Report of Project AKNN

邢庆辉

代码汇总

一、基础题

代码见 Github

1. 实现思路（主要参考[1]）

(1) 算法流程

Algorithm. 1	KNN Search(D, G, S, L, k, Q)
Input： Base Data D, constructed KNN-Graph G, Candidates Pool S, size of S L, required number of nearest neighbors k $(k \leq L)$, Query Point Q	
Output： k nearest neighbors of Q。	
1： randomly choose a starting point p from D; $S = \emptyset$	
2： insert p into S	
3： **loop**	
4： choose the first point s in S which has never been starting point before	
5： mark s as *'has been starting point'*	
6： **for** all neighbors ns of s in G	
7： insert ns into S and keep S in increasing order on Euclidean distance to Q	
8： **end for**	
9： **if** S.length() > L	
10： S.resize(L)	
11： **end if**	
12： **until** no such s in S	
13： return first k point (their indices) in S	

(2) 数据结构

candidate pool

candidate pool 的结构是一维线性表(std:: vector)，基本元素为代表数据点的结构体(struct)。数据点结构体定义如下：

```
struct neighbor {
    unsigned id; //global index
    float distance; //Euclidean Distance(square) to query point
    bool flag; // true if checked
}
```

（**flag**标记此数据点是否曾被考虑过纳入candidate pool）

Algorithm.1 默认使用简单插入排序向S中合适位置插入数据点。使用二分查找能降低时间复杂度，在L较大的情况下，改进更加明显。（Fig.2，Table.1）

基本数据的载入格式

将Base Data, Query Data, Ground-Truth, KNN-Graph均按以下规则以一维数组 d 载入。以SIFT-1M的Base Data为例，数据点特征维度为128，共有10^6个数据点：

$$d[i * dim + j] = D[i][j] \qquad (1)$$
$$dim = 128, \qquad (0 \leq i < 10^6, 0 \leq j < 128), \qquad i, j \epsilon \mathbb{R}^n$$

其中，D为10^6行,128列的矩阵，每行代表一个数据点，从上到下，按 Base Data 中的索引升序排列。d(0-based)的长度为$10^6 * 128$。

(3) 讨论

算法在 SIFT-1M 和 GIST-1M 的实验效果如 Fig.1 所示

缺点及改进

① 缺点：

GIST-1M数据维度高，由于“维度灾难”[4]，L2距离无法体现数据点的关系，因此造成查询性能恶化

可能的改进方向：

对高维数据，使用PCA预处理数据，降低维度到可以接受的范围。（最好能降到不同维度，观察维度与搜索性能曲线的关系，看是否有最佳维度；Query Data同样处理）

② 缺点：

KNN-Graph图结构较冗余，带来较大的内存载入。对SIFT-1M，载入KNN-Graph所需内存已经与Base Data所需内存相当。

从原理上，KNN-Graph将每个点最近邻的K个数据点连接起来作为图的边，这并不是针对KNN-Search问题的数据结构，因此可能会存在冗余信息。（Fig.3 和Fig.4显示，60NN到100NN之间的QPS-Precision曲线非常接近）

可能的改进方向：

更理想的状态应该是MSNET（Monotonic Search NETwork）[2]。而KNN-Graph可以作为构建MSNET过程中的一种优化手段，比如[2]从KNN-Graph而不是全局构建NSG，从而降低建立索引时间(indexing time)。

③ 缺点：

面对数据库需要大量快速增删的情况，KNN-Graph的更新要遍历整个数据库，更新索引时间(indexing time)较大。

优点：

精度高速度快：相对于严格的遍历全局的最近邻搜索算法，KNN-Graph的速度有所提高，同时能保持较高的精度。[1]

2. Queries per Seconds – Precision & 峰值内存占用

(1) 实验设置：

- 简单 L2 距离计算 + 直接插入排序
- 搜索最近邻点数目 $K = 100$
- 100NN-Graph
- $L \in [100:10:400]$[1]
- Dataset : SIFT-1M, GIST-1M
- CPU i5-7200U

(2) QPS-Precision：

算法在 SIFT-1M 和 GIST-1M 的实验效果如 Fig.1 所示。其中，GIST-1M 的曲线靠近左下角，远低于在 SIFT-1M 的性能。即，GIST-1M 相比 SIFT-1M：

① 精度低：可能由于GIST-1M数据维度较高，出现“维度灾难”，即，当数据维度过高时，欧式距离计算难以反映数据点之间的关系[4]。可以考虑使用PCA降维后数据做最近邻查找。

② 速度慢：可能由于数据点维度更高，简单L2距离计算方法需要更长时间。

(3) 峰值内存占用：

实验方法：

实验中记录峰值内存占用的方法是：针对一个确定的L值，统计处理 10k (1k for GIST-1M)个待查询点(Query Data)的过程中程序占用的峰值。实际过程发现，**当其他条件相同时，不同L值对应的内存占用量差别很小（在$1MB$之内）**，因此不区分同一条件下不同L值的内存占用。

内存记录工具：Visual Studio 2017 诊断工具，Windows10 任务管理器

结果与分析：

SIFT-1M的峰值内存占用为$\mathbf{885.5MB}$, GIST-1M的峰值内存占用为$\mathbf{4056.4MB}$.

