

Optimising Pairwise Distance Calculations

Student Name: Manan Isak

Supervisor Name: Chris Marcotte

Industry Supervisor Name: Eleni Vlachapoulou

Submitted as part of the degree of M.Sc. MISCADA to the

Board of Examiners in the Department of Computer Sciences, Durham University

Abstract — Distance calculations act as a bottleneck for popular machine learning algorithms such as K-Nearest Neighbours and K-Means Clustering. Most implementations focus on speeding up these algorithms by skipping as many of the pairwise distance calculations as possible. This work aims to optimise distance calculations for general usage without relying on skipping computations. Six different types of distance calculations are optimised within AMD’s AOCL-DA library while maintaining accuracy. Optimisations include changes to the mathematical and computational formulations of the distance computations, SIMD parallelisation, matrix blocking and shared-memory parallelisation, among others. The final computation times of the optimised distance calculations were compared against the original baseline performance of AOCL-DA 5.1. There was a $4\times$ speedup on average over all six distance types and strong scaling tests indicate parallel runtime fractions between 0.7 and 0.998.

Keywords — Optimisation, Distance Calculation, Distance Metric, Data Analytics, SIMD, Blocking, OpenMP, AOCL

I INTRODUCTION

Distance calculations are simple in concept but broad in their applications. They are useful in a geometric sense for measuring absolute distances between points and also serve as a measure of similarity between points of data. This makes distance measures integral to several popular machine learning algorithms such as K-Means Clustering (KMeans) and K-Nearest Neighbours (KNN). However, they can be very slow since they perform full distance computations between datasets many times per run. The time spent doing these distance computations takes up the majority of the runtime, especially as datasets become larger. Improving the speed of these computations is a necessity to accelerate development time and analysis time. Additionally, faster and more optimised distance computations are more energy-efficient in a race-to-sleep context, especially for fixed-clock CPUs.

This project was carried out in collaboration with AMD. All optimisations were implemented within the framework of their AOCL collection of libraries. A lot of development time was spent within the AOCL Data Analytics library (AOCL-DA) which already had somewhat optimised distance calculation implementations that would serve as a baseline. This collaboration allowed the usage of previous work and documentation created by experts at AMD while also allowing for contribution towards contemporary open-source code. The distance calculations already available within AOCL-DA are Euclidean, Square Euclidean, Manhattan, Minkowski and Cosine distances.

The goals for this project were selected to contribute towards the overall objective of optimising these distance computations. First, the existing distance calculations in AOCL must be analysed and any performance issues or opportunities for improvement must be identified. Second, the performance of the distance computations must be benchmarked to better deduce exactly what is causing the issues. Third, optimisations that could be implemented to improve performance must be selected. Finally, the optimised distance implementations must be compared to the original implementations to see if the execution speed successfully improved over a breadth of different use cases. A secondary objective is to add an entirely new distance calculation to AOCL and optimise it using the same process as the AOCL distances. During the course of this project, AOCL was thoroughly stress-tested and some bugs and issues were identified which were then brought to AMD’s attention.

To achieve these goals many possible optimisations were researched, applied, and then tested using a custom testing framework written separately to AOCL. AMD’s own AOCL testing framework was also used to verify compatibility and correctness of the final distance calculation implementations.

Distance calculations generally take two input matrices containing data – $X_{m \times k}$, which has m entries, and $Y_{n \times k}$, which has n entries. Each data point is represented by a vector x_i or y_j in matrices X and Y respectively

where $1 \leq i \leq m$ and $1 \leq j \leq n$. Each vector has k features thus lying in \mathbb{R}^k . The standard brute force method of computing the distances between two matrices X and Y is to compute the distance of all feature vectors x_i and y_j using a function $d(x_i, y_j)$. Each of these distances is then stored in the i^{th} row and j^{th} column of the distance matrix, $D_{m \times n}$, with the formula shown in (1). This is then repeated for every combination between the m vectors in X and the n vectors in Y , leading to a final runtime complexity of $\mathcal{O}(mnk)$.

$$d(x_i, y_j) : \mathbb{R}^k \times \mathbb{R}^k \mapsto \mathbb{R} \quad (1)$$

This scales poorly as these datasets get very large, necessitating optimisation. It also scales differently depending on which dimensions of the datasets grow larger. If the number of data points m and n becomes much larger with the complexity of the data k remaining the same, $m \sim n \gg k$, different optimisations would be necessitated than if $k \gg m \sim n$, as this would actually grow more linearly with k and would not result in as detrimental of a performance impact. These two cases of dataset shapes will be referred to as “tall and thin” and “short and wide” datasets respectively.

II BACKGROUND

1 Distance Metrics

Distance calculations are often also distance metrics, which obey the following strict mathematical axioms:

$$d(x, x) = 0 \quad (2)$$

$$x \neq y \implies d(x, y) > 0 \quad (3)$$

$$d(x, y) = d(y, x) \quad (4)$$

$$d(x, y) \leq d(x, z) + d(z, y) \quad (5)$$

Not all distance calculations are distance metrics. While distance metrics are considered to be “true” distances, in many cases non-metric distance calculations are just as useful. For example, two of the distance calculations already available in AOCL-DA – Square Euclidean and Cosine – are not distance metrics due to breaking the triangle inequality defined in (5). While this distinction is important, everything will be generally referred to as a distance calculation for simplicity.

2 Triangle Symmetry

A specific but common case is the symmetric case where the distances are only computed between the rows of a single matrix. In this case, the resultant distance matrix is square and symmetric with zeroes down the diagonal. This matrix can be said to have triangle symmetry, as the upper and lower triangles of the distance matrix are mirrored about the diagonal.

3 Data Ordering

When distance calculations are implemented in code, the way these computations interact with hardware must be considered. For example, matrices are stored in memory by being flattened into 1D arrays. Arrays ideally have their data stored contiguously in memory for faster access speeds. There are two ways of storing flattened matrices as arrays. Storing arrays in column-major order means the columns of the arrays are stored contiguously in memory, while storing in row-major order means the rows of the arrays are stored contiguously. This needs to be considered when designing optimisations since separate code for column-major and row-major arrays will need to be written. Not only is the indexing of the arrays different, but the order in which computations are performed is also different to ensure memory is accessed in a uniform manner as often as possible.

4 AOCL

The AMD Optimizing CPU Libraries (AOCL) is a collection of numerical libraries provided by AMD optimized for their Zen processors (AMD 2025). One of these libraries is AOCL Data Analytics (AOCL-DA) which uses distance calculations in several data analysis algorithms. These distance calculation implementations are currently used in their versions of KNN and KMeans. The optimisations described in this project will be designed for general use which may include usage outside of these algorithms. Focus was put on low-level optimisations that are informed by the hardware-level interactions to get each individual distance calculation to be executed as quickly as possible. It is also important to note that AOCL-DA has a minimum CPU SIMD register requirement of AVX2. All Zen CPUs have SIMD registers available for instruction-level parallelism and these come with two main instruction sets – AVX2 and AVX512. AVX2 is for 256-bit SIMD registers while AVX512 is for 512-bit SIMD registers. Much of the software in AOCL-DA has both AVX2 and AVX512 register compatible code which will need to be taken into account when implementing the new optimisations.

III DISTANCE CALCULATIONS

1 Euclidean

$$d(x, y) = \sqrt{\sum_{l=1}^k (y_l - x_l)^2} \quad (6)$$

Euclidean distance is the sum of square differences between the features of vectors x and y . It is one of the most popular measures of distance and is often thought of as the default due to it intuitively representing a straight line between points in Euclidean space. It is also often one of the most accurate distance measures in classification problems, especially on standardised data (Prasath et al. 2019). In computation, a naive implementation can be quite slow due to the square root operation.

2 Square Euclidean

$$d(x, y) = \sum_{l=1}^k (y_l - x_l)^2 \quad (7)$$

Square Euclidean distance is Euclidean distance squared. This is to get rid of the square root operation which slows the whole computation down. Unlike Euclidean distance, Square Euclidean distance is not a true distance metric as it does not obey the triangle inequality, limiting its use cases. Its main usage within data analysis algorithms like KNN is for first finding the distances between all points, then finding the exact distances in Euclidean distance by square rooting the distances to all the nearest neighbours. This is possible because distance values, which are always positive, keep their relative order after being squared i.e. $a < b < c \iff a^2 < b^2 < c^2$ is always true for distances. This leads to speedups compared to using Euclidean distance alone since this method skips most of the square root operations.

3 Manhattan

$$d(x, y) = \sum_{l=1}^k |y_l - x_l| \quad (8)$$

Also known as city block distance, Manhattan distance is the sum of absolute differences between the features of vectors x and y . It is also very popular and accurate in clustering problems just like Euclidean distance. It is often used instead of Euclidean distance since it is also a true distance metric and works well for data with features that are not mutually interchangeable in the same sense as spatial coordinates.

4 Minkowski

$$d(x, y) = \left(\sum_{l=1}^k |y_l - x_l|^p \right)^{\frac{1}{p}} \quad (9)$$

Minkowski distance is the generalisation of Euclidean and Manhattan distance. It is computationally different due to the continuous, positive and non-zero variable p which is used as an exponent for the inside sum and also as a final inverse exponent. Operations which compute continuous exponents are computationally taxing, making Minkowski computations much slower than Euclidean and Manhattan with a naive implementation. When $p = 1$ or $p = 2$ Manhattan or Euclidean distance can be used respectively instead as they are equivalent to Minkowski distance for those values of p .

5 Cosine

$$\|x\| = \sqrt{\sum_{l=1}^k x_l^2} \quad (10)$$

$$S_C(x, y) = \frac{x \cdot y}{\|x\| \|y\|} \quad (11)$$

$$d(x, y) = 1 - S_C(x, y) \quad (12)$$

Cosine distance is formulated much differently than the previous three distance calculations. However, it does still operate within Euclidean space and uses the Euclidean norm, defined in (10), within its formula. The Cosine similarity measure, shown in (11), takes the cosine of the angle between the vectors x and y about the origin. Cosine is bound from -1 to 1 for individual feature distances which makes it inappropriate for use as a distance measure since distances cannot be negative. The Cosine distance, shown in (12), takes the cosine similarity and subtracts it from 1, which bounds Cosine distance from 0 to 2 instead. While not being a true distance metric, Cosine distance is still useful when relative distance is more desirable than absolute distance.

