南開大學
Nankai University

计算机学院

并行程序设计实验报告

Gauss 消去 openmp 并行优化研究

姓名：刘宇轩

学号：2012677

专业：计算机科学与技术

2022 年 5 月 16 日

目录

1 问题描述

在进行科学计算的过程中，经常会遇到对于多元线性方程组的求解，而求解线性方程组的一种常用方法就是 Gauss 消去法。即通过一种自上而下，逐行消去的方法，将线性方程组的系数矩阵消去为主对角线元素均为 1 的上三角矩阵。

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

考虑在整个消去的过程中，如图1.1所示，主要包含两个过程:

1. 对于当前的消元行，应该全部除以行首元素，使得改行转化为首元素为 1 的一行
2. 对于之后的每一行，用改行减去当前的消元行，得到本次的消元结果

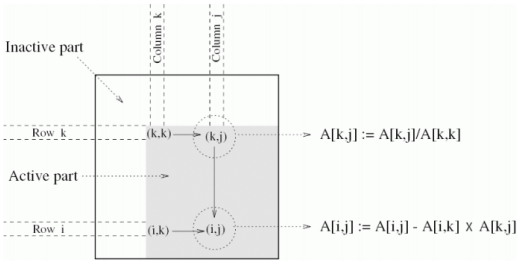

图 1.1: Gauss 消去算法逻辑图

而针对这两个过程，都适合采用并行的方式进行性能的优化。本次实验，采用 oenmp 多线程编程，结合 SIMD 并行指令架构，针对上述两个过程进行并行优化，并考虑采用不同的任务划分方式，对比不同的任务划分方式的性能差异，对比 openmp 的 SIMD 优化和手动的 SIMD 优化之间的差异，对比 openmp 多线程任务与手动 pthread 多线程之间的性能差异。此外还将考虑并行优化加速比随问题规模和线程数量的变化情况，本次实验还将同时设计 ARM 架构和 x86 架构两个平台的实验，对比在不同平台上多线程编程的性能差异。此外本次实验还尝试使用 GPU 进行

2 实验环境

	ARM	x86
CPU 型号	华为鲲鹏 920 处理器	Intel Core i7-11800H
CPU 主频	2.6GHz	2.3GHz
L1 Cash	64KB	48K
L2 Cash	512KB	1.25MB
L3 Cash	48MB	24MB
指令集	Neon	SSE、AVX、AVX512F
核心数	1	8
线程数	8	16

3 实验设计

考虑 Gauss 消去的整个过程中主要涉及到两个阶段，一个是在消元行行内除法过程，一个是其余行减去消元行的过程。而就每个阶段而言，其所做的工作基本是一致的，只是在不同的消元轮次时，消元的起始位置不同。尤其是针对第二个阶段，即其余行依次减去消元行的过程，这个阶段每一行所做的工作是完全一致的，十分适合并行化处理，即将待消去的行平均分配给几个线程，由于这些数据之间不存在依赖性，因此每个线程只需要各自完成好自己的工作即可，不存在线程之间进行通信的额外开销。

而对于第一阶段，即消元行行内进行除法操作时，由于这个问题规模相对较小，如果将待操作的数据分配给不同的线程进行处理的话，线程挂起到唤醒这部分的时间开销相较于要处理的问题而言占比很高，因此不适合进行多线程并行处理，但是仍可以结合 SIMD 的向量化处理。同样在第二阶段，被消元行依次减去消元行的过程中，每一行内的减法运算同样也不适合进行多线程的并行处理，也可以采用 SIMD 进行向量化处理。

在本次实验中，将设计以下实验进行探究:

1. 采用 openmp 对于 Gauss 消去的算法进行多线程优化，分析在不同任务规模下的性能表现
2. 设计不同的任务划分方式，调整 schedule 的参数，对比在不同任务规模下的性能表现
3. 对比不同的任务划分方式之间的性能差异，主要对比按行划分和按列划分
4. 对比在 openmp 下采用不同的线程创建方式的性能差异，主要对比动态线程创建和静态线程创建
5. 使用 openmp 进行 SIMD 向量化处理，并与之前手动 SIMD 的性能进行对比
6. 使用 openmp 进行多线程处理，并与之前手动 pthread 的性能进行对比
7. 对比在不同线程数量下的 openmp 的性能表现情况
8. 尝试将 openmp 优化方法从 arm 平台迁移到 x86 平台上，分析性能表现
9. 在 devcloud 上尝试任务卸载，将 Gauss 消元的消去过程卸载到 GPU 上运算，测试其性能表现

