FAST SINGLE-CORE K-NEAREST NEIGHBOR GRAPH COMPUTATION

Dan Kluser, Jonas Bokstaller, Samuel Rutz, Tobias Buner

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Fast and reliable K-Nearest Neighbor Graph algorithms are more important than ever due to their widespread use in many data processing techniques. This paper presents a runtime optimized C implementation of the heuristic "NN-Descent" algorithm by Wei Dong et al. [1] for the l2-distance metric. Various implementation optimizations are explained which improve performance for low-dimensional as well as high dimensional datasets.
Optimizations to speed up the selection of which datapoint pairs to evaluate the distance for are primarily impactful for low-dimensional datasets. A heuristic which exploits the iterative nature of NN-Descent to reorder data in memory is presented which enables better use of locality and thereby improves the runtime. The restriction to the l2-distance metric allows for the use of blocked distance evaluations which significantly increase performance for high dimensional datasets.
In combination the optimizations yield an implementation which significantly outperforms a widely used implementation of NN-Descent on all considered datasets. For instance, the runtime on the popular MNIST handwritten digits dataset is halved.

1. INTRODUCTION

K-nearest neighbor graphs (K-NNG), which contain the identities of the closest datapoints for each datum, are fundamental to many data science and machine learning techniques. The rapid growth of attainable data has therefore increased the need for practical K-NNG algorithms and efficient implementations thereof.
This paper presents an optimized implementation of the NN-Descent algorithm by Wei Dong et al. [1]. NN-Descent is an iterative, randomized heuristic which improves a random guess of the K-NNG.
A particularly popular implementation of NN-Descent is PyNNDescent [2] which is implemented in Python and uses the Numba JIT copiler [3] to achieve sufficient runtime performance. Particular strengths of PyNNDescent are its ease of use and support of custom distance metrics. PyNNDescent is used in the sci-kit learn compatible implementation of UMAP [4] [5]. UMAP is currently gaining popularity as an alternative dimensionality reduction technique to t-SNE due to favorable embedding properties and because it is faster than current implementations of t-SNE [6].
The use of NN-Descent in practice makes performance optimized implementations especially desirable.
A major challenge in optimizing the performance of NN-Descent implementations is the irregular memory access pattern stemming from unordered input data together with the neighbor-of-neighbor heuristic explained in Section 2.
This paper presents a single-core implementation which is limited to the l2-distance metric at the benefit of a vast runtime reduction when compared to PyNNDescent.
Optimization techniques are discussed which performance for both low or high-dimensional input respectively. In particular a novel heuristic is introduced which increases locality by improving the otherwise irregular memory access pattern.
The overall performance advantage over PyNNDescent motivates the use of specialized implementations of NN-Descent for commonly used distance metrics and use cases.

Algorithms for K-NNG computation are still an active area of research due to their enormous importance. Alternative approaches include "Goldfinger" [7] which is based on bit-comparisons of hashes as well as methods building upon the same ideas as NN-Descent [8].

2. K-NEAREST NEIGHBOR GRAPH AND NN-DESCENT

In this section we present an overview of the K-Nearest Neighbor Graph (K-NNG) problem and the NN-Descent algorithm optimized in our implementation.

A K-NNG is a directed simple graph $G = (V, E)$ such that we have a directed edge $(u, v) \in E$ iff v is one of the k-nearest neighbors of u according to a given distance metric. Let V be the set of datapoints in $\mathbb{R}^d$ space where $|V| = n$.

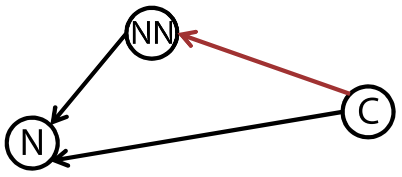

Fig. 1. A neighbor of a neighbor is likely to be also a neighbor [1]. In the current estimate of the K-NNG, N is considered a neighbor of NN and also of C. NN-Descent evaluates the distance between NN and C updates the current approximation of the K-NNG accordingly by replacing edge (C,N) with (C,NN).

The dimensionality of all datapoints is denoted by d. The k nodes adjacent to some node u in G, denoted as $\text{adj}_G(u)$, are thus the k nearest neighbors according to the given metric.

One trivial way of computing the KNN-Graph is to compute all $\frac{n(n-1)}{2}$ mutual distances and selecting the best k points. This is not computationally viable for most datasets of practical interest, especially in data science applications due to the $O(n^2)$ asymptotic number of distance evaluations.
NN-Descent by Wei Dong et al.[1] is a randomized, iterative heuristic which computes an approximation of the K-NNG. The NN-Descent algorithm requires far fewer distance evaluations (empirical cost $O(n^{1.14})$ [1]) at the expense of the quality of the resulting KNN-Graph.

The key insight used is that "a neighbor of a neighbor is also likely to be a neighbor"[1] (Figure 1).

