Smart LSM-tree with parallel 报告

陈梓萌 523031910121

2025 年 5 月 20 日

1 背景介绍

在当今数据爆炸的时代，从多媒体（图像、视频、音频）、自然语言处理（文本、语音）到推荐系统和生物信息学，越来越多的应用依赖于对高维向量数据的高效处理和检索。其中，K-Nearest Neighbors (KNN) 搜索是这些领域的核心操作，它旨在在一个庞大的向量数据集中，为给定的查询向量找到其最相似的 K 个数据点。KNN 搜索的效率直接决定了这些应用的响应速度和用户体验。

传统的精确 KNN 搜索，尤其是针对高维数据，面临着严峻的挑战。随着数据维度和数据集规模的增长，计算每个查询向量与数据集中所有向量之间的相似度变得极其耗时，这就是所谓的“维度灾难”（Curse of Dimensionality）效应。即使在数据量不那么庞大的情况下，如果核心相似度计算（如余弦相似度、欧氏距离）或数据检索本身涉及到大量计算或 I/O 操作，其串行执行的效率也难以满足实时性需求。

尽管存在近似最近邻（Approximate Nearest Neighbor, ANN）等方法来牺牲一定精度换取速度，但在许多对精度要求较高的场景下，精确 KNN 搜索依然不可或缺。此时，如何在保证精度的前提下，提升精确 KNN 搜索的计算效率，成为一个亟待解决的问题。

在 `KVStore` 实现中，`search_knn` 方法是这一挑战的集中体现。该方法需要遍历存储在 `Cache` 中的大量向量数据，对每个向量计算与查询向量的相似度，并维护一个动态的 Top-K 列表。特别是，当需要获取最终 Top-K 结果对应的详细信息（如通过 `get(key)` 方法获取字符串值）时，如果这个获取操作也是耗时操作，那么整体延迟将进一步增加。

本研究的核心关注点正是 `search_knn` 过程中存在的可并行化计算瓶颈。我们识别出，无论是在遍历大规模向量数据集并进行相似度计算的阶段，还是在最终确定 Top-K 结果后并行检索相应元数据的阶段，都存在显著的并行潜力。通过引入 ** 多线程并发 ** 技术，我们可以将数据集的相似度计算任务拆分到多个处理器核心上并行执行，并能同时加速 Top-K 结果的元数据获取。

通过对 `search_knn` 阶段的并行化优化，我旨在：

1. 显著降低大规模向量相似度搜索的延迟，使其更适应实时应用的需求。

2. 有效利用现代多核 CPU 资源，提高系统吞吐量。

3. 在保证精确搜索结果的前提下，为高维向量数据的检索提供一个高效且实用的解决方案。

2 测试

2.1 实验设置

2.1.1 实验平台（编译环境）

- C 编译器：GNU GCC 13.3.0
- C++ 编译器：GNU G++ 14.2.0
- CUDA 编译器：NVIDIA NVCC 12.6

2.1.2 开发环境

- 操作系统：Windows 10（版本 10.0.26100）
- 子系统：WSL2（Ubuntu 24.04 LTS）
- CUDA 工具包：CUDA Toolkit 12.6（支持 GPU 加速开发）
- IDE：Visual Studio Code（通过 Remote - WSL 插件远程开发）
- CPU: 24 核心，32 线程

2.1.3 测试量

对总数据量 $n = 49998$ 在线程数量 $num\_threads = 1, 2, 4, 8, 16, 24, 28, 32, 36$ 下依次进行测试。每次都进行 128 次查找，取平均值。其中每一句文本都提前计算好其对应的 embedding vector。

2.2 预期结果

2.2.1 优化改进点

1. 原始方法是暴力搜索，即所有的相似度计算和 `get` 操作都在循环中串行执行，对于大量的 `Cache` 会导致效率大幅下降。并行优化后的搜索方式，将 `Cache` 分割成 `num_threads` 个逻辑上的块，分解成独立的、更小的子任务。

2. 使用 `std::async` 并发地启动多个 `find_top_k_in_chunk` 任务。每个任务都在 `Cache` 的一个专用块上运行。`find_top_k_in_chunk` 使用自己的最小优先队列，在其分配的块内找到本地的 Top `k_per_chunk` 相似度。

