HDBSCAN: FAST(ER) HIERARCHICAL DENSITY BASED CLUSTERING

Tobia Claglüna, Martin Erhart, Alexander Hägele, Tom Lausberg

Department of Computer Science
ETH Zürich, Switzerland

ABSTRACT

We propose an efficient implementation of the hierarchical, density-based clustering algorithm HDBSCAN. We analyze and optimize many aspects of our algorithm with a focus on the distance computation, k-th nearest neighbor search and cache/memory optimizations. Our final implementation is a versatile algorithm which consistently offers 5x speed and performance improvements compared to a similar reference implementation written in C++ and up to 3.5x speedup compared to our own baseline.

1. INTRODUCTION

Clustering algorithms often provide an important first step in arranging and preparing unstructured and unclassified data. A major hurdle in this initial phase of data exploration is that one usually has little prior knowledge about the structure of the data and no intuition about suitable parameters and assumptions. In these circumstances the HDBSCAN algorithm is very well suited.

Density & Hierarchical Clustering. The strengths of HDBSCAN come from its two defining characteristics. The first characteristic, which the algorithm inherits from DBSCAN, is the density-based approach. Whilst often needing more data than centroid-based clustering schemes, density-based schemes make no prior assumptions about the structure of the data and can easily process datasets containing variably shaped clusters. The other main characteristic is its hierarchical nature. This means that the output is not limited to a set of labels but also gives deeper information on the relationship between the different structures. The hierarchical clustering also allows varying density between the clusters and automatic selection of the optimal clusters.

This work. The goal of this project is to create a holistic implementation which offers good performance in native C/C++ regardless of the dataset, parameters or architecture. Our implementation is designed to be a versatile clustering algorithm which can be applied quickly to any dataset without prior inspection of the data.

We achieve this by systematically benchmarking many aspects of the algorithm with the different variables and parameters and designing the optimizations to work effectively on Intel and AMD architectures and multiple datasets.

Related work. HDBSCAN was originally proposed by Campello et al. [1] in 2013. It has become the go to solution to the problems caused by the density resolution parameter defined in the classic DBSCAN algorithm [2]. With a revised paper [3], the authors published an initial Java implementation of the algorithm.

In their work "Accelerated Hierarchical Density Clustering" [4], Healy and McInnes propose a restructured formulation of the algorithm which achieves $\mathcal{O}(n \log n)$ asymptotic complexity. This is done by taking advantage of spatial indexing trees and reformulating the dual tree Boruvka algorithm to use the mutual reachability distance. This implementation is integrated into Scikit-learn [5] and is the most commonly used version of this algorithm.

For this project, we focus on the initial formulation of the algorithm with quadratic runtime as implemented in [6].

2. THE HDBSCAN ALGORITHM

The HDBSCAN algorithm takes a set of points in a d-dimensional space and returns both a hierarchical tree of possible clusters and by default a set of labels corresponding to the optimal selection of these clusters. The algorithm can be split into three main steps: i) the pairwise distance computation, ii) computing the minimal spanning tree (MST), and iii) the cluster extraction.

Mutual reachability distance (d_{m_reach}). Initially, we compute the d_{m_reach} between the points in our data set. To calculate the d_{m_reach}, we first define a core distance as the distance between a point and its k_{th} nearest neighbor. It is defined as $d_{m\_reach}(a, b) = max(core_k(a), core_k(b), d(a, b))$. By calculating the d_{m_reach} instead of the Euclidean distance, we cause dense points to remain close to each other whilst spreading sparse points further apart. The algorithmic complexity of pairwise distance calculations is $\mathcal{O}(n^2)$. This includes the calculation of the core distance, which can be done in $O(n)$ with e.g. quickselect.

MST. After computing the pairwise distance between each point, we then create a weighted graph in which the points from our dataset are represented as nodes and the

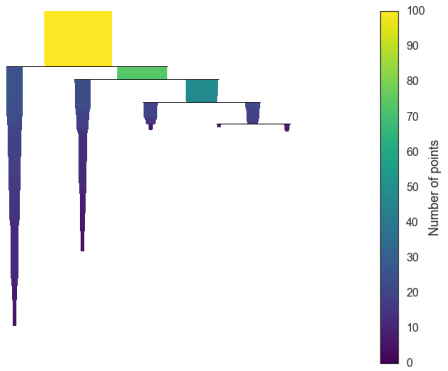

Fig. 1. Example of a hierarchical cluster tree. [8]

pairwise distances are represented as weights of the edges. To efficiently compute the relationship between the densely connected components of the graph, we compute its minimum spanning tree. This can be done using Prim's algorithm. The time complexity of the MST computation is $\mathcal{O}(n^2)$ [7].