以下是详细的实验设计方案

3.1 openmp 并行处理

对于 Gauss 消去的过程，在每一轮消去中主要包含两个阶段，首先是针对消元行做除法运算，然后是对于剩余的被消元行，依次减去消元行的某个倍数。在每一轮的过程中，除法操作和消元减法操作之间是有着严格的先后顺序的，即必须首先完成消元行的除法操作之后，才能够执行被消元行的减法操作。因此在除法运算和消元之间需要进行以及每一轮次消元结束之后需要进行一次同步。

在考虑使用 openmp 进行多线程的实验设计时，考虑使用单个线程首先来处理消元行的除法操作。当这一行的除法操作完成之后，再将后续的消元过程分配给多个线程，并将循环变量全部声明为线程私有，将待消元的矩阵声明为线程间共享。

由于 openmp 还提供的 simd 的自动向量化预编译选项，因此在实验探究的过程中，还将探究单线程条件下使用了 simd 预编译选项后的性能表现，并且同之前手动设计的 SIMD 向量化算法的性能表现进行对比。此外还会将 openmp 设计的多线程程序和之前设计的 pthread 多线程程序进行性能对比，对比两种不同的算法设计的性能差异。

可以计算出采用了 openmp 方式的优化算法，其理论加速比应该和线程数量成正比，最优效果下应该能够达到 NUM_THREADS 倍的性能提升。而当同时结合了 SIMD 算法之后，这个加速比又将提升，并于向量化处理的宽度有关。当采用四路向量化优化时，整体的最优加速比应该能够达到 4 NUM_THREADS 倍。

3.2 数据划分设计

openmp 为我们提供了多种的任务划分方式，可以通过设置 schedule 中的参数选择不同的任务划分方式。针对不同的负载特性，可以考虑使用 static、dynamic 和 guided 这三种不同的任务划分方式。

最朴素的想法就是采用静态数据划分的方式，即在给各个线程分配任务的时候就已经确定好了每个线程负责的任务范围。这种任务划分方式，在有些情况下，即任务分布不均匀的时候会导致比较严重的负载不均。而在 Gauss 消去的过程中，由于每个阶段都是重新进行任务划分的，因此负载不均的问题并不明显，因而直接采用静态数据划分的方式也应该能够收获不错的效果。

而从负载均衡的角度出发，可以使用动态数据划分的方式。由于在每一轮消元的过程中，最终决定本轮消元时间的是运行时间最长的线程。因此可能存在个别线程提前完成任务而进入空闲等待浪费了计算资源。因此可以使用动态任务划分的方式，但可能会导致较大的额外线程调度开销。

由于高斯消元的过程中，任务的规模在不断减小，因此如果划分给每个线程的任务范围是固定的，就会导致随着任务的逐步推进，个别线程在某一轮消元的过程中没有被分配到任何任务，而其他线程则需要执行很多的任务，因此浪费了比较多的计算资源。因此可以使用 schedule 中提供的 guided 参数，随着任务推进，指数缩减到指定的步幅，这样就可以尽可能地利用上全部线程地计算资源。

3.3 行列划分设计

考虑不同的任务划分方式，对于 Gauss 消元的问题，可以考虑从行列两种划分角度进行实验设计。通常的，我们都是采用按行划分的方式，原因在于，在高斯消元的过程中，每一行的所需要做的操作是完全相同的，并且行内的数据是连续的，因此具有比较好的空间局部性。但是由于外层循环的迭代，每一轮各个线程所处理的行号可能是完全不同的，这将会导致在不同的消元轮次中，各个线程的 L1 和 L2cache 中不存在所需要的数据，而需要到其他线程的 cache 中去访问，这就造成了伪共享。而线程之间访问 cache 也是一个很大的开销，因此按行划分的方式会受到时间局部性的制约。

考虑按列划分的方式。在进行高斯消元的过程中，每一列所做的操作也是完全相同的，即将消元行该列的值依次减到该列中的其余位置处，而各列之间又不存在着数据依赖。因此也可以考虑按列划分的任务方式。但是由于在计算的过程中，每一个线程中的数据访问是跨行的，因此可能存在着比较大的几率导致 cache miss，因此这种方法受到空间局部性的制约。而由于 Gauss 消元的过程是逐渐向右下角收缩的过程，因此按列划分的方式也会受到时间局部性的制约。但是按列划分也是一种合理的尝试。

