Project – Hybrid Vector Search Queries

Jonas Fritsch

Project Outline

- 100 Dimensional Vectors (float)
- Find 100 nearest neighbours to a given query vector
- (also additional search constraints like vector category and timestamp)

- 2 Datasets:
  - N number of data vectors
  - M number of query vectors

Starting Point – Baseline Impl.

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

Starting Point – Baseline Impl.

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

Indexing

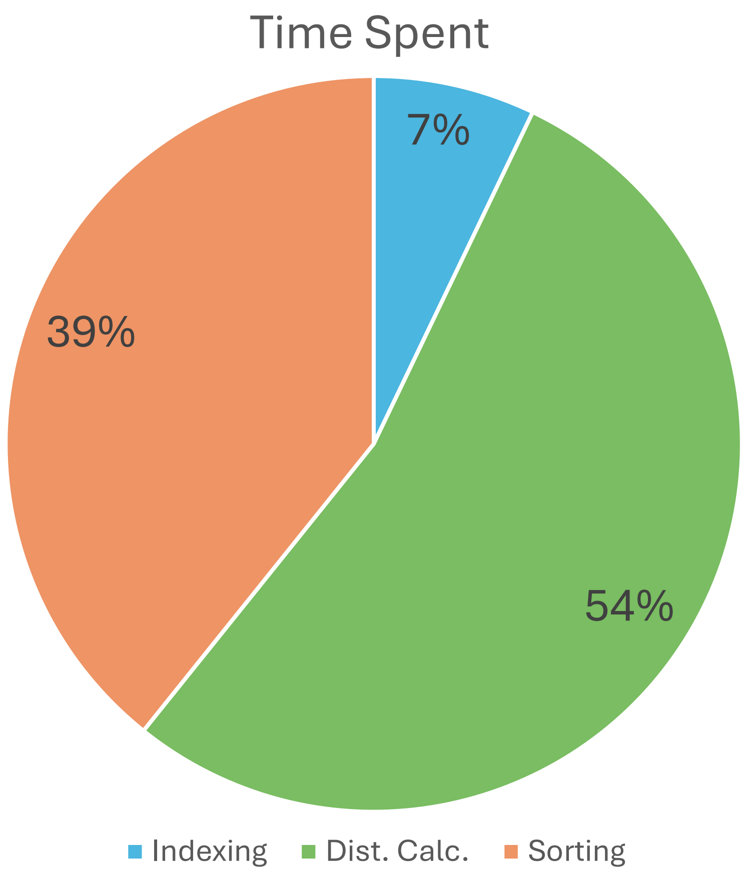
```
    list<float> knn_dist  //distances
    foreach c in knn_cand
        dist = calc_dist(c, q)
        knn_dist.add(dist)
```

Dist. Calculation

```
    sort_cand_based_on_dist(knn_cand, knn_dist)
```

Sorting

```
    result.add(knn_cand.take(100))
```



#1 Optimize Distance Calculation

```
// Calculates 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
```

Optimizations:

1. SIMD (AVX2 – 256 bit)
2. Early Bailout

#1 Optimize Distance Calculation

```
// Calculates 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
```

Optimizations:

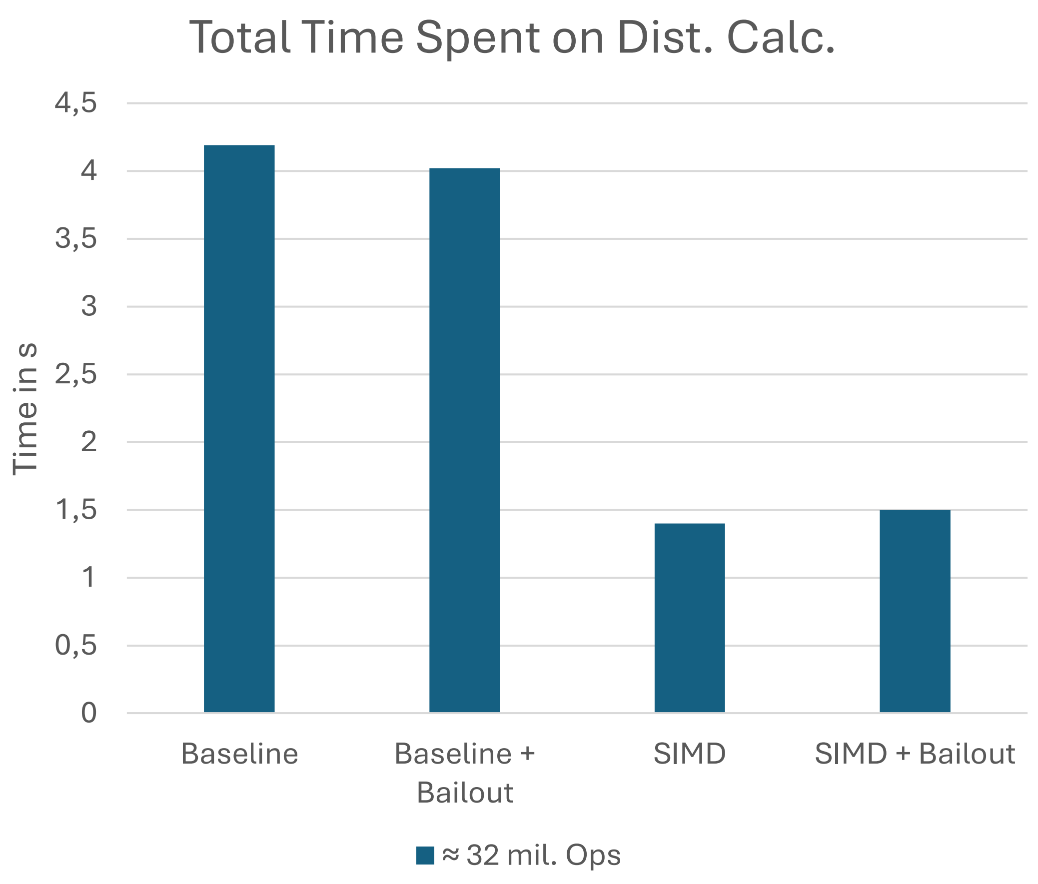
1. SIMD (AVX2 – 256 bit)
2. Early Bailout

Note – Floating Point Imprecision:

- Different Results
- SIMD more accurate (partial sums)

Distance Calculation – Optimized

- SIMD with 200% performance gain
- Early bailout for ~ 50% of calculations at ¾ of sum but no real performance gain