Using this assumption allows NN-Descent to iteratively improve the KNN-Graph estimate by evaluating distances primarily for promising pairs of vertices.
NN-Descent begins with a random initialization of G, so that for every vertex the set of k nearest neighbors are uniformly sampled from V. Multiple iterations of the NN-Descent algorithm are then executed. Every iteration improves the K-NNG estimate by performing the following two steps for every vertex in V:

1. Select candidates: find a suitable set S with vertices that are neighbors-of-neighbors

2. Calculate and update: calculate all pairwise distances in S and update G if closer vertices were found.

We will refer to the first step as *Selection step* and the second one as *Calculation step*. Iterations are performed until the number of changes to G falls below a specified threshold. In this case the estimated K-NNG is a reasonable approximation of the true K-NNG.
Section 3.1 contains more details on the selection of candidates. For more details on the algorithm itself refer to [1].
As the NN-Descent algorithm is a heuristic it is necessary to validate both the quality of the returned K-NNG as well as the computational cost. Recall is used to measure how close the K-NNG approximation is to the true K-NNG. Our implementation achieved a recall of over 99% on all examined datasets. Multiple parameters could if desired be altered to change the runtime-quality trade-off.
To calculate the number of floating point operations, the number of distance evaluations is counted. This enables performance comparisons between different code versions which have small but varying amounts of additional floating point comparisons. Together with the dimensionality of the datapoints the number of operations can be computed. For each l2-distance evaluation d subtractions, d multiplications, and $d-1$ additions are performed.

3. OPTIMIZATION OF NN-DESCENT IMPLEMENTATION

Numerous possible optimizations were implemented and evaluated to yield a fast single-core implementation of NN-Descent. Some approaches aim at making the selection step highly efficient while others aim to improve the locality or optimize the mutual distance calculations. The relative impact of these optimizations differs depending on parameters of the problem, such as the dimensionality d of the datapoints.

We started out with a C implementation[9] which adheres closely to the pseudo code found in [1] (NNDescent-Full). We then studied the most popular implementation of NN-Descent, PyNNDescent [2], which was introduced in Section 1. Although asymptotically the two implementations behave the same, the Python implementation was considerably faster than the straightforward C implementation. We identified the main differences and adopted the improvements.

3.1. Selection Step

An important improvement found in PyNNDescent is the selection step, which is performed in each iteration for every vertex (see Section 2). These improvements were then expanded upon for an additional speed-up.

In the selection step of every iteration we need to find the neighborhood $N(u)$ for every node u in the current KNN-graph approximation. The neighborhood $N(u)$ contains every node $v \in V$ for which $(u, v) \in E$, i.e. node v is one of the k-nearest neighbors according to the current approximation. More interestingly, it also contains every node w for which $(w, u) \in E$, so every node w which has node u as one of its k-nearest neighbors. In the pseudo code as presented [1] such nodes w are found by first inverting all the directed edges in the current KNN-graph G, resulting in a graph $G' = (V, E')$ for which $E' = \{(u, v)|(v, u) \in E\}$.

After the *reverse* step, we then proceed by setting $N(u) = \text{adj}_G(u) \cup \text{adj}_{G'}(u)$ for all u which we refer to as *union* step. Note that while $\text{adj}_G(u)$ is bounded in size by k, $\text{adj}_{G'}(u)$ can contain up to n elements, which requires the usage of a dynamically growing data structure.

Finally we *sample* the neighborhood $N(u)$ to contain $\rho \cdot k$ elements. This reduces the number of the pairwise distance calculations of the neighborhood $N(u)$ which is quadratic in $|N(u)|$. The whole process of finding a suitably sized neighborhood for every node can be regarded as a composition of three functions: *reverse, union* and *sample*. In the basic implementation the intermediate results are stored in memory and three full passes over the entire K-NNG are required.

PyNNDescent improves over this by only doing one pass over the data but introducing a heap in the sampling process. Instead of building each neighborhood $N(u)$ and then sampling from it, PyNNDescent does both simultaneously in one pass over the KNN-graph. We adopted this change. For each edge $r = (u, v)$ a weight r_e is drawn uniformly at random (u.a.r.) in $[0, 1]$. Both $N(u)$ and $N(v)$ are implemented as heaps and we insert the node v in $N(u)$ with weight r_e, we do the symmetric thing for $N(v)$. This corresponds to both the reverse and union step of the naïve selection implementation. Because the heaps are bounded in size, every neighborhood $N(v)$ ends up containing at most $\rho \cdot k$ elements. Selecting a subset of size $\rho \cdot k$ is equivalent to assigning a random weight u.a.r. to each element and selecting the $\rho \cdot k$ elements with the smallest weights. This gives a considerable speedup; on our synthetic dataset we observed a 16x runtime speedup (see Section 4.1).