内存占用主来来自于Base Data。但对于SIFT-1M来说，100NN-Graph的内存占用量与Base Data相当。我们做一个不严格的实验证明这个论断：对SIFT-1M，其他实验参数不变，在运行过程中只读入100NN-Graph，得到峰值内存占用为382.8MB；如果只读入Base Data，得到峰值内存占用为489.8MB。两个数值接近。

[1] 即，L从 100 开始，每次增加 10，直到 400，包含两端数据。报告余下部分此符号表示意义相同，不再说明

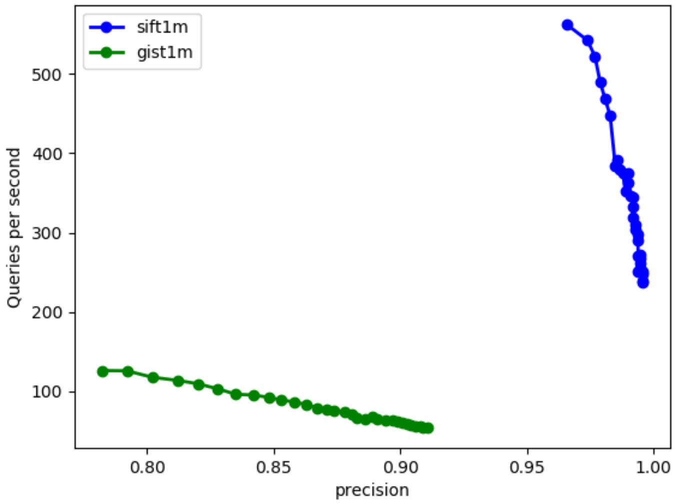

Figure 1. QPS-Precision. **同一种算法，相比 SIFT1M，GIST1M 速度低、精度低。精度低主要由于 GIST-1M 数据点高维度引发“维度灾难”，使得 L2 距离无法有效表示数据点之间的关系。速度低是由于数据维度较高，使用简单 L2 距离计算方式，计算两个点之间的距离所需时间长。**

3. 选做题 (1) & (3)

(1) 实验设置

- AVX-256 / 简单L2距离计算 + 直接插入排序 / 二分查找优化
- $L \in [400:100:3000]$
- 搜索最近邻点数目 $K = 100$
- 100NN-Graph
- Dataset: SIFT-1M
- CPU i5-7200U

为了更加明显地对比 AVX 和二分插入排序的优化效果，分别在 a) 简单 L2 距离计算, b) 二分插入排序优化, c) AVX 优化，d) 二分插入排序+AVX 优化，四种情况下，测量 QPS-Precision 性能。详细结果见 Fig.2 和 Table.1

(2) 二分查找优化

二分查找主要优化阶段在将新元素插入 candidate pool 步骤，使用二分查找代替遍历查找降低时间复杂度。由于优化效果在L较大的情况下比较明显，因此选择在$L \in [400:100:3000]$的范围进行对比。从 Fig.2 和 Table.1 可以看出二分查找对速度有明显的优化。

(3) AVX256

AVX-256 的优势在于“并行计算”，主要针对 L2 距离计算阶段进行优化。不同 AVX 的版本有不同的寄存器长度，本次实验选择 AVX-256。[10, 11]

(4) 实验结果与分析

AVX和二分插入排序的优化部分不同，所以可以协同使用，使优化效果叠加。实验显示，与简单L2距离计算相比，叠加优化后，**同样的精度，速度提升了一倍左右**。

另外，由于 AVX 和二分插入排序只是优化了距离计算和排序过程，精度仍然只与L有关。观察 Fig.2 可以发现，不同优化方式的数据点在 precision 轴的位置相同，可以验证这个论断。

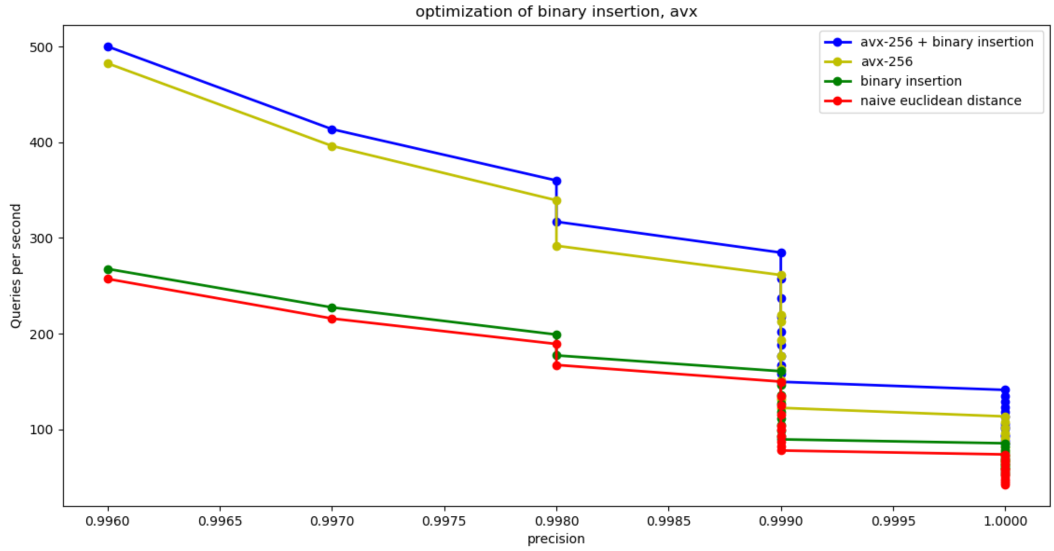

Figure 2. AVX-256 和二分查找的优化效果。(1)AVX-256 的优化能力主要体现在"并行计算"，尤其对高维数据计算有明显的优化效果。(2)二分查找主要对 candidate pool 插入新元素的查找过程优化，因此当L较大时优化效果更加明显。另外，Fig.2 中 precision=1 处的数据较密集，所以把更直观的比较结果放在了 Table 1

