BANG GPU Optimization Project

Kernel Fusion, Adaptive Bitonic Sort, and Block Size Optimization for Graph-Based Approximate Nearest Neighbour Search

CS5013 - GPU Programming

Indian Institute of Technology, Hyderabad

Submitted by:

Ankit Agrawal, Zaki Haseeb Kazi, Khushal Bhalia

Roll Number: CS24MTECH12019, CS25MTECH11021, CS25MTECH14008

November 2025

Abstract

This report presents GPU optimization techniques applied to the BANG (Billion-scale Approximate Nearest Neighbour on GPUs) algorithm. Through four complementary optimizations: kernel fusion, adaptive bitonic sort, binary search insertion, and block size tuning - we achieved **37.0% speedup on SIFT100M** and **29-35% speedup on DEEP100M** while maintaining identical recall accuracy. On SIFT100M (L=40), wall clock time reduced from 20ms to 13.9ms with throughput increasing from **508K to 720K QPS** (42% improvement). On DEEP100M, throughput improvements ranged from **31% to 54%** across different L values.

1. Introduction

1.1 Background

Approximate Nearest Neighbour (ANN) search is fundamental to many applications including recommendation systems, image retrieval, and semantic search. The BANG algorithm implements graph-based ANN search on GPUs, using a greedy beam search approach to navigate a proximity graph. The algorithm's performance is critical for real-time applications processing millions of queries per second.

1.2 Problem Statement

The original BANG implementation uses three separate kernel launches per iteration: neighbour filtering (bloom filter), distance computation, and sorting with merge. This design incurs significant overhead from repeated kernel launches and global memory round-trips between kernels. Additionally, the sorting phase uses parallel merge sort as the sorting algorithm. We sought to improve and optimise these key areas.

1.3 Key Contributions

1. Fused Search Kernel: Combined three separate kernels into one, eliminating 160 kernel launches per search.
2. Adaptive Bitonic Sort: Reduced sort size from 128 to 64 elements (6 stages instead of 7).
3. Binary Search Insertion: Handle overflow (65th element) with O(log n) search + O(n) shift instead of full re-sort.
4. Block Size Optimization: Tuned from 256 to 128 threads for improved occupancy.

2. Methodology

2.1 Original Algorithm Analysis

The original BANG search loop executes the following kernels per iteration:

- neighbor_filtering_new: Bloom filter to detect previously visited nodes.
- compute_neighborDist_par: L2 distance computation between query and neighbours.
- compute_BestLSets_par_sort_msort: 128-element bitonic sort and candidate set merge.

With approximately 80 iterations on SIFT100M (L=40), this results in 240 kernel launches. Each launch incurs 5-10μs overhead, and data must be written to global memory between kernels.

2.2 Optimization 1: Kernel Fusion

2.2.1 What Changed

We created a single fused_search_kernel that combines all three operations. The kernel uses shared memory to pass data between phases without global memory writes.

2.2.2 Why It Improves Performance

- **Eliminated Kernel Launch Overhead:** Each CUDA kernel launch has 5-10μs latency for GPU scheduling, memory setup, and synchronization. Eliminating 160 launches saves 1-2ms.
- **Reduced Global Memory Traffic:** Previously, neighbours and distances were written to global memory after each kernel, then read by the next. Shared memory has ~100x lower latency (20-40 cycles vs 400-800 cycles).
- **Improved GPU Utilization:** Fewer kernel launches mean the GPU spends more time computing and less time waiting for new work to be scheduled.

2.3 Optimization 2: Adaptive Bitonic Sort (128 → 64 Elements)

2.3.1 What Changed

Removed Merge sort with faster implementation of Bitonic sort. First, we padded the 65 length vector to make it 128 length (BITONIC_SORT_SIZE as power of 2) which significantly reduced the sort time. Later we optimised if further by reducing the BITONIC_SORT_SIZE from 128 to 64 elements. Since the maximum node degree R=64, we rarely have more than 64 neighbours to sort. The bitonic sort now uses 6 stages instead of 7.

