High-Performance Top-K Nearest Neighbor Search with CUDA Gather-Scatter and Streams

Abstract

In this project, I implement and benchmark a high-dimensional nearest neighbor search engine leveraging CUDA for massive acceleration. I compare CPU-based brute-force computation with stream-based CUDA kernels and a gather-scatter top-K kernel. The application supports query-to-database distance computation, nearest neighbor identification, and retrieval of the top-K closest database vectors. My implementation achieves over 9× speedup using CUDA streams and showcases the power of warp-efficient gather-scatter kernels.

1 Introduction

Nearest neighbor (NN) search is a fundamental operation in a wide range of domains, including recommendation systems, image retrieval, and vector databases. While the brute-force algorithm guarantees exact results, it is computationally expensive at scale. In this project, I investigate multiple CUDA strategies to accelerate NN search, with a particular focus on stream-based parallelism and gather-scatter techniques for top-K retrieval.

2 CPU Baseline

The baseline implementation uses a nested loop on the CPU to compute squared L2 distances between each query vector and all database vectors. With a vector dimension of 960 and a query batch size of 1000 against 49,000 database vectors, the CPU implementation took an average of **63,162.938 ms**, providing a reference point for evaluating GPU acceleration.

3 CUDA Stream-Based Parallelism

I first implemented a CUDA kernel where each thread block handles a single query vector. Threads cooperatively load the query into shared memory and compute distances in a strided pattern across the database. The kernel uses shared memory reduction to identify the best match for each query.

I launched this kernel using `cudaMemcpyAsync` and four CUDA streams to overlap memory transfers and computation. The streamed kernel achieved an average runtime of **6,812.737 ms**, representing a **9.27× speedup** over the CPU baseline.

Performance metrics from `nvprof` indicate excellent kernel performance:

- Achieved Occupancy: 99.27%
- Warp Execution Efficiency: 100%
- Eligible Warps per Cycle: 0.0156

4 Top-K CUDA Gather-Scatter Kernel

To extend functionality beyond top-1 nearest neighbor, I implemented a custom gather-scatter kernel for top-K ($k = 10$) retrieval. Each thread maintains a local top-K list of distances and corresponding indices. These partial results are stored in shared memory, and a single thread merges them to produce the final top-K output for each query.

This kernel illustrates the gather-scatter pattern in CUDA: threads gather partial results by independently processing subsets of the database and scatter their results into a cooperative structure to finalize top-K selection.

Despite higher shared memory usage, warp execution efficiency remained high:

- Achieved Occupancy: 49.4%
- Warp Execution Efficiency: 99.86%
- Eligible Warps per Cycle: 0.0305

5 Memory and Access Patterns

For each batch:

- Database size: **183.11 MB**
- Query batch size: **3.66 MB**
- Result buffer: **4 KB**
- Total GPU usage: **186.8 MB**

Each thread reads one shared query vector and multiple database vectors using a strided memory access pattern. This results in global memory reads of approximately **44.1 million floats (168 MB)** per batch.

6 Conclusion

This project demonstrates that principled use of CUDA streams and gather-scatter patterns can dramatically accelerate nearest neighbor search. My streamed kernel achieves near-peak GPU occupancy and maximum warp efficiency. The top-K kernel, though more complex, generalizes the problem to ranked retrieval and maintains high execution efficiency.

This work lays a foundation for thinking about these components in recommender systems, dense retrieval pipelines, and vector databases. Future extensions include persistent kernels, warp-aggregated reduction using `__shfl_down_sync`, and multi-GPU support.

This work closely aligns with what I do on a daily basis and would be extremely useful for my team.

7 Implementation Limitations and Future Improvements

While the current implementation delivers accurate results and reliable performance for batchwise nearest neighbor benchmarking, several optimizations and robustness enhancements are possible. These were consciously deferred to prioritize simplicity and maintainability.

- **Kernel Boundary Checks:** The CUDA kernels assume that block and thread indices are always within bounds (e.g., `qid < nq`). This holds under current controlled batch configurations but should be safeguarded if that was used to serve reach requests.
- **Shared Memory Usage:** Query vectors are loaded into statically-sized shared memory arrays (e.g., `s_query[DIM]`). While this is efficient under the constraint that `DIM = 960`, it risks exceeding shared memory limits on devices with smaller SM capacity. Which is why I decided to go with this implementation and not HNSW
- **Stream Management:** Streams are created and destroyed within the batch loop, limiting kernel concurrency and increasing overhead. A more efficient approach would retain and reuse streams across all batches to maximize overlap and reduce stream creation latency. However I wanted to capture the overhead of each operation. Overall this is neglegable/
- **Partial Result Copying:** The top-k kernel displays only the results for the first 10 queries, yet allocates and computes results for the full batch. Incomplete host-side copying may lead to stale data in other contexts and should be replaced with a safer extraction pattern or full-buffer host transfer when needed.

These limitations do not impact the correctness or fairness of the benchmark results under current assumptions (e.g., fixed dimensionality, uniform batch sizes, and sufficient GPU memory). However, addressing them in future iterations would enable the system to scale to larger and more heterogeneous workloads while enhancing fault tolerance and resource efficiency.

Notes

- Due to GPU memory constraints, the current implementation reliably supports computing only the top-10 nearest neighbors per query. Scaling to higher values of k or larger datasets would require either multi-GPU support or significant kernel optimization.
- Attempts to implement HNSW indexing on the GPU proved prohibitive within current memory and resource limits. The dynamic memory access patterns and graph-based structure of HNSW are ill-suited to static GPU memory layouts without advanced memory management techniques or sparse data structures.
- Changing the threads per block did not really have any impact. Again the data set is large enough to where such optimizations will not make an impact yet.
- The stream kernel is near-optimal in occupancy and efficiency, ideal for high-throughput NN distance computations.
- The top-K kernel, while showing lower occupancy, still executes with near-perfect warp efficiency, which validates the gather-scatter design. The trade off here is complexity.
- Because of the nature of the problem I use the map technique in combination with the other advanced techniques mentioned.
- The dataset is in the README. Please download and place in the data directory. A sample output is in the out director.
- Please be careful. This workload is memory intensive and changing the defaults will hinder performance and results.

- The use of scatter-gather kernels and CUDA streams is well-suited to this workload. Streams enable overlapping memory transfers with computation, improving throughput under batching. Scatter-gather simplifies memory access patterns and permits fine-grained control over parallel reductions, which is critical in top-k selection and efficient metric-space computations across high-dimensional vectors.
- One day I'll implement HNSW for GPUs. Thanks for a great class