

浙江大学

本科实验报告

课程名称:	汇编与接口
姓 名:	秦嘉俊
学 院:	计算机科学与技术学院
专 业:	计算机科学与技术
指导教师:	蔡铭

2023 年 12 月 27 日

浙江大学实验报告

课程名称: 汇编与接口 实验类型: 综合
实验项目名称: SIMD 向量化优化 Kmeans 算法
学生姓名: 秦嘉俊 学号: 3210106182 同组学生姓名: 秦嘉俊
实验地点: 玉泉 32 舍 411 实验日期: 2023 年 12 月 25 日

摘要

向量化常常用来优化程序性能。本次探究实验我就基于 Kmeans 算法的经典实现,通过向量化的手段优化 Kmeans 的代码,提出了低维度、高维度下的实现思路。同时我也使用了多线程的方法来优化,得到了高维度下多线程的 Kmeans 算法实现。然后我随机生成数据,测量程序执行时间来对比优化前后的性能差异。

一、背景说明

向量化是一种单指令多数据(即 SIMD)的并行处理数据的方法,是一种常见的提高程序性能的手段。在《汇编与接口》这门课中我学习了 Intel SIMD 的指令集,希望用这些指令来优化程序。

同时在本学期,我在计算机学院某导师的指导下参与了一个项目:在 FPGA 上加速 Kmeans 算法,其中就要将 Kmeans 算法在 FPGA 上的性能和在 CPU 上的性能进行对比。因此我希望借这次探索实验的机会,使用向量化的方法对 CPU 上 Kmeans 算法进行优化,以便更好地对比 FPGA 和 CPU 上的执行效率。

这里简要地介绍经典的 Kmeans 算法,算法过程如下 [1]:

- 初始化,随机选择 K 个样本点作为聚类中心。
- 对样本进行聚类,对于每个固定的类中心,计算每个样本到类中心的距离。将每个样本指派到与其最近的中心的类中,构成聚类结果。
- 计算新的类中心,对聚类结果中的每个类,计算当前类中所有样本的均值,作为新的类中心。
- 如果迭代收敛或符合停止条件,则结束。

二、探索过程

2.1 经典 Kmeans 算法的实现

首先我们实现经典的 Kmeans 算法，这里的实现比较简单，按照算法过程书写即可。

这里 N 表示要聚类的点的数量， D 表示每个点的维度， K 表示聚类的类别数， $ITERATION$ 表示迭代次数。这里 a , $centers$ 都是 `Point` 结构体的数组，里面记录了每个点的坐标，`init()` 方法会初始化这个点，即把坐标全部设为 0.0，`get_dis()` 方法会获得这个点和另一个点的欧几里得距离的平方（即没有开根号）。这部分的逻辑比较简单，这里不做赘述。

Kmeans 部分的代码如下：

```
void standard_km()
{
    int tmp_cnt[K] = { 0 };
    // step 1
    for (int i = 0; i < K; i++) centers[i] = a[i];

    for (int k = 0; k < ITERATION; k++) {
        for (int i = 0; i < K; i++) {
            tmp_cnt[i] = 0;
            update_centers[i].init();
        }
        // step 2
        for (int i = 0; i < N; i++) {
            float min_dis = numeric_limits<float>::max();
            int min_idx = -1;
            for (int j = 0; j < K; j++) {
                float now_dis = a[i].get_dis(centers[j]);
                if (now_dis < min_dis) {
                    min_dis = now_dis;
                    min_idx = j;
                }
            }
            // assign a[i] to centers[min_idx]
            belong[i] = min_idx;
        }
    }
}
```

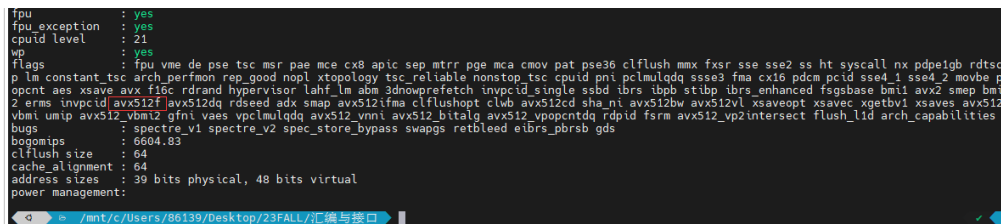
```

        tmp_cnt[min_idx]++;
    for (int j = 0; j < D; j++) {
        update_centers[min_idx].x[j] += a[i].x[j];
    }
}
// step 3
for (int i = 0; i < K; i++) {
    for (int j = 0; j < D; j++) {
        centers[i].x[j] = update_centers[i].x[j] /
            tmp_cnt[i];
    }
}
}
}

```

2.2 向量化的 Kmeans 算法的实现

毫无疑问上面的代码是可以通过向量化进行优化的。为了选择对应的向量化指令集，我使用了 `cat /proc/cpuinfo` 命令，查看了我的 CPU 支持什么样的 SIMD 指令集。结果如下：



```

flags              : fpu vme de pse tsc mtr pae mce cx8 apic sep mtrr pgs mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtsc
p lm constant tsc arch perfmon rep_good nopl xtopology tsc_reliable nonstop_tsc cpuid pni pclmulqdq sse3 fma cx16 pdcm pcid sse4_1 sse4_2 movbe p
opcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid single ssbd ibrs ibpb stibp ibrs_enhanced fsgsbase bmi1 avx2 smep bmi
2 erms invpcid avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves avx512
vbmi umip avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg avx512_vpoperndq rdpid fsrm avx512_vp2intersect flush_lid arch_capabilities
bugs              : spectre_v1 spectre_v2 spec_store_bypass swapgs retbleed eibrs_pbrsb gds
bogomips          : 6604.83
clflush size      : 64
cache_alignment   : 64
address sizes     : 39 bits physical, 48 bits virtual
power management:

```

可以看到我们是支持 AVX512F 指令的，因此我使用了 AVX512F 的相关寄存器。而这里我们的数据是 32 位浮点数，因此这里向量化时，`__m512` 类型的寄存器可以存放 16 个浮点数。