Table 1. 在 SIFT-1M 数据集上，不同方法在 10k 个待查询点(Query Data)上的搜索用时。在 precision=1 的结果中选取 4 个代表。其中 AVX 代表只有 AVX256 优化结果，Bi-Insert 代表只有二分查找优化结果，Both 代表同时应用两种技巧的优化结果。由于 AVX 和二分插入排序的优化区域不同，因此优化效果可以叠加。**叠加以后，同样的精度，速度提升了一倍左右**。（由于 AVX 和二分查找只减小计算复杂度，不影响算法精度，因此 precision 只和L有关）

$search\ time\ on\ 10K\ query\ points\ of\ different\ methods\ on\ SIFT-1M\ (seconds)$				
$Both$	AVX	$Bi-Insert$	$Naïve$	$\#\ L$
117.65	166.58	190.15	235.36	3000
99.52	137.51	158.99	197.44	2600
84.67	110.33	137.76	169.74	2200
70.84	88.13	117.13	135.64	1800

4. （拓展）XNN-Graph 的性能比较

(1) 问题描述

改变 KNN-Graph 的 K 值，即改变图中数据点的出度，观察搜索性能变化。

(2) 直观推测

降低 KNN-Graph 的 K 值，让图结构更加稀疏，因此 K 值减小可能带来：

缺点：a)精度降低：KNN 图结构稀疏化后，可能丧失信息。例如由 100NN 减少到 80NN，删除的邻居点中可能包含所有邻居中距待查询点(Query Data)最近的邻居。

优点：a)速度升高：每个结点的出度减小，对每个点的搜索次数减少了，速度可能会提高。b)内存占用降低：图结构稀疏后，可以使用更少的内存来存储 KNN 图。对类似 SIFT-1M 这种维度 128 很接近 100 的数据集，100NN-Graph 占用内存量和 base data 占用内存量已经相当，因此如果精度损失不是很大，降低 K 值也不失为降低内存占用和提高速度的好方法。

(3) 实验设置

我们仍然使用 Algorithm.1 进行最近邻搜索，使用不同稀疏程度的图结构（XNN-Graph）,分别测试在 SIFT-1M 上的搜索性能。

- 100NN 到 10NN，步长 10NN。
- AVX+二分插入排序
- 搜索最近邻点数目 $K = 100$
- $L = [400:100:1800]$
- Dataset: SIFT-1M
- CPU i5-7200U
- 实验过程中只载入需要的部分图结构。
- 度量目标：precision，QPS，内存占用

(4) 实验结果与分析：

不同 K 值对应的 QPS-Precision 曲线和内存占用分别在 Fig.3，Fig.4。 分析实验结果，有如下结论

① K 值越小，搜索精度越低、内存占用越小、搜索速度越快

Fig.3 中，K 值越小，QPS-Precision 曲线越靠近左上角；这表明，更小的 K 值对应更快的搜索速度和更低的搜索精度。Fig.4 中内存占用与 K 值几乎呈正比例线性关系。

从应用层面上看，更小的 K 值适合于对精度要求不高，但对内存占用和搜索速度有要求的应用场景（比如一些移动设备）。

由此推测，稀疏化图结构是一个不错的内存+速度与精度 trade off 的方法。以 70NNGraph 为例，与 100NNGraph 相比，精度从0.996下降到0.992，内存占用从$884.8MB$降低到$770.0MB$，QPS 从483.01提升到578.57。精度损失在一定程度上可以接受，但对内存有将近13%的优化，速度也有较大提升。另一方面，如果希望适当地提高精度，还可以通过增大L来实现（不过对搜索速度有损害）。

② 太过稀疏的图结构丢失重要信息

Fig.3 中，当图结构过于稀疏，如 10NN，增加L获得的精度增长随着L增大逐渐变缓，并且精度的上限降低。这说明太过稀疏的图结构将丢失查询所需的重要信息。

综上，使用 KNN 最近邻搜索算法，如果想要在精度、速度、内存占用上都取得较好的效果，需要根据实际应用场景适当地调整图结构节点出度 K 与L值

(5) 对于改进图结构的启发:

- 稀疏化图结构能: a)降低内存占用，b)提高搜索速度

- KNN-Graph 中可能存在一些冗余信息，对精度提升贡献不大，但是额外占用了内存和搜索代价。从 100NN 到 60NN 之间的 QPS-Precision 曲线很接近可看出，图结构并非越稠密越好。

因此，优化图结构可以对 KNN-Graph 剪枝，或者基于 KNN-Graph 构建一个 MSNET，使得构建好的图结构只保存**获得最优精度所需的必要信息**。