Because the heaps incurred a lot of cache misses we further optimized the fused selection function described above in order to get rid of the heaps. Upon every update of the KNN-graph we keep track of how large the neighborhood of every node v is. Since when doing these updates we access the relevant data structures anyway, we do not incur any additional cache misses by these modifications. Knowing how large each neighborhood is allows us to simplify the sampling process: for every edge $e = (u, v)$ we insert v into $N(u)$ with probability $\frac{\rho \cdot k}{|N(u)|}$. In expectation this is equivalent to the previous sampling procedure, but it works without heaps. This gives a small speedup of around 1.12x.

The optimized one-pass sampling step reduced the number of memory accesses by eliminating the intermediate results of the sequentially applied functions (*reverse, union* and *sample*). Furthermore, we avoided the difficulty of controlling the (previously unbounded) size of the reverse graph G'. This is important since for many relevant input sizes the KNN-graph does not fit into the caches found in commodity hardware. Multiple passes over the KNN-graph then lead to many cache misses.

3.2. Greedy Reordering Heuristic

Datapoints which are close in the dataspace are frequently accessed together but the underlying data is not usually located closely together. This leads to a difficulty in exploiting spatial locality. Our roofline model analysis (cf. Section 4.2) indicates that our implementation is memory-bound for low-dimensional inputs. The major difficulty of exploiting locality in spite of the non-uniform memory access pattern is the primary issue to be solved. Introducing an assumption on the data space distribution of the input data allows the development of a heuristic approach to tackle this problem.

Without any assumptions about the input distribution, our access pattern is irregular and we cannot improve locality. This is due to the tight relationship of temporal locality of two nodes and their distance in data space. We proceeded by assuming our input is clustered, meaning for every node all its k nearest neighbors are within the same cluster (*clustered assumption*). This assumption will allow us to partly recover those clusters from an early approximation of the K-NNG. After reordering memory such that the clusters are close together, we proceed with the remaining iterations of NN-Descent. Experiments on a synthetic dataset and on real world data set are promising (consult section 4 for more details).

Recall that during the selection step we iterate over all edges $e \in E$ of the current KNN-graph approximation $G = (V, E)$. For one edge $e = (u, v)$ we will access both $\text{adj}_G(u)$ and $\text{adj}_G(v)$. Those two lists are likely to be in completely different locations in memory. Since the edge e is part of our current KNN-graph, u and v are likely close in data space according to the given metric. This is why, especially after the initial iteration when our KNN-graph approximation becomes more accurate, closeness in data-space and temporal locality in the access pattern are highly correlated. For the remainder of this section we will consider clustered inputs (*clustered assumption*).

Intuitively, after the first iteration a nodes nearest neighbor is likely to be in the same cluster. Recall that we start with a randomly initialized approximation, meaning every node has k u.a.r. chosen nearest neighbors. The probability that within those k nodes we do not have any node from the same cluster can be bounded from above by (for c clusters, and k neighbors):

$$\Pr[\text{all } k \text{ nodes not in cluster } i] \leq \left(\frac{c-1}{c}\right)^k$$

For a wide range of practical k and c this probability is sufficiently small. Armed with the intuition that the nearest neighbor of every node in our approximation is likely to be within the same cluster, we may now try to exploit that to reorder our memory. As a first step we want an algorithm that:

- may only use the existence of those clusters, the input is *not* ordered in any way revealing information about the structure of those clusters
- recovers most of the clusters. Moreover, it should output a permutation $\sigma : [n] \rightarrow [n]$ which we may use in the end to permute our data in memory all at once to bring the clusters together.
- makes at most one pass over the KNN-graph

The above requirements inform the design of our greedy clustering heuristic. In the pseudo code (Algorithm 1), the permutations σ and σ^{-1} are modeled as n-dimensional arrays. We initialize them with the identity function: for each $i \in [n]$ we have $\sigma(i) = i$. We proceed by looking at node $i = 0$. In each iteration of the outermost loop we would like to find a good candidate for the spot $i + 1$, meaning whichever node permutation σ maps onto $i + 1$, it should be close in data space to node i. To achieve that we now sort the adjacency list of i by distance, so $a_i[j]$ contains the identifier of the j'th closest node. Now we check whether the spot assigned to by $a_i[j]$ by permutation σ is smaller than our current position, if so, we assume that $a_i[j]$ already has a good spot where it is close to its (data-space) neighbors in memory space. If not we check whether $a_i[j]$ already occupies the spot we would like to have it at ($\sigma(a_i[j]) = i + 1$) - then we conclude the search and break out of the inner loop.

Otherwise if $\sigma(a_i[j]) > i + 1$, we would like to set σ such that $\sigma(a_i[j]) = i + 1$, such that the node $a_i[j]$ will occupy spot $i + 1$. Note that this is the desired outcome, the nodes on spots i and $i + 1$ are close together in data space. This specific sequence of two swaps in the permutation σ and its inverse turn out to give the desired result. By creating and updating both the permutation and its inverse at the same time, we save ourselves a costly inversions of the permutation at several steps. This way we can satisfy the second requirement of only doing one pass through the KNN-graph.