The cosine distance currently implemented within AOCL-DA is able to return both cosine similarity and cosine distance based on an inputted boolean variable called “compute_distance”, which means both formulas (11) and (12) have to be optimised.

6 Hassanat

$$d(x, y) = \sum_{l=1}^k \begin{cases} 1 - \frac{1 + \min(x_l, y_l)}{1 + \max(x_l, y_l)} & , \min(x_l, y_l) \geq 0 \\ 1 - \frac{1 + \min(x_l, y_l) + |\min(x_l, y_l)|}{1 + \max(x_l, y_l) + |\min(x_l, y_l)|} & , \min(x_l, y_l) < 0 \end{cases} \quad (13)$$

Hassanat distance is a distance which scales from 0 to 1 as the absolute difference between a single feature of two vectors approaches infinity. It is bounded from 0 to 1 while being a true distance metric which gives it the benefit of having meaningful values of the distances between points while also being much less susceptible to errors introduced by noisy data, outliers and differing scales between features. It outperforms Euclidean distance and other distances when classifying non-standardised and noisy data but performs the same or slightly worse when classifying standardised data (Prasath et al. 2019). The main use case is to use it when the data cannot be standardised beforehand.

Hassanat distance is the only distance calculation which was not already implemented within AOCL-DA – it was chosen to be a completely original contribution to this project. Hassanat distance was chosen because it is consistently stated in the literature to be incredibly slow (Lindi 2016) (Prasath et al. 2019) (Hassanat et al. 2022). Such an effective yet slow distance calculation would be perfect to demonstrate the effectiveness of the optimisation strategies described in this project, while also contributing to the usefulness of the distance metric since it is currently highly impractical to use.

Hassanat distance has a large number of operations and it is also conditional. These attributes both contribute heavily towards the performance issues of the formula. The original creators of Hassanat distance developed a newer version which is mathematically identical but has fewer operations to improve the performance (Hassanat et al. 2022). This is shown in (14).

$$d(x, y) = \sum_{l=1}^k \begin{cases} \frac{|x_l - y_l|}{1 + \max(x_l, y_l)} & , \min(x_l, y_l) \geq 0 \\ \frac{|x_l - y_l|}{1 + \max(x_l, y_l) + |\min(x_l, y_l)|} & , \min(x_l, y_l) < 0 \end{cases} \quad (14)$$

IV RELATED WORK (LITERATURE REVIEW)

Improvements to distance computation speeds often come from the field of data science and machine learning, where these distance computations represent a performance bottleneck within popular machine learning algorithms like KNN and KMeans. Most efforts to reduce this bottleneck involve skipping as many distance computations as possible while maintaining the accuracy of the results produced by KNN and KMeans. This generally reduces the runtime complexity from $\mathcal{O}(mnk)$ to $\mathcal{O}(\bar{m}\bar{n}k)$ where $\bar{m} < m$ and $\bar{n} < n$. This approach is not applicable to this project due to the goal of optimising these distance calculations for general usage. However, it is still important to recognise the current state of the field and how distance calculations with large datasets are often used.

For example, (Mussabayev et al. 2023) developed an alternative approach to KMeans which was optimised for processing big datasets called the Big-Means clustering algorithm. It achieves improved speeds while maintaining competitive accuracy compared to the standard KMeans algorithm through a divide and conquer approach. Up to N processors perform KMeans on a significantly smaller subset of the dataset in parallel, resulting in N sets of centroids. The best performing centroids, determined by the within-cluster-sum-of-squares metric, are then kept for the next iteration step. This succeeds in decomposing the problem into many subprocesses which can be executed in parallel, improving speed and scalability. However, the size and number of these data samples are determined manually, which can lead to a time-consuming search for the ideal parameters. This was improved upon later in (Mussabayev & Mussabayev 2024) where they developed a method of randomly changing the size of each sample independently during execution and saving the sample sizes that led to improved accuracy. This list of better values could then be accessed after the algorithm terminates which can then be used to inform a manual decision on which sample size to use. While this leads to better accuracy and speed, the need to run the algorithm twice seems to be a significant oversight in regards to the goal of improving the overall speed of KMeans.

Another approach is to make use of the triangle inequality property of distance metrics to prune large amounts of unnecessary distance computations in both KNN (Ramasubramanian & Paliwal 1992) and KMeans (Hamerly 2010). This method uses the following two axioms derived from the triangle inequality, defined in (5), to establish lower bounds for distances between a data point x and two centroids b and c .

$$\begin{aligned} d(b, c) \geq 2d(x, b) &\implies d(x, c) \geq d(x, b) \\ d(x, c) \geq \max(0, d(x, b) - d(b, c)) \end{aligned}$$

When used, only enough distances to establish these lower bounds are needed and any pair of points that can be determined to belong to centroids b or c using the axioms can have their distance computations skipped, reducing the runtime complexity to $\mathcal{O}(\bar{m}\bar{n}k)$ similarly to Big-Means. Unlike Big-Means, this method still requires every datapoint in the dataset to be accessed and checked against the axioms, making the total runtime complexity $\mathcal{O}(\bar{m}\bar{n}k + \bar{m}\bar{n})$ in reality. For large values of k this slowdown is negligible compared to the time saved by skipping distance computations. For smaller values of k however the speedup is lessened.

Due to this method's reliance on the triangle inequality, it does not apply to all distance calculations. As described in Section II.1, not all distance calculations obey the triangle inequality, and so this approach does not work for many popular distance calculations, such as Square Euclidean or Cosine distance. Additionally, similar to Big Means, it only applies to clustering problems and does not address the base issue that distance computations themselves are slow and need to be optimised.

One paper estimated lower bounds for Euclidean distances by splitting high dimensional data points into blocks, then computing distances between the corresponding blocks of two points incrementally (Bottesch

et al. 2016). This established a lower bound on the distance of the whole data point during the computation which allowed for early termination in the KMeans algorithm if the lower bound already exceeded the distance threshold to not be included within a certain cluster. What is novel about this approach is that, whilst most algorithmic improvements try to skip as many of the $m + n$ points as possible, this approach tries to skip as many of the k dimensions within a single data point as possible, reducing the runtime complexity to $\mathcal{O}(mn\bar{k})$ where $\bar{k} < k$. Thus it has the best speedup when the dimensionality of the data is high.

In the special case where the datapoints are arranged to form a regular orthogonal grid within a hypercube in any dimension, optimisations can be made to skip a significant number of Euclidean distance computations (Sadílek & Vořechovský 2018). This is promising for specific datasets which contain points lying on a regular orthogonal grids since it fully and accurately computes all distances. However, it is outside of the scope of this project due to being too application-specific whereas the optimisations in this project aim to be much more general purpose. This goes to show that an approach that aims to optimise or select the data itself before performing the distance calculations is generally not widely applicable.

A feature of distance computations is that each computation is independent which opens up parallelisation opportunities. For example, because Euclidean distances can be rearranged to include a matrix-multiply, (Gallet 2023) explored the usage of special tensor cores within specific GPUs which are designed to accelerate matrix operations with parallelism at the hardware level. Beyond GPU parallelism, they also optimised Euclidean distances for computation across the CPU and GPU with load balancing to make the most of both processing units. Other works have optimised machine learning algorithms for parallelisation instead of the distance computations themselves, such as KNN with multiple-GPUs (Masek et al. 2015), KNN with a load-balanced CPU and GPU (Gowanlock 2021), and clustering with a CPU and GPU which makes use of the GPU’s limited but faster memory (Gowanlock et al. 2017).

The approach taken in this project is more broad and aims to optimise the distance computations at several more levels of abstraction compared to the current literature. On the lowest level, the throughput of the distance computations between two points will be increased with low-level, hardware-aware optimisations. Higher than that is the matrix level where the data matrices will be partitioned in memory to better make use of optimisations such as blocking or matrix symmetry. Then at the highest level, shared-memory multi-core threading will be performed on the sub-blocks of the matrices to enable scalability with multi-core CPUs.

V OPTIMISATIONS

The number of distance computations between all points in the two input matrices, $X_{m \times k}$ and $Y_{n \times k}$, grows with $\mathcal{O}(mnk)$ as the amount of data, both number of entries and number of features per entry, increases. Additionally, the way the matrices X and Y are accessed results in entries in X or Y being re-loaded into cache and registers many times from main memory, slowing down the computations further. Finally, the sheer number of variations of input metadata – column- or row-major data ordering, tall and thin or short and wide datasets, which distance calculation to use, etc. – makes it difficult to apply a single change that speeds up all possible usage cases.

This section describes each optimisation applied to the distance computation code and discusses the advantages and disadvantages each optimisation posed for each distance calculation.

1 Compiler Optimisations

Using different compilers to compile the C++ code can sometimes result in executables compiled with different optimisations which can lead to performance differences. AOCC 5.0.0 and GCC 14.1.0 were both tested in this project, but no significant differences between the execution times of the two was found. All final testing was carried out in GCC except where indicated.

The easiest way to speed up code is to allow the compiler to do the optimisations. Under the most aggressive level of optimisation, compilers can drastically alter the way code executes with auto-vectorisation. This makes use of the SIMD capability of the hardware, increasing the speedups by up to four times for AVX2 registers or eight times with AVX512 registers since they can operate on four or eight doubles at a time respectively. While vectorisation is not an optimisation exclusive to the compiler, having the compiler auto-vectorise the code saves development time, reduces code size, improves readability and is already well-tested compared to implementing vectorisations manually. However, loops need to meet many conditions before the compiler can

vectorise them (GNU-Project 2005). Three of the most relevant conditions to this project are the “control-flow” condition, the “vectorisable statements” condition and the “memory access pattern” condition.

