南開大學
Nankai University

计算机学院

并行程序设计作业报告

OpenMp 多线程编程

姓名：徐文斌

学号：2010234

专业：计算机科学与技术

2022 年 5 月 18 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。

OpenMP（Open Multi-Processing）是一套支持跨平台共享内存方式的多线程并发的编程 API，使用 C,C++ 和 Fortran 语言，可以在大多数的处理器体系和操作系统中运行，包括 Solaris,AIX,HP-UX,GNU/Linux,MacOS X, 和 MicrosoftWindows。包括一套编译器指令、库和一些能够影响运行行为的环境变量。

本实验将在 ARM 和 x86 平台上利用 OpenMp 多线程编程技术进行普通高斯算法的多线程优化实验，从而对 OpenMp 多线程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

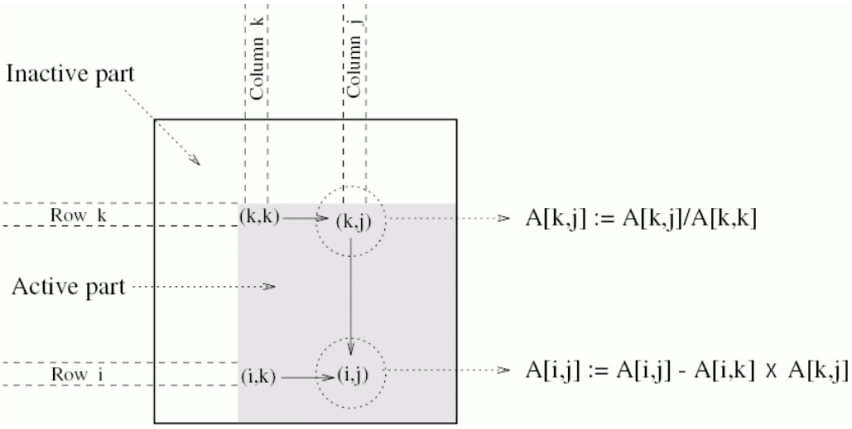

图 1.1: 高斯消去算法示意图

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，可以对内层除法循环后的内层二重循环进行 OpenMp 多线程优化。具体代码见文末 GitHub 链接。

1.3 实验环境说明

实验中的 ARM 部分使用鲲鹏服务器的 ARM 平台，使用 ARM 平台的 g++ 编译器进行程序的 OpenMp 多线程并行优化；实验中的 x86 部分使用 Intel 提供的 DevCloud 平台，利用 icpx 编译器进行程序的编译。

1.4 实验方案设计

普通高斯算法 OpenMp 多线程编程实验分为 ARM 平台和 x86 平台两部分。在 ARM 平台中，我们主要进行以下三部分内容：

- Part 1 探索 OpenMp 和 pthread 多线程编程对程序性能影响的差异。
- Part 2 探索 OpenMp 结合 NEON 超标量优化编程对程序性能的提升。同时探索使用 OpenMp 的预处理指令 #pragma omp simd 对 OpenMp 多线程程序进行 simd 并行化优化。
- Part 3 探索 OpenMp 不同任务分配方式 (static,dynamic,guided) 对程序性能的影响。

