technische universität
dortmund

Bachelor's Thesis

Vector Similarity Search Optimization

Thorben Simon
Dortmund, January 16, 2025

Supervisors:
Prof. Dr. Jian-Jia Chen
Dr. Yun-Chih Chen

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
Design Automation for Embedded Systems Group
https://daes.cs.tu-dortmund.de/

Contents

1 Introduction

This chapter creates and overview of Vector Similarity Search, its importance in the data driven world, and outlines the motivation for this research.

1.1 Motivation

1.1.1 The Power of Vector Similarity Search

Everyday massive amounts of digital data like texts, documents, images, videos, etc. are created. Tomorrow even more. A large part of that data is unstructured, which presents a challenge: How can we search efficiently through this growing amount of information? The traditional way would be using keyword-based search, where the data is searched for matching words or strings. Like searching for a text passage in a document or searching for file by its name. However, this method only works on text data and not on a semantic level.

Therefore, vector similarity search can play an important role. It is able to compare the semantic meaning of a text based search queries to a database of images, texts, etc. This enables you, for example, to search a local photo gallery for pictures containing cats with the simple query 'cat'. Vector Similarity Search works by converting the input data (texts, images, ...) into embeddings, which are just high dimensional - numerical vectors. Encoded in these vectors are for example the meaning of its encoded text or the visual details of a picture. Recommendation systems of streaming platforms [1], social networks [2] and ecommerce [3] platforms also often use vector similarity search. It can even be used for machine language translation. [4]

To experience the results of this thesis first hand, the reader can download the project from git which includes a small python web server to search for Wikipedia articles using the implementations presented in this thesis. More info in section A.2.

1.1.2 The need for optimization

Consider a mobile photo gallery with 10,000 images. Each embedding requires 4 KB of memory, when using the float32 format. This would total 40 MB of memory use. An offline machine translator with 250,000 translations would require 1 GB of memory. With a typical device having only 4 GB of RAM, searching these embeddings could become slow and memory intensive. Now that our world is becoming increasingly mobile first a new problem arises. Because Vector Similarity Search is currently mostly run in datacenters, where the amount of RAM starts to get into the Terabyte territory, it is very expensive on resources like processing power and

memory. But there are valid reasons for running vector similarity search on resource constrained mobile devices. Like offline gallery search without the need to share all data with a cloud provider to offload computation onto the datacenters. Another use can be smart home devices, where the device matches the semantic meaning of voice commands to actions. Here latency would be very important for natural interaction. Embedded systems in vehicles can use vector similarity search for object detection and classification. Which also requires low latency and high reliability, while being resource constrained.

This creates the fundamental question: How can we perform vector similarity search on devices with limited resources while maintaining as much speed and accuracy as possible?

1.2 Research Objectives

The objective of this thesis is to addresses the previously mentioned optimization challenges for resource constrained devices by investigating and developing optimization techniques for Vector Similarity Search. The following approaches will be implemented, fine-tuned and compared to each other by evaluating their performance (speed, accuracy and memory usage):

- Quantization methods which can decrease memory usage and increase speed at the cost of accuracy
- Vector dimension reduction also can decrease memory usage and increase speed at the cost of accuracy
- Two-Step approach to balance speed and accuracy
- SIMD optimization to accelerate calculations by using modern hardware features

After this it should be possible to evaluate the effectiveness of these approaches and decide what is actually useful for a specific use case.

1.3 Structure

In chapter 1 an introduction to vector similarity search by describing its use cases and the importance of optimization is given. Additionally, the goals of this thesis are presented.

Chapter 2 gives a more in depth explanation on what exactly vector embeddings are, their capabilities and how the input data is encoded. Then the mathematical foundations used in the similarity computation are explained too. Next, hardware characteristics and capabilities that are important for similarity search are introduced. Finally, the metrics evaluating the approaches performance will be explained.

Chapter 3 gives a general overview of the software architecture that the different implementations will share.

In chapter 4 the implementation of each approach is presented. Additionally, it shows the progress for some methods from no optimizations to full optimizations. This way the decision on why something is done in a certain way becomes more clear.

Chapter 5 gives the parameters of the test environment first and then presents and analyzes the results with the help of plots generated from the recorded metrics.

In chapter 6 potential new research that could be done with the insights gained in this thesis are outlined.

Lastly, chapter 7 wraps this thesis up by summarizing the results and potential new questions that have surfaced from this research.

2 Theoretical Background

This chapter provides more background information about embeddings and their properties. The mathematical theory of vector similarity is also explained. This is important to fully understand the later chapters.

2.1 Vector Similarity Search

2.1.1 Vector embeddings and their properties

Vector embeddings are the numerical representations of the inherent properties of the encoded data in an n-dimensional vector space. The data is encoded using an embedding model. Models are specific for a datatype. There are models for text-embeddings, picture-embeddings, etc. The resulting vector dimension depends on the model itself. The most advanced text models use up to 8192 dimensions. [5] Different models produce also different embedding data on the same input data. The common number format used to handle these embeddings is float32. It guarantees high accuracy, probably way more than needed, at the cost of high processing power and high memory usage. Because we do not want to compare only two vectors but search for the vector most similar to vector a in a set of vectors. That means, that the computer has to load an array of vectors into memory. One float uses 4bytes and if one embedding has 1024 dimensions it will use `4bytes * 1024 = 4 KB` of memory.

Example 2.1: Text-to-Vector Mapping

How semantically similar phrases are mapped to similar vectors in the embedding space:

$$\texttt{"Hello world"} \quad \mapsto \quad [0.1, -0.3, 0.8, \ldots, 0.4]$$
$$\texttt{"Hi universe"} \quad \mapsto \quad [0.2, -0.2, 0.7, \ldots, 0.5]$$

Notice how these greetings, which have similar semantic meaning, are mapped to nearby points in the vector space, as illustrated in Figure 2.1.

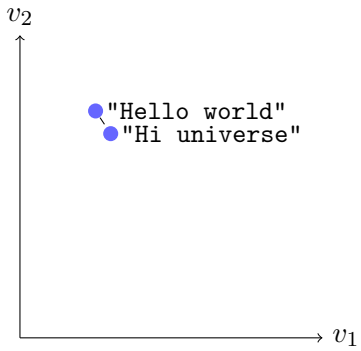

Figure 2.1: Visualization of similar phrases mapped to nearby vectors (projected onto 2D for illustration)

Remark 2.1: Dimensionality Note

We typically work with high-dimensional vectors (e.g., 768 or 1024 dimensions), the 2D visualization above just illustrates how semantically similar texts are closer together in the vector space.

2.1.2 Cosine similarity and other similarity metrics

Vector similarity is defined as the angle between two vectors. Perfect similarity means the angle between the vectors is $0°$. No similarity at all means the vectors point exactly in the opposite direction with an angle of $180°$ (or π).

Dot product

Given two vectors a and b with n dimensions the dot product is calculated as follows:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

The dot product gives the following information of the relation of two vectors:

- positive dot product $\rightarrow$ angle is $< 90°$
- dot product close to zero $\rightarrow$ angle is $= 90°$
- negative dot product $\rightarrow$ angle is $> 90°$

The dot product already helps to roughly estimate the similarity of two vectors.

Cosine similarity

The cosine similarity is derived from the dot product:

$$\mathbf{a} \cdot \mathbf{b} = |a||b| \cos(\theta)$$

Which means the norms of both vectors are multiplied with each other and then multiplied with the cosine of the angle between the vectors. This equitation can be rearranged in the following way to get the cosine similarity function:

$$\cos(\theta) = \frac{a \cdot b}{|a||b|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \cdot \sqrt{\sum_{i=1}^{n} b_i^2}}$$

The output values evidently range from -1 to 1. The Vectors in the cosine function are automatically normalized and one can not only roughly estimate the angle between the vectors, like with the dot product, but tell the exact angle. It's also possible to compare different angles of different vector pairs and tell which pair is the most similar with the cosine similarity.

- $\cos(\pi + 2n\pi) = -1 \rightarrow$ vectors point in exactly opposite directions ($180°$)
- $\cos(\frac{1}{2}\pi + n\pi) = 0 \rightarrow$ vectors are perpendicular ($90°$)
- $\cos(2n\pi) = 1 \rightarrow$ vectors point in the same directions ($0°$)

2.2 Hardware considerations

2.2.1 Resource constraints on mobile devices

Hardware specifications of devices where vector similarity search can be useful vary drastically:

- Typical server: 256GB+ RAM, 64+ cores
- High-end automotive system with object detection: 4-16GB RAM, 2-8 cores, various processing accelerators
- High-end phone: 8GB RAM, 8 cores
- Mid-range phone: 4GB RAM, 6 cores
- Mid-range automotive system: 512MB-2GB RAM, 1-2 cores, small processing accelerator
- IoT device: MT8516 A35 SoM, 512MB RAM, 4 cores

2.2.2 SIMD instructions and AVX2

Single instruction, multiple data, abbreviated by SIMD, is a type of processor organization that operates using "a master instruction applied over a vector of related operands." [6] This type of processor organization excells especially vector/array processing with minimal branching operations, large data parallelism and limited communication needs. Considering vector similarity search involves operations on large vectors using SIMD seems fitting. In comparison using multiprocessor systems (MIMD, multiple instruction, multiple data) is less optimal, because of the increased communication overhead between processors. On top of that many modern

processors (mobile arm, desktop -, mobile - and server x86-64 processors) support SIMD and enables them to process the data more efficiently in terms of power usage and computation time. [7, 8, 9]

Advanced Vector Extensions (AVX) are SIMD instructions for the x86-64 architecture. They operate on 256bit registers and enables the processor to do math operations on them. They can for example multiply 8 32bit floats with one instruction. AVX2 adds more instructions to the existing set. AVX2 works on most desktop and laptop processors released after 2015. AVX2 will be used to optimize the vector similarity search program.

2.2.3 Memory hierarchy and access patterns

In most cases of vector similarity search one search query gets compared to every embedding to get the best matching embeddings for this query. This implies that embedding vectors are loaded sequentually from memory. When we consider the memory hierarchy of modern systems is

```
non-volatile memory > main memory > L3 > L2 > L1 cache > CPU registers
```

with size and access time of the memory decreasing drastically from left to right as seen in Table 2.1. The search query most likely stays in cache all the time, because its always one of the two vectors that gets accessed during search. The predictable access patterns of the embedding vectors enables either automatic preloading of the soon to be used vectors by the CPU prefetcher or loading them manually by using a prefetch instruction, while computing similarity of the current embedding. But the prefetcher can't cross page boundaries which are the memory segments managed by the system. On most modern systems one page is 4kB large. [10] The importance of this will be explained in chapter 4.

But CPU cache can vary depending on the processor architecture and an effective prefetching strategy for one cpu might be slower for another. Cache is always relatively small so cache pollution by prefetching to much data has to be avoided.

Memory Level	Latency (CPU Cycles)	Latency (ns)	typical size
L1 Cache	∼4	∼1	64kB (per core)
L2 Cache	∼12	∼3	512kB (per core)
L3 Cache	∼60	∼15	8-32MB (shared)
RAM	∼400	∼100	8+GB (shared)

Table 2.1: Memory hierarchy access latencies. [11] Understanding these timing differences explains why memory access patterns significantly impact performance.

2.3 Evaluation Metrics

2.3.1 Jaccard Index

The Jaccard index is a statistical metric suitible for gauging the similarity of two sets. It is defined as the size of the intersection divided by the size of the union of the two sets:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The definition implies that $0 \leq J(A,B) \leq 1$. The Jaccard index of two sets that match exactly is 1 and 0 for two sets that have no matching elements. The Jaccard Index will be one metric used for comparing the accuracy of different optimization strategies by comparing the result set to a baseline result that used no optimization techniques.

2.3.2 Normaized Discounted Cumulative Gain (NDCG)

The Discounted Cumulative Gain (DCG) is an evaluation metric for information retrieval systems. It assumes that documents found later in the results i.e. results that are less similar than the first results, are less valuable to the users. As a result it uses a logarithmic discount factor to reduce the weight of documents appearing lower in the results. The NDCG is formally defined as [12]:

$$DCG(p) = \sum_{i=1}^{p} \frac{rel_i}{log_2(i+1)}$$

rel_i is the relevance of the element with index i. $log_2(i+1)$ is the logarithmic discount factor. The relevance can be either assigned manually by a human judging the relavance of the result or by calculating it by comparing it to an optimal result set. The algorithm used here calcultes the relavance score based on the position difference. Perfect match gets a score of 1. There is an exponential decay of the lost relevance for large differences. It uses $log_2(i+2)$ instead of $log_2(i+1)$ because in the algorithm the index i starts at 0.