2.2.1 优化 1 – 维度较小时的向量化

首先我想到的是，在经典算法的实现里，我们每次取出一个点，计算这个点与所有中心的距离。每次迭代只计算了一个点的距离，通过向量化我们可以一次在寄存器里加载若干个点的值，这样就可以一次计算多个点的距离，从而提高效率。

这里我们设 `para_d` 并行度为 16，表示 AVX512F 寄存器可以存放 32 个 float 变量。方便起见我们假设这里的 `D` 能被 16 整除，这样我们就可以一次加载 `para_d/D`

个点，而且不用担心除不尽的问题导致最后的尾巴需要特殊处理。

具体的处理如下：

- 首先我们从上面的实现中的 **Point** 结构体数组中加载数据到数组中，以便后续处理。

这里 **data_xy** 表示数据点，**centers_xy** 表示聚类中心点。值得注意的是 **centers_load_xy** 也是表示的聚类中心点，但是区别在于，它是一个 $K \times \text{para_d}$ 的数组，每一行都有 **para_d** 个值，但其实表示的是同一个点，只是重复了 $\text{para_d}/D$ （即 **tuple**，表示一个 512 寄存器里可以放多少个 D 维的点）遍。这样的好处是在后面处理时，我们一次性取出 $\text{para_d}/D$ 个数据点（共 **para_d** 坐标数据）时，我们可以从 **centers_load_xy** 中同样加载出 **para_d** 个坐标数据放到 512 位浮点寄存器里，然后两个寄存器直接运算就可以得到这 $\text{para_d}/D$ 个数据点的距离。

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < D; j++) data_xy[i * D + j] = a[i].x[j];
for (int i = 0; i < K; i++)
    for (int j = 0; j < D; j++) centers_xy[i * D + j] = a[i].x[j];

for (int i = 0; i < K; i++) {
    for (int k = 0; k < D; k++) {
        for (int t = 0; t < tuple; t++) {
            centers_load_xy[i * para_d + k + t * D] =
                centers_xy[i * D + k];
        }
    }
}
```

需要注意的是，后续我们会将 **data_xy**, **centers_xy**, **centers_load_xy** 的数据加载到 512 浮点寄存器中，因此需要内存对齐，所以我们在声明时就要求 64 字节对齐。

```
#define para_d 16
// 需要对齐
__attribute__((aligned(64))) float data_xy[N * D];
__attribute__((aligned(64))) float centers_xy[K * D];
```

```
__attribute__((aligned(64))) float centers_load_xy[K *
    para_d];
__attribute__((aligned(64))) float update_centers_xy[K * D
    ];
```

- 每次迭代中，在计算之前我们也要做相应的初始化操作：`tmp_cnt` 存的是对于每个聚类，有多少点属于这个聚类。这里我们使用向量的方式进行初始化，即每次将 `para_d` 个元素设为 0。需要注意的是这里 `tmp_cnt` 是 $K \times D$ 的，这样我们后续在更新聚类坐标时，可以直接将两个向量相除得到新的聚类中心（一个向量是聚类内数据点的和，一个向量是这个聚类的元素个数重复 `para_d` 次后组成的向量）。

```
__attribute__((aligned(64))) int tmp_cnt[K * D] = { 0 };
...
for (int i = 0; i < (K * D - 1) / para_d + 1; i++) {
    __m512i cnt = _mm512_set1_epi32(0);
    _mm512_store_epi32(tmp_cnt + i * para_d, cnt);
}
```

- 每次迭代中，我们要遍历所有的数据点。这里我们不再是一个点一个点遍历，而是一次性遍历 `tuple` 个点，即 `para_d` 个坐标，这样可以恰好放在一个 512 位浮点寄存器里。

这里我们先将数据点从 `data_xy` 中加载出来，随后初始化 `min_dis`，这里的 `min_dis` 也是一个向量，用来记录数据点与当前被分配到的聚类的距离。还有 `now_belong` 向量寄存器，用来存数据点被分配到哪一个聚类里。注意到这里的 `now_belong` 和 `min_dis` 实际上是有 `para_d` 个数据，但是这里我们只处理了 `tuple` 个点。那么剩余的数据就被我们丢弃了，在这里可以视作无用。（这也是设计的一个缺陷）

```
for (int i = 0; i < N * D / para_d; i++) {
    __m512 point = _mm512_load_ps(data_xy + i * para_d);
    __m512i now_belong = _mm512_set1_epi32(-1);
    __m512 min_dis = _mm512_set1_ps(numeric_limits<float>::
        max());
    ...
}
```

随后对于这几个数据点，我们要做的就是遍历所有的聚类中心，计算数据点与聚类中心的距离，然后更新 `min_dis` 和 `now_belong`。这里计算可以直接使

用 `_mm512_sub_ps` 和 `_mm512_fmadd_ps` 实现欧氏距离的计算。随后我们要做的就是每个数据点的各维度距离差的平方加起来。深思之后，我选择将向量左移 1 位，然后与原向量相加。依次这样执行 D 次，这样 D 维的距离和就在向量每部分的第一个数据里了。

这里以 $D=4$ 为例，假设经过 `_mm512_sub_ps` 和 `_mm512_fmadd_ps` 后结果存在 `tmp` 中，那么我们移位并相加后 `tmp` 的第 1 个浮点数就表示第 1 个点的聚类中心距离，第 5 个浮点数就表示第 2 个点的聚类中心距离，第 9 个浮点数就表示第 3 个点的聚类中心距离，第 13 个浮点数就表示第 4 个点的聚类中心距离。其余浮点数我们这里不予考虑。

为了实现移位这一点，我们使用了 `_mm512_permutexvar_ps` 这个指令，它需要一个索引来指示移位的方式。这里我们使用了 `_mm512_set_epi32` 来生成索引。