3.4 不同数据规模和线程数下的性能探究

考虑到线程的创建，调度，挂起和唤醒等操作相对于简单的计算操作而言，所需要的时间开销是非常大的。因此可以推测，当问题规模比较小的时候，由于线程调度导致的额外开销会抵消掉多线程优化效果，甚至还会表现出多线程比串行算法更慢的情况。而随着问题规模的增加，线程之间调度切换所需要的时间开销相对于线程完成任务所需要的时间而言已经占比很低，这样就能够正常反映出多线程并行优化的效果。因此，设计实验探究在不同数据规模下，多线程并行优化算法的优化效果。此外还将探究在所使用的线程数量不同的情况下，并行算法优化效果的变化情况。

3.5 x86 平台迁移

本次实验除了对 ARM 架构下采用 neon 指令集架构结合 pthread 多线程编程，对 Gauss 消去算法进行并行化处理，还将算法迁移到了 x86 平台上，采用 x86 中的 SSE、AVX 和 AVX512 指令集架构分别对算法进行重构，然后对比实验效果。

考察了本机所支持的指令集架构，情况如表1所示

指令集架构	版本
SSE	SSE/SSE2/SSE3/SSE4.1/SSE4.2
AVX	AVX/AVX2
AVX512	AVX512F

表 1: 本机支持指令集架构及版本

虽然本机只支持 AVX512F，但是 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。

在本次实验中还将使用 VTune 性能分析工具来分析 static、dynamic 和 guided 三种数据划分方式产生性能差异的深层原因

3.6 任务卸载尝试

Intel 的在线平台 devcloud 为实验提供了很好的平台，能够在在线平台上尝试使用 GPU 进行运算卸载。因此可以考虑将数据以来较小，运算密集的任务卸载到 GPU 上进行加速运算。

针对 Gauss 消元的问题，可以发现，对于每一轮循环，可以将消元的过程卸载到 GPU 中进行运算，针对每个 GPU 计算单元，需要将矩阵作为共享数据分发到每个 GPU 运算单元的内存中，而将其余的循环控制变量作为私有变量，防止不同的 GPU 运算单元之间相互影响。

因此，在设计实验的时候，在每轮循环中，首先让主线程去处理除法操作，然后将隔行的消元操作卸载到 GPU 上。对于卸载到 GPU 中的运算部分，可以使用 SIMD 进行向量化的处理，充分利用 GPU 中的矩阵计算优势。整体架构如下图3.2所示。

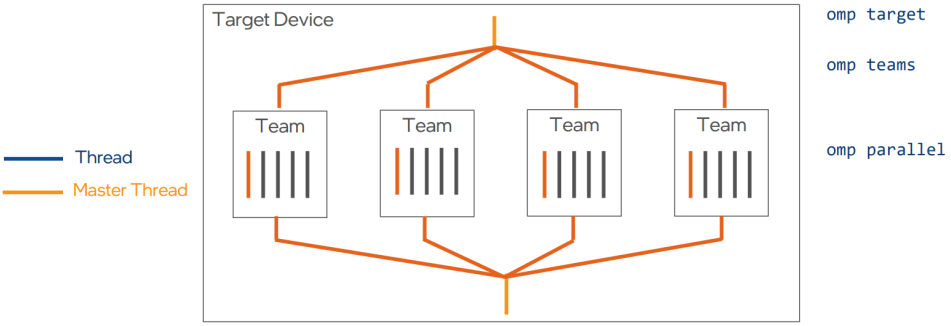

图 3.2: offloading 架构

4 实验结果分析

4.1 ARM 平台

4.1.1 openmp 并行实现及性能对比

为了能够探究 openmp 并行算法的优化效果，考虑调整问题规模，测量在不同任务规模下，串行算法，手动 SIMD 算法，手动 pthread 算法以及 openmp 版本的 SIMD 算法、openmp 版本的多线程算法的时间性能表现。其中 pthread 多线程算法和 openmp 多线程算法都和 SIMD 算法进行了结合，启用了 8 条线程，并采用了四路向量化处理。为了能够比较全面的展现并行优化效果随问题规模的变化情况，在问题规模小于 1000 时采用步长为 100，而当问题规模大于 1000 时，步长调整为 500。五种算法在不同问题规模下的表现如表2所示。

