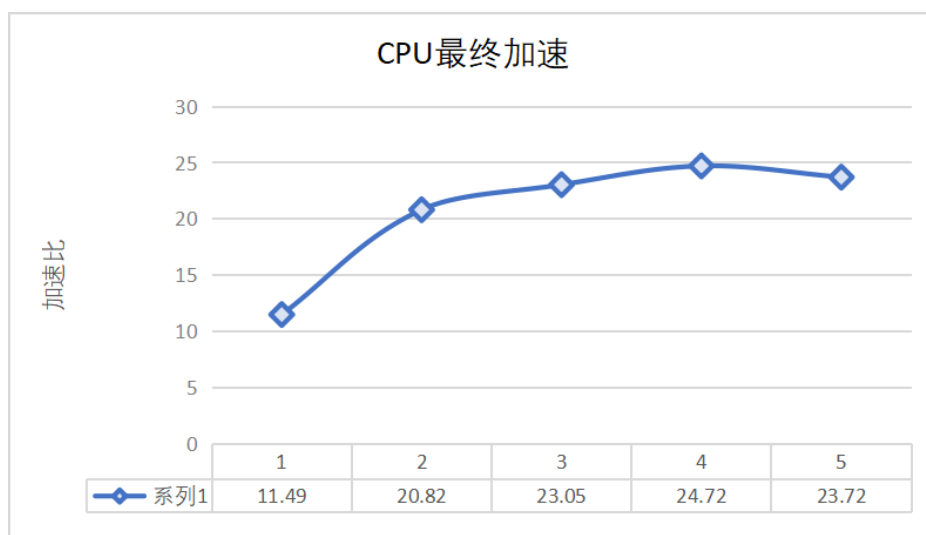


图 3.32: CPU 最终加速

- **数据规模与加速比的关系:** 在数据规模为 500 时, 加速比相对较低, 这可能是由于数据规模较小, 导致并行处理的优势没有完全发挥出来。同时, 进程通信和线程管理的开销可能在这个规模下占据了一定比例, 影响了整体的加速效果。
- **高数据规模下的稳定表现:** 当数据规模达到 1000 时, 加速比稳定在 20 倍以上, 最高达到了 24.72。这表明随着数据规模的增加, 本学期采用的并行加速方法能够更好地利用计算资源, 实现更高效的计算。
- **算法优化的成果:** 通过并行加速工作, 高斯消元算法的耗时从千级、万级毫秒降低到了数百毫秒, 这是一个巨大的进步。这不仅提高了算法的执行效率, 也为处理大规模问题提供了可能。

总结: 本学期的工作不仅在理论上探索了多种加速方法, 而且在实践中取得了显著的成果。通过结合 SIMD 加速、多线程 OpenMP 和非阻塞通信的 MPI, 实现了高斯消元算法的高效并行化。

3.7 GPU 计算的优化探究

GPU 是专为图形渲染设计的计算机硬件, 与 CPU 相比, 它基于高吞吐量设计, 拥有大量计算单元, 非常适合处理大规模并行计算任务。GPU 在 AI 训练等需要大规模并行处理的领域中表现出色, 得益于其较大的内存带宽, 可以高效地一次性读取大量数据, 而 CPU 则在这方面相对较弱。

在进行 GPU 优化时, 我采用了 NVIDIA 的 CUDA 编程模型, CUDA 能够充分发挥 NVIDIA 显卡的高性能并行计算能力, 适用于执行复杂的大规模计算任务。

3.7.1 CUDA 编程优化

CUDA 的编程模式图如下:

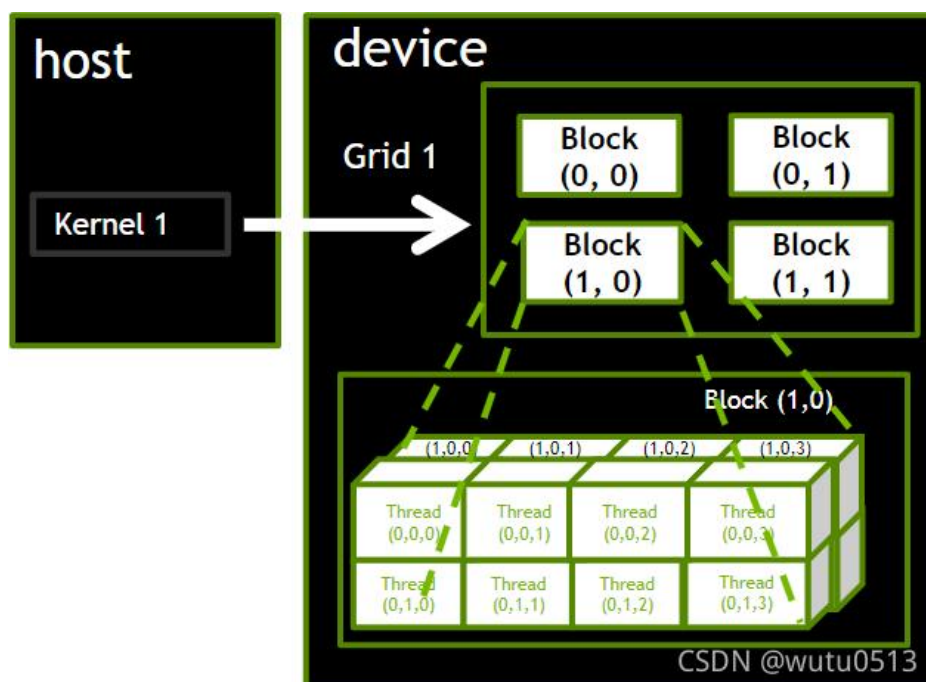


图 3.33: CUDA 编程模式图

首先对矩阵进行初始化，然后将其赋值到一维数组 *temp* 中，用于传输给 GPU 进行高斯消元，接下来的步骤包括：

- 调用 `cudaMalloc()` 函数在 GPU 端分配一个 $\text{sizeof(float)} \times N \times N$ 大小的内存空间，并以 *gpudata* 作为这一块空间的内存句柄。
- 使用 `cudaMemcpy` 函数将数据从 CPU 复制到 GPU 中。
- 使用 `dimBlock()` 和 `dimGrid()` 定义线程块和网格的维度。线程块的维度是 $(\text{BLOCK_SIZE}, 1)$ ，每个线程块包含 `BLOCK_SIZE` 个线程，但在 *y* 方向上只划分一个。线程网格 *dimGrid* 的维度是 $(1, 1)$ ，只有一个线程块组成线程网格。
- 使用 `cudaEvent` 计时器开始计时。

在 CUDA 优化代码中，定义了两个核函数 `division_kernel` 和 `eliminate_kernel`，分别用于高斯消元的除法和消元步骤，具体代码见 [CUDA 优化加速](#)。

实验结果如下表（单位 ms）：

CPU VS GPU	64	256	512	1024	2048	4096
CPU_time	0.19	10.63	71.13	210.78	1748.35	4850.21
GPU_time	5.01	6.56	12.85	20.35	58.33	118.30
加速比	0.04	1.68	5.86	10.55	30.01	41.00