```
// 索引，每次往左移动一个 float
__m512i idx = _mm512_set_epi32(15, 15, 14, 13, 12, 11, 10,
    9, 8, 7, 6, 5, 4, 3, 2, 1);
for (int j = 0; j < K; j++) {
    // 这里假设 D <= 16(para_d) 且 D 能被 16 整除
    __m512 center = _mm512_load_ps(centers_load_xy + j *
        para_d);
    __m512 sub_res = _mm512_sub_ps(point, center);
    __m512 square_res = _mm512_set1_ps(0);
    square_res = _mm512_fmadd_ps(sub_res, sub_res,
        square_res);
    __m512 dis_res = square_res;
    // 依次左移，计算出距离和
    for (int k = 1; k < D; k++) {
        __m512 permuted_res = _mm512_permutexvar_ps(idx,
            square_res);
        dis_res = _mm512_add_ps(dis_res, permuted_res);
    }
}
```

得到距离之后，我们就要和 `min_dis` 进行比较，使用 `_mm512_cmp_ps_mask` 这个命令，传入参数 `_CMP_LT_0Q`，表示如果前者小于后者，那么这一部分的结果为 1。随后根据这个掩码，通过 `_mm512_mask_blend_epi32` 命令更新 `now_belong`，即属于哪个聚类。如果是距离更小，即对应的掩码为 1，就把 `now_long` 中对应的值改为 `j`，即我们当前枚举的聚类中心 `j`。

```

__mmask16 mask = _mm512_cmp_ps_mask(dis_res, min_dis,
    _CMP_LT_OQ);
min_dis = _mm512_mask_blend_ps(mask, min_dis, dis_res);
// mask=1 的时候, 说明 dis_res < min_dis, 那么要更新, 选 j
now_belong = _mm512_mask_blend_epi32(mask, now_belong,
    _mm512_set1_epi32(j));

```

遍历完所有的聚类中心后, 每个数据点都有了被分配到的聚类。我们把 因此我们需要计算这个聚类的数据数和坐标和。这里 `tmp_belong` 是一个数组, 用来将 `now_belong` 寄存器中的值存到内存中, 以便后续处理。

```

_mm512_store_epi32(tmp_belong, now_belong);
for (int j = 0; j < tuple; j++) {
    int now_belong_j = tmp_belong[j * D];
    belong[i * tuple + j] = now_belong_j;
    for (int k = 0; k < D; k++) {
        tmp_cnt[now_belong_j * D + k]++;
    }
    for (int k = 0; k < D; k++) {
        update_centers_xy[now_belong_j * D + k] += data_xy
            [(i * tuple + j) * D + k];
    }
}

```

- 遍历完所有点后, 我们就需要更新聚类中心坐标。这里我们也使用向量化的方法, 即每次更新 `tuple` 个聚类点。

```

// 更新聚类中心坐标, 一次性更新 tuple 个
for (int i = 0; i < (K * D - 1) / para_d + 1; i++) {
    __m512 update = _mm512_load_ps(update_centers_xy + i *
        para_d);
    __m512i cnt = _mm512_load_epi32(tmp_cnt + i * para_d);
    __m512 res = _mm512_div_ps(update, _mm512_cvtepi32_ps(
        cnt));
    _mm512_store_ps(update_centers_xy + i * para_d, res);
}

```

随后我们需要将用来存放每次迭代累加和的数组 (`update_centers_xy`) 清

零，并把更新后的聚类中心坐标存回内存中 (`centers_load_xy`)，以便下一次迭代使用。

```
for (int i = 0; i < K; i++) {
    for (int k = 0; k < D; k++) {
        float tmp = update_centers_xy[i * D + k];
        update_centers_xy[i * D + k] = 0.0;
        for (int t = 0; t < tuple; t++) {
            centers_load_xy[i * para_d + k + t * D] = tmp;
        }
    }
}
```

- 最后我们将结果写回到 `Point` 的数组 `centers` 中，以便后续输出。

```
for (int i = 0; i < K; i++) {
    for (int j = 0; j < D; j++) {
        centers[i].x[j] = centers_load_xy[i * para_d + j];
    }
}
```

这里我们实现了低维度下的向量化，但是在后续测试中发现性能不佳，最多带来 23 倍的提速。究其原因，我认为是因为维度低导致向量化的粒度太细，这使得我们操作非常麻烦，而且每次迭代的数据量也不大，导致向量化的优势并没有得到充分发挥。

因此后续我考虑重构代码，对于维度较高的情况进行了向量化。

2.2.2 优化 2 – 维度较大时的向量化

维度较大时，代码的思路更接近 2.1 中的代码。不再需要太多复杂的操作来实现细粒度的控制，具体如下：

- 初始化，我们将 `Point` 中的数据读到 `data_xy` 和 `new_centers` 中。这里 `data_xy` 和 `new_centers` 都是 $K \times D$ 的，即不再需要重复坐标元素。同时这里也需要内存对齐，以便后续向量化操作。

```
__attribute__((aligned(64))) float new_centers[K * D];
__attribute__((aligned(64))) float centers_sum[K * D];
for (int i = 0; i < N; i++)
```

```

        for (int j = 0; j < D; j++) data_xy[i * D + j] = a[i].x
            [j];
    for (int i = 0; i < K; i++)
        for (int j = 0; j < D; j++) new_centers[i * D + j] = a[
            i].x[j];

```

- 随后我们就可以开始迭代了。首先依然是对 `tmp_cnt` 和 `centers_sum` 进行初始化（使用向量化来初始化）。这里 `num_K` 表示多少个向量寄存器可以放下所有的 `K` 个点对应的 `cnt`（这里可能无法整除，因此我们需要额外的循环来处理尾数），`num` 表示一个数据点需要多少个向量寄存器来存放。这里 `tmp_cnt` 表示每个聚类中有多少个数据点（这里是 $K \times 1$ 维的），`centers_sum` 表示每个聚类中所有数据点的坐标和（这里是 $K \times D$ 维的）。