N	serial	SIMD	openmp_SIMD	pthread	openmp
100	2.66	1.72	2.04	2.87	1.08
200	20.98	13.41	15.75	6.88	3.75
300	71.00	45.02	53.04	14.99	9.35
400	169.70	107.47	126.63	29.22	20.87
500	340.58	214.38	252.67	51.55	39.08
600	597.55	375.19	441.19	83.77	61.69
700	952.12	595.74	702.35	122.31	99.93
800	1425.64	890.34	1041.39	176.05	145.94
900	2044.56	1263.94	1486.40	245.22	205.55
1000	2737.41	1711.53	2011.04	372.97	275.59
1500	9493.09	5741.21	6772.34	976.94	887.13
2000	23178.90	13670.60	16068.50	2215.51	2051.97

表 2: ARM 平台性能随问题规模变化情况

为了能够更加直观的观察算法的性能表现随问题规模的变化情况，特意利用测量的数据计算了四种并行优化算法的加速比随时间的变化情况，如图4.3所示。

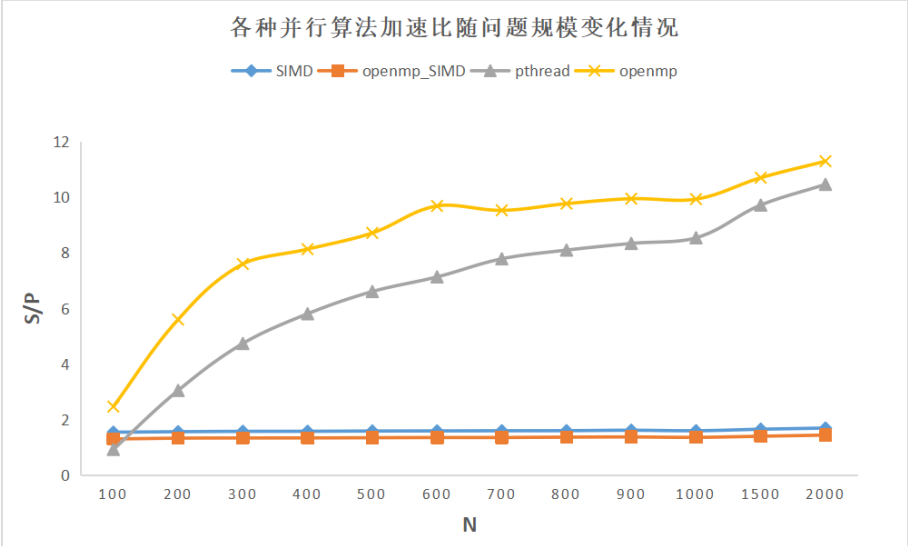

图 4.3: ARM 平台各种并行算法加速比随问题规模变化情况

从图像中可以看出，随着问题规模的增加，四种并行优化算法的加速比都呈现一个递增的趋势。由于在计算的过程中，出了乘法操作和消元操作，还有很多和分支跳转等不能够进行向量化处理的运算，因此采用 SIMD 思路进行优化的两种方法并没有能够接近理论加速比 4 倍，之保持在了一个 1.5 左右的加速比。而对于两种多线程的并行优化算法，其加速比随着问题规模的增加呈现快速增长的趋势，并且由于两种多线程的算法都与 SIMD 算法进行了融合，因此在评价其性能的时候可以用 SIMD 的性能作为 baseline。以此为参考时，可以发现，随着问题规模的增加，两种并行优化算法都接近了对 SIMD 的理论加速比 8 倍，这说明 Gauss 消元的问题是十分适合进行多线程优化的，原因是因为不同的线程之间在分配任务的时候并不存在严重的数据以来问题，因此线程之间额外的通信开销很少。

此外，实验还对比了手动的 SIMD 算法和 openmp 版本的 SIMD 算法之间的性能差异，从图像中可以看出，手动的 SIMD 算法要略优于 openmp 版本，通过 perf 来分析其性能，如图4.5所示。可以发现，openmp 版本的所需要的指令数 instructions 要明显高于手动 SIMD 的方法，并且在时钟周期 cycles 指标上也要高于手动版本，因此其性能表现更差。