在 x86 平台中，我们主要进行以下两部分内容：

- Part 1 探索 OpenMp 结合 SSE 对程序性能的提升。
- Part 2 尝试进行 OpenMp 卸载到加速器设备，探索其对程序性能的影响。

1.5 ARM 平台实验结果呈现及分析

1.5.1 Part 1 OpenMp 和 pthread 多线程编程对程序性能影响的差异

首先，我们分别固定线程数为 2、4、6、8，探究线程数固定的情况下不同算法的性能随问题规模增大的变化。程序运行时间随问题规模变化趋势如图1.2所示。

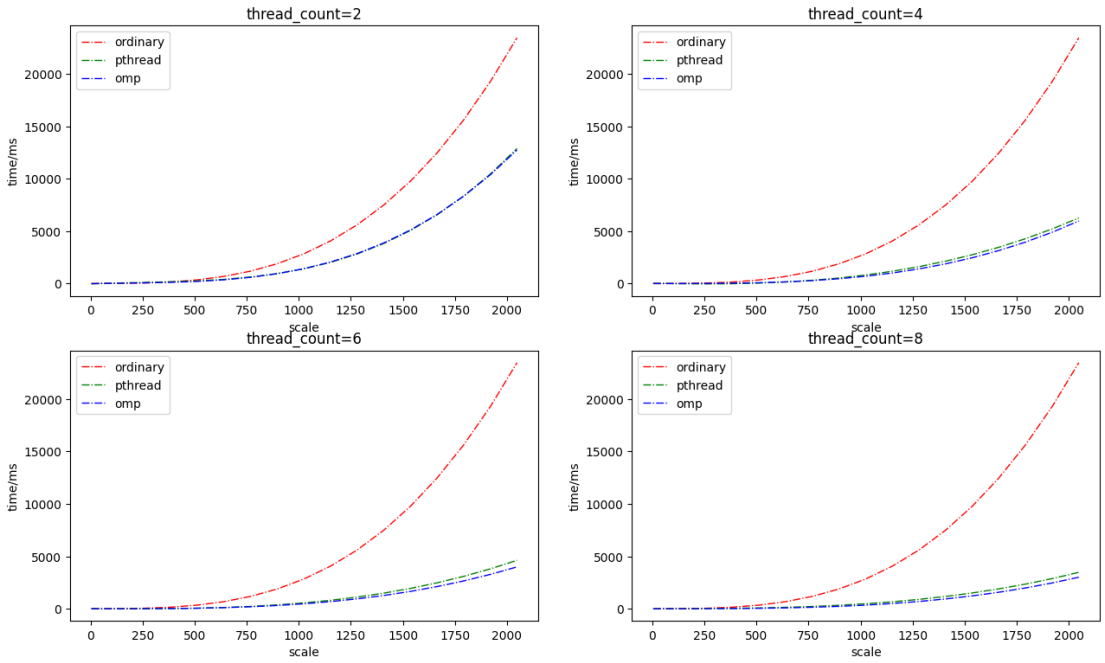

图 1.2: 多线程算法运行时间比较

可以看到，OpenMp 多线程算法对程序有明显的性能提升。且线程数相同情况下，OpenMp 多线程算法和 pthread 多线程算法运行时间大致相同，变化趋势和 pthread 算法一致，由此可见 OpenMp 的方便和高效。

此外,我们发现问题规模较大时,OpenMp 多线程算法运行时间在给定的线程数下均略小于 pthread 多线程算法。而两个算法进行的运算操作是相同的，因此，我们有理由认为 OpenMp 编写的多线程程序与 pthread 程序相比具有更小的线程同步开销。

OpenMp 算法的加速比如图所示1.3所示，在各个线程数下，程序均有接近甚至超越线程数的加速比，OpenMp 不失为一个替代 pthread 编程的良好编程范式。

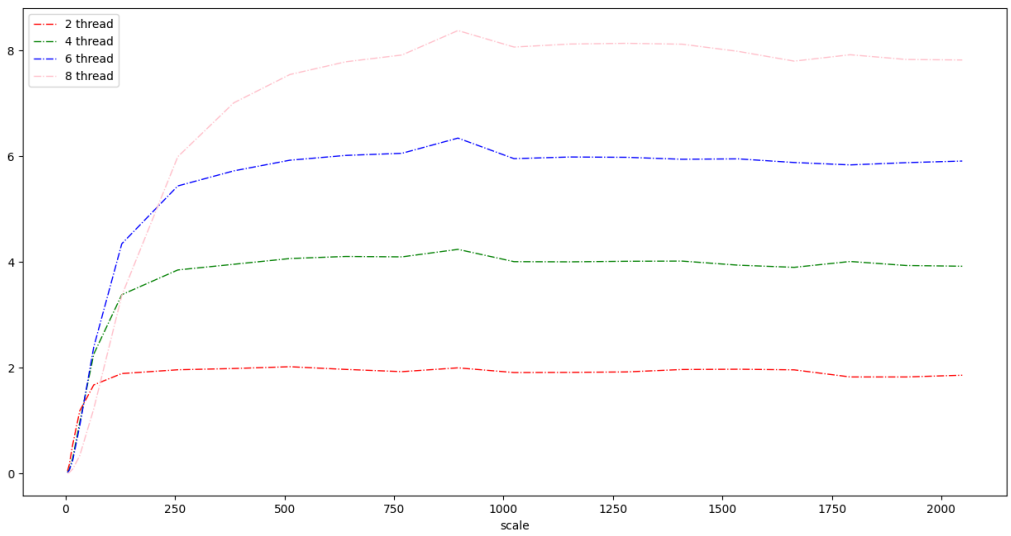

图 1.3: 不同线程数下 OpenMp 算法加速比

1.5.2 Part 2 探索 OpenMp 结合 SIMD

接下来我们探索 OpenMp 结合 SIMD 编程。首先我们在 Part1 的 OpenMp 算法的基础上结合 NEON 对算法进行优化，并将该算法与平凡算法和普通 OpenMp 算法运行时间进行对比，结果如图1.4所示。