Algorithm 2.3.1: Excerpt from algorithm used to calculate the NDCG.

```
// Calculate DCG
double dcg = 0.0;
// k is the size of the result sets
for (size_t i = 0; i < k; ++i) {
    auto it = truthPositions.find(prediction[i].second);
    if (it != truthPositions.end()) {
    // Calculate relevance score based on position difference
    double posDiff = abs(static_cast<double>(it->second) - i);
    double relevance = exp(-posDiff / k); // exp decay
    dcg += relevance / log2(i + 2); // DCG formula
    }
}
```

The normalized DCG (NDCG) is calculated by dividing the the DCG score by the ideal DCG possible for that query:

$$IDCG = \sum_{i=1}^{p} \frac{1}{log_2(i+1)}$$

$$NDCG = \frac{DCG}{IDCG}$$

The NDCG produces scores between 0 and 1, where 1 represents the ideal result. This property allows the comparison between different queries. In conclusion the NDCG is a relevant metric for judging the performance of the different optimization techniques that will analyzed in this thesis. It accounts for both document relevance and rank position. The relevance is not just binary 0 and 1 like it is for the jaccard index. It also models realistic user behaviour by discounting documents appearing further back in the results.

3 Implementation

3.1 Software Architecture

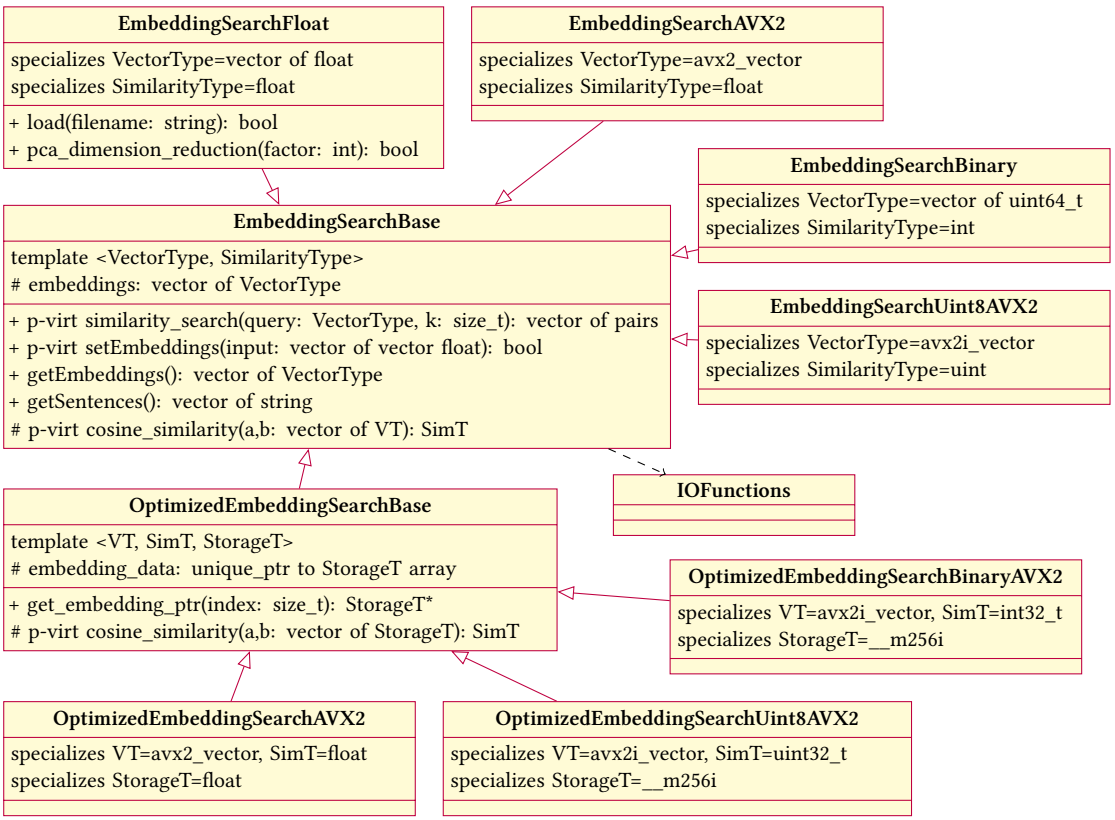

Figure 3.1: Simplified class relation diagram of the simsearch c++ program. p-virt mean the function is a pure virtual function

The Program which implements and benchmarks the different vector similarity search optimizations is written in modern C++. A simplified diagram of the class hierarchy can be seen in Figure 3.1.

3.1.1 Design philosophy

Overall the system is realized through template based class hierarchies. The base functionality is implemented in the base class like the loading of embeddings from disk. Functions like `cosine_similarity` are separated into the specialized classes because that's these functions are the one that will be optimized. Furthermore, the classes are implemented in a type-safe manner. This way type errors when using different vector formats (float, binary, int8) are caught at compile time rather than runtime. Also, many benchmark parameters are configurable when running the program from command line.

3.1.2 Class hierarchy

The mentioned class hierarchy allows implementing the different vector similarity search implementations without having duplicates of code. This also guarantees that the classes have a consistent interface. There are two abstract main classes which provide the foundation for all implementations: *EmbeddingSearchBase* and *OptimizedEmbeddingSearchBase*. The *EmbeddingIO* class allows loading of embeddings from the disk. Different formats are supported, but most of them are non-standard. The *load* function in the *EmbeddingSearchBase* class uses this class. After it has loaded the embeddings from the disk it passes the float32 vectors to the classes corresponding *setEmbeddings* implementation. This function converts the vectors into the expected format. For the classes that inherit from the *EmbeddingSearchBase* class the embeddings are stored in a vector of *VectorType*. Where *VectorType* can be a vector of floats, vector of integers or an aligned vector datatype. One element of *VectorType* represents one embedding and the vector of these are interpreted as the list or array of embeddings.

There also is the *PyEmbeddingSearch* class. This class is used to create python bindings. These bindings allow to use the different searcher implementations in python. These bindings will be used to gather benchmark metrics in python. One can also experiment and write their own queries and search for similar embeddings with the C++ implementation. This should help Understanding the functionality of the program.

Memory management strategies

The *AlignedTypes* class defines the memory aligned vector types *avx2_vector* and *avx2i_vector*. They represent a memory aligned vector of the *__m256* and *__m256i* datatype respectively, see: Algorithm 3.1.2. The main purpose of this class is to ensure proper memory alignment (done by Algorithm 3.1.1) for the SIMD operations. It is used by the *EmbeddingSearchAVX2* and *EmbeddingSearchUint8AVX2* classes. This allows these classes to store the AVX2 vectors in a memory aligned manner, which in turn enables these classes to use aligned loads into AVX registers which is faster than the unaligned load AVX instruction.

Algorithm 3.1.1: *AlignedAllocator* class template

```cpp
template <typename T>
class AlignedAllocator {
    static constexpr size_t alignment = 32; // AVX2 needs 32B alignment
    T* allocate(size_t n) {
        void* ptr = std::aligned_alloc(alignment, n * sizeof(T));
        if (!ptr) throw std::bad_alloc();
        return static_cast<T*>(ptr);
    }
    void deallocate(T* p, size_t) noexcept {
        std::free(p);
    }
};
```

Algorithm 3.1.2: Specialized vector types for AVX2

```cpp
template <typename T>
using aligned_vector = std::vector<T, AlignedAllocator<T>>;
// Vector of 256-bit float vectors
using avx2_vector = aligned_vector<__m256>;
// Vector of 256-bit integer vectors
using avx2i_vector = aligned_vector<__m256i>;
```

The optimized classes also use aligned memory to store the vectors. But in these classes a single contiguous block of memory is allocated to store the embeddings. Which allows better usage of the CPU cache. Memory is managed manually: Algorithm 3.1.3. It allows direct pointer arithmetic for even faster access. This approach removes the minimal overhead that *std::vector* has. This also enables the possibility to use memory prefetching, because the memory address of vectors accessed in the future is more predictable.

Algorithm 3.1.3: Excerpt from *OptimizedEmbeddingSearchBase*

```cpp
std::unique_ptr<StorageType[]> embedding_data;
// Used in derived classes:
StorageType* get_embedding_ptr(size_t index) {
    return embedding_data.get() + index * vectors_per_embedding;
}
bool allocateAlignedMemory(size_t total_size) {
  embedding_data.reset(static_cast<StorageType *>(std::aligned_alloc(
    config_.memory.alignmentSize, total_size * sizeof(StorageType))));
  return embedding_data != nullptr;
}
```

4 Optimization Approaches

This chapter introduces and discusses the optimization approaches that will be benchmarked later. It will start with the simple implementations and will gradually present the implementations with more performance optimizations. In chapter 5 only the most optimized versions will be tested.

4.1 Baseline float32 implementation

Algorithm 4.1.1: setEmbeddings function from the float implementation.

```
bool EmbeddingSearchFloat::setEmbeddings(
        const std::vector<std::vector<float>> &input_vectors) {
    initializeDimensions(input_vectors);
    embeddings = input_vectors;
    return true;
}
```

Algorithm 4.1.2: Loop calculates the similarity for every embedding with the query.

```
for (size_t i = 0; i < embeddings.size(); ++i) {
    float sim = cosine_similarity(query, embeddings[i]);
    similarities.emplace_back(sim, i);
}
```

The naive and simple float implementation resides in the class *EmbeddingSearchFloat*. In this class the embeddings vectors are set by simply copying the vectors from the load function: Algorithm 4.1.1. The similarities are calculated by iterating over all embeddings and comparing the similarity for each with the query. This stays the same for all searcher implementations. See Algorithm 4.1.2. The cosine similarity for two vectors is calculated by iterating over every vector element. Then the corresponding vector elements of the input vectors are multiplied to each other. The result is then added to the dot product. After the loop we have the correct dot product value in the value *dot_product*. The cosine similarity is then derived from the dot product like its explained in section 2.1.2. After all similarities are calculated in Algorithm 4.1.2, the results are sorted by the cosine similarity and returned. Generally this

Algorithm 4.1.3: Naive cosine similarity function.

```
float cosine_similarity(const std::vector<float> &a,
                        const std::vector<float> &b) {
    float dot_product = 0.0f;
    float mag_a = 0.0f;
    float mag_b = 0.0f;
    for (size_t i = 0; i < a.size(); ++i) {
        dot_product += a[i] * b[i];
        mag_a += a[i] * a[i];
        mag_b += b[i] * b[i];
    }
    return dot_product / (std::sqrt(mag_a) * std::sqrt(mag_b));
}
```

version is the slowest and uses the most memory because it doesn't use any quantization to reduce memory usage or special instructions to accelerate the computation.

4.2 Cosine Similarity Optimization

Normalizing the embeddings during loading or quantization eliminates the normalization calculation during cosine calculation. This greatly decreases the FLOPs required. This saves power and can also accelerate the calculation if it's not memory bottlenecked.

4.3 Quantization Methods

4.3.1 Int8 quantization

Algorithm 4.3.1: Int8 cosine similarity function.

```
int cosine_similarity(const std::vector<int8> &a,
                      const std::vector<int8> &b) {
    int dot_product = 0;
    for (size_t i = 0; i < a.size(); ++i) {
        dot_product += a[i] * b[i];
    }
    return dot_product;
}
```

While the int8 quantization is not implemented without any other optimization (like AVX2 or optimized memory management) it works by multiplying each vector element by 127. The embeddings have to be normalized for this. If they are not normalized the original values can exceed -1 or 1 and cause the 8-bit integer to overflow. The loop iterating over every embedding looks like the one shown in Algorithm 4.1.2. The cosine similarity is calculated by simply calculating the dot product as seen in Algorithm 4.3.1. Dividing by the magnitude of the vectors

isn't necessary, because the vectors have already been normalized. Int8 embeddings use 1/4 of the memory that the float32 embeddings use. The cosine calculation is also faster, because multiplying int8 values is faster than multiplying float32 values. Another speedup factor is the vector normalization before calculating the cosine similarity. One downside is that the multiplication results of the two vector elements have to be stored in an int16 value because in the worst case you need 16bit for the result of an 8-bit int multiplication.

4.3.2 Binary quantization

For the binary quantization the conversion from the float value to binary is very simple. Because each vector element has to be represented by one bit, it represents negative values as 0 and positive values as 1. Instead of just creating a vector of booleans, 64 vector elements are represented by one 64bit int value. Now the binary embeddings have 1/64 the original vector size. The algorithm that does the conversion is shown in Algorithm 4.3.2. Here the vectors don't need to be normalized, because a binary vector can have just one length anyway. Again the function iterating over every embedding and calculating the cosine similarity is like the one shown in Algorithm 4.1.2. Then we can use XNOR-based binary multiplication to build the dot product. The whole 64bit integer of the query gets XNORd with the integer from the embedding. Then popcount is used to count the ones as ones represent the matching bits. This is shown in Algorithm 4.3.3. The higher the returned dot product is the more similar the vectors are.

