IMPROVING LOCAL OUTLIER FACTOR COMPUTATION USING MATRIX-MATRIX MULTIPLICATION AND HEURISTIC SEARCH

Utkarsh Bajpai   Florin Vasluianu   Razvan Pasca   Anastasia Sycheva

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

In this paper we introduce two new, single core C implementations of the density based local outlier factor algorithm. The first thing we exploit is the fact that distances are symmetric to reduce the computation burden by half. Then, the performance is further improved by reformulating the euclidean distance using matrix-matrix multiplication. Using this we obtain comparable runtimes to the baseline implementation which uses high performance, professionally written linear algebra libraries. On the other hand we introduce a heuristic approach which for a small decrease in precision manages to beat the baseline in runtime.

1. INTRODUCTION

Today's applications from all domains of activity and research have been affected by the big data revolution. These applications are usually formed by multiple stages of transformations of the input data. One of those is outlier detection and removal which can be an important preprocessing step that significantly impacts the final performance of the algorithm. On the other hand it can be an application in itself in the form of anomaly detection, used for example to detect potentially fraudulent banking transactions.

The local outlier factor (LOF) based algorithm [1] is a popular, density based approach used to identify the outliers in the dataset by assigning a score which is in itself a more refined approach than classical, global methods which often fail to find relevant outliers as shown in [1]. Moreover, it is considerably easier to tune than other similar algorithms because it has just 1 hyperparameter.

Improving the performance of such methods is essential because a faster preprocessing stage for modern, iteration-based big data application provides more opportunities for the developers to further improve the application in the same amount of time. In extreme cases owed to the volume of the data, it can make previously impossible tasks possible. On the other hand, certain applications operate on close to real-time e.g. trying to identify fraudulent ATM transactions with minimum downtime naturally require the highest performance and lowest runtime attainable.

Writing high performance code is a fairly difficult task nowadays which requires solid knowledge of the full computer science stack: algorithms, compilers and operating hardware. Making use of those in order to efficiently exploit the CPU resources includes fine tuning for architecture specific memory-hierarchy to minimize the trips to the main memory, vectorizing operations using dedicated circuitry, reordering and rephrasing operations and parallelizing the computation on multiple cores.

This paper presents two new, single core oriented C based implementations using the single core optimization techniques presented above. We provide two variants of the algorithm. The first one gives correct results up to double precision rounding error with respect to the reference SciKit implementation while closely following it in runtime. The second one uses a novel heuristic search which trades some precision for a lower runtime managing to actually beat the reference implementation.

The reference SciKit implementation [2] uses the Numpy library which is a Python wrapper around high performance, professional BLAS implementations which can be changed by the user. By default it uses the open source OpenBLAS [3] implementation.

2. ALGORITHM DESCRIPTION

The method is based on computing statistics in the neighborhood of each point, such as the *local reachability density score* and *local outlier factor score*. The last one is a value which is then used to asses whether the point represent an outlier or not in the dataset. The LOF algorithm receives as input an $n \times dim$ matrix where n represents the number of points and dim the point dimensionality and k, the number of points based on which to compute the local statistics. The

Function	$W(n)$	$Q(n)$	$I(n)$	$T(n)$	$P(n)$
PairDist	$n \cdot \frac{n}{2}(4 \cdot dim + 1)$	$8(n^2 + n \cdot dim)$	$\frac{dim}{4}$	$\frac{1}{2} n \cdot n \cdot (lat^{sqrt} + dim(3\frac{lat^{add}}{thr^{add}} + \frac{lat^{mul}}{thr^{mul}}))$	$\frac{4 \cdot dim+1}{lat^{sqrt}+dim(3\frac{lat^{add}}{thr^{add}}+\frac{lat^{mul}}{thr^{mul}})}$
K-Dist	$n(n-1)\log(n-1)$	$8(n^2 + n)$	$O(\log(n))$	$n(n-1)\log(n-1)\frac{lat^{comp}}{thr^{comp}}$	$\frac{thr^{comp}}{lat^{comp}}$
K-Nhood	$n \times (n-1)$	$10n^2 + 8n$	$\frac{1}{10}$	$n \cdot (n-1)\frac{lat^{comp}}{thr^{comp}}$	$\frac{thr^{comp}}{lat^{comp}}$
ReachDist	n^2	$8(n^2 + 2n)$	$\frac{1}{8}$	$n^2 \frac{lat^{comp}}{thr^{comp}}$	$\frac{thr^{comp}}{lat^{comp}}$
LRD	$n(k+2)$	$2nk + 16n$	$\frac{k+2}{k+8} \approx 1$	$n \cdot (2 \cdot lat^{div} + k\frac{lat^{add}}{th^{add}})$	$\frac{k+2}{k\frac{lat^{add}}{th^{add}}+2 \cdot lat^{div}}$
LOF	$n(k+2)$	$2nk + 16n$	$\frac{k+2}{k+8} \approx 1$	$n \cdot (2 \cdot lat^{div} + k\frac{lat^{add}}{th^{add}})$	$\frac{k+2}{k\frac{lat^{add}}{th^{add}}+2 \cdot lat^{div}}$

