Jonas Fritsch

Project – Hybrid Vector Search Queries

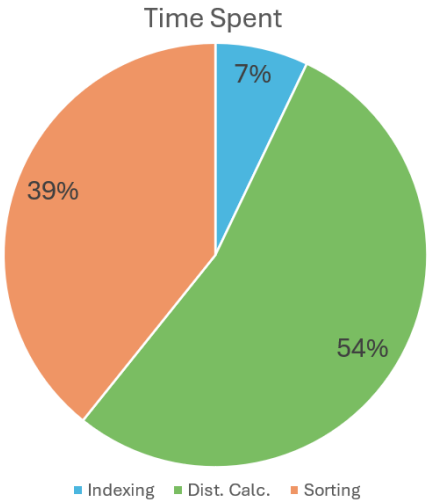
Test System Setup

All benchmarks were run on a system with a 6 core (no hyperthreading, no AVX512) Intel Core i5-8400 CPU with a L1 cache size of 64 KB, L2 cache size of 256 KB and L3 cache size of 9 MB running at max boost clock speed of ~ 3.8 GHz and 16 GB of dual-channel DDR4-2666 RAM.

Initial Baseline Evaluation

Starting with a performance evaluation of the initial baseline implementation gives the following rough runtime distribution:

```
foreach q in queries
    list<vec_t> knn_cand  //candidates
    foreach d in data                                  Indexing
        if matches_query(d, q)
            knn_cand.add(d)

    list<float> knn_dist  //distances
    foreach c in knn_cand
        dist = calc_dist(c, q)  Dist. Calculation
        knn_dist.add(dist)

    sort_cand_based_on_dist(knn_cand, knn_dist)
                                                       Sorting
    result.add(knn_cand.take(100))
```



Optimization #1 – Distance Calculation

The distance calculation calculates a simple squared euclidean distance:

```
fn calc_dist(vec_t d, vec_t q)
    float sum
    for i = 0 to 100
        float diff = d[i] - q[i]
        sum += diff * diff
    return sum
```

Early Bailout Optimization:

Since we potentially calculate millions of distances while only ever being interested in the top-K vectors (K = 100) and the calculation seems to be a performance bottleneck it might be beneficial to stop halfway/three-quarters through the calculation and compare the current partial distance to the worst distance so far in our top-K vector set.

Pseudo-Code

```
fn calc_dist(vec_t d, vec_t q, float worst_dist)
    float sum
    for i = 0 to 50
        float diff = d[i] - q[i]
        sum += diff * diff

    if (sum >= worst_dist)
        return infinity

    for i = 50 to 100
        float diff = d[i] - q[i]
        sum += diff * diff
    return sum
```

AVX2-SIMD Optimization:

We simply optimize the distance calculation using SIMD. The implementation is straightforward except for building the final sum out of the vectorized partial sums. I've compared the performance of different approaches and found the fastest one to be from this StackOverflow answer (see section "__m256 float with AVX").

Floating-Point Inaccuracy

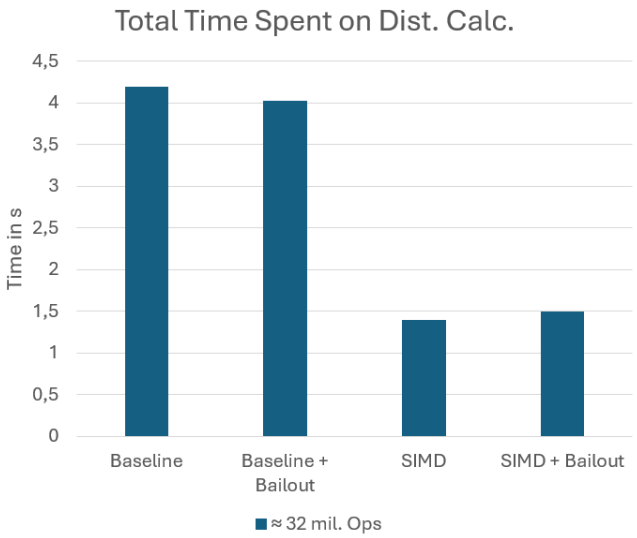
Using the SIMD approach results in different/more-accurate results than the non-vectorized calculation. This is due to the summation of partial sums which helps with loss of precision due to absorption (adding a small floating-point number to a large floating-point number).