Algorithm 4.3.2: Quantization of binary embeddings

```
for (size_t i = 0; i < num_vectors; ++i) { // iterate over vectors
    for (size_t j = 0; j < float_vector_size; ++j) { // vec elements
        if (float_data[i][j] >= 0) { // when float has positive val
            size_t chunk_idx = j / 64;
            size_t bit_pos = j % 64;
            embeddings[i][chunk_idx] |= (1ULL << (63 - bit_pos));
        }
    }
}
```

Algorithm 4.3.3: Cosine similarity for binary embeddings

```
int cosine_similarity(const std::vector<uint64_t> &a,
                      const std::vector<uint64_t> &b) {
    int dot_product = 0;
    for (size_t i = 0; i < a.size(); ++i) {
        // Count matching bits using XOR and NOT
        dot_product += __builtin_popcountll(~(a[i] ^ b[i]));
  }
  return dot_product;
}
```

4.3.3 Float16 quantization

The float16 quantization is implemented using the software float16 implementation from C++. All the functions are similar to the naive float32 functions. Except that the embeddings in *setEmbeddings* are converted using the standard float16 constructor as seen in Algorithm 4.3.4. For this implementation its only relevant to compare accuracy and memory metrics. Time based metrics will be excluded, because the float16 software implementation is very slow and most desktop/laptop processors don't support float16 natively yet.

Algorithm 4.3.4: Excerpt from setEmbeddings function from the float16 class

```
embeddings[i][j] = std::float16_t(input_vectors[i][j]);
```

4.3.4 Mapped float quantization

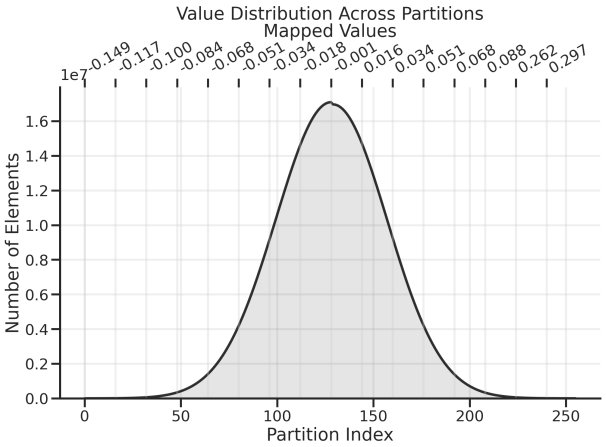

Figure 4.1: Value distribution of quantized test dataset

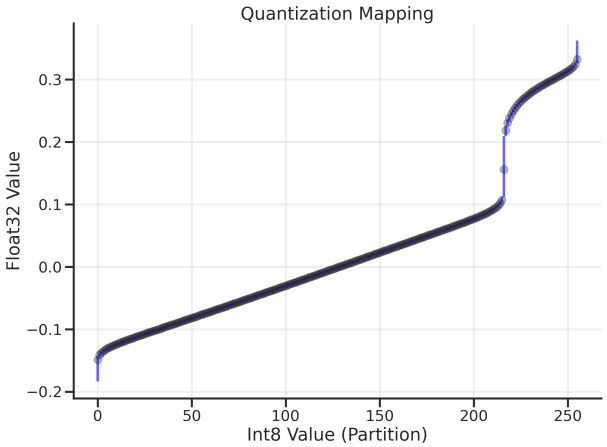

Figure 4.2: Quantization Mapping of quantized test dataset

The mapped float quantization uses a novel technique that combines quantization with value mapping to achieve high accuracy while reducing memory usage. Like the int8 quantization it uses 8bit integers to store the quantized values. But it doesn't just linearly map the values to the [-127, 127] range. This method analyzes the value distribution of the embedding vectors during quantization and adapts the quantized values to it.

In the first step of quantization the embedding matrix is flattened into a single array in which the float values are sorted. This gives the distribution of all values across all embeddings. For the test dataset with 1 200 000 embeddings with 1024 dimensions, this would result in 1.2 billion float values to sort.

Now the sorted values are partitioned into 256 segments using a Gaussian distribution weighting. This is implemented in the `partitionAndAverage` function: Algorithm 4.3.5. Here *relative_pos* is the normalized position (-1 to 1). *factor* controls the shape of the distribution and can be used for fine tuning. This weighting guarantees more precision in dense regions of

Algorithm 4.3.5: Distribution weighting

```
double weight = std::exp(-relative_pos * relative_pos * factor);
```

the distribution, while also allowing outliers that are closer to -1 or 1 to keep their relatively high impact during the similarity computation. For each partition its most important values are stored inside the *PartitionInfo* struct. The average values of each partition are also stored in the *mapped_floats* array. At the end of the quantization process, each float value in the

Algorithm 4.3.6: PartitionInfo and mapped_float array

```
struct PartitionInfo {
  float start;    // First value in partition
  float end;      // Last value in partition
  float average;  // Average of all values in partition
};
alignas(32) float mapped_floats[256];
```

original embedding is mapped to an uint8 index defined by the partitions' boundaries it fits into. The partition is efficiently searched via binary search. The cosine similarity function uses AVX2 gather instructions to efficiently load the corresponding mapped float values for the given uint8 indices as shown in Algorithm 4.3.7. The query vector stays as is because it's

Algorithm 4.3.7: Excerpt from mapped float cosine similarity

```
__m256i indices_b = _mm256_cvtepu8_epi32(...);
__m256 values_b = _mm256_i32gather_ps(mapped_floats, indices_b, 4);
```

kept as a float32 vector the entire time as it would make no sense to convert it to the uint8 indices and then back to float. In conclusion this method has the following advantages:

- It only uses 1/4th of the memory the float32 based searchers needs.
- The weighted partitioning gives precise Quantization in dense areas but still can represent outliers good enough to impact the dot product.
- Small, 1 kB mapping table can stay in L1 cache.
- Gather instructions are usually to slow to make sense combining them with vector instructions. But as the mapping table stays entirely in L1 cache the latency penalty is less bad.

4.4 Dimension reduction

Principal Component Analysis (PCA) is another approach to reduce memory usage and accelerating the search by reducing vector dimensions. The simsearch program has a function

that takes the embedding matrix and a reduction factor as input. When the function is finished it will return the dimension reduced embedding matrix, the PCA-matrix and the mean vector. The latter two are required to convert search queries to the same dimension as the dimension reduced embeddings. This method uses the Eigen library to simplify the matrix and vector operations. From now on the original vector dimension will be d and n is the number of embeddings. The embedding matrix will be called M. Logically M is an $n \times d$ Matrix.

In the first step the mean vector across all embeddings is calculated and then subtracted form each vector. This centers the data around the origin and increases the PCA effectiveness.

In the next Step the $d \times d$ covariance matrix gets calculated: $Cov = \frac{(M^T \cdot M)}{n-1}$ This matrix describes the relationship between dimensions.

Now the eigenvalues and eigenvectors of the covariance matrix are calculated. The eigenvectors get sorted by descending eigenvalue magnitude. Larger eigenvalues suggest more important principal components.

Next we do the dimensionality reduction: Select top $k = d/reduction\_factor$ eigenvectors. With these top k eigenvectors, the PCA-matrix P of size $d \times k$ gets created. The original embeddings now can get dimension reduction by multiplying it with the PCA-matrix: $M_{reduced} = M \cdot P$
$M_{reduced}$ now has the dimensions $n \times k$.

Finally, in the last step, the vectors have to get normalized again.

4.5 SIMD optimization

4.5.1 Float32 implementation with AVX2

Algorithm 4.5.1: Converting float vector to AVX2

```
void convertEmbeddingToAVX2(const std::vector<float> &input,
                            avx2_vector &output, size_t vector_dim) {
    for (size_t j = 0; j < vector_dim; j++) {
        size_t k = j * 8; // Each AVX2 vector holds 8 floats
        // will load 8*32bit -> input[i] .. input[i+7]
        output[j] = _mm256_loadu_ps(&input[k]);
    }
}
```

Algorithm 4.5.2: Cosine similarity for AVX2 float vectors

```
1  inline float _mm256_reduce_add_ps(__m256 x) {   // Sum all 8 floats in x
2    __m128 high128 = _mm256_extractf128_ps(x, 1); // Get upper 4 floats
3    __m128 low128 = _mm256_castps256_ps128(x);    // Get lower 4 floats
4    __m128 sum = _mm_add_ps(high128, low128);     // Add hi+lo -> 4 floats
5    sum = _mm_hadd_ps(sum, sum);     // Adjacent pairs -> (a+b,c+d,a+b,c+d)
6    sum = _mm_hadd_ps(sum, sum);     // Final sum in lowest float
7    return _mm_cvtss_f32(sum);       // Extract lowest 32-bit float
8  }
9  float cosine_similarity(const avx2_vector &a,
10                         const avx2_vector &b) {
11   __m256 dot_product = _mm256_setzero_ps(); // <-
12   __m256 mag_a = _mm256_setzero_ps();       // <- init all 3 vars with 0
13   __m256 mag_b = _mm256_setzero_ps();       // <-
14   for (size_t i = 0; i < a.size(); ++i) {
15     __m256 prod = _mm256_mul_ps(a[i], b[i]);        // multiply vectors
16     dot_product = _mm256_add_ps(dot_product, prod); // add to running sum
17     mag_a = _mm256_add_ps(mag_a, _mm256_mul_ps(a[i], a[i]));//mag_a+=a^2
18     mag_b = _mm256_add_ps(mag_b, _mm256_mul_ps(b[i], b[i]));//mag_b+=b^2
19   }
20   float dot_product_sum = _mm256_reduce_add_ps(dot_product);
21   float mag_a_sum = _mm256_reduce_add_ps(mag_a);
22   float mag_b_sum = _mm256_reduce_add_ps(mag_b);
23   return dot_product_sum / (sqrt(mag_a_sum) * sqrt(mag_b_sum));
24 }
```

This implementation will have the same accuracy and memory usage as the naive float implementation but it will be faster because AVX2 can multiply two vectors, consisting of 8 floats each, at once. We convert the default float vectors into AVX2 by simply loading the original vectors in 256 bit / 32 Byte steps into the AVX2 Vector as shown in Algorithm 4.5.1.

Computing the cosine similarity works generally like in the naive float version but with AVX2 instructions. The computation will be explained with the algorithm in Algorithm 4.5.2. Line 11-13 initializes the running sums for the dot product and magnitudes. The for loop iterates over the whole vector where each element is actually an AVX2 vector with 8 float elements. Next the two vectors are multiplied. After that the product is added to the dot product. Line 17, 18 squares each vector and then adds the result to the magnitude for the corresponding vector. After the for loop the sum of all float values in the three AVX2 vectors (*dot_product*, *mag_a*, *mag_b*), also called horizontal sum, are calculated with the helper function *_mm256_reduce_add_ps* in Algorithm 4.5.2.

4.5.2 Int8 quantization with AVX2

The conversion from float to int8 works like explained in section subsection 4.3.1. But this time we can store 32 int8 values in one AVX2 vector. The similarity gets computet in two steps: Because the results gets stored as int16 because of the potential overflow we can actually just multiply 16 values at a time. For this we first extract the lower 128bit in line 7 and 8 and multiply in line 6. The result gets stored as 256bit value, because we extend from 16*8bit to

16*16bit during multiplication. The same happens with the higher 128 bits in line 9-11. In the last step we add the *mul_lo* and *mul_hi* products to their corresponding running sum *sum_lo* and *sum_hi*. After the loop we build the horizontal sum of the two running sum vectors.

Algorithm 4.5.3: Cosine similarity for int8 with AVX

```
uint cosine_similarity(const avx2i_vector &a,
                       const avx2i_vector &b) {
  __m256i sum_lo = _mm256_setzero_si256();// Accum. for lower 128 bits
  __m256i sum_hi = _mm256_setzero_si256();// Accum. for upper 128 bits
  for (size_t i = 0; i < a.size(); ++i) { // 32 elements per iteration
    __m256i mul_lo = _mm256_mullo_epi16(  // extr and mult lower 128 bits
        _mm256_cvtepi8_epi16(_mm256_extracti128_si256(a[i], 0)),
        _mm256_cvtepi8_epi16(_mm256_extracti128_si256(b[i], 0)));
    __m256i mul_hi = _mm256_mullo_epi16(  // extr and mult upper 128 bits
        _mm256_cvtepi8_epi16(_mm256_extracti128_si256(a[i], 1)),
        _mm256_cvtepi8_epi16(_mm256_extracti128_si256(b[i], 1)));
    sum_lo = _mm256_add_epi32(sum_lo, // Accum. into 32bit to prev overfl
               _mm256_madd_epi16(mul_lo, _mm256_set1_epi16(1)));
    sum_hi = _mm256_add_epi32(sum_hi, // Accum. into 32bit to prev overfl
               _mm256_madd_epi16(mul_hi, _mm256_set1_epi16(1)));
  } // horizontal sum gets built and returned here
}
```

4.5.3 Binary quantization with AVX2

Algorithm 4.5.4: Cosine similarity for binary with AVX

```
int EmbeddingSearchBinaryAVX2::cosine_similarity(const avx2i_vector &a,
                                                 const avx2i_vector &b) {
  int dot_product = 0;
  __m256i all_ones = _mm256_set1_epi32(-1); // set every bit to 1
  for (size_t i = 0; i < a.size(); ++i) {
    __m256i result = _mm256_xor_si256(a[i], b[i]); // XOR
    result = _mm256_xor_si256(result, all_ones);   // NOT
    uint64_t *result_ptr = reinterpret_cast<uint64_t *>(&result);
    dot_product += __builtin_popcountll(result_ptr[0]); // sum all 64 bit
    dot_product += __builtin_popcountll(result_ptr[1]); // popcounts for
    dot_product += __builtin_popcountll(result_ptr[2]); // 256 bit result
    dot_product += __builtin_popcountll(result_ptr[3]);
  }
  return dot_product;
}
```

Just like in subsection 4.3.2 we just store the sign of the float embeddings. This time we can store 256 binary values in one AVX2 vector. Cosine similarity is computed by using XOR on both 256 bit vectors and then using XOR with a vector full of ones on the result. This flips

every bit of the result. After that we have the XNOR result. We have to use *popcountll* four times because each can only build the popcount of 64 bits. See Algorithm 4.5.4.