Cluster Extraction. Lastly, we use the MST to create a hierarchical cluster tree and extract the clusters. After sorting the edges of the MST in ascending order, HDBSCAN iteratively merges the subcomponents connected by the current edge. This can be efficiently done using a union-find data structure and results in a dendrogram. By defining a cluster to be a subcomponent containing at least m_{pts} points and only tracking the mergers of clusters, we can reduce the dendrogram to a condensed hierarchical cluster tree. Optionally, the hierarchical cluster tree can then be exported to allow for further analysis of the clusters. By default, we take this tree and extract an optimal selection of the clusters. To make this selection, we use a "stability" value as defined by Campello et al. [1] for each cluster. The stability represents the colored area of each cluster in Figure 1. The optimal selection of cluster is given as the set of clusters with the largest cumulative stability under the condition, that no element of the set contains a descendant of itself. Once the optimal selection has been made, the labels for each point in the original dataset is returned as an integer value. The asymptotic complexity of the hierarchical tree construction and cluster extraction is $\mathcal{O}(n)$, as the union-find structure guarantees an amortized constant time *find* and *union* instructions when using the path compression technique.

We define our cost measure as follows:
$C(n) = (Adds(n), Mults(n), Sqrts(n))$.

3. OPTIMIZING HDBSCAN

We now define the different optimizations we perform and explain, in detail, why they are taken. Starting from the baseline implementation, we focus on the computational bulk of the algorithm - the pairwise distance computation - and gain significant speedup through memory optimizations and manual vectorization. As a final step, we introduce a specialized version of the Prim algorithm, coined Prim advanced, that greatly reduces the memory footprint of HDBSCAN while pushing the performance further.

Baseline. We start from a simple implementation in C that we code from scratch, and which closely follows the algorithm outlined above. That is, it loops through all individual points and, for a fixed point, computes i) the distance to every other point and ii) the distance to the k-th nearest point (core distance). The latter is done with the quickselect algorithm. After storing this mutual reachability distance in a matrix of size $n \times n$, it uses the Prim algorithm to obtain the MST of the graph. We then sort the edges in ascending order of weights and extract the clusters through a linear traversal and a Union-Find datastructure.

For all performance critical parts, a suitable code style is maintained and only native C-arrays are used as datastructures. The implementation uses double-precision floating point numbers for all computations.

FUNCTION	Cycles Spent [%]
Pairwise Mutual Reach. Distance	**71.7**
Iterative Quickselect	11.3
Prim Algorithm	5.43
Merge Clusters (Union-Find)	4.96

Table 1. Results of profiling the baseline implementation of HDBSCAN on an AMD Ryzen 7 4800H with $n = 10240$ and $d = 64$.

Distance Matrix. Early profiling[1] of the baseline implementation leads to the clear observation that the majority of time is spent on computing the pairwise distance; an example of which can be seen in Table 1. This majority, accounting to more than 70% of cycles, is therefore the main point of our attention for the first optimizations.

A natural step is to apply blocking to the computation of the $n \times n$ distance matrix, which results in a better cache locality and improves performance.

Moreover, we can exploit the symmetric nature of the distance computation. In particular, we implement a special triangular matrix of size $\frac{n(n+1)}{2}$ laid out linearly in memory. We thereby save not only half of the computations but halve the memory requirement. However, as we see in our experiments in Section 4, this leads to a bad, irregular access pattern that strongly limits the performance.

Vectorizing Euclidean Distance. We start with a simple scalar implementation to compute the Euclidean distance of two points.

[1] Profiling done with perf (`https://perf.wiki.kernel.org`) and Hotspot for Linux (`https://github.com/KDAB/hotspot`)

Analyzing the generated assembly code of a function allows to assess the quality of the optimizations done by the compiler and to decide whether better performance can be achieved by manually vectorizing it or restructuring the C code to enable further optimizations. All the observations and optimizations regarding distance calculations that follow were performed on an Intel i5 7300U CPU based on the Skylake microarchitecture.

Looking at the assembly produced by GCC 11.1 with flags "-O3 -march=native -ffast-math", we observe that the compiler is not able to unroll the loop and use multiple accumulators to better fill the processor pipeline. However, it takes advantage of fused-multiply-add-operations. Therefore, we manually unrolled it by a factor of 16 and use AVX2 intrinsics to have four vectorized accumulators. Manually vectorizing this distance function allows us to precisely implement and assess different accumulation and reduction variations, finally settling with the structure shown in Figure 2.

We need to calculate $(a_i - b_i)^2$ for every index i of the two points a and b and sum them up. We use a subtraction operation and a FMA (fused-multiply-add) for every index to calculate the square and add it to the total sum at the same time. This means we continuously occupy one port of the CPU with an FMA and the other with the subtraction by using four accumulators, as each of these instructions has a latency of four cycles and one of each can be issued every cycle. Once all indices are processed and summed up in the accumulators, we add them together in a tree of additions resulting in a vector register with four double-precision FP values. To reduce this to a scalar, we extract the upper part of the register and add it to the lower part, then we permute the two values in the lower part and add it to the non-permuted value. As a result, the scalar register now contains the summed up value of which we can calculate the square root as the last step.

