南開大學
Nankai University

计算机学院

并行程序设计期末报告

K-means 并行优化算法研究

姓名：刘宇轩

学号：2012677

专业：计算机科学与技术

2022 年 7 月 10 日

目录

1 引言

K-means 聚类算法是机器学习中无监督学习的原型算法，其主要思想为: 首先在数据集中随机确定 k 个初始聚类中心，将剩下的数据划分到距离它最近的聚类中心形成 k 个簇，然后求出每个簇的均值作为新的聚类中心，重新分配其他样本，重复这一过程直到聚类中心不再变化为止。与其他聚类算法相比，K-means 算法最显著的优点是其原理简单、算法容易实现、聚类收敛速度快，为此 K-means 算法更适合处理大型数据，但是也有一些不足之处 [**?**]，例如:

1. K-means 算法随着数据规模的增加和维度的升高，收敛速度以及聚类的准确性都会受到影响
2. 聚类结果依赖于聚类中心的初始化，易陷入局部最优
3. K-means 算法对噪声和孤立的数据非常敏感，导致聚类结果中心偏移

在期末的实验中，将重点围绕如何采用并行化的优化方式，提升 K-means 算法在大数据规模下的收敛速度，考虑采用多种并行方法在多种实验平台上进行测试。

2 问题描述

当前处于大数据的时代，数据量爆发式增加，而对于海量的大数据，很多时候，数据之间的类别关系是很难人为做出明确的划分的，这就需要我们能够找到一种高效可行的分类算法。对于没有明确分类标签，只是规定了类别的数量或者是并没有划定类别数量的问题，这就属于聚类问题，也就是要根据数据之间的特征关系，通过对于数据相似度的计算，将数据划分成几个类别，实现一个无标签的分类过程。

K-Means 聚类算法，是一种无监督的学习方式，试图在没有任何先验知识的情况下推断数据中的模式。正是由于没有任何先验知识，因此聚类算法是根据数据的属性将数据分为多组的任务，更确切地说，是基于数据中明显的某些模式。其采用的思想便是，通过计算各个数据点之间的相似性，将数据划分成多个类别，以实现在各个类别内部，数据点之间的相似性尽可能高，而在各个类别之间的相似性要尽可能低。

2.1 算法描述

对于给定的 n 组数据 $\{x_1, x_2, \cdots, x_n\}$，每个数据属性的维度均为 d 维 $x(a_1, a_2, \cdots, a_d)$，通过计算数据之间的相似度，将这 n 个数据划分成为 K 组。假设这 K 组分类记为 $S = \{s_1, s_2, \cdots, s_k\}$，$K$ 组分类中心的集合记为 $C = \{c_1, c_2, \cdots, c_k\}$。

2.1.1 聚类中心的计算

$$c_i(a_1, \cdots, a_d) = \frac{1}{n_i} \sum_{x_k \in c_i} x_k(a_1, \cdots, a_d)$$

2.1.2 相似度计算

1. 绝对距离

$$L = \sum_{i=1}^{K} \sum_{x_j \in c_i} |x_j(a_1, \cdots, a_d) - c_i(a_1, \cdots, a_d)|$$

2. 欧几里得距离

$$L = \sqrt{\sum_{k=1}^{K} \sum_{x_i \in c_k} \sum_{j=1}^{d} (x_i(a_j) - c_k(a_j))^2}$$

3. 平方误差

$$L = \sum_{i=1}^{K} \sum_{x_j \in c_i} ||x_j(a_1, \cdots, a_d) - c_i(a_1, \cdots, a_d)||^2$$

2.1.3 算法流程

对于给定的 n 个数据，要将其划分为 K 组，首先要初始化聚类中心。在本次实验中采用的是随机初始化的方式，虽然可能会导致局部最优解的出现，但是由于实验主要探究的是 KMeans 算法的并行优化效果，为了能够更好的对比不同优化方式的效果，在这里依旧选择随机初始化的方式。

根据初始化的 K 个聚类中心，依次计算所有节点到这个聚类中心的相似度，为了能取得相对准确的聚类结果并且更好的并行化计算处理，我们选用欧氏距离来估计各个数据点到聚类中心的相似性。对于每个节点而言，选择将其暂时划分入距离他最近的聚类中心中，形成一次聚类的划分。

对于新形成的 K 个聚类，重新计算每一个分组中的聚类中心的位置，更新所有聚类中心的位置，然后重复上述操作，重新计算所有数据点到这些新形成的聚类中心的相似度，再将节点划分入距离他最近的聚类中心中。

不断重复上述操作，直到聚类中心的位置不再变化，或者前后两次聚类中心的位置变化小于给定的阈值，便可以退出循环，得到聚类结果。

算法伪代码如图2.1所示

Algorithm 1: K-Means clustering algorithm

```
Input: Dataset, K, Tolerance
Output: Mean table, Assignment of each datapoint to a
  cluster
Initialize
Assign data to nearest cluster
Calculate new means
if Converged then
    Output assignment of data
return;
else
    Continue from step 4 with new means
```

图 2.1: 算法伪代码

在整个算法的执行过程中，我们可以通过下图2.2来感受一下聚类执行的过程。以二分类问题为例，在算法开始时首先会选择两个初始的聚类中心，然后根据各个节点到这两个聚类中心的距离，将节点划分入不同的分组中，得到图 (3)。然后根据新生成的分组，重新计算出聚类中心的位置，得到图 (4)。重复上述的过程，直至聚类中心的位置不再发生显著变化，即聚类结果收敛，最终得到了两个分组划分，如图 (9) 所示。

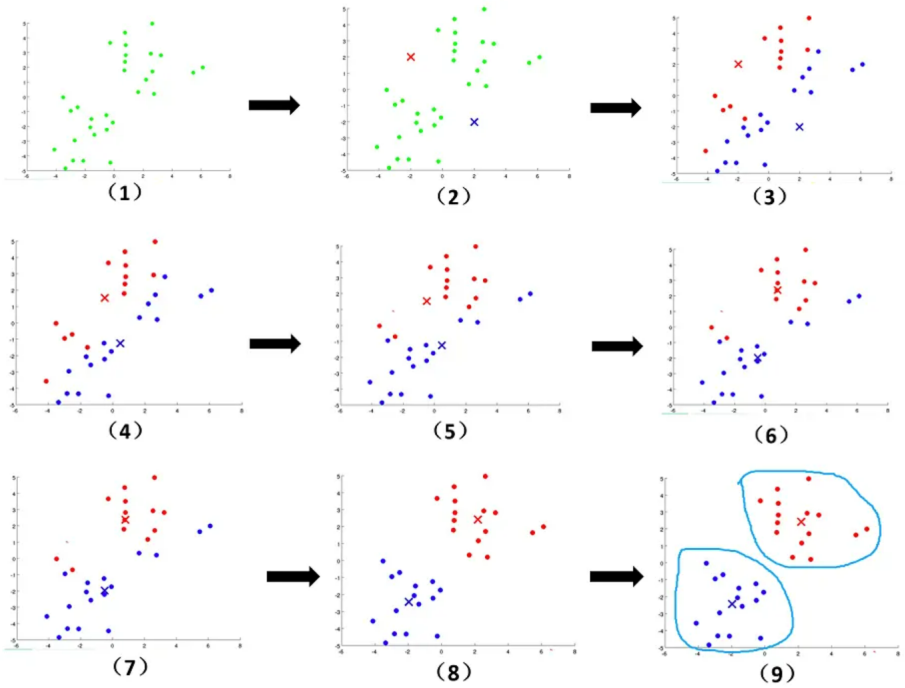

图 2.2: 执行过程

3 工作综述

在本学习学期中的学习过程中，我跟随课程进度，完成了 Gauss 消元的不同并行优化策略的探究。有关 KMeans 的并行优化策略探究的全部工作，都是在期末完成的，也就是说本学期完成了 Gauss 消元和 KMeans 两个算法的并行优化策略探究。

对于期末选题 KMeans 的并行优化，结合本学期的课程所学，分别设计了指令级 SIMD 的并行优化策略，线程级 pthread 和 OMP 的并行优化策略，进程级 MPI 的并行优化策略，以及基于 CUDA 的 GPU 并行优化策略。在不同的优化策略下，对比了该方法下采取不同的 trick 进行小幅度改进的提升效果。并且在 ARM 和 x86 两个平台进行了性能测试。主要工作如图所示

	ARM/x86
SIMD	设计SIMD算法
	对比内存对齐
	对比预取优化
pthread	设计pthread多线程算法
	对比动态数据划分和静态数据划分
	与SIMD结合
OMP	设计OMP算法
	对比不同数据划分方式
	对比动态和静态线程管理
	与SIMD结合
MPI	设计MPI算法
	对比不同数据划分方式
	对比主从和从从模式
	非阻塞通信优化
	与SIMD结合
	与OMP结合
GPU	设计CUDA算法

图 3.3: 工作综述

3.1 SIMD 并行优化

根据课程所学的知识，对于 KMeans 算法中距离计算以及质心更新两个阶段进行了 SIMD 多路向量化处理，并且根据不同的平台，选取了不同的指令级架构，分别设计了 ARM 平台的 Neon 和 x86 平台的 SSE、AVX、AVX512。在算法实现的过程中，加入了一些优化 trick，例如考虑内存对齐减少数据合并打包，考虑数据预取提高访存效率。

3.2 pthread 并行优化

根据课程所学的知识，对于 KMeans 算法中数据划分聚类的过程进行多线程并行优化，不同的线程分别处理一部分数据，在实现了 pthread 的基础上，增加了一些优化 trick，例如从负载均衡的角度考虑动态数据划分和静态数据划分之间的差异，考虑 pthread 是线程级的并行可以融合 SIMD 增加指令级的并行。

3.3 OMP 并行优化

根据课程所学的知识，对于 KMeans 算法中数据划分聚类的过程进行多线程并行优化，不同的线程分别处理一部分数据，在实现了 OMP 的基础上，对比了不同数据划分方式的性能差异，并且对比了动态线程管理和静态线程管理之间的开销。同样考虑 OMP 是线程级的并行可以融合 SIMD 指令级的并行。

3.4 MPI 并行优化

根据课程所学的知识，对于 KMeans 算法中数据划分聚类的过程以及质心更新的过程进行多进程并行优化，不同的进程处理一部分数据以及质心的更新任务。在实现了 MPI 的基础上，从负载均衡的角度出发考虑了不同数据划分方式，并且对比了主从模式和从从模式的性能差异，增加了非阻塞通信的 trick，考虑到 MPI 是进程级的并行，可以融合 OMP 线程级并行和 SIMD 指令级并行。

3.5 GPU 并行优化

根据课程所学的知识，对于 KMeans 算法中数据划分聚类的过程以及质心更新的过程进行 GPU 并行优化，使用 CUDA 实现了基础的并行优化算法。

4 实验环境

	ARM	x86
CPU 型号	华为鲲鹏 920 处理器	Intel Xeon Processor
CPU 主频	2.6GHz	2.6GHz
L1 Cash	64KB	32K
L2 Cash	512KB	1MB
L3 Cash	48MB	24MB
指令集	Neon	SSE、AVX
核心数	1	1
线程数	1	1

5 SIMD 并行优化

5.1 实验设计

考虑到 KMeans 算法的迭代过程中，主要包含两个阶段，第一阶段是对于每一个数据点计算其到各个质心之间的相似度，第二阶段是每一轮迭代完成前，需要根据本轮的划分结果，重新更新质心。这两个过程都会涉及到类矩阵运算，二对于这类运算，是十分适合使用 SIMD 进行指令级的并行优化的。

对于第一阶段而言，在本次实验中，相似度的计算都是使用欧氏距离来衡量的，而对于欧式距离的计算，是可以采用多路向量化展开提高并行效率的，即可以一次取出同一个数据点在多个维度上的数值，与某一个质心做差之后再求平方，然后使用规约加法将最终的结果累加。对于第二阶段，当更新质心的时候，同样需要先将划分给某个质心的数据点各个维度的值求均值，这个过程同样适合进行多路向量化并行。

考虑到当数据的内存地址对齐方式与所采用的指令级架构不符时，会出现额外的数据打包操作，因此本次实验中还增加了内存地址对齐的 trick，并且对比了其性能提升效果。考虑到当数据规模增大的时候，可能会导致数据访问很难在 L1-cache 命中，这样就会导致额外的访存开销，因此在本次实验中还设计了增加内存预取的 trick，并且对比了其性能提升效果。

在本次实验中，将设计以下实验进行探究:

1. 采用 SIMD 对 KMeans 算法进行多路向量化并行优化，分析在不同任务规模下的性能表现
2. 对比内存对齐优化方法的性能提升效果
3. 对比内存预取优化方法的性能提升效果
4. 尝试将 SIMD 优化方法从 x86 平台迁移到 arm 平台上，分析性能表现