4.5.4 Optimized float32 implementation with AVX2

Like explained in section 3.1.2 the optimized implementation uses manually managed memory to store the embeddings. The embeddings are set using memcpy. A key optimization of this implementation is the use of strided memory access which increases memory bandwidth in combination with a more efficient vector loading strategy. This seems to be contrary to common knowledge, because sequential memory access should always be preferred it was also tested in isolation here section A.1. The stride distance automatically gets adapted to the vector size as the stride length being a multiple of 4 KB seems to give the highest memory bandwidth. A reason for this could be the systems' page size, which is 4 KB. So accessing multiple pages at once could increase the bandwidth. Another reason could be memory interleaving. This implementation also uses fused multiply-add (FMA), which has the benefit of being faster and more accurate due to less rounding and being more predictable. [13]

The optimized vector loading leverages the fact, that the query vector stays the same across all comparisons. The result of this is just 7 loads (1 for query, 6 for embeddings) for 6 FMA instructions instead of 12 loads (6 for query, 6 for embeddings) for the same amount of FMA instructions. The loaded part of the query vector can also stay in one register while being multiplied with the 6 embeddings. This can be seen in Algorithm 4.5.5 Additionally, the regular pattern of 7 loads followed by 6 FMA instructions allows the CPU to better predict and optimize instruction execution. The function expects both vectors to be normalized already. This way it doesn't have to calculate that magnitudes of the vectors, which also saves some computation time.

Algorithm 4.5.5: Cosine similarity for optimized AVX2 float vectors

```
1  inline void cosine_similarity_optimized(
2      const float *vec_a, float *sim, float *emb_ptr[]) {
3    __m256 sum[NUM_STRIDES] = {_mm256_setzero_ps()}; // sum for all embeds
4    __m256 a;               // 1 query
5    __m256 b[NUM_STRIDES]; // load NUM_STRIDES(=6) embeddings at once
6    for (int i = 0; i < padded_dim; i += 8) {
7      a = _mm256_load_ps(vec_a + i);        // load query vec
8      b[0] = _mm256_load_ps(emb_ptr[0] + i);//emb_ptr[0] pnts to curr. vec
9      // do the same for indices 2,3,4 ...
10     b[5] = _mm256_load_ps(emb_ptr[5] + i);//pnts to 5*strd_dst vec ahead
11     sum[0] = _mm256_fmadd_ps(a, b[0], sum[0]); // FMA: sum += a * b[0]
12     // do the same for indices 2,3,4 ...
13     sum[5] = _mm256_fmadd_ps(a, b[5], sum[5]); // FMA: sum += a * b[5]
14   } // compute horizontal sum for each sum and write to sim array ...
15 }
```

In Algorithm 4.5.6 can be seen how the *embedding_search* function prepares the array of pointers to the vectors *STRIDE_DIST* apart (for loop line 6-8). The loop starting at line 1 incre-

ments the index by *NUM_STRIDES * STRIDE_DIST*, because that's the distance the inner loop (line 3-11) covers.

Algorithm 4.5.6: Excerpt from embedding_search function of optimized AVX2 float class

```
for (size_t i = 0; (i + (NUM_STRIDES * STRIDE_DIST) - 1) < num_vectors;
       i += (NUM_STRIDES * STRIDE_DIST)) { // inc by total dist covered
    for (size_t j = i; j < (i + STRIDE_DIST); j++) { // by inner loop
      float sim[NUM_STRIDES] = {}; // cosine fn writes result into this
      float *emb_ptr[NUM_STRIDES]; // arr of ptrs to vectors STRIDE_DIST
      for (int k = 0; k < NUM_STRIDES; k++) { // apart
        emb_ptr[k] = get_embedding_ptr(j + k * STRIDE_DIST);
      }
      cosine_similarity_optimized(query_aligned.data(), sim, emb_ptr);
      // store results...
    }
  }
```

4.5.5 Optimized int8 implementation with AVX2

The cosine function of this is mostly similar to the one already presented in subsection 4.5.2. It uses the memory in the same way as the optimized float32 implementation in subsection 4.5.4.

4.5.6 Optimized binary implementation with AVX2

Algorithm 4.5.7: Cosine similarity for optimized AVX2 binary vectors

```
int32_t cosine_similarity_optimized(const __m256i *vec_a,
                                    const __m256i *vec_b) const {
  // prefetch 2 cache lines for 4 vectors 10 loops ahead
  _mm_prefetch(vec_a + 4 * 10, _MM_HINT_T0);
  _mm_prefetch(vec_a + 4 * 10 + 2, _MM_HINT_T0);
  __m256i all_ones = _mm256_set1_epi32(-1);
  __m256i xor_result[4];
  xor_result[0] = _mm256_xor_si256(vec_a[0], vec_b[0]);      // XOR
  xor_result[0] = _mm256_xor_si256(xor_result[0], all_ones); // NOT
  xor_result[1] = _mm256_xor_si256(vec_a[1], vec_b[1]);      // XOR
  xor_result[1] = _mm256_xor_si256(xor_result[1], all_ones); // NOT
  // do the same for indices 2 and 3
  // popcnt lookup is faster than harley seal and builtin
  return counter.popcnt_AVX2_lookup( // optimized version using AVX2
      reinterpret_cast<const uint8_t *>(xor_result),
      4 * sizeof(__m256i));
}
```

The embeddings are set similarly to method described in subsection 4.3.2 and subsection 4.5.3. Like the optimized float implementation in subsection 4.5.4. This cosine implementation also

uses loop unrolling and manual prefetching. The XNOR based binary multiplication matches the previous one in subsection 4.5.3. But a big improvement is the usage of 256-bit popcount algorithm implemented with AVX2 instructions, which was developed by Muła, Kurz, and Lemire. [14] Cosine similarity algorithm can be seen here Algorithm 4.5.7.

4.6 Two-step search with rescoring

The two-step approach uses a fast but lower accuracy search first to filter for candidates. Then a full accuracy search is done on these candidates. The top results from the second search are the final result. The goal of this method is to combine the speed of the lower accuracy search with the high accuracy of full precision search. The amount of documents the first search retrieves can be tuned with the rescoring factor. The amount of documents retrieved by the first searcher is defined by $rescoring\_factor * k$. In the second step the top k documents will be retrieved. There are two two-step searchers that will be tested: The first is implemented using the optimized versions of the binary and float32 searchers displayed in Algorithm 4.6.1. The second one uses the optimized binary searcher and the mapped float searcher for rescoring.

Algorithm 4.6.1: Two-step search with binary searcher and full precision searcher

```
// Binary search for initial filtering
auto binary_results = binary_avx2_searcher->similarity_search(
    queryBinaryAvx2,     // Binary query vector
    k * rescoring_factor // Get more candidates for rescoring
);
// Full precision rescoring
auto avx2_results = avx2_searcher->similarity_search(
    query,              // Original float query
    k,                  // Final number of results
    binary_results      // Pre-filtered candidates
);
```

5 Experimental evaluation

5.1 Methodology

5.1.1 Dataset

The dataset which will be used for benchmarking the optimization methods described in the previous chapters is the Wikipedia article dataset from Wikimedia available on hugging face[1]. This set contains over 6 million English articles from Wikipedia. Instead of encoding all articles, which would result in almost 23 GB in vector embeddings, 1.2 million randomly selected articles were encoded resulting in 4.6 GB of embeddings for 1024 dimensions. The data was converted by two different embedding models: The first one is the `mixedbread-ai/mxbai-embed-large-v1` model which is a very good performing model on the *Massive Text Embedding Benchmark* (available here[2]) paper. [5] This is a vector angle optimized model [15] which should give binary quantization an advantage compared to traditional models. Additionally, this model embeds to 1024 vector dimensions and has 335M parameters. To have a comparison the dataset of 1.2 million articles will be encoded with the `sentence-transformers/all-mpnet-base-v2` model available here[3]. This model maps the encoded text to 768 dimensions and has 109M parameters. It scores a bit worse on the *Massive Text Embedding Benchmark* but is still a solid model (rank 117 of 476 for the sentence-transformer model compared to 35 of 476 for the mixedbread-ai model as of 2024-11-29).

Unless mentioned or annotated otherwise the results and figures presented in this chapter use the `mixedbread-ai/mxbai-embed-large-v1` model with retrieval for top 100 (k=100) documents.

To benchmark the different search methods a list containing 303 queries will be used. It contains long and short-, specific and unspecific-, pointless queries, etc.

5.1.2 Evaluation metrics

To evaluate the benchmark results the Jaccard index and NDCG will be used to compare the performance to the naive float implementation as a baseline. Additionally, the time taken for each search will be measured. This includes initialization of the results array, calculating the similarity for every embedding, sorting for top k results and returning the result. At last the

[1] `https://huggingface.co/datasets/wikimedia/wikipedia`
[2] `https://huggingface.co/spaces/mteb/leaderboard`
[3] `https://huggingface.co/sentence-transformers/all-mpnet-base-v2`

Method Name	Short Name	Section
naive float	float	section 4.1
Optimized Bianry with AVX2	binary	subsection 4.5.6
Optimized Int8 with AVX2	int8	subsection 4.5.5
Float16	float16	subsection 4.3.3
Mapped Float	mf	subsection 4.3.4
PCAX (X indicates reduction factor)	pcaX	section 4.4
Two-Step Binary + Float (X indicates rescoring factor)	twostep_rfX | ts_rfX	section 4.6
Two-Step Binary + Mapped Float (X ind. resc. factor)	ts_mf_rfX	section 4.6

Table 5.1: Short names used in plots

theoretical memory usage will be calculated, see Table 5.2.

Method	Formula for embedding size in bytes
general	$sizeof(datatype) * vector\_dim * num\_embeddings$
rescoring	$general(binary) + general(rescoring\_method)$
PCA	$general(datatype) + vector\_dim * (reduced\_dim + 1) * sizeof(datatype)$

Table 5.2: Calculation of memory used for different methods

Method	Memory usage	Mem. usage test Dataset
float (including avx2 version)	$base$	4.58 GB
binary	$\frac{base}{32}$	0.14 GB
int8	$\frac{base}{4}$	1.14 GB
float16	$\frac{base}{2}$	2.29 GB
mapped float	$\frac{base}{4} + 1kB$ for mapping table	1.14 GB
PCA by factor x (float)	$\frac{base}{x} + d * (\frac{d}{x} + 1) * 4B$	$\sim \frac{4.58GB}{x}$
two-step binary+float	$base + \frac{base}{32}$	4.72 GB
two-step binary+mapped float	$base + \frac{base}{4} + 1kB$	1.29 GB

Table 5.3: Memory usage as ratio to baseline
d=vector dimension

5.1.3 Testing environment

All benchmarks were recorded on Linux using kernel version 6.11. Specifically version `6.11.10-300.fc41.x86_64` shipped by fedora. The test system has an AMD 5950X CPU (16C/32T) with disabled turbo boost to get a consistent frequency of 3.4GHz. Furthermore, 128 GB of dual-channel/quad-rank DDR4 SDRAM running at 3600MTs/1800MHz with CL18 was used.

5.2 Expected Outcome

No implementation can have the same accuracy as the float32 variants as every type of quantization and dimension has some kind of information loss. The binary method should be by far the fastest as it has very little data to work through, resulting from its heavy quantization, and efficient multiplication method. In return the accuracy should be fairly low. The float16 should be fairly good as using it for machine learning is already quiet common. The int8 implementation should be around 2 to 4 times faster than the optimized float32 version, as it does 16 multiplications at once compared to 8 and integer math can be faster depending on architecture. The mapped float implementation should have good accuracy, better than int8 but worse than float16 as its accuracy is much higher. The performance can be good if it's not bottlenecked by the random access latency on the mapping table. The search speed on the PCA variants should scale linear as the dimension decreases. PCA2 should decrease the search speed to about half (plus a little overhead) compared to the AVX2 float variant, as they both use the same implementation. With the only difference that PCA has the applied dimension reduction. The accuracy for the PCA searchers is hard to predict, but if there are redundant dimensions they should perform quiet good.