Specializing for some dimensions. By writing a generic version for vectorized distance computations in a d dimensional Euclidean space, we are not able to reach a performance which is close to the theoretical maximum. Additionally, the general implementation uses a 16 times unrolled loop and thus dimensions $d < 16$ do not benefit from this optimization but instead use the scalar loop used to compute any remaining elements.

We therefore create specialized versions of the distance-function for $d = 2$ and $d = 4$. With these dimensions we are able to reduce the number of floating-point operations and runtime and increase performance. In the general implementation we initialize the accumulator to zero and then add the squared difference for each index. In the specialized version we save the useless addition with zero and save even more additions because the general implementation uses a simple scalar loop for these low dimensions ($d < 16$) and thus the vector registers are not fully utilized.

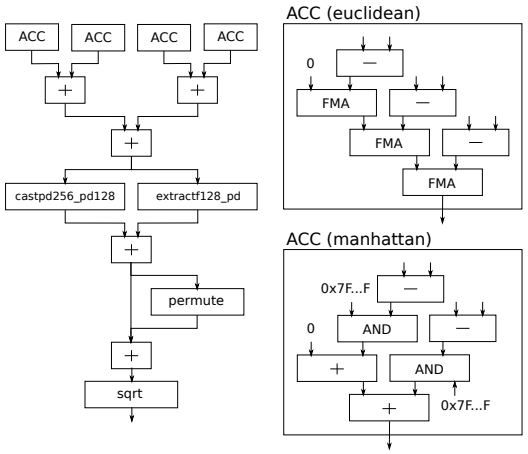

Fig. 2. Structure of the vectorized Euclidean and Manhattan distance calculation. The number of stages in the ACC modules are implemented using a loop and depend on the number of dimensions d.

For $d = 4$ we calculate the difference of one point to four other points and multiply the result with itself (to compute the square), then a series of permutes and additions sum up these squared differences of the four indices for each point, such that they end up in sequence in a single vector register and the square root operation can be performed on all four points at the same time (vectorized) instead of the scalar square root used in the general implementation. This, however, requires the number of points n to be divisible by four. A precise schematic of the additions and permutions is shown in Figure 3. In total, we require six blends and six permutes which are quite cheap on Skylake (especially the blends) and two more expensive cross-lane permutes.

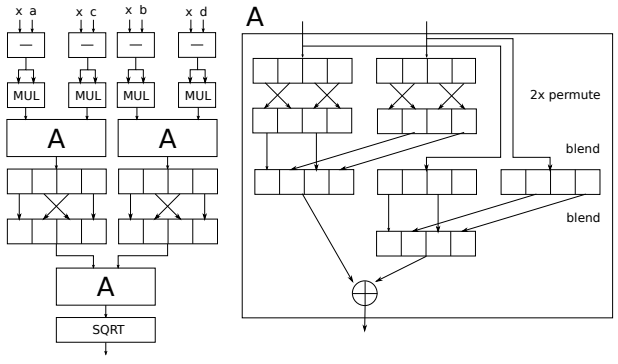

Fig. 3. Structure of the specialized vectorization of Euclidean distance for four dimensions.

Manhattan distance. In addition to Euclidean distance, we also implemented and optimized Manhattan distance which

can be toggled via a *CMake* flag using the preprocessor for conditional compilation.

The general implementation that works for all dimensions d is implemented very similarly to the Euclidean distance. Only the operations in the accumulator are different. As shown in Figure 2, it uses a bitwise AND to unset the sign bit and then only requires a simple addition instead of an FMA as we do not have to calculate the square.

Furthermore, we also implemented specializations for $d \in \{2, 4\}$. During the optimization process we designed and benchmarked different structures with different instruction mixes. The one we present here is different from the Euclidean specializations (although we also implemented this structure for Euclidean distance) as they use horizontal addition operations instead of blends, permutes, and regular additions. The detailed structure is shown in Figure 4.

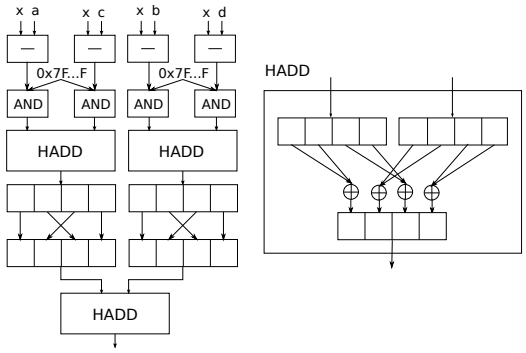

Fig. 4. Structure of the specialized vectorization of Manhattan distance for four dimensions.

Quickselect & Partitioning. As laid out above, HDBSCAN uses the *mutual reachability distance*. This requires us to compute the kth-nearest-neighbors (parameter k) of the two points for which we are calculating the distance.

There exist several algorithms with different trade-offs. We decide to start with a standard iterative quickselect as its runtime does not depend on k. We then introduce a vectorized version of the same algorithm using AVX2 intrinsics, making it more performant.