以下是详细的实验设计方案

5.1.1 SIMD 并行处理

考虑第一阶段计算数据点和各个质心之间相似度的过程，可以一次取出多个维度的数值，然后和某一个质心相应维度的值做向量减法，然后再将这个结果平方，将平方后的结果采用向量的横向加法进行归约，并累加规约结果。由于此时数据维度的遍历步长变成了向量化的并行路数，因此需要注意结尾的剩余项数的处理，这里统一采用串行的方式进行处理。

考虑第二阶段更新质心的过程，首先将质心各个维度的数据清零，然后循环遍历每一个数据点的划分标签，将这个数据点的各个维度的数值累加到其所属质心上，在这个累加的过程中，可以采用多路向量化的并行方式，一次累加多个维度的数值。所有的数据点累加完成后，需要求出各个划分标签的平均值作为新的质心，这个求平均值的过程也可以向量化处理，即取出某个质心多个维度的数据，采用向量除法除以属于该标签的数据点的个数。

同样对于不同的 x86 和 ARM 不同的实验平台，采用了不同的指令级架构，并且在 x86 平台上，对比了不同的指令级架构下，采用不同的向量化路数所取得的性能优化差异。

5.1.2 内存对齐性能优化

当采用向量化的数据存取的时候，CPU 会一次性取出多条数据，当内存地址按照向量位宽对齐的时候，CPU 只需要一次就能够取出所需要的全部数据，而当内存地址未按照向量位宽对齐的时候，

CPU 就要通过两次访存拿到所需要的数据，并将这份数据打包成一个向量返回，这样不仅造成了额外的访存开销还有额外的指令操作。因此考虑增加内存对齐的优化方法，减少这种额外的开销。

考虑到内存对齐的方式有两种，一种是动态的内存对齐，即首先在行首串行处理未对齐的数据，然后向量化处理对齐的数据，最后进行尾处理；另一种是静态的内存对齐，即在程序最开始就首先将数据的每一行都按照向量位宽对齐。动态内存对齐的方式，可能每一行的对齐都不一致，导致代码实现难度高，并且执行的过程中也会造成很多的控制流判断，因此实验采用静态内存对齐的方式，即根据所选用的指令级，提前将数据的每一行都按照向量位宽对齐。

5.1.3 内存预取性能优化

考虑到当问题规模增加的时候，尤其是在数据的维度增加的时候，会导致一行数据可能不能在 L1-cache 中完全存下，或者是下一行要访问的数据并不在缓存中，这样就需要到更高级别的 cache 中甚至内存中去获取数据，这会导致额外的访存开销，并且这种访存开销相对于计算过程是比较大的。因此考虑增加内存预取进行访存优化。

当每计算完一个数据点到最近的一个质心的划分后，在循环内提前将下一行数据预取到内存中，虽然此时可能多了一条指令的周期，但是由于 CPU 的流水线机制，其额外的时间开销并不明显。并且相较于访存的额外开销，这样的指令开销还是相对很小的。

5.1.4 平台迁移

考虑到 SIMD 在不同平台有不同的指令集架构，因此本次实验中在 x86 平台设计了基于 SSE、AVX 和 AVX512 的 SIMD 并行优化算法，在 ARM 平台设计了基于 Neon 的并行优化算法，并在两个平台都测试了在不同数据规模下，算法的性能表现，以及增加了内存对齐 trick 后的性能提升效果。

5.2 实验结果分析

5.2.1 SIMD 并行处理

考虑到在我们的实验数据中总共有两个影响因素，一个是数据点的数量 N，另一个因素是数据的维度 D，因此当考虑并行算法的优化效果的时候，需要综合考虑这两方面的因素。为了能够充分显示出 SIMD 并行优化的效果，在实验中将数据点的个数 N 设置在 [500,4000] 的区间内，将数据维度 D 设置在 [500,4000] 的区间内。为了能够消除收敛速度的影响，迭代轮次统一选择迭代 500 次。

在 x86 平台上设计了 SSE、AVX 和 AVX512 三种并行优化算法，并且分别调整数据点的数量 N 以及数据维度 D，得到实验结果如图5.4所示。

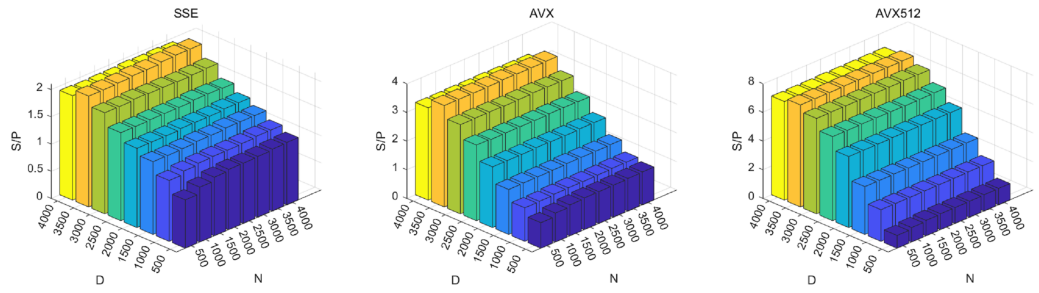

图 5.4: x86 平台 SIMD 实验结果

从柱状图中可以看出，随着数据维度的升高，三种并行优化算法的加速比均得到了一定程度的提高，SSE 算法最高能够达到 2.11 的加速比，AVX 算法最高能够达到 3.57 的加速比，AVX512 算法最高能够达到 7.16 的加速比。随着数据维度的增加，这个加速比呈现一个上升的趋势，这是由于 SIMD 多路向量化展开是在数据维度的角度去优化的，如计算相似度以及更新质心，都是一次处理多个维度的数值，因此随着数据维度的增加，这个并行的效果会更加明显，而三种算法都没能够达到理论加速比，是因为出了计算相似度和更新的运算外还有一些判断是否是最近点对的以及循环迭代等其他的运算，因此加速比会低于理论加速比。

从柱状图中可以看出，随着数据点数量的增加，三种算法的性能提升效果并不明显，只有小幅度的增加。这是由于在 SIMD 向量化处理的时候并没有从数据点数量 N 的角度进行优化，因此加速比不会随着 N 的变化有明显变化。

对于这三个算法，提取 $N = 2000$ 时的加速比随数据维度 D 变化的切片，如图5.5所示。

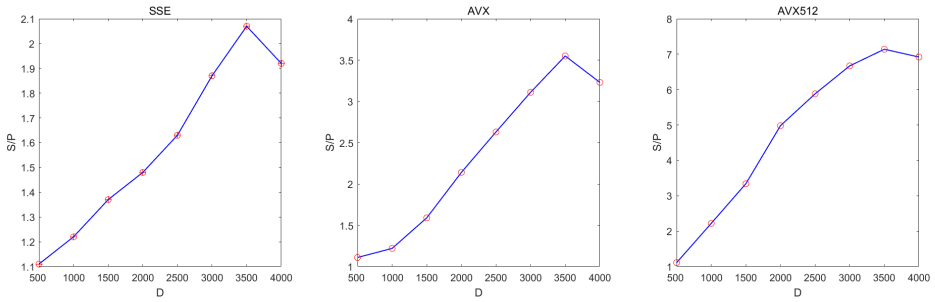

图 5.5: 不同算法加速比随数据维度 D 的变化趋势

从图像中可以看出，随着数据维度的增加，加速比呈现先上升后下降的趋势，分析其原因可以推测，由于数据维度的增加，导致一行数据就可能填满 L1-cache，因此在访问下一行数据的时候，就会导致不能够在 L1-cache 中命中，需要到更高级别的 cache 中去访存，因此造成了额外的访存开销，也在一定程度上影响了。在实验中使用了 VTune 性能分析工具，以 SSE 算法为例，分析了其各级 cache 的命中率随着数据维度的增加的变化情况，如表1所示。可以看到当 $D = 4000$ 的时候，算法在 L1 的命中率出现了小幅下滑，L2 的命中率则产生了较大的下降，这也是其加速比出现下降趋势的原因。

N	**500**	**1000**	**1500**	**2000**	**2500**	**3000**	**3500**	**4000**
L1-cache hit	>99	>99	>99	98.8	98.8	98.6	98.6	98.1
L2-cache hit	>99	>99	98.5	97.8	97.8	97.2	97.4	88.9
L3-cache hit	98.2	98.2	98.3	98.2	98.1	98.4	98.2	98.2

表 1: SSE 各级 cache 命中率随数据维度变化

在实验中，还对比了采用 SSE、AVX 和 AVX512 这三种 SIMD 指令级架构的算法的性能表现，以及其加速比随数据维度 D 的变化情况，如图5.6所示。从图中可以看出，三种算法都能够随着数据维度 D 的增加逐渐上升，但是均低于理论加速比，并且相对而言，向量化路数越多，其算法加速比的增长速度也就越快，可见其具有比较良好的伸缩性。

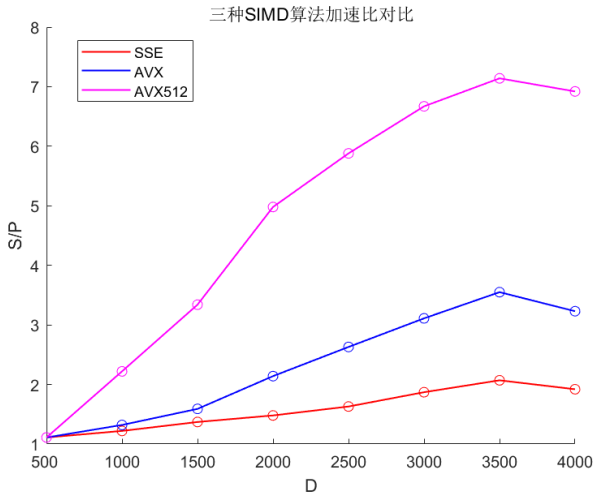

图 5.6: 不同算法加速比随数据维度 D 的变化趋势

选取数据点数量 $N = 2000$，数据维度 $D = 3500$，通过 VTune 性能分析工具，分析了三种算法的指令数量和时钟周期数，结果如图5.7所示。通过图像可以发现，三种算法所需要的指令数大致相同，而时钟周期数随着向量化路数的增加减少，这是因为循环迭代的步长变大了，这大幅减少了时钟周期的数量。因此推测算法的性能与所需要的时钟周期数存在正相关。

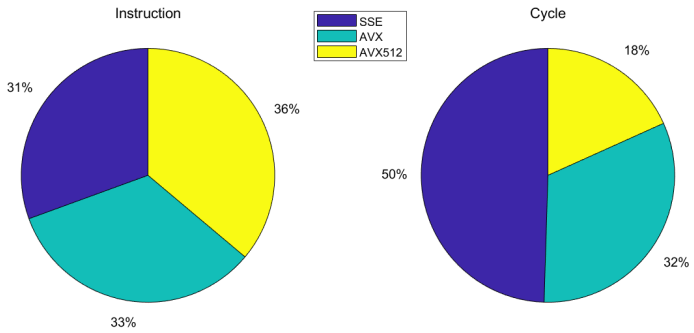

图 5.7: 不同算法指令数和时钟周期数对比

5.2.2 内存对齐优化

考虑到，由于 SIMD 在进行数据存取的时候是打包操作的，即如果内存地址是按照向量化路数对齐的，则可以一次将全部数据取出，而如果内存地址未对齐，则需要通过两次访存拿到所需数据，然后将这个数据打包之后返回个程序使用，这回造成额外的访存开销和指令开销。在本次实验中，对比了内存对齐和内存未对齐两种算法的性能差异，并探究了其在不同数据维度下的性能表现，实验结果如图5.8所示。

从图中可以看出，采用了内存对齐策略的算法能够取得小幅度的性能提升，三种算法在进行了内存对齐优化之后，大概都获得了 1.1 左右的性能优化。分析其原因应该就是通过内存对齐，减少了数据打包造成的额外开销，这包含额外的访存开销和额外的指令开销。在实验中使用了 VTune 性能分析工具，从 CPU 事件角度进行分析，具体数据如表2所示。从表中能够看出，采用了内存对齐策略后，能够在一定程度上降低指令数和时钟周期数，并且减少额外的访存开销。

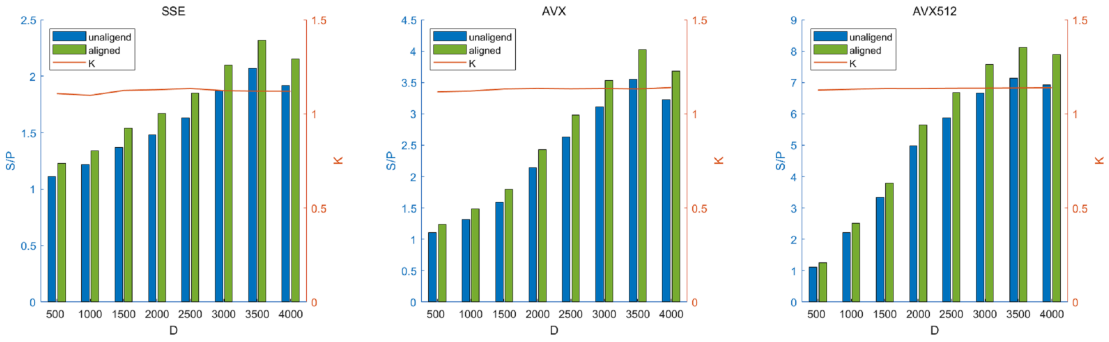

图 5.8: 内存对齐优化对比

	unaligned	aligned
L1-cache hit	98.6	98.8
L2-cache hit	97.4	98.1
L3-cache hit	98.8	99.2
Instructions	1.23E+9	1.18E+9
Cycles	2.47E+10	2.35E+10