```

__attribute__((aligned(64))) int tmp_cnt[K] = { 0 };
int num = D / para_d;
int num_K = K / para_d;
for (int iter = 0; iter < ITERATION; iter++) {
    __m512 zero_ps = _mm512_set1_ps(0);
    __m512i zero_epi = _mm512_set1_epi32(0);
    // 优先向量化初始化 K 个点的 cnt
    for (int i = 0; i < num_K; i++) {
        _mm512_store_epi32(tmp_cnt + i * para_d, zero_epi);
    }
    // 可能有多余的
    for (int i = num_K * para_d; i < K; i++) {
        tmp_cnt[i] = 0;
    }
    for (int i = 0; i < K; i++) {
        for (int j = 0; j < num; j++) {
            _mm512_store_ps(centers_sum + i * D + j *
                para_d, zero_ps);
        }
    }
    ...
}

```

- 随后我们开始遍历所有的数据点。这里因为一个点过大，我们只能一个一个遍历，在计算与聚类中心的距离时，可以一次性取出 `para_d` 维，计算中心和数据点在这 `para_d` 维上的距离并累加。遍历 `num` 次即可得到与这个中心的距

离。随后我们就可以更新 `min_dis` 和 `min_idx` 了（这里的逻辑与原始算法中相同）。

```
for (int i = 0; i < N; i++) {
    float min_dis = numeric_limits<float>::max();
    int min_idx = -1;
    for (int j = 0; j < K; j++) {
        float now_dis = 0.0;
        for (int k = 0; k < num; k++) {
            __m512 point = _mm512_load_ps(data_xy + i * D +
                k * para_d);
            __m512 center = _mm512_load_ps(new_centers + j
                * D + k * para_d);
            __m512 sub_res = _mm512_sub_ps(point, center);
            __m512 square_res = _mm512_set1_ps(0);
            square_res = _mm512_fmadd_ps(sub_res, sub_res,
                square_res);
            now_dis += _mm512_reduce_add_ps(square_res);
        }
        if (now_dis < min_dis) {
            min_dis = now_dis;
            min_idx = j;
        }
    }
    ...
}
```

接着我们设置好这个数据点属于的聚类，将对应聚类的数量增加，随后可以将这个聚类的坐标和更新。思路依然和原始算法中相同。

```
// assign a[i] to centers[min_idx]
belong[i] = min_idx;
tmp_cnt[min_idx]++;
for (int j = 0; j < num; j++) {
    __m512 point = _mm512_load_ps(data_xy + i * D + j *
        para_d);
    __m512 sum = _mm512_load_ps(centers_sum + min_idx * D +
        j * para_d);
    __m512 new_sum = _mm512_add_ps(sum, point);
    _mm512_store_ps(centers_sum + min_idx * D + j * para_d,
```

```

        new_sum);
    }

```

- 每次迭代遍历完所有数据点后，我们就需要更新聚类中心坐标。这里我们也使用向量的方法，即每次更新 `para_d` 维，对于每个聚类我们只需要 `num` 次既可更新完所有的坐标。

```

for (int i = 0; i < K; i++) {
    for (int j = 0; j < num; j++) {
        __m512 sum = _mm512_load_ps(centers_sum + i * D + j
            * para_d);
        __m512 cnt = _mm512_set1_ps(tmp_cnt[i]);
        __m512 res = _mm512_div_ps(sum, cnt);
        _mm512_store_ps(new_centers + i * D + j * para_d,
            res);
    }
}

```

- 最后我们将结果写回到 `Point` 的数组 `centers` 中，以便后续输出。与之前类似这里不再重复。

可以看到我们的代码和原始算法中类似，向量化也更加直观。不过在测试中，这样的做法也只能带来 34 倍的提升，性能仍然不够理想。

为此我决定基于高维的代码，进行多线程的优化，以期达到更高的性能提升。

2.2.3 优化 3 – 多线程 + 向量化

这里我使用了 `pthread` 来实现多线程。具体的思路是，每次迭代我们要遍历 `N` 个点，这些点可以分到 `THREADS` 个线程分别展开，每个线程只遍历对应一个范围的数据点，并记录他们应该被归类到哪个聚类中，这些线程都遍历一遍后，我们在主线程中进行归类和更新聚类中心的操作。这样迭代 `ITERATION` 次。

- 首先我们定义一个结构体，用来存放线程的信息，包括线程编号（用来计算数据点的对应范围），用来存放坐标和、数量的数组指针。

```

struct ThreadInfo
{
    int id;
    int *len;

```

```
float *sum;
};
```

- 我们进行初始化：首先是把数据拷贝到 `data_xy` 和 `new_centers` 中，与之前类似这里不再重复。接着我们用 `pthread_barrier_init` 初始化一个 `barrier`，以便后续同步线程。随后我们要为 `THREADS` 个线程分配内存，用来存储当前线程下，每个聚类的点数量，以及坐标和。需要注意的是这里也需要按 64 字节对齐，因此我使用了 `_mm_malloc` 这个函数。

```
void *align_malloc(size_t bytes){
    return _mm_malloc(bytes, 64);
}
...
pthread_barrier_init(&barrier, NULL, THREADS + 1);

int **len = (int **)align_malloc(sizeof(int *) * THREADS);
float **sum = (float **)align_malloc(sizeof(float *) *
    THREADS);
for (int i = 0; i < THREADS; i++) {
    len[i] = (int *)align_malloc(sizeof(int) * K);
    sum[i] = (float *)align_malloc(sizeof(float) * K * D);
}
```

并把每个线程的结构体信息初始化：

```
for (int i = 0; i < THREADS; i++) {
    p[i].id = i;
    p[i].len = len[i];
    p[i].sum = sum[i];
}
```

- 开启多线程，每轮迭代的时候，我们都创建 `THREADS` 个线程，每个线程都会执行 `per_km` 函数，传入的参数是线程的结构体信息。

```
for (int iter = 0; iter < ITERATION; iter++) {
    for (int i = 0; i < THREADS; i++) {
        memset(len[i], 0, sizeof(int) * K);
        memset(sum[i], 0, sizeof(float) * K * D);
        pthread_create(&Thread[i], NULL, per_km, &p[i]);
    }
}
```