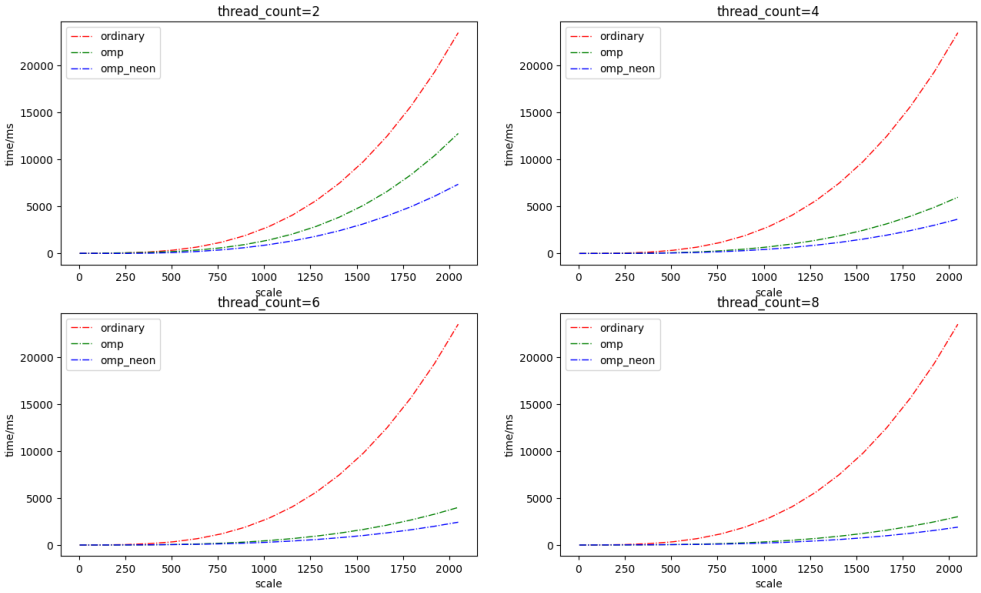

图 1.4: OpenMp 多线程算法结合 NEON 优化程序运行时间对比

我们得到了与 pthread 实验中类似的结果，在问题规模较大的情况下，OpenMp 多线程算法结合 Neon 后程序得到了较为明显的性能，显然，这是因为 SIMD 并行化提升了程序的 IPC。我们绘制 OpenMp 结合 Neon 优化算法的加速比，如图1.5所示，与图1.3进行对比，结合 SIMD 优化后的程序加速比有了明显的提升，各个线程数下程序均达到了超线性加速比。SIMD 编程是通过增大数据运算的吞吐量而提升程序性能，多线程编程则是通过增大指令执行的吞吐量而提升程序性能，两者结合可以起到更加明显的加速效果。

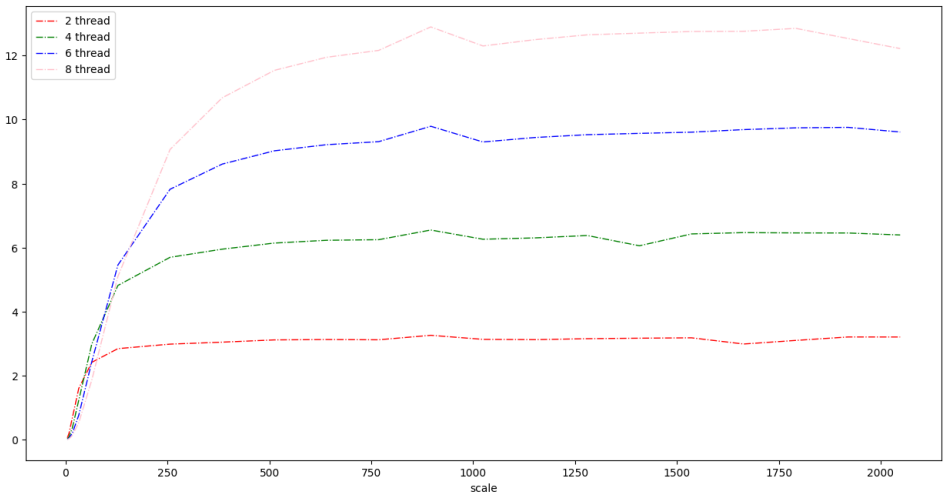

图 1.5: 不同线程数下 OpenMp 多线程结合 Neon 优化算法加速比

此外，我还进行了使用 #pragma omp simd 预处理指令进行程序自动向量化的 SIMD 优化方式。在此对其中遇到的问题及结果进行大致叙述。一开始我在程序中添加 #pragma omp simd 预处理指令后对程序进行编译运行，发现程序并未取得更好的性能，程序好像并未对我期望的代码进行向量化。经过上网查阅得知，使用 OpenMp 的自动向量化功能需要在编译时开启编译器的优化功能，而我一开始并未添加相应的编译选项。经过尝试，我发现鲲鹏服务器中的 g++ 编译器需要开启 O2 级别的优化才可以实现 OpenMp 的自动向量化功能。因此我添加-O2 编译选项，成功实现 OpenMp 的自动向量化，图1.6为线程数为 4 的情况下 OpenMp 结合 Neon 优化代码和 OpenMp 自动向量化代码运行时间对比结果。其他线程数下结果类似。可以看到，OpenMp 自动向量化代码具有更好的性能，再次使我感到 OpenMp 的强大。