The “control-flow” condition assesses whether there are any significant if-else statements that change the operations within the loop. Anything beyond simple, single line ternary statements are unable to be vectorised. This interfered with Hassanat distance – the formula itself has a conditional statement within it which prevented vectorisation as shown in (13) and (14). The formula itself had to be redesigned to remove this conditional statement while maintaining mathematical equivalence which is detailed further in section V.3.

The “vectorisable statements” condition assesses whether each operation within the loop has an equivalent vector operation within the AVX2 or AVX512 vector instruction sets. If a matching vector operation does not exist, the loop is not vectorised. This interfered with Cosine and Minkowski distance. For Cosine, the calls to the AOCL BLAS functions (Van Zee & van de Geijn 2015) for the norm and dot product operations could not be turned into equivalent vector operations. This meant the division and multiplication operations within the formula, which do have vectorised equivalents, were also not vectorised. This was fixed by separating the division and multiplication operations into a separate loop which was then auto-vectorised successfully. For Minkowski, the continuous p^{th} power and inverse power operations have no vector equivalent. To solve this, manual vectorisation must be implemented.

The “memory access pattern” condition assesses whether arrays accessed within the loop are accessed in a consistent and independent manner. If a single entry in an array is attempted to be updated by the sum of all values produced by a SIMD operation, the vectorisation would fail. All row-major computation loops in Manhattan, Minkowski and Hassanat distance failed to auto-vectorise because row-major vector operations update a single value of D as shown in Fig. 1. This can only be solved with manual vectorisation.

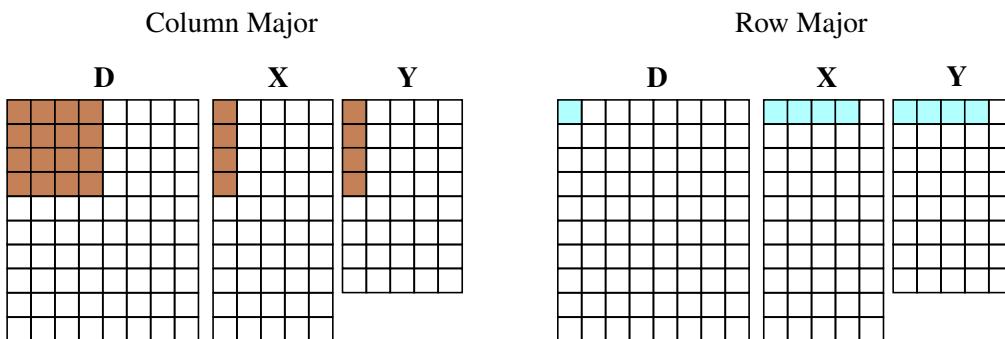


Figure 1: Comparison of the SIMD distance computation memory accesses in column- and row-major data ordering with a distance matrix $D_{10 \times 8}$, input matrices $X_{10 \times 5}$ and $Y_{8 \times 5}$, and a register width of four doubles.

Intrinsics are built-in compiler functions that map to specialised assembly code. GCC and AOCC both have vector intrinsics which map directly to SIMD assembly code for both AVX2 and AVX512. Writing code with intrinsics allows vectorisation of loops which are incompatible with compiler auto-vectorisation. While the speedups obtained can be significant, it results in much larger code size due to requiring different syntax between doubles and floats and between AVX2 and AVX512. AVX2 is particularly challenging to write code with due to its older and more limited instruction set. For example, Manhattan distance requires an absolute value operation which AVX2 does not have. A workaround was designed which flipped the sign of any negative numbers produced by the vector subtraction in the Manhattan computation. This difference in approach is shown in Fig. 2.

For Minkowski distance, the initial problem involved the vectorisation of the continuous power operations shown in (9). Neither the AVX2 nor AVX512 vector instruction set has equivalent operations for the continuous power function which means that there is no native method of compiling SIMD code for Minkowski distances. The AOCL-LIBM library does have its own custom intrinsics for continuous power operations for both AVX2 and AVX512 when compiled with AOCC. When these custom intrinsics were implemented into Minkowski distance, only a row-major version of the vector computations was able to be created. For the column-major version, the matrices were transposed from column-major to row-major ordering to make use of the row-major intrinsic vectorisation. Separate column-major intrinsic code was not developed due to the time constraints of the project.

For the square root loop in Euclidean distance, the square root vector intrinsic available in both AVX2 and

```

1 void manhattan_row_avx512(int k, const double *row_X, const double *row_Y, double *D) {
2     const int reg_cap = 8; // How many doubles can fit in the register
3
4     // Go through the columns of X and Y
5     for (int l = 0; l <= k-reg_cap; l+=reg_cap) {
6         __m512d vec_x = _mm512_loadu_pd(row_X + l);
7         __m512d vec_y = _mm512_loadu_pd(row_Y + l);
8
9         // Full subtract and full abs. float abs only available in avx512
10        vec_y = _mm512_sub_pd(vec_x, vec_y);
11        vec_y = _mm512_abs_pd(vec_y);
12
13        // Update D
14        *D += _mm512_reduce_add_pd(vec_y);
15    }
16 }
17 void manhattan_row_avx2(int k, const double *row_X, const double *row_Y, double *D) {
18     const int reg_cap = 4; // How many doubles can fit in the register
19     double temp_vec[reg_cap]; // Helper array for doing a reduction sum to D
20
21     // Go through the columns of X and Y
22     for (int l = 0; l <= k-reg_cap; l+=reg_cap) {
23         __m256d vec_x = _mm256_loadu_pd(row_X + l);
24         __m256d vec_y = _mm256_loadu_pd(row_Y + l);
25
26         // Generate mask to detect negative numbers
27         __m256d mask = _mm256_cmp_pd(vec_x, vec_y, _CMP_GT_OQ);
28
29         // Full subtract. Reuse vec_x for the result
30         vec_x = _mm256_sub_pd(vec_x, vec_y);
31
32         // Negative version of the subtract. Reuse vec_y for the result
33         vec_y = _mm256_mul_pd(vec_x, _mm256_set1_pd(-1.0));
34
35         // Blend the positive results from each vector together with the mask
36         vec_x = _mm256_blendv_pd(vec_y, vec_x, mask);
37
38         // Update D
39         _mm256_storeu_pd(temp_vec, vec_x);
40         *D += temp_vec[0] + temp_vec[1] + temp_vec[2] + temp_vec[3];
41     }
42 }
```

Figure 2: A snippet of source code showing the different ways the AVX512 and AVX2 row-major Manhattan distance computations were implemented.

AVX512 was used. Because this final square root is a one-to-one operation between the distance matrix and itself, the distance matrix can be flattened and linearly iterated through without having to worry about blocking or the data ordering.

The optimal loop ordering for row-major before vectorisation is to iterate through the rows of X , then the columns of X and Y , and then finally the rows of Y . After adding intrinsics to Manhattan and Minkowski, it was better to have the column loop be in the innermost layer of the nested for loop because the SIMD operations happened along the columns of X and Y .

2 Precomputation

Precomputation is the process of identifying values within a calculation that are reused repeatedly, then computing them beforehand and storing them in memory for reuse within the computation instead of recomputing them each time. This saves computation time at the cost of increased memory usage – although the increase in memory in even the worst case was more than manageable. This was successfully implemented for

Cosine distance by identifying that the norms of x and y in the formula for Cosine similarity, (11), are reused repeatedly and are thus eligible for precomputation. It also presented the opportunity to move the divide by zero check from within the main loop to the norm precomputation instead, reducing the number of operations further. In the formula for Cosine similarity, it can be seen that a zero division can only occur if either the norm of x or the norm of y is 0. The norm of a vector is only equal to 0 for the zero vector. It is also known that the dot product of any vector with the zero vector equals 0. Therefore the only time a zero division can occur is when it is 0/0, which according to convention for Cosine distance should be interpreted as 0. This can be put into code by setting any norm calculated as equalling 0 to instead equal 1 (or any positive non-zero number) with no loss in accuracy. By doing this any time a zero division could occur it is instead replaced by 0/1 which equals 0.

3 Equation changes

The actual best way to improve the performance of a distance calculation is to find some mathematically equivalent formula that performs better – either by having fewer or less complicated operations. This improves computation speed before making any significant changes to the code. However, with most distance calculations this avenue has been thoroughly explored, with the current versions of each formula being the simplest versions there are. The exception within this project is Hassanat distance, since it was invented only recently (Hassanat 2014) and detailed exploration into improving the performance of the formula has been scant to say the least. The most recent iteration of Hassanat distance shown in (14) improves upon the number of computations greatly (Hassanat et al. 2022) and two further improvements were made for this project.

The first improvement was small and saved only a single operation per computation. In the lower condition of the formula the value $\min(x_l, y_l)$ will always be negative, owing to the condition $\min(x_l, y_l) < 0$. This means the absolute value operation is unneeded and instead the addition can be replaced with a subtraction. The improved formula is shown below.

$$d(x, y) = \sum_{l=1}^k \begin{cases} \frac{|x_l - y_l|}{1 + \max(x_l, y_l)} & , \min(x_l, y_l) \geq 0 \\ \frac{|x_l - y_l|}{1 + \max(x_l, y_l) - \min(x_l, y_l)} & , \min(x_l, y_l) < 0 \end{cases} \quad (15)$$

The second and most impactful change made was consolidating the two conditions of the formula into a single calculation. The conditional was preventing any attempts from the compiler to auto-vectorise the computation loop, as described in section V.1. The only difference between the two conditions was the inclusion of the $-\min(x_l, y_l)$ in the denominator of the lower formula. To combine the two conditions this value would have to equal zero when the upper condition was met – when both x_l and y_l were positive or zero. This was achieved by turning it into a three-way minimum between x_l , y_l and zero. A three-way minimum is implemented by performing a two-level nested min but for ease of readability it is notated as $\min(a, b, c)$ instead of $\min(a, \min(b, c))$. The final formula is shown below.

$$d(x, y) = \sum_{l=1}^k \frac{|x_l - y_l|}{1 + \max(x_l, y_l) - \min(x_l, y_l, 0)} \quad (16)$$

While (16) technically has an increased average operational intensity, the overall speed increase from the removal of the conditional statements is more than worth it. It also decreased the code size and complexity as an added bonus.