表 16: CUDA 加速运行时间表

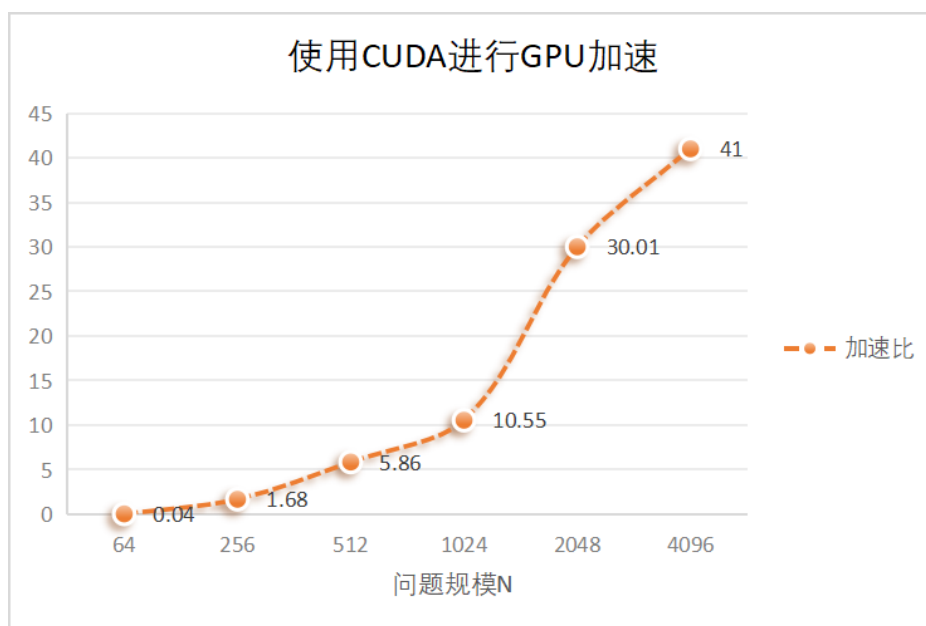


图 3.34: 使用 CUDA 进行 GPU 加速

根据实验结果和对图3.34的分析，得出以下结论：

- **初期加速不明显：**在数据规模较小（例如 $N=64$ ）时，GPU 加速效果并不明显，甚至可能比 CPU 慢。这可能是因为小规模数据下，CPU 的计算能力已经足够处理，而 GPU 在这种情况下需要进行额外的线程管理和任务分配，导致效率降低。
- **加速比的非线性增长：**随着数据规模的增加，GPU 的加速比呈现出非线性增长的趋势。这表明 GPU 在处理大规模数据时，其并行处理的优势逐渐显现，加速效果越来越明显。
- **GPU 的计算优势：**GPU 拥有大量计算单元和高内存带宽，这使得它在处理大数据量时能够一次性读取和处理更多数据，从而实现高吞吐量和快速计算，达到最高加速比 41.0。相比之下，CPU 的内存带宽较小，数据读取能力有限，导致在大规模数据处理时性能受限。
- **架构差异的影响：**GPU 的架构天然适合并行计算，特别是在面对大量数据时，其并行加速效果尤为明显。而 CPU 由于架构和内存带宽的限制，在并行计算方面的表现不如 GPU。

3.7.2 OneAPI 平台上的 GPU 加速尝试

期末新的工作任务之一是用 OneAPI 实现了 GPU 加速程序的编写，即学习了部分 SYCL 之后，继续对高斯消元算法进行了部分优化，思路清晰简单，对于高斯消元的除法和消去部分，可以用 `parallel_for` 语句来将循环部分代码进行并行化，总体思路并不复杂，具体代码见：[OneAPI 优化实验](#)，执行结果如下（单位：ms）：

	128	256	512	1024
串行算法	55.21	415.87	3569.26	24648.12
GPU 加速	4.24	10.56	80.28	612.88