Table 1. Cost and runtime of the main functions. $lat^{operation}, th^{operation}$ correspond to *operation* lattency and throughput.

computation can be split in several steps, as follows:

Computing pairwise distances (PairDist). This step requires computing for each input point the distance to all the other points in the dataset. The metric which was used in this project was L2.

Computing K Distance (K-Dist). This step requires computing the *k-distance* of an input point *p* with respect to the others. It is defined as the distance to the kth farthest object from our point *p*.

Computing K Neighborhood (K-Nhood). This step requires computing all the *k* points whose distance to the reference point *p* is $\leq$ the *k-distance* of *p*. The above 3 steps combined can be seen as the classical K Nearest Neighbors method.

Computing Reachability Distance (ReachDist). This step requires computing the pairwise reachability distance between any 2 points *p* and *o*. It is defined as the maximum between the k-distance of *o* and the actual distance between *o* and *p*. It is important to note that this value is not symmetric.

Computing Local Reachibility Density (LRD). This step requires computing the local reachability density for each point *p*. It is defined as

$$lrd(p) = \frac{K}{\sum\limits_{o \in K-Nhood(p)} reach-distance(p,o)} \tag{1}$$

where K is the number of neighbors fed as input to the algorithm. This gives a rough idea of how sparse/dense the volume defined by the K-Nhood of *p* is and enables computing values which summarize the local dataset characteristics.

Computing Local Outlier Score (LOF). This step requires computing the lof score for each point *p* in the dataset, which is then used to asses whether that point is an outlier. It is defined as

$$lof(p) = \frac{\sum\limits_{o \in K-Nhood(p)} \frac{lrd(o)}{lrd(p)}}{K} \tag{2}$$

Intuitively, the lower the density of *p* becomes compared to its neighbors, the ”more” of an outlier it becomes.

2.1. Cost Analysis

We have performed the cost analysis to guide our subsequent optimization efforts after the initial baseline implementation. The majority of our floating point computations were additions and comparisons, followed by multiplications, divisions and square root computations. Apart from the **K-Dist** function, which uses QSORT as a subroutine, the exact number of operations can be exactly determined by counting. Table 1 contains the results of the cost and run time analysis.

We observe that the first four functions constitute the computational bottleneck with number of flops exceeding $O(n^2)$. Last two functions present a different kind of challenge: random memory access pattern which is a big performance killer. To address them we have developed several different pipelines that will be presented in the next section.

3. PIPELINES

While the *local reachability density score* is computed as a reduction operation using the $k-distance$ of the points in the k neighborhood of a local point, the *local outlier factor score* computation is also a reduction over the *lrd* score for the k-nearest neighbors of a fixed point. This structure of the algorithm yields the necessity to implement a pipeline structure, where we require all the input results to be computed before moving to the next phase, incurring extra memory trips.

3.1. Baseline pipeline

For the baseline pipeline, the algorithm starts by computing the pairwise distances between the points in the input set. The distances are stored in a n^2 sized double precision floating point matrix, and, because the matrix is symmetric, only half of the memory locations are going to be accessed. The pairwise distance matrix is going to be used to compute the k-neighborhood of each point, and, as the k-distance is defined as the distance to it’s k nearest neighbour, it will be