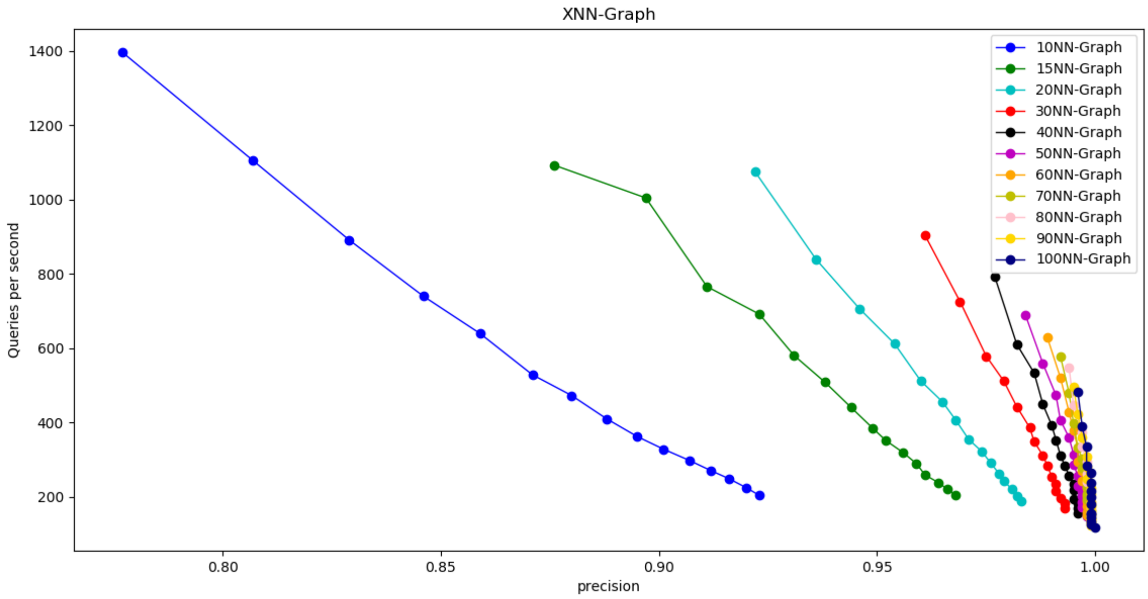

Figure 3. **不同** K **值（出度）的** KNN-Graph **搜索性能。**

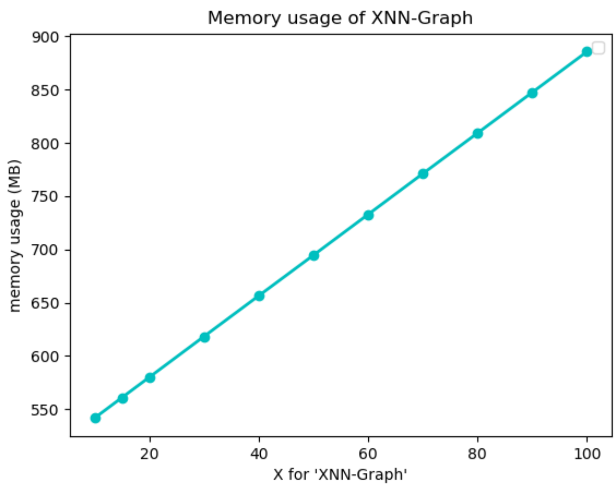

Figure 4. **不同** K **值的** KNN-Graph **对应的内存占用大小，**K **与内存占用几乎呈线性关系**

二、进阶题

1. 调参过程

OPQ 算法需要调整的参数有：每个子空间中 codewords 的数量k、算法迭代次数n_{iter}、code length n_{bits}

(1) 每个子空间(subspace)的codewords的数量k（假设每个子空间codewords的数量相等）。

理论估计

为节省内存及方便存取，k一般取2^8，2^{16}（对应 uint8, uint16）。

不取2^{32}的原因有 ：

a) 此时 codewords 数量过多，会造成建立索引的时间（indexing time）过大。

b) $2^{32}=4294967296$个聚类中心，适用于数据量在千亿级或者更高的数据集，（至少对于 GIST-1B 来说）不太实用。

c) lookup table 的内存消耗急剧上升，对 SDC 方法不够友好：以存储一个 float32 子空间为例，理论估计内存

当$k=\ 2^8=256$时为

$$2^8\times2^8\times4=2^{18}byte=\ 256KB \quad (2)$$

当$k=2^{16}=65536$时为

$$2^{16}\times2^{16}\times4=2^{34}\ byte=16GB \quad (3)$$

当$k=2^{32}=4294967296$时为

$$2^{32}\times2^{32}\times4=2^{66}=64EB \quad (4)$$

考虑到 lookup-table 的对称性，实际可以只载入一半数据。但是，以$n_{bits}=256$为例，需要 32 个子空间，因此估计总的内存占用：当$k=2^8$，2^{16}，2^{32}时，分别为$4\ MB$，$256\ GB$，$1\ ZB$，远远超出内存负荷。

实际上[2]对k取值的建议是 “In practice, k is often kept as the largest affordable number”。因此根据实验要求和实际条件(数据集和硬件条件)，取$k=2^8=256$

(2) k-means 算法迭代次数 n_{iter}

关于n_{iter}的参数选择，[2]对此的讨论较充分，并且数据集及其他实验条件与本报告所用基本一致，所以选择借鉴其实验结果。

实验设置:

- $k=256$
- $n_{bits}=32$
- OPQ 的 Matlab 源代码实现(non-parameters OPQ) [3]
- Dataset: SIFT-1M base-data

实验结果

当n_{iter}比 100 小，distortion 随迭代次数增加较缓慢，超过 100 后 distortion 趋向平缓[2] (Fig. 5)。考虑增大迭代次数对 indexing time 的影响，再增加n_{iter}显得不划算。

因此选择按照[2]的做法，**将n_{iter}取为 100**。

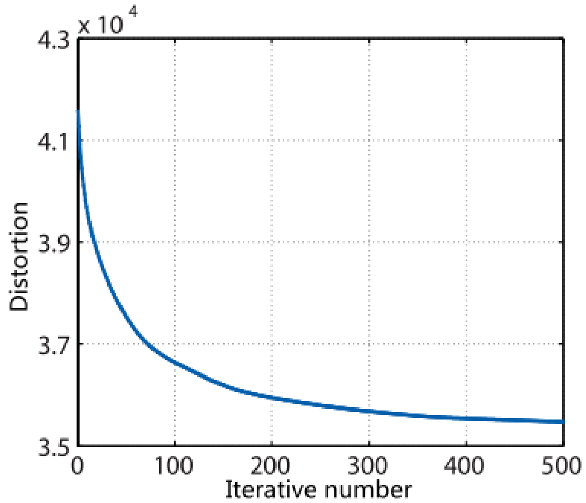

Figure 5. distortion 随迭代次数的变化($k = 256$,$M = 4$)

(3) 原始空间每个数据点的编码长度n_{bits}（二进制长度为单位）。

n_{bits}与内存预算(memory budget)直接相关。当n_{iter}和k确定后，子空间的数目M就确定了

$$M = \frac{n_{iter}}{\log k} \qquad (5)$$

（暂不考虑不能整除情况。）

① 精度的考虑

由于 distortion 能代表 KNN 搜索的搜索精度[2]，因此选择测试不同n_{bits}对 distortion 的影响。

实验设置：

- $k = 256$
- $n_{iter} = 10$
- Matlab code[3]
- Dataset: SIFT-1M (Base Data)
- $n_{bits} \in [32, 64, 128, 256, 512, 1024]$

注：实际过程中，由于迭代次数$n_{iter} = 100$时建立索引时间（indexing time）较大（$k = 256$,$n_{iter} = 100$，$n_{bits} = 256$，CPU i5-7200U, Matlab 源代码[3]对 SIFT-1M 的 Base Data 编码大约需要 24.6h）。为方便实验，在**选择 code length 参数取值的过程中取$\boldsymbol{n_{iter}} = \mathbf{10}$**。

实验结果：

Fig.6显示：编码长度（code length）越大，distortion越小，一般来说，对应的AKNN搜索精度(recall, precision)也就越好[2]

② 内存占用的考虑

理论估计

固定以下参数，对不同的 code length 所需的内存做简单的理论估计，估计结果见 Table.2

- $k = 256$
- Dataset: SIFT-1M base-data
- $n_{bits} \in [32, 64, 128, 256, 512, 1024]$

综合 Fig.6 与 Table.2 可得：更大的 code length 对应更高的精度和更大的内存代价。因此我们选择两个适中的值：$\boldsymbol{n_{bits} = 256}$ **和** $\boldsymbol{n_{iter} = 512}$，分别进行测试。

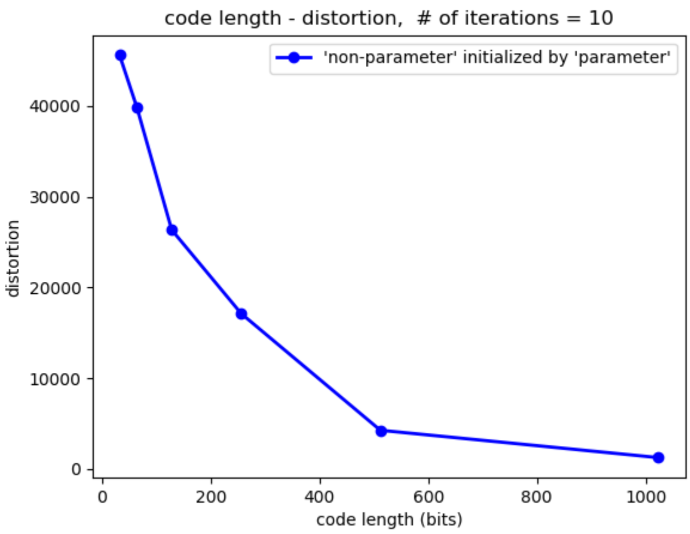

Figure 6. **编码长度**(code length) – distortion **曲线图。**n_{bits}**取离散值**

Table 2. **内存占用的理论估计，**$k = 256$。（SIFT-1M 与 GIST-1B 代表码本(codebook)的理论估计内存占用）

$code\ length$	32	64	128	256	512	1024
$\#\ subspace$	4	8	16	32	64	128
$look\_up\ table$	$1\ MB$	$2\ MB$	$4\ MB$	$8\ MB$	$16\ MB$	$32\ MB$
$SIFT1M$	$4\ MB$	$8\ MB$	$16MB$	$32\ MB$	$64\ MB$	$128\ MB$
$GIST1B$	$4\ GB$	$8\ GB$	$16\ GB$	$32\ GB$	$64\ GB$	$128\ GB$