4 Matrix Blocking

Matrix blocking, also known as matrix partitioning, is a technique for dividing matrices into smaller sub-matrices, or blocks, to manually balance the loading of data with the computation of that data. If these blocks fit within hardware cache, the CPU does not need to retrieve elements from main memory as often, increasing data locality and decreasing the number of cache misses (Goto & Van De Geijn 2008). This improves performance for operations which are memory intensive, like distance calculations. The main disadvantage is that this technique places the decision on things such as block size, shapes and ordering on the programmer instead of the compiler which comes with several associated design challenges, especially if different cache sizes from different hardware are to be utilised effectively by the same code.

Blocking has various degrees to which it can be utilised. In this project, there were three options available for partitioning the matrices – the rows of input matrix X and the rows of distance matrix D , the rows of input matrix Y and the columns of distance matrix D , or the columns of X and Y . Ultimately, blocking was only applied along the rows of X and the rows of D , and the rows of Y and the columns of D due to the fact that blocking the columns of X and Y has little performance impact since that does not effect the blocking of matrix D . Additionally, it would interfere with the row-major vectorisation implementations described in section V.1 which iterate along the columns of X and Y .

5 Triangle Symmetry

When all the distances between the rows of a single matrix X are computed, it results in a distance matrix that is symmetric with zeroes down the diagonal. This allows $n(n + 1)/2$ computations to be skipped by only calculating the upper or lower triangle of the matrix, setting the diagonal entries to be zero and then copying all the distances to the opposite triangle in the matrix. The advantages are obvious – while copying half the matrix is not free, skipping over half the calculations can result in just under a two times speedup. However, it also affects the uniformity of the memory access pattern which holds back the total performance improvement. Additionally, when combined with the basic blocking method described in the previous section, the blocks along the diagonal of the matrix still compute all calculations within that block. Thus, the total number of computations skipped is influenced greatly by the block size as shown in Fig. 3.

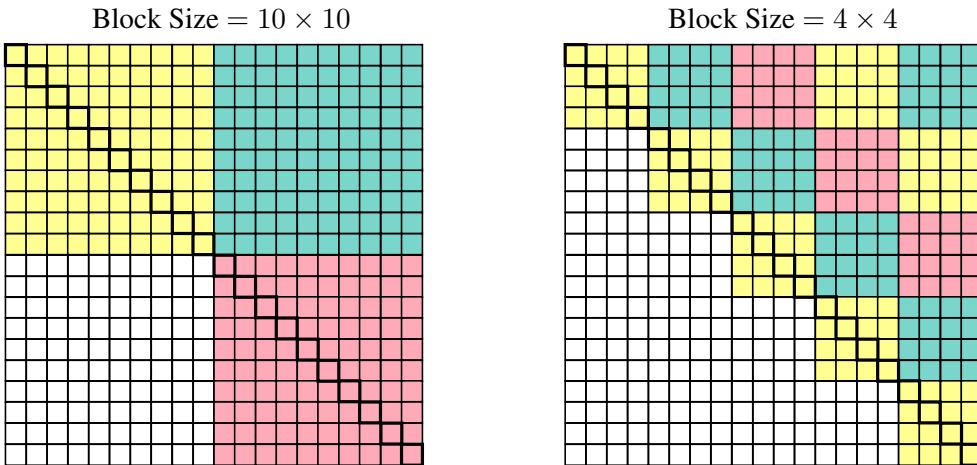


Figure 3: Demonstration of how the block size affects the number of distance calculations skipped. Displayed is symmetric distance matrix $D_{20 \times 20}$ blocked with two different block sizes. Each coloured square represents a block with the white region representing skipped distance calculations. The diagonal separating the equal and opposite sides of the symmetric distance matrix is bolded. The larger blocking scheme only skips 25% of the total distance calculations while the smaller blocking scheme skips 40%.

6 OpenMP Parallelisation

AOCL includes support for OpenMP shared-memory multi-threading which is an API available for C++ which enables code to be compiled to make use of shared-memory multi-processing (OpenMP 2025). Parallelisation applies well to the problem of distance calculations because each computation is independent and data race concerns only arise when updating the same entry in the distance matrix. These data races can be avoided with OpenMP's reduction clauses however. OpenMP was implemented in combination with matrix blocking. Simple OpenMP pragma statements were inserted into the outer blocking loops of the core computations and in any precomputation loops. Distributing the matrix sub-blocks between OpenMP threads encouraged data-locality per thread, further improving memory access speeds. This meant that using OpenMP in conjunction with blocking saw better strong scaling with multiple cores than with OpenMP alone.

Finally, a summary of which optimisations were implemented for which distance calculations is shown in Table 1.

Table 1: Table showing which optimisations were applied to each distance calculation

	Euclidean	Square Euclidean	Manhattan	Minkowski	Cosine	Hassanat
Blocking			✓	✓	✓	✓
OpenMP	✓	✓	✓	✓	✓	✓
Compiler			✓	✓	✓	✓
Intrinsics	✓		✓	✓		✓
Precomputation	*	*			*	
Formula	*	*				✓
Symmetry	*	*	✓	✓	†	

✓ Implemented as part of this project

* Already implemented in AOCL-DA

† Only the Cosine precomputation of norms made use of symmetry

VI BENCHMARKING

How performance is measured is important for a project where improving performance is the main goal. We present data on how long a full calculation between two matrices took to complete, the cache miss percentage of a full calculation and the peak memory usage of the implementation. Completion time is a direct indicator of computation speed and efficiency on platforms with a fixed clock-speed. Cache miss percentage is an indicator of poor cache memory utilisation which can be a direct cause of memory bottlenecks. Peak memory usage was also measured because some optimisation methods found speedups by increasing memory usage and it is important to identify when these memory usage spikes become significant. The C++ chrono library was used to measure the calculation time, perf was used to measure the cache miss percentage, and heaptrack was used to measure the peak memory usage which is an open source heap memory profiler available for C++ (KDE 2025). The main testing will involve recording benchmarking data for a full distance calculation between two equal matrices. Using two equal matrices simplifies the testing process and can also be used to directly compare generic distance calculations with symmetric distance calculation, which is discussed later in this section. These variables – calculation time, cache miss percentage and peak memory usage – will be the dependent variables in all testing. The rest of this section discusses the independent variables of the tests.

There are a number of independent variables that need to be tested to cover the full range of use cases available in AOCL. These are data ordering, dataset shape, matrix symmetry, hardware register width and number of cores. Data ordering refers to whether the data is stored in column-major ordering, where the columns of the matrix are contiguous in memory, or row-major ordering, where the rows are contiguous in memory. The optimisations affected performance for each data order differently and so tests with both orders were carried out to ensure one implementation was not accidentally made slower. The two dataset shapes tested were a tall and thin dataset and a short and wide dataset. The tall and thin dataset was 21,263 by 81 elements (Hamidieh 2018) while the short and wide dataset was 801 by 20,531 elements (Fiorini 2016). They are both made of real data retrieved from the UCI Machine Learning Repository (Kelly et al. 2023). The general distance calculation case is when two matrices are inputted. The symmetric case is when all the distances between a single matrix and itself are computed. These are both common usage cases and have access to different optimisations so it is important to test both. The hardware register width options were either AVX2 (256-bit) or AVX512 (512-bit). Finally, the strong scaling performance of each distance calculation was recorded as the number of cores increased up to 64 cores.

It is important to address that the difference in performance between single and double precision floating point numbers could also have been measured. Ultimately, only double precision was used in testing as making optimisations specifically for single or double precision floating point numbers was determined to be outside of the scope of this project. Since no such optimisations were made, there would be no significant interpretations of any measurements taken from any benchmarking beyond “single precision operations are faster”. While all final benchmarking used double precision numbers, tests were ran to ensure that any optimisations implemented remained compatible with single precision data.

VII RESULTS AND DISCUSSION

This section describes a small selection of notable results taken from the data collected from the benchmarking tests. All tests were performed on two CPUs: the AMD EPYC 7702 on Durham University's Hamilton 8 compute cluster with AVX2 registers (Hamilton8 2021) and the AMD EPYC 9654 from Durham University's HPC Hardware Lab with AVX512 registers (HPC-Lab 2023). All distance calculations saw improvements in performance for single-core tests with the exception of Square Euclidean distance. Success was also found in implementing OpenMP parallelisation to all distances which all scaled well with multiple cores. The final distance calculations had an average $4\times$ speedup which demonstrates the efficacy of the optimisations. All remaining data not included in this discussion is instead shown in the Appendix.

1 Euclidean

The main improvement to Euclidean distance was the manual vectorisation of the square root operation with vector intrinsics as described in section V.1. As shown in Fig. 4, the Euclidean distance calculation can be split into three main parts - the precomputation of the norms, the general matrix multiply that uses BLAS:GEMM (Van Zee & van de Geijn 2015) and the final square root of the whole distance matrix. The GEMM operation is expected to take up the majority of the computation time. However, as can be seen in the original implementation calculation time distribution, the square root is taking up a disproportionate amount of time. After applying vector intrinsics, the square root then takes up a much smaller portion of the computation time.