3. 所有并行 `find_top_k_in_chunk` 任务完成后,主线程通过 `fut.get()` 收集它们的 `chunk_results`。然后，它将这些局部的 Top-K 列表合并到一个全局的 `top_k_global` 最小优先队列中，以找到最终的 k 个最相似项。

4. 由于只查找到最终 Top-k 相似项的键值，需要启动另一组并行任务，进行 `get(key)` 操作。

我的优化方法带来的主要优势：

1. **性能提升 (并行化)**：通过将相似度计算分布到多个线程，可以显著减少总执行时间，特别是对于大型数据集。

2. **减少 I/O/昂贵操作**：最具影响力的优化是延迟和并行化 `get(key)` 调用。这避免了对那些最终不会进入 Top-k 的项进行不必要的查找，如果 `get` 操作很昂贵，这将带来显著的性能提升。

3. **可扩展性**：这种并行方法对于更大的数据集来说更具可扩展性，而不是纯粹的顺序扫描。

2.3 实验结果与分析

表 1: Thread Performance Metrics

Threads	Total Time (ms)	Time Per Iteration (ms)
1	9772	76.34375
2	5018	39.203125
4	2642	20.640625
8	1831	14.3046875
16	1213	9.4765625
24	1116	8.71875
28	1016	7.9375
32	1161	9.0703125
36	1208	9.4375