2. Queries per Seconds – Precision & 峰值内存占用

实验设置

- non-parameter-OPQ initialized by parameter-OPQ
- KNN 搜索算法：Algorithm. 1
- $n_{iter} = 100$，$k = 256$，$n_{bits} = 256\ (512)$
- $L \in [110: 10: 300] \cup [300: 100: 1800]$
- 待搜索最近邻数$K = 100$
- 100NN-Graph

- AVX256 (ADC) +二分插入排序
- Dataset：SIFT-1M
- CPU i5-7200U

ADC 方法：将待查询点(Query Data)的“子段”与查询路径上遇到的数据点的子空间对应的 codewords 求 L2 距离后求和，代替原来直接求解 L2 距离。ADC 方法不需载入 lookup-table。同时由于子空间的数据维度小于 AVX256 寄存器的长度，无法使用 AVX 加速计算。

SDC 方法：对每个待查询点(Query Data)，从 codewords 查找每个子空间最近的聚类代表点，利用代表点的索引对 Query Data 编码。将 Query Data 和 Base Data 的码书分别记作 querybook 和 basebook。计算 L2 距离时，首先从 querybook 和 basebook 找到当前子空间的两个索引，再从对应子空间的 lookup-table 查找二者距离；将所有子空间对应的距离求和，代替原来直接求解 L2 距离。SDC 方法必须载入 lookup-table，可以不载入 codewords (由于占用内存很小所以实际上影响不大)。由于不涉及向量计算所以无法使用 AVX 加速。

二者均使用二分插入排序优化。

注意：
- OPQ 算法涉及到一个正交矩阵R，用于变换坐标轴（不改变数据点之间距离）。由于最终得到的子空间的聚类代表点(codewords)是经过坐标变换后的结果，所以在使用 ADC 方法以及 SDC 编码 lookup-table 之前，需要将所有的待查询点（Query Data）乘以R，即变换坐标轴后，才能计算欧氏距离。
- 要注意索引是从 0 开始还是从 1 开始并及时进行变换。

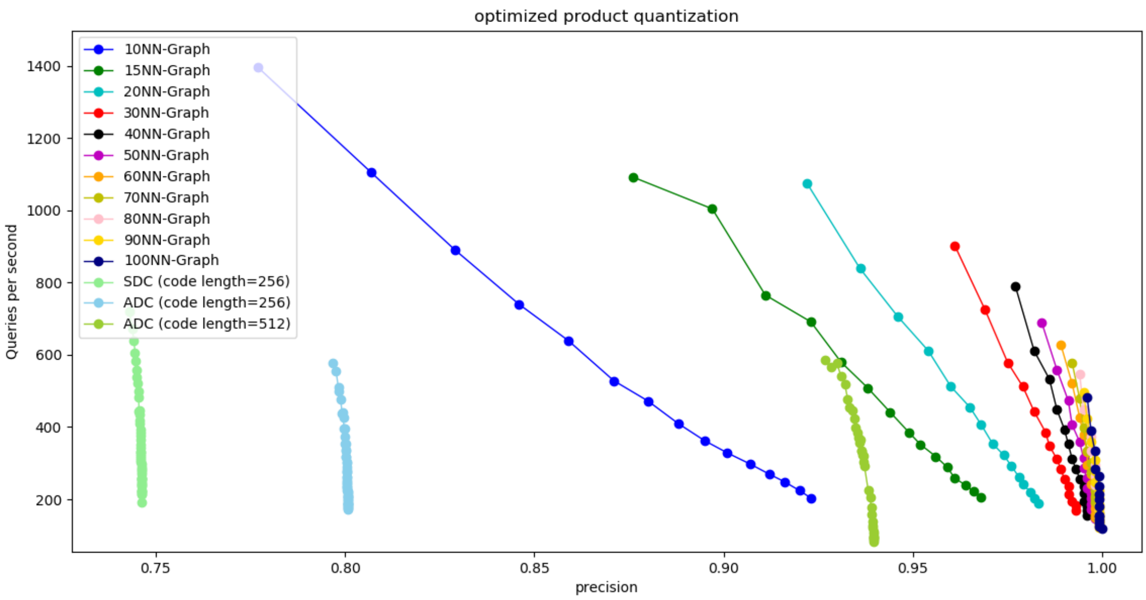

Figure 7. Optimized Product Quantization 与 XNN-Graph 的 OPQ-Precision 曲线对比（Code length=256, 512）。

实验结果与分析：

QPS-Precision

实验结果见 Fig.7

① 一般来说，QPS-Precision 曲线越接近右上角表示算法效果越好。从 Fig.7 看，使用 OPQ 的 SDC 方法和 ADC 方法代替 KNN-Graph 的 L2 距离计算在 SIFT-1M 上表现效果不佳：精度较差，速度也低。