```
}  
...
```

- `per_km` 函数内做的事情和 2.2.2 中的每次迭代的任务类似。不同点在于，对于每个线程，我们拿到他的 `id`，随后计算出这个线程负责的数据点范围。我们规定这个范围是 `id * (N / THREADS)` 到 `(id + 1) * (N / THREADS)`，如果是最后一个线程，那么就到 `N`。同时我们更新 `tmp_cnt`，`center_sum` 时直接基于结构体中传入的 `len`，`sum` 即可。

需要注意的是中间我们需要 `pthread_barrier_wait` 函数来确保线程的进度一致，最后要使用 `pthread_exit(NULL)`；销毁线程。

```
void *per_km(void *info)  
{  
    int num = D / para_d;  
    int num_K = K / para_d;  
    int id = ((ThreadInfo *)info)->id;  
    int st_point = id * N / THREADS;  
    int ed_point = (id + 1) * N / THREADS;  
    int *tmp_cnt = ((ThreadInfo *)info)->len;  
    float *centers_sum = ((ThreadInfo *)info)->sum;  
    pthread_barrier_wait(&barrier);  
    for (int i = st_point; i < min(ed_point, N); i++) {  
        float min_dis = numeric_limits<float>::max();  
        int min_idx = -1;  
        for (int j = 0; j < K; j++) {  
            float now_dis = 0.0;  
            for (int k = 0; k < num; k++) {  
                __m512 point = _mm512_load_ps(data_xy + i *  
                    D + k * para_d);  
                __m512 center = _mm512_load_ps(new_centers  
                    + j * D + k * para_d);  
                __m512 sub_res = _mm512_sub_ps(point,  
                    center);  
                __m512 square_res = _mm512_set1_ps(0);  
                square_res = _mm512_fmadd_ps(sub_res,  
                    sub_res, square_res);  
                now_dis += _mm512_reduce_add_ps(square_res)  
                    ;  
            }  
        }  
    }  
}
```

```

        }
        if (now_dis < min_dis) {
            min_dis = now_dis;
            min_idx = j;
        }
    }
    // assign a[i] to centers[min_idx]
    belong[i] = min_idx;
    tmp_cnt[min_idx]++;
    for (int j = 0; j < num; j++) {
        __m512 point = _mm512_load_ps(data_xy + i * D +
            j * para_d);
        __m512 sum = _mm512_load_ps(centers_sum +
            min_idx * D + j * para_d);
        __m512 new_sum = _mm512_add_ps(sum, point);
        _mm512_store_ps(centers_sum + min_idx * D + j *
            para_d, new_sum);
    }
}
pthread_barrier_wait(&barrier);

pthread_exit(NULL);
}

```

- 每轮迭代中，新线程都执行完毕后，我们就可以在主线程中进行归类和更新聚类中心的操作了。这里我们依次处理每个聚类中心，首先是统计所有线程中的 **len**，得到这个聚类的数据点个数，随后统计所有线程中的 **sum**，得到这个聚类的坐标和。最后将计算均值得到新的中心坐标，写回到 **new_centers** 中，以便下一轮迭代使用。

```

pthread_barrier_wait(&barrier);
pthread_barrier_wait(&barrier);

for (int i = 0; i < K; i++) {
    int tmp_cnt = 0;
    for (int j = 0; j < THREADS; j++) {
        tmp_cnt += len[j][i];
    }
}

```

```

        if (tmp_cnt <= 0) { continue; }
        for (int j = 0; j < num; j++) {
            __m512 tmp_sum = _mm512_set1_ps(0);
            for (int k = 0; k < THREADS; k++) {
                tmp_sum = _mm512_add_ps(tmp_sum, _mm512_load_ps
                    (sum[k] + i * D + j * para_d));
            }
            __m512 tmp_cnt_ps = _mm512_set1_ps(tmp_cnt);
            __m512 res = _mm512_div_ps(tmp_sum, tmp_cnt_ps);
            _mm512_store_ps(new_centers + i * D + j * para_d,
                res);
        }
    }

    for (int i = 0; i < THREADS; i++) {
        pthread_join(Thread[i], NULL);
    }
}

```

需要注意的是，这里最后我们要通过 `pthread_join` 来等待所有线程执行结束。

- 最后我们将结果写回到 `Point` 的数组 `centers` 中，以便后续输出。与之前相同这里不再重复。

经过多线程 + 向量化的优化，我们的程序性能终于有了显著的提升。在有些条件下已经可以得到 8、9 倍的增速。到此我的优化工作就告一段落了。

2.3 测试思路

`defs.h` 里有基本的 `struct Point` 的定义，以及 `N`, `D`, `K`, `ITERATION` 的参数设置。需要调整输入规模和迭代次数时直接在这里调整即可。

为了方便测试，我在 `test.cpp` 里书写了对应的测试代码。测试方式是，先随机生成数据，对于给定的数据点个数 `N`，维数 `D`，其每一维的坐标都由 `rand()` 随机生成，并且每个数据点的坐标都在 (0,1) 之间。随后我们依次调用标准的 `kmeans` 算法和优化后的 `kmeans` 算法，统计并返回运行时间。

为了检查代码的正确性，我还编写了一个函数 `output_result`，将聚类后的结果（每个数据点属于哪个聚类，聚类中心的坐标）输出到 `txt` 文件中，随后我们有一个 `draw.py` 来画出聚类的结果（每次只能画一个文本的内容，因此需要手动调整

里面的输入文件），以便观察。（这里因为平面仅支持 2 维的情况，所以我们只会在 $D=2$ 的时候使用 `draw.py` 进行观察）这里它会把数据点都标出，同时不同的聚类会使用不同的颜色来标明，对应的聚类中心也是相同的颜色，并用星号突出。

```
/* 随机生成数据 */
void init_dataset()
{
    for(int i = 0; i < N; i++) {
        for (int j = 0; j < D; j++) {
            a[i].x[j] = (float)(rand()%100) / 100.0;
        }
    }
}