表 2: 内存对齐策略 CPU 事件对比

5.2.3 内存预取优化

考虑到当数据的维度增加时，可能会导致接下来需要访问的数据无法存放到缓存中，这时只有当发生数据在缓存中未命中的时候才会去访存，而这个时候访存时 CPU 出于空闲等待状态。而如果增加了内存预取策略，则可以提前将下一行要访问的数据取到内存中，这个访存过程中，CPU 还能够执行其他的运算操作，而不需要完全空闲等待。实验设计了增加内存预取优化策略的算法，并对比了其性能提升效果，这里以 AVX 算法为例展示实验结果，如图5.9所示。

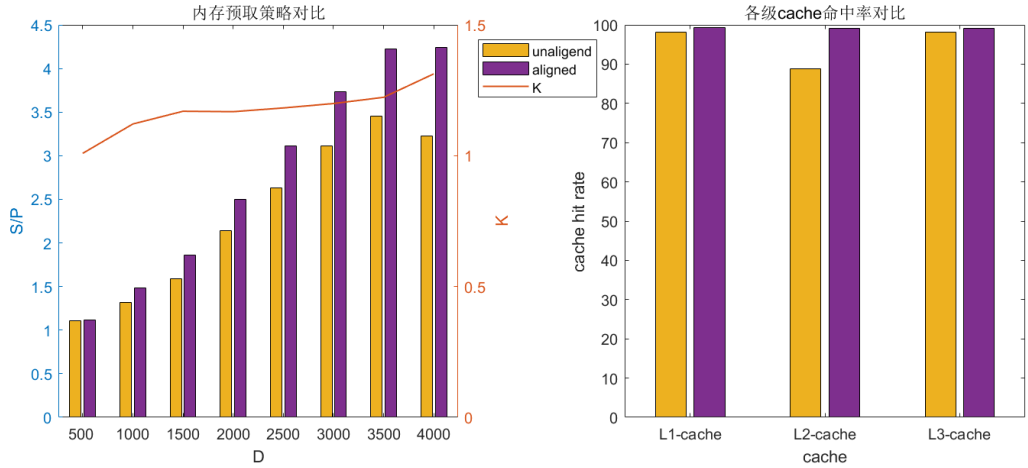

图 5.9: 内存预取优化对比

通过图像观察可以发现，随着数据维度 D 的增加，采用了内存预取优化策略的算法具有更好的延展性，由于访存开销更小，因此其加速比整体上个小幅度的提升，保持了较为稳定的增长趋势。也可以发现，增加了内存预取策略后，优化算法在各级 cache 中的命中率都要高于未优化之前。

5.2.4 平台迁移

在本次实验中，除了完成了在 x86 平台的实验外，还尝试在 ARM 平台使用 Neon 指令集进行 SIMD 算法设计，完成了 KMeans 在计算相似度和质心更新两阶段的 SIMD 并行优化。但由于 ARM 下的 Neon 指令集只支持四路向量化，因此实验也只围绕着这一种优化方式展开探究。

同样重复了在 x86 平台上的测试实验，在实验中将数据点的个数 N 设置在 [500,4000] 的区间内，将数据维度 D 设置在 [500,4000] 的区间内。为了能够消除收敛速度的影响，迭代轮次统一选择迭代 500 次。测试了基于 Neon 的 SIMD 算法的性能表现，得到和 x86 平台相似的结果，如图5.10所示。从图像中可以看出随着数据维度 D 的增加，算法的加速比一直呈现一个上升的趋势，当数据维度 $D = 4000$ 的时候，加速比能够达到 2.24。并且由于 ARM 平台上的 L1-cache 更大，因此其并没有出现因为数据维度增加导致 cache 未命中而降低性能。通过 perf 性能分析工具也能够证实，在 $D = 3500$ 和 $D = 4000$ 两个点上的 L1-cache 命中率几乎相同。

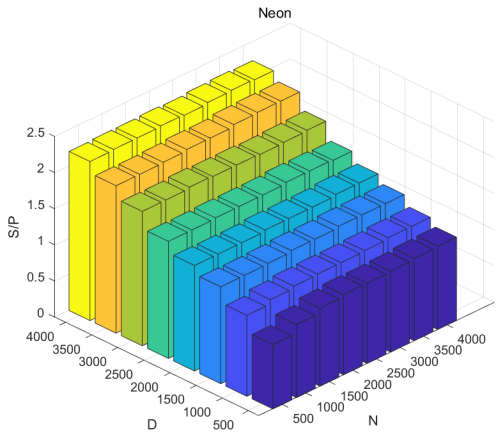

图 5.10: ARM 平台 SIMD 实验结果

在 ARM 平台上同样进行了内存对齐的优化算法设计，在申请内存的时候，将内存地址按照 16 字节对齐，保证了数据每行开始都是按照 4 路向量对齐的。在本次实验中，对比了内存对齐和内存未对齐两种算法的性能差异，并探究了其在不同数据维度下的性能表现，实验结果如图5.11所示。从图中可以看出，采用内存对齐算法后，加速比能够得到小幅度的提升，整体来看，内存对齐算法能够提升 11% 的性能。

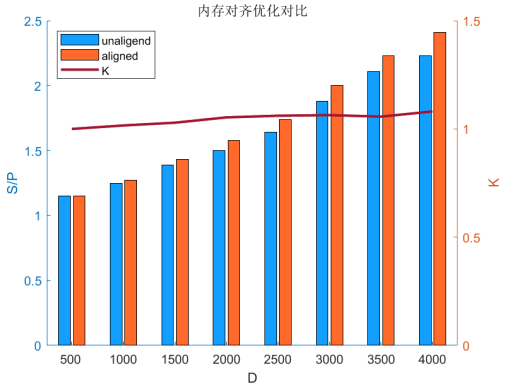

图 5.11: ARM 平台内存对齐优化对比

6 pthread 并行优化

6.1 实验设计

考虑到 KMeans 算法的迭代过程中，主要包含两个阶段，第一阶段是对于每一个数据点计算其到各个质心之间的相似度，第二阶段是每一轮迭代完成前，需要根据本轮的划分结果，重新更新质心。对于第一个过程而言，由于数据点的数量很多，因此可以将数据点平均分配给多个线程去执行相似度的计算和聚类划分的过程。而对于第二个阶段，由于聚类的数量 K 般不会太多，因此在本次实验中不考虑将这个阶段进行多线程的并行化处理。

对于第一阶段而言，可以启用多条线程来同时计算数据点的相似度和聚类划分。根据线程的标号，确定每一个线程所负责的任务范围，由于在本次实验中，每一个数据点的计算量大致相同，因此可以直接考虑使用块划分的方式，即给每个线程按照线程编号划分连续的一段数据。所有的线程计算完相似度并完成聚类划分之后，会进行一次同步，同步完成后，0 号线程会更新质心，然后就可以进行下一轮迭代。

考虑到由于各个线程可能计算的速度不同，并且每一行数据的计算难度也可能不同，因此虽然整体来看每一行数据的计算量大致相同，但是也可能会存在小幅度的负载不均，在实验中考虑采用动态数据划分的方式来提高负载均衡。而对于每一个线程而言，其计算数据点和质心之间相似度的过程，仍然适合采用 SIMD 向量化并行，因此实验中还设计了 pthread 和 SIMD 相结合的算法。

在本次实验中，将设计以下实验进行探究:

1. 采用 pthread 对 KMeans 算法进行多线程并行优化，分析在不同任务规模下的性能表现
2. 对比动态数据划分优化方法的性能提升效果
3. 对比线程内部增加 SIMD 向量化优化方法的性能提升效果
4. 测试在不同线程数量下算法的性能表现
5. 尝试将 pthread 优化方法从 x86 平台迁移到 arm 平台上，分析性能表现

以下是详细的实验设计方案

6.1.1 pthread 并行处理

对于每一轮迭代而言，主要包含两个阶段，第一阶段是对于每一个数据点计算其到各个质心之间的相似度，第二阶段是每一轮迭代完成前，需要根据本轮的划分结果，重新更新质心。本次实验设计中，主要针对第一个阶段进行多线程并行优化。

考虑到在进行聚类算法的时候，涉及到的数据点一般都会比较多，因此可以将数据点平均划分给多个线程。各个线程都会有一个线程的 ID 用以标识，每个线程根据自己的 ID 确定负责的任务范围，然后并行计算数据点和质心之间的相似度，并完成聚类的划分。在每个线程完成了聚类划分之后，所有线程会进行一次同步操作。此时 0 号线程要负责根本轮的聚类划分结果，重新计算质心。完成了质心的更新之后，就可以开始下一轮的迭代。

完成了 pthread 算法的实现之后，将会在平台上测试在不同数据点数量以及数据维度下的性能表现；此外还会测试对于相同数据点数量以及数据维度下，采用不同线程数量的性能表现。

6.1.2 动态数据划分优化

从整体上来开，对于 KMeans 算法而言，每一个数据点在每一轮的计算量大致是相同的，但是仍可能存在线程之间的计算能力存在差别，数据点间的计算难度不同而导致最终的计算时间产生细微差别，而当数据量十分大的时候，这种差别就会更加明显。因此从负载均衡的角度出发，考虑在数据划分的时候，抛弃原来的静态划分方式，改用动态划分的方式。

全部线程都共享同一个任务标签，这个标签标识着当前所有的线程已经处理到了哪一行数据。当某一个线程完成之后，将会从任务标签标识的行开始，连续获取 t 行数据，并将这个任务标签进行修改。由于任务标签是全部线程共享的，因此为了能够保证多线程算法的正确性，当某一个线程准备获取新的任务时，需要首先对这个任务标签上锁，防止此时其他线程同时来操作任务标签，导致算法错误，当任务或去完成、修改了任务标签之后，再释放这个同步锁。

整个算法从负载均衡的角度出发，会在一定程度上提升算法的时间性能。但由于引入任务标签，导致了更多的线程通信和同步操作，因此可能在任务规模较小的情况下，这种算法并不能表现出很好的性能，但随着任务规模的增大，算法的性能提升会逐渐表现出来。

6.1.3 与 SIMD 结合优化

对于每个线程而言，在线程内计算数据点和质心之间相似度的过程并不需要按照传统的串行方法来执行，同样可以参考在SIMD 并行处理一节中采用的 SIMD 向量化并行处理，在计算相似度的时候，一次性计算多个维度的数值并做向量归约加法。在更新质心的时候，同样不需要采用串行的方法，也可以使用 SIMD 进行向量化并行优化。在本次实验中，便将 pthread 多线程算法和 SIMD 向量化优化相结合，并结合不同的平台架构选择了不同的指令集架构进行实现。

6.1.4 平台迁移

在本次实验中，不仅尝试了在 x86 平台实现 pthread 算法，并且还将算法迁移到了 ARM 平台上，由于单纯使用 pthread 算法在两个平台之间没有任何差异，只是与 SIMD 算法相结合的时候，在不同的平台需要选择不同的指令集架构，在 x86 平台使用了 SSE、AVX 和 AVX512 三种指令集架构实现算法，在 ARM 平台使用了 Neon 指令集架构实现算法。

6.2 实验结果分析

6.2.1 pthread 并行处理

考虑到在我们的实验数据中总共有两个影响因素，一个是数据点的数量 N，另一个因素是数据的维度 D，因此当考虑并行算法的优化效果的时候，需要综合考虑这两方面的因素。为了能够充分显示出 pthread 并行优化的效果，在实验中将数据点的个数 N 设置在 [500,4000] 的区间内，将数据维度 D 设置在 [500,4000] 的区间内。为了能够消除收敛速度的影响，迭代轮次统一选择迭代 500 次，线程数量选择使用 8 条线程。

在 x86 平台上完成了 pthread 算法的实现，并且分别调整数据点的数量 N 和数据维度 D，得到实验结果如图6.12所示。从柱状图中可以看出，当数据点数量很少的时候，算法的加速比小于 1，这是由于线程的调度开销很大，导致了其带来的性能提升被线程调度抵消。但随着数据点数量的增加，pthread 算法的加速比逐渐上升，而由于在实验中拉起了八条线程，因此算法的理论加速比为 8。但是由于循环迭代以及其他的运算并没有进行加速，因此理论加速比会略低于 8. 在实验中，加速比一直在上升，在

