HIERARCHICAL DENSITY BASED CLUSTERING

Mark Frey, Beat Hubmann, Martin Tschechne, Robin Worreby

ETH Zurich, Switzerland
{freymark, bhubmann, martints, rworreby}@ethz.ch

ABSTRACT

We provide our own high performance C implementation[1] of the HDBSCAN clustering algorithm, based on the original JAVA code [1]. The code is optimized using various techniques and shown to outperform another state-of-the-art implementation while retaining the same solution. We report on the optimizations used, the gained speedups of certain sections, and the performance during critical parts of our code, all on a competitive high-end Zen2 CPU supporting AVX2 SIMD instructions.

1. INTRODUCTION

In many real life machine learning applications annotated ground truth labels generated by experts are not available or simply prohibitive to acquire considering time or cost. These circumstances lead to rise of the subfield *unsupervised learning*. One of the most commonly used methods based on unsupervised learning is *clustering*, where it is desired to find a grouping of data points (i.e. clusters) based on their similarity. Often times these data points live in a high dimensional space and it is not straight forward to visually inspect the data and cluster it with bare eyes. Over recent decades numerous clustering algorithms have been invented to overcome this issue and find clusters even for high dimensional scenarios.
Generally, all clustering algorithms make specific assumptions about the underlying data distribution or produce different kinds of partitions and therefore have advantages and disadvantages in specific scenarios [1]. These clustering algorithms can broadly be categorized into *parametric* (where the shape of the data distribution is explicitly or implicitly assumed, e.g. Gaussian ball assumption) and *non-parametric* (where the density of the underlying data distribution is estimated from the data). Algorithms can be further classified depending on what kind of partitioning they produce, either a *flat* one (single grouping of data) or a *hierarchical* one (sets of relationships of data). Most prominent example are k-means [2] as a parametric flat algorithm, Ward-Linkage [3] as parametric hierarchical and DBSCAN (Density Based Clustering of Applications with Noise) [4] as non-parametric flat. DBSCAN was further extended by Campello et al. [5, 1] (HDBSCAN) to produce hierarchical partitions and therefore to overcome the limitation of a single flat clustering results.
HDBSCAN yields meaningful partitions for arbitrary shapes of clusters, different cluster densities, different cluster sizes and for noisy data sets, making it an ideal candidate for high-dimensional real-world data sets. Major drawback of HDBSCAN is its high computational complexity of $\mathcal{O}(n^2)$, which makes its use prohibitive for very large data sets. To deal with this issue, the use of more efficient data structures and more suitable minimum-spanning-tree algorithms has been proposed [6], resulting in a widely used high performance Python implementation [7].
In this work we continue to optimize the runtime of the HDBSCAN algorithm by lower-level processor optimizations (unrolling, vectorization and blocking) and reducing the number of floating point operations by exploiting symmetry and by caching intermediate calculations. We show that that the majority of runtime is spent in the first steps of the algorithm and therefore provide highly optimized implementations of these steps in C. We present individual benchmarks for all performance critical steps for different sizes of data and commonly used distance metrics. Our optimizations speed up individual parts of the algorithm by up to 150 times compare to our unoptimized implementation. Moreover, to assess the total runtime we benchmark on a real-world sized high-dimensional synthetic data set, yielding a total speedup of 34 compared to our base implementation. Finally, we compare our best version to the high-performance Python implementation where we can present a 5-fold speed up. Note that we do not reduce the computational complexity nor make use of any numerical approximations, moreover partitions produced by our optimized implementation are identical to the ones of a reference implementations.

[1]Code can be found at: https://gitlab.inf.ethz.ch/COURSE-ASL/asl21/team33

2. BACKGROUND ON THE ALGORITHM

HDBSCAN is built upon and extends the widely used DBSCAN [4] algorithm. Familiarity with the base algorithm (DBSCAN) is assumed and we focus on the difference HDBSCAN introduces. A short introduction to DBSCAN can be found in App. A. After establishing HDBSCAN's complexity, a cost analysis is performed.

HDBSCAN. As hinted at above, HDBSCAN improves upon DBSCAN implementing two main changes. As a first major change, HDBSCAN abolishes the concept of border points as employed by DBSCAN: Only core points can be part of a cluster, while the former border points are counted as outliers. Firstly, this approach is more consistent with the concept of a density level set considering border points' estimated density technically falls below the threshold [8]. Secondly, this also resolves the anomaly that assignment of border points to cluster might be ambiguous as border points reachable from multiple clusters are assigned to a certain cluster depending on the order in which the points are assigned to clusters. Having thus done away with the edge case of border points, distance measures can be simplified: The *core distance* of a data point p is the radius within which exactly *minPts*, again including the data point under consideration itself, come to lie (Fig. 1).

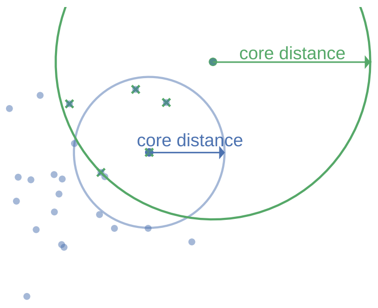

Fig. 1. Illustration of the core distance concept: *minPts*=6 [6].

Building on the core distance, the all-important *mutual reachability distance* (MRD) between two data points p and q is defined as the maximum out of the two respective core distances of p and q and the metric distance between p and q (Fig. 2).