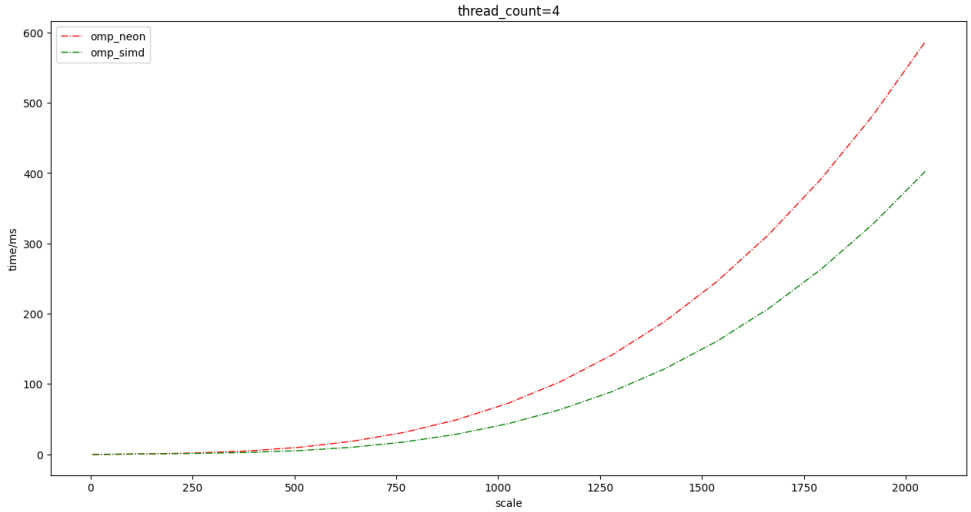

图 1.6: OpenMp 结合 Neon 优化程序与 OpenMp 自动 SIMD 优化程序运行时间对比

1.5.3 Part 3 OpenMp 不同任务分配方式 (static,dynamic,guided) 对程序性能的影响

本部分我们主要探究 OpenMp 不同任务分配方式对程序性能的影响。具体为固定线程数为 4，将 chunk 设置为 128，分别运行平凡算法、OpenMp 默认数据分配算法、OpenMp static 数据分配算法、OpenMp dynamic 数据分配算法以及 OpenMP guided 数据分配算法，对比它们运行所需的时间。运行结果如图1.7、1.8、1.9所示。