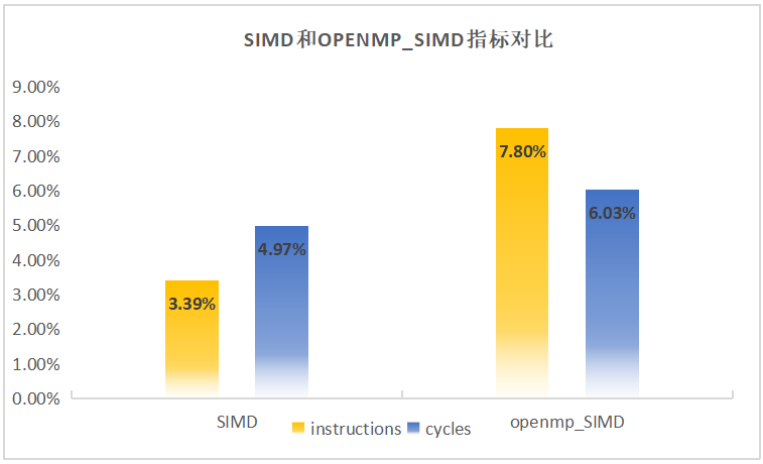

图 4.4: ARM 平台手动 SIMD 和 OPENMP 版本指标对比

同时，实验还对比了手动的 pthread 多线程和 openmp 多线程之间的性能差异，从图像中可以看出，手动的 pthread 算法其性能表现要一直差于 openmp，这是因为在手动实现 pthread 多线程算法的时候，由于 Gauss 消元的各个轮次之间存在着严格的先后关系，每一轮内部的除法操作和消元操作也都存在着严格的依赖关系，因此在手动实现多线程的时候，线程的同步开销要比 openmp 管理所需要的时间更多。

4.1.2 数据划分对比

在本次实验中，从负载均衡的角度触发，探究不同的任务划分方式和任务划分粒度对于算法性能的影响。openmp 在进行数据划分 schedule 的时候，为我们提供了三种不同的选项，即 static、dynamic 和 guided。static 方法是在线程创建完成之后，就明确划分了任务，dynamic 方法则是在线程执行的过程中去动态的划分任务，而 guided 方法则是随着任务的推进逐步缩减任务划分的粒度。在本次实验中，我们对比了这三种方法在不同任务规模下的性能表现，并将绘制出了加速比随问题规模的变化情况，如图所示。

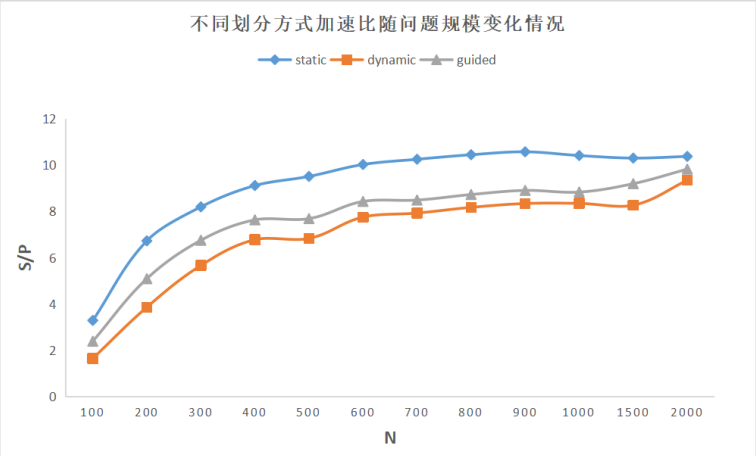

图 4.5: ARM 平台不同任务划分方式加速比随问题规模变化情况

从图像中可以看出，随着任务规模的增加，这三种划分方式的加速比都逐渐增加，而 static 方法首先达到了瓶颈，这是因为 10-12 左右就是 openmp 的一个最大理论加速比的范围，因此在 Gauss 消元这个问题上 static 的性能表现是最好的。原因是，在 Gauss 消元的过程中，每一轮都会重新进行任务划分，因此虽然整体上来看，矩阵的每个部分的计算量是不同的，但是在每一个消元轮次内，任务基本上是均匀的，因此不存在严重的负载不均的问题。而 static 和 guided 主要是为了解决这种负载不均的现象，为了平衡负载不均，必定要采用更细粒度的任务划分，因此会涉及到更多的额外条件判断以及通信开销，因此在 Gauss 消元问题上的表现不如 static 方法。