5.3 Results and Analysis

5.3.1 Search Time comparison

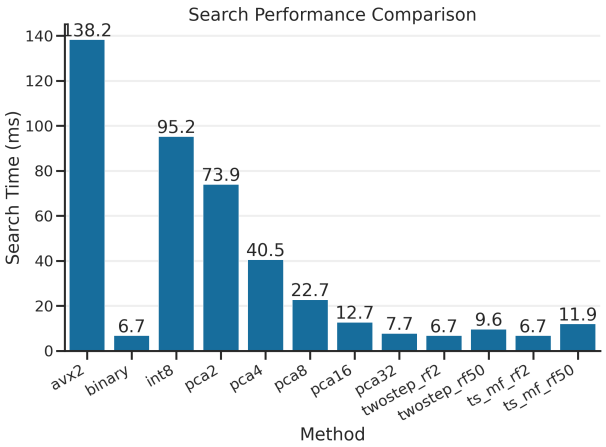

Figure 5.1: Search Speed vec_dim=1024
float (1100 ms) and mapped float (540 ms) ommited

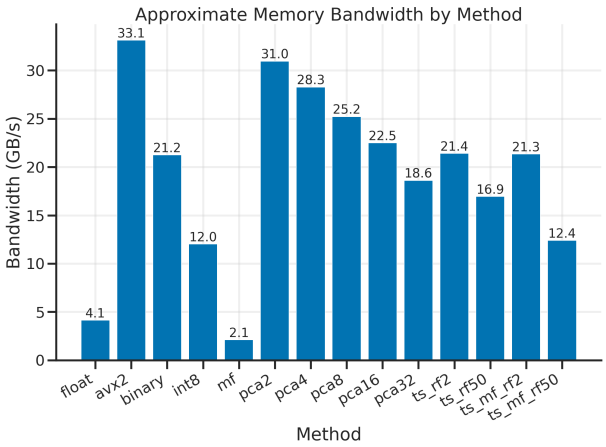

Figure 5.2: Memory bandwidth vec_dim=1024

Starting with Figure 5.1: As expected the AVX2 optimized version is much faster than the naive implementation. The fact it's 8 times faster, taking only 138.2 ms compared to 1100 ms for float, shows that this implementation makes efficient use of the AVX2 instructions.

The binary implementation, which also uses AVX2, is over 23 times faster than the AVX2 float equivalent and over 160 times faster than the baseline float implementation. Which makes

sense, because it only has to iterate through $1/32$th of the data and uses efficient nor, xor and popcount instructions.

With 95 ms The int8 implementation is slower than expected. But that is likely due to the fact that AVX2 can't multiply 32 int8 values at once. It has to extract the lower/higher 128 bits, extend them to 16bit and then do 2 multiplications for the high and low values. After that it has to build the horizontal sum and add the high and low multiplication result to the running sum. This is a significant amount of instructions compared to the AVX2 float implementation where 32 floats are multiplied and added to the running sum by just 4 FMA instructions.

The PCA variants perform quiet well. The dimension reduction to half almost doubles the search speed compared to the float AVX2 implementation, which makes sense, given the fact that the vectors only have half the dimensions. The other PCA factors almost scale linearly as well. Which suggests, that the overhead from the functions involved calculating the similarities is very low. PCA4 is more than 2 times faster than int8 which has to go through the same amount of data. This shows that AVX2 doesn't really perform well at multiplying int8 values.

The two-step method based on the binary and float AVX2 implementation is very close to the pure binary search time. Even for high numbers of retrieved documents and rescoring factors, the amount of embeddings the second step method has to search is very small, e.g. 5000 for k=100 and rescoring factor of 50 (like in Figure 5.1), compared to the total number of embeddings.

With 540ms the mapped float searcher has the second-worst search time after the naive float. The reason for this being, that the CPU prefetcher can only predict the int8 values that will be loaded next. But after this gather instructions are used to load the corresponding float values from the mapping table. Even though it's very small (1 KB) and likely stays in L1 cache the entire time, even the very low L1 cache latency of around 1ns adds up: Taking the benchmarked dataset with 1.2M embeddings with 1024 dimensions it would take $10^{-9}s * 1024 * 1.2 * 10^6/8 = 153.6ms$ just to load the float values for the embeddings. And this makes the assumption, that loading 8 float values at once takes 1ns. In conclusion, this methods' reliance on random memory access slows the search significantly. This is also seen in the low memory bandwidth in Figure 5.2.

Finally, the two-step method using binary and the mapped float performs similar to binary just like the two-step method that uses full accuracy search in the second step. Only as the rescoring factor increases it takes slightly longer than its competitor. That happens, because the mapped float searcher is much slower than the avx2 searcher and as the rescoring factor increases the speed of the second-step searcher becomes more relevant. Nonetheless, this search method performs really well, as long the rescoring factors are reasonable.

5.3.2 Accuracy Analysis

The AVX2 implementation has perfect accuracy as seen in Figure 5.3. But one can spot one outlier for the Jaccard index and NDCG score in Figure 5.5 and Figure 5.6 respectively. But that is likely due to the fact that the AVX2 implementation can actually be more accurate with the fused multiply add instruction, which eliminates one rounding step.

The binary searcher performs decent with an average NDCG of 0.576 and Jaccard index of 0.438 when considering its heavy quantization, the fast search time and low memory usage.

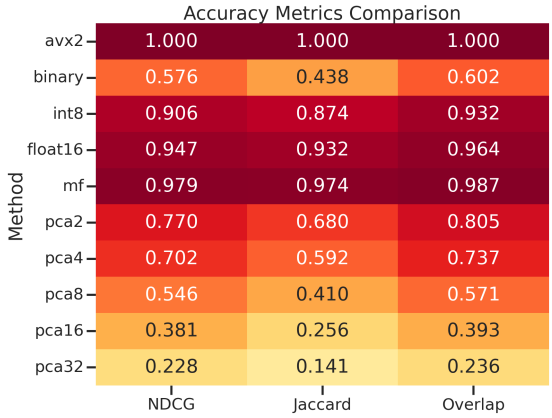

Figure 5.3: Accuracy of searchers

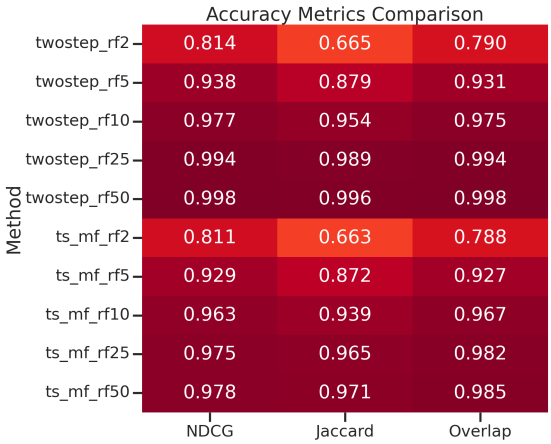

Figure 5.4: Accuracy of two-step searchers

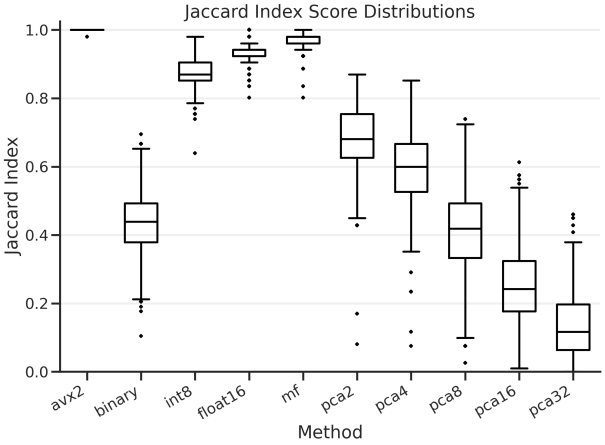

Figure 5.5: Jaccard index of searchers

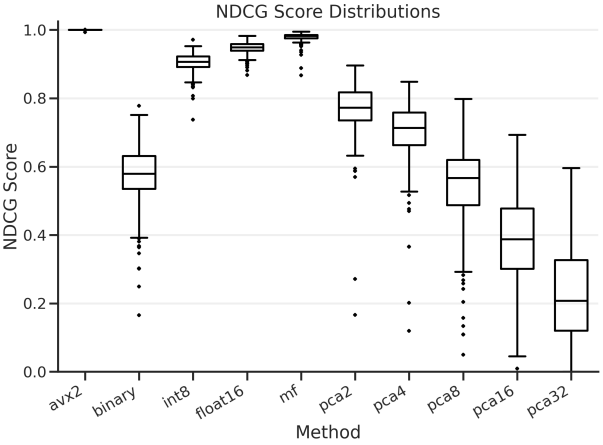

Figure 5.6: NDCG score of searchers

The PCA32 method, for example, uses the same amount of memory but has a much lower score which makes it useless in comparison to the binary searcher. The NDCG being higher than the Jaccard index indicates that the binary searcher also retrieves the important documents quiet reliably. In the box plots in Figure 5.5 and Figure 5.6 show some outliers close to, and below 0.2 Jaccard index and slightly above 0.2 NDCG. This means that for some queries it can be unreliable on its own.

The int8 based searcher performs well with an average NDCG of 0.9 and Jaccard index of 0.87. It performs exclusively above the binary searcher. The outliers are also a lot less drastic. Especially the NDCG stays above 0.8 for all queries except one.

Float16 or half precision float has very good average accuracy scores Figure 5.3. The box plots in Figure 5.5 and Figure 5.6 also show it performing very good. Especially the first and third quartile have a very close range and stay above 0.92 Jaccard index and 0.94 NDCG. The outliers are also still very good with at least 0.8 Jaccard index and 0.87 NDCG. This makes it

suitable to fully replace the float32 approach as it gives good memory savings, reliable results and is much faster on supported hardware.

Even better performs the mapped float searcher. On average, it gets close to perfect scores (Figure 5.3). The outliers are no worse than the float16 outliers and the lower percentiles are above 0.96 for both metrics. This makes it the most accurate quantization tested.

The variants with PCA dimension reduction applied to the embeddings mostly perform worse than methods with the similar memory usage and search speed. Int8 is more accurate than PCA2 and only uses half the memory, while PCA2 only got a slight speed advantage. When comparing it to the binary search method, we see, that PCA8 performs worse while taking 3 times longer and using quadruple the memory. The bad performance when using PCA for embeddings is also mentioned here. [16]

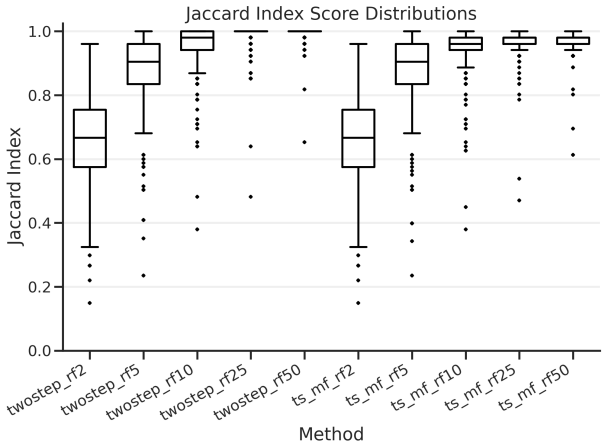

Figure 5.7: Jaccard index of two-step searchers

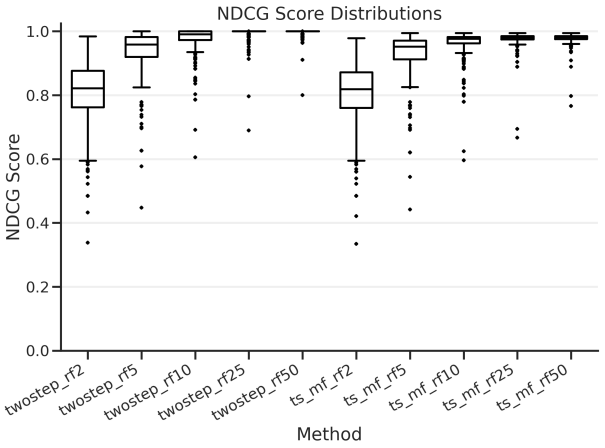

Figure 5.8: NDCG score of two-step searchers

Both two-step methods (binary+float and binary+mapped float) perform really well. Especially with rescoring factors of 10 and higher. For a rescoring factor of 10 they have a first quartile NDCG score of 0.96 and 0.97 respectively. The outliers from the binary searcher still exists and only get slightly better as the rescoring factor increases. With a rescoring factor of 25 or higher the binary+float searcher mostly gets perfect scores, except for the outliers. The score for the binary+mapped float searcher is capped by the mapped float searcher which is still very high.

