南開大學
Nankai University

计算机学院

并行程序设计实验报告

KMeans 算法 SIMD 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 4 月 11 日

目录

1 问题重述

1.1 研究任务

对于 KMeans 聚类算法，其基本思想是根据数据点在所有属性上的相似性，将数据点划分成给定的 K 个集合。在划分的过程中，会涉及到大量有关距离的计算，而这类计算规则简单，大量重复，而且数据之间没有依赖性，十分适合进行并行化处理。在本次基于 SIMD 的并行优化实验中，我们主要针对在 KMeans 算法中，每次迭代中对数据点的划分进行并行化处理。

每次迭代的过程中主要包含两个过程:

1. 对于每个数据点，计算其到各个质心之间的距离
2. 针对每个数据点，选择其中距离最近的质心，将其划分到其所属的集合中。

而本次实验也正是针对这两个过程，采用 SIMD 的方式进行并行化处理，探究在不同平台，不同指令集架构下，SIMD 对于 KMeans 算法的并行优化效果。

1.2 数学描述

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 $C = \{c_1, c_2, \cdots, c_k\}$。

针对第 i 个节点，计算这个节点到第 k 个质心的距离

$$L_{ik} = \sqrt{\sum_{j=1}^{d}(x_i(a_j) - c_k(a_j))^2} \quad , 1 \leq k \leq K$$

针对第 i 个节点，选择距离他最近的质心，并将其划分如该质心所属集合中

$$\min\{L_{ik}\}, 1 \leq k \leq K$$

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

3 实验设计

考虑到 KMeans 算法每次迭代过程中，都需要重复计算各个数据点到所有质心的距离，而这个过程，对于每个数据点而言，其计算的方法是相同的，并且在不同数据点之间是数据是没有依赖性的，因此，这个过程十分适合进行向量化的处理，因此我们对 KMeans 的并行优化首先从对各个数据点计算其到所有质心的距离这一过程开始。

3.1 并行处理

针对计算距离这一过程，我们考虑采用向量化的手段进行处理，即每次取出多个数据点同一维度的数据存储到向量寄存器当中，再构造某一个质心同一维度的一个向量寄存器，首先利用 CPU 中有关向量减法的指令，计算出这多个数据点和该质心在这一维度上的距离差，然后将这个距离差利用向量对位乘法相乘，得到平方距离。再循环对这多个数据点在其他维度上重复上述操作，并累加这个距离，直到将这多个数据点各个维度的距离全部计算累加完成。再继续去取第二组数据点重复计算上述过程。

为了能够让向量寄存器一次从连续的内存地址中取出多个数据点在某一维度的值，我们考虑将存储数据点和各个维度数值的矩阵进行转置，即用每一行记录 N 个数据点在某一维度上的数据，因此转置后的矩阵为 D 行 N 列。这样，我们在取出多个数据点同一维度的数据时，就不需要进行通道取值或者数据拼接，而是直接传入起始元素的地址即可。这样能够提高我们并行算法的优化效率。

而对于如何高效筛选数据点所属质心，考虑采用控制流的方式，将每一轮计算出来的数据点到某一质心的距离与存放最近质心距离的数组做差，根据这个差值是否为零来建立控制流，根据这个控制流来筛选是否要更新最近的质心距离和最近的质心标号。

3.2 cache 优化

在初步的并行处理中，我们是每次取出同一行中多个数据点某一维度的数据，接下来是对这多个数据点的各个维度上的数据重复计算距离质心距离的操作。这种操作方式，访问时是按照列去访问的，即当前维度的数据计算完之后，将会到下一行去取下一个维度的数据重复计算操作。当数据规模比较大的时候，这种操作会导致大量 cache miss，因此会产生很多访存操作，这回极大影响并行算法的优化效果。

因此我们考虑对初步的并行算法进行 cache 优化，即尽可能使得数据能够在缓存中命中，来减少访存导致的额外开销。对于并行算法的 cache 优化，我们考虑改变循环的顺序，最外层循环是对维度的迭代，内层循环是对数据的迭代。即每一次内层循环会将所有数据点某一维度与质心之间的距离全部计算出来，然后在外层循环中去迭代各个维度的数据。