同样是考虑了负载均衡，由于 guided 方法会逐步缩减任务划分的粒度，尽可能让所有线程都被分配任务，而动态划分的方式可能出现随着任务推进，个别线程不会被分配到任务的情况，因此浪费了一定的计算资源。所以从图像中也可以看出，采用 guided 逐步调整划分粒度的方法其性能表现要比 dynamic 方法更优。

4.1.3 行列划分对比

在本次实验中，除了从负载均衡的角度出发，探究不同的粒度的任务划分方式以外，还从 cache 的角度出发，考虑空间局部性和时间局部性，探究了行列划分两种方式的性能表现，其性能表现如图4.6所示。

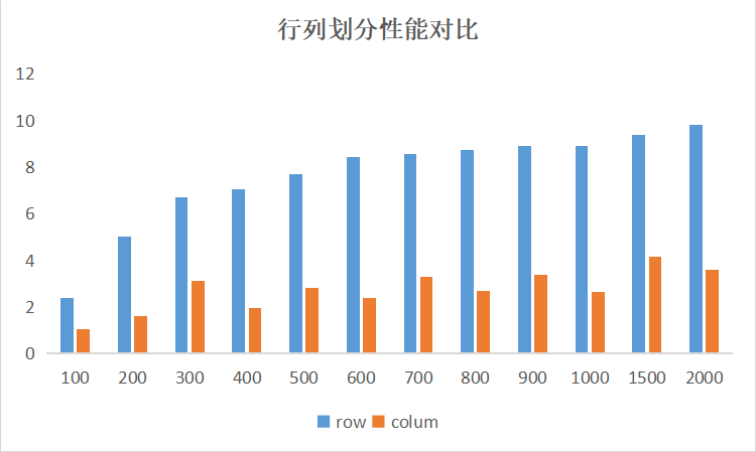

图 4.6: ARM 平台行列划分性能对比

从图像中可以很明显的看到，由于按列计算的方式，对于矩阵的访问来说是跨行访问的，因此随着任务规模的增加，由于空间局部性的限制，会产生比较严重的 cache miss，因此在访存的时候需要到内存中或者其他线程的 cache 中去访问，这会导致很严重的访存开销。通过 perf 分析两种方式的 L1 dcache miss 情况，如图4.8所示，这也能够很好的印证我们的推断。

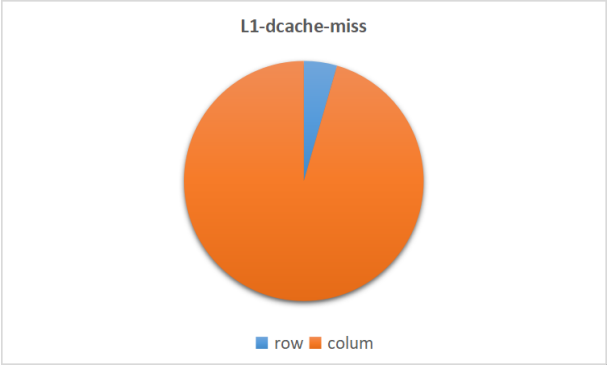

图 4.7: ARM 平台行列划分 L1-dcache-miss 总体占比

而从时间局部性考虑，这两种方法都会受到时间局部性的制约。这是由于随着任务的推进，任务执行的区域逐渐缩减，因此对于每个线程而言，其被分配的任务行号或者列号可能每次都不一样，因此就不能够利用上一轮计算中已经缓存到 cache 中的数据。因此可以从这个角度出发进一步改良算法。

4.1.4 线程数量对比

在本次实验中，探究了不同线程数量下，openmp 多线程算法的性能表现，其变化趋势大致如下图所示。从图像中可以看出，这三种划分方式的加速比都随着线程数量的增加呈现一个线性增加的趋势，说明 openmp 的多线程算法具有很好的扩展性。