The partition step of quickselect consists of comparison operations and data movement depending on the comparison result. We vectorize this comparison and move all four compared values at the same time by filtering out the elements smaller than the pivot and compressing them such that they are stored in sequence without any gaps. The same is done for elements bigger than the pivot. The compression could be done using a single AVX-512 operation, but as we are restricted to AVX2 we implement that functionality using `movemask`, permutes, and a look-up-table or `pdep/pext`.

Moreover, depending on the choice of the parameter k, we propose a different algorithm, which we call *bubbleselect*. The idea is based on simply using an array of size k that stores the k smallest values found in the pairwise distance so far. When traversing through the distances for a particular point, we insert new values in the same fashion as bubblesort. This way, as we demonstrate in Section 4, we speed up the computation in particular for small values of k which are more commonly used in the HDBSCAN algorithm.

We do not further optimize this *bubbleselect* algorithm as it consists mostly of branches with complex conditions and only scalar value operations that are not applied to many values in sequence (if k is small).

Vectorizing Prim. The basic prim algorithm is almost fully vectorizable. Apart from the obvious vectorizations for the initialization step before looping through all edges, the bottleneck lies in the expensive comparison operations needed to determine which points could reduce the cost of the current edge and the subsequent cost-updating of the nodes not yet contained in the MST. Luckily though, these two for-loops can be reformulated as reduction operations which are then vectorizable with masking and comparison intrinsics.

Algorithm 1: Prim Advanced

Data: Data $X \in \mathbb{R}^{n,d}$, Core Distances $C \in \mathbb{R}^n$, Distance Function dist()
Result: MST of Mutual Reachability Graph for X

for $i = 1, ..., n$ **do**
  $d_{min,i} = \infty$;
end
start with random point in MST;
while *# edges* $\neq (n-1)$ **do**
  let i be the last added point;
  init e_{min};
  for j *not yet in MST* **do**
    d = dist(X_i, X_j);
    m = max(d, C_i, C_j);
    $d_{min,j} = \min(m, d_{min,j})$;
    if $d_{min,j}$ *is minimal* **then**
      update e_{min};
    end
  end
  add e_{min} to MST;
end

Advanced Prim. While the optimizations to the basic algorithm already obtain major improvements, the limitation of the algorithm remains its memory footprint. In particular, it heavily limits the input size as the distance matrix quickly reaches multiple Gigabytes of memory.

However, computing the full distance matrix is not needed. Instead, we make use of the special structure of the mutual reachability graph and propose a specialized algorithm for

the MST computation, called Prim Advanced. Campello et al. [3] suggest a similar approach. Algorithm 1 shows the gist of this algorithm.

The main idea is to not fully compute the pairwise distances and store the entire mutual reachability matrix, but to compute the distances on the fly in the search for the next edge of the MST. The graph is implicitly given by the input data and the core distances, which still have to be computed beforehand. Note that this structural change of the algorithm leads to roughly twice the flop count overall.

We again apply the vectorizations mentioned above to improve the runtime and performance. In particular, we use the SIMD implementations for the distance computation and the k-th nearest neighbor search.

4. EXPERIMENTAL RESULTS

In this section we are going to evaluate the different optimizations described in the previous section in terms of their performance and runtime.

Experimental setup. The experiments are carried out on an AMD Ryzen 7 4800H, 2.9 GHz with 2x4MB LL-Cache. The code is compiled with Clang (version 11.1) with compiler flags "-O3 -march=native" which already enables auto-vectorization wherever possible[2]. Unless otherwise stated, the reported speedups for the experiments are with respect to a fixed dataset consisting of circular blobs of 14336 data-points with 64 features each, generated at random by the Scikit-learn library [5]. All our performance and runtime plots are done with respect to such datasets (with varying size n) that are created once and used for benchmarking all different versions. Please note that we experimented with different randomly created datasets and did not see a large variance in performance or runtime.

Additionally, we analyze the impact of switching the compiler (GCC-11.1) as well as the possible shortcomings and advantages of changing to another processor architecture (Intel i5-7300U, 2.6 GHz with 3MB LL-Cache).

All benchmarking is done using the RDTSC[3] tool for counting cycles. Floating point operations are measured using the Perf API [4] to dynamically count raw events measured by the CPU. The algorithm is run at least 3 times or 10^8 cycles and the metrics are then averaged out.

Baseline and reference. To evaluate the quality of our implementation, we directly compare it to a reference implementation of the algorithm written entirely in C++ [6]. It must be mentioned that our implementation also uses C++ for the code infrastructure and non-performance-critical sections. The parts of the algorithm referred to in the following section, however, are written entirely in C. We provide an overview of all major versions of the algorithm, including the reference, in Figure 5.

[2]Please note that experiments with the "-ffast-math" flag result in incorrect computations for our implementation, therefore we do not use it.