表 17: OneAPI 加速运行时间表

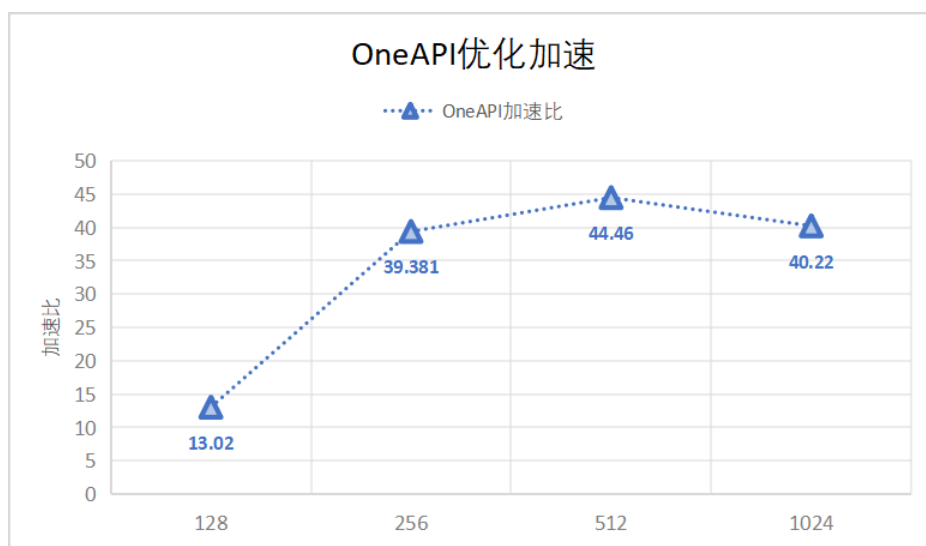


图 3.35: OneAPI 加速比

- 与本地 GPU 加速的比较：根据描述，OneAPI 在小规模数据上的加速比高于本地主机 GPU 加速。这可能归功于 OneAPI 的优化策略，包括内存管理、线程调度和内核执行等方面。
- 浮点运算的适用性：OneAPI 的高加速比进一步证明了 GPU 在进行大规模浮点运算时的适用性和优势。GPU 的高吞吐量和并行处理能力使其成为处理这类任务的理想选择。
- 大规模数据的性能：在数据规模达到 1024 时，**OneAPI 仍然能够保持在 40 倍左右的加速比**。这说明即使在处理大规模数据时，OneAPI 也能够提供强大的性能支持。
- OneAPI 的优化潜力：图表中的数据也表明，OneAPI 在不同数据规模下都展现出了优化的潜力。这为未来的性能提升和应用开发提供了广阔的空间。

4 特殊高斯消元的并行化

4.1 SIMD 优化

期末新的工作：完善了此前进行优化时的不足，此前进行 SIMD 优化时，**仅仅考虑了消去时的 SIMD 优化，没有考虑到进行消元子升格时也能进行 SIMD 优化**。优化后的代码为：**SIMD 优化特殊高斯消元算法**。

4.1.1 X86 平台——SSE、AVX、AVX512 指令集

代码的编写步骤同样与上一节相似，并且 SSE、AVX、AVX512 指令集的代码重合度极高，同样对是否对齐的情况进行了实验，并对结果进行了总结分析。

4.1.2 实验结果

(1) 对于非对齐算法：

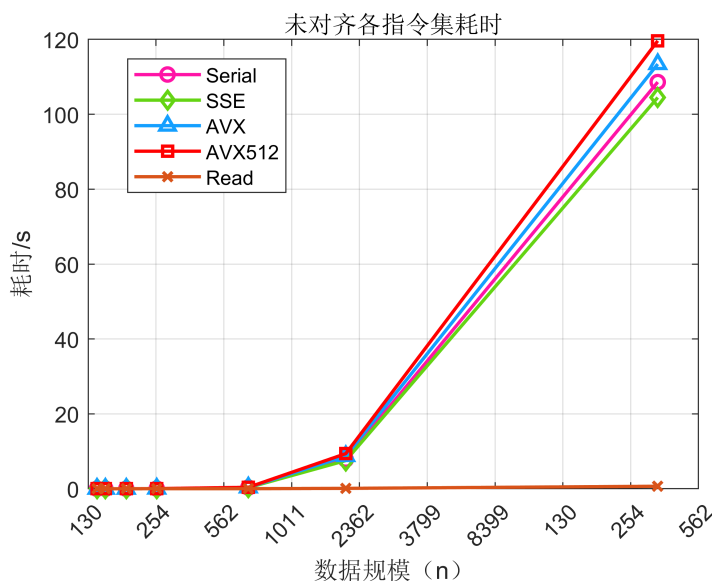


图 4.36: 特殊高斯消元非对齐并行化

SSE、AVX 和 AVX512 架构的耗时曲线变化趋势相似，表明这些架构对算法性能的影响具有一定的共性。SSE 和 AVX 架构对提升算法性能有轻微的正面影响，但效果有限；相比之下，AVX512 架构在本实验的并行化尝试中并未展现出优势，其耗时反而超过了串行算法。

(2) 对于对齐算法：

由于 X86 平台下的指令集在对齐的情况下使用的指令与非对齐时略有不同，因此还要进行指令的修改。对代码进行修改后，编译和运行的操作同上，结果如下所示：

综合分析表明，对于本程序，AVX 和 AVX512 指令集并未实现有效的性能优化，而 SSE 指令集在并行化方面表现最佳，尽管其优化效果有限。对于 cache 规模等因素，结论与未对齐时相同，即它们对算法性能的影响不明显。