5.3.3 Accuracy vs Rescoring Factor

As mentioned in the previous section the search time increases when increasing the rescoring factor. The increase in time is linear as seen in Figure 5.9 and Figure 5.10. The search time overall stays very low as even with high rescore factors the number of prefiltered embeddings is very small compared to the total number of embeddings. At a rescoring factor of 25 the score is very close to the full search equivalent of the second method. The only point in increasing it further is to reduce outliers.

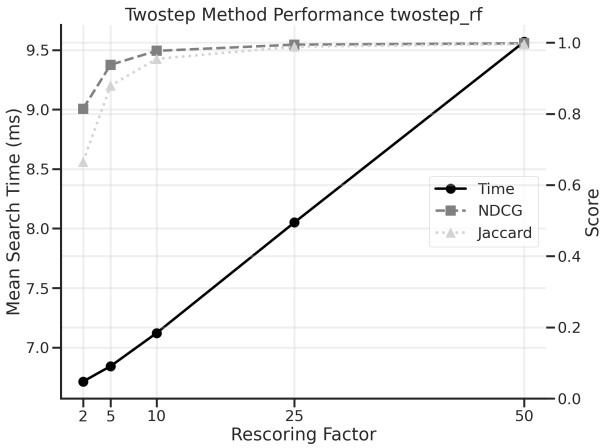

Figure 5.9: Time, NDCG, Jaccard vs Rescoring Factor Binary+Float

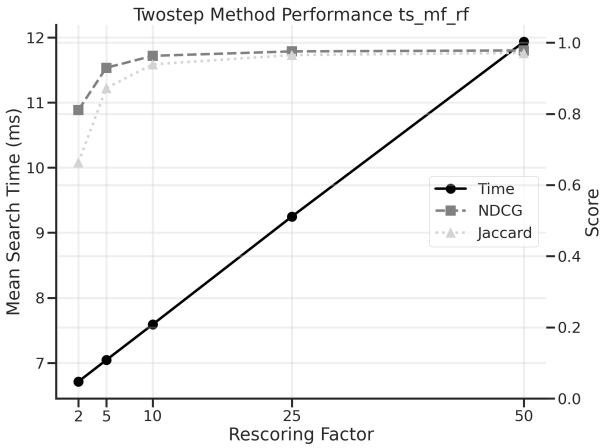

Figure 5.10: Time, NDCG, Jaccard vs Rescoring Factor Binary+Mapped Float

5.3.4 Comparing Benchmark Results from Different Models

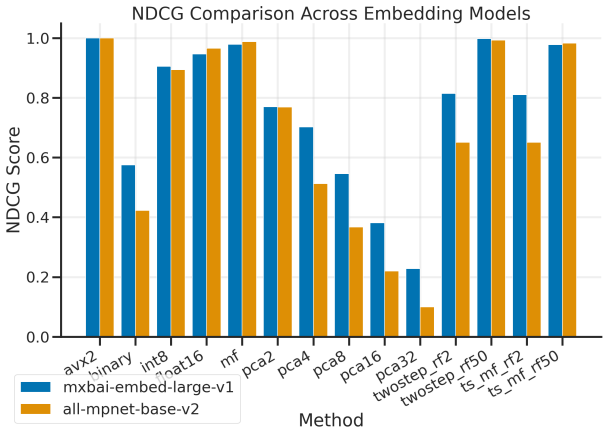

Figure 5.11: NDCG score comparison between different models

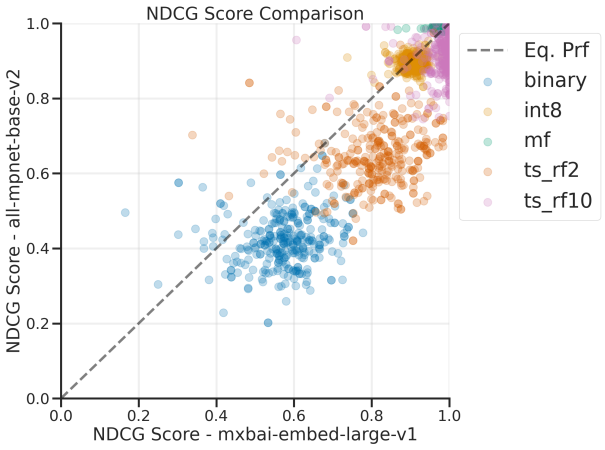

Figure 5.12: NDCG score comparison between different models as scatter plot

As mentioned earlier the model used is an angle optimized model, which should enable the binary searcher to get better results. [17, 15] To verify the previous results, the scores of the searchers when using the `mixedbread-ai/mxbai-embed-large-v1` model will be compared against the scores when using the `sentence-transformers/all-mpnet-base-v2` text embedding model. The same queries will be used.

Looking at the bar plot Figure 5.11 we see that the binary searcher indeed performs better with the angle optimized model. So does the two-step search as it's using the binary searcher in the first step. The PCA reduced searcher also perform better. This indicates, that the PCA algorithm is better at removing redundancies and grouping correlating dimensions for the embedding vectors created by the optimized model. There is no significant difference for the other

searchers.

5.3.5 Memory Usage Analysis

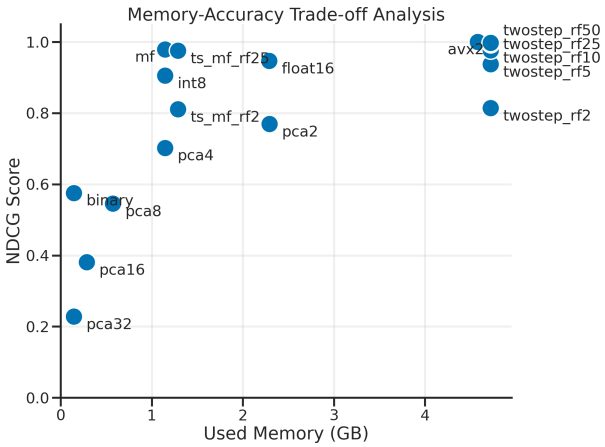

Figure 5.13: NDCG Score vs Memory Usage

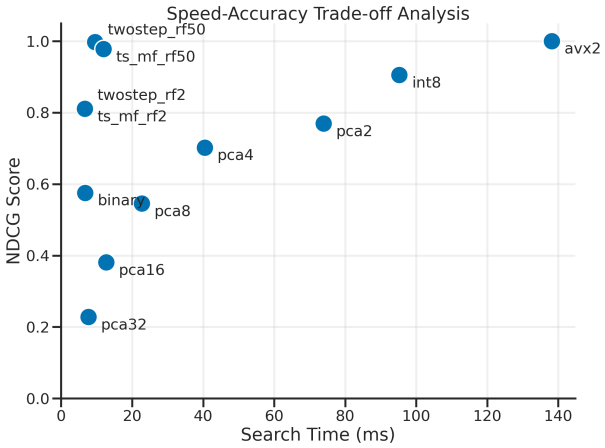

Figure 5.14: NDCG Score vs Retrieval Time

The binary+float searcher method uses the most memory, as it has to keep the binary and float32 embeddings in memory which totals to 33/32 times more memory usage compared to the baseline. Exactly the same memory usage as the baseline has the avx2 search, as it only optimizes the computation. And obviously the float16 variant uses half the memory of baseline.

The mapped float uses 1/4 of the baseline memory which makes it the best performing model in that memory range when factoring in its very high accuracy scores. This method is also one of the few not limited by memory bandwidth but by the L1 cache latency. Which makes it able to profit from multithreading on multicore CPUs as each core has its own L1 cache.

The binary+mapped float variant uses slightly more memory as it also has to load the binary embeddings, which, in return, makes it very fast. On its own the binary searcher uses the least memory except the PCA32 variant which has bad accuracy. The other PCA variants are also outperformed in search time and accuracy by other methods in the same memory range.

5.3.6 Search Query Performance

In the Table 5.4 are some queries listed that were also used in the benchmarks. The table lists the corresponding scores for some search implementations. 4 categories of queries were selected: Good and bad performing queries, queries with high score variance and randomly selected queries. The worst performing query is just a ":)" it's an unconventional query that has no point. Even a human would have problems interpreting a distinct semantic meaning into it. But still, the int8, mapped float and two-step method perform quiet well. The binary and PCA4 score is with around 0.2 really low. But this explains the big outliers in the box plots 5.5 5.6 5.7 5.8. Which makes the outliers less bad, because you can't expect good results from a query like this. "paper", a really generic query that also has quiet low scores, especially for the

Query	NDCG score					type
	binary	int8	mf	pca4	ts_rf10	
:)	0.250	0.799	0.867	0.202	0.691	bad
paper	0.302	0.835	0.958	0.366	0.785	bad
maps of places that don't exist yet	0.392	0.846	0.941	0.546	0.844	bad
history of the number zero	0.464	0.907	0.971	0.564	0.918	bad
cars	0.489	0.881	0.957	0.623	0.966	var
evolution of human color vision and its gene...	0.584	0.921	0.984	0.691	1.000	var
sun	0.617	0.892	0.982	0.575	1.000	var
archaeological evidence of prehistoric human...	0.570	0.878	0.990	0.720	1.000	rand
PLEASE HELP ME FIND INF[...] ABOUT DOLPHINS!!!	0.598	0.914	0.978	0.791	0.976	rand
Roman legion size	0.604	0.937	0.986	0.786	1.000	good
development of early mechanical calculators in...	0.575	0.938	0.990	0.770	1.000	good
pizza pizza pizza pizza pizza pizza pizza	0.778	0.971	0.995	0.839	1.000	good

Table 5.4: Accuracy for different search queries

binary and PCA4 searcher. The other implementations perform good enough with this query. An example for more specific queries would be "Roman legion size". Here its very apparent what the user would want as an answer. All searchers have good score with this query.

In conclusion, more specific queries, which also implies more semantic meaning, seem to perform better. It also explains the outliers of the lower accuracy models and gives more confidence in rating them as usable.

5.4 Summary

Most results are roughly in line with the expectations. The accuracy of the mapped float searcher is very surprising. The fact it performs better than float16 with only 8 bits was unexpected. But the reason for this could be the accuracy loss during float addition. During the calculation of the dot product the products of the vector dimensions are added to the dot product where each addition can generate accuracy loss. And for the float16 this running sum is also in float16 format, while the mapped float uses float32 for this, as its using float32 values during multiplication.

The slow speed of the int8 search can be explained by the instruction intensive splitting, moving and extending of vectors/values, which is needed to prevent overflow. Maybe quantization with a limit of 4 used bits would be better, as one could then simply multiply the int8 values without risking overflow.

PCA performed much worse than expected as even PCA2s accuracy is outperformed by simple int8 quantization.

Also, the binary and two-step methods are overall the most useful as they are very fast. For the two-step methods comes the addition of being very accurate. And two-step with the mapped float even hast a fairly low memory usage which makes it overall the best performers

when considering all 3 main metrics of memory usage, accuracy and speed.

6 Future Work

This chapter gives ideas on future research directions for vector similarity search optimization, based on the findings presented in this thesis.

6.1 Disk-Based Two-Step Search

The presented two-step implementations keep all embeddings of both searchers in memory, which limits its usage on devices with constrained memory. A promising extension would be disk based approach that work like this: The binary embeddings are stored in memory. This reduces the memory usage by 1/32 compared to full precision. The embeddings of the second searcher (full accuracy or mapped float for example) for rescoring stay on disk in a format that allows fast retrieval of embeddings. Then the binary search is used to get promising candidates. In the second step the high precision embeddings from these candidates are loaded from disk. For 10 documents with a rescoring factor of 25 and a disk access time of 1ms this would take $10 * 25 * 1ms = 250ms$. But this can potentially be way faster with an optimized I/O strategy. And most modern flash storage has a latency of less than 1ms. Another challenge is designing an efficient disk storage format that allows fast random access to the specific embeddings. Additionally, frequently accessed embeddings could be cached.

6.2 Hardware Specific Optimizations

6.2.1 Float16 Hardware support

In the future more processors should have native support for float16 operations. AVX512 can, for example, load 32 float16 values into one 512 bit register. Float16 had very good accuracy and should be much faster than float32 on supported hardware. Hardware accelerated float16 should be used when available. Many modern GPUs also support float16 and advertise double throughput compared to full precision.

6.2.2 ARM NEON support

This thesis focused on AVX2 optimizations. But many mobile and embedded devices use ARM processors. Future work could port the SIMD optimizations to NEON instructions, as most AVX2 instructions have a NEON equivalent counterpart. This would also allow to compare the optimizations between ARM and x86.

6.3 Additional Optimization Approaches

Some promising optimizations warrant more investigation:

6.3.1 Hybrid/Adaptive Quantization Methods

- Quantization that quantizes with different precision levels for different parts of the vector.
- Compress parts of the vector that has repeating or similar information. This probably becomes especially effective when have already low precision elements. At 8 bit for example, there are way more elements in a vector than distinct values they can have (256 values and 1024 elements).