An extreme example of this can be seen in the project under *src/fp_inaccuracy_test.cpp* where the difference between the non-vectorized and the AVX2 SIMD distance calculation of two 100 dimensional vectors results in an absolute arithmetic difference of **0.0625 !**

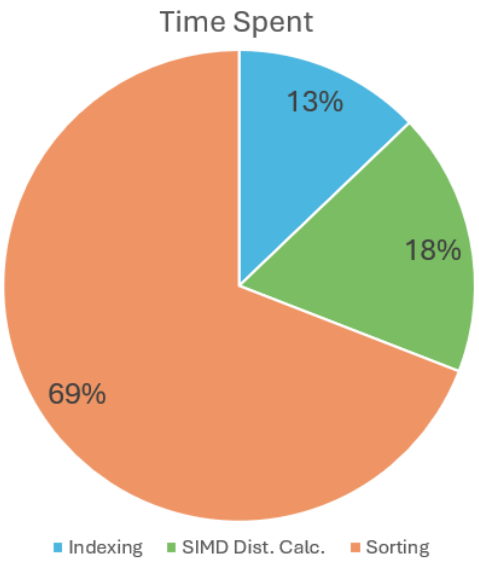
Performance Results:

This chart shows the total aggregated time in seconds for around 32 million different distance calculations:

- SIMD provides ~ 200% performance gain
- While around half of the 32 million distance calculations could bailed out early when checking after three quarters of the sum, the resulting performance gain is minimal at best to even negative for SIMD



After using SIMD for distance calculation, the runtime distribution looks as follows.



Conclusion:

SIMD significantly improves performance. Using 512-bit vector registers with AVX512 would yield even greater performance benefits.
Due to the speed of the SIMD calculation, the overhead of the extra horizontal sum for the early bailout check is worse than the overhead of simply calculating the rest of the distance. However, for non-uniform, higher dimensional vector sets I would eventually expect the bail-out to yield performance benefits even for SIMD. On the other hand, at this point an actual space-partitioning data structure might be a better fit.

Optimization #2 – Using a Data Structure for the KNN

Instead of storing and sorting all the vectors of the dataset when only ever interested in the top-K (K = 100), we at any given point only keep track of the current top-K data vectors.

This results in roughly the following pseudo-code:

```
class Knn
    constructor(q, knn_size)

    // check if vector d belongs in current
    // KNN set and if so add it, only ever
    // storing max 100 (knn_size) vectors
    check_add(d)

    // returns KNNs sorted ascending by dist.
    get_sorted()
```

```
foreach q in queries
    Knn knn_cand(q, 100)  //candidates

    foreach d in data                          Indexing +
        if matches_query(d, q)                 Dist. Calc.
            knn_cand.check_add(d)

    result.add(knn_cand.get_sorted())  Sorting
```

Implementing the Knn data structure can be done in different ways:

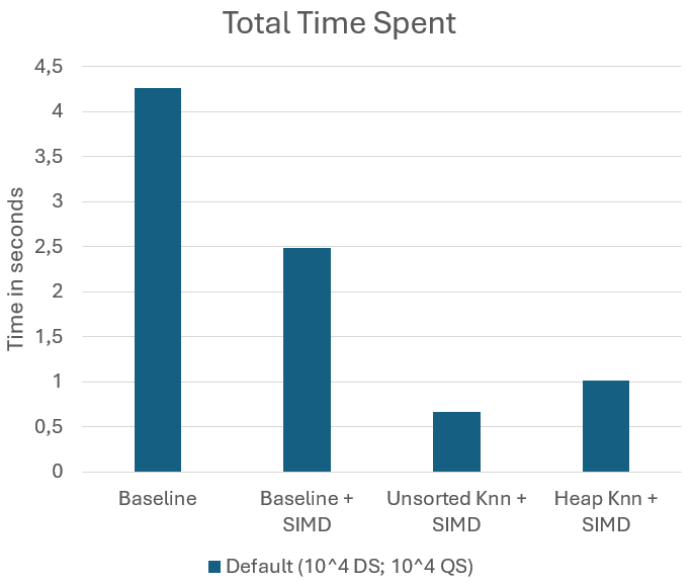
1. **Sorted array**
   + getting the current worst KNN for comparison is *O(1)*
   + returning the sorted result is *O(1)*
   – inserting a new KNN is *O(N)* due to shifting elements!
2. **Unsorted array and cache the current worst KNN**
   + inserting a new KNN is *O(1)* by replacing the current worst KNN