4.1.3 优化编译选项

本次期末新的工作：在 X86 平台上，调整编译优化参数 (-O1、-O2、-O3)，来提升程序的运行速度，以下是实验总结：

- 在 X86 平台上，串行算法的优化效率排序为 $-O0 < -O2 = -O3 < -O1$ ，而并行算法的排序为 $-O0 < -O3 < -O2 < -O1$ 。
- 优化编译选项在 x86 上能显著提升程序运行速度。例如，当问题规模 $col=3799$ 时，在 X86 平台使用 -O1 参数时，速度提升约 3 倍。
- 不同指令集之间优化速率略有不同，SSE, AVX 优化效果较差，平均提速为 2 倍左右；AVX512 优化效果较好，最高提速可达到 4 左右。

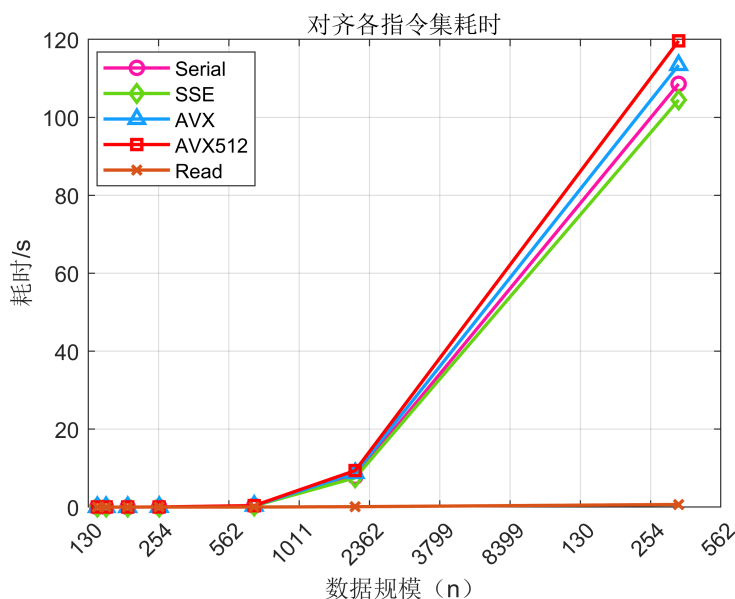


图 4.37: 特殊高斯消元对齐并行化

4.2 openmp 和 pthread 优化

4.2.1 pthread 优化思路

在多线程高斯消元算法中，我们的目标是同时处理多行以加速计算。但这样做可能会引起冲突，特别是当消元后多行的首项系数相同时，若直接将任一行设为消元行，会导致错误。为解决这个问题，我采用了互斥锁（Mutex）来管理线程对消元行的访问，代码见[特殊高斯消元算法 pthread 优化](#)。

- 当一行准备成为消元行时，它会先锁定互斥锁，防止其他线程同时修改消元行。
- 锁定后，该线程会更新消元行，并在完成后释放锁。
- 其他线程在锁被释放后，会再次检查是否有重复的首项系数，如果有，它们将重新进行消元；如果没有，它们将更新消元行并锁定互斥锁。

4.2.2 OpenMP 优化思路

选用 guided 任务划分方式，在需要进行加锁阻塞的消元子写入段的代码，**本次期末汇总做了更正**：使用 `#pragma omp for ordered schedule(guided)` 结合 `#pragma omp ordered` 既保证了没有数据冲突冒险——只有一个线程执行写入消元子，又保证了写入消元子的顺序依赖不被打破。经测试，这样的方式能准确得到结果。代码见[特殊高斯消元算法的 OpenMP 优化](#)。

此外，我还尝试了 pthread、OpenMP 与 SIMD 指令集的结合，探究如何进一步提升加速比，选取的是目前更新、更有运用价值的 AVX 指令集，代码见[特殊高斯消元算法 OpenMP 优化](#)。