The algorithm then returns a permutation σ. We proceed by permuting all of our data in memory using that permutation. Afterwards we continue with the remaining iterations of NN-Descent using the permuted memory layout. The copying itself is done all at once using σ.

When the clustered assumption is given, it is intuitive that our heuristic will succeed in clustering most of the data. Consult section 4 for an experimental evaluation on a synthetic data set. More surprisingly we have even seen a small speedup on real world datasets where the clustered assumption does not hold.

3.3. Compute Step

For higher dimensional datasets each l2-distance evaluation becomes more costly while the overhead of sampling and

Algorithm 1: Greedy Clustering Heuristic

```
Result: Permutation $\sigma$
$\sigma \leftarrow id$;
$\sigma^{-1} \leftarrow id$;
for $i \leftarrow 0$ to $n - 1$ do
    $a_i \leftarrow$ sorted(adj$_G(i)$);
    for $j \leftarrow 0$ to $k - 1$ do
        if $\sigma(a_i[j]) < i + 1$ then
            continue;
        else if $\sigma(a_i[j]) = i + 1$ then
            break;
        else if $\sigma(a_i[j]) > i + 1$ then
            swap in $\sigma$ entries $a_i[j]$ and $\sigma^{-1}(i + 1)$ ;
            swap in $\sigma^{-1}$ entries $\sigma(a_i[j])$ and $i + 1$ ;
            break;
        end
    end
end
```

updating data structures remains constant. In such cases optimizing the distance evaluations becomes significantly more important than the optimizations described in previous sections.
A single l2 distance evaluation for two vectors is computed by summing the component-wise distances and taking the square root. As the actual value of the l2-distance is unimportant, the square root is omitted and the implementation uses the squared l2-distance. This improvement is not significantly impactful as for high dimensional vectors the cost of computing and summing the differences is dominant.
We decided to limit our implementation to vector dimensions which are divisible by 8 in order to simplify the use of AVX2 SIMD intrinsics. As each AVX2 register can hold 8 single-precision floating point numbers this alleviates the need for auxiliary code to handle the last (fewer than 8) components. The real-world datasets described in Section 4 happen to fulfill this requirement without modification.
We use a AVX2 vector of accumulators for each distance evaluation and process 8 components at a time by subtracting to compute the difference and using an *fmadd* instruction to square the difference and add it to the accumulator vector (cf. tag *l2intrinsics* in Section 4).
We noticed that the restriction of the dimensionality to divisibles by 8 allowed for an easy modification to the way the datapoints are stored in memory. This modification allocates the data neatly aligned to 256 bits at the cost of at most additional 192 bits. This change significantly improves the performance since the *loadu* instructions become faster. We did not observe a speedup by replacing *loadu* intrinsic instructions by the equivalent *load* intrinsic (cf. tag *memalign*). The compute step for a single node can be further

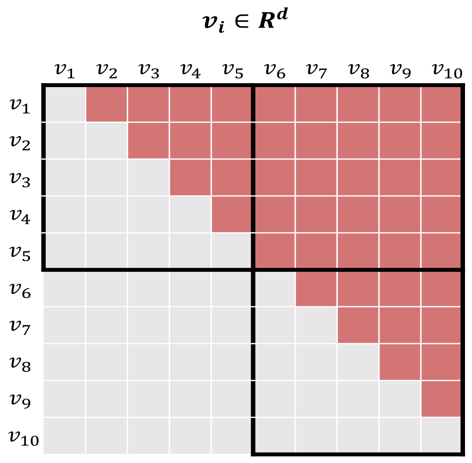
$v_i \in R^d$
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10}

Fig. 2. Illustration of blocked distance evaluations. For the neighborhood containing v_1 through v_{10} the red cells represent distance evaluation. All distance evaluations within a 5 by 5 block are computed simultaneously.

improved by blocking in order to compute the mutual distances between multiple vectors simultaneously. Modifications to the basic NN-Descent algorithms sampling step lead to a neighbourhood never exceeding a fixed size (in our implementation 50 nodes). For these, all the mutual distances have to be computed which can effectively be blocked. We use a blocksize of 5 by 5 vectors (not scalars) in our implementation (Figure 2).

For each such block we allocate an 256 bit AVX2 accumulator for every distance evaluation (either 10 or 25 in total). We then proceed with the computation of the squared l2-distance 8 features at a time for all combinations. In a block where 25 mutual distances are computed simultaneously only 10 AVX2 vectors of data are loaded per 8 dimensions. Without blocking each of theses would be loaded once for every distance evaluation. (1 vs. 25 loads per component). For high dimensional vectors especially this drastic reduction of data loads has a major impact on the performance (cf. tag *blocked*).
Choosing a blocksize of 5x5 allows for each of the 25 accumulators to be allocated to a register which can be seen by inspecting the assembly. If the number of vectors to be compared to one-another is not divisible by 5 a more flexible but slower function is used for the remaining pairs.