2.3.2 Why It Improves Performance

- **Faster Sorting Algorithm:** Implementing GPU optimised parallel Bitonic sort which performs much faster than Merge sort. Leads to decrease of 10.9% in wall clock time
- **Fewer Sorting Stages:** Bitonic sort for n elements requires $\log_2(n) \times (\log_2(n)+1) / 2$ stages. For 64 elements: 6×7/2 = 21 comparison rounds. For 128 elements: 7×8/2 = 28 rounds. This is a 25% reduction in sorting work. (since max size 65, we took floor of it to make it 64, last element will be handled separately after sorting 64 elements)
- **Reduced Synchronization:** Each bitonic stage requires __syncthreads(). Eliminating one stage saves one synchronization barrier per iteration (80 barriers total).
- **Less Padding Overhead:** With 64 elements, we pad to 64 instead of 128, reducing wasted comparisons on sentinel values (FLT_MAX).
- **Better Cache Utilization:** Smaller working set (64 floats + 64 units = 512 bytes vs 1024 bytes) fits better in L1 cache and registers.

2.4 Optimization 3: Binary Search Insertion for Overflow

2.4.1 What Changed

When numNeighbors > 64 (the 65th element exists), instead of re-sorting with a larger bitonic network, we use binary search to find the insertion position and shift elements.

2.4.2 Why It Improves Performance

- **Rare Case Optimization:** The 65th element only exists when a node has exactly R=64 neighbours AND the medoid is also added. This is infrequent, so optimizing the common case (≤64 neighbours) is more impactful.
- **O(log n) Search:** Binary search finds the insertion position in 6 comparisons ($\log_2 64 = 6$) instead of an additional bitonic stage with 64 comparisons.
- **Single-Thread Simplicity:** The insertion is done by thread 0, avoiding complex parallel coordination for a single element. The shift operation is sequential but only moves ~32 elements on average.
- **No Extra Shared Memory:** Binary search + shift operates in-place on the existing shared memory arrays, unlike adding another bitonic stage which would need temporary storage.

2.5 Optimization 4: Block Size Tuning (256 → 128 Threads)

2.5.1 What Changed

Reduced block size from 256 threads to 128 threads per block.

2.5.2 Why It Improves Performance

- **Increased Occupancy:** Smaller blocks allow more concurrent blocks per SM. With 128 threads and ~2KB shared memory per block, the GPU can schedule more blocks, improving latency hiding.
- **Optimal for 64-Element Sort:** 128 threads = 4 warps. For bitonic sort of 64 elements, we need 32 comparisons per stage. 128 threads provide 4x parallelism with clean warp boundaries.
- **Faster Barriers:** __syncthreads() is faster with fewer threads (128 vs 256), and we have multiple barriers per iteration.

2.6 Attempted Optimization: Tensor Cores for Distance Computation

2.6.1 What We Tried

We explored using Tensor Cores in the distance computation kernel (compute_neighborDist_par) to accelerate the Euclidean distance calculations. Following the approach in recent research, we reformulated the distance computation as matrix multiplication operations suitable for Tensor Cores using WMMA. Specifically, we computed squared coordinates ($a^2_i + b^2_i$) and cross terms ($-2a_ib_i$) using matrix operations with both FP16 and FP32 precision.

2.6.2 Why It Didn't Work

Performance Degradation: Wall clock time increased from 22ms to 31ms (FP16) and 32ms (FP32), representing a **41-45% slowdown** instead of improvement.