/* 输出结果到文件 */
void output_result(string s)
{
    ofstream tmp;
    tmp.open(s, ios::out);
    tmp << N << " " << K << endl;
    for(int i = 0; i < N; i++) {
        tmp << a[i].x[0] << " " << a[i].x[1] << " " << belong[i] << endl;
    }
    for (int i = 0; i < K; i++) {
        tmp << centers[i].x[0] << " " << centers[i].x[1] << endl;
    }
    tmp.close();
}

int main()
{
    init_dataset();
    cout << "Standard_km_start!" << endl;
    double st = clock();
    standard_km();
    double ed = clock();
    cout << "TIME:" << (ed - st) / CLOCKS_PER_SEC << "s" << endl;
    // output_result("res1.txt");
}
```

```

/* 对于低维我们调用这个函数，否则调用后面的两个函数 */
if (D < 16) {
    cout << "AVX_km(for_low_d)_start!" << endl;
    st = clock();
    km_avx();
    ed = clock();
    cout << "TIME:_" << (ed - st) / CLOCKS_PER_SEC << "s" <<
        endl;
    output_result("res2.txt");
    return 0;
}

cout << "AVX_km(for_high_d)_start!" << endl;
st = clock();
km_avx_higher_d();
ed = clock();
cout << "TIME:_" << (ed - st) / CLOCKS_PER_SEC << "s" << endl;
// output_result("res3.txt");

/* 多线程时要使用 timespec 来测量时间 */
struct timespec start, end;
cout << "Multithread_km_start!" << endl;
clock_gettime(CLOCK_MONOTONIC, &start);
km_avx_multithread();
clock_gettime(CLOCK_MONOTONIC, &end);
double time_taken = end.tv_sec - start.tv_sec;
time_taken += (end.tv_nsec - start.tv_nsec) / 1000000000.0;
cout << "TIME:_" << time_taken << "s" << endl;
// output_result("res4.txt");
}

```

使用 `g++ -mavx512f km.cpp test.cpp -lpthread` 命令即可编译出我们的可执行文件 `a.out`，随后运行即可。如果想要运行我们的画图程序，需要安装 `matplotlib`，并且需要在 `draw.py` 中手动修改输入文件名，随后运行 `python3 draw.py` 即可。

对于不同的算法结果，我们可以通过 `diff` 命令来查看输出的结果是否一致，如果没有输出，说明结果一致。

三、效果与结论

实验环境：

- 操作系统：Windows 10 64 位，Windows Subsystem for Linux (WSL) Ubuntu 20.04.5
- 编译器：gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1 20.04.2)
- CPU：11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz
- 支持 AVX512 系列指令集

3.1 正确性测试

- 对于 $D=2$ 的情况，我们可以通过 `draw.py` 查看聚类结果。这里以 $N=16$ ， $k=4$ ， $ITERATION=100$ 为例：

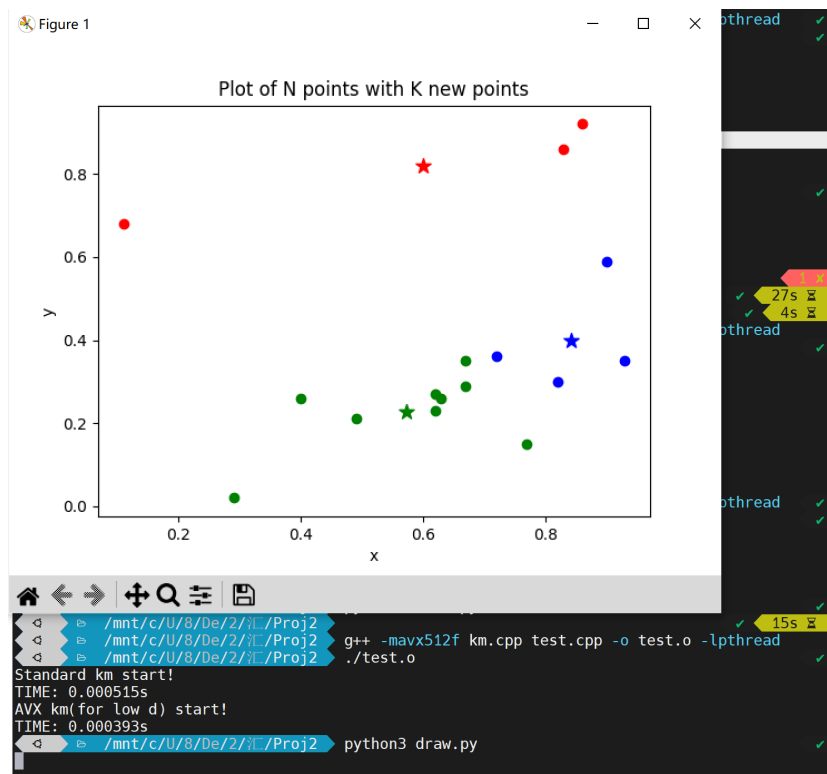


图 1: 原始 Kmeans 算法聚类结果

对于其他算法的结果，我们可以通过 `diff` 命令来查看，如果没有输出，说明结果一致。在当前情况下，低维，因此我们只需要考虑原始算法和 2.2.1 中的算法的比较。

```

< > /mnt/c/U/8/De/2/Proj2 diff res1.txt res2.txt
< > /mnt/c/U/8/De/2/Proj2

```

图 2: 原始 Kmeans 算法和 2.2.1 中的算法的聚类结果

- 对于 D 较大时的情况，我们比较结果：（这里以 N=1024, D=16, K=4, ITERATION=1000 为例）