4.2.3 实验结果

(1) 耗时对比

优化类型	例编号										
	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 10	例 11
普通算法	0	3	3	81	355	3999	24745	186101	279961	878052	485
平凡算法 (pthread 优化)	0	1	1	46	238	2704	16967	123238	185819	578223	198
平凡算法 (OpenMP 优化)	0	0	1	8	39	533	3141	26111	44197	132280	36
AVX 优化 (pthread)	0	1	1	12	60	734	4664	32222	53451	222136	80
AVX 优化 (OpenMP)	0	0	0	7	30	505	3716	27781	46044	151274	35

表 18: 特殊高斯消元的 pthread、OpenMP 优化, 结合 AVX 指令集

由于例 1 至例 3 加速到已经无法测得具体耗时, 因此将从例 4 开始讨论加速比。仅从数据表18而言, 可见运用 pthread 和 OpenMP 进行多线程优化后运行耗时明显降低了许多, 且由于 pthread 和 OpenMP 均用 8 线程进行并行加速, 所以二者在很多时候表现相近。

(2) 加速比对比

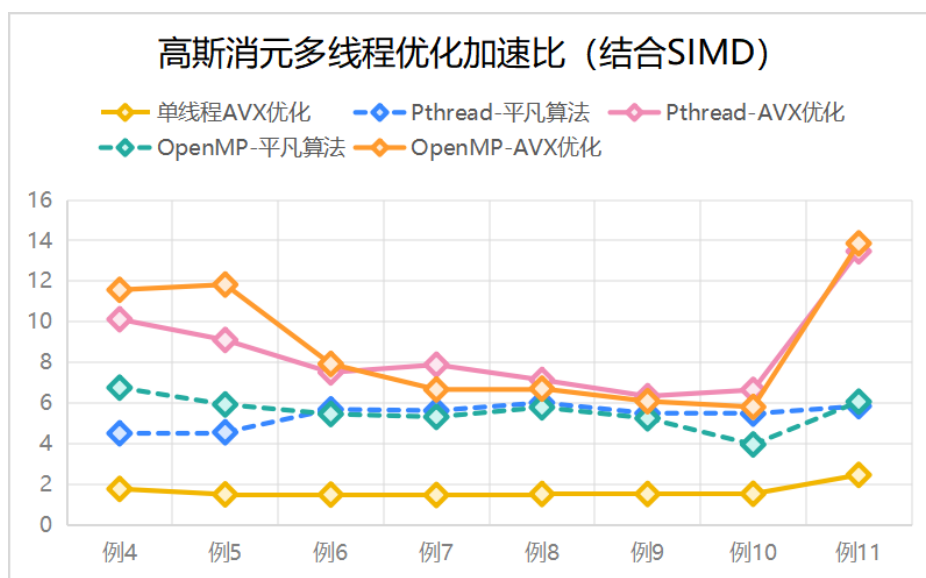


图 4.38: 高斯消元多线程优化加速比 (结合 SIMD)

可知, 相比于单纯的 AVX 指令集的 SIMD 优化, 加入 pthread 和 OpenMP 后性能获得了很大的提升, 从实验结果图像4.38中可以获得以下结论:

- 多线程优化的重要性: 实验显示, 单纯的 AVX 优化带来的性能提升有限, 而结合多线程优化后, 性能提升显著, 加速比可高达 7 倍甚至更多。
- 线程库的选择: 在轻量级计算任务中, OpenMP 优化效果普遍优于 pthread, 显示出其在任务调度和执行上的高效性。对于计算量大的任务, pthread 优化表现更佳, 这可能是因为 pthread 提供了更接近底层的线程管理能力。
- 多线程与 SIMD 优化的比较: 即便对于平凡算法, 多线程优化也能带来显著的加速效果, 有时甚至超过 SIMD 优化, 这表明多线程优化在某些情况下更为有效。

总之, 此次特殊高斯消元多线程加速达到了很好的效果。相比上一次的 SIMD 优化, 多线程达到了更好的提升幅度, 而二者结合之后效果更加显著, 在面临巨大计算量时也能有 6 到 7 的加速比, 而在任务量较轻时能达到超过 10 的加速比, 说明这一次的多线程加速达到了很好的效果。