Another better approach to reduce the embedding size than PCA dimension reduction is the usage Learning-to-Hash algorithms as they outperform PCA for embeddings. [16]

6.3.2 Memory Access Optimization

Different layouts of the embedding matrix to optimize bandwidth could be investigated. Also, the reasons for strided access being faster are not clear and should be looked into.

6.4 Research directions

6.4.1 Domain Specific Optimization

As this thesis only worked with text embeddings, future research could explore how different domains (text, image, audio, ...) affect optimization effectiveness. Especially with techniques like the mapped float embeddings. Additionally, the impact of different embedding models could be analyzed. With the results from this one could investigate domain specific optimizations.

6.4.2 Theoretical Analysis

The relationship between binary quantization or PCA with angular similarity needs further investigation. As both have a big accuracy difference between traditional embedding models and the angle optimized model.

6.5 Real World Applications

Future work could also focus on these real world applications:

- Implementation in production search systems.
- Implementation in existing vector databases.

- Creating more standardized benchmarks for vector search.
- Analyze optimization strategies for different use cases.

All these directions would build on the foundation created by this thesis and would advance the efficient vector similarity search on resource constrained devices.

7 Conclusion

This thesis investigated various optimization techniques for vector similarity search with focus on increasing performance, keeping good accuracy and lowering the memory required to enable better search on resource constrained devices. With extensive experimentation followed by good analysis of the recorded results, it has made multiple contributions towards further optimizations for vector similarity search.

7.1 Summary of findings

The baseline float32 implementation was the metric against which the optimizations were compared. The findings from experimenting and analyzing different optimizations are as follows:

7.1.1 Quantization Methods

Overall the quantization based methods performed very good. Int8 quantization achieved a 4x reduction in memory usage while main around 90% accuracy. The binary quantization performed even better: With 32x memory reduction while maintaining ∼58% accuracy. This performance dropped to slightly over 40% with a non-angle optimized model. It is about 20 times faster than the highly optimized float32 implementation with AVX2 and around 160 times faster than the naive float32 implementation. The biggest improvement came from switching the native 64bit popcount with a lookup table based popcount using AVX2. Another interesting result is from the mapped float approach, which combines quantization, adapting to dataset and value mapping. It achieved 97% accuracy while quartering the memory usage. It was the highest accuracy non-float32 searcher, even outperforming float16. But looking at Figure 4.1 its likely that the Gauss distribution actually just matches the distribution of values contained in the embeddings very closely. That might also be the reason why the Quantization mapping in Figure 4.2 might be so linear. But to confirm this suspicion one should compare int8 Quantization where then int8 minimum value maps to the minimum values contained in the embeddings and int8 maximum value maps to the max value contained in the embeddings. The values in between should just be scaled linearly.

7.1.2 SIMD Optimization

The AVX2-optimized float32 implementation achieved 8x speedup over the baseline. At the end the memory bandwidth became the bottleneck. The method of comparing the query vector to

multiple embeddings at once was highly effective as it greatly reduces the number of loads from memory/cache. This also increased the memory bandwidth. The int8 implementation gained less performance from the AVX2 implementation as it has to move the data around quiet a bit before being able to multiply the values. Additionally, it can only multiply 16 int8 values at a time. But there are other instructions, I didn't discover in time, like *PMADDUBSW* which multiples 32 int8 and adds the int16 results of neighboring numbers together.

7.1.3 Two-Step Approach

Binary pre-filtering followed by high precision rescoring is highly effective. With a rescoring factor of 10 it achieved a NDCG score of 97.7% with float32 rescoring and 96.3% with mapped float rescoring. Both just very slightly increased the search time. This demonstrated, that this approach is highly effective. It's very close to binary speed and very close to float32 accuracy. With the mapped float as second step it additionally only uses 9/32 of the baseline memory.

7.1.4 Dimension Reduction

The dimension reduction using the PCA technique had worse accuracy than quantization methods with comparable memory savings. This makes it not useful for accelerating search and saving memory. This is also mentioned in this paper: [16] Other dimension reduction- or vector compression methods should be investigated.

7.2 Key Insights

7.2.1 Memory Access Patterns

Memory access pattern showed to be very important as most implementations were bottle necked either by memory bandwidth or latency. Optimizing towards memory bandwidth/latency is very important as the actual calculations are very fast. Strided access increased bandwidth even though its common knowledge that strided access is usually slower than purely sequential access. The systems' memory pages, memory ranks or memory interleaving are a suspected reason for this behavior. Prefetching, either manually or automatically by the CPU, is also very important and effective as it's able to hide the extremely high system memory latency.

7.2.2 Quantization Trade-Offs

The different quantization methods showed distinct trade-offs between accuracy and performance. Furthermore, the embedding model can influence the performance of the quantization methods. The mapped float approach demonstrated, that "intelligent" quantization can preserve more semantic meaning than linear quantization with the same memory usage.

7.2.3 Hardware Utilization

SIMD instructions can provide a substantial performance improvement. With this memory bandwidth becomes a limiting factor fast. Even though the implementations don't use multi-threading at all.

7.3 Limitations and Future Work

All methods implemented keep the embeddings in memory at all times. Which may not be needed especially for the two-step approach. Then it only has to load the embeddings from the documents needed for rescoring form disk. Additionally, the SIMD optimizations are specific to the x86 architecture but should be able to be transferred to other architectures using similar instructions. The findings should also be verified on different architectures.

7.4 Final Remarks

It was demonstrated, that significant performance improvements in vector similarity search can be achieved by optimizing hardware usage and using quantization. Especially combining smart quantization, SIMD and the two-step approach provide close to full accuracy, while being 100 times faster than baseline and more than 10 times faster than the optimized full accuracy search. All while using less memory. Which can be very beneficial for resource constrained devices.

A Appendix

A.1 Isolated Testing of Memory Bandwidth

Algorithm A.1.1: Loop with Strided Memory Access

```
// Uses 8 strides spaced out 4 KB
for (int iter = 0; iter < NUM_ITERATIONS; iter++) {
  for (size_t i = 0; i < VEC_SIZE - TOTAL_STRIDE_DIST + 1;
       i += TOTAL_STRIDE_DIST) {
    #pragma GCC unroll 1 // prevent loop unrolling
    for (size_t j = i; j < STRIDE_LENGTH + i; j += 8) {
      asm volatile(                          //
          "vmovaps (%0),       %%ymm0\n\t" //
          "vmovaps 0x1000(%0), %%ymm1\n\t" // offset 1 * 4096 bytes
          "vmovaps 0x2000(%0), %%ymm2\n\t" // offset 2 * 4096 bytes
          "vmovaps 0x3000(%0), %%ymm3\n\t" // offset 3 * 4096 bytes
          "vmovaps 0x4000(%0), %%ymm4\n\t" // offset 4 * 4096 bytes
          "vmovaps 0x5000(%0), %%ymm5\n\t" // offset 5 * 4096 bytes
          "vmovaps 0x6000(%0), %%ymm6\n\t" // offset 6 * 4096 bytes
          "vmovaps 0x7000(%0), %%ymm7"     // offset 7 * 4096 bytes
          :
          : "r"(&array[j])                   //
          : "ymm0", "ymm1", "ymm2", "ymm3", //
            "ymm4", "ymm5", "ymm6", "ymm7");
    }
  }
} // Uses sequential memory access
for (int iter = 0; iter < NUM_ITERATIONS; iter++) {
  #pragma GCC unroll 1 // prevent loop unrolling
  for (size_t i = 0; i < VEC_SIZE - 63; i += 64) {
    asm volatile(                         //
        "vmovaps (%0),     %%ymm0\n\t" //
        "vmovaps 0x20(%0), %%ymm1\n\t" // offset 32 bytes <=> 1 AVX2 vector
        "vmovaps 0x40(%0), %%ymm2\n\t" // offset 64 bytes
        "vmovaps 0x60(%0), %%ymm3\n\t" // ...
        "vmovaps 0x80(%0), %%ymm4\n\t"
        "vmovaps 0xa0(%0), %%ymm5\n\t"
        "vmovaps 0xc0(%0), %%ymm6\n\t"
        "vmovaps 0xe0(%0), %%ymm7"
        :
        : "r"(&array[i])                   //
        : "ymm0", "ymm1", "ymm2", "ymm3", //
          "ymm4", "ymm5", "ymm6", "ymm7");
  }
}
```

A C program to test the bandwidth of strided against non-strided memory access was created to isolate ”the problem” from other variables. In Algorithm A.1.1 are the two loops used in the methods testing the memory read bandwidth. The loops iterate 16 times over a 12 GB array. These are the results from two different machines:

```
Results from Machine used in previous experiments:
SIMD Read Bandwidth (Sequential)           : 29.46 GB/s
SIMD Read Bandwidth with stride length 4096: 36.71 GB/s
AMD 4800U 8C/16T, 16 GB LPDDR4 2666MHz:
SIMD Read Bandwidth (Sequential)           : 16.39 GB/s
SIMD Read Bandwidth with stride length 4096: 22.74 GB/s
```

The results line up very closely with the bandwidth results obtained from the vector similarity search program. But I have no Intel PC at my disposal to test this on a different architecture.

A.2 Trying out the Similarity Search Implementation

Go to `https://github.com/thubn/simsearch` for more information. In the main README go to the ”Embedding Search Server” link in the index.

A.3 Additional Plots

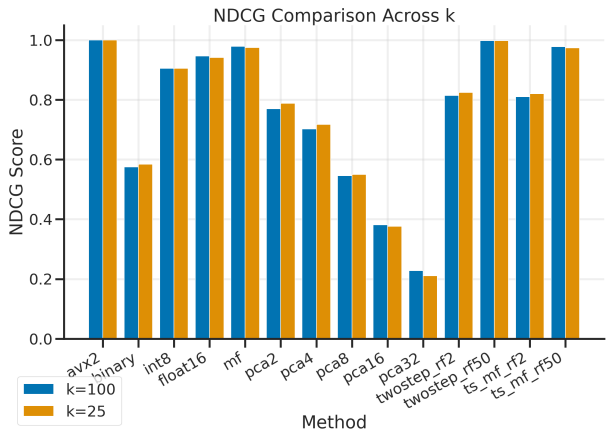

Figure A.1: Comparing the retrieval accuracy across different k values

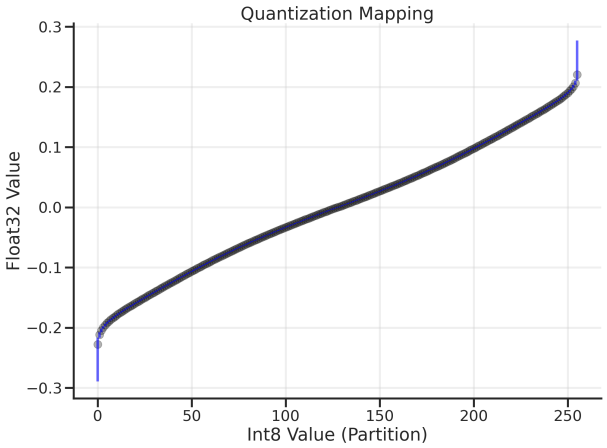

Figure A.2: Mapped Float Quantization mapping for the sentence-transformers/all-mpnet-base-v2 model

The quantization mapping seen in Figure A.2 of the embeddings from model `sentence-transformers/all-mpnet-base-v2` doesnt have a gap like the angle optimized model.

Varying k as seen in Figure A.1 doesn't create a big difference in accuracy.
The memory bandwidth in Figure A.4 when using the `sentence-transformers/all-mpnet-base-v2` model is even higher. The reason might be that higher striding distance is even better as this model creates embeddings of size 3072 bytes, so it has to use a striding distance

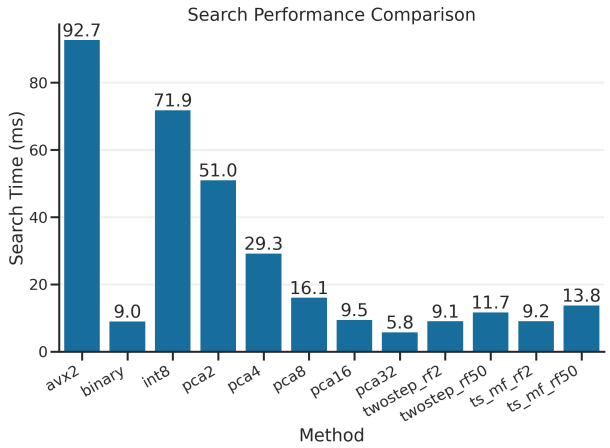

Figure A.3: Search Speed vec_dim=768 model: sentence-transformers/all-mpnet-base-v2

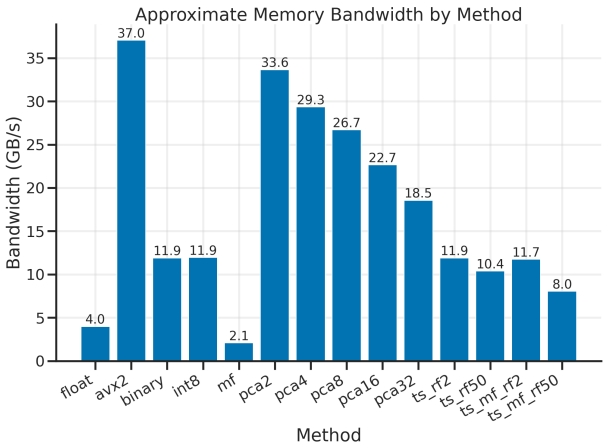

Figure A.4: Memory bandwidth vec_dim=768 model: sentence-transformers/all-mpnet-base-v2

of 12288 bytes as it's the first multiple that's divisible by 4096. Binary performs worse because it can't use the optimized unrolled loop as it only uses 3 avx2 vectors for 768 dimensions.