实验数据覆盖范围内的加速比最高达到了 4.36。而从数据维度的角度来看，算法的加速比并没有明显的变化，

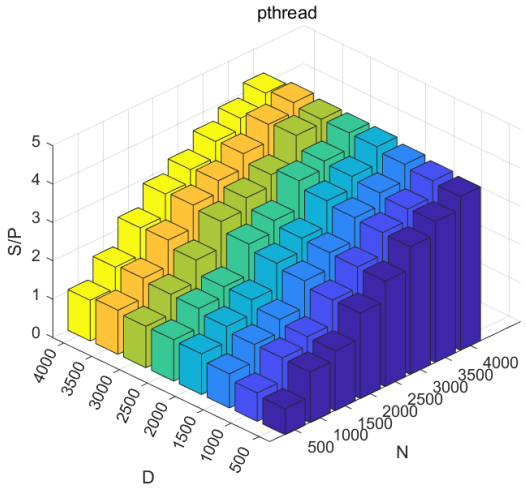

图 6.12: x86 平台 pthread 实验结果

在实验中，分别选取了 $D = 4000$ 和 $N = 4000$ 两个切片，如图6.13所示，可以直观地看到加速比随着数据点数量以及数据维度的变化情况。

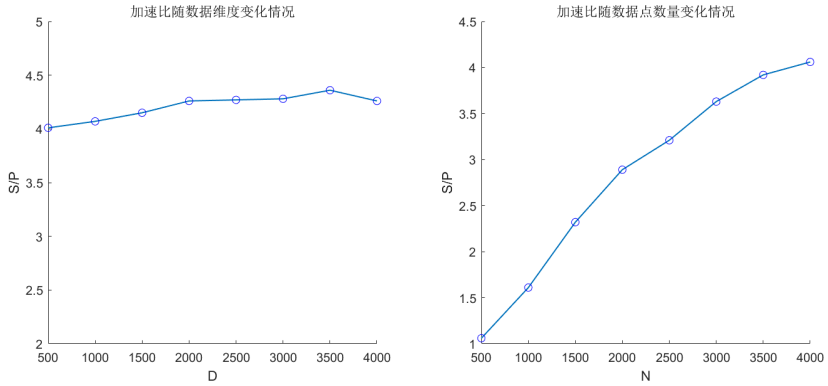

图 6.13: x86 平台 pthread 单变量实验结果

从图像中可以看出，加速比随着数据维度的增加变化不明显，这是因为多线程的优化是从数据点数量的角度出发的，即将数据点平均分配给多个线程，增加数据维度对于算法的性能表现影响不大。但是也能够看到，随着数据维度的增加，加速比呈现了先上升后下降的现象，这是由于数据维度的增加，导致一行中的数据过多填满了 cache，程序在访问下一行数据的时候不能够在 cache 中命中而造成了额外的访存开销。通过 VTune 分析也能够证实，在 $D = 3500$ 时的 cache 命中率要高于 $D = 4000$ 时的命中率，如表3所示。而从数据点个数的角度出发，可以发现加速比随着数据点的增加逐渐上升，但是增速逐渐放缓，推测时逐渐逼近了理论加速比。这主要是因为随着数据点数量的增加，多线程的性能提升效果逐渐显现，抵消掉了线程调度产生的额外开销。

	$D = 3500$	$D = 4000$
L1-cache hit	98.8	98.1
L2-cache hit	98.4	97.2
L3-cache hit	99.2	98.9

表 3: 不同数据维度下 CPU 事件对比

6.2.2 动态数据划分优化

实验从负载均衡的角度出发，设计了动态数据划分的优化策略，即任务不是程序一开始就指定的，而是在运行的过程中，根据各个线程实际的工作情况分配的，工作效率高的线程会处理更多的任务，而工作效率低的线程处理任务较少。这样能够最大限度利用每条线程的计算资源，避免了由于负载不均导致的计算资源浪费。在本次实验中，对比了采用动态任务划分的方式和静态任务划分方式的性能差异，为了能够最大程度显示出两种算法的差异，选定数据维度 $D = 2000$，调整数据点数量进行试验，结果如图6.14所示。

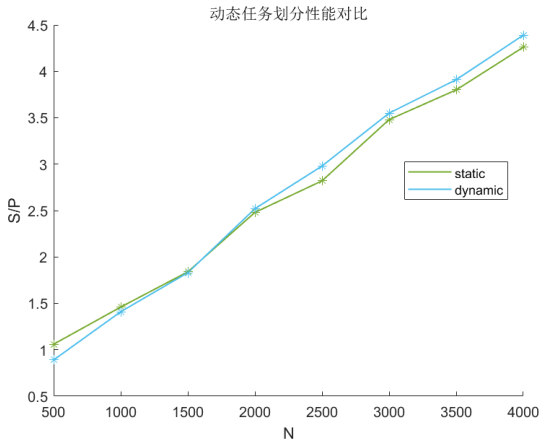

图 6.14: 动态数据划分优化对比

从图像中可以看到，当数据点数量比较少的时候，动态数据划分的方式性能要差于静态数据划分，这是由于动态数据划分需要更多的线程同步，对于任务标识的修改是需要加锁同步的，因此会有比较大线程调度开销，因此在小规模下其性能较差。而随着数据点数量的增加，可以看到其加速比逐渐超过了静态方法，说明尽管 KMeans 算法整体来看是负载均衡的，但是线程之间仍存在着一些细微的差异导致了较小的负载不均。动态数据划分方式的性能提升效果不明显就是由于这个负载不均现象并不明显。在实验中使用了 VTune 分析工具分析了任务标识同步所需要的时间，如表4所示，可以看到，小规模下其有一定的占比，但当数据规模增大时，其占比几乎为零。

N	**500**	**1000**	**1500**	**2000**	**2500**	**3000**	**3500**	**4000**
Time	4.9%	3.1%	0.9%	<0.5%	<0.5%	<0.5%	<0.5%	<0.5%

表 4: 不同数据点数量下任务标识同步用时占比

6.2.3 与 SIMD 结合优化

对于每个线程而言，在线程内计算数据点和质心之间相似度的过程并不需要按照传统的串行方法来执行，同样可以参考在SIMD 并行处理一节中采用的 SIMD 向量化并行处理，在计算相似度的时候，

一次性计算多个维度的数值并做向量归约加法。在更新质心的时候，同样不需要采用串行的方法，也可以使用 SIMD 进行向量化并行优化。在本次实验中，便将 pthread 多线程算法和 SIMD 向量化优化相结合，在 x86 平台实现了 SSE、AVX 和 AVX512 三种算法，并测试了在不同数据点数量以及数据维度下的性能表现，以 SSE 算法为例，结果如图6.15所示。

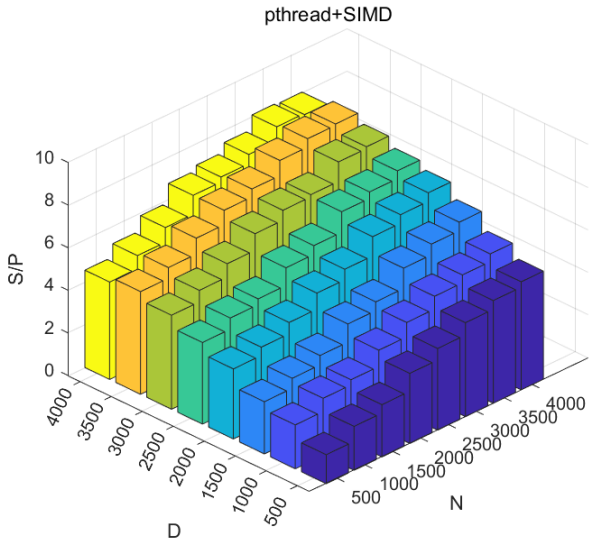

图 6.15: SIMD 优化实验结果

从图像中可以看出，算法的加速比随着数据点数量以及数据维度的增加均呈现上升趋势，随着数据维度的增加，SIMD 向量化处理的性能优化效果得以表现，随着数据点数量的增加，pthread 多线程处理的性能优化效果得以表现。通过对比可以发现，加入了 SIMD 优化后，如图6.16所示，既优化了算法在数据维度方面的伸缩性，又有效提高了算法在数据点数量角度的性能。

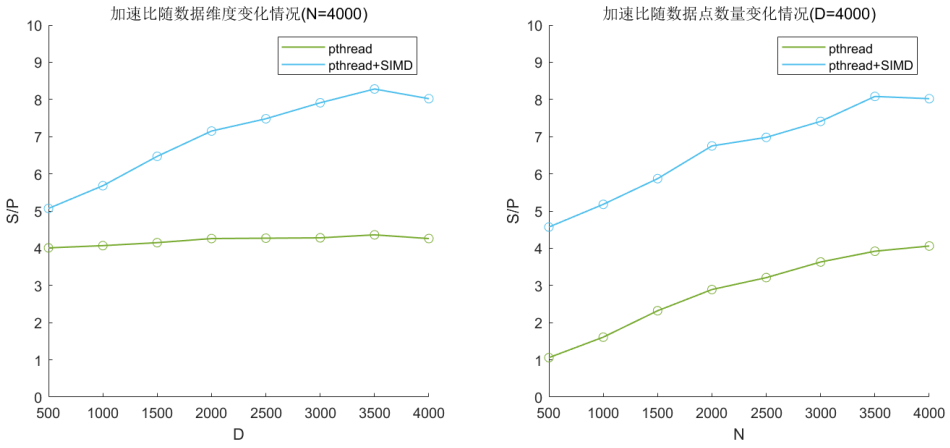

图 6.16: SIMD 优化对比

6.2.4 线程数量探究

考虑到 pthread 算法还会涉及到一个影响因素就是线程的数量，因此在本次实验中探究了采用不同的线程数量，算法的性能表现。实验选定的是 pthread 算法进行测试，分别测试了 $D = N = 4000$ 时，在不同线程数量下的加速比，实验结果如图6.17所示。当线程数量小 10 的时候，整体的加速比随

着线程的数量增加呈现线性上升趋势。而当线程数量超过 16 后，加速比增幅出现了小幅度的下降。分析原因可能是由于线程数量过多，分配给每个线程的任务量变少，因而其线程调度的开销显现出来，导致了算法的伸缩性有所下降。

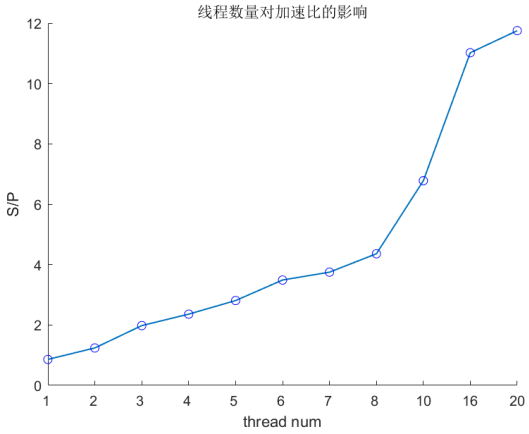

图 6.17: 线程数量对加速比的影响

6.2.5 平台迁移

对于 pthread 算法自身而言，在 ARM 平台和 x86 平台上是完全一致的，因此没有必要对普通的 pthread 算法进行平台迁移，在实验中对 pthread+SIMD 优化算法进行了平台迁移，并对实验结果进行了测试，如图6.18所示。整体结果与 x86 平台无异，只是由于 ARM 平台上当数据维度达到 4000 的时候还没有受到 cache 的影响，因此加速比一直呈现一个上升的趋势。

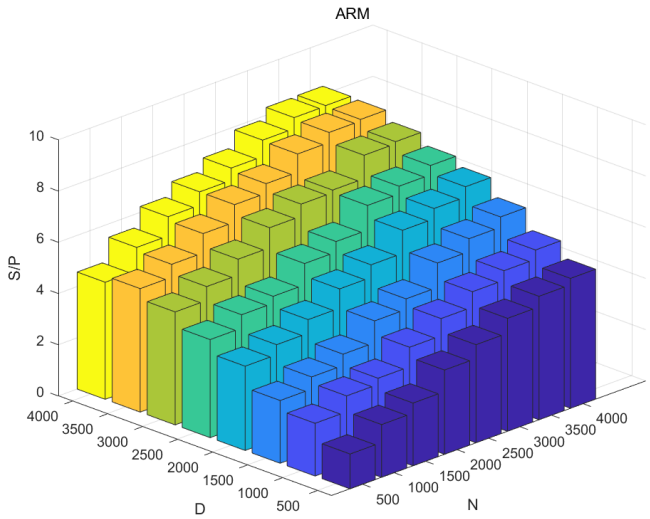

图 6.18: ARM 平台算法性能测试结果

7 OMP 并行优化

7.1 实验设计

考虑到 KMeans 算法的迭代过程中，主要包含两个阶段，第一阶段是对于每一个数据点计算其到各个质心之间的相似度，第二阶段是每一轮迭代完成前，需要根据本轮的划分结果，重新更新质心。对于第一个过程而言，由于数据点的数量很多，因此可以将数据点平均分配给多个线程去执行相似度的计算和聚类划分的过程。而对于第二个阶段，由于聚类的数量 K 般不会太多，因此在本次实验中不考虑将这个阶段进行多线程的并行化处理。