It can be shown [9] that mutual reachability distance is well-suited to the second major change introduced by HDBSCAN: By adapting a single linkage hierarchical clustering approach, HDBSCAN no longer requires the distance measure threshold ϵ hyperparameter, doing away with both the domain knowledge requirement and the fixed density restriction of DBSCAN and thus only requiring the minimum cluster size *minPts* as a sole remaining hyperparameter.

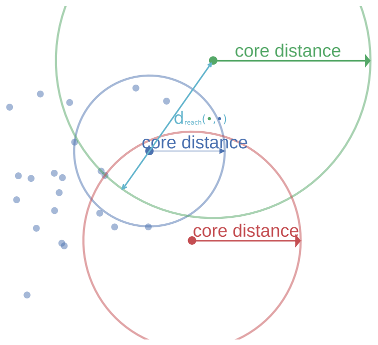

Fig. 2. Illustration of the mutual reachability distance concept: *minPts*=6 [6].

To obtain the single linkage hierarchical clustering, the data points are considered as vertices in an undirected weighted graph where the edge between any two data points carries a weight equal to the mutual reachability distance between those two points. It is a key insight of HDBSCAN that instead of starting with a complete undirected weighted graph with n^2 edges from which edges with decreasingly large mutual reachability distances are pruned until a single linkage hierarchical cluster is obtained, an identical result can be obtained far more efficiently by building a minimum spanning tree using e.g. the DJP ("Prim's") algorithm (Fig. 3).

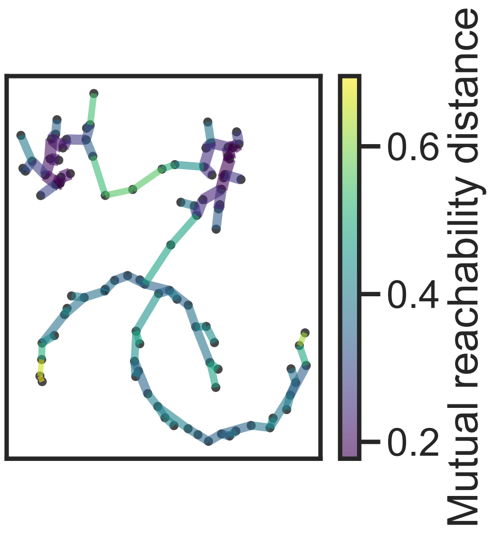

Fig. 3. Illustration of a sample mutual reachability distance-based minimum spanning tree [6].

In a next step the minimum spanning tree is transformed into a dendrogram reflecting the hierarchical nature of the connected graph components: After sorting the MST's edges by edge weight, a new merged cluster is created for every edge (Fig. 4).

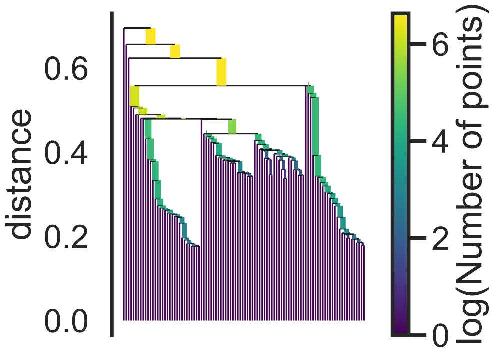

Fig. 4. Illustration of a sample mutual reachability distance-based hierarchical single linkage dendrogram [6].

It is noteworthy that at this stage, a horizontal global cut across the obtained dendrogram would yield a result identical to the one obtained by the original DBSCAN algorithm: A set of clusters at a fixed mutual reachability distance equal in meaning to a fixed density. Before extracting the final HDBSCAN clusters, the dendrogram is condensed (Fig. 5) by pruning leaves representing small sized child clusters. This is done without reattaching the affected points to any other clusters at a later point.

Selecting the final clusters requires a notion of subcluster mass (alternatively called stability) and a single selection rule: The mass M of subcluster C containing points p is defined as

$$M(C) = \int_{p \in C} (\lambda(p) - \lambda_{\text{min}}(C))dx$$

which intuitively can be understood as the amount of real estate covered by a plot of subcluster C. Traversing the condensed dendrogram tree from the leaves towards the root to obtain a flat clustering, we select each subcluster whose mass exceeds the combined mass of its child clusters as a final cluster (Fig. 5). All points not part of a final selected cluster are considered noise points. Alg. 1 renders an abstract description of HDBSCAN.

Cost Analysis. HDBSCAN (Alg. 1) as described above consists of the two main parts `calculate_distances` and `construct_mst`, whereas the remainder of the algorithm can be considered as scaffolding not requiring significant computational effort. This assertion can easily be confirmed without preempting in-depth experiments by observing a plain base implementation of HDBSCAN (Fig. 7). It therefore is justifiable to focus the cost analysis on the two main components: For `calculate_distances` based on the commonly employed Euclidean distance, a suitable floating point cost measure for n d-dimensional points is

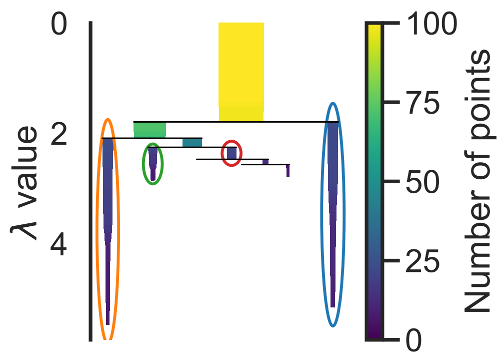

Fig. 5. Illustration of a final cluster selection from a sample condensed hierarchical single linkage dendrogram ($\lambda = 1/\text{distance}$) [6].