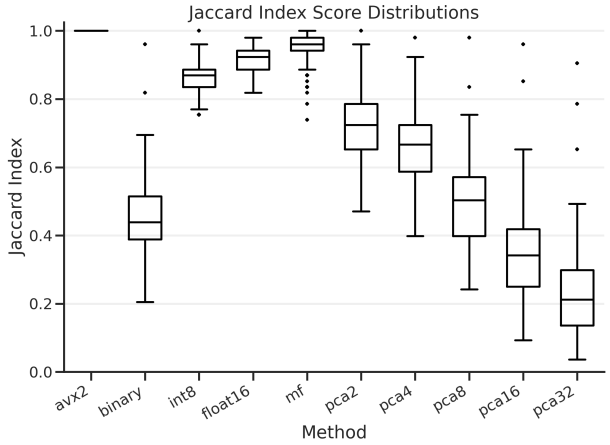

Figure A.5: Jaccard index of searchers with randomly generated queries

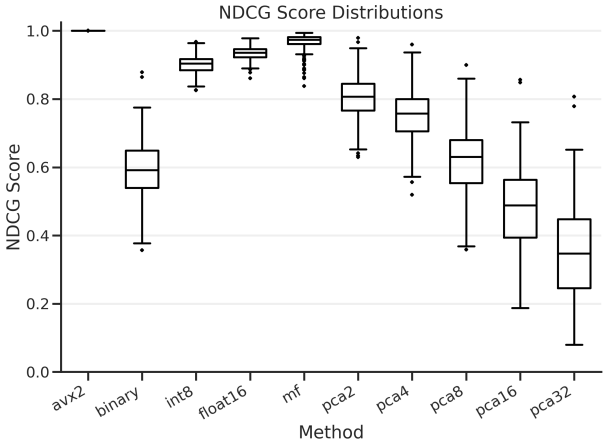

Figure A.6: NDCG score of searchers with randomly generated queries

Looking at Figure A.5 A.6 A.7 A.8: The searchers perform worse in accuracy across the board. But this isn't a drawback as this is not a realistic use case for embedding search.

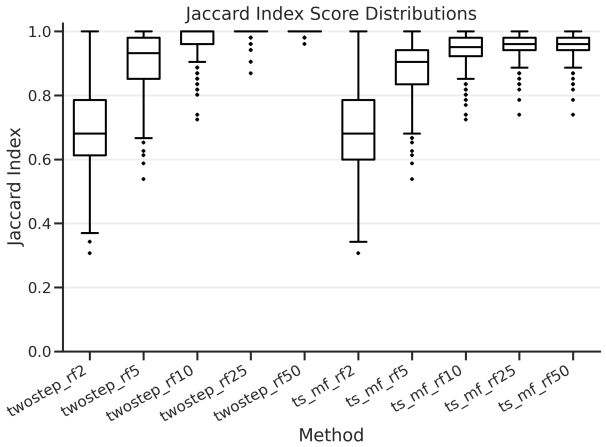

Figure A.7: Jaccard index of two-step searchers with randomly generated queries

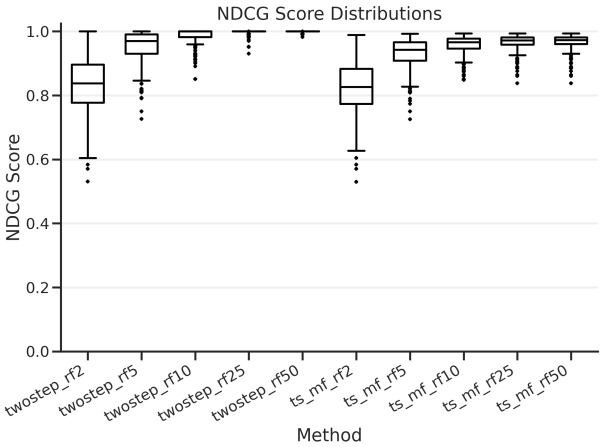

Figure A.8: NDCG score of two-step searchers with randomly generated queries

Method	Mean (ms)	Std Time	NDCG	Jaccard	Overlap	M (GB)	BW (GB/s)
float	1105.232	0.948	1.000	1.000	100.000	4.578	4.142
avx2	135.972	1.105	1.000	1.000	100.000	4.578	33.666
binary	6.662	0.048	0.589	0.451	61.496	0.143	21.474
int8	95.215	0.055	0.901	0.867	92.836	1.144	12.019
float16	185.023	6.713	0.934	0.914	95.500	2.289	12.370
mf	540.883	0.325	0.966	0.956	97.692	1.144	2.116
pca2	71.421	0.607	0.803	0.724	83.700	2.291	32.047
pca4	38.725	0.314	0.750	0.653	78.544	1.145	29.552
pca8	22.146	0.217	0.616	0.489	64.820	0.573	25.838
pca16	12.469	0.201	0.482	0.346	50.140	0.286	22.944
pca32	7.566	0.129	0.351	0.228	35.720	0.143	18.908
twos_rf2	6.583	0.065	0.830	0.689	80.772	4.721	21.846
twos_rf5	6.727	0.063	0.952	0.904	94.660	4.721	21.550
twos_rf10	7.006	0.067	0.985	0.969	98.344	4.721	20.963
twos_rf25	7.952	0.089	0.998	0.995	99.748	4.721	19.188
twos_rf50	9.486	0.124	1.000	0.999	99.948	4.721	17.091
ts_mf_rf2	6.600	0.054	0.821	0.684	80.452	1.287	21.702
ts_mf_rf5	6.930	0.053	0.930	0.880	93.384	1.287	20.711
ts_mf_rf10	7.476	0.062	0.956	0.934	96.516	1.287	19.262
ts_mf_rf25	9.131	0.086	0.965	0.953	97.548	1.287	15.927
ts_mf_rf50	11.820	0.136	0.966	0.955	97.668	1.287	12.506

Table A.1: Comparison of methods for mixedbread-ai/mxbai-embed-large-v1

Method	Mean (ms)	Std Time	NDCG	Jaccard	Overlap	M (GB)	BW (GB/s)
float	862.587	0.625	1.000	1.000	100.000	3.433	3.980
avx2	91.731	1.469	1.000	1.000	100.000	3.433	37.427
binary	9.022	0.063	0.605	0.475	63.664	0.107	11.891
int8	72.253	0.094	0.930	0.910	95.248	0.858	11.879
float16	142.844	18.197	0.967	0.959	97.880	1.717	12.017
mf	408.601	0.357	0.992	0.989	99.468	0.858	2.101
pca2	51.656	0.481	0.901	0.864	92.624	1.718	33.231
pca4	29.478	0.331	0.820	0.754	85.648	0.859	29.117
pca8	16.082	0.156	0.626	0.505	66.076	0.429	26.685
pca16	9.487	0.141	0.441	0.309	45.680	0.215	22.618
pca32	5.767	0.112	0.285	0.174	28.260	0.107	18.602
twos_rf2	9.076	0.031	0.848	0.717	82.772	3.541	11.884
twos_rf5	9.241	0.031	0.962	0.924	95.828	3.541	11.764
twos_rf10	9.521	0.034	0.991	0.981	99.008	3.541	11.569
twos_rf25	10.404	0.060	0.999	0.998	99.900	3.541	11.000
twos_rf50	11.816	0.101	1.000	1.000	99.976	3.541	10.291
ts_mf_rf2	9.157	0.065	0.847	0.717	82.760	0.966	11.732
ts_mf_rf5	9.465	0.024	0.959	0.921	95.684	0.966	11.373
ts_mf_rf10	9.965	0.034	0.985	0.974	98.632	0.966	10.839
ts_mf_rf25	11.479	0.063	0.991	0.988	99.384	0.966	9.502
ts_mf_rf50	13.903	0.113	0.992	0.989	99.444	0.966	7.974

Table A.2: Comparison of methods for sentence-transformers/all-mpnet-base-v2

Bibliography

[1] Brian McFee, Luke Barrington, and Gert Lanckriet. "Learning Content Similarity for Music Recommendation". In: *IEEE Transactions on Audio, Speech, and Language Processing* 20.8 (2012), pp. 2207–2218. DOI: 10.1109/TASL.2012.2199109.

[2] Makbule Gulcin Ozsoy. *From Word Embeddings to Item Recommendation*. 2016. arXiv: 1601.01356 [cs.LG]. URL: https://arxiv.org/abs/1601.01356.

[3] Rahul Shrivastava and Dilip Singh Sisodia. "Product Recommendations Using Textual Similarity Based Learning Models". In: *2019 International Conference on Computer Communication and Informatics (ICCCI)*. 2019, pp. 1–7. DOI: 10.1109/ICCCI.2019.8821893.

[4] Tomas Mikolov, Quoc V. Le, and Ilya Sutskever. *Exploiting Similarities among Languages for Machine Translation*. 2013. arXiv: 1309.4168 [cs.CL]. URL: https://arxiv.org/abs/1309.4168.

[5] Niklas Muennighoff et al. *MTEB: Massive Text Embedding Benchmark*. 2023. arXiv: 2210.07316 [cs.CL]. URL: https://arxiv.org/abs/2210.07316.

[6] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[7] Yining Karl Li. "Comparing SIMD on x86-64 and arm64". Article on authors personal blog. 2021. URL: https://blog.yiningkarlli.com/2021/09/neon-vs-sse.html.

[8] Gaurav Mitra et al. "Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms". In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. 2013, pp. 1107–1116. DOI: 10.1109/IPDPSW.2013.207.

[9] Alireza Khadem et al. *Vector-Processing for Mobile Devices: Benchmark and Analysis*. 2023. arXiv: 2309.02680 [cs.AR]. URL: https://arxiv.org/abs/2309.02680.

[10] Ulrich Drepper. *What Every Programmer Should Know About Memory*. Nov. 2007. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[11] Daniel Molka et al. "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture". In: *2015 44th International Conference on Parallel Processing*. 2015, pp. 739–748. DOI: 10.1109/ICPP.2015.83.

[12] Kalervo Järvelin and Jaana Kekäläinen. “Cumulated gain-based evaluation of IR techniques”. In: *ACM Trans. Inf. Syst.* 20.4 (Oct. 2002), pp. 422–446. ISSN: 1046-8188. DOI: 10.1145/582415.582418. URL: https://doi.org/10.1145/582415.582418.

[13] Christoph Peters. “fma: A faster, more accurate instruction”. Article on authors personal blog. 2021. URL: https://momentsingraphics.de/FMA.html.

[14] Wojciech Muła, Nathan Kurz, and Daniel Lemire. “Faster Population Counts Using AVX2 Instructions”. In: *The Computer Journal* 61.1 (May 2017), pp. 111–120. ISSN: 1460-2067. DOI: 10.1093/comjnl/bxx046. URL: http://dx.doi.org/10.1093/comjnl/bxx046.

[15] Xianming Li and Jing Li. *AnglE-optimized Text Embeddings*. 2024. arXiv: 2309.12871 [cs.CL]. URL: https://arxiv.org/abs/2309.12871.

[16] Nandan Thakur, Nils Reimers, and Jimmy Lin. *Injecting Domain Adaptation with Learning-to-hash for Effective and Efficient Zero-shot Dense Retrieval*. 2023. arXiv: 2205.11498 [cs.IR]. URL: https://arxiv.org/abs/2205.11498.

[17] Sean Lee et al. *Open Source Strikes Bread - New Fluffy Embeddings Model*. 2024. URL: https://www.mixedbread.ai/blog/mxbai-embed-large-v1.

Eidesstattliche Versicherung

(Affidavit)

Simon, Thorben

Name, Vorname
(surname, first name)

187069

Matrikelnummer
(student ID number)

☒ Bachelorarbeit
(Bachelor's thesis)

☐ Masterarbeit
(Master's thesis)

Titel
(Title)

Vector Similarity Search Optimization

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Dortmund, 16.01.2024

Ort, Datum
(place, date)

T. Simon

Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Dortmund, 16.01.2024

Ort, Datum
(place, date)

T. Simon

Unterschrift
(signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**