4.3 MPI 优化

4.3.1 算法设计

其 MPI 优化算法的代码如下所示，其中 $R[i]$ 代表首项为 i 的消元子， $E[i]$ 代表第 i 个被消元行， $lp(E[i])$ 代表被消元行第 i 行的首项。

Algorithm 5 MPI 优化的特殊高斯消元算法

```

1: function GAUSSIAN_ELIMINATION(MatriA, begin, end, total)
2:   for  $i = \text{begin}$  to  $\text{end}$  do
3:     while  $Rip(Ei) \neq NULL$  do                                ▷ 先按照原本的消元子对行向量进行消元
4:       进行异或消元
5:     end while
6:     消元完毕先不升格为消元子
7:   end for
8:   for  $i = 0$  to  $\text{rank}$  do
9:     MPI_Recv from  $i$       ▷ 接收来自此前进程的结果，可能可以对自己进程的任务进行消元
10:    将接收的结果作为消元子，再次消元
11:   end for
12:   将自己进程的行向量逐个升格为消元子，如果顺序执行时有冲突，则再次消元
13:   for  $i = \text{rank} + 1$  to  $\text{total}$  do
14:     MPI_Send to  $i$         ▷ 将自身消元结果发送至此后的进程
15:   end for
16: end function

```

探究特殊高斯算法设计中的思路，具体代码可见[MPI 优化特殊高斯消元算法代码](#)。

- **数据划分：**算法不适合使用循环划分，因为这会增加算法的复杂性并破坏行之间的依赖关系，普通的块划分方式更有助于保持行的前后依赖关系不变。
- **通信方式：**接收消元行时，需要记录消元行首项的信息，因此接收部分必须是阻塞通信以避免错误。在发送消元结果时，由于进程即将完成其任务，即使使用非阻塞发送，目标进程也需要阻塞接收，因此使用非阻塞通信在这里并不必要。
- **结合多方法并行优化：**消元部分和升格消元子部分可以通过结合 OpenMP 和 AVX 指令集进一步优化，以提高加速比。

4.3.2 实验结果

实验环境与 x86 的 MPI 优化普通高斯消元相同（单位:ms）：

结合方式	Example Number									
	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 10
平凡算法	0	10	11	298	929	10259	58730	401253	758432	2358545
4 进程 MPI	0	4	4	115	308	4362	19646	140619	276549	920784
4 进程 +8 线程 OpenMP+AVX	0	7	7	102	272	2764	10532	71056	98524	320412
8 进程 MPI	0	6	4	98	203	2374	12293	84908	131237	381230
8 进程 +8 线程 OpenMP+AVX	1	9	10	108	238	1825	9918	45251	80160	240411

表 19: 特殊高斯消元 MPI 优化耗时

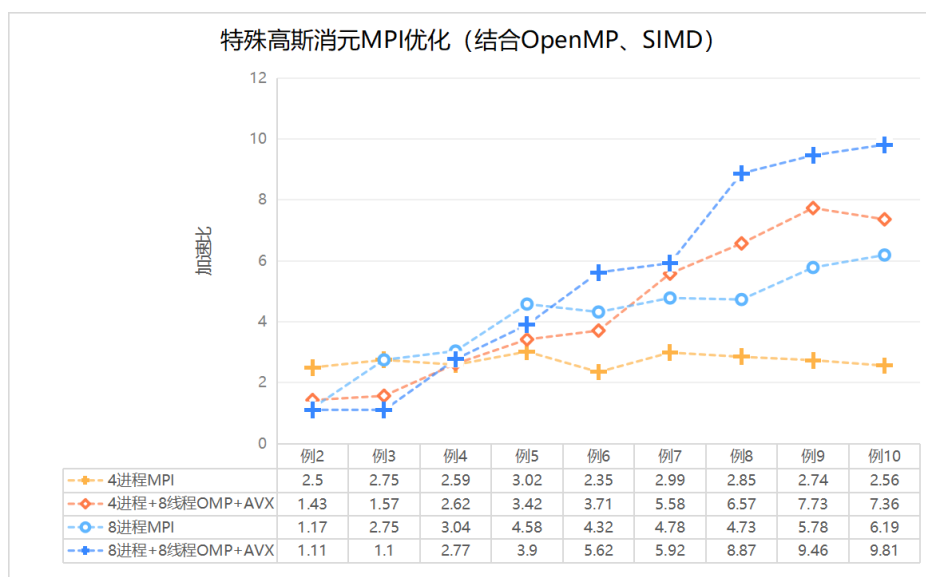


图 4.39: 特殊高斯消元 MPI 优化 (结合 OpenMP、SIMD)