这样的改进方式，使得内层循环在每一步中都是相邻的移动，不会出现跨行的数据读取，这样就会提高数据的 cache 命中率，进而能够进一步优化并行算法的效率。

3.3 内存地址对齐

考虑到对数据进行向量化存取的时候，如果取数据的原始内存不是按照存取规模的大小内存对齐的，则会导致在每一次数据向量化读取的时候，由于内存不对齐，而需要进行两次内存访问，然后将得到的数据再进行一次拼接，返回给向量寄存器。当数据量很大的时候，这种额外的开销就会十分明显。

因此考虑对并行算法进行内存对齐。为了能够简化算法实现，并且还能够对比出内存对齐与否对并行算法的影响，我们将数据规模全部设置为 16 的倍数，这样只需要对数组的第一行做一次对齐处理

即可。因为每一行是紧密排列的，因此当对第一行进行对齐处理之后，后续其他行做对应位置的存取时，内存也是对齐的。

对于内存地址的对齐，首先判断存放数据数组的每一行的首地址是否是对齐的，如果是对齐的则可以不做任何处理。如果不对齐，则对该行起始的 m 个未对齐的元素进行串行化处理，由于在这个过程中，后续行的起始 m 个元素也会同样做串行处理，因此在此之后，每一行的剩余部分都是地址对齐的。此外，对于开头做串行处理之后，还可能会导致每一行剩余的元素不足，因此需要对每一行剩余的元素同样做串行处理，保证算法不遗漏数据。

3.4 x86 平台

本次实验除了对 ARM 架构下采用 neon 指令集架构对 KMeans 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。

4 实验结果分析

4.1 ARM 平台

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。为了能够对比出在不同数据规模和不同数据维度下，并行算法的优化效果，我们采用控制变量的方式，分别研究并行算法的优化效果随着数据规模和数据维度两个因素的变化情况。

为了能够保证实验设计在后续内存对齐方面研究相对简单，我们将数据规模全部设定为 4^n，这样在研究内存是否对齐对实验效果的影响时，各行的内存对齐情况是完全一致的，不需要对每一行的数据进行单独的内存对齐调整。为了数据的规整，我们将数据的维度数值也设定为 4^d。

分别调整 n 和 d 的取值，研究串行算法、普通并行算法、cache 优化算法以及内存对齐算法的时间表现情况，具体分析如下。

4.1.1 数据规模角度

选定数据维度为 4^2 时，各种优化算法表现效果随数据规模 n 的变化趋势如表2所示，根据所得的实验数据，计算出各种优化算法的加速比，如图4.1所示。

n	d	serial	parallel	parallel_cache	parallel_aligned
6	2	2.140	1.852	2.317	1.755
7	2	8.527	7.399	8.209	7.377
8	2	91.474	40.036	36.941	38.256
9	2	354.098	164.989	151.499	160.560
10	2	1353.450	583.546	467.250	554.698

表 2: 相同数据维度不同数据规模下的实验结果对比

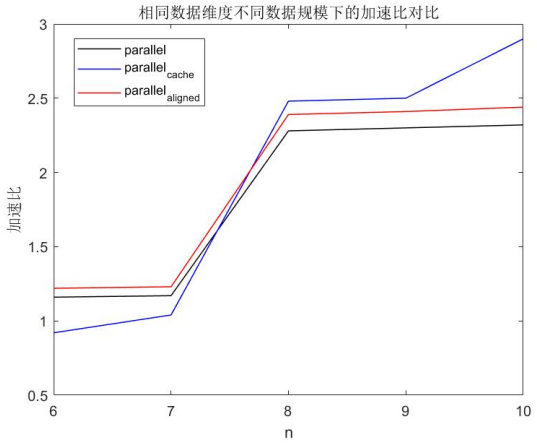

图 4.1: 相同数据维度不同数据规模下的加速比对比

可以发现，在较小数据规模下，串行算法的优化效果并不明显，这是因为串行算法虽然能够同时并行多条数据，但是每一条指令所消耗的时间比加长，因此在较小的数据规模之下，这种优化效果并不明显。尤其是对于 cache 优化的并行算法，由于 cache 优化算法所需要的指令数比较多，因此在较小规模的问题上反倒是表现效果更差。当数据规模达到 4^8 以上时，三种并行优化算法都能够取得十分明显的优化效果。这符合我们常规的认知。通过 perf 进行对三种方法的相关数据进行分析，所得数据如下表3所示。

	cycles	instructions	L1-dcache-loads	L1-dcache-load-misses
serial	22.53%	16.54%	17.61%	58.31%
parallel	10.19%	7.13%	10.35%	14.75%
parallel_cache	9.73%	12.97%	12.58%	3.67%
parallel_aligned	10.71%	6.97%	9.92%	15.74%