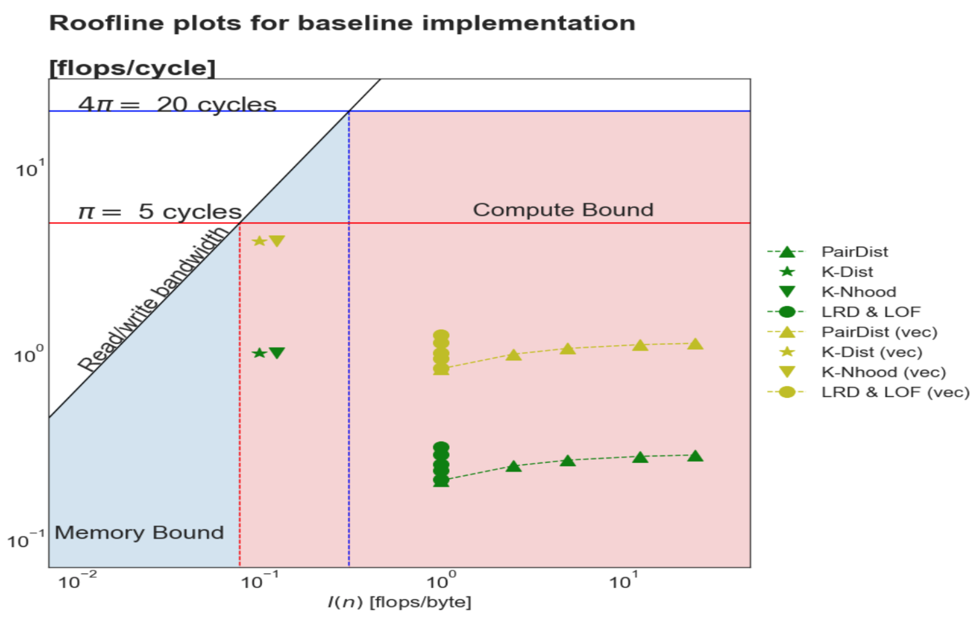

Fig. 1. Roofline plots for the subroutines from the baseline pipeline.

computed as the k'th double precision value in a $n-1$ size sorted array, representing the pairwise distances to all of the other points. Using this distance, the indices of the points in the neighborhood will be computed by a $O(n)$ brute-force search and will be kept in a $n \times k$ sized integer matrix. The k-neighborhood index matrix will be used then to compute the *local reachability density score*, and, finally, the *local outlier factor score* will be produced.

3.2. Merged KNN pipeline

As it can be easily seen in the description of the baseline approach (subsection 3.1), the computation of the pairwise distances will produce a $O(n^2)$ double precision floating point matrix, and only a half of it is going to be accessed. This in highly sub-optimal in terms of memory footprint and cache accessing mechanics.

The *KNN*[4] algorithm will produce the $k-distance$ of a point and the indices of it's neighbors by method's definition, and so, this pipeline merges the first steps of the algorithms into the *KNN* function. The further steps described in the subsection 3.1 will be kept the same.

The merging of the computation of the k-distance and the indexing table for the neighboring points indices will imply the sorting of $n-1$ tuple structures containing the index of the neighbor and it's characteristic distance. As known, the expensive part of the sorting is given by the the swaps done between different elements of the array sorted, and so, the necessity of swapping also the indices of the points (using the packed record structure or different arrays) will induce additional runtime, in an amount that was investigated. A trade-off between an elegant computation and a brute-force searching procedure for the indices of the neighbors had to be addressed, and so, as the ordering of the neighbors does not matter in the next computations, the searching procedure seemed to be a better option.

The diversity of the tabular representations of the partial results (distances to the neighbors and indices) will imply a chaotic memory accessing behaviour, and so, hashing possibilities for array-like representations were proposed. The computation "on the fly" of the distances between a pair of points was sub-optimal, as intense computation was introduced for the hashing procedure used in the results representation. As described, both in the section 2 and subsection 3.1, the computation of the reduction operations over the local neighborhood of each point in the input dataset, will be conditioned on the k-neighborhood index table computed in the previous step. As the k-neighborhood index table depends on the distribution of the points in the dataset, a random accessing pattern will characterize the memory accessing pattern during the later stages of the algorithm (computation of the *lrd* and *lof*). The impossibility of benefiting from the cache memory implies the necessity of exploring the representation of partial results in memory contiguous structures.

3.3. Exploiting the Euclidean distance property

As described in both subsections 3.1, and 3.2, the hashing of the pairwise distances in a memory friendly array-like structure will imply index computations that will not be accounted in the measurement of the overall performance. If some performance enhancement was possible, the first step of the pipeline described in the subsection 3.1 was to be updated.

For the particular case of the Euclidean distance characterizing a point pair (a, b) , it is equivalent to the L2-norm of the difference vector $\vec{a} - \vec{b}$, computed between the representations of the points a and b.

$$d_e(a,b) = \left|\left|\vec{a} - \vec{b}\right|\right|_2 = \sqrt{(\vec{a} - \vec{b})^T(\vec{a} - \vec{b})} \qquad (3)$$

By exploiting the property introduced in the equation 3, the computation of the pairwise distance matrix can be formulated as a matrix-matrix multiplication between the tabular representation of the input point set and it's transpose, formalized below.

$$d_e(a,b) = \left|\left|\vec{a} - \vec{b}\right|\right|_2 = \sqrt{\vec{a}^T\vec{a} - 2\vec{a}^T\vec{b} + \vec{b}^T\vec{b}} \qquad (4)$$

This will enable all the optimizations introduced for the MMM algorithm, such as cache level blocking and register level blocking. A SIMD instructions based implementation was proposed, to enhance the general performance of our method. As the pairwise distance matrix is still symmetric, only half of it is to be computed. Consequently, the room for performance gain is more limited compared to the classical MMM scenario.