[3]en.wikipedia.org/wiki/Time_Stamp_Counter

[4]www.man7.org/linux/man-pages/man2/perf_event_open.2.html

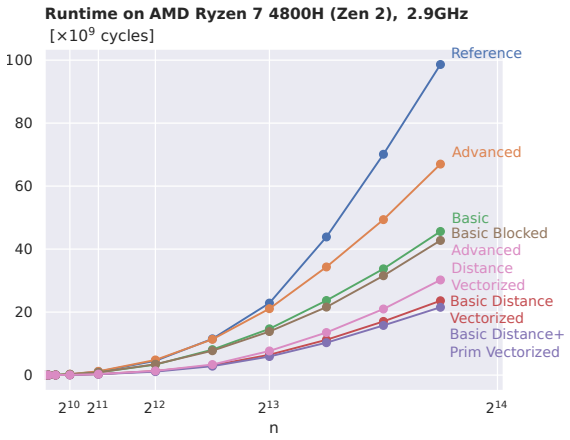

Fig. 5. Runtime of the major versions of the algorithm, including the reference implementation, in cycles. Note that the x-axis is linear.

Memory optimizations. Introducing blocking to the distance matrix computation directly results in a better cache locality and thus fewer cache misses (e.g. the cache miss-rate changes from 6.0% to 2.0%, see Table 2). Unfortunately, the block size remains highly hardware and dataset dependent, whereas the achieved speedup stays moderately low at 1.3x (see Figure 5).

OPTIMIZATION	miss-rate [%]	reads [$\times 10^9$]	writes [$\times 10^9$]
-	**6.0**	**15.4**	**0.6**
blocked	2.0	15.0	0.5
triangular	6.7	8.4	0.6
distance AVX2	18.7	4.7	0.4
distance+prim AVX2	13.6	4.5	0.5

Table 2. Cache analysis for the basic algorithm, with SIMD optimizations in the last three rows; made with Cachegrind [9] on a dataset with $n = 10240$ and $d = 64$.

We see that computing only the upper triangular distance-matrix reduces the number of expensive writes to DRAM. It should not come as a surprise, though, that the speedup achieved through this optimization is negligible and for large n even deteriorates the runtime. This is due to the inherently irregular access pattern when, instead of recalculating the distances, the previous results are read in a sequence that is not linear with respect to the memory layout which doesn't favor cache reuse.

Vectorizing Distance Calculations. Introducing manual vectorization of the distance computation is algorithm agnostic and thus has a direct impact on both the basic and advanced Prim implementation. We are able to reduce the number of reads from DRAM by a factor of 3, as can be

seen in the last three rows in Table 2. We strive to only load and store vectors at the beginning and the end of the computation, therefore keeping the data in vector registers for as long as possible. This effort, combined with the higher performance of vector operations, results in a speedup of maximally 3.5 for Euclidean distance, as shown for the basic algorithm in Figure 6 (orange line).

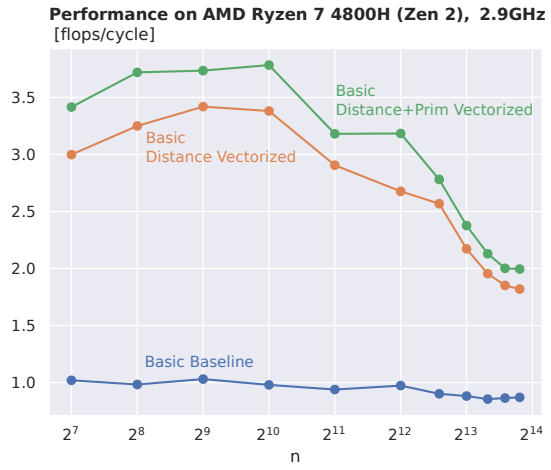

Fig. 6. Performance of our manually vectorized basic algorithm using Euclidean distance.

Specializing for some dimensions. Writing specialized versions of the distance computation for $d \in \{2, 4\}$ turns out to give good advantage over the auto-vectorized code, underpinning our hypothesis that merging the intermediate results during the reduction of multiple points and thus using the vector operations more efficiently leads to speedup. Measurements for both Euclidean and Manhattan distance are shown in Figure 7.

In practice we achieve a speedup in runtime for $d = 4$ of about 1.24 for Euclidean distance and 1.27 for Manhattan distance on the AMD Ryzen 7. Measurements on the Intel i5 result in speedups of 1.38 for Euclidean distance and 1.37 for Manhattan distance, indicating that the different reduction strategies lead to very minor runtime differences and, furthermore, heavily depend on the microarchitecture of the used CPU (e.g., *Zen 2* has worse gap for blend operations, but the same gap and latency for horizontal addition compared to *Skylake*).

When consulting the lines in Figure 7 one can observe an interesting peak at $d = 8$. As we described in Section 3, we perform a 16 fold loop unrolling in the general version, but since this cannot be used for $d = 8$, the code has to fall back to the scalar loop. Hypothetically, it would also be advantageous to create a custom version for $d = 8$ to avoid this spike in runtime. For $d = 16$ or higher we observe increased performance due to increased instruction-level parallelism leveraged by the four accumulators described in Section 3.

A negative impact from the if-condition choosing the appropriate distance function at runtime cannot be found in Figure 7.