分析：ADC 与 100NN-Graph 相比：

a)在精度方面：base-data 的信息被“模糊化”了，因此用 L2 距离表征数据点之间的关系时，ADC 只能得到模糊的距离信息，精度不如 100NN-Graph 也是合理的。

b)在速度方面：ADC 与 100NN-Graph 计算距离时的向量长度相同，但因为无法使用 AVX 优化，故速度更低。在本实验参数下，以 SIFT-1M，$n_{bits}=256$为例，每个子空间的数据维度为 4（128 维，32 个子空间），小于 AVX256 的寄存器最小容量 8（8 × 32bits），因此无法使用 AVX 进行优化（$n_{bits}=512$时，子空间维度为 2，也无法使用 AVX256），相比有 AVX 优化效果的 100NN 速度肯定会慢。而且，由于代码编写细节（分段多次计算后叠加），甚至会比没有 AVX 优化的简单 L2 距离计算还要慢。事实上，由于 OPQ 将高维向量分段的特点，每次计算距离的数据维度必然不会很大，子空间维度一般不高，所以不容易利用 AVX 做优化。

② $n_{bits}=256$的 ADC 方法相比 SDC，能够在同样的速度下能提高 0.05 左右的精度。在L相同时，SDC 比 ADC 速度快。

分析：ADC 与 SDC 相比，Query Data 的精细信息没有丢失，因此距离计算上更加准确；同时，由于需要计算高维距离，相比 SDC 直接查表的方法速度要慢也是合理的。

③ 编码后，调节L，**精度提升存在上限**：与简单 L2 距离计算不同，编码后调节L对于精度的提高作用已经不够显著甚至十分微弱。事实上，当$L>400$时，精度变化只在10^{-5}量级，并且还是上下波动，基本上可以认为是噪声，因此可认为精度已达到上限。

分析：OPQ 表现不好的问题和基础题中 GIST-1M 性能差的问题类似。两者相同之处在于性能差的原因都是 **L2 距离对数据点之间的关系表示能力不足**。只不过 GIST-1M 是由于“维度灾难”，而 OPQ 方法是由于量子化坐标的过程中损失了精细的距离信息。

量子化带来的信息损失只能通过“更加精细地划分量子化基本单位”来补偿（如提高 code length，提高子空间 codewords 数量k等）。而解决 GIST-1M（高维数据）性能差问题的关键是提升高维数据的距离表示能力（维度灾难[4]）。

峰值内存占用(SIFT-1M)

不同设置情况下的峰值内存占用分别为：

$n_{bits}=256$，SDC：431.1 MB

$n_{bits}=256$，ADC：435.8 MB

$n_{bits}=512$，ADC：457.0 MB

简单距离计算（100NN-Graph）: 885.5MB

① OPQ 优化后，内存占用主要来自于 100NN-Graph。根据基础题 2 (2) 的简单实验的结果估计，在 OPQ 方法中，100NN-Graph 占据了总内存消耗的83.7%~88.8%

② 与 100NN-Graph 相比，OPQ 方法降低了50%左右的内存占用，但对类似 SIFT-1M 这样的数据集来说，当

图结构稀疏化后，OPQ 方法的优势就没有这么明显了。这主要是因为 SIFT-1M 的 Base Data 的内存占用量和 100NN-Graph 相当。以 15NNGraph 为例，15NNGraph 需要内存 560.9MB，比 OPQ 方法中性能最好的$n_{bits}=512$的 ADC 方法增加了 103.9MB；但是 15NN-Graph 的曲线在$n_{bits}=512$的 ADC 方法的曲线右上方（$n_{bits}=512$的 ADC 方法最接近 15NN-Graph 的点的精度为 0.921, 速度为 577.39 QPS; 而 15NN-Graph 的精度为$0.931>0.921$，速度为$579.89>577.39$ QPS）

性能对比的启发

从实验结果看，有趣的是，当 K 比较小时，KNN-Graph 的内存消耗比 OPQ 高一点，但相比之下，速度和准确率性能更好，作为补偿是值得的。这说明针对 SIFT-1M 这类数据量较小的数据集，优化图结构比 Product Quantization 方法更容易得到性能优越的解。但要注意，当图结构的内存占用与 Base Data 相比不值一提时（如 GIST-1M 或 GIST-1B），可以预计，OPQ 方法对内存仍会有较大的优化，而稀疏化 KNN-Graph 就无能为力了。

Table 3. SIFT-1M Base Data **使用**[3]**在不同** code length **时的** indexing time

n_{bits}	128	256	512
$indexing\ time$	$20.6h$	$24.6h$	$27.9h$

思考：如何尽量减少 OPQ 的精度损失？在 SIFT-1M 上能否增大到与稀疏化的 KNN-Graph 相抗衡的水平？

a)增大k值，最好是比 256 大一些但不至于太大（如 512，避免陡增的 indexing time），同时还要适合读取与存储。

如果$k=256$，每个子空间的索引就可以用 uint8 存储，如果增大k值，可能需要使用 uint16 存储，因此 Table.2 对 codebook 的内存占用的估计会提高为原来的两倍，即数据集规模越大，额外内存成本增加越多。