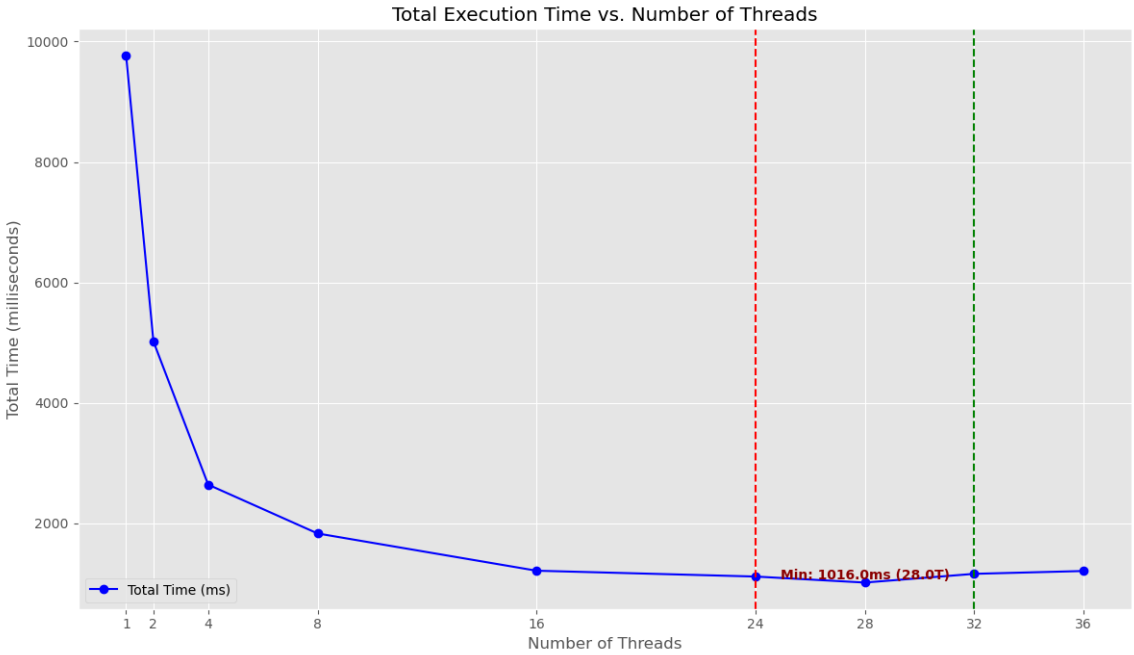

图 1: $\text{total}_time_v s_t hreads$

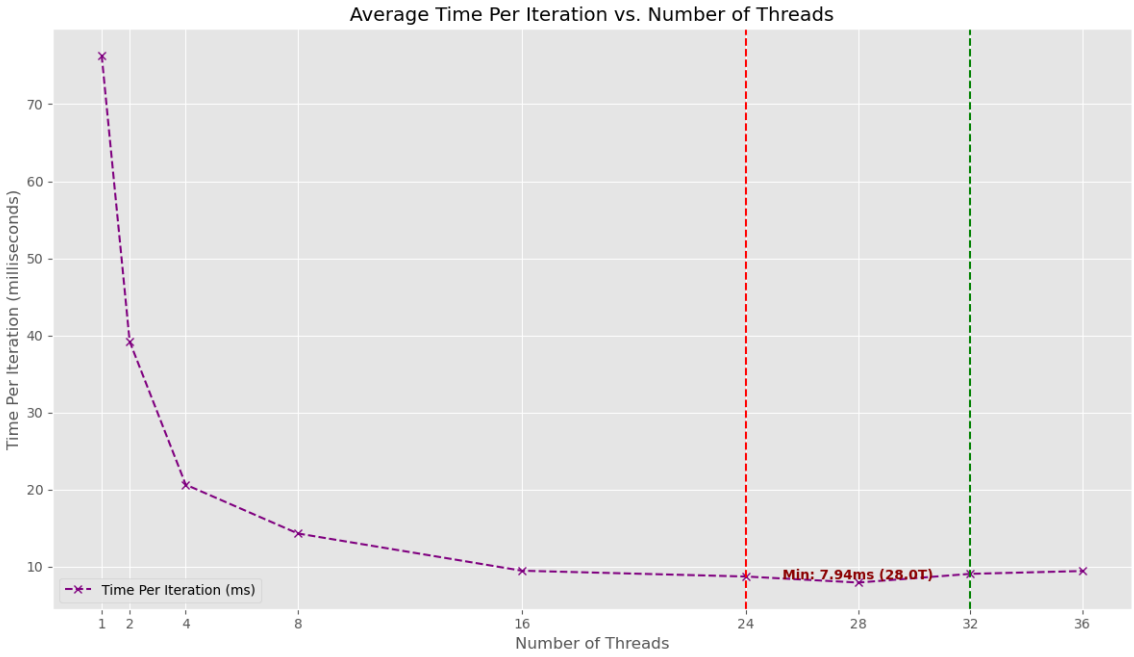

图 2: time$_p er_i teration_v s_t hreads$

图 1给出了 128 次操作下不同线程数量的搜索操作的总耗时，图 2给出了不同线程数量的搜索操作的单次耗时。由图 2可以看出，随着线程数增加，单次操作耗时呈降低趋势。

1. 从单线程转变成 16 线程的过程中，单次操作耗时显著减少。

   原因：将数据处理分成了多个独立的“块”（chunk），并通过 std::async 将这些块的相似度计算（Map 阶段）和最终结果的 get 操作（Reduce 阶段的一部分）并行执行。随着线程数的增加，有更多的 CPU 核心可以同时执行这些独立的计算任务。每个线程负责处理一部分数据，从而减少了完成整个任务所需的总时间。这期间，CPU 核心基本能得到充分利用，任务之间的并行度随着线程数的增加而提高。

2. 在 28 线程达到最小值，再增加线程数耗时小幅波动，没有明显的变化。

   原因：在 24 到 32 线程这个区间，CPU 核心已经接近或达到饱和状态。增加更多的线程（例如到 32 或 36）并不会带来显著的性能提升，因为没有足够的额外物理核心来独立运行这些新线程。新创建的线程需要与现有线程竞争 CPU 资源，这会引入更多的上下文切换开销，导致性能不再提升，甚至可能因为调度开销而略有下降。

3 结论

1. 初期加速区：线程数从 1 增加到接近物理核心数时，性能提升最显著，因为 CPU 核心得到有效利用。
2. 饱和区/瓶颈区：当线程数达到或略超过物理核心数时，性能提升趋于平缓，甚至可能因为过度线程化（thrashing）而略有下降。此时，瓶颈可能已经从 CPU 计算转移到了其他方面，例如内存带宽、I/O 等待、或者线程同步/调度开销。

4 致谢

1. 感谢巩皓文同学，共同讨论了 `search_knn` 测试集如何修改。
2. 感谢 这篇关于 `std::async` 的博客。
3. 感谢 这篇关于 `std::future` 的博客。