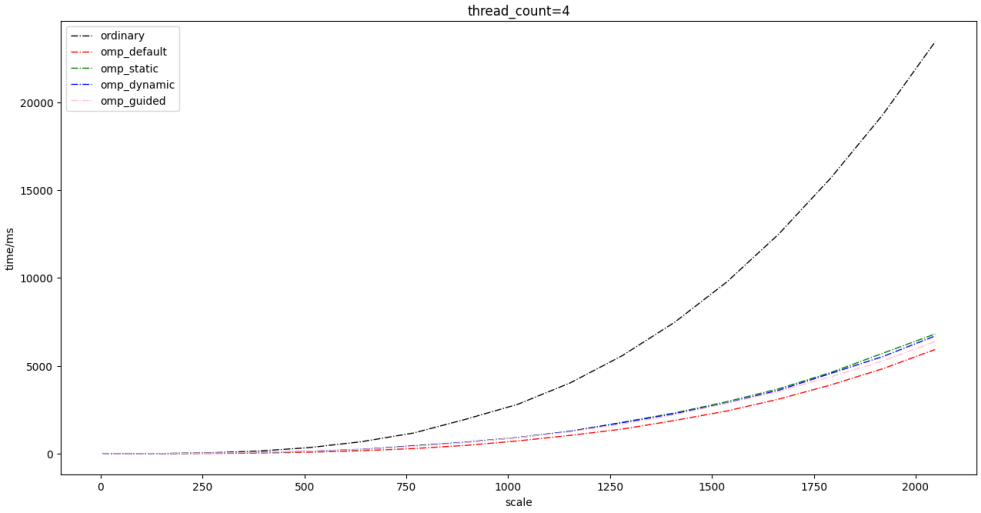

图 1.7: 不同任务分配方式算法运行时间对比

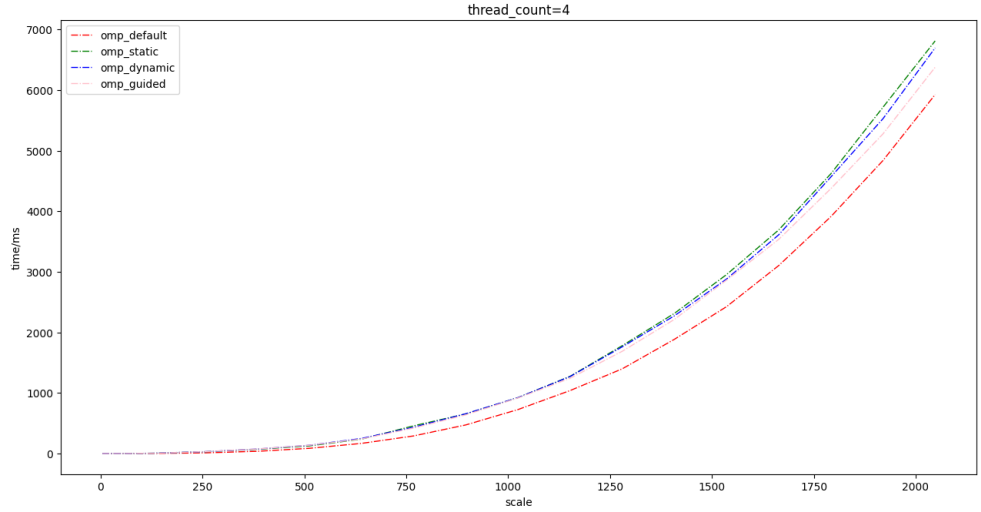

图 1.8: 不同任务分配方式算法运行时间对比

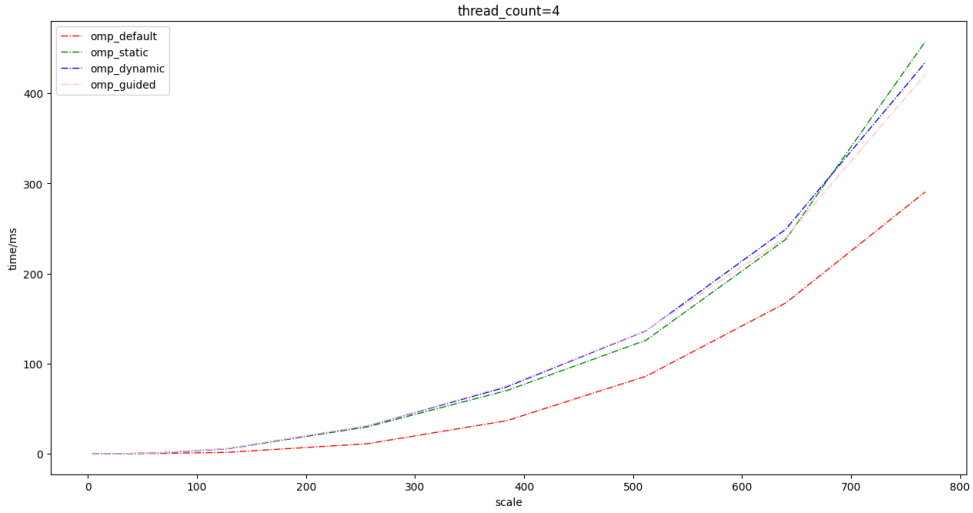

图 1.9: 不同任务分配方式算法运行时间对比

由图1.7我们可以看到，几种不同的任务分配方式相比于平凡算法均有较好的性能。

分析图1.8，我们发现，问题规模较大时，OpenMp 静态任务分配算法运行时间大于动态分配运行时间，动态任务分配算法运行时间大于 guided 任务分配算法运行时间。这是符合我们的直觉的，因为动态任务分配相比于静态任务分配数据具有更好的负载均衡，故而其运行时间要小于静态任务分配方式。而 guided 任务分配方式则均衡了动态分配的负载均衡特性和静态分配的通信开销小的特性，在三者中具有最小运行时间。但是为什么 OpenMp 默认的任务分配方式运行时间最短呢？思考原因，高斯消去过程中每次外层循环内，待消去的各个行所进行的消去任务是相近的，因此我们可以在最初的时候将待消去的行平均分配给每个线程，从而既具有较好的负载均衡，也大大减少了在线程间分配任务的开销。

接下来我们分析图1.9，我们发现，当问题规模较小时，OpenMp 默认任务分配方式算法运行时间仍然最小，这符合我们的上述分析。但是，与图1.8进行对比我们发现，动态任务分配算法和 guided 任务分配算法运行时间却大于静态任务分配算法运行时间，应该是此时动态分配算法和 guided 分配算法的通信开销较大，未能弥补他们带来的负载均衡方面的提升。同时我们可以看到，当问题规模大约大于 700 时，各个算法的运行时间开始与图1.8所示类似。