given by

$$\begin{aligned} C_{\texttt{calculate\_distances}}(n,d) &= C_{\text{add}} \cdot N_{\text{add}} \\ &+ C_{\text{mult}} \cdot N_{\text{mult}} \\ &+ C_{\text{sqrt}} \cdot N_{\text{sqrt}} \end{aligned}$$

where as

$$C_{\texttt{construct\_mst}}(n) = C_{\text{compare}} \cdot N_{\text{compare}}$$

is a cost measure best suited to the combinatoric nature of `construct_mst` based on Prim's algorithm. First further detailing $C_{\texttt{calculate\_distances}}(n)$, we observe that

$$\begin{aligned} N_{\text{add}} &= (n^2 - n) \cdot 2d \\ N_{\text{mult}} &= (n^2 - n) \cdot d \\ N_{\text{sqrt}} &= (n^2 - n) \cdot 1 \end{aligned}$$

and hence obtain

$$\begin{aligned} C_{\texttt{calculate\_distances}}(n,d) &= C_{\text{add}} \cdot (n^2 - n) \cdot 2d \\ &+ C_{\text{mult}} \cdot (n^2 - n) \cdot d \\ &+ C_{\text{sqrt}} \cdot (n^2 - n) \end{aligned}$$

as a final cost measure for that part. Moving on to `construct_mst`, we note that when starting Prim's algorithm at any given vertex (i.e. data point) in a complete weighted graph, $n-1$ edges are to be considered, resulting in $(n-1)-1$ floating point edge weight (i.e. core distance) comparisons. During the second step, there are $2 \cdot (n-2) - 1$ floating point comparisons to be made. The principle of induction therefore leads us to

$$\begin{aligned} N_{\text{compare}} = &\; 1 \cdot (n-1) - 1 \\ &+ 2 \cdot (n-2) - 1 \\ &+ 3 \cdot (n-3) - 1 \\ &+ \ldots \\ &+ (n-1) \cdot (n-(n-1)) - 1 \end{aligned}$$

which by repeated application of young Gauss' insight can be shortened to

$$N_{\text{compare}} = \frac{n^3 - 2n^2 - 5n + 6}{6}$$

yielding

$$C_{\texttt{construct\_mst}}(n) = C_{\text{compare}} \cdot \frac{n^3 - 2n^2 - 5n + 6}{6}$$

for the second main part of the algorithm.
Overall, the theoretical floating point operations cost of the core elements of the algorithm therefore can be summed up as:

$$\begin{aligned} C(n, d) &= C_{\text{add}} \cdot (n^2 - n) \cdot 2d \\ &+ C_{\text{mult}} \cdot (n^2 - n) \cdot d \\ &+ C_{\text{sqrt}} \cdot (n^2 - n) \\ &+ C_{\text{compare}} \cdot \frac{n^3 - 2n^2 - 5n + 6}{6} \end{aligned}$$

Asymptotic operational intensities for all implemented distance measures are given in Tab. 1, whereas the corresponding roofline model analysis is given in Fig. 6.

Distance	$W(n,d)$	$Q(n,d)$	$I = W/Q$	
			$d << n$	$d \to n$
Euclidean	$n^2(3d+1)$	$8(nd+n^2)$	$\frac{3}{8}d$	$\frac{3}{16}n$
Manhattan	$2n^2d$	"	$\frac{1}{4}d$	$\frac{3}{16}n$
Cosine	$n^2(6d+4)$	"	$\frac{6}{8}d$	$\frac{6}{16}n$
Pearson	$n^2(14d+6)$	"	$\frac{14}{8}d$	$\frac{14}{16}n$
Supremum	$2n^2d$	"	$\frac{1}{4}d$	$\frac{3}{16}n$

Table 1. Operational Intensities for different distance functions. For $d << n$ all distance functions are of order $\mathcal{O}(1)$, whereas for $d \to n$ the function are of order $\mathcal{O}(n)$. We work with double precision (i.e. 8 byte per float), assume infinite cache and count reads as well as writes. Further only $+,-,\star,/,$`sqrt` are counted as floating point operation, operations like `max` for `abs` are not.

Complexity. As given in the original publication [5], constructing the minimum spanning tree on n data points with d dimensions each using Prim's algorithm based on an ordinary list search results in HDBSCAN having a running time of $\mathcal{O}(dn^2)$ with a space footprint of $\mathcal{O}(dn)$. At the cost of an increased space requirement of $\mathcal{O}(n^2)$ for keeping the point distance matrix once computed, the time complexity reduces to $\mathcal{O}(n^2)$.

3. APPLIED OPTIMIZATIONS

In this section we point out potential bottlenecks of the naive base implementation and reason about which parts of the

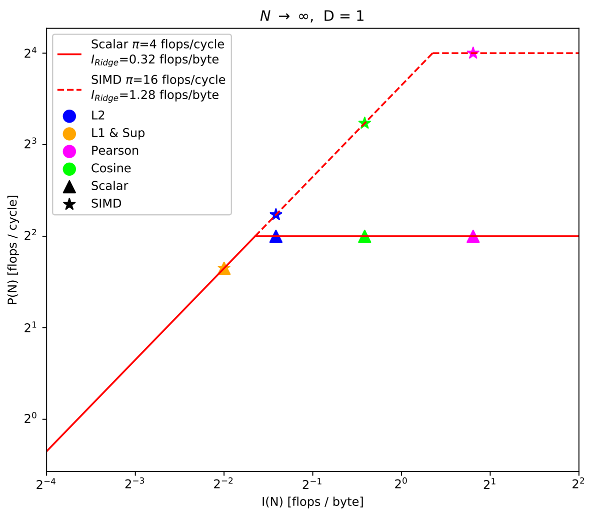

Fig. 6. Roofline plot for all considered distance functions for the unfavorable case of $d = 1$. Even for the case of small d most functions quickly become compute bound on the benchmarking hardware.