```

< > /mnt/c/U/8/De/2/Proj2 g++ -mavx512f km.cpp test.cpp -o test.o -lpthread
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.285371s
AVX km(for high d) start!
TIME: 0.076007s
Multithread km start!
TIME: 0.074303s
< > /mnt/c/U/8/De/2/Proj2 diff res1.txt res3.txt
< > /mnt/c/U/8/De/2/Proj2 diff res1.txt res4.txt
1026c1026
< 0.575645 0.571693
> 0.575645 0.571694
< > /mnt/c/U/8/De/2/Proj2

```

图 3: 原始 Kmeans 算法和 2.2.2, 2.2.3 中的算法的聚类结果

可以看到结果基本一致，除了在线程的算法中，因为我们统计 batch 的方式有了变化，所以可能会有精度上的损失，但是不影响结果。

3.2 性能测试

篇幅原因，这里仅展示部分结果。

- 2.2.1 中，对于低维度的测试：

```

< > /mnt/c/U/8/De/2/Proj2 g++ -mavx512f
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.534567s
AVX km(for low d) start!
TIME: 0.259084s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.457223s
AVX km(for low d) start!
TIME: 0.268407s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.536025s
AVX km(for low d) start!
TIME: 0.298669s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.496147s
AVX km(for low d) start!
TIME: 0.272313s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.452244s
AVX km(for low d) start!
TIME: 0.256686s
< > /mnt/c/U/8/De/2/Proj2

```

```

< > /mnt/c/U/8/De/2/Proj2 g++ -mavx512f k
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 6.19099s
AVX km(for low d) start!
TIME: 3.31962s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 5.82544s
AVX km(for low d) start!
TIME: 3.16637s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 6.24798s
AVX km(for low d) start!
TIME: 3.21498s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 5.77949s
AVX km(for low d) start!
TIME: 3.13763s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 6.14253s
AVX km(for low d) start!
TIME: 3.1913s
< > /mnt/c/U/8/De/2/Proj2

```

```

< > /mnt/c/U/8/De/2/Proj2 g++ -mavx512f
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.915296s
AVX km(for low d) start!
TIME: 0.626319s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.729072s
AVX km(for low d) start!
TIME: 0.578101s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.823158s
AVX km(for low d) start!
TIME: 0.623054s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.878266s
AVX km(for low d) start!
TIME: 0.694029s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 0.874589s
AVX km(for low d) start!
TIME: 0.685187s
< > /mnt/c/U/8/De/2/Proj2

```

```

TIME: 0.002103s
< > /mnt/c/U/8/De/2/Proj2 g++ -mavx512f
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 9.6093s
AVX km(for low d) start!
TIME: 7.21071s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 9.78965s
AVX km(for low d) start!
TIME: 8.06309s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 9.70725s
AVX km(for low d) start!
TIME: 6.94376s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 8.90795s
AVX km(for low d) start!
TIME: 6.83781s
< > /mnt/c/U/8/De/2/Proj2 ./test.o
Standard km start!
TIME: 9.51733s
AVX km(for low d) start!
TIME: 6.89078s
< > /mnt/c/U/8/De/2/Proj2

```

(a) N=1024, D=2, K=4, (b) N=10240, D=2, K=4, (c) N=1024, D=4, K=4, (d) N=10240, D=4, K=4, ITER=10000

```
g++ -mavx512f -std=c++11 -O3 -c *.cpp
g++ -mavx512f -std=c++11 -O3 *.o -o test.o
Standard km start!
TIME: 16.9312s
AVX km (for low d) start!
TIME: 17.3872s
Standard km start!
TIME: 14.7074s
AVX km (for low d) start!
TIME: 17.5252s
Standard km start!
TIME: 15.9709s
AVX km (for low d) start!
TIME: 17.1691s
Standard km start!
TIME: 15.2943s
AVX km (for low d) start!
TIME: 17.8347s
Standard km start!
TIME: 14.439s
AVX km (for low d) start!
TIME: 18.1971s
```

```
g++ -mavx512f -std=c++11 -O3 -c *.cpp
g++ -mavx512f -std=c++11 -O3 *.o -o test.o
Standard km start!
TIME: 19.021s
AVX km (for low d) start!
TIME: 3.98175s
Standard km start!
TIME: 4.16794s
Standard km start!
TIME: 19.6885s
AVX km (for low d) start!
TIME: 4.16794s
Standard km start!
TIME: 19.3801s
AVX km (for low d) start!
TIME: 3.81528s
Standard km start!
TIME: 9.91934s
AVX km (for low d) start!
TIME: 3.91292s
Standard km start!
TIME: 9.59848s
AVX km (for low d) start!
TIME: 3.9157s
```

```
g++ -mavx512f -std=c++11 -O3 -c *.cpp
g++ -mavx512f -std=c++11 -O3 *.o -o test.o
Standard km start!
TIME: 17.8589s
AVX km (for low d) start!
TIME: 5.66918s
Standard km start!
TIME: 17.8025s
AVX km (for low d) start!
TIME: 5.66152s
Standard km start!
TIME: 17.9563s
AVX km (for low d) start!
TIME: 6.1881s
Standard km start!
TIME: 17.0942s
AVX km (for low d) start!
TIME: 5.39324s
Standard km start!
TIME: 12.7116s
AVX km (for low d) start!
TIME: 5.39536s
```

```
g++ -mavx512f -std=c++11 -O3 -c *.cpp
g++ -mavx512f -std=c++11 -O3 *.o -o test.o
Standard km start!
TIME: 31.9243s
AVX km (for low d) start!
TIME: 9.1126s
Standard km start!
TIME: 30.3805s
AVX km (for low d) start!
TIME: 8.845s
Standard km start!
TIME: 8.8781s
Standard km start!
TIME: 31.6675s
AVX km (for low d) start!
TIME: 9.7927s
Standard km start!
TIME: 31.6675s
AVX km (for low d) start!
TIME: 8.93062s
```

(e) N=10240, D=8, K=4, (f) N=10240, D=8, K=8, (g) N=10240, D=8, K=16, (h) N=10240, D=8, K=32, ITER=10000

可以看到向量化的确对于程序的效率有所提升。但是对于 2.2.1 中的算法，随着数据点个数的增加，性能提升并不明显。至多只有接近 2 倍的加速，且随着数据规模的增加，性能提升越来越低。

但是我们如果增加聚类的数量 K，可以看到此时可以得到 3 倍的加速，且随着 K 增加我们的性能提升也越来越明显，可以达到接近 4 倍。

我认为，这是因为在我们的实现中，我们是一次性往向量里放入若干个点的，也就是说我们能取得的并行度取决于 D 的大小。随着 D 增大我们能放入一个向量的点的数量就增加了，因此向量化算法能获得的增益就少了。

- 2.2.2、2.2.3 中，对于高维度的测试：