The distance matrix will be used by the KNN algorithm, and then, the local neighborhood statistics will be computed as presented in the subsections 3.1 and 3.2.

3.4. A topology based representation

As it can be easily concluded after reading the subsections 3.1, 3.2, and 3.3, the major bottleneck of the algorithm is the necessity of a sorting solution, in in order to produce the information representing the properties of the k-neighborhood. The sorting is the bottleneck of the given method, because of the costly memory trips induced when swapping different elements in the sorted array. A solution regarding the limitation of the number of elements sorted was required.

Considering the possibilities for improvement, a multi-dimensional lattice was used in order to limit the number of candidates for the k-neighborhood of a point, by using some information regarding the topology of the analyzed point set.

So, a splitting scheme based on the discretization of the input points projection along different axis was used in order to populate a tree with the number of children nodes equal to the resolution characterising the discretization procedure. In order to facilitate the searching procedure, the tree was hashed using a hash rule based on the basis representations, with the basis equal to the resolution of the

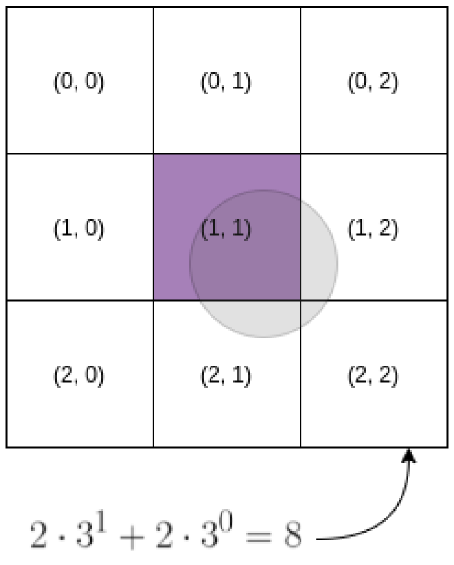

Fig. 2. Representation of the input space splitting and hashing in 2D scenario with resolution equal to 3. Geometrical limitations of the structure are represented in the center of the diagram where the purple square is the searching space for the neighborhood and the gray disc is the ground truth k-neighborhood of a given point.

discretization, and the number of bits equal to the number of splitting dimensions.

As it can be observed in the figure 2, the limitations of the topology based structure are produced by the geometrical properties of the multi-dimensional discretization procedure. As in the euclidean distance case the representation of the k-neighborhood will be a multi-dimensional ball of a radius equal to the k-distance of a given point, the discretization will carve the space into multidimensional cubes. For the searching procedure to give the exact result, the ground truth ball representing the k-neighborhood space has to be enclosed in the set of cubes considered in the breadth-first search procedure, used to determine the list of candidate points for the searched neighborhood (see figure 3). This condition is impossible to fulfill for any point in the input set with other algorithm than the brute-force search.

As the BFS algorithm is very difficult to vectorize, the usage of this hashing rule enabled a structured algorithm to determine the neighboring cells, by generating binary numbers with a given displacement around the index of the cell containing the point for which the neighborhood is computed.

However, the representation structure needs to be populated, and statistics regarding the spread of the points along given dimensions is to be known in order to perform the discretization of the input space. The performance in terms of precision will be enhanced by selecting as splitting dimensions the axis characterize by the largest spread, but this will yield the same random accessing pattern behaviour, conditioned on the input dataset.

The usage of this topology based representation structure provides the support for a much faster k-nearest neighborhood computation, but while it is based on a breadth-first search procedure in the tree representing the splits, exploiting the hashing function properties, it can be labeled as an heuristic search procedure, and hyper-parameters tuning is required in order to produce the best results.

In conclusion, this pipeline was equipped with a topology based BFS computation of the k-nearest neighborhood, providing both the k-distance array and the k-neighborhood index table, in order to perform further computations, as described in the previous subsections of the document.

4. PERFORMANCE OPTIMIZATION

4.1. Unrolling optimizations

The first step of the pipeline described in the subsection 3.1 is based on the definitions for the metrics used in statistics computation. As the metric is, as definition, a reduction operation of the vector difference, a 10 accumulators implementation was used for the **EuclideanDistance** function.