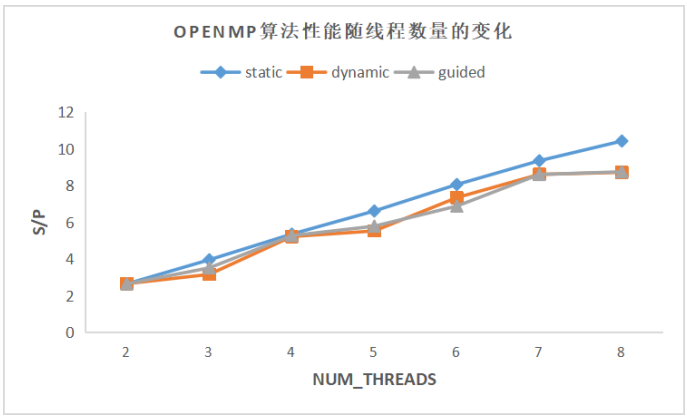

图 4.8: ARM 平台线程数量对性能的影响

4.1.5 线程管理对比

在本次实验中，探究了动态线程创建和静态线程创建这两种线程管理方式的性能对比。实验结果如表3所示。动态线程创建的方式就是在每一轮消元的过程中，重新创建线程，然后进行任务的划分，而静态线程创建的方式则是在最初一次性创建好全部线程，然后在每一轮中重新进行任务的划分。动态线程创建的方式适合于在不同的阶段需要处理完全不同的任务，而静态线程创建的方式适合于各个阶段所需要处理的任务近似相同。因此在本次实验中，更适合采用静态线程创建的方式，来减小线程

创建、初始化以及任务划分的额外开销，这种开销相对于简单的计算而言是比较大的。这也可以从实验数据中得到进行印证。

	static		dynamic	
N	**S/P**	**creat_time**	**S/P**	**creat_time**
100	2.12	3.35%	2.38	5.35%
200	4.73	3.21%	5.04	5.13%
300	6.48	2.89%	6.70	4.82%
400	7.16	2.54%	7.05	4.51%
500	8.07	2.21%	7.69	4.23%
600	8.67	1.98%	8.44	4.01%
700	9.03	1.50%	8.55	3.89%
800	9.26	1.32%	8.77	3.82%
900	9.24	0.89%	8.92	3.79%
1000	9.48	0.49%	8.90	3.68%
1500	9.67	<0.1%	9.40	3.62%
2000	10.41	<0.1%	9.89	3.56%

表 3: ARM 平台动态静态线程管理对比

4.2 x86 平台迁移

4.2.1 结合多种 SIMD 指令集架构对比

基于前文在 ARM 平台上对于 openmp 多线程编程的探究，在本次实验中还将 openmp 多线程优化方法迁移到 x86 平台上，做同样的实验探究。由于 x86 平台上拥有更多的 SIMD 指令集架构，因此实验中分别探究了 SSE、AVX 和 AVX512 三种指令集架构配合 openmp 多线程的优化效果，测量在不同问题规模下的运行时间，如表4所示。可以看出，openmp 多线程可以结合多种 SIMD 指令集架构，并且在各种指令集架构上的表现基本保持稳定，并没有出现在某种指令集架构下不能够发挥很好的多线程优势的现象。

N	**SSE**	**AVX**	**AVX512**	**omp_SSE**	**omp_AVX**	**omp_AVX512**
100	1.86	3.31	6.25	1.45	2.58	4.87
200	1.87	3.32	6.28	3.98	7.08	13.38
300	1.88	3.34	6.31	6.05	10.77	20.35
400	1.88	3.35	6.33	7.81	13.89	26.26
500	1.87	3.33	6.30	9.02	16.06	30.35
600	1.85	3.30	6.23	9.89	17.61	33.28
700	1.87	3.33	6.29	10.65	18.95	35.82
800	1.87	3.32	6.28	11.26	20.04	37.87
900	1.87	3.33	6.30	11.61	20.66	39.04
1000	1.87	3.33	6.30	11.70	20.83	39.36
1000	1.87	3.33	6.29	11.93	21.23	40.13
1500	1.88	3.34	6.31	12.96	23.06	43.59
2000	1.88	3.34	6.32	13.44	23.93	45.22
2500	1.86	3.32	6.27	13.96	24.84	46.95
3000	1.88	3.34	6.31	14.03	24.97	47.19

表 4: x86 平台不同 SIMD 指令集架构性能对比

从数据中可以看出，随着问题规模的增加，SSE、AVX 和 AVX512 三种指令集架构的加速比均保持在一个比较稳定的水平上，这说明向量化的优化已经达到了一个瓶颈，而之所以没有能够达到理论

加速比是因为整个程序并不能完全进行向量化的展开，因此其余未能够向量化的部分极大的影响了整体的加速比。

而横向对比同一指令集架构下的 SIMD 和结合了 SIMD 的 openmp 方法，则可以发现，由于在实验中总共拉起了 8 条线程，因此随着问题规模的增加，当问题规模达到 3000 时，这三种指令集架构的 openmp 算法相对于 SIMD 算法的加速比已经能够逼近 8 倍了，说明在 devcloud 平台上，多线程的性能能够得到充分的释放。