4. EXPERIMENTAL RESULTS

This section discusses the experimental performance of the optimizations discussed in the previous section. The main questions we aim to answer are the following:

1. Does our clustering heuristic successfully cluster our data in memory?
2. How does performance scale in terms of dataset size n and dataset dimension d?
3. Which optimizations are beneficial specifically in low or high dimensional settings?
4. How does our final implementation perform to the numba based Python implementation PyNNDescent on real world data?

For all experiments in this section we used the squared euclidean distance and $k = 20$ on single precision floats.

Experimental setup. All the experiments were conducted on a computer running Ubuntu 20.04 LTS. The CPU is a Intel Core i7-9700K CPU @3.60GHz (turbo boost disabled) with the cache sizes L1: 256 KiB, L2: 2 MiB and L3: 12 MiB. The GCC version 9.3.0 was used in all experiments with the following flags: *03*, *ffast-math*, *march=native*, *flto*. The flags *p* and *pg* were used to add run time instrumentation to the code for specific profiling tasks.

Four datasets were used to evaluate the performance and recall of our modified implementation. To investigate scaling behaviours, synthetically generated datasets were used. Additionally real-world datasets including the MNIST handwritten digits [10] were used to verify the generalization of the performance from the synthetic datasets.

Synthetic Gaussian Dataset The input parameter for the generation of this data set is the number of dimensions and the number of points. For the *Single Gaussian Dataset* all points are drawn from one gaussian distribution centered at the origin. In the non-single variant, for each dimension a gaussian is created and centered around the canonical basis vector. For all evaluations the covariance is $2 \cdot I_d$.

Synthetic Clustered Dataset A dataset designed to fulfill our clustered assumption. For every cluster we draw its points from a multivariate Gaussian. Mean and covariance are chosen such that the clustered assumption holds with high probability.

MNIST Dataset [10] The MNIST database contains 70'000 images of handwritten digits given as 784 dimensional vectors of pixel intensity values.

Audio Dataset The audio dataset used in the NN-Descent publication by Wei Dong et al. [1]. Each of the 54'387 points consists 192 features which were extracted from recordings of English sentences.

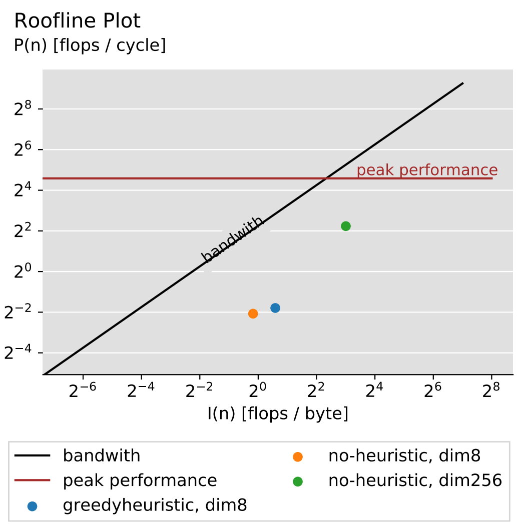

Fig. 3. Roofline plot with peak performance $\pi = 24$ [flops/cycle] and bandwidth $\beta = 4.77$ [bytes/cycle]. The *Synthetic Gaussian Dataset* was used with $n = 131'072$ and the dimensions 8 and 256 respectively.

4.1. Selection Step

Every iteration of NN-Descent consists of a selection and computation step (cf. Section 2). Here we evaluate the improvements detailed in Section 3.1. As previously mentioned the PyNNDescent inspired sampling is up to 16 times faster than the naïve sampling implementation in C. Our improved sampling, which we call *turbosampling*, gives a speedup on top of that of up to 1.12x. The speedup was measured in terms of runtime since the flop count varies across those three implementations and a performance plot would be misleading. We used the *Synthetic Gaussian Dataset* with parameters $n = 16'384$, $d = 8$ for said measurements. Due to space constraints, we omit further evaluations of improvements inspired by PyNNDescent and focus on our own contributions.

4.2. Roofline model

Our optimizations in Section 3.2 and 3.3 are motivated by the fact that the bottleneck of our algorithm changes as we change the dimension. For low dimensional data the computation is memory bound whereas for higher dimension we actually become compute bound as shown in figure 3. In the following the derivation of the roofline model parameters is explained.

As stated in section 2 the work $W(n)$ is calculated from the number of distance evaluations and the dimensionality of the datapoints. For the sake of the roofline model analysis, we additionally need to reason about the data movement $Q(n)$, the bandwidth β as well as the peak performance π.

	LL read misses	LL write misses
no-heuristic ($d = 8$)	122'150'286	14'777'070
greedyheuristic ($d = 8$)	69'653'838	12'328'994
no-heuristic ($d = 256$)	450'209'609	20'438'131

Table 1. Cachegrind results for versions of our implementation with (*greedyheuristic*) or without (*no-heuristic*) memory reordering on the *Synthetic Clustered Dataset* ($n = 131'072$ and 16 clusters) for the specified dimension. Increasing d by a factor of 32 increases the last-level read misses by a smaller factor.