Algorithm 1 Density-based clustering based on hierarchical density estimates

```
    function HDBSCAN($X \in \mathbb{R}^{n \times d}$)
2:      $D \leftarrow$ calculate_distances($X$)
        $MST \leftarrow$ construct_mst($D$)
4:      mst_quicksort($MST$)
        $C, \eta \leftarrow$ compute_hierarchy($MST$)
6:      propagate_tree($C$)
        $\hat{C} \leftarrow$ find_clusters($C$)
8:      $O \leftarrow$ calculate_outliers($C, D, \eta$)
        return $\hat{C}, O$        ▷ Prominent clusters and outliers
```

algorithm yield the most potential for performance gains. Moreover we will introduce each individual optimization step we have applied in this work.

Bottlenecks. Our naive unoptimized version is based on the original JAVA code released by the authors of the HDBSCAN publication [1]. The Java code was translated into pure C code (besides some non-performance critical parts facilitating input and outputs which make use of C++ standard library data structures) without altering algorithm steps or applying any optimizations. This translation compiled with `-O0` is our base implementation against which we measure speedups of further optimizations. The relative proportion of cycles spent in each step of Alg. 1 can be seen in Fig. 7. Cycles are measured on the benchmarking hardware (see Sec. 4) on the benchmarking data set. The majority of execution time is used to calculate the core distances amounting to over 60% of the cycles. That observa-

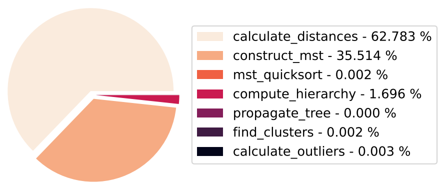

Fig. 7. Relative amount of cycles spent in each algorithm step in our unoptimized base implementation. Almost two third of the total measured cycles are spend in `calculate_distances` and little more than one third on the construction of the MST `construct_mst`. Other algorithm steps are neglectable in comparison.

tion is in alignment with our complexity analysis, in which `calculate_distances` has an asymptotic complexity of $\mathcal{O}(dn^2)$, which is the highest compared to other steps. The second next costly step is the creation of the minimum spanning tree which amounts to roughly a third of the executed cycles. Again this is in alignment with our complexity analysis, where it has the second largest asymptotic complexity $\mathcal{O}(n^2)$. All remaining steps of the HDBSCAN algorithm consume considerably less cycles since most of them have a small complexity as well as far less FLOPs and are therefore almost not worth optimizing. Considering these measurements we focused on optimizing the two most costly steps first.

Distance functions. Possible distance metrics for our HDBSCAN implementation are Euclidean, Manhattan, Cosine, Pearson-Correlation and Supremum, where p and q are two points in $\mathbb{R}^d$.

$$\begin{aligned}
\ell_2(p,q) &= \|p-q\|_2 = \sqrt{\sum_i^d (p_i - q_i)^2} \\
\ell_1(p,q) &= \|p-q\|_1 = \sum_i^d |p_i - q_i| \\
D_C(p,q) &= 1 - \frac{p \cdot q}{\|p\|_2 \, \|q\|_2} \\
\rho(p,q) &= 1 - \frac{\operatorname{cov}(p,q)}{\sigma_p \sigma_q} \\
\ell_\infty(p,q) &= \max_i |p_i - q_i|
\end{aligned}$$

To optimize the computation of a single point-to-point distance calculation, we apply basic optimization strategies like unrolling, vectorization via AVX and the use of fused multiply-add instructions (FMA). Since all of these optimization are applied over the dimension d of the points, their beneficial effects are more prominent for high dimensional data, where the overhead becomes negligible. The case of high dimensional data is a commonly encountered in clustering and actually one of the most common application scenarios for HDBSCAN and therefore does not impose any major restrictions to our implementation.

Core distances. To find the core distance of each point (i.e. the distance to the kth nearest neighbor) one needs to calculate the distance to all other points first and then extract the kth closest distance. The base implementation calculates all the pairwise distances between all points explicitly resulting and n^2 calls to the selected distance function and only keeps track of the k smallest distances. This is very effective in terms of space complexity since only k distances have to be stored temporally. The disadvantage of this approach is the number of redundant computations, which becomes problematic as soon as the number of dimensions d becomes large. Since pairwise distances are symmetric $d(p,q) = d(q,p)$ one can cache and reuse a previous computation. We exploit this this by constructing a symmetric distance matrix first and then computing the core distance for each point. This way we can save roughly half the numbers of calls to the distance function. It should be noted that this optimization requires additional memory of $\mathcal{O}(n^2)$. We further reduces the required space by only storing one triangular half of the distance matrix. For a very large number of points n the physical cache size becomes a limiting factor. We introduced blocking in order to avoid effects of limited cache size. Since the Euclidean distance is probably one of the most commonly used distance in applications, we took further steps specifically for this case.