Function **K-Dist** contains a single loop, in which the function **K-Dist** is called. The loop is unrolled 5 times for maximum performance, and the function call is made inside the loop with different arguments pertaining to the unrolled loop.

Function **K-Nhood** contains three loops where two loops are nested inside the outer most loop. Both the outer and two inner loops are unrolled 5 times with separate accumulators each to get the maximum performance. Code motion is used to precompute the multiplication of constants used inside the inner most loop, to make the code even faster. Strength reduction is used to precompute costly operations, with a simpler one. The unrolled loops also have indented if statements to calculate neighborhood index table pointer. After unrolling the function using different unrolling factors and accumulators, unrolling the function by 5 gave the best results.

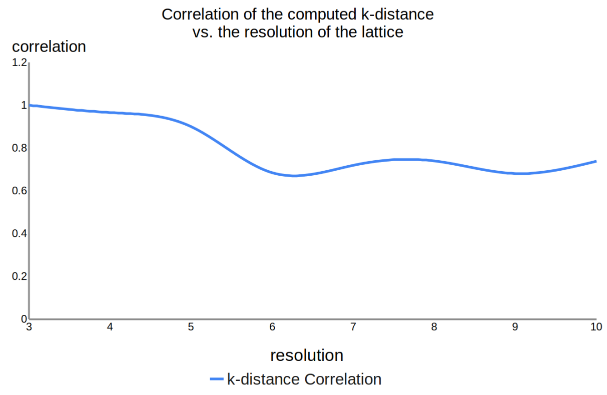

Fig. 3. Correlation between the N4-connected neighborhood BFS k-distance results and the N8-connected BFS k-distance vs. the resolution of the lattice. The decrease in performance is associated with the approximation of the unknown k-neighborhood by more, but smaller, multi-dimensional cubes.

Function **KNN** (subsection 3.2) does not represent any meaningful computation which would benefit from unrolling via better usage of the CPU resources (the workload is represented by the qsort call). We have attempted to unroll it by 4 times, both in the outer loop and inner loop, with no improvement, confirming our hypothesis.

Function **PairDist MMM** represents the implementation of the approach described in 3.3. It first computes $p^T p$ for each point of the dataset and then performs $\mathbf{A}^T\mathbf{A}$ where A is the input dataset matrix. First optimization performed was scalar replacement, then we experimented with different unroll schemes. The $p^T p$ calculation was implemented as 2 for loops; the best unrolling scheme was unrolling the outer loop by 8. The matrix-matrix multiplication was modified in order to compute only half of the resulting matrix, since the distance is symmetric. *Blocking* was done in 3 iterations, 2 at cache level and one at register level which was then unrolled: 2 for rows index, 4 for column index and 4 for dimension index. The blocking ranges were then empirically obtained and described in the next section. Finally, the actual distance computation was implemented by 2 for loops. It is unrolled by 4 times on the outer loop while computing just the lower triangular part of the pairwise distance matrix. Empirically this was the best approach to ensure that we reuse the optimum amount of points to compute their distances in a single sweep through the dataset.

Function **LRD** (subsection 3.1) performs a reduction type operation on k values for each of the n points in the dataset. It has been improved by performing scalar replacement on the accumulators. Afterwards we have experiment with different unrolling strategies, the best one being unrolling the inner loop by 8. The explanation for this is the fact the outer loop unrolling disturbs the caching mechanics by loading multiple neighbor sets which are most likely randomly scattered in the matrix, hence not in a contiguous block. Thus, extra misses will be incurred by the cache and the performance will degrade as a consequence.

Function **LOF** contains two loops: outer loop over all points in the dataset inner loop over k nearest neighbors. Strength reduction whereby two divisions were substituted by a multiplication and division, improved the performance. Our experiment reveal that outer loop unrolling by 8 with 8 accumulators and inner loop unrolling by 4 achieves the best speedup. Furthermore following code motion and strength reduction for index computation we observe a marked improvement in performance, which suggests that our cost function could be further refined by taking them into account.

4.2. AVX optimizations

Functions **EuclideanDistance** and **CosineSimilarity** were fully vectorized, as the tabular representation of the data does the implementation relatively easy to produce. These functions are going to be used by the different variants of the KNN implementations, to produce the *k-distance* and the *k-neighborhood index*.

The original implementation of the functions **LRD** and **LOF** posed a challenge due to random memory access pattern. Consequently, for some operations there was no viable alternative to using _mm256_i32gather_pd, which has a high latency. Furthermore, both subroutines required a significant number of integer computations, and since our machines do not support AVX-512 the choice of intrinsics was very limited.