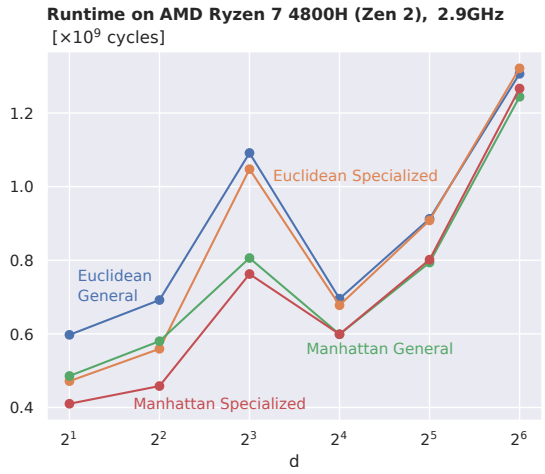

Fig. 7. Runtime comparison of the specialized and non-specialized versions for both Euclidean and Manhattan distance. Note that the specialized version uses if-statements to select the appropriate specialization at runtime, while the non-specialized version does not have these conditionals at all. The input size is $n = 4096$.

Quickselect & Partitioning. When comparing the performance of the baseline and optimized iterative quickselect as well as the *bubbleselect* (as shown by the AMD lines in Figure 8), we can clearly see the trade-off mentioned in Section 3. Depending on the parameter k, either the optimized quickselect or the *bubbleselect* perform better. The intersection is at about $k = 40$ which is already larger than used in most cases. This makes *bubbleselect* the better choice on *Zen 2*. Note that *bubbleselect* will perform worse than the baseline quickselect starting at some large k, but this is too large to be relevant.

Interestingly, on the Intel i5 the optimized quickselect always performs better than *bubbleselect* as shown by the Intel lines in Figure 8. This means, not only certain optimizations should be tailored to the specific platform the code is running on, but entirely different algorithms should be considered as well. However, here, the baseline quickselect also performs a lot worse than the optimized version.

Advanced Prim. When comparing the achieved runtime of the advanced Prim algorithm to the basic one (see Figure 5), it becomes apparent that it does not scale well due to the increased flop count. Thus for smaller n and large d the basic algorithm still has the advantage that its distance matrix might still fit into cache and is not limited by the memory bandwidth. As can be seen in Figure 5, this is exactly what seems to be the case since the chosen $d = 64$ is quite large. Although, there will be a turning point where

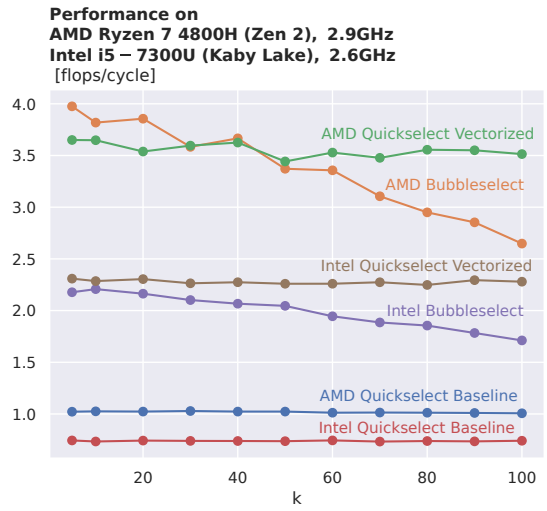

Fig. 8. Performance of HDBSCAN advanced for varying k in the core distance computation. We compare the standard iterative quickselect, its vectorized version and a simple version called bubbleselect. These algorithms do not change the flop count as they do not contain any operations in our cost measure. The input size is $n = 4096$ with $d = 64$.

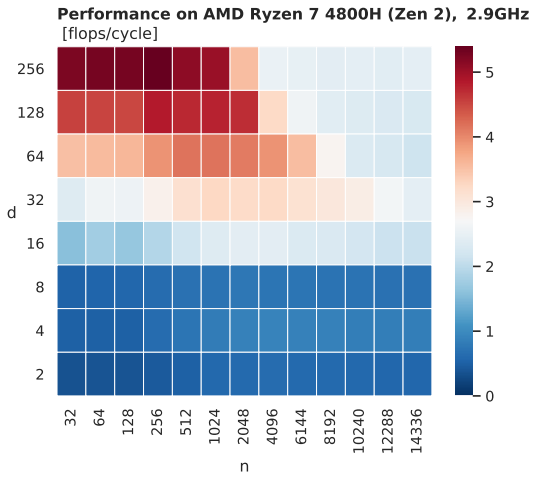

Fig. 9. Heatmap showing the achieved performance for varying n and d with the vectorized Advanced Prim implementation.

computing the distances twice is cheaper as it is not bandwidth limited. This is especially the case for large n and small d. Interestingly, the effect of the optimizations converge to similar performance for large n. One can hypothesize that this could be attributed to the LL-cache size not being large enough to serve the requests and thus the frequent data movement starts to govern the achieved performance (8MB is close to 7.4MB ≈ 14366 doubles $\times\ 64 \times 8\frac{\text{byte}}{\text{double}}$).