Minimum Spanning Tree. As stated before, HDBSCAN uses Prim's algorithm to create the MST with the Mutual Reachability Distance as weights for the edges. The algorithm starts by adding one vertex to the tree at initialization and then successively adds the vertex closest to the tree. By computing the MRD within the loop searching for the next vertex, we are using temporal locality. To compute the MRD of two points the core distances of each point and the distance between them are needed. In the reference code, the focus lies on saving memory, calculating the distance between the two points within the MST algorithm. However, as explained in the section before, we are caching the calculated distances to save computations. Using these cached distances again results in a speed-up of 6.2 over the base implementation (on a dataset with 50k points and 64 dimensions). The remaining parts of the MST algorithm are only comparisons that can be vectorized. Masks are used to keep track of points already attached to the tree. In case that all points in the vector have already been added to the tree, we can skip the computations entirely. Vectorization together with an unrolling factor of two achieves an additional speed-up of 3.7. Interestingly, vectorization and unrolling by a factor of 4 decreases the performance and unrolling without vectorization turned out to have a negative effect on performance in general. This is most likely due to effects resulting from the masking. The greater the unrolling factor the more computations we have to make, which will later be discarded due to the point being already part of the tree.

Data Structures. For `list`, implemented as doubly-linked list, whose instances once created do not undergo a considerable amount of mutations, no optimization apart from cache-aligned memory allocation was pursued. As `map` is implemented based on `ordered_set` and `vector`, its optimizations consist of cache-aligned memory allocation as well as vectorized insert and erase operations. `ordered_set`, implemented based on pointers, received cache-aligned memory allocation along with vectorized insert, erase and find operations. Furthermore, a hybrid search function was implemented where vectorized linear search for ordered sets sized smaller than a crossover parameter is used before switching to binary search for larger ordered sets. Finally `vector`, implemented based on pointers to pointers, received cache-aligned memory allocation together with vectorized insert, erase and contains operations.

4. EXPERIMENTAL RESULTS

In the following we introduce the benchmarking hardware and present our experimental results for all applied optimizations to the core distance calculation and the minimum-spanning-tree creation.

Experimental setup. All experiments are executed on an AMD Ryzen 9 3900X[2] chip with 3.8 GHz base frequency (turbo boost disabled) with cache level sizes of 786 KiB for L1, 6 MiB L2 and 60 MiB L3 cache. The code is compiled with gcc 9.3.0 on Ubuntu 20.04. Speedups are always compared to our baseline implementation which uses no optimizations and is compiled with `-O0`. Further we disable automatic unrolling done by the compiler with `-fno-tree--vectorize` when increasing the optimization level to `-O3`. The distance benchmarking is conducted on random data of different size and dimension. The minimum spanning tree construction is benchmarked on a medium-sized data set with 10.000 random points of dimension 32. To benchmark cumulative optimizations and compare with a python high-performance library, we created our own real-life sized synthetic data set with 50.000 64-dimensional points randomly drawn from three multidimensional Gaussian distributions with different centers and equal variance.

Results. To assess the performance of optimizations applied to the distance calculations `calculate_distances` we benchmarke each optimization for different sized data sets n as well as for different dimensions d. Fig. 8 shows the results for the ℓ_2 distance function. We compare

[2]https://en.wikichip.org/wiki/amd/ryzen_9/3900x

Optimization	Operations GFLOPs	Intensity $\frac{\text{GFLOPs}}{\text{s}}$	Performance $\frac{\text{FLOPs}}{\text{cycle}}$
Baseline	482.49	0.945	0.26
`-O3`	482.49	4.265	1.18
Symmetry	241.24	1.852	0.51
Unroll 4	245.00	2.048	0.56
Vectorize	261.24	3.852	1.06
Blocking	246.33	12.468	3.45

Table 2. Number of operations, compute intensity and performance of euclidean distance in `calculate_distances`. Measurements were made on the benchmarking infrastructure with our data set of dimension d of 64 and 50.000 points.

from top left to bottom right 2-fold unrolling, 4-fold unrolling, vectorization via AVX2, using FMAs instructions, blocking and further optimizations specific to the ℓ_2 distance. For a small d the introduced overhead is too large and no speedup is visible. Once the points have a sufficiently large dimensionality, speedups become visible for all optimizations. Highest speedups are naturally achieved for very high dimensions and a relative small number of points n, where the vectorization effects show their full advantage and data fits into L1 cache. For a increasing number of data we see speedups declining again. This can explained with the limited cache size. Once the whole data does not fit into cache anymore, the latency for fetching data increases. These cache effects can be limited by blocking, as one can see in the bottom mid heatmap of Fig. 8. Finally, the bottom right plot shows results for specific optimization only applicable for the ℓ_2 distance. Interestingly, one can observe that the speedup decreases again for larger dimensionality. This has to do with the selected block-size of 16×16, which was empirically chosen and not fine tuned. A possible solution to overcome this issue would be to have adaptive blocking. Tab. 2 shows the operations, intensity and performance for each optimization. Just changing the optimization flag from `-O0` to `-O3` already dramatically increases the performance. Avoiding redundant distance calculations reduces the number of FLOPs executed. However, this introduces a worse pattern for cache access, as the entries that need to be calculated are lying in the upper triagonal part of a square matrix. With the worse spatial locality the performance drops as well. From this point on the FLOPs per cycle increases with each of our applied optimizations again to 3.45 FLOPs per cycle, 13 times more than our baseline and roughly three times as many as the "trivial" `-O3` optimization while only performing half the FLOPs.