对于第一阶段而言，可以启用多条线程来同时计算数据点的相似度和聚类划分。对于 OMP 而言，提供了多种任务划分的方式，在实验中进行了尝试多种任务划分方式，并对比了其性能表现。当所有的线程计算完相似度并完成聚类划分之后，会进行一次同步，同步完成后，由单个线程完成质心更新，然后就可以进行下一轮迭代。

在实验中还考虑了线程管理的不同方法，设计了静态线程创建和动态线程创建两种方法，并对比了其性能。而对于每一个线程而言，其计算数据点和质心之间相似度的过程，仍然适合采用 SIMD 向量化并行，因此实验中还设计了 OMP 和 SIMD 相结合的算法。实验中还对比了 OMP 提供的 SIMD 算法和手动 SIMD 算法之间的性能差异，以及 OMP 提供的多线程算法和手动 pthread 多线程算法之间的性能差异。

在本次实验中，将设计以下实验进行探究:

1. 采用 OMP 对 KMeans 算法进行多线程并行优化，分析在不同任务规模下的性能表现
2. 对比不同数据划分方式的性能差异
3. 对比动态线程创建和静态线程创建两种方法的性能差异
4. 对比线程内部增加 SIMD 向量化优化方法的性能提升效果
5. 对比 OMP 提供的 SIMD 和手动 SIMD 之间的性能差异
6. 对比 OMP 提供的多线程算法和 pthread 多线程算法之间的性能差异
7. 测试在不同线程数量下算法的性能表现
8. 尝试将 OMP 优化方法从 x86 平台迁移到 arm 平台上，分析性能表现

以下是详细的实验设计方案

7.1.1 OMP 并行处理

对于每一轮迭代而言，主要包含两个阶段，第一阶段是对于每一个数据点计算其到各个质心之间的相似度，第二阶段是每一轮迭代完成前，需要根据本轮的划分结果，重新更新质心。本次实验设计中，主要针对第一个阶段进行多线程并行优化。

考虑到在进行聚类算法的时候，涉及到的数据点一般都会比较多，因此可以将数据点平均划分给多个线程。OMP 会自动将任务划分给指定的线程，然后每个线程并行计算数据点和质心之间的相似度，并完成聚类的划分。在每个线程完成了聚类划分之后，所有线程会进行一次同步操作。最后再指定单个线程要负责根本轮的聚类划分结果，重新计算质心。完成了质心的更新之后，就可以开始下一轮的迭代。

完成了 OMP 算法的实现之后，将会在平台上测试在不同数据点数量以及数据维度下的性能表现；此外还会测试对于相同数据点数量以及数据维度下，采用不同线程数量的性能表现。由于 OMP 实现了自动的多线程编程，因此实验还将与之前通过 pthread 手动实现多线程优化的算法进行对比。

7.1.2 数据划分方式对比

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 KMeans 迭代的过程中，由于每个阶段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。由于在每一轮迭代的过程中，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。

7.1.3 线程创建方式对比

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。动态线程创建的方式就是在每一轮迭代的过程中，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。

7.1.4 与 SIMD 结合优化

对于每个线程而言，在线程内计算数据点和质心之间相似度的过程并不需要按照传统的串行方法来执行，同样可以参考在SIMD 并行处理一节中采用的 SIMD 向量化并行处理，在计算相似度的时候，一次性计算多个维度的数值并做向量归约加法。在更新质心的时候，同样不需要采用串行的方法，也可以使用 SIMD 进行向量化并行优化。OMP 提供了可以进行 SIMD 向量化处理的预编译选项，因此可以直接使用 OMP 提供的 SIMD 方法。在本次实验中，便将 OMP 多线程算法和 SIMD 向量化优化相结合。在本次实验中，还对手动的 SIMD 算法与 OMP 提供的 SIMD 算法进行的对比。

7.1.5 平台迁移

在本次实验中，不仅尝试了在 x86 平台实现 OMP 算法，并且还将算法迁移到了 ARM 平台上，由于单纯使用 OMP 算法在两个平台之间没有任何差异并且 SIMD 算法相结合的时候，如果采用 OMP 提供的 SIMD 方法，也具有很好的平台一致性。当 OMP 与手动的 SIMD 结合的时候，在不同的平台需要选择不同的指令集架构，在 x86 平台使用了 SSE、AVX 和 AVX512 三种指令集架构实现算法，在 ARM 平台使用了 Neon 指令集架构实现算法。

7.2 实验结果分析

7.2.1 OMP 并行处理

考虑到在我们的实验数据中总共有两个影响因素，一个是数据点的数量 N，另一个因素是数据的维度 D，因此当考虑并行算法的优化效果的时候，需要综合考虑这两方面的因素。为了能够充分显示出 OMP 并行优化的效果，在实验中将数据点的个数 N 设置在 [500,4000] 的区间内，将数据维度 D 设置在 [500,4000] 的区间内。为了能够消除收敛速度的影响，迭代轮次统一选择迭代 500 次，线程数量选择使用 8 条线程。

在 x86 平台上完成了 OMP 算法的实现，并且分别调整数据点的数量 N 和数据维度 D，得到实验结果如图7.19所示。从图中可以看出，当数据点数量很少的时候，算法的加速比在 1 左右，这是由于线程的调度开销很大，导致了其带来的性能提升被线程调度抵消。但随着数据点数量的增加，OMP 算法的加速比逐渐上升，而由于在实验中拉起了八条线程，因此算法的理论加速比为 8。但是由于循环迭代以及其他的运算并没有进行加速，因此理论加速比会略低于 8. 在实验中，加速比一直在上升，在实验数据覆盖范围内的加速比最高达到了 5.83。而从数据维度的角度来看，算法的加速比并没有明显的变化，这是由于多线程的优化操作是从数据点数量的角度出发的，因此当数据维度增加的时候，并不能够很好体现出算法的优化性能。

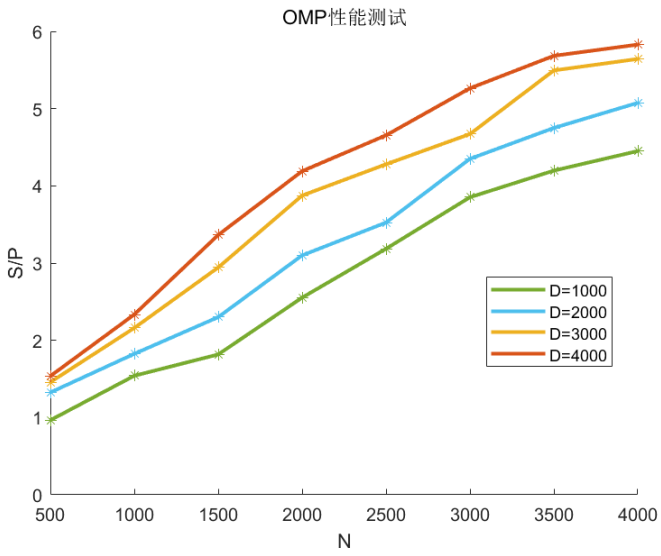

图 7.19: x86 平台 OMP 性能测试结果

在pthread 并行优化一节中，利用 pthread 手动实现了多线程的算法，在本节中使用 OMP 自动实现了多线程的并行算法，两种方法的多线程的策略是一致的，都是将计算各个数据点到质心的相似度并完成聚类划分的任务分配给多个线程，然后进行线程之间的同步，同步完成后由一个线程更新质心，之后进行下一轮迭代。在本次实验中，对比了 OMP 自动实现多线程算法和 pthread 手动实现多线程算法之间的性能差异，为了能够充分比较两者的差异，选取了 $N = 4000$ 时的加速比进行对比，如图7.20所示。可以看出，通过 OMP 自动实现的多线程算法要明显由于笔者手动 pthread 实现的多线程算法，但笔者并未能够分析出其中的具体原因，通过 VTune 性能分析工具，分析两种算法的 CPU 事件，如表5所示，从表中可以看出，两种算法所使用的指令数大致相同，但是 OMP 所需要的时钟周期数要少于 pthread，推测应该是线程调度效率 OMP 要高于笔者实现的 pthread 算法。

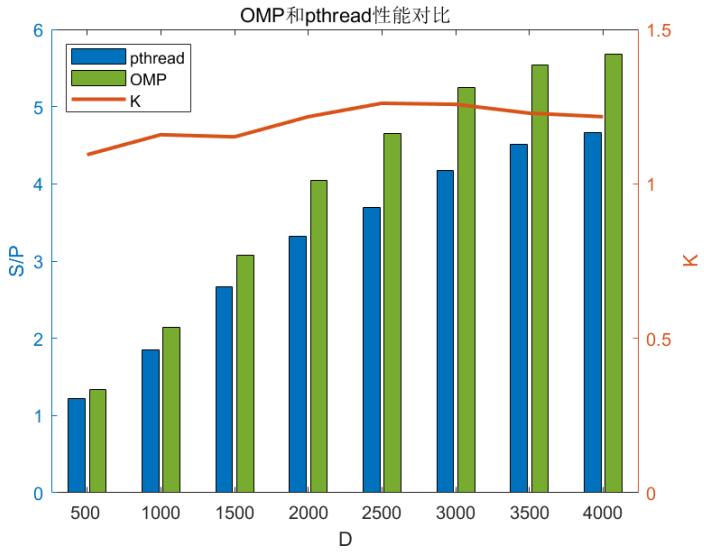

图 7.20: OMP 和 pthread 性能对比结果

	pthread	OMP
Instructions	1.43E+9	1.38E+9
Cycles	3.47E+10	3.15E+10

表 5: 内存对齐策略 CPU 事件对比

7.2.2 数据划分方式对比

OMP 为编程提供了多种任务划分方式，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。在本次实验中分别尝试了 static、dynamic 和 guided 三种任务划分方式，并且对比了当数据维度 $D = 4000$ 时，在不同数据点数量下的性能表现，如图7.21所示。

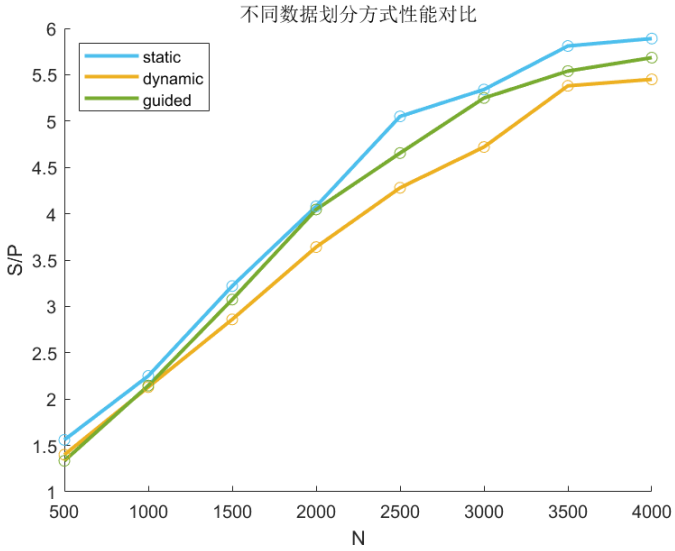

图 7.21: OMP 不同数据划分性能对比结果

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，static 方法表现出了较好的性能，而 dynamic 和 guided 两种方法的性能要略差于 static。原因是在 KMeans 迭代的过程中，每一轮每一行的计算量是大致相同的，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 dynamic 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。

在实验中使用 VTune 性能分析工具对于这三种方式的 CPU 占用时间进行监测，并使用以下公式来计算各线程之间 CPU 占用时间的极差，来衡量线程之间的负载均衡的情况。

$$\frac{time_{max} - time_{min}}{mean(time)}$$

从图中可以看出，static 方式的 CPU 占用时间分配十分的不均衡，极差约在 4.32%，这也体现了这种方法会受到负载不均的制约，因此其运行时间为 178.4ms。在本次实验中，dynamic 方式的 CPU 占用时间分布较为均衡，极差约在 4.12%，而且通过实验测量程序的运行时间可以发现 dynamic 方式所消耗的时间为 179.2ms。可以印证，dynamic 方式能够取得更好的负载均衡，但是由于其需要更细粒度的划分导致额外开销加大，在 KMeans 这种负载均衡的问题上表现不理想。

7.2.3 线程创建方式对比

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6所示。动态线程创建的方式就是在每一轮迭代的过程中，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**
500	1.33	3.35%	1.38	5.35%
1000	2.14	0.49%	1.99	3.68%
1500	3.07	<0.1%	2.90	3.62%
2000	4.04	<0.1%	3.89	3.56%
2500	4.65	<0.1%	4.40	3.42%
3000	5.24	<0.1%	4.98	3.21%
3500	5.53	<0.1%	5.24	2.89%
4000	5.68	<0.1%	5.40	2.54%