4.2.2 数据划分对比

openmp 为任务划分提供了三种选项：static、dynamic 和 guided。static 方法是在线程创建完成之后，就明确划分了任务，dynamic 方法则是在线程执行的过程中去动态的划分任务，而 guided 方法则是随着任务的推进逐步缩减任务划分的粒度。在 x86 平台的实验中，我们对比了这三种方法在数据规模为 1000 时的性能表现，并将使用 VTune 性能分析工具对于这三种方式的 CPU 占用时间进行监测，得到对比如图4.9所示。

图 4.9: x86 平台数据划分方式 CPU 占用对比

从图中可以看出，static 方式的 CPU 占用时间分配十分的不均衡，这也体现了这种方法会受到负载不均的制约，因此其运行时间较长，为 178.4ms。在本次实验中，dynamic 方式的 CPU 占用时间分布较为均衡，而且通过实验测量程序的运行时间可以发现 dynamic 方式所消耗的时间最少，为 169.2ms。对于 guided 方法，在本次实验中并没有能够表现出很好的性能，其消耗的时间略高于 dynamic 方式但仍低于 static 方式，说明其在一定程度上可以减轻负载不均的影响。

4.3 GPU 任务卸载

在本次实验中还尝试了在 Devcloud 平台上进行 GPU 运算卸载，将 Gauss 消元的消元过程卸载到 GPU，利用 GPU 运算单元加速运算，具体代码如下。

```
  void calculate_openmp_offloading(float * buffer)
{
    float * buf = buffer;
    {
        int i, j, k;
        for (k = 0; k < N; k++) {
            {
#pragma omp master
                {
                    for (j = k + 1; j < N; j++) {
                        buf[k*N+j] = buf[k*N+j] / buf[k*N+k];
                    }
                    buf[k*N+k] = 1;
                }
#pragma omp target map(tofrom: buf[0:N*N]) map(to: N)
                {
#pragma omp teams num_teams(32)
                    {
#pragma omp distribute
                        {
                            for (i = k + 1; i < N; i++) {
#pragma omp parallel for simd default(none), private(j), shared(buf, N, i, k)
                                for (j = k + 1; j < N; j++) {
                                    buf[i*N+j] = buf[i*N+j] - buf[i*N+k] * buf[k*N+j];
                                }
                                buf[i*N+k] = 0;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

在调整数据规模进行多次实验之后，发现 offloading 方式的性能表现甚至差于普通的串行算法。分析其原因，可能是因为在不同消元轮次之间和每一轮中除法操作和消元操作之间都存在这比较严重的数据依赖，因此会造成 GPU 各个运算单元之间需要不断地通过 CPU 进行通信和同步，而这种通信的开销是非常大的。因此当数据规模比较小的时候，会严重的影响性能。笔者猜测这种优化的效果可能会在数据规模很大的情况下，并且尽可能减少数据依赖后体现出来。

5 总结

在本次 openmp 多线程并行实验中，基于 Gauss 消元问题，对于过程中的消元操作采用 openmp 多线程优化，并在线程内进行除法或者减法运算时，结合 SIMD 向量化处理，在 ARM 平台上采用 openmp+neon 的方式，在 x86 平台上采用 openmp+SSE/AVX/AVX512 的方式，探究了其并行优化效果。除此之外，还基于 cache 的空间局部性和时间局部性对比了行划分和列划分的性能差异，基于负载均衡考虑对比了 openmp 提供的三种 schedule 方式的性能差异，通过实验可以验证，考虑了 cache 特性的行划分方式和考虑了负载均衡的动态划分方式均能够取得一定的性能提升。实验还探究了开启线程的数量同优化性能之间的关系，并结合实验平台的硬件参数进行合理假设和分析。在实验的过程中，利用 perf 和 VTune 等性能分析工具，对于深层次的内核和硬件事件进行分析，从底层的角度解释了表面上性能差异的原因。本次实验中第一次尝试了通过 offloading 的方式将部分运算卸载到 GPU 上进行加速，但是由于数据依赖导致的通信开销，并未能够取得很好的效果，后续会对相关的问题进行进一步研究和优化。本次实验的相关代码和文档已经上传至GitHub。