Bandwidth. Using the stream benchmark tool [11], we measured the bandwidth $\beta = 4.77$[bytes/cycle] which seems reasonable as the maximal bandwidth from the manual is rarely attainable ($41.6GB/s$ [12] results in 12.41[bytes/cycle]).

Peak performance. Utilizing 8-way AVX2 SIMD instructions for single precision floating point numbers the peak performance is 32 [flops/cycle] on the Coffee Lake processor based on the Skylake microarchitecture. This bound does not account for the instruction mix which is about 50% 8-way subtractions and 50% 8-way FMA's leading to a bound of 24 [flops/cycle] which is used as our peak performance π.

Data movement. The number of bytes transferred from memory to the cache far exceeds the size of the cache, which necessitates repeated loading and rewriting of data. To reason about the bytes transferred, cachegrind (Valgrind extension)[13] was used. Cachegrind enables examination of the code's cache behaviour by simulating the memory behaviour in terms of first and last level (LL) caches. Using this tool we measured the LL cache data read and write misses. Table 1 summarizes the cachegrind output on our *Synthetic Clustered Dataset* with $n = 131072$ and 16 clusters.

A major improvement of locality when using the greedy clustering heuristic can be seen in terms of the cache misses. Our heuristic nearly halves the number of LL cache data read misses. The increase in operational intensity by the reduction in cache misses moves the computation to the right in the roofline plot (*no-heuristic, dim8* to *greedyheuristic, dim8* in figure 3).

If we increase the dimension we do more work $W(n)$ and achieve a higher operational intensity because the last-level read misses increase by a smaller factor. Since features for a single datapoint are continuous in memory, we attribute this to spatial locality.

4.3. Greedy Clustering Heuristic Evaluation

Quality of Clustering with Greedy Heuristic. This sec-

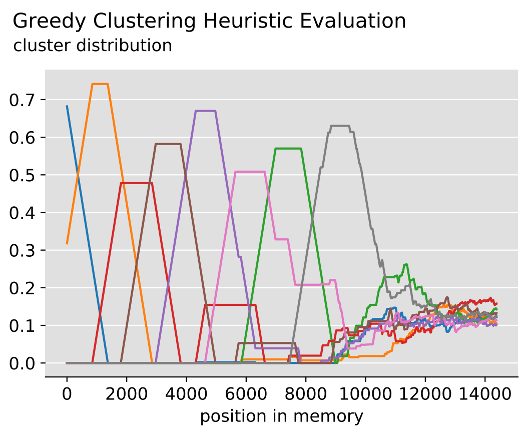

Fig. 4. Each line represents the fraction of datapoints belonging to single cluster in a 2000 sized window (y-axis) starting at the position given in the x-axis. Plot represents the cluster distribution *greedyclustering* reordering retrieved on our *Synthetic Clustered Dataset* with parameters $n = 16'384$, $d = 8$ and 8 clusters.

tion investigates both how well the greedy clustering algorithm recovers the clusters of the underlying dataset and how this causes the speedup.

To test this, we ran our greedy clustering algorithm on the *Synthetic Clustered Dataset*. We then reordered our data according to the permutation σ found by the greedy clustering algorithm. Figure 4 shows that the algorithm successfully recovers many of the clusters in the beginning of the dataset, where the heuristic can chose from many unassigned nodes.

Towards the end of the dataset all of the relative frequencies of the eight clusters are around $\frac{1}{8}$ - our heuristic stops working. This is expected since our algorithm is restricted by design to a single pass through the data. If a certain cluster was already handled and most of the points belonging to it were moved to some location, then the missing few will end up at the end of our simplified memory layout.

Empirical Evaluation on Synthetic Dataset. To test whether our heuristic leads to a speedup when the clustered assumption is satisfied we created such a dataset (*Synthetic Clustered Dataset*). We ran our algorithm twice on this dataset, once with greedy clustering and subsequent reordering of the data in memory and once without those changes. As reasoned in the previous section, we expect a shorter running time when doing the reordering. One can see in Figure 5 that the first iteration takes slightly longer to finish in the greedy clustering version of our algorithm due to overhead, but we can profit from the reordering in all subsequent iterations where we are consistently faster than the non reordered version. In total this amounts to a speedup of 18.46% over all iterations.