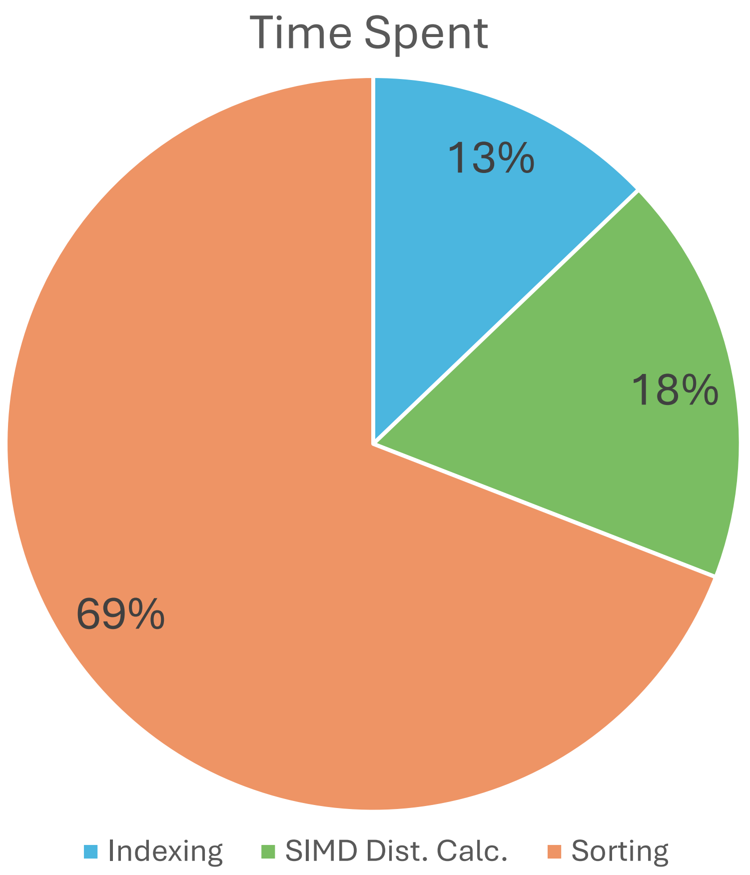


With SIMD Distance Calculation

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

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

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



#2 Use Better Data Structures for KNN

```
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
        if matches_query(d, q)
            knn_cand.check_add(d)

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

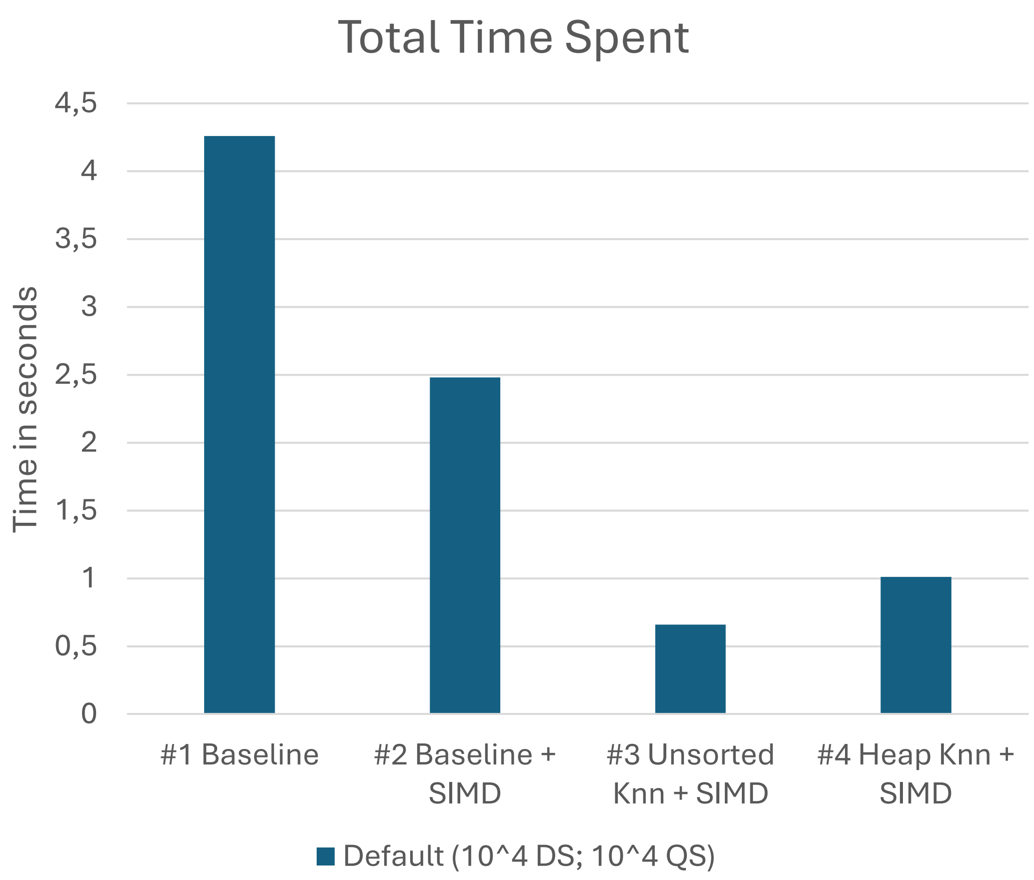
Indexing +

Dist. Calc.

Sorting

KNN Data Structure – Optimized

- Heap Knn asymptotically better than Unsorted Knn
- Unsorted Knn faster due to small Knn size (100) and SIMD optimizations
- Up to 275% performance gain using Knn data structure
- 545% performance gain over baseline



#3 Parallelization using Thread Pool

```
foreach q in queries
    ThreadPool<Knn> tp(num_threads, {q, 100})

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

    Knn final_knn = merge_knns(ts)

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

Parallelization