The second most important step according to the bottle-

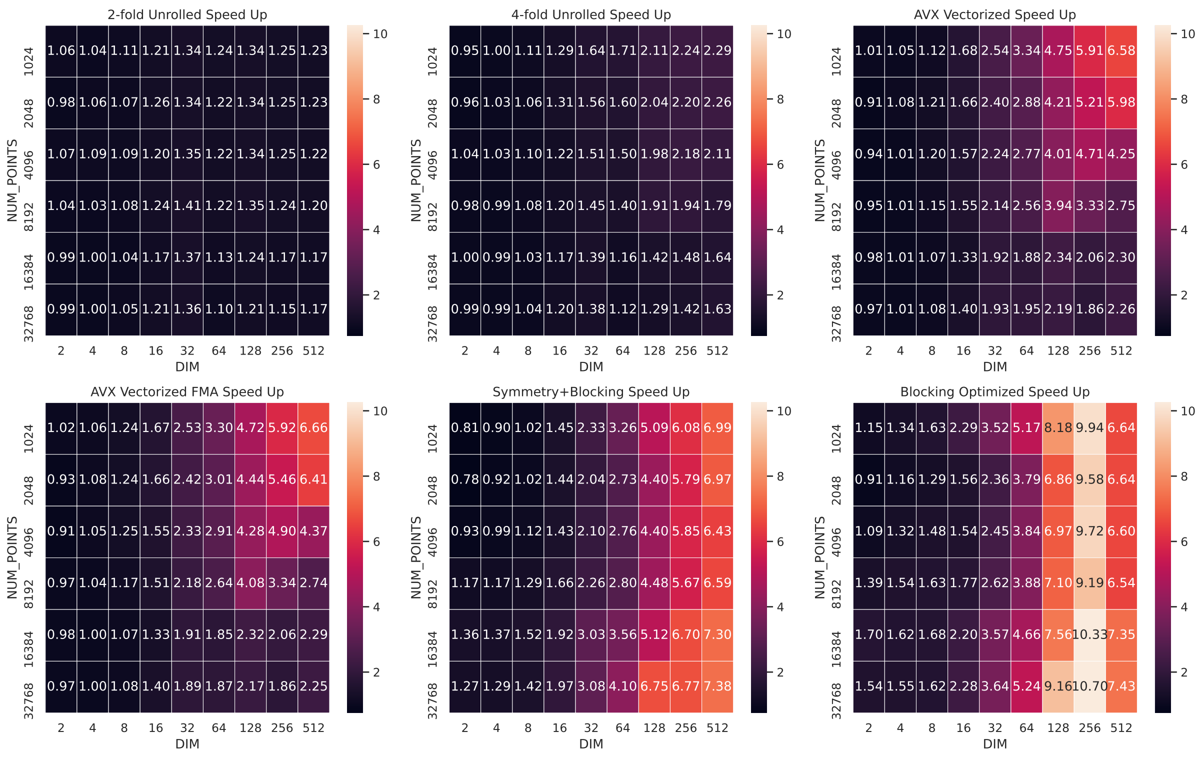

Fig. 8. Speedups of Euclidean distance function for all applied optimizations depending on size of data set and dimension of data. Speedup is compared to the naive base implementation compiled with `-O0`.

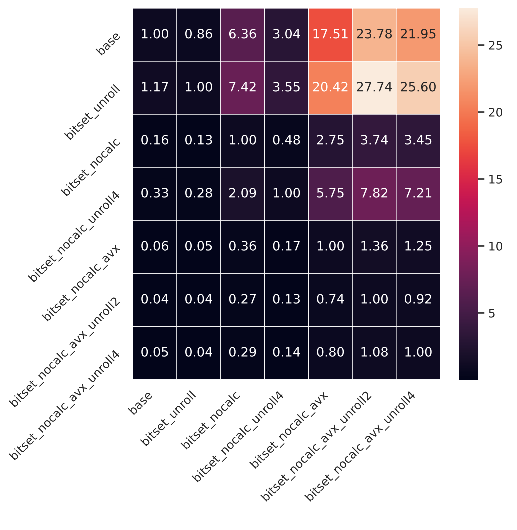

Fig. 9. Speedup matrix for `construct_mst`. Each entry represents the measured cycles of the row ID version divided by the measured cycles of column ID version (i.e. the speedup). For example `bitset_nocalc` optimization needs 6.36 times less cycles than the `base` version. Benchmark was executed on a data set with 10.000 points each with a dimension d of 32.

neck analysis (Fig. 7), the MST construction `construct_mst`, is benchmarked on a single medium sized data set. Results of that benchmark can be seen in Fig. 9. The speedup matrix compares all optimization with each other where the name of the optimizations are the row and column ID's. In this case `base` is already compiled with `-O3`. From the speedups we can see that the first step, the unrolling and usage of bitset itself does not result in a speedup, but instead yields only a speed of 0.86 times the base speed. When introducing caching to not recompute the core distances, as described in Sec. 3, we can see the clear performance boost of a factor of 7.42. Other noteworthy observations are that the unrolling for `construct_mst` has limited effects on the speedup, which we attribute to the sequential nature of the MST construction. The highest speedup is reached with 2-fold unrolling and vectorization using the AVX2 instruction set, leading to an overall improvement of 27.74 times the speed of the base `-O3` implementation.