表 6: 动态静态线程管理对比

7.2.4 与 SIMD 结合优化

对于每个线程而言，在线程内计算数据点和质心之间相似度的过程并不需要按照传统的串行方法来执行，同样可以参考在SIMD 并行处理一节中采用的 SIMD 向量化并行处理，在计算相似度的时候，一次性计算多个维度的数值并做向量归约加法。在更新质心的时候，同样不需要采用串行的方法，也可以使用 SIMD 进行向量化并行优化。在本次实验中，便将 OMP 多线程算法和 SIMD 向量化优化相结合，在 x86 平台实现了 SSE、AVX 和 AVX512 三种算法，并测试了在不同数据点数量以及数据维度下的性能表现，以 SSE 算法为例，结果如图7.22所示。在测试数据覆盖范围内达到的最大加速比为 9.82。

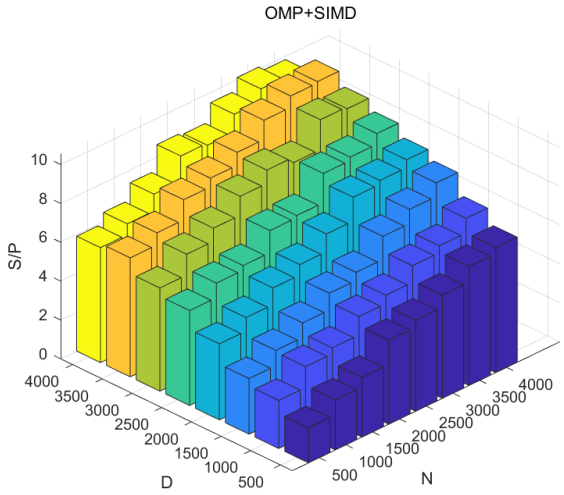

图 7.22: OMP 结合 SIMD 性能优化结果

从图像中可以看出，算法的加速比随着数据点数量以及数据维度的增加均呈现上升趋势，随着数据维度的增加，SIMD 向量化处理的性能优化效果得以表现，随着数据点数量的增加，OMPd 多线程处理的性能优化效果得以表现。通过对比可以发现，加入了 SIMD 优化后，如图7.23所示，既优化了算法在数据维度方面的伸缩性，又有效提高了算法在数据点数量角度的性能。

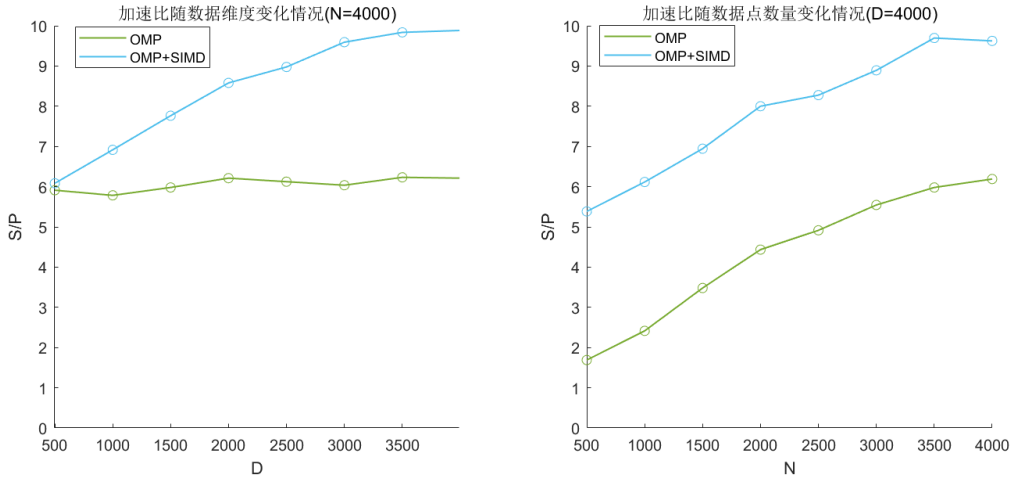

图 7.23: OMP 结合 SIMD 性能优化对比结果

OMP 还提供了 SIMD 的预编译选项，通过设置 SIMD 的向量位宽，可以实现类似于 SSE、AVX 和 AVX512 不同指令集架构的 SIMD 并行算法。在本次实验中，还对比了采用 OMP 提供的 SIMD 算法和手动实现的基于 SSE、AVX 和 AVX512 的并行算法之间的性能差异。由于实验主要是为了探究 SIMD 处理之间的性能差异，因此选定数据点个数 $N = 4000$，对比结果如图7.24所示。可以看到，总体来说，手动 SIMD 算法的性能要优于 OMP 提供的 simd 算法。

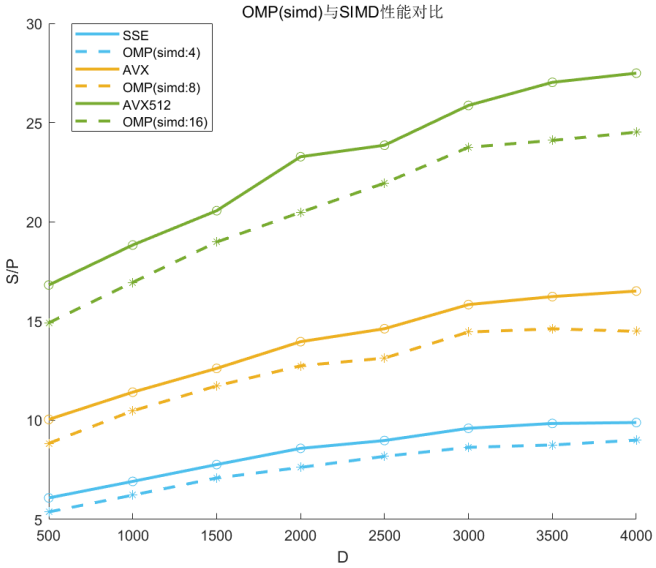

图 7.24: OMP(simd) 与 SIMD 性能对比结果

7.2.5 线程数量对比

在本次实验中，探究了不同线程数量下，OMP 多线程算法的性能表现，其变化趋势大致如图7.25所示。从图像中可以看出，算法的加速比随着线程数量的增加呈现一个线性增加的趋势，说明 OMP 的多线程算法具有很好的扩展性。

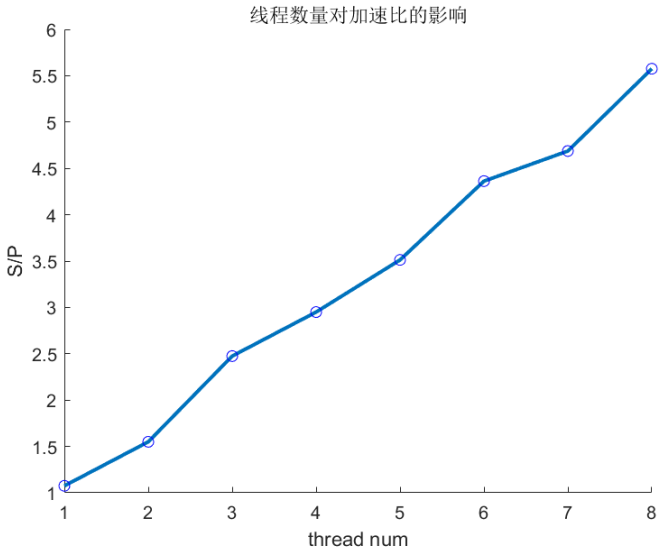

图 7.25: x86 平台线程数量对性能的影响

7.2.6 平台迁移

对于 OMP 算法自身而言，在 ARM 平台和 x86 平台上是完全一致的，因此没有必要对普通的 OMP 算法进行平台迁移，在实验中对 OMP+SIMD 优化算法进行了平台迁移，并对实验结果进行了测试，如图7.26所示。整体结果与 x86 平台无异，只是由于 ARM 平台上当数据维度达到 4000 的时候还没有受到 cache 的影响，因此加速比一直呈现一个上升的趋势。在测试数据覆盖范围内达到的最大加速比为 9.98。

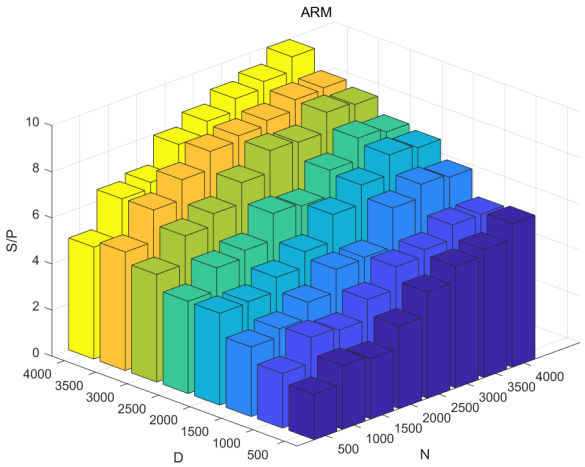

图 7.26: ARM 平台 OMP+SIMD 性能测试

从图像中可以看出，算法的加速比随着数据点数量以及数据维度的增加均呈现上升趋势，随着数据维度的增加，SIMD 向量化处理的性能优化效果得以表现，随着数据点数量的增加，OMP 多线程处理的性能优化效果得以表现。

在 ARM 平台还测试了使用不同线程数量，算法的性能表现，如图7.27所示。从图像中可以看出，在 ARM 平台上，OMP 算法的加速比和线程数量同样呈现线性正相关，说明 OMP 的多线程算法具有较好的扩展性。

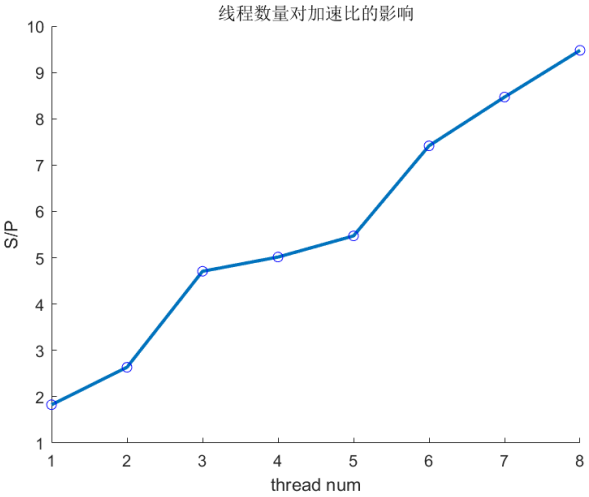

图 7.27: ARM 平台线程数量对性能的影响

8 MPI 并行优化

8.1 实验设计

考虑 KMeans 聚类的整个过程中主要涉及到两个阶段，一个是计算各个数据点到质心的相似度并划分聚类，一个是根据本轮的划分结果更新质心。而就每个阶段而言，其所做的工作基本是一致的。尤其是针对第一个阶段，即计算各个数据点到质心的相似度并划分聚类，这个阶段每一个数据点所做的工作是完全一致的，十分适合并行化处理，即将待计算的数据点平均分配给几个不同的进程，由于这些数据之间不存在依赖性，因此每个进程只需要各自完成好自己的工作即可，不存在进程之间进行通信的额外开销，只需要在各个进程完成了分配的工作之后进行一次同步。

而当任务下发给不同的进程之后，在各个进程内又可以开启多条线程来处理任务，由于任务之间是不存在数据依赖的，因此不需要进行数据的通信，因此考虑在同一个进程内开启多条线程来处理任务是十分合理并且有效的。

而对于第一阶段，即计算各个数据点到质心的相似度时，可以结合 SIMD 的向量化处理。同样在第二阶段，根据本轮的划分结果更新质心的过程中，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 MPI 对于 KMeans 算法进行多进程优化，分析在不同任务规模下的性能表现
2. 设计不同的任务分配模式，即采用主从模式和从从模式，对比不同的任务分配模式的性能表现
3. 设计不同的数据通信模式，即采用阻塞通信和非阻塞通信，对比不同数据通信模式下的性能表现
4. 使用 MPI 配合 SIMD 向量化处理，对比其性能提升
5. 使用 MPI 配合 OMP 进行多线程优化，对比其性能提升
6. 使用 MPI 配合 SIMD 和 OMP 同时进行多进程多线程向量化优化，对比其性能提升
7. 对比采用不同进程数量，MPI 的性能表现情况

以下是详细的实验设计

8.1.1 MPI 并行处理

考虑到 KMeans 算法在执行的过程中主要包括两个阶段，第一个阶段是对于每一个数据点计算到各个质心之间的相似度并划分聚类标签，第二个阶段是所有数据点划分聚类完成之后，根据新的聚类标签更新质心信息。对于第一个过程而言，对每一个数据点所做的操作都是完全一致的，因此这个过程十分适合并行处理。MPI 是进程级的并行方法，因此可以考虑将数据点平均分配给多个进程，各个进程之间并行完成有关数据点和质心之间的相似度计算以及聚类划分工作，最终由单个进程完成质心的更新操作。