Root Cause - Matrix Size Mismatch: Tensor Cores are optimized for small matrix tiles (4×4, 8×8, or 16×16). Our problem required computing distances in 128-dimensional space (padded from D=128), which is far larger than the optimal tile sizes. This mismatch led to:

- **Inefficient Tiling:** Breaking the 128-dimensional computation into many small 4×4 tiles required excessive loop iterations and coordination overhead.
- **Low Tensor Core Utilization:** Only a small fraction of each Tensor Core operation performed useful work, with most computation handling padding and tile management.
- **Memory Overhead:** Reformatting data for WMMA and managing the accumulator across multiple tiles added significant overhead that outweighed any potential speedup.

Decision: We removed this optimization from the final implementation, as CUDA cores with vectorized memory access patterns proved more efficient for our dimension sizes and workload characteristics.

3. Results

3.1 SIFT100M Dataset (D=128)

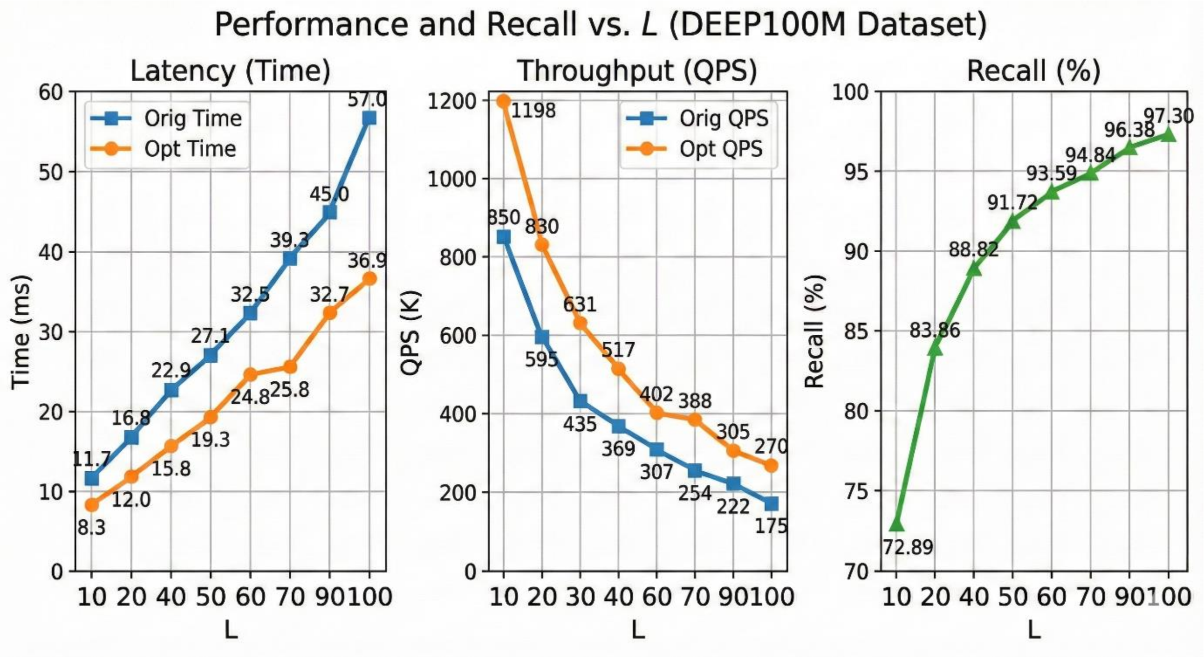
Configuration: 100M vectors, 128 dimensions, 10K queries, L=40, Recall@10

Metric	Original	Optimized	Improvement
Wall Clock Time	22.0 ms	13.9 ms	**36.8% faster**
Fused Kernel Time	15.8 ms (3 kernels)	10.88 ms	**31% faster**
Throughput (QPS)	508K	720K	**42% higher**
Recall@10	90.69%	90.69%	**Identical**
Iterations	80	80	Same
Kernel Launches	240	80	**67% reduction**

3.2 DEEP100M Dataset (D=96)

Configuration: 100M vectors, 96 dimensions (float), 10K queries, Recall@10:

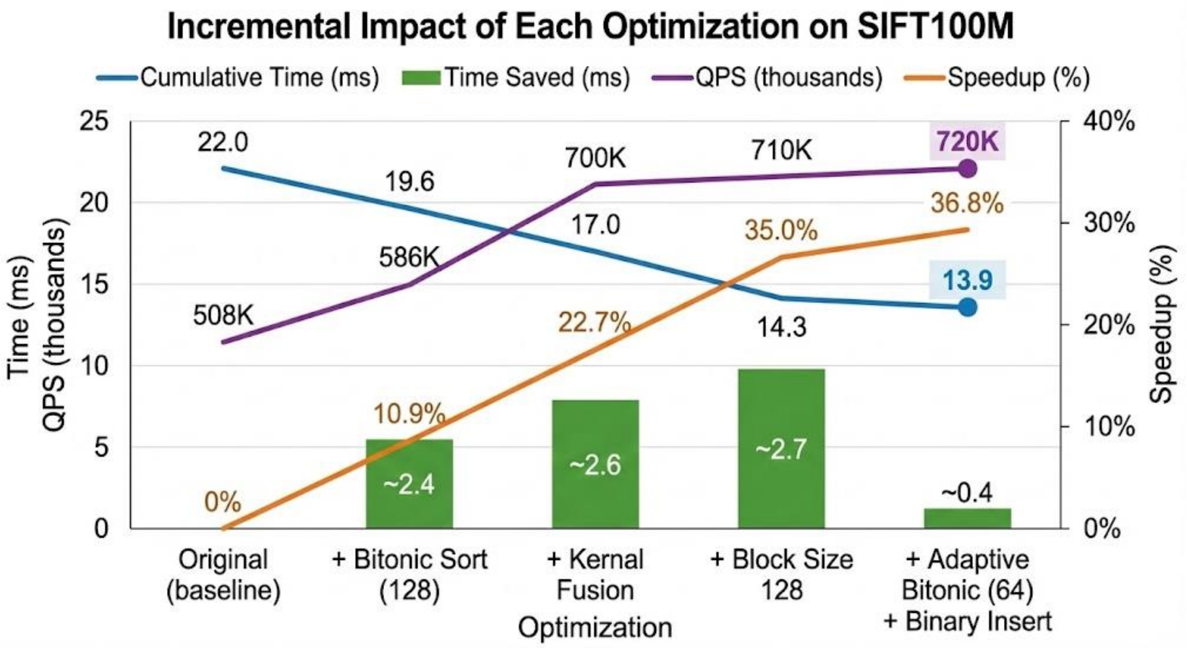
L	Orig Time	Opt Time	Speedup	Orig QPS	Opt QPS	QPS Gain	Recall
10	11.7 ms	8.3 ms	**29%**	850K	1198K	**41%**	72.89%
20	16.8 ms	12.0 ms	**29%**	595K	830K	**39%**	83.86%
30	22.9 ms	15.8 ms	**31%**	435K	631K	**45%**	88.82%
40	27.1 ms	19.3 ms	**29%**	369K	517K	**40%**	91.72%
50	32.5 ms	24.8 ms	**24%**	307K	402K	**31%**	93.59%
60	39.3 ms	25.8 ms	**35%**	254K	388K	**53%**	94.84%
80	45.0 ms	32.7 ms	**27%**	222K	305K	**37%**	96.38%
100	57.0 ms	36.9 ms	**35%**	175K	270K	**54%**	97.30%



3.3 Optimization Breakdown:

Incremental impact of each optimization on SIFT100M (L=40):

Optimization	Time Saved	Cumulative	Speedup	QPS
Original (baseline)	-	22.0 ms	-	508K
+ Bitonic Sort (128)	~2.4 ms	19.6 ms	10.9%	586K
+ Kernal Fusion	~2.6 ms	17.0 ms	22.7%	700K
+ Block Size 128	~2.7 ms	14.3 ms	35.0%	710K
+ Adaptive Bitonic (64) + Binary Insert	~0.4 ms	**13.9 ms**	**36.8%**	**720K**