表 3: 相同数据维度不同数据规模下 CPU 事件对比

从表格中可以看出，对于未进行 cache 优化的两个并行算法，其优化主要是减少了程序运行的所需要的 cycle，因为其一次可以取出多条数据进行计算，因此其所需要的周期数自然要少于串行算法，这也是普通并行优化算法能够取得性能提升的原因。而对于内存是否对齐，我们可以看到，内存对齐的算法由于不涉及到数据的拼接，因此其所需要的指令数会略低于内存不对齐的算法，这也是其性能要略高于内存未对齐的算法的原因。注意到，利用了 cache 优化的算法当数据规模增大时，具有明显的优化效果，通过观察 perf 的结果可以得到，在相同的数据规模下，未采用 cache 优化的并行算法，其 L1 cache 未命中的概率要远远高于利用 cache 优化的算法，因此随着问题规模的增加，不采用 cache 优化的算法会遇到性能瓶颈，而采用了 cache 优化的算法能够取得更好的性能提升。

4.1.2 数据维度角度

选定数据维度为 4^8 时，各种优化算法表现效果随数据维度 d 的变化趋势如表4所示，根据所得的实验数据，计算出各种优化算法的加速比，如图4.2所示。

n	d	serial	parallel	parallel_cache	parallel_aligned
8	2	91.40	37.17	36.83	35.74
8	3	414.07	177.16	144.12	166.16
8	4	3606.37	1013.93	576.52	954.64
8	5	18240.40	5133.28	2520.60	4792.78
8	6	70869.70	17848.60	9529.51	15436.50

表 4: 相同数据规模不同数据维度下的实验结果对比

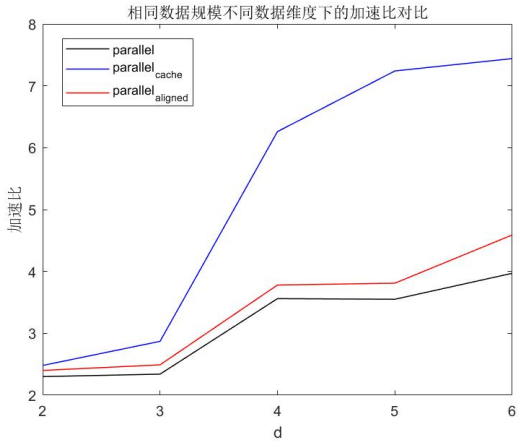

图 4.2: 相同数据规模不同数据维度下的加速比对比

由于我们选定的数据规模已经达到了一定程度，所以在较小的问题规模下，三种并行优化算法都能够取得不错的优化算法。但是随着数据维度的指数级增加，未采用 cache 优化的两种算法在总体的增长趋势和幅度上大致相同，但是采用了 cache 优化的算法却能够取得极高的加速比。未采用 cache 优化的两种算法其加速比基本已经达到了理论上的 4 倍，原因是同时进行了四路展开。而采用了 cache 优化的算法却能够取得远超过 4 倍的加速比，这是因为 cache 优化能够极大的提高数据在缓存中的命中率。而且在本次实验的算法设计中，行存储的是 N 个数据同一维度的数据，而每一列存储的是同一个数据点不同维度的值。因此当数据规模较大，且数据维度较大的时候，每一次访问下一行的维度值时，都可能导致 cache miss，而 cache 优化的算法是按照行进行计算的，因此 cache miss 率比较低。这种效应当数据维度较高时会更加明显。通过 perf 对程序执行过程中的具体指标进行观测，如下表5所示。

	cycles	instructions	L1-dcache-loads	L1-dcache-load-misses
serial	41.46%	16.42%	17.44%	59.23%
parallel	12.83%	6.70%	9.96%	15.66%
parallel_cache	5.91%	12.73%	12.43%	3.31%
parallel_aligned	11.45%	6.53%	9.55%	14.68%

表 5: 相同数据规模不同数据维度下 CPU 事件对比

从表中的数据我们也能够看出，内存对齐和内存不对齐的算法其差异主要体现在指令数上，由于内存不对齐导致数据拼接会产生额外的指令。而 cache 优化后，可以发现，其 cache 命中率是远远高于