– updating the worst KNN is *O(N)* [1]
– returning the sorted result is *O(N log N)*

3. **Max-Heap over the distance**
   + getting the current worst KNN for comparison is *O(1)*
   + inserting a new KNN is *O(log N)* [2]
   – returning the sorted result is *O(N log N)*

Performance Results:

This chart shows the total execution time in seconds for a benchmark that uses 10^4 random data vectors and 10^4 random query vectors (sample proportion = 1):



- Knn provides ~ 275% performance gain
- Total optimized performance gain over baseline now at ~ 545%
- Unsorted Knn outperforms Heap Knn despite being asymptotically worse due to small K and vectorized linear scan

Conclusion:

I did not implement the sorted array Knn but would be very confident that the resulting performance would not beat the unsorted array Knn and potentially be even worse than the Heap Knn simply due to the element shifting overhead when inserting.

For much larger K I would expect the Heap Knn to eventually outperform the Unsorted Knn. Testing on my setup and with “only” AVX2 SIMD the Heap Knn became faster at around K = 800.

When implementing the unsorted Knn I also played around with a branchless insertion operation which was quite interesting but due to the length of my report and the (surprisingly?) almost non-existent performance benefit I won’t go into details. (See here for implementation.)

[1] While O(N), for reasonable small K, vectorizing this linear scan for the worst element can be done quite efficiently
[2] Can also be O(1) for more complex heap structures, but then the elements would not be stored consecutively in memory like for a binary heap

Optimization #3 – Parallelization

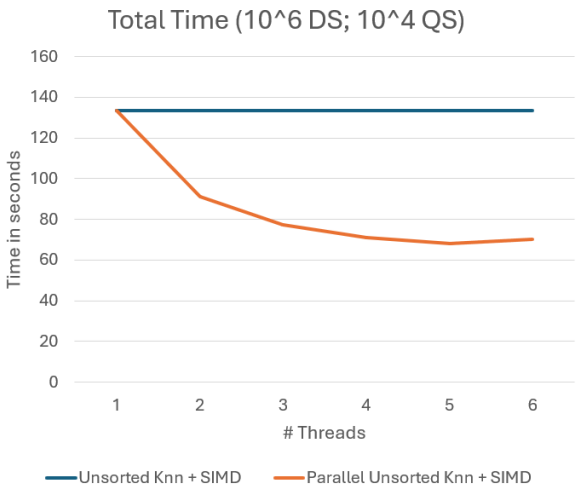
Finally, to be able to run the biggest benchmarks, 10^7 data vectors and 10^4 query vectors (sample proportion = 1) I wanted to parallelize the scanning through the data vectors. To not be bottlenecked by the overhead of creating the threads I decided to implement my own thread pool (see include/threading.hpp).