To assess the overall performance gain from our optimizations we execute each algorithm from beginning to end and measure the executed cycles for each step in Alg. 1. Here we only focused on the three most important optimization levels: starting from the baseline implementation we first change the compile flag from `-O0` to `-O3`. This can

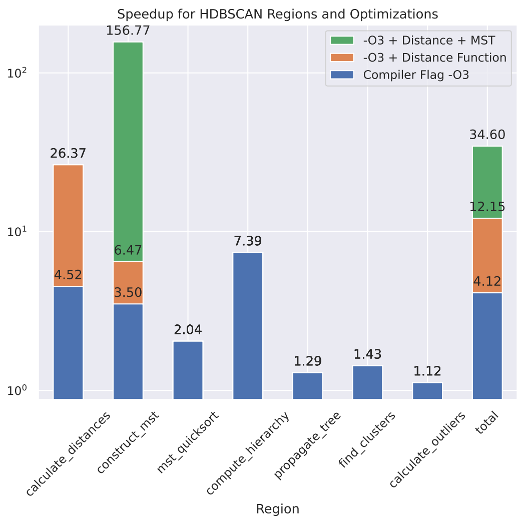

Fig. 10. Region-wise speedup for different optimization steps. Blue shows the achieved speedup in all regions from using the compiler flag -O3. The orange regions shows the additional speedup gained from our calculate_distances optimizations. The green regions shows the additional speedup gain from the MST optimizations.

be considered as "performance gain for free" as it does not require any modifications to the code. The blue region in Fig. 10 shows how much speedup can already be achieved with this simple adjustment. Almost all regions profit from this, the total number of cycles can be reduced by factor of 4.12. In a next step the best performing core distance distance calculation optimization is turned on (Fig. 10 orange) which yields a 26.37-fold speedup for calculate_distances compared to the base implementation and over five times less cycles compared to the previous -O3 version. One can also see that construct_mst benefits from faster distance calculations since at this time the distance function is also called during the construction of the MST. Overall we can see a speedup of 12.15 times, which is roughly three times faster than the -O3 version. Lastly, optimizations to the MST construction are applied. From our previous benchmarking results (Fig. 9) on can observe that bitset_nocalc_avx_unroll2 performs best on the particular hardware for the given data set, therefore we selected this as final optimization level (Fig. 10 green). This final step reduced the number of cycles necessary to construct the MST by a factor of 156.77 and overall by a factor of 34.6 for the whole algorithm. All other algorithm steps unrelated to the MST remain constant in terms of speedup compared to the previous.

In order to evaluate our implementation we compare it against a widely-used state-of-the-art implementation, namely the Python scikit-contrib HDBSCAN library [6]. On our

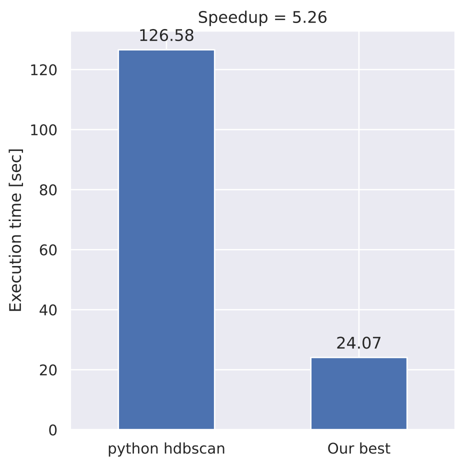

Fig. 11. Execution times of the high-performance Python library [6] and our best version. Both codes were executed on the same hardware and on the same data.

synthetic data set we reach a speedup of factor 5.26 compared to the Python code, as shown in Fig. 11. With this speedup the execution time on a data size in the magnitude of real-world applications can be reduced from minutes to seconds.

5. CONCLUSIONS

We have provided a high performance C implementation of the HDBSCAN algorithm, making use of many low-level optimizations techniques like strength reduction, loop unrolling, vectorization and optimization with AVX2 SIMD instructions, blocking, and memoization. We have shown that our implementation outperforms a current state-of-the-art implementation on a real-world sized high-dimensional synthetic data set by a factor of more than five while still producing the same results. The current implementation allows for five different distance metrics to be used, where the most commonly used one, the euclidean, has been the furthest optimized. Possible future extension include the optimization of code parts that are not contributing much to the overall execution time, for example the compute_hierarchy and further improving the distance functions. Other extensions would be to investigate the usage of other algorithms for distinct parts of HDBSCAN, for example replacing Prim's algorithm by Kruskal's algorithm in construct_mst in order to optimize cache access. Additionally, the usage of other compiler families and other hardware could be investigated. The latter one, especially the AVX512 SIMD instructions, could have provided further optimization to construct_mst by using the bit masking extensions introduced in AVX512, but were unfortunately not available to the authors.

6. CONTRIBUTIONS OF TEAM MEMBERS

Individual contribution of team members.

Mark Frey. C++ reference implementation of the distance functions, I/O, MST computation, hierarchy computation, propagation and finding prominent clusters. Conversion to C base implementation using Beat's C data structures. Optimization of the MST code and blocking / specialization of the core distance calculation.

Beat Hubmann. list, map, ordered set, vector; UndirectedGraph: Implement w/ bare-bone functionality tailored to use case, AVX vectorizations, hybrid search for ordered set, memory aligns. Distance calculations: Initial AVX vectorizations up to and including 2-fold unroll. Common/helpers: FP/int reduction operations. Python scikit-contrib HDBSCAN benchmarking script.