在程序开始时，在零号进程进行数据的初始化工作，初始化完成后，零号进程会根据拉起的进程数量将任务进行平均分配，然后给各个进程下发数据和质心信息。每个进程在接受了数据和质心之后，计算相似度并完成聚类划分，将划分结果的聚类标签返回给 0 号进程。0 号进程收到聚类标签之后，根据新的划分结果更新质心，并进行下一轮操作。

8.1.2 不同的任务分配模式对比

一般而言，MPI 的任务分配模式包含两种：主从模式和从从模式。主从模式就是有一个主进程来负责任务的分配和收集，并完成同步和汇总计算工作，而其他的工作进程只完成并行的计算工作。从从模式就是不存在这样一个只负责数据分配和收集的进程，所有的进程都是工作进程，并且有一个进程需要额外完成数据分发收集和同步工作。一般而言，主从模式的设计会更为简单，但是可能会浪费掉一个进程的计算资源，而从从模式能够更充分地利用每个进程的计算资源。

在实验中就设计了主从模式和从从模式两种任务分配模式。主从模式中，零号进程首先会完成数据和质心的分发工作，然后就进入等待状态，等待工作进程完成数据点的聚类划分后将划分标签发送给零号进程。在零号进程收集了所有的聚类标签后，会根据本轮的划分结果更新质心，至此完成了一个迭代的工作。从从模式中，零号进程同样会先完成数据和质心的分发工作，然后和所有的进程一起完成数据点的完成数据点的聚类划分，等待收集其他进程发来的聚类标签，并根据聚类结果更新质心，完成一个迭代。

8.1.3 不同的数据通信模式对比

MPI 中提供了多种数据通信的方式，在本次实验中，主要对比了阻塞通信和非阻塞通信之间的性能表现。阻塞式通信是消息发出之后会进入阻塞状态，直到对方收到消息之后才会继续向下执行，非阻塞通信是消息发送之后并不进入阻塞状态直接向下执行。在实验中，分别使用阻塞通信和非阻塞通信实现了 MPI 算法。

阻塞通信的实现方式是：在每轮迭代中，零号进程会向各个进程一次性发送所有的质心信息，然后每个工作进程在接收到所有的质心之后才会开始计算工作。而非阻塞通信是，零号进程每更新一个质心，都会向各个进程发送一次质心的信息，工作进程每接收到一个质心信息后，就开始计算数据点的相似度，不需要等待所有的质心信息都接收到再开始工作。

8.1.4 MPI 与 SIMD、OMP 结合优化

考虑到 MPI 是进程级的并行处理，而在每个进程内，计算数据点到各个质心的相似度的过程也具有高度一致性，因此仍然可以参考OMP 并行优化一节中的多线程并行处理，即在进程内部，可以将每个进程负责的工作平均分配给多个线程并行执行。而在每个线程内，考虑到计算相似度的过程十分适合使用向量化处理，因此可以参考SIMD 并行处理一节中的向量化处理，一次计算多位数据，提高算法的并行性。并且，MPI 进程级并行和 OMP 线程级并行的优化是从数据点数量的角度出发的，而 SIMD 指令级并行的优化是从数据维度的角度出发的，这样综合设计可以很好的适应在不同数据点数量和数据维度下算法的伸缩性。

8.2 实验结果分析

8.2.1 MPI 并行处理

考虑到在我们的实验数据中总共有两个影响因素，一个是数据点的数量 N，另一个因素是数据的维度 D，因此当考虑并行算法的优化效果的时候，需要综合考虑这两方面的因素。为了能够充分显示出 OMP 并行优化的效果，在实验中将数据点的个数 N 设置在 [500,4000] 的区间内，将数据维度 D 设置在 [500,4000] 的区间内。为了能够消除收敛速度的影响，迭代轮次统一选择迭代 500 次，进程数量选择使用 8 个进程。

在 x86 平台上完成了 OMP 算法的实现，并且分别调整数据点的数量 N 和数据维度 D，得到实验结果如图8.28所示。

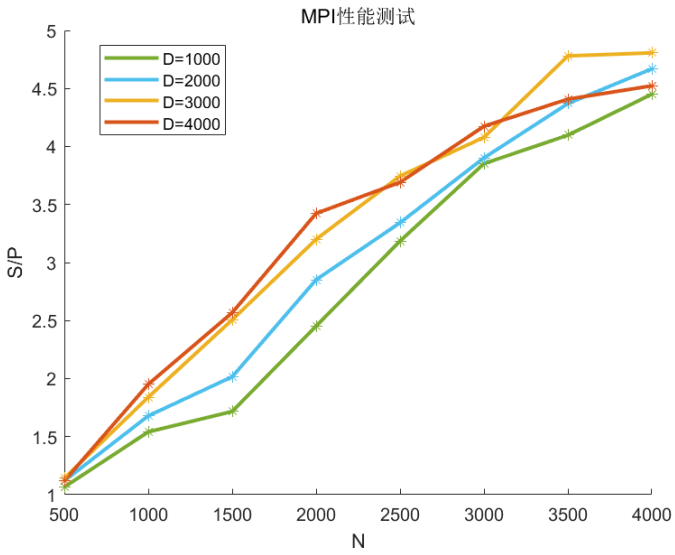

图 8.28: MPI 性能测试

从图像中可以看到，随着数据点的增加，在不同数据维度下的算法性能都在上升，但从趋势上来看，上升的幅度逐渐降低，应该是逐渐逼近了 MPI 的理论加速比。当数据点数量在 500 时，算法的加速比大概只在 1 左右，而在所有测试数据覆盖范围内，算法的最大加速比达到了 4.82。从不同的数据维度的角度来看，算法在不同数据维度上的差异并不大，这是由于 MPI 的并行处理还是从数据点数量的角度出发的，增肌数据维度并不会对算法的性能产生较大的影响，相反增加数据点的数量能够在很大程度上使得算法性能得到充分发挥。并且注意到，当 $D = 4000$ 的时候，算法的性能要略低于 $D = 2000$ 和 $D = 3000$ 的性能，分析原因主要是由于数据维度的增加，导致一行中的数据点数量增加，降低了 cache 的命中率。这一点也可以通过 CPU 事件来印证，如表7所示。

D	**2000**	**3000**	**4000**
L1-cache hit	99.2	98.9	98.2
L2-cache hit	98.3	98.6	96.5
L3-cache hit	97.2	97.6	98.2

表 7: 不同数据维度下 CPU 事件对比

8.2.2 不同的任务分配模式对比

在实验中就设计了主从模式和从从模式两种任务分配模式。主从模式中，零号进程首先会完成数据和质心的分发工作，然后就进入等待状态，等待工作进程完成数据点的聚类划分后将划分标签发送给零号进程。在零号进程收集了所有的聚类标签后，会根据本轮的划分结果更新质心，至此完成了一个迭代的工作。从从模式中，零号进程同样会先完成数据和质心的分发工作，然后和所有的进程一起完成数据点的完成数据点的聚类划分，等待收集其他进程发来的聚类标签，并根据聚类结果更新质心，完成一个迭代。

在实验中，对比了当数据维度 $D = 3000$ 时，启用 8 个进程，主从模式和从从模式在不同数据点数量下的性能差异，实验结果如图8.29所示。

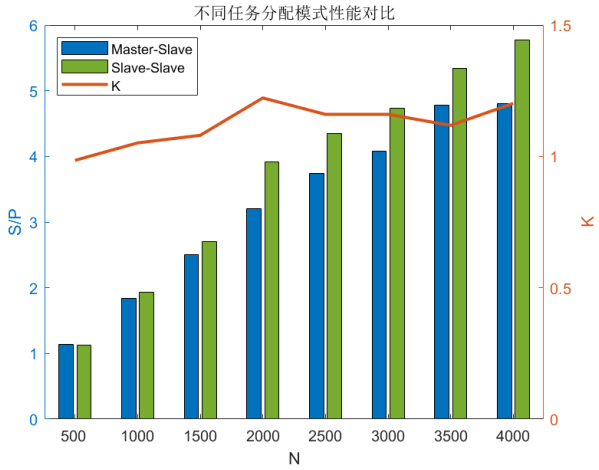

图 8.29: 不同任务分配模式性能对比

从图像中可以看出，随着数据点数量的增加，两种算法的加速比均在上升，但是相对来讲，采用从从模式的算法能够获得更高的加速比，这是因为从从模式中，充分利用了每一个进程的计算单元，而在主从模式中缺浪费了一个进程的计算资源。并且从图像中我们可以看出，随着数据点个数的增加，两种从从模式相对于主从模式的加速比也整体呈现上升的趋势，直到达到进程数量之比。

8.2.3 不同的数据通信模式对比

MPI 中提供了多种数据通信的方式，在本次实验中，主要对比了阻塞通信和非阻塞通信之间的性能表现。阻塞式通信是消息发出之后会进入阻塞状态，直到对方收到消息之后才会继续向下执行，非阻塞通信是消息发送之后并不进入阻塞状态直接向下执行。在实验中，分别使用阻塞通信和非阻塞通信实现了 MPI 算法。阻塞通信的实现方式是：在每轮迭代中，零号进程会向各个进程一次性发送所有的质心信息，然后每个工作进程在接收到所有的质心之后才会开始计算工作。而非阻塞通信是，零号进程每更新一个质心，都会向各个进程发送一次质心的信息，工作进程每接收到一个质心信息后，就开始计算数据点的相似度，不需要等待所有的质心信息都接收到再开始工作。

在实验中，对比了当数据维度 $D = 3000$ 时，启用 8 个进程，阻塞通信和非阻塞通信在不同数据点数量下的性能差异，实验结果如图8.30所示。

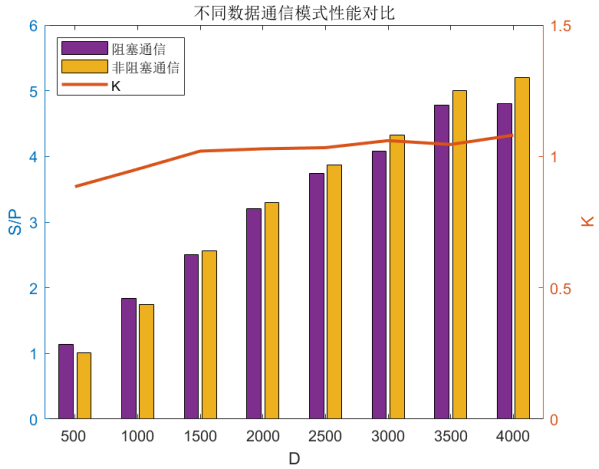

图 8.30: 不同数据通信模式性能对比

从图像中可以看到，当数据规模比较小的时候非阻塞通信的性能表现要差于阻塞通信，这是由于非阻塞通信中由更多的数据收发操作，在小规模下这种开销占比很高，影响了性能表现。当数据规模增加的时候，非阻塞通信由于减少了等待的事件，提高了并行度，因此获得了一定的性能提升，相对于阻塞通信大约提升了 6% 左右的性能。

8.2.4 MPI 与 SIMD、OMP 结合优化

考虑到 MPI 是进程级的并行处理，而在每个进程内，计算数据点到各个质心的相似度的过程也具有高度一致性，因此仍然可以参考OMP 并行优化一节中的多线程并行处理，即在进程内部，可以将每个进程负责的工作平均分配给多个线程并行执行。而在每个线程内，考虑到计算相似度的过程十分适合使用向量化处理，因此可以参考SIMD 并行处理一节中的向量化处理，一次计算多位数据，提高算法的并行性。并且，MPI 进程级并行和 OMP 线程级并行的优化是从数据点数量的角度出发的，而 SIMD 指令级并行的优化是从数据维度的角度出发的，这样综合设计可以很好的适应在不同数据点数量和数据维度下算法的伸缩性。

在实验中，分别设计了 MPI+OMP、MPI+SIMD 和 MPI+OMP+SIMD 三种结合优化算法，并且选定 MPI 使用 8 个进程，OMP 使用 8 条线程，SIMD 使用 SSE 的 4 路向量化进行算法实现。为了能够充分表现算法的优化效果，分别选取了 $N = 4000$ 时加速比随着数据维度 D 的变化情况，以及 $D = 4000$ 时加速比随着数据点数量 N 的变化情况，如图8.31所示。