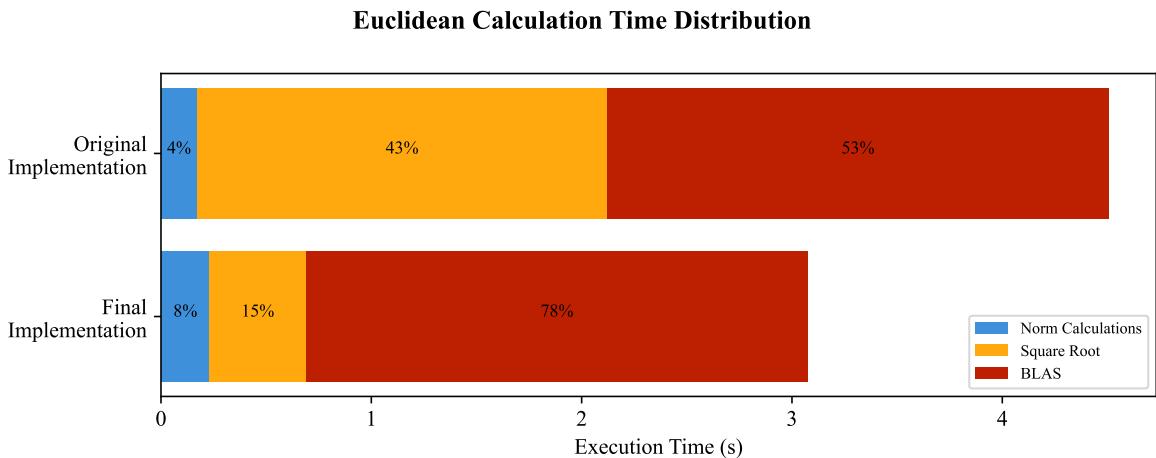


Figure 4: Stacked bar chart showing the percentage of the total Euclidean calculation time taken up by the precomputation of norms, BLAS:GEMM and the final square root before and after adding intrinsic vectorisation for the square root operation. These proportions were recorded for benchmarks with the tall and thin dataset.

The improvements in overall computation time with AVX2 are shown in Fig. 5. It shows that all tests done with the tall and thin datasets saw significant speedups while the short and wide datasets had only small and negligible improvements. This is because the number of square root operations is equal to the total number of entries within the distance matrix $D_{m \times n}$. For tall datasets where $m \sim n \gg k$ this is multiple orders of calculations more than for short datasets where $k \gg m \sim n$. This explains why there are little to no speedups for the short and wide dataset because the square roots did not take up the same significant portion of the computation time as shown in Fig. 4 for tall and thin datasets.

The cache miss percentage for the tall and thin dataset was also measured and is shown in Fig. 6 to demonstrate the relationship between cache miss percentage and calculation speed. Firstly, despite having faster execution times, the symmetric cases ($X==Y$) had significantly higher cache miss percentages. This is caused by the non-uniform way the distance matrix is accessed by only using the upper or lower triangle for storing distances. Even though a higher cache miss percentage means slower memory access speeds, skipping over half the distance calculations makes this a worthy trade-off and shows that a lower cache miss percentage does not necessarily equate to faster times overall.

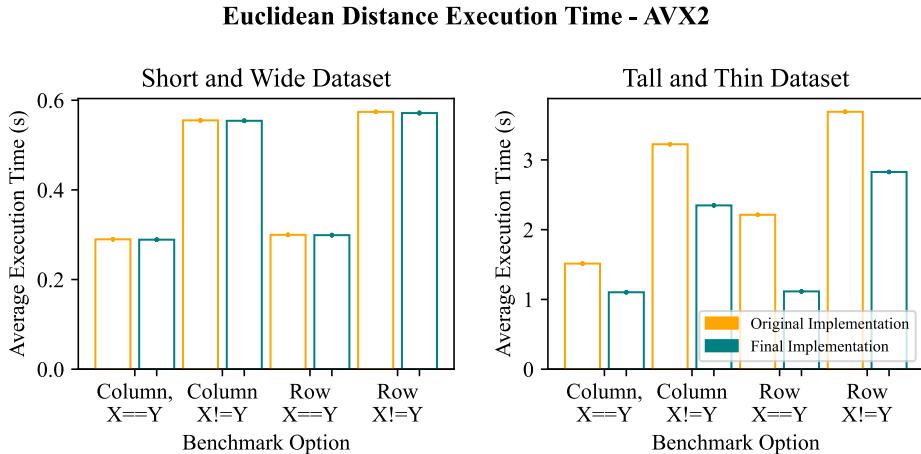


Figure 5: Comparison of the original Euclidean implementation with the final implementation’s execution time on the AMD EPYC 7702 (AVX2) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots.

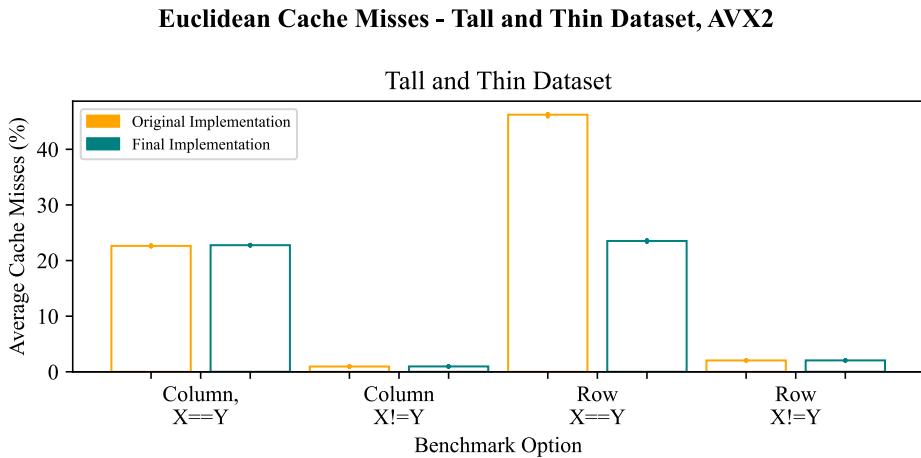


Figure 6: Comparison of the original Euclidean implementation with the final implementation’s cache miss percentage on the AMD EPYC 7702 (AVX2) with the tall and thin dataset. The cache miss percentages are plotted as bars with each of the ten individual percentages plotted as dots.

While implementing the triangle symmetry optimisation resulted in an overall speedup, when combined with OpenMP parallelisation, the performance does not improve as well as the general case as the number of cores increases and oftentimes becomes slower than the single-core performance. This can be partially remedied by blocking the symmetric case. After applying blocking, the strong-scaling with OpenMP improves with the downside that it is no longer skipping as many calculations.

This meant that some distance calculations (Hassanat and Cosine) had no optimisations for the symmetric case since it was determined the blocked triangle symmetry optimisation was not improving the overall computation time enough as the number of cores increased.

2 Square Euclidean

The only improvement made to square Euclidean distance was adding parallelisation with OpenMP. The strong scaling up to 64 cores is shown in Fig. 7 for both processors and with and without matrix symmetry. The results are also compared with Amdahl’s law with an approximate parallel portion, p . Square Euclidean actually scaled the worst out of all the distance calculations due to the computation itself being incredibly fast and thus representing a smaller parallel portion of the whole program compared to the other distance calculations. This lead to the serial portions and thread overhead taking up a larger proportion of the execution time as the number

of cores increased, leading to decreasing speedups and eventually slowdowns. The approximate value for p is always lower for the symmetric matrix case, likely due to the total number of computations in the parallel region being roughly halved and thus taking a comparatively smaller portion of the overall calculation, thus making p smaller. On paper, this logic should also hold for the comparison between the AVX2 and AVX512 CPUs. However, the reverse is observed instead – p is higher despite the theory that doubled SIMD register width should lead to the distance calculation taking up a smaller portion of the overall program execution time. This observation begins to make more sense once the memory access speeds of each test platform are taken into account. The AVX512 CPU has access to faster memory and the serial portion of the program is mostly spent loading the datasets into memory. A faster memory speed for this serial portion leads to it running faster and thus taking a smaller portion of the overall execution time. Therefore the higher values for p for the AVX512 CPU are likely caused by the differences in memory access speeds rather than the difference in the SIMD register width.

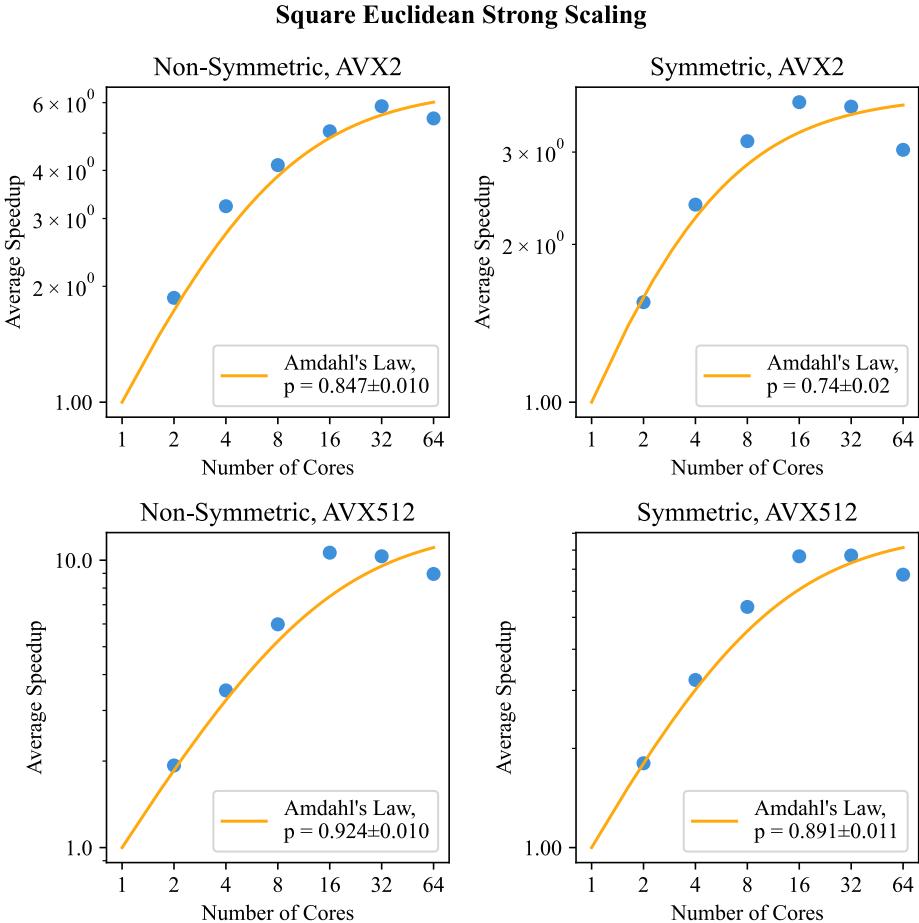


Figure 7: A collection of log-log strong scaling graphs for Square Euclidean distances. Strong scaling tests were conducted with row-major data ordering and tall and thin datasets due to consistently being the slowest benchmark option and ideal for showing the best case strong scaling. These four graphs were chosen to display how the scaling changes between SIMD register width and with or without matrix symmetry. Each test was fitted against Amdahl’s law using the damped least-squares method to approximate the parallel portion, p .