因此，如果给出足够的内存，可以考虑增大k的同时，将 codebook 使用 uint16 类型存储。这样会浪费一些存储空间，但也许得到的精度提升是值得的（评判标准是速度和精度至少能够达到与 15NNGraph 等相匹敌甚至更好的效果，这样才能体现出内存优化的价值；或者使用复杂的编程技巧避免这些内存损耗。）这样，由(5)计算的子空间数量为：

$$M=\frac{n_{bits}}{\log 65526}=\frac{n_{bits}}{16} \qquad (6)$$

b)增大n_{bits}（效果可能不好）。Fig.6 显示，从 512 以后，继续增大n_{bits}，distortion 降低不够显著，精度提高空间可能不大。对 SIFT-1M 来说，最多有 128 个子空间，对应$n_{bits}=1024$，从 Table.3 来看，indexing time 将至少大于$28\ h$。

3. (选做题) C++ implementation of OPQ

代码见 Github

(1) 编程过程简记
- 使用 Eigen 存储计算中间结果时，容易涉及高维稠密矩阵，因此在设计函数时要尽量避免值传递，以免引起爆栈。
- 使用 Eigen 库进行矩阵运算，在 Eigen Decomposition 和稠密矩阵乘法计算上速度太慢，于是链接 Intel MKL 进

行加速。但是矩阵运算的速度还是不如专门优化过的 Matlab.

- Eigen 库的安装与使用参考[5, 6, 7]
- Intel MKL 库的安装与使用参考[8, 9]

(2) 代码测试

取 SIFT-1M 前 10k 个数据点，在自己的代码上测试不同 code length 对 distortion 的影响。distortion-code length 曲线见 Fig.8，与[3]给出的结果(Fig.6)对比，两者趋势保持一致，说明代码的正确性。

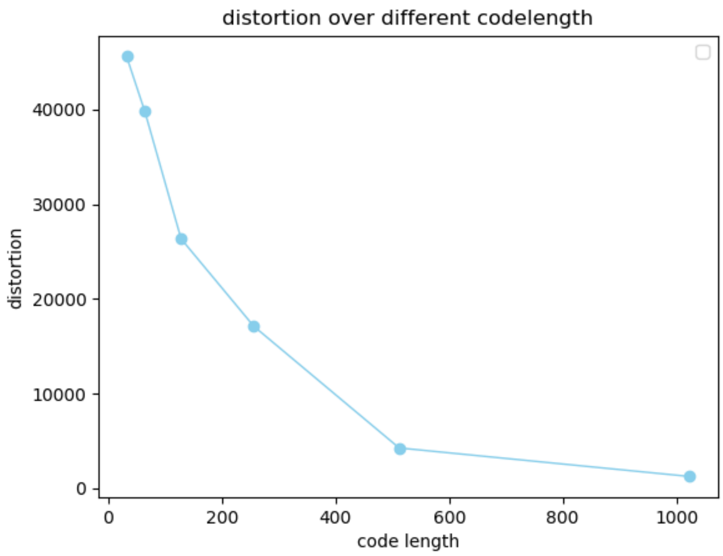

Figure 8. **不同编码长度**(code length)**下的** distortion**，与** Fig.6 **基本保持一致**

三、开放题

1. 设计图结构

[1]是图搜索 state-of-art 的方法，给出了四个优化方向：

1)保持连通性（ensuring the connectivity of the graph）
2)减小数据点的平均出度（lowering the average out-degree of the graph）
3)缩短搜索路径长度（shortening the search path）
4)减小索引规模（reducing index size）

基于[1]的一些想法：

- MRNG[1]借鉴 RNG 的思路，改进了 RNG 的边选择策略，构建一个极度稀疏的 MSNET。MENG 的边选择策略足够好吗？有没有更好的边选择策略？使得 Graph 在保持搜索精度的同时尽量稀疏。
- 为了减少索引构建时间，并没有基于全部数据点、而是在 KNN-Graph 上实施边选择策略来构建图结构。因此当需要缩短索引构建时间可以借鉴这个思路：基于 KNN-Graph 而不是全局
- 保持连通性的方法是固定 starting point，保证 starting point 到余下所有点都至少有一条路径：从 starting point 出发进行深度搜索，把没有连接的点连接上。

参考文献

[1] Cong Fu, Chao Xiang, Changxu Wang, Deng Cai. Fast Approximate Nearest Neighbor Search With The Navigating Spreading- out Graph. PVLDB, 12(5): 461-474, 2019.

[2] Ge, Tiezheng et al. “Optimized Product Quantization.” IEEE Transactions on Pattern Analysis and Machine Intelligence 36 (2014): 744-755.

[3] Source Matlab code of Optimized Product Quantization
http://kaiminghe.com/cvpr13/matlab_OPQ_release_v1.1.rar

[4] 机器学习中的维度灾难 - 红色石头的文章 - 知乎
https://zhuanlan.zhihu.com/p/26945814

[5] Eigen 教程
https://www.cnblogs.com/houkai/category/716820.html

[6] Eigen official documentation
http://eigen.tuxfamily.org/index.php?title=Main_Page

[7] Eigen 库在 VS2017 下的配置与使用
https://blog.csdn.net/Kerwines/article/details/82807596

[8] Intel MKL 在 VS 中的配置与安装笔记
https://blog.csdn.net/caoenze/article/details/46699327

[9] vs2015+eigen+intel MKL
https://blog.csdn.net/pukitoto/article/details/70838039

[10] AVX 指令（Intrinsic）使用介绍
https://www.jianshu.com/p/2244f21422c4

[11] AVX official documentation
https://software.intel.com/sites/landingpage/IntrinsicsGuide/#cats=Arithmetic