1.6 x86 平台实验结果呈现及分析

1.6.1 Part 1 OpenMp 结合 SSE 对程序性能影响

此部分我们编写基于 x86 平台的 OpenMp 结合 SSE 优化算法。并对比平凡算法、普通 OpenMp 算法、OpenMp 结合 SSE 优化算法的运行性能。我们固定线程数为 2、4、6、8，测试不同算法在不同问题规模下的运行时间如图1.10所示。

可以看到，x86 平台中普通 OpenMP 算法和 OpenMp 结合 SSE 优化算法均取得了较好的加速比，且 OpenMp 结合 SSE 优化算法具有更好的性能。

x86 平台中普通 OpenMp 算法和 OpenMp 结合 SSE 优化算法加速比如图1.11所示。可以看到，随问题规模的增大，两算法加速比大体呈现增加的变化趋势。且结合 SSE 优化的 OpenMp 算法明显具有更大的加速比，这符合我们的预期。

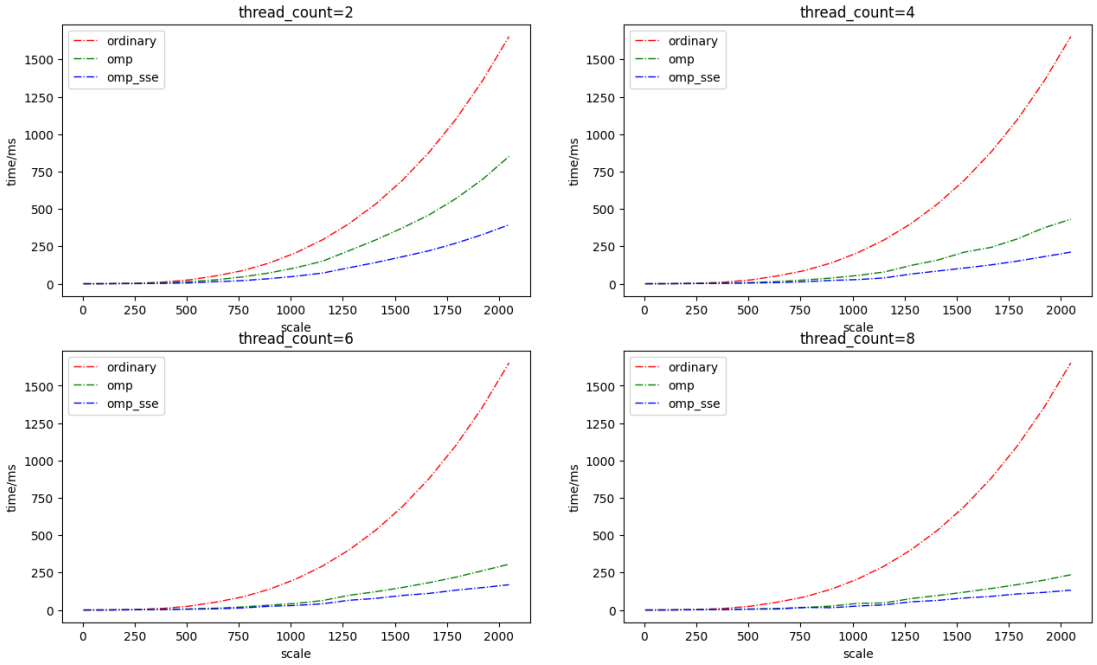

图 1.10: 不同算法运行时间对比

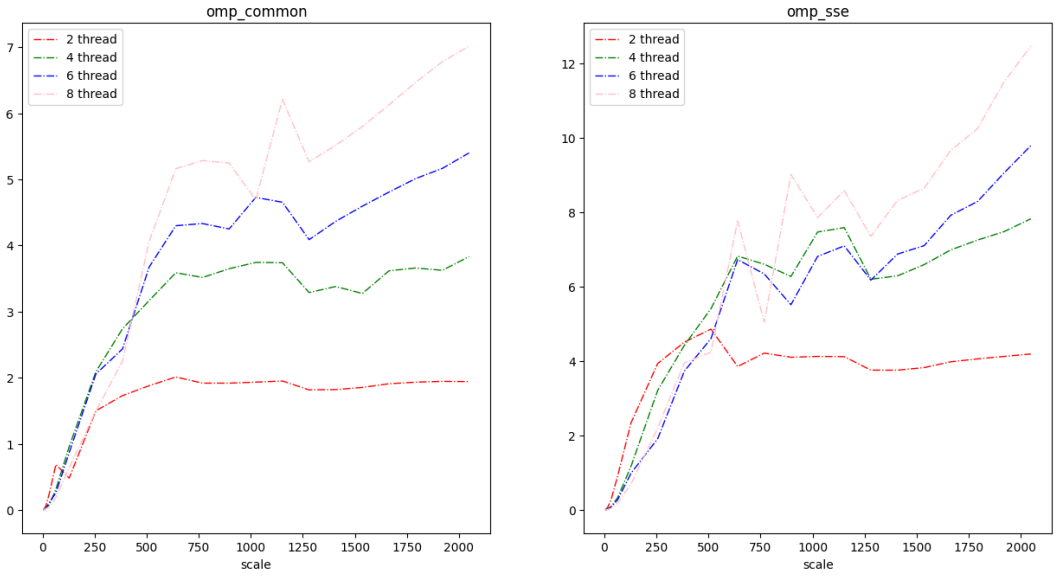

图 1.11: 普通 OpenMp 算法和 OpenMp 结合 SSE 优化算法加速比

1.6.2 Part 2 OpenMp 卸载到加速器设备

我们尝试探索使用 OpenMp 卸载到加速器设备。具体预处理指令及编译选项如下:

```
1 //预处理指令，我们设置8个team，每个team最多128线程，对高斯消去内层二重循环进行并行化
2 #pragma omp target teams distribute parallel for \
3                 num_teams(8) map(tofrom:A) thread_limit(128)