3 Manhattan

Fig. 8 shows the performance improvements for Manhattan on the AVX512 CPU. All test permutations saw improvements. Notably, the row-major tests saw the best improvements. In fact, the symmetric row-major test saw a speedup of 861% – the largest relative speedup of any distance calculation already available in AOCL. The column-major implementation however only saw small improvements. This aligns with the optimisations used – the row-major computations had to be manually vectorised with intrinsics while the column-major com-

putations were already auto-vectorising. Despite the manual vectorisation, column-major remained faster than row-major even in the final version. This is because the row-major code still requires a serial summation reduction of the SIMD result before adding the result to the distance matrix, as shown in Fig. 2. The column-major calculations' improvements were mostly the result of blocking. Blocking saw greater performance benefits for the tall and thin dataset due to the distance matrix being larger and having more addresses to be accessed and blocked.

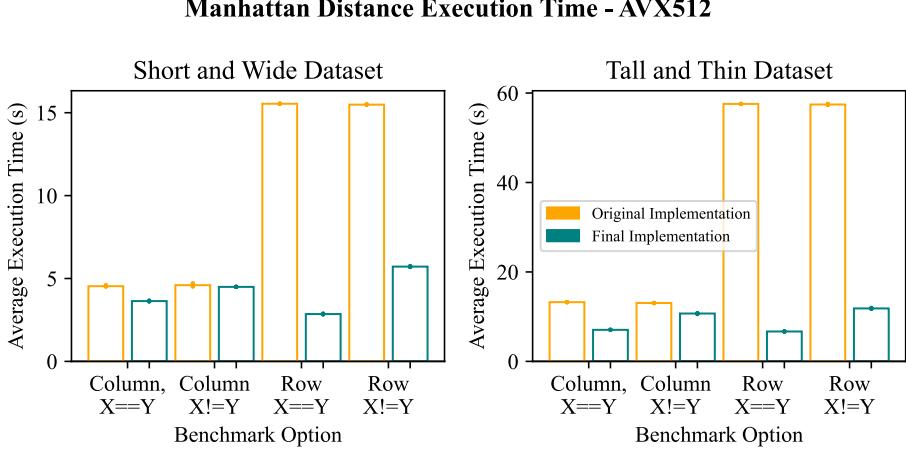


Figure 8: Comparison of the original Manhattan implementation with the final implementation's execution time on the AMD EPYC 9654 (AVX512) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots.

4 Minkowski

Out of all the distance calculations described in this project, Minkowski had the slowest original implementation by far. On the other hand, this meant the final version had the largest absolute speedup out of all the distance calculations of around 500 seconds for symmetric, row-major data ordering with the tall and thin dataset, as shown in Fig. 9.

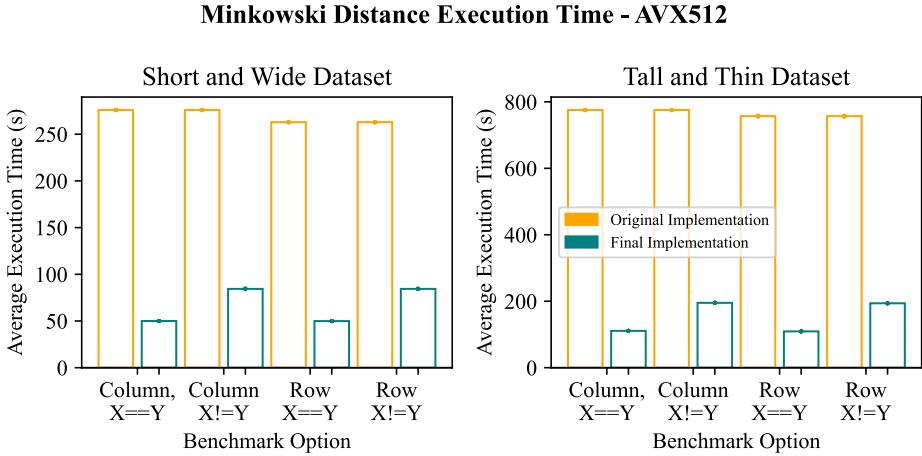


Figure 9: Comparison of the original Minkowski implementation with the final implementation's execution time on the AMD EPYC 9654 (AVX512) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots. Compiled with AOCC 5.0.0.

The fact that Minkowski distance is so slow even after applying all of the optimisations results in the parallel fraction of the distance calculation that makes up the main computations to be very large. This is

demonstrated in Fig. 10 which displays the best strong scaling out of all the distance calculations. It shows that the serial portion is practically negligible and near-ideal speedups could possibly still be attained by increasing the number of cores further past 64.

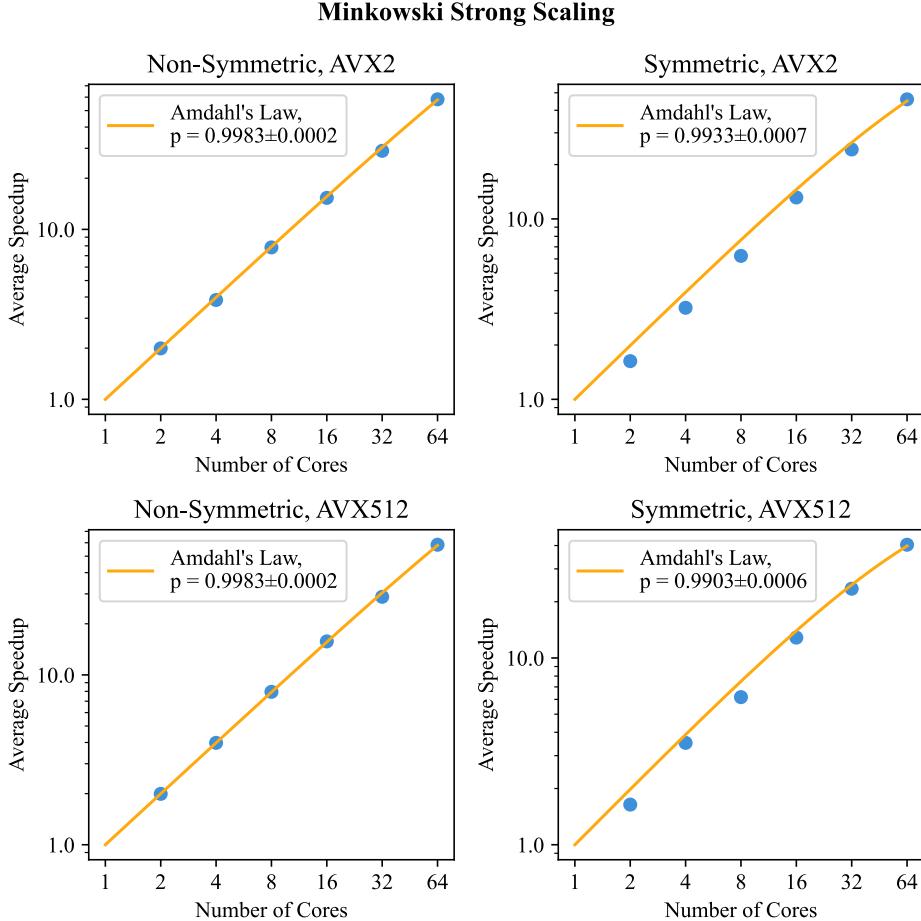


Figure 10: A collection of log-log strong scaling graphs for Minkowski distances. Strong scaling tests were conducted with row-major data ordering and tall and thin datasets due to consistently being the slowest benchmark option and ideal for showing the best case strong scaling. These four graphs were chosen to display how the scaling changes between SIMD register width and with or without matrix symmetry. Each test was fitted against Amdahl’s law using the damped least-squares method to approximate the parallel portion, p . Compiled with AOCC 5.0.0.

5 Cosine

The execution time improvements for Cosine distance with the AVX512 CPU are shown in Fig. 11. While all tests showed improvements in the execution time, the relative speedup of the symmetric cases compared to the non-symmetric cases was lost. This is because when implementing the cosine distance optimisations the matrix symmetry optimisation was actually removed from the original distance calculation. This was to achieve a better strong scaling with blocking, which was ultimately beneficial as it resulted in very good strong scaling as shown in Fig. 12. While the symmetric case can roughly double the speed of the computations, it is not applicable to all distance calculation use cases. This is in contrast to the benefits of multi-core parallelisation which can double the speed of the computation by only using a single extra core while also being applicable to all distance calculation use cases. These trade-offs justify prioritising the strong scalability.