Above mentioned challenges motivated the creation of the memory optimized pipeline, which aims to significantly limit the number of random accesses by providing an additional double array with distances to all k neighbors. Consequently, $lrd(p)$ and the $lof(p)$ computations for individual points could be vectorized efficiently. Figure 5.1 clearly indicates that the memory optimized pipeline achieves a much better performance.

Function **LOF** is the unrolled and vectorized version of the function to calculate local outlier factors. The function contains two nested loops. The outer loop is unrolled 8 times and the inner loop is unrolled 4 times to provide maximum performance.

Initially _i32gather was used to load data from unaligned memory. That was replaced by loadu which loads data from unaligned memory more efficiently. For summing up the vectors, we used the hadd, blend and permute2f128 operation, which resulted in considerable speedup. Using these operation inside the inner most loop was still a lot of over head, so we replaced the inner most loop operations and computed the sum of the vector after the inner most loop using hadd, blend and permute2f128 function, which resulted in the fastest version of the function.

Function **PairDist MMM** was vectorized using the techniques introduced in the lecture and adapted to our use case. The register level unrolling was done by 4 to ensure that a full slice of 4 doubles could be vectorized and computed with one fma instruction. A major challenge was to vectorize all the remaining computations after the blocking and register level unrolling was performed. In the unfortunate cases, odd remainders were obtained which means that vectorization gains could not have been exploited, thus scalar replacement and classic unrolling was performed instead. In total 4 variants of vectorized MMM were compared. The squaring of the input vectors and the actual computation of the distances roughly followed the unrolled counterparts, with adjustement for the 4-way computations done by AVX instructions.

For the pipeline described in the subsection 3.4, in the **Topo KNN** function, the hashing rule was exploited to reduce the runtime for searching the bin of a given point. By exploiting the properties of the hash function, the indices of the neighboring cells were relatively easy to produce by generating binary numbers and mask the displacement along different dimensions. In this way, the BFS procedure was vectorized, and the candidate points for the k-neighborhood were identified using the bin structure produced by the discretization mechanism.

5. EXPERIMENTAL RESULTS

Results of our cost analysis in Table 1 suggest that three parameters affect the performance of the algorithm: 1) number of points in the input dataset (**num pts**) 2) number of dimesions (**dim**) 3) number of neighbors used for computing local statistics (**k**). Since **num pts** has an effect on all subroutines we have optimized, it served as the main "variable" for our visualizations. We have also produced a plot to illustrate the optimizations of **LOF** and **LRD** functions depending on **k**.

Following the suggestion from lectures, we have used **tcs** to measure the number of cycles. To grasp the amount of variability in the performance, we have produced multiple measurements. To visuaize this information we have used boxplots, with lines connecting the median performance values.

5.1. Neighbor plots

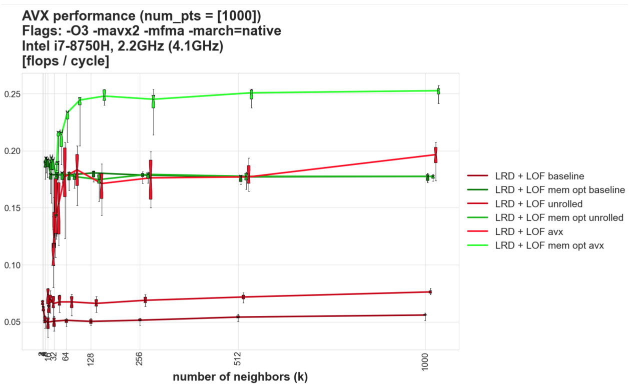

Fig. 4. Performance boxplots of the LOF and LRD functions with and without memory optimization

Figure 5.1 compares the joint performance of **LOF** and **LRD** functions with and without memory optimization, described in Section 3, depending on **k**. We observe that reducing the number of random accesses significantly boosts the performance even for a modest values of **k**.

5.2. Unroll Optimizations

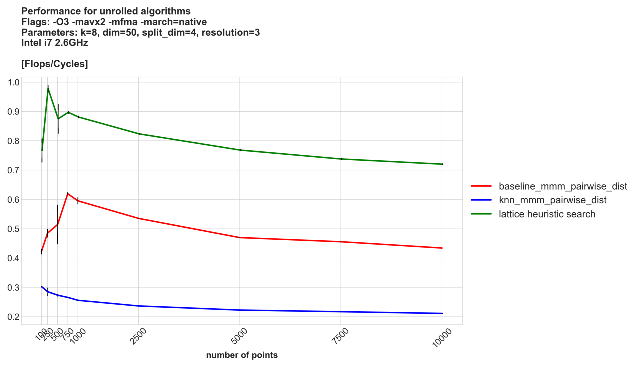

Fig. 5. Unrolled Optimizations for all the pipelines.