```
g++ -mavx512f -std=c++11 -O3 -c *.cpp
g++ -mavx512f -std=c++11 -O3 *.o -o test.o
Standard km start!
TIME: 33.506s
AVX km (for high d) start!
TIME: 12.6023s
Multithread km start!
TIME: 16.479s
Standard km start!
TIME: 30.5587s
AVX km (for high d) start!
TIME: 11.9729s
Multithread km start!
TIME: 18.1443s
Standard km start!
TIME: 30.4037s
AVX km (for high d) start!
TIME: 12.3372s
Multithread km start!
TIME: 19.822s
```

```
g++ -mavx512f -std=c++11 -O3 -c *.cpp
g++ -mavx512f -std=c++11 -O3 *.o -o test.o
Standard km start!
TIME: 66.5447s
AVX km (for high d) start!
TIME: 21.3702s
Standard km start!
TIME: 59.7764s
AVX km (for high d) start!
TIME: 20.7714s
Standard km start!
TIME: 59.5122s
AVX km (for high d) start!
TIME: 18.3793s
Multithread km start!
TIME: 20.6016s
```

```
g++ -mavx512f -std=c++11 -O3 -c *.cpp
g++ -mavx512f -std=c++11 -O3 *.o -o test.o
Standard km start!
TIME: 112.314s
AVX km (for high d) start!
TIME: 31.2925s
Multithread km start!
TIME: 28.1056s
Standard km start!
TIME: 111.744s
AVX km (for high d) start!
TIME: 32.8184s
Multithread km start!
TIME: 28.1107s
Standard km start!
TIME: 112.023s
AVX km (for high d) start!
TIME: 32.1546s
Multithread km start!
TIME: 27.8396s
```

(i) N=10240, D=16, K=4, (j) N=10240, D=32, K=4, (k) N=10240, D=64, K=4, ITER=10000

(l) $N=10240, D=16, K=8, (m) \quad N=10240, \quad D=16, (n) \quad N=10240, \quad D=16,$
 $ITER=10000 \quad K=16, ITER=10000 \quad K=32, ITER=10000$

可以看到，高维度下 2.2.2 中的算法能够得到 34 倍的加速，且随着 D, K 的增加，性能提升也越来越明显。此外我们可以看到多线程的算法的效率与当前数据规模有关，当数据规模较小时，多线程的算法效率并不高，相反因为频繁的创建线程、销毁线程以及最后更新聚类的操作反而会消耗更多的时间导致此时 2.2.3 的算法可能耗时比 2.2.2 算法更久，但是随着数据规模的增加，多线程的算法效率也越来越高。

为了探究多线程算法的效率，我们还展示一组测试数据：这里我们是针对数据集较大的情况，其中 $N=102400, 1024000, 409600, D=32, 64, 64, K=4$ 的情况。

图 2: 多线程算法的效率

可以看到当规模较大时，多线程的算法能够得到 78 倍的加速。此外在我调整参数以及规模时，发现在有的时候多线程的算法能够达到十倍的加速比，这就取决于当前的数据集的分布情况，以及聚类的数量等因素。但是总体来说，多线程的算法能够得到较好的加速比。

四、实验体会与经验教训

在这次探究实验之前，我仅在翁恺老师的《程序设计与算法基础》中体验过并行计算，其他时候都是在单线程的环境下编程，也没有考虑过通过向量化的方法优化代码。因此这次实验对我来说是一次全新的体验，为我打开了一扇新的大门。

在自己实践中，我的确感受到向量化给代码性能的提升；同时也发现虽然上课时听老师讲例子时较为容易，而且看上去思路非常“简单”的向量化，实现起来并不轻松。在改写 Kmeans 代码时，我时常陷入不知道怎么写的困境。而且因为之前没有接触过 AVX 指令集，在需要用到的时候也需要去查阅 Intel 手册和网上的一些博客，才能够理解和使用。

但当我真正完成了这个实验，看到了性能的提升，我感到十分的过瘾。经过向量化和多线程的优化，我们的代码能够得到如此大的性能提升，这让我不禁感到之前写的代码是多么浪费 CPU 资源，效率是多么的低下。我认为**让代码能够正确的运行，让代码能够更快的运行**，是每一个程序员的两个最终目标。而这次探究实验很好地填补了我这方面的空白。

而且对于未来的科研、学习，并行计算的知识都对我有着重要的意义，我原先的实验室项目恰好需要 CPU 上 Kmeans 的并行程序来做对比实验，而这次实验就给我提供了这次机会。

比较可惜的是，最后对代码的性能也没能达到优化十倍的程度，这与我的电脑本身的性能、实验环境、我的代码实现都有关。正如我之前所说程序员应该让代码能够更快的运行，我们对性能的追求也不应停在此。在这次实验中，我也发现了一些可以继续优化的地方。虽然本次探究实验暂时告一段落，但我会以此为基础，继续优化已有的代码以期达到更高的性能（比如让代码能在大部分时候达到十倍以上加速）。同时学习在 High Performance Computing 中的其他工具，丰富我的技术栈，这样才能更好地优化。对于其他代码我也会运用本次向量化实验的经验和知识，尽可能地优化性能。

非常感谢老师和《汇编与接口》这门课程能够给我这次学习和探索向量化的机会，让我能够在这个方向上有所收获。

综上所述，如有错误还请老师批评指正。

五、参考资料

- [1] 《统计学习方法》，李航，清华大学出版社，14.3 节 k 均值聚类
- [2] [Intel® Intrinsics Guide](#)