Vectorized Prim. We were able to almost fully vectorize the basic Prim algorithm, as the main loop essentially just contains comparison reductions. In Figure 6 we see that it is possible to further boost the performance on top of the already vectorized distance computations. Unfortunately, this cannot be asserted for vectorizing the advanced Prim algorithm, as this results in drastically decreased performance. We found this to be due to the irregular access pattern with which one selects the data points not yet contained in the MST in the for-loop at line 8 (Algorithm 1). It poses a major bottleneck within the algorithm as it also prevents the compiler to apply additional optimizations.

Varying dimensions. Changing the dimension d of the input dataset can dramatically change the achieved performance, similarly does changing the number of datapoints n. An illustration of this is shown as a heatmap of achieved performance for the vectorized variant of the advanced prim algorithm in Figure 9.

The number of flops increase linearly with the number of dimensions, but higher dimensions allow for efficient access to the coordinates of each point. In particular, our vectorized code boosts the performance by exploiting the instruction-level parallelism of the distance computation. At the peak, we reach roughly 5.5 flop/cycle whereas for smaller dimensions we have less than 1 flop/cycle. It is also nicely visible how the performance is best for large d and a small n (upper left corner), where data fits into cache, and how increasing n limits the performance (upper right corner) by exceeding the cache.

AMD vs. Intel. While vectorizing our quickselect implementation, we found some interesting differences between Intel's Skylake architecture and AMD's Zen 2. In particular, we observed that the hardware implementation of the `pdep` and `pext` instructions of the BMI2 extension are implemented natively in Skylake and also the newer Zen 3 architecture, but it is done via microcodes in Zen 2 [10, 11]. This results in a very high latency and a gap of at least 19, respectively, for these instructions. However, it might even become as high as 300 cycles in latency, depending on the input[5]. This is clearly visible in the performance plot in Figure 10, where the `pdep`/`pext` instructions greatly decrease the performance on the AMD system while being fast on Intel.

We tried to avoid these instructions by implementing a vectorized quicksort with a specialized look-up-table that acts as a proxy for the two instructions. This resulted in a speedup not only on AMD but also Intel, with AMD even beating the Intel system for large n.

Clang vs GCC. The choice of the compiler can greatly impact the performance a code is able to reach and how well it scales with increasing input size. This is why we want to

[5]There are many discussions online, e.g. `https://twitter.com/InstLatX64/status/1209095219087585281`

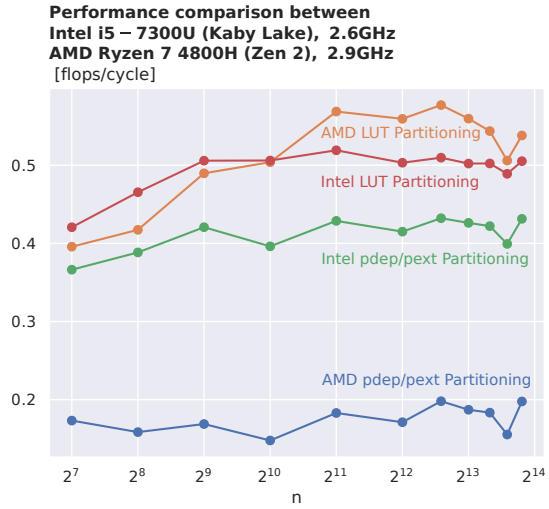

Fig. 10. Comparison of the performance with BMI2 extension and our look-up table (LUT) approach. Note that the two algorithms perform the same number of flops as they do not contain any operations in our cost measure. Here $d = 4$.

analyze if we can find a substantial difference for our implementations when compiled with GCC instead of Clang. We test this for both Prim algorithms compiled with the aforementioned compilers in terms of its achieved performance. Both compilers follow a similar trend for both algorithms with negligible differences for the basic algorithm, but for the advanced algorithm we observe constantly worse performance if compiled with GCC. For certain n this discrepancy even grows up to almost 0.5 flop/cycle.

Final overview. With Figure 5 we summarize the major optimizatons and their delivered speedup. HDBSCAN with vectorized distance computations combined with the vectorized basic Prim algorithm results in a final speedup of up to 5x w.r.t. to the reference and 3.5x w.r.t our own baseline.

In order to evaluate our optimized algorithm w.r.t. the best possible performance, we can make use of the roofline model. Figure 11 shows a roofline plot with all of our major optimizations. We observe three clusters in the plot. The reference implementation has far worse operational intensity and performance even compared to our baseline implementations. The unoptimized basic and advanced prim implementations as well as the blocking optimized version have roughly the same operational intensity - still within the memory bound region - and a performance of about one flop/cycle. The most optimized basic and advanced prim implementations occupy about the same spot at a little over two flops/cycle and are well outside the memory-bound region.