For unrolling, standard C optimization techniques such as unrolling, code motion, strength reduction etc. were used. The blue (knn_mmm_pairwise_dist) has a very bad performance, even with unrolling, since it involves sorting multiple times. Initially, there is an increase in performance for the rest of the pipelines, since they take advantage of unrolling and the data can be readily accessed from the cache. They reach their performance peak when they hit the cache limit. After that, the performance decreases since the data has to be accessed from the memory. Baseline mmm pairwise dist sorts only distances and then uses linear indexes to access elements, which gives it the shape as shown in the graph above.

5.3. AVX Optimizations

Baseline_mmm_pairwise_dist has much more to gain from AVX, as it is just triple matrix multiplication, so there is much more room for improvement. For the pipeline 3.4, there is not much to be gained by exploiting the properties of hash values. It uses AVX for computing indices and for distance metric. In the boxplots, the error bar variation depends on how lucky we are with the neighbors, randomness in input data and the random access pattern. When the distance and number of points are small, then all neighbors enter the cache(until 500). Then few elements have to be stored in cache (500 – 1000), so if the neighbors are not in cache, the variation would increase. After that, the variation decreases since neighbors cannot be accessed from cache anyway, as the number of points are too large.

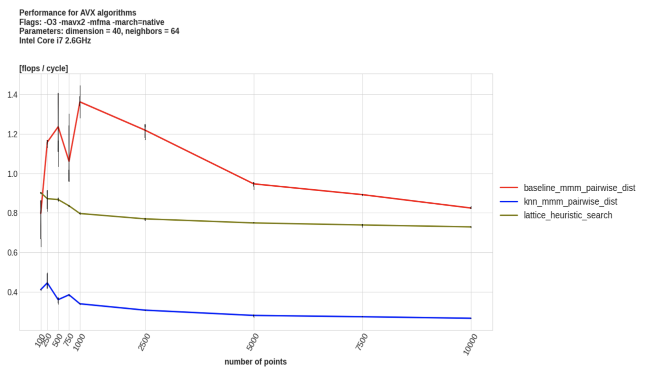

Fig. 6. AVX Optimizations for all the pipelines.

5.4. Cache Blocking

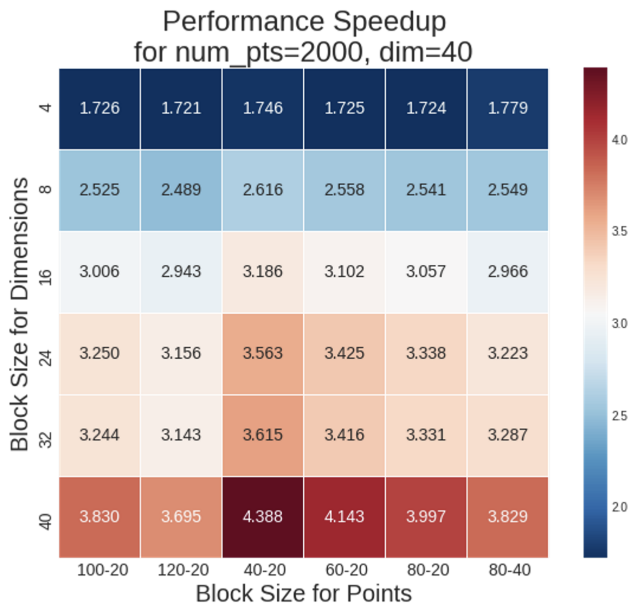

Fig. 7. Performance speedup for Euclidean Distance MMM

For euclidean distance, we use cache blocking to improve the access locality. The block size depends on the point dimension, and we are choosing the largest multiple of 4 less than or equal to the dimension, so that it is easier to vectorize and the least amount of data is left to be computed without vectorization. From the heat map, we reach the maximum performance at 40 after a consistent improvement, which is the dimension of our input data. The 40-20 on the axis represent the 1st and 2nd cache blocking dimensions.