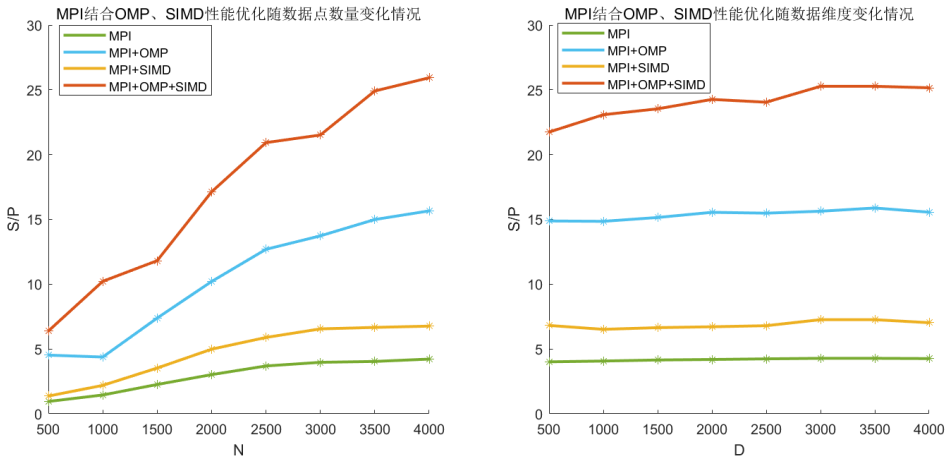

图 8.31: MPI 结合 OMP、SIMD 性能优化测试

从图中可以看出，当固定了数据点维度时，四种算法都随着数据点数量的增加逐渐上升，并且增速逐渐放缓，说明算法在逐渐逼近理论加速比。而且可以看到 MPI+OMP 优化并没有能够达到理论加速比，甚至可以说和理论加速比相差甚远，其主要原因是 MPI 将数据分发给了不同的进程，在进程内数据又被分给了不同线程，经过两次拆分之后，数据量已经不大，因此线程调度开销变大，极大地影响了程序的实际性能。

9 GPU 并行优化

9.1 实验设计

考虑 KMeans 聚类的整个过程中主要涉及到两个阶段，一个是计算各个数据点到质心的相似度并划分聚类，一个是根据本轮的划分结果更新质心。而就每个阶段而言，其所做的工作基本是一致的。尤其是针对第一个阶段，即计算各个数据点到质心的相似度并划分聚类，这个阶段每一个数据点所做的工作是完全一致的，十分适合并行化处理，并且考虑到这个过程就是简单的数学运算，并不涉及到复杂的逻辑判断，属于计算密集任务，十分适合使用 GPU 进行加速运算。

结合本学期所学内容，使用 NVIDIA 提供的在线平台，基于 CUDA 完成了对于 KMeans 的并行加速，具体的算法设计如下：

首先在 CPU 端初始化计算工具，获得数据点信息，并且将质心随机初始化完成。随后将数据点信息拷贝到 GPU 中，同时将质心的信息也拷贝到 GPU 中，完成了在 GPU 端的数据初始化工作。然后根据实验环境所提供的硬件设备的流处理器个数，声明使用的网格和线程块的数量。到此为止，就完成了算法前期的初始化准本工作。

接下来开始 KMeans 的核心迭代过程。对于第一阶段计算数据点到各个质心相似度的过程，设计了核函数实现。在核函数中，根据网格线程块的 ID 以及线程 ID 来确定任务，每个线程负责一部分数据点的相似度计算工作，根据计算的结果来更新当前数据点的聚类划分标签。当所有的线程都完成工作之后，会进行一次 GPU 的同步操作。之后，采用多个线程来计算质心的更新操作，这里利用了多线程进行了类似 SIMD 的操作。到此为止就完成了一轮循环迭代。

算法的流程图如图9.32所示。

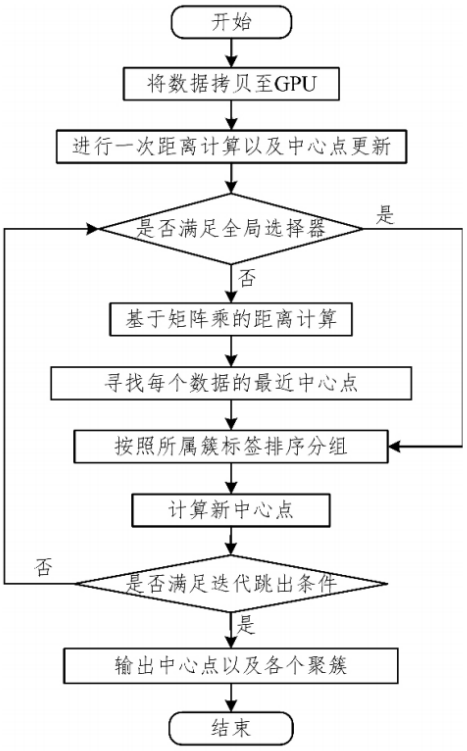

图 9.32: CUDA 算法流程图

9.2 实验结果分析

在完成了 CUDA 的算法实现之后，在 NVIDIA 提供的在线平台上测试了算法的性能表现。考虑到 KMeans 算法主要有两个参数：数据点数量 N 和数据维度 D，因此在实验中将数据点的个数 N 设置在 [500,4000] 的区间内，将数据维度 D 设置在 [500,4000] 的区间内。为了能够消除收敛速度的影响，迭代轮次统一选择迭代 500 次。将线程块的数量设置为流处理器的 32 倍，每个线程块中线程的数量为 256。具体的实验结果如图9.33所示。

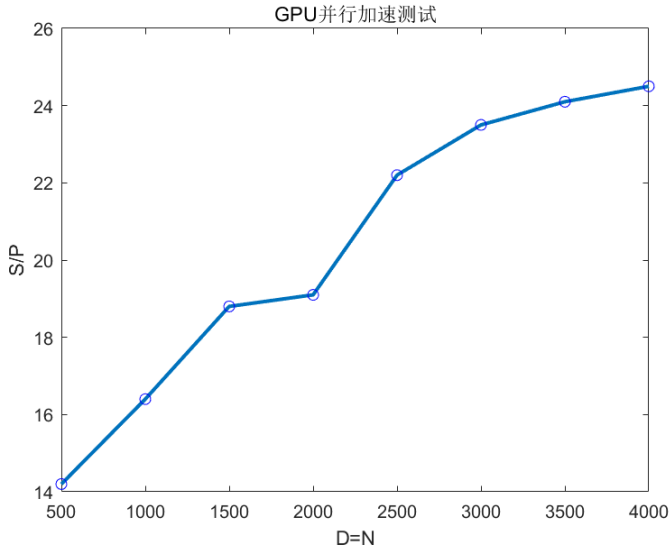

图 9.33: GPU 并行优化测试

从实验的结果来看，GPU 加速并没有取得非常高的性能提升，在前期调研的过程中，曾经看到相关的 CUDA 并行优化能够取得 100 倍左右的性能提升，如图9.34所示。通过分析，推测主要原因是并没有充分利用 GPU 中的多线程优势，由于 GPU 中的线程数量十分丰富，因此可以将细粒度的工作尽可能交给一个线程完成，而不需要一个线程完成一个较粗粒度的工作，充分发挥 GPU 的并行效率。由于时间原因，相关的实验并未能够开展。

n	CPU	GPUMiner	GS_kmeans	与 CPU 的加速比	与 GPUMiner 的加速比
20 k	155.21	4.85	1.72	90.24	2.82
50 k	203.13	6.21	1.91	106.35	3.25
100 k	343.82	7.98	2.62	131.23	3.05
200 k	609.33	15.64	3.65	166.94	4.29
400 k	1042.11	23.72	4.73	220.32	5.01

图 9.34: CUDA 相关实验结果

10 综合优化

通过本次实验的探究过程，分别探究了 SIMD 向量化优化、pthread 多线程优化、OMP 多线程优化、MPI 多进程优化以及 GPU 优化多种途径对于 KMeans 算法的优化效果。通过实验探究，总结出了从进程级并行，到线程级并行再到指令级并行的多级并行优化策略。

在进程级并行采用了 MPI 进行实现，根据MPI 并行优化一节中，提出了基本的 MPI 并行算法，为了能够充分利用每一个进程的计算资源考虑使用从从模式进行任务处理，并使用非阻塞通信提高并行度。并且应该将 MPI 与线程级并行优化 OMP、指令级并行优化 SIMD 结合使用。

在线程级并行采用了 OMP 进行实现，根据OMP 并行优化一节中，对比了 pthread 算法和 OMP 算法的时间效率，选择使用 OMP 进行多线程并行。在 OMP 的实验探究中，由于 KMeans 算法不存在严重的负载不均现象，因此为了减少额外开销，在数据划分的时候就采用静态数据划分的方式。此外，在线程内的计算工作还应该与 SIMD 进行结合。

在指令级并行采用了 SIMD 进行实现，根据SIMD 并行处理一节中，考虑使用平台兼容性更高的 SSE 进行实现，并且为了能够更好的应对随着数据维度增加导致的 cache 命中率降低，增加了数据预取优化策略。并且为了能够减少数据打包操作，提前将数据按照向量寄存器宽度进行内存对齐。

在 x86 平台测试了综合优化算法，其中 MPI 启用了 8 个进程，OMP 启用了 8 条线程，SIMD 采用了 SSE 的 4 路向量化，测试数据涵盖两方面，数据点数量 N 和数据维度 D，两者的取值都来自区间 [500,4000]，实验结果如图10.35所示。

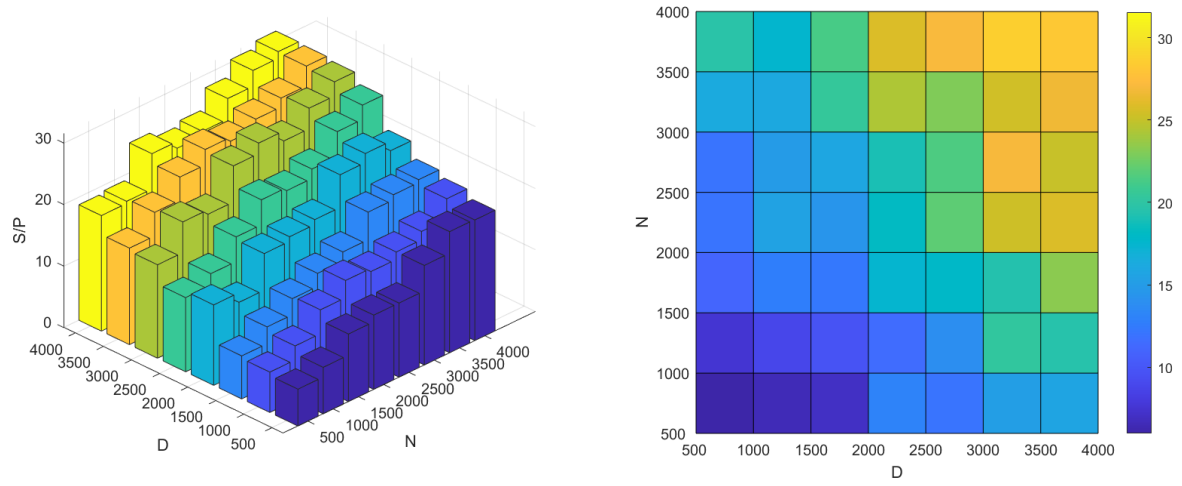

图 10.35: KMeans 综合优化结果

从图像中可以看出，在测试数据覆盖范围内，算法的最大加速比达到了 30.21，并且从数据点数量以及数据维度的角度来看，算法都具有良好的伸缩性和扩展性。其中 MPI 多进程并行和 OMP 多线程并行是从数据点数量的角度进行并行优化的，SIMD 向量化并行是从数据维度的角度进行并行优化的。因此最终的综合优化算法在两个维度上都表现出了良好的性能。

11 总结

通过本学期课程的学习，跟随课程进度完成了 Gauss 消元的并行优化工作，并在期末结合课程所学，完成了对于 KMeans 算法的并行优化探究。分别尝试了 SIMD 向量化处理、pthread 多线程优化、OMP 多线程优化、MPI 多进程优化以及 GPU 优化。在 SIMD 实验中，基于不同的平台实现了不同指令集架构的 SIMD 算法，在 x86 平台实现了 SSE、AVX 和 AVX512 的并行算法，在 ARM 平台实现了 Neon 的并行算法，并且探究了增加数据内存地址对齐和数据预取优化后的性能提升。在 pthread 实验中，实现了 pthread 多线程算法，并对比了采用静态数据划分和动态数据划分不同方法的性能差异，并于 SIMD 相结合进行优化。在 OMP 的实验中，实现了 OMP 多线程算法，并对比了基于不同预编译选项的数据划分方式的性能差异，探究了静态和动态线程管理的性能差异，并于 SIMD 相结合进行优化。在 MPI 的实验中，实现了 MPI 多进程算法，并对比了主从模式和从从模式的性能差异，对比了阻塞通信和非阻塞通信的性能差异，并且于 OMP 和 SIMD 相结合进行优化。在 GPU 实验中，则结合课程所学，使用 CUDA 完成了 GPU 并行优化算法。最终，根据上述实验的探究结果，完成了综合优化算法，设计了分级并行优化算法，使用 MPI 进行进程级并行，并采用非阻塞通信的从从模式；使用 OMP 进行线程级并行，并采用静态数据划分的方式；使用 SIMD 进行指令级并行，增加了内存对齐和数据预取优化。最终的综合优化算法相对于传统的串行算法取得了 30.21 倍的加速比，并且在数据点数量以及数据维度两个方面都表现出了良好的扩展性。本次课程的全部代码已经上传GitHub。