One caveat of Cosine distance calculations is the extra memory usage when using column-major data ordering. As shown in Fig. 13, the peak memory usage with column-major data ordering for the tall and thin dataset is doubled when compared to Manhattan distance. This is because any column-major ordered matrix is transposed to row-major ordering for the computation by making copies of both input matrices and also the

Cosine Distance Execution Time - AVX512

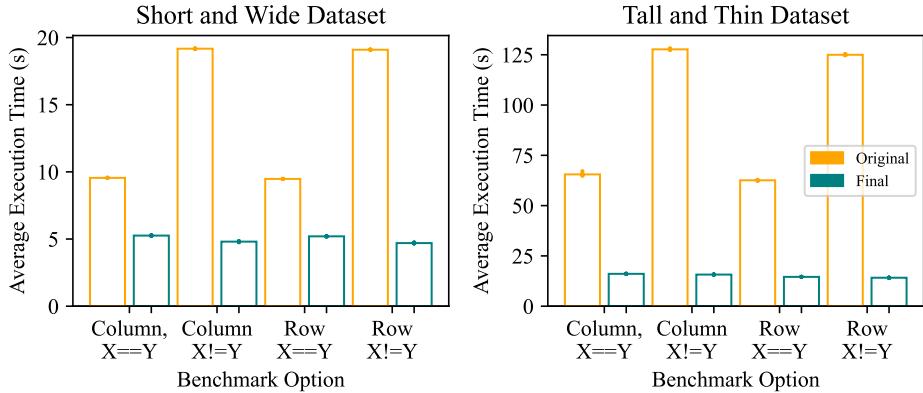


Figure 11: Comparison of the original Cosine implementation with the final implementation’s execution time on the AMD EPYC 9654 (AVX512) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots.

distance matrix. However, this doubling of the peak memory usage is not observed with the short and wide dataset despite the fact that the number of elements in the short and wide dataset and the tall and thin dataset are roughly the same. The actual difference in memory usage is caused by the size of the distance matrix $D_{m \times n}$ which is significantly larger when used with a taller dataset. Since the process of transposing the graphs is quite fast relative to the main distance calculations this is not necessarily a performance concern. Rather, this poses a practical issue where if a user wants to use a large dataset for distance calculations with column-major data ordering, their memory requirements are doubled. With especially large datasets a doubled memory requirement cannot always be fulfilled. Fixing this issue is outside of the scope of this project since it does not affect the overall speed of the program significantly but it is important to recognise the further improvements that could be made.

6 Hassanat

Hassanat was previously known as an impractically slow distance calculation and saw little use despite its advantages when working with unsanitised data. However, with the improvements and optimisations described in this work it is now operating with speeds in the same order of magnitude as Manhattan distance. In fact, out of all the distance calculations in this project it saw the biggest relative speedup of roughly 1520% for row-major data ordering with the short and wide dataset, as shown in Fig. 14. Notably, there is no benefit for making use of matrix symmetry. This was to make Hassanat distance scale better with multi-core parallelisation, similar to Cosine distance. For the same reasons as Cosine distance this decision is justified by strong scaling results shown in Fig. 15.

7 AOCL Bug Fixes

Within this project the AOCL-DA library was thoroughly stress-tested and one compiler issue and one AOCL-DA bug was found during the development process that were reported to AMD.

In Euclidean distance, the square root operation was not auto-vectorising due to failing to meet the “control-flow” condition described in section V.1. This is unusual since the square root operation itself has no conditional statements in the code. GCC and AOCC both add an additional, unnecessary check for negative numbers at compile time. This check ensures *Nan* is returned if the square root of a negative number is attempted, however the square root operation already returns *Nan* for negative numbers without this check. This check can be disabled in the compiler options to allow for auto-vectorisation. However, it is categorised as a fast-math compiler flag, which is disabled in AOCL to maintain numerical accuracy. This means any square root operation loop within AOCL compiled with GCC or AOCC will never auto-vectorise. The only option was to instead implement manual vectorisation with intrinsics.

Cosine Strong Scaling

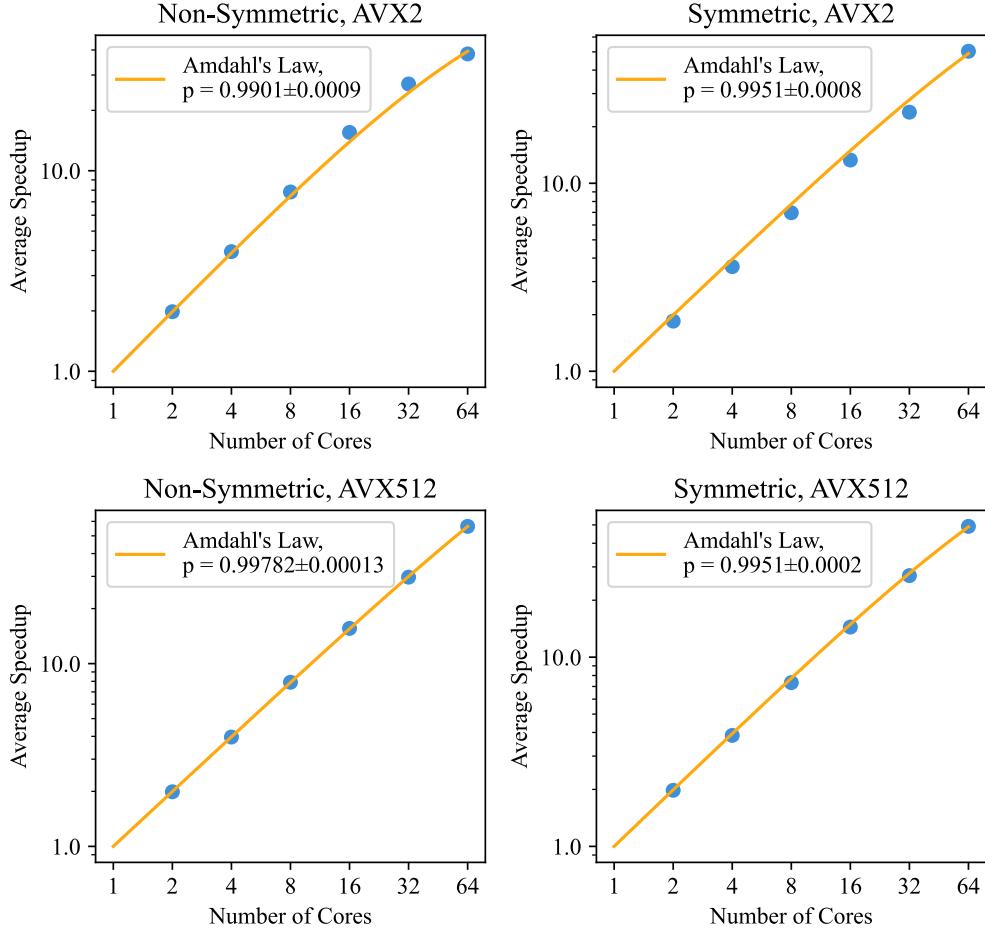


Figure 12: A collection of log-log strong scaling graphs for Cosine distances. Strong scaling tests were conducted with row-major data ordering and tall and thin datasets due to consistently being the slowest benchmark option and ideal for showing the best case strong scaling. These four graphs were chosen to display how the scaling changes between SIMD register width and with or without matrix symmetry. Each test was fitted against Amdahl's law using the damped least-squares method to approximate the parallel portion, p .

Cosine vs Manhattan Peak Memory Usage

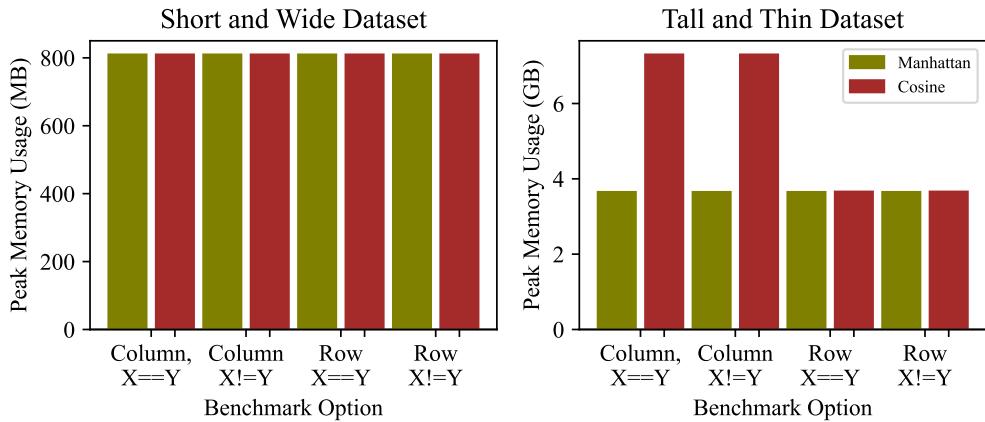


Figure 13: Comparison of the peak memory usage between Cosine and Manhattan distance.

Hassanat Distance Execution Time - AVX512

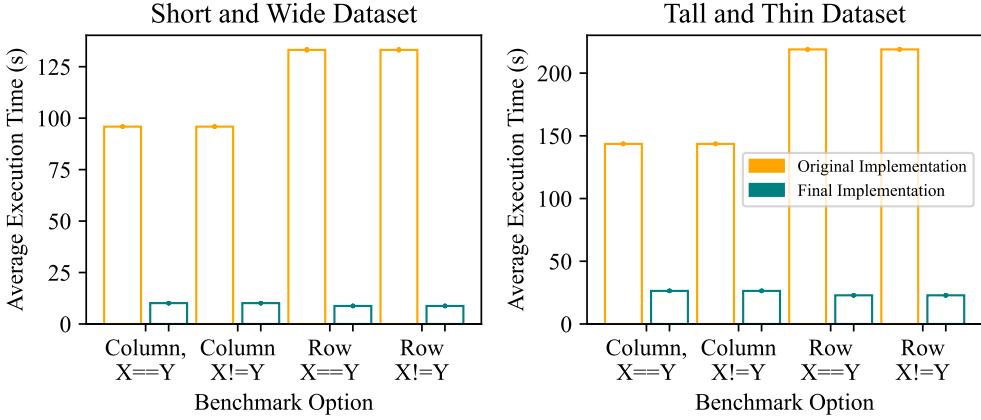


Figure 14: Comparison of the naive Hassanat implementation with the final implementation’s execution time on the AMD EPYC 9654 (AVX512) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots.