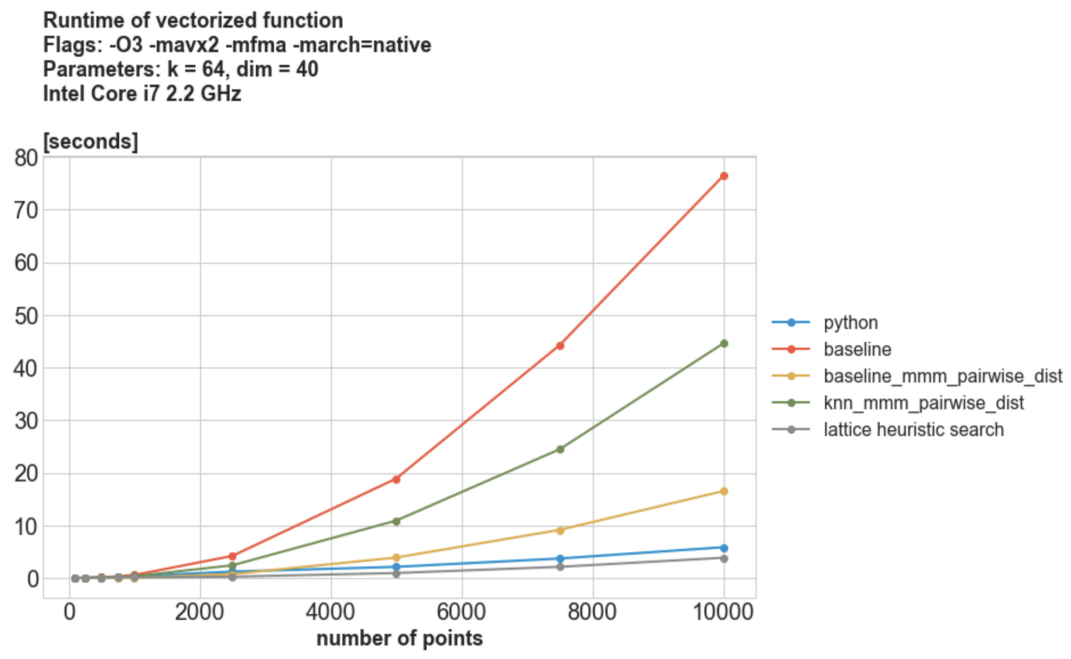

Fig. 8. Runtime comparison for all the pipelines with python

5.5. Run time

The figure 8 shows the run time comparisons of our pipelines with the python version which uses linear algebra optimization library written in c. On comparing the baseline(red) the fastest pipeline Baseline_mmm_pairwise_dist (yellow), we have achieved a 5x speedup. knn_mmm_pairwise_dist is 2x faster than the original baseline. The optimized version of the pipeline described in 3.4 is faster than the python version of the LOF, but we have to consider the amount of speed-up produced by the reduction of the number of sorted elements, by the deployment of the heuristic search.

6. OTHER EXPERIMENTS

Indexing structure for indices was used to compute indices once and feed them in a hash table. We also used array representation of upper triangular matrices, but the load given by computation of the indices is really high and it is not counted while measuring performance, so final performance was be lower. We had a recursive version of **LRD** in which the neighbors were processed one after another. It did not have much improvement since there was too much overhead with the computation. For each displacement all possible binary combinations for jumps in every direction were generated which was source of additional run time. It was a much wider search than the BFS with the results were the same, so it was not included. We tried vectoring index calculation but it was not successful on our machines. We tried merging KNN which involves swapping indices corresponding to the distances, but it performed badly because of the $n \log n$ sorting complexity,which is repeated n times for each point.

7. FURTHER EXPERIMENTS

For improving the performance further, we can experiment with smarter ways of finding the k neighbors and their distances. This includes vectorizing the sorting algorithms or even using a dedicated and vectorized sorting network for the tasks. Experimenting with other topology based representation structures can increase the performance in terms of precision. Another option is to explore even more sophisticated matrix multiplication algorithms for our usecase.

8. CONCLUSIONS

We were able to gain 5x speedup compared to our original baseline using standard C optimizations, loop unrolling, AVX optimizations and cache blocking. Avoiding random accessing pattern causes and prefetching some results implied switches also for indices helped in increasing the speedup. Formulating problems using well-studied algorithms such as using MMM multiplication for euclidean distances can help in tackling the problem better. Experimenting with topology based structures proved fruitful too, as the optimized pipeline described in 3.4 is faster than the python version of local outlier factor.

9. REFERENCES

[1] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander, "Lof: Identifying density-based local outliers," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2000, SIGMOD '00, p. 93–104, Association for Computing Machinery.

[2] 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.

[3] Zhang Xianyi, Wang Qian, and Zaheer Chothia, "Openblas," *URL: http://xianyi. github. io/OpenBLAS*, p. 88, 2012.

[4] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.