It is important to note that major parts of the algorithm (quickselect, bubbleselect, and Prim) primarily depend on comparison operations. Unfortunately, they cannot be counted through the Perf API and thus require manual instrumentation for each occurrence. Because of that and the fact that we first focused on the distance computation, they are not included in our cost measure. We believe that if we were to incorporate them, we would both get a higher performance and gain a better understanding of the speedup limit we have reached.

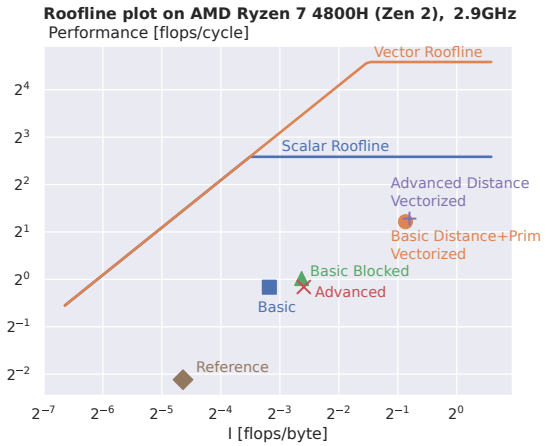

Fig. 11. Roofline showing all our major optimizations as well as the baseline and reference implementation for $n = 10240$ and $d = 64$.

5. CONCLUSION

In the process of optimizing the HDBSCAN algorithm we came upon many interesting paths of improving our baseline implementation. We achieved a total speedup of up to 3.5x w.r.t our own baseline and 5x w.r.t. the reference implementation.

Due to the high data dependence of our algorithm, further progress to the theoretical performance limit has no simple route. Nonetheless, possible future optimization paths include introducing fine-grained blocking to the distance computation for large d (e.g., $d = 64$), more detailed cache-analyses (e.g., more detailed analysis and reduction of conflict misses) and a more extensive cost-measure to also capture the number of comparisons.

Hypothetically, HDBSCAN would allow for more freedom of optimization if certain parameters (n, d, k) were fixed. By maintaining the variability of our parameters, we contend with the fact that many optimizations to certain parameters, do not translate or lead to slowdowns in others. Regrettably, the concretization of HDBSCAN is not applicable to the general use case of exploratory data analysis. Nevertheless, our final implementation offers a well-rounded package which ensures high performance for the two systems and the many configurations discussed in our work.

6. CONTRIBUTIONS OF TEAM MEMBERS

We shortly summarize the contribution of each in our group.

Tobia. Focused on SIMD optimizations for both basic and advanced Prim algorithm. In charge of doing cache analysis of all optimizations considered, as well as creating and evaluating Roofline plots. Created the initial benchmarking script which is being called at every program execution.

Martin. Implemented and optimized all distance calculations. This includes both Euclidean and Manhattan distance and their manual vectorizations for general dimensions and specializations for $d \in \{2, 4\}$.

Implemented quickselect and its initial vectorization (the one with pdep/pext presented in AMD vs. Intel), and helped Alexander to optimize it for AMD.

Various changes and additions to the benchmark script.

C++ benchmarking utilities including functions to measure cycles, flops (perf event API and manual instrumentation for *sqrt*), and performance.

Intel benchmarks were done on his machine.

Alexander. Designed the advanced variant of the Prim algorithm and also the triangular distance. Worked together with Martin on the SIMD optimization of quickselect and partitioning, in particular coded the AMD specialized version and bubbleselect.

Coded initial benchmarking script and various additions, as well as code for analyzing the reference implementation.

Analysis of performance for varying dimensions (heatmap). All benchmarking, analyses of AMD done on his machine.

Tom. Implemented the dendrogram construction, hierarchical cluster tree condensation algorithm and the optimal cluster extraction. Optimized hierarchical cluster tree creation by extending the efficient C based union-find structure used to create the dendrogram to simultaneously track cluster creation and merging using more high-level C++ functionalities.

Implemented cache optimized computation of the distance matrix using blocking.

7. REFERENCES

[1] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander, "Density-based clustering based on hierarchical density estimates," in *Advances in Knowledge Discovery and Data Mining*, Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, Eds., Berlin, Heidelberg, 2013, pp. 160–172, Springer Berlin Heidelberg.

[2] 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.

[3] 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.

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

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] Rohan Mohapatra and Basarkod Sumedh, "HDBSCAN-CPP, fast and efficient implementation of HDBSCAN in C++ using STL," `https://github.com/rohanmohapatra/hdbscan-cpp`, 2019.

[7] Ashish K. Nemani and Ravindra K. Ahuja, *Minimum Spanning Trees*, American Cancer Society, 2011.

[8] Steve Astels Leland McInnes, John Healy, "How hdbscan works," `https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html`, 2016.

[9] Nicholas Nethercote and Julian Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," 06 2007, vol. 42, pp. 89–100.

[10] Andreas Abel and Jan Reineke, "Uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2019, ASPLOS '19, p. 673–686, Association for Computing Machinery.

[11] Agner Fog et al., "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," *Copenhagen University College of Engineering*, vol. 93, pp. 110, 2011.