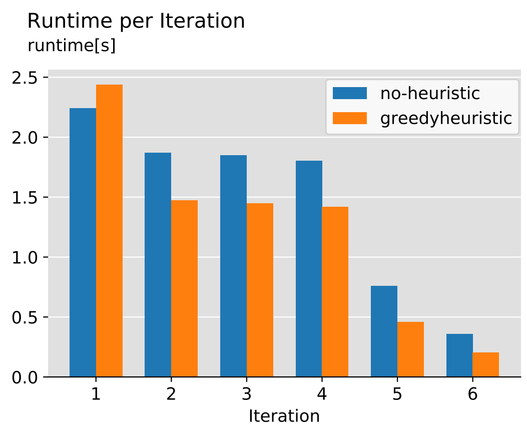

Fig. 5. Time spent on each iteration on Synthetic Clustered Dataset with 16'384 points, 16 clusters and 8 dimensions. While the first iteration is slower due to the overhead of the heuristic, we profit in later iterations.

runtime	MNIST	Audio
no-heuristic	12.12s	4.78s
greedyheuristic	11.45s	4.53s
PyNNDescent	24.41s	14.47s

Table 2. Runtimes on the real-world *MNIST* and *Audio* datasets. Note how even though the clustered assumption might not hold, we still get a speedup on both datasets using our greedy clustering and subsequent reordering of data in memory. Our final implementation *greedyclustering* is significantly faster on both datasets than PyNNDescent.

Real World Data Evaluation. As previously hinted at we observed a speedup on real world data when applying our clustering heuristic. Consult Table 2 to see the runtimes on the *MNIST* and *Audio* datasets. According to our observations, the clustering heuristic does not manage to cluster MNIST semantically. Still, the reordering does improve locality which makes sense considering we move nodes together which are close in data space.

4.4. Performance

To show how the performance scales with number of input points n, we used the *Synthetic Gaussian Dataset* with dimension 256. Every line in Figure 6 corresponds to a specific version of our code, building on top of the improvements of the lines below.

turbosampling Selection step improvements (Section 3.1)
l2intrinsics Optimized L2 distance calculation using intrinsics (Section 3.3)

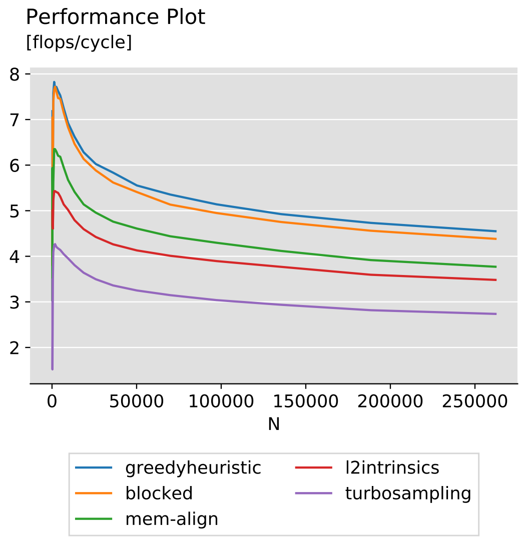

Fig. 6. Performance of the discussed improvements on the *Synthetic Gaussian Dataset*. The dimensions is fixed at 256, while the n increases along the x-axis. Lines correspond to specific tags of our code where every versions contains the improvements of previous versions to showcase the effect of specific changes.

mem-align Data aligned to 256bits (Section 3.3)
blocked Blocking of the L2 distance calculations (Section 3.3)
greedyheuristic Reordering of memory using heuristic (Section 3.2)

Even for small n the performance rapidly degrades as cache sizes are exceeded. Note how all of the improvements lead to speedups, resulting in a performance gain of 1.5x. Considering the difficult setting of a randomized approximation algorithm, this is a remarkable improvement from our *turbosampling* baseline.

4.5. Impact of Dimensionality

We previously argued that increasing the dimension of our input data makes our implementation compute bound. In low dimensions, our algorithm spends a large portion of its runtime in the selection step. For higher dimensions, the focus changes to the calculation step. Some improvements can therefore only unfold their true potential when we increase the dataset dimensionality. In Figure 7 we kept the number of datapoints n fixed at $16'384$ and varied the dimension from 8 up to 3144 in increments of 64. We used our *Synthetic Single Gaussian Dataset* for this evaluation. Note how *turbosampling* only sees a 3.52x performance gain when increasing the dimension while our optimizations targeted at high-dimensional data profit much more from the higher dimension - *blocked* sees a 8.90x speedup going from $d = 8$ to $d = 3144$. Those results indicate that the impact of our optimizations varies with the dimension as expected.

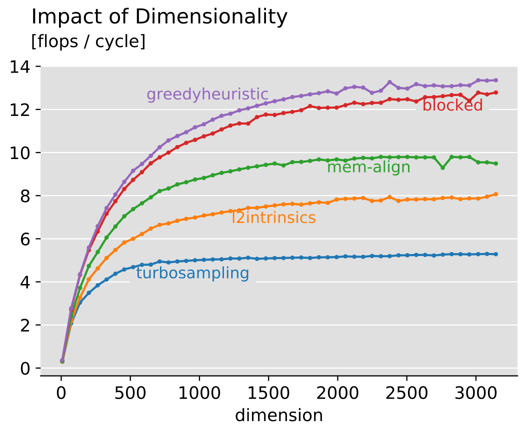

Fig. 7. Performance for the various versions of our code on the *Synthetic Single Gaussian Dataset* with $n = 16'384$ and increasing dimension along the x-axis.