Martin Tschechne. Roofline plot (Fig. 6) and operational intensities (Tab. 1), 2-fold unrolling for all 5 distance calculation functions, generating benchmark and plotting scripts for (Fig. 8, Fig. 10).

Robin Worreby. Distance calculations: 4-fold AVX vectorizations and FMA vectorizations. Benchmarking: Implemented benchmarking with LIKWID (cycle counting and FLOPs counting) and TSC (cycles) from the lectures. Script for analyzing the parts of the code, as splitted in Algorithm 1.

7. REFERENCES

[1] Ricardo J. G. B. Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander, "Hierarchical density estimates for data clustering, visualization, and outlier detection," *ACM Trans. Knowl. Discov. Data*, vol. 10, no. 1, July 2015.

[2] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[3] Joe H. Ward, "Hierarchical grouping to optimize an objective function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, 1963.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. 1996, KDD'96, p. 226–231, AAAI Press.

[5] Ricardo Campello, Davoud Moulavi, and Joerg Sander, "Density-based clustering based on hierarchical density estimates," in *Proceedings of the 17th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, Apr 2013, vol. 7819, pp. 160–172.

[6] Leland McInnes and John Healy, "Accelerated hierarchical density based clustering," *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, Nov 2017.

[7] Leland McInnes, John Healy, and Steve Astels, "hdbscan: Hierarchical density based clustering," *The Journal of Open Source Software*, vol. 2, no. 11, Mar 2017.

[8] John A. Hartigan, *Clustering Algorithms*, John Wiley & Sons, Inc., USA, 99th edition, 1975.

[9] Justin Eldridge, Mikhail Belkin, and Yusu Wang, "Beyond hartigan consistency: Merge distortion metric for hierarchical clustering," in *Proceedings of The 28th Conference on Learning Theory*, Peter Grünwald, Elad Hazan, and Satyen Kale, Eds., Paris, France, 03–06 Jul 2015, vol. 40 of *Proceedings of Machine Learning Research*, pp. 588–606, PMLR.

[10] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu, "Dbscan revisited, revisited: Why and how you should (still) use dbscan," *ACM Trans. Database Syst.*, vol. 42, no. 3, July 2017.

[11] R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection, *Dynamic Programming*, Rand Corporation research study. Princeton University Press, 1957.

A. DBSCAN

HDBSCAN is an extension of **Density-based spatial clustering of applications with noise (DBSCAN)** [4], a popular and widely implemented and used clustering algorithm [10]. DBSCAN is based on the notion that clusters are identified by data space regions denser than their surroundings. The algorithm uses two hyperparameters, a distance measure threshold ϵ and a minimum required number of (data) points *minPts*, to decide whether a point forms part of a dense region: A point containing at least *minPts* points within its ϵ-neighborhood according to any consistently applied distance metric is defined to be in a dense region and thus part of a cluster. Conversely, any point not fulfilling this condition is considered a *noise point*. DBSCAN makes a further distinction for points being part of a cluster: Points whose neighbor count (including the starting point itself) exceeds *minPts* are considered *core points* whereas those having exactly *minPts* neighbors within their ϵ-neighborhood are labeled *border points*. Any point being within radius ϵ of a core point is considered *direct density reachable*, and in case this point again is a core point, its ϵ-neighborhood is defined as *density reachable*. Density reachability thus is a transitive property rendering a cluster *density connected* from border point to border point. Being purely density-based, DBSCAN is agnostic to the shape of clusters as well as their number while being robust to noise in the form of outliers. The downside of DBSCAN is that it requires domain knowledge when deciding the hyperparameter ϵ which locks in a fixed cluster density even before the algorithm is run. While these two drawbacks are remedied by HDBSCAN as investigated in this report, both DBSCAN and HDBSCAN are prone to suffer from the *curse of dimensionality* [11] when employing the ubiquitous Euclidean distance measure dealing with high-dimensional data. In this context, *high-dimensional* may come down to as little as three to five dimensions [10]. Alg. 2 renders an abstract high-level algorithmic description of DBSCAN.

Algorithm 2 Density-based clustering of applications with noise

```
1: function DBSCAN($X \in \mathbb{R}^{n \times d}$)
2:     $D \leftarrow$ calculate_distances($X$)
3:     $C \leftarrow$ find_connected_components($X, D, \epsilon$)
4:     $\hat{C} \leftarrow$ assign_border_points($X, C, \epsilon$)
5:     $O \leftarrow$ get_remaining_points($X, \hat{C}$)
6:     return $\hat{C}, O$          ▷ Prominent clusters and outliers
```

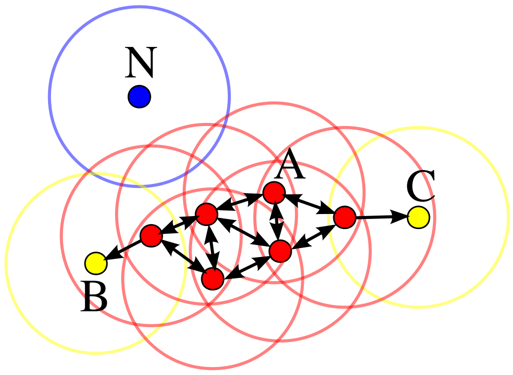

Fig. 12. Illustration of the DBSCAN cluster model: Core point *A*, border points *B* and *C*, noise point *N*; *minPts*=4, all circle radii ϵ [10].