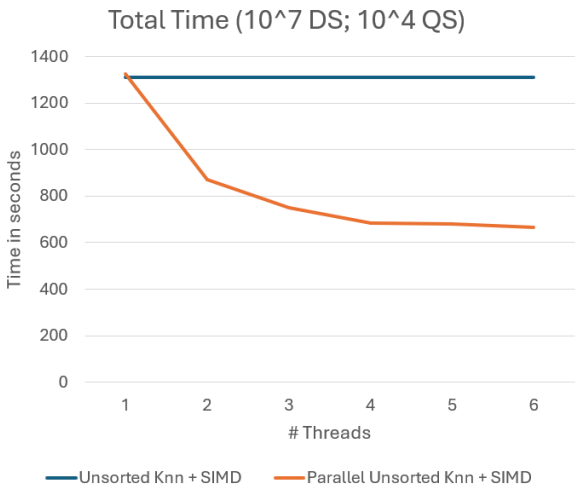
Pseudo-Code:

```
ThreadPool<Knn> tp(num_threads, 100)

foreach q in queries
    tp.set_query(q)

    tp.parallel_for(data, (d) => {
        if matches_query(d, q)
            knn_cand.check_add(d)
    })                                  Multithreading

    Knn final_knn = merge_knns(tp)

    result.add(final_knn.get_sorted())
```

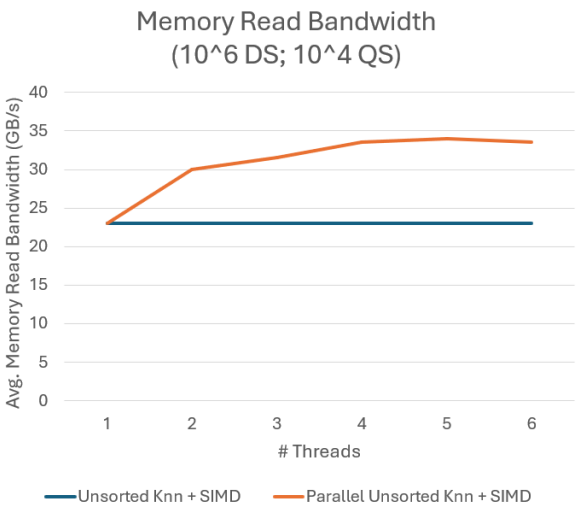
Performance Results:

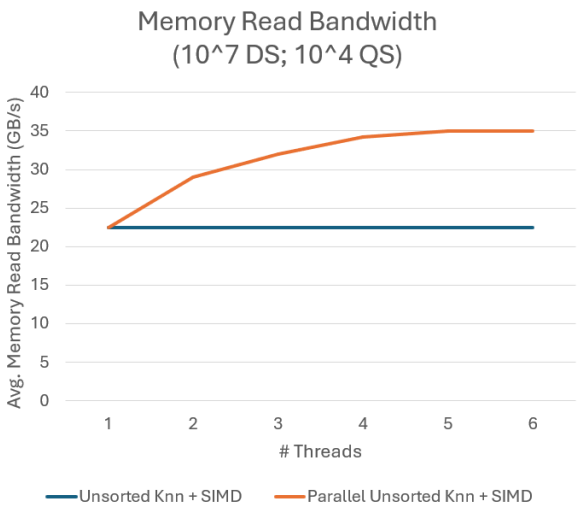
These two graphs show the total execution time in seconds when running the parallelized vector search with N threads. Note that again that I'm not using the sample proportion parameter thus the single-threaded large benchmark took up to 20 minutes. While performance improves noticeably up to 4 concurrent threads beyond this point performance gains are only minimal at best.





The next two graphs show the average memory read bandwidth for the same benchmark (measured using Intel PCM). Like the execution time above the memory bandwidth does not noticeably increase after using more than 4 threads.





Conclusion:

The read memory bandwidth not increasing with increasing thread count would most likely be the limiting performance factor. However, my system should theoretically allow for a maximum memory bandwidth of around 42 GB/s, so only reaching ~ 35 GB/s as maximum seems kind of low. Maybe other factors might have become bottlenecks on my system for 5 or more threads, but this would require further in-depth testing. For now, all performance measurements and characteristics seem to indicate the memory bandwidth as bottleneck. ¯_(ツ)_/¯

Additionally, I noticed my CPU's clock speed decreasing from ~ 3.8 GHz to ~ 3.66 GHz when using more and more threads. This might be due to some Intel Turbo Boost while Hyperthreading shenanigans.