根据实验结果，以下是对特殊高斯消元算法在 MPI 多进程优化、OpenMP 多线程和 SIMD 指令集 (AVX) 结合使用时的性能表现的总结：

- MPI 多进程优化效果显著：4 进程 MPI 普遍能够达到大约 2.5 到 3 倍的加速比，而 8 进程 MPI 在处理大规模数据时能够达到 5 到 6 倍的加速比，显示出良好的扩展性。
- 小规模数据的优化策略：对于小规模数据，进程和线程的管理开销以及进程间通信的开销可能会随着进程数量的增加而增长，从而抵消了并行计算带来的性能提升。
- 大规模数据下的最优配置：在处理极大规模的数据（如例 10 所示）时，结合 8 个进程、8 个线程以及 AVX 指令集的优化策略表现出最佳性能，实现了接近 10 倍的加速比。**这是本学期在特殊高斯消元算法中达到的最高加速比。**

综上所述，特殊高斯消元算法在并行化时需要根据数据规模的不同选择合适的并行策略，MPI 多进程优化结合 OpenMP 多线程和 AVX (SIMD) 指令集可以在大规模数据处理中实现显著的加速效果。

5 总结与反思

5.1 期末新的工作

- 对于特殊高斯消元代码进一步 SIMD 优化，同时特殊高斯消元的 SIMD 优化进行编译探究。
- 修正特殊高斯 OpenMP 优化代码致命错误，重新进行了 OpenMP 优化特殊高斯消元算法的测试和探究的工作。
- 用 OneAPI 实现了 GPU 加速程序的编写，并与 cuda 优化作对比。
- 新增流水线版本的 MPI 优化，并融入自己思考和想法。

- 在“CPU 优化的最终加速方法”一节中，使用 **AVX 指令集 + 内存对齐的处理 + OpenMP 的多线程优化 + MPI 的非阻塞流水线版本的有机结合**，达到了本学期以来普通高斯消元最高的加速比 24.74。

5.2 最终成果总结

本学期，我深入研究了多种并行计算技术，包括 SIMD、多线程 (pthread 和 OpenMP)、分布式计算 (MPI) 以及 GPU 加速，并成功将这些技术应用于高斯消元算法的优化。通过结合内存对齐、AVX 指令集、OpenMP 多线程以及 MPI 的非阻塞通信，我在普通高斯消元算法上**实现了高达 24.72 倍的加速效果**。此外，利用 GPU 的天然优势，**我进一步将加速比提升至 41 倍**，显著缩短了程序的执行时间。

在特殊高斯消元算法的优化中，我采用了 MPI 和 OpenMP 的结合，加上 AVX 指令集，实现了 9.81 倍的加速比，对于处理庞大数据量的任务来说，这是一个令人满意的成果。

总结来说，本学期的学习和实践让我对并行程序设计有了深刻的理解，并成功提升了两种高斯消元算法的性能，展示了 CPU 和 GPU 在并行计算中的潜力和应用前景。

5.3 反思

对高斯消元并行问题有了更深的把握，不仅掌握了 MPI、SIMD、OpenMP 等并行技术，还在实际项目中应用了这些技术，实现了显著的加速效果。实验内容有难度梯度，由于刚开始对并行编程的不熟悉，导致进行实验的时候 debug 较为困难，得到实验数据的准确性不高，返工情况较多。在实验中逐渐养成了规划好再动手，扎实编程基础的习惯。编程上的，非编程的，都有了长足的进步。

并行思想不仅提升了我的编程能力，还启发了我在程序设计上追求效率和性能，学会设计能够充分利用硬件资源的高效程序。这些收获和感悟，都是本学期学习并行计算课程的重要成果。