5. CONCLUSION

NN-Descent is an efficient algorithm for computing K-NNGs. The presented fast single-core C implementation incorporates numerous orthogonal optimizations which improve the runtime significantly in low-dimensional, as well as high dimensional usecases.
The presented optimizations to the selection step which reduce the number of memory passes and simplify the datastructures are particularly primarily for low dimensional data. For high dimensional input data the implementation becomes compute bound and beneficial memory alignment and the use of blocked distance evaluations become paramount. Blocking is made possible by the restriction to the l2-distance while restricing the dimensionality makes memory alignment cheaper.
An interesting topic for future work would be to further explore heuristics for reordering the data. The evaluation shows that such heuristics can improve performance even on real world data. Such reordering heuristics are made possible by the iterative nature of the NN-Descent algorithm.
The significantly lower runtime on real world datasets compared to the popular PyNNDescent implementation illustrates the value of specialized implementations for common usecases.

6. CONTRIBUTIONS OF TEAM MEMBERS

Dan. All the first optimizations detailed in Section 3.1 by myself, the optimizations around the greedy clustering heuristic (Section 3.2) were done in tight collaboration with Tobias. Some optimizations involving the L2 norm calculation (tag l2intrinsics), again done in collaboration with Tobias.

Jonas. Optimizations that made it into the master branch were converting recursive functions to iterative ones, scalar replacement, brute-force implementation and the selection of the most suitable norm. A lot of time went into the implementation/optimization of a new approach called " Ball Tree NN Descent" [14] which was later discarded because of its complexity.

Samuel. Presented the project to the TA. Experiments with optimal ordering of MNIST data with UMAP to determine potential speedup of re-ordering heuristics. Work exploring alternative random number generators for use in the sampling step. Implementation of blocked mutual distance computation functionality. Modification of input data storage for 256-bit alignment.

Tobias. Optimizations that made it into the master branch were mainly done around the greedy heuristic project (section 3.2) done in collaboration with Dan. Some work on L2 intrinsics calculation with Dan (tag l2intrinsics). Other notable optimization attempts: rewriting the heaps to support integer weights for less floating point operations, improving the L2 horizontal sum by reducing the number of lane crossings.

References

[1] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web - WWW '11*, Hyderabad, India: ACM Press, 2011, p. 577. DOI: 10.1145/1963405.1963487.

[2] L. McInnes, *Lmcinnes/pynndescent*, original-date: 2018-02-07T23:23:54Z, Jun. 7, 2020. [Online]. Available: `https://github.com/lmcinnes/pynndescent` (visited on 06/09/2020).

[3] *Numba: A high performance python compiler*. [Online]. Available: `https://numba.pydata.org/` (visited on 06/11/2020).

[4] *Umap implementation*, original-date: 2017-07-02T01:11:17Z, Jun. 11, 2020. [Online]. Available: `https://github.com/lmcinnes/umap` (visited on 06/11/2020).

[5] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform manifold approximation and projection for dimension reduction," *arXiv:1802.03426 [cs, stat]*, Dec. 6, 2018. arXiv: 1802.03426.

[6] E. Becht, L. McInnes, J. Healy, C.-A. Dutertre, I. W. H. Kwok, L. G. Ng, F. Ginhoux, and E. W. Newell, "Dimensionality reduction for visualizing single-cell data using UMAP," *Nature Biotechnology*, vol. 37, no. 1, pp. 38–44, Jan. 2019, ISSN: 1087-0156, 1546-1696. DOI: 10.1038/nbt.4314. [Online]. Available: `http://www.nature.com/articles/nbt.4314` (visited on 06/11/2020).

[7] R. Guerraoui, A.-M. Kermarrec, O. Ruas, and F. Taïani, "Smaller, faster & lighter KNN graph constructions," in *Proceedings of The Web Conference 2020*, Taipei Taiwan: ACM, Apr. 20, 2020, pp. 1060–1070. DOI: 10.1145/3366423.3380184.

[8] J. D. Baron and R. W. R. Darling, "K-nearest neighbor approximation via the friend-of-a-friend principle," Mar. 6, 2020. arXiv: 1908.07645.

[9] E. Skomski, *Eskomski/nn_descent*, original-date: 2019-01-10T05:35:25Z, Apr. 22, 2020. [Online]. Available: `https://github.com/eskomski/nn_descent` (visited on 06/09/2020).

[10] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: `http://yann.lecun.com/exdb/mnist/`.

[11] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[12] *Intel core i7-9700k processor*. [Online]. Available: `https://ark.intel.com/content/www/us/en/ark/products/186604/intel-core-i7-9700k-processor-12m-cache-up-to-4-90-ghz.html` (visited on 06/10/2020).

[13] N. Nethercote, "Dynamic binary analysis and instrumentation," Jan. 2004.

[14] S. M. Omohundro, "Five balltree construction algorithms," p. 22, Nov. 20, 1989.