串行算法和另外俩各种为进行 cache 优化的算法。而这三种算法的加速比表现均要高于对数据规模探索的实验结果，其原因主要是由于增加了外层循环数，导致串行算法所需要的时钟周期大大增加，而这对并行算法的影响不大，因此三种并行算法的加速比均有较显著的提升。

4.2 x86 平台

在 x86 平台上，主要尝试了 SSE、AVX 和 AVX512 这三种并行指令集架构，编写了不同的串行算法和采用这三种并行架构的并行算法，并且对比了这三种架构下数据是否内存对齐对于并行算法的优化效果的影响。

在 x86 平台下的实验设计思路基本与 ARM 平台一致，即采用向量化的存取手段，一次性存取多个数据元素，准备好连续多个数据某一维度的数值作为一个向量寄存器的数值，再构造一个某一质心该维度对应的一个向量寄存器，然后利用向量化的运算，首先进行对位相减，然后再将差对位相乘，在每一维度累加这个操作的结果，就可以得到这些数据点到某一质心的距离。再通过控制流来判断是否要更新这些数据点所属的划分以及到最近质心的距离。

为了能够充分体现出并行算法的优化效果，进而能够去横向对比不同指令集架构的优化效果，在实验测试时，直接选取了较大的数据规模和较高的数据维度进行测试，当数据规模为 4^6，数据维度为 4^8 时，各种指令集架构下的并行算法优化效果如下表6所示

serial		**SSE_aligned**	**AVX_aligned**	**AVX512_aligned**
15672.2	**time**	4579.1	2628.4	1570.7
	s/p	3.42	5.96	9.98
		SSE_unaligned	**AVX_unaligned**	**AVX512_unaligned**
	time	4776.7	2791.2	1692.8
	s/p	3.28	5.61	9.26
	unalign/align	1.04	1.06	1.08

表 6: 不同指令集并行算法性能对比

通过数据对比可以发现，由于采取的指令集不同，随着向量寄存器所能容纳的数据增多，并行算法的优化效果也在不断提升。这主要是由于采用向量化的优化，能够一次性处理多条数据，当数据规模和数据维度足够大的时候，各种并行算法的理论加速比应该逼近其向量寄存器一次性能够处理的数据的数量。对于内存是否对齐的算法，在本次实验测试中的效果并不明显，但是也能够看出内存对齐的并行算法在各种指令集架构下，均较内存未对齐的并行算法有更好的效果表现。

实验使用了 VTune 性能分析工具对本次实验结果的所对应的 CPU 硬件事件进行进一步的分析，其具体数据如下表7所示。

serial	**SSE_aligned**	**AVX_aligned**	**AVX512_aligned**
536613	144210	76626	39744
	SSE_unaligned	**AVX_unaligned**	**AVX512_unaligned**
	146970	78453	41986

表 7: 不同指令集并行算法指令数对比

通过对比指令数可以发现，三种并行优化算法在指令数上均远远小于串行算法，并且其指令数与串行算法的比值也基本接近加速比，因此可以得出并行算法的优化效果主要体现在能够减少指令数上。对比了对齐和不对齐算法的差异，也主要体现在了算法所需要的指令数上，由于不对齐算法需要进行数据的拼接，因此其指令数均略高于内存对齐的算法。

5 总结

在本次 SIMD 的并行实验中，对 KMeans 算法进行了并行化处理，主要针对算法中的两部分进行优化：第一部分是在每次循环迭代中对所有的数据点计算到各个质心的距离，第二部分是对每个数据点筛选距离其最近的质心进行划分。对这两部分分别进行了向量化的处理，并辅以控制流进行选择控制。实验不仅在 ARM 平台上完成了 Neon 指令集下的并行算法，对比了内存对齐与内存不对齐算法的性能表现，并且对该算法进行了 cache 优化，取得了显著的效果提升。此外，在本次实验中还尝试了在 x86 平台上，选择 SSE、AVX 和 AVX512 指令集架构重构 KMeans 并行算法，并且对比了不同指令集下内存对齐与内存不对齐算法的性能表现。通过使用 perf 和 VTune 等性能分析工具，进一步分析并行算法能够取得性能提升的深层原因，也能够对比发现内存对齐算法和内存不对齐算法之间产生性能差异的原因。本次实验的相关代码和文档已经上传至GitHub。