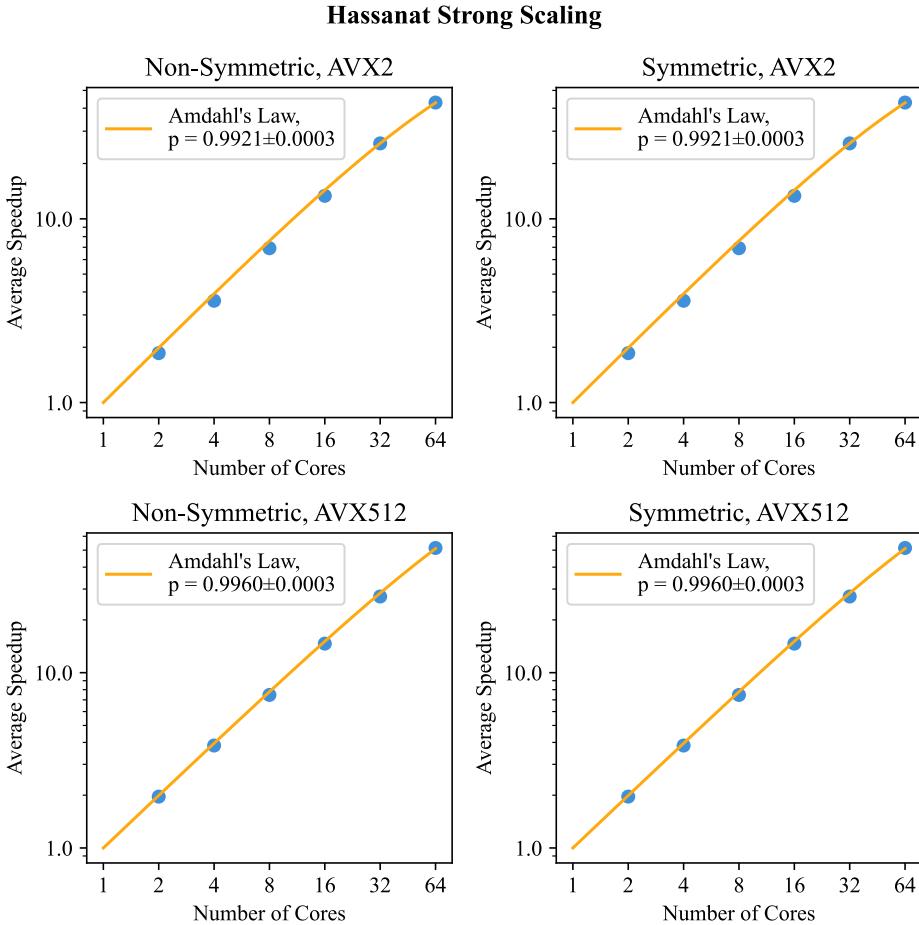


Figure 15: A collection of log-log strong scaling graphs for Hassanat distances. Strong scaling tests were conducted with row-major data ordering and tall and thin datasets due to consistently being the slowest benchmark option and ideal for showing the best case strong scaling. These four graphs were chosen to display how the scaling changes between SIMD register width and with or without matrix symmetry. Each test was fitted against Amdahl’s law using the damped least-squares method to approximate the parallel portion, p .

In Euclidean and Square Euclidean distance, there was a numerical stability issue being cause by BLAS: GEMM when combined with OpenMP. In the symmetric case, the diagonal of the distance matrix must have only zeroes. Due to numerical instability when performing the BLAS operations in a different order, these zeroes could sometimes be negative. The symmetric case has a fix for this which manually sets the diagonal of the distance matrix to be all zeroes before performing a square root or returning. However, this same fail-safe does not exist in the general case. Since the decision to use matrix symmetry is based on user input rather than any check to see if the matrices are equal, two equal matrices can be inputted while using the code from the general case, in which case these negative zeroes along the diagonal can appear. To remedy this a negative number check was added to the square root intrinsic function.

VIII CONCLUSION

In this project six different distance calculations were optimised at multiple layers of abstraction within AMD’s AOCL-DA library. Optimisations ranged from mathematical changes to the distance formulas, to low-level hardware-aware optimisation techniques using SIMD parallelism, to matrix manipulation within memory, to shared-memory multi-core parallelisation.

The key contributions of this project are as follows: benchmarking the initial AOCL-DA distance calculation implementations and identifying speedup opportunities, applying optimisations to improve performance, and benchmarking the final versions against the original code to quantify the achieved performance improvement. In addition, Hassanat distance, infamous for its poor performance, was implemented in the AOCL-DA library and subsequently optimised to be much more practical to use. Furthermore, a bug within the distance calculation code was identified and fixed within AOCL-DA.

Distance calculations in AOCL-DA have been made more performant across a range of parameters and hardware. The average speedup across all distances is approximately $4\times$ compared to their original or naive implementations. The optimised Hassanat distance achieves a roughly $15\times$ speedup over the naive design, the best improvement out of all the distances. Shared-memory multi-core parallelisation was successfully implemented with OpenMP in all distance calculations, with the approximate parallel fraction of the runtime ranging from 0.7 to 0.998.

Future work could explore more optimisation techniques such as using half-precision floating point numbers or combining multiple different precision numbers, balancing accuracy and speed to gain further speedups. The restriction on performing optimisations for CPUs only could be relaxed and optimisations that utilise GPUs or APUs could be explored too.

It would also be beneficial in future to benchmark these distance calculations within AOCL-DA against Scikit-Learn in Python. Scikit-Learn is not only a popular data analysis library but also often the baseline that a standard user might compare AOCL to. Furthermore, while the focus of this project was on improving the throughput of compute-bound distances with large datasets, Scikit Learn may have comparable or even better computation latency on small datasets. How the throughput and latency contributes to the performance of KNN and KMeans for different dataset sizes in AOCL-DA and Scikit Learn is worth exploring rigorously.

Furthermore, some optimisations in this project could be improved further.

For instance, Minkowski and Cosine distance relied on transposing the input matrices from column-major ordering to row-major ordering. If a native column-major compatible method were designed, these two distance calculations could yield a further improvement of up to 20% for column-major matrices based on the implemented Manhattan row- and column-major performances. Additionally, the blocking with the symmetric matrix does not refine the blocks, and thus does not achieve the optimal savings on redundant computations. Furthermore, the blocking approach for symmetric matrices can interfere with the standard OpenMP loop-parallel approach, which led to the decision to exclude any triangle symmetry optimisations from Cosine and Hassanat entirely. A more dynamic blocking method which skips all redundant distance calculations in the symmetric case would result in significant improvements for all distances, especially Cosine and Hassanat, and could also improve compatibility with OpenMP.

References

- AMD (2025), ‘AMD Optimizing CPU Libraries (AOCL)’.
URL: <https://www.amd.com/en/developer/aocl.html>
- Bottesch, T., Bühler, T. & Kächele, M. (2016), Speeding up k-means by approximating Euclidean distances via block vectors, in ‘Proceedings of The 33rd International Conference on Machine Learning’, PMLR, pp. 2578–2586. ISSN: 1938-7228.
URL: <https://proceedings.mlr.press/v48/bottesch16.html>
- Fiorini, S. (2016), ‘gene expression cancer RNA-Seq’.
URL: <https://archive.ics.uci.edu/dataset/401>
- Gallet, B. (2023), ‘Efficient Euclidean Distance Calculations and Distance Similarity Searches: An Examination of Heterogeneous CPU, GPU, and Tensor Core Architectures’, *Northern Arizona University*.
- GNU-Project (2005), ‘Auto-vectorization in GCC - GNU Project’.
URL: <https://gcc.gnu.org/projects/tree-ssa/vectorization.html#high-level>
- Goto, K. & Van De Geijn, R. A. (2008), ‘Anatomy of high-performance matrix multiplication’, *ACM Trans. Math. Softw.* **34**(3), 12:1–12:25.
URL: <https://doi.org/10.1145/1356052.1356053>
- Gowanlock, M. (2021), ‘Hybrid KNN-join: Parallel nearest neighbor searches exploiting CPU and GPU architectural features’, *Journal of Parallel and Distributed Computing* **149**, 119–137.
URL: <https://www.sciencedirect.com/science/article/pii/S0743731520304056>
- Gowanlock, M., Rude, C. M., Blair, D. M., Li, J. D. & Pankratius, V. (2017), ‘Clustering Throughput Optimization on the GPU: 31st IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017’, *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium, IPDPS 2017* pp. 832–841. Publisher: Institute of Electrical and Electronics Engineers Inc.
URL: <http://www.scopus.com/inward/record.url?scp=85027715033&partnerID=8YFLogxK>
- Hamerly, G. (2010), Making k-means even faster, in ‘Proceedings of the 2010 SIAM International Conference on Data Mining (SDM)’, Proceedings, Society for Industrial and Applied Mathematics, pp. 130–140.
URL: <https://pubs.siam.org/doi/abs/10.1137/1.9781611972801.12>
- Hamidieh, K. (2018), ‘Superconductivty Data (\textit{sic})’.
URL: <https://archive.ics.uci.edu/dataset/464>
- Hamilton8 (2021), ‘Hamilton - Durham University’.
URL: <https://www.durham.ac.uk/research/institutes-and-centres/advanced-research-computing/hamilton-supercomputer/>
- Hassanat, A., Alkafaween, E., Tarawneh, A. S. & Elmougy, S. (2022), Applications Review of Hassanat Distance Metric, in ‘2022 International Conference on Emerging Trends in Computing and Engineering Applications (ETCEA)’, pp. 1–6.
URL: <https://ieeexplore.ieee.org/document/10009844>
- Hassanat, A. B. (2014), ‘Dimensionality Invariant Similarity Measure’. arXiv:1409.0923 [cs].
URL: <http://arxiv.org/abs/1409.0923>
- HPC-Lab (2023), ‘Durham HPC Hardware Lab CPU compute nodes’.
URL: <https://durham.readthedocs.io/en/latest/hardwarelab/cpu.html>
- KDE (2025), ‘SDK / Heaptrack · GitLab’.
URL: <https://invent.kde.org/sdk/heaptrack>

Kelly, M., Longjohn, R. & Nottingham, K. (2023), ‘The UCI Machine Learning Repository’.

URL: <https://archive.ics.uci.edu>

Lindi, G. A. (2016), ‘Development of face recognition system for use on the NAO robot.’, *Stavanger University, Norway*.

Masek, J., Burget, R