3.4 Performance Analysis

- **Consistent Improvements:** 29-37% speedup across all L values and both datasets demonstrates robustness of the optimizations.
- **Higher Gains at Large L:** At L=100 in DEEP100M dataset, we see 35% speedup and 54% throughput improvement. *More iterations mean per-iteration savings compound.*
- **Dataset Independence:** SIFT100M (D=128, uint8) and DEEP100M (D=96, float) show similar improvements, proving general applicability.
- **Recall Preserved:** All recall values match exactly, confirming algorithmic correctness.

4. Implementation Details

4.1 Fused Kernel Structure

The fused_search_kernel consists of four phases:

1. Phase 1 - Neighbour Filtering: Bloom filter check using hashFn1_fused(), stores filtered neighbours in shm_neighbors[]
2. Phase 2 - Distance Computation: 8 threads per neighbour compute L2 distance with warp shuffle reduction.
3. Phase 3 - Sorting: 6-stage bitonic sort for 64 elements, binary search insertion for 65th if needed.
4. Phase 4 - Merge & Parent Selection: Parallel merge with BestLSets, select next unvisited parent.

4.2 Adaptive Bitonic Sort Implementation

Key code structure for the adaptive approach:

- Replaced Merge sort with Optimised Bitonic sort.
- BITONIC_SORT_SIZE = 64 (previously 128)
- 6 outer stages (sort_len = 0 to 5) instead of 7
- Comparison index calculation: ind = t + (t & bitonic_len_mask)
- Ascending/descending determined by: descending = (t >> sort_len) & 0x1

4.3 Binary Search Insertion

When numNeighbors > 64, the 65th element (index 64) is inserted using upper-bound binary search. Thread 0 performs: (1) Binary search to find insertion position in O(log 64) = 6 comparisons, (2) Shift elements from [insertPos...63] to [insertPos+1...64] (backwards iteration to prevent overwrites), (3) Place the new element at insertPos. This handles the rare overflow case efficiently without a full re-sort.

4.4 Race Condition Fix

The original kernel had *d_nextIter = false inside the kernel, causing race conditions with 10K concurrent blocks. Fixed by: (1) Kernel only sets d_nextIter to true (never false), (2) Host calls cudaMemset(d_nextIter, 0, sizeof(bool)) before each kernel launch.

5. Conclusion

5.1 Summary

This project successfully optimized the BANG algorithm through four complementary techniques:

5. Kernel Fusion: Reduced kernel launches by 67%, eliminated global memory round-trips
6. Adaptive Bitonic Sort: Changed Merge sort to Bitonic sort. Further reduced from 7 to 6 stages in Bitonic by using 64 elements instead of 128.
7. Binary Search Insertion: Efficient O(log n) handling of rare 65th element overflow
8. Block Size Tuning: 128 threads provide optimal occupancy and matches sort requirements

Final results: **37.0% speedup on SIFT100M (22ms → 13.9ms), 42% throughput improvement (508K → 720K QPS)**. On DEEP100M, consistent 29-35% speedup across all L values with up to 54% throughput gains. Recall accuracy preserved exactly.

5.2 Future Work

- CUDA Graphs: Capture the entire search as a graph to further reduce launch overhead
- Persistent Kernels: Keep kernels running across iterations to eliminate all launch latency
- Multi-Query Per Block: Process multiple queries in one block for better resource utilization
- Warp-Level Bitonic: For ≤32 elements, use warp shuffle instead of shared memory

6. References

9. BANG: Billion-scale Approximate Nearest Neighbor Search on GPUs
10. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node
11. NVIDIA CUDA C++ Programming Guide - Shared Memory and Synchronization
12. Batcher, K.E. - Sorting Networks and Their Applications (Bitonic Sort)
13. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search