4
```

```
//编译指令，这里我们使用icpx编译器对程序进行编译
icpx -qopenmp -fopenmp-targets=spir64 gauss.cpp -o gauss
```

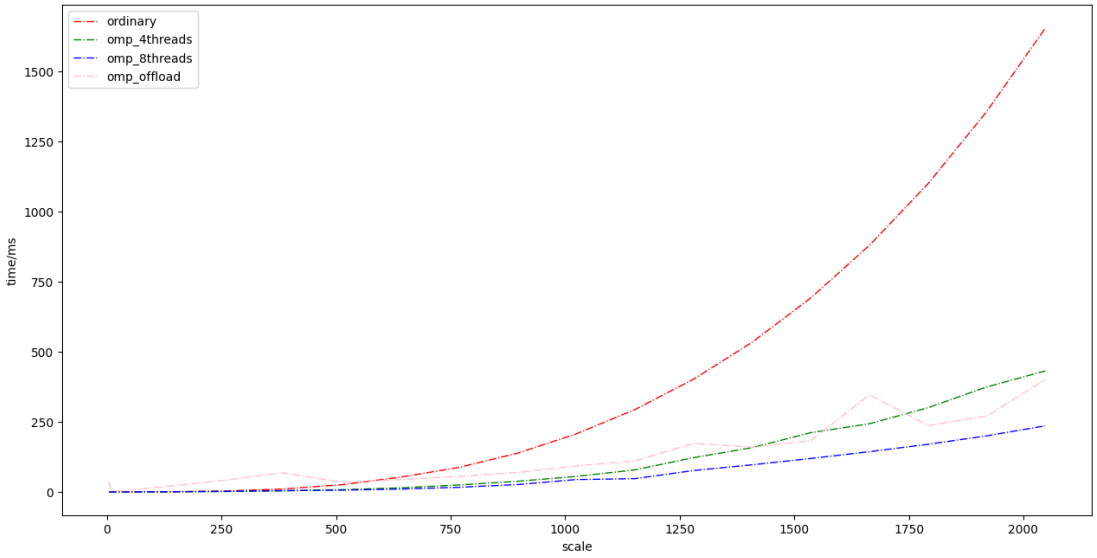

图 1.12: OpenMp 卸载到加速器运行时间对比

卸载到加速器设备算法的运行时间如图1.12所示。问题规模较大时，OpenMp 卸载到加速器算法具有与线程数为 4 的普通 OpenMp 算法相近的性能，且其运行时间始终大于线程数为 8 的普通 OpenMp 算法。令我感到惊讶的是问题规模较小时 Offload 算法所表现出的性能，如表1所示。可以看到当问题规模较小时，offload 算法运行时间甚至可以达到平凡算法近百倍。推测可能原因为 offload 算法需要在 host 和 device 之间进行数据的传输，且 IO 操作是较为消耗时间的，因此在问题规模较小时，数据传输带来的代价所占的比重较大，offload 算法未取得好的性能。

算法\规模	32	64	128	256	512
平凡算法	0.0065281	0.0502739	0.391502	3.16482	25.658
offload 算法	4.24884	8.78063	19.5201	41.495	34.523

表 1: 平凡算法与 offload 算法小规模下运行时间（ms）

由于我对 offload 功能及 GPU 架构认识浅显，此部分实验中使用的参数(num_teams(8)、thread_limit(128))可能不是十分的合理。后续我会继续学习该部分内容，希望可以得到更好的性能。

2 特殊高斯消去算法

2.1 实验介绍

在 pthread 多线程实验中，所编写的 pthread 程序并未取得好的性能提升。本实验将利用 OpenMp 编程对特殊高斯消去算法进行优化。并在 ARM 平台和 x86 平台上进行程序性能测试。

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

我们观察平凡算法以寻找算法中可以进行 OpenMP 多线程优化的部分。平凡算法如下所示，不难发现，平凡算法中只有第 8 到 10 行的对消元行消元的循环可以进行 OpenMp 优化，即让每个线程负责进行消元行中一部分数据的消元。但由于特殊高斯消去算法具有较为复杂的控制流，OpenMp 算法可能与 pthread 算法类似，均无法取得较好的性能提升。

```
//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;
        }
    }
}
```

具体优化算法请见文末 Github 链接。

2.3 实验环境说明

实验环境和普通高斯消去实验环境相同。ARM 平台为鲲鹏服务器，使用 g++ 编译器。x86 平台为 DevCloud 服务器，使用 icpx 编译器。

2.4 实验方案设计

实验将分别在 ARM 平台和 x86 平台进行。在两平台中分别测试 OpenMp 算法以及 OpenMp 结合 SIMD 算法的性能，并将其与平凡算法及 pthread 算法进行对比。实验将分别使程序在 2、4、8 线

程数目下运行给定的前 7 组测试数据，并对比不同算法的运行时间。

2.5 实验结果呈现及分析

实验结果数据如表2和表3所示。

算法\数据集编号		1	2	3	4	5	6	7
oridinary		0.00168126	0.0855458	0.11732	4.04767	24.465	521.816	6272.9
thread=2	pthread	0.357605	9.72246	8.21281	222.604	815.224	13045.1	82296.5
	OpenMp	0.0366877	1.38973	1.38923	38.5723	154.745	2085.42	14685.2
	OpenMp_simd	0.0365758	1.36024	1.36894	37.4309	150.936	1890.74	12622.1
thread=4	pthread	0.624288	16.0543	14.6927	385.573	1501.82	20018.7	133814
	OpenMp	0.0851278	3.29226	3.20804	86.733	332.274	3795.97	50298.5
	OpenMp_simd	0.0843615	3.30396	3.21848	87.484	334.6	3820.35	52077.2
thread=8	pthread	1.55593	50.111	48.8632	1327.55	5019.44	43941.2	251895
	OpenMp	0.163059	6.73288	6.50736	176.824	686.815	11857	69863.9
	OpenMp_simd	0.165388	6.70789	6.50338	177.092	749.663	11654.1	41401.1

表 2: ARM 平台下特殊高斯消去不同算法运行时间（单位：ms）

算法\数据集编号		1	2	3	4	5	6	7
oridinary		0.0016195	0.063741	0.086081	3.39692	24.019	253.158	2860.83
thread=2	pthread	0.823395	14.0153	16.247	277.182	1067.67	12769.5	80741
	OpenMp	0.0650811	1.13435	2.55243	72.9886	283.828	3436.17	21677.4
	OpenMp_simd	0.0299685	2.17192	2.3207	69.098	276.604	3211.51	21695.3
thread=4	pthread	1.39741	27.2938	32.426	511.07	1873.81	21466.9	134131
	OpenMp	0.0908785	3.07769	4.12242	110.175	425.243	4754.4	28155.7
	OpenMp_simd	0.0477417	4.03641	3.80731	104.446	422.511	4828.54	30641.7
thread=8	pthread	2.80016	51.9665	56.8114	2237.67	8497.78	91478.5	576468
	OpenMp	0.151287	3.47391	5.41607	153.64	505.347	5697.93	38692.6
	OpenMp_simd	0.0788958	3.21907	4.71191	120.32	485.075	5824.51	40395.9

表 3: x86 平台下特殊高斯消去不同算法运行时间（单位：ms）

我们看到，两平台中算法所体现出来的性能是相似的。与 pthread 算法类似的，OpenMp 算法也没有取得好的性能，但是 OpenMp 算法相比于 pthread 算法明显有更短的运行时间，可见 OpenMp 算法线程间的同步开销要更小一些。此外，相同测试数据下，随着线程数的增加，OpenMp 算法运行时间也在增加，这应该是因为线程数增加带来的同步开销过大导致的。令我感到惊讶的是，OpenMp 结合 SIMD 算法相比于普通 OpenMp 算法并没有取得较为明显的性能提升，甚至在某些情况下其运行时间略大于普通 OpenMp 算法的运行时间，对于这一现象，我还没有思考出可能原因。

3 实验源码项目 GitHub 链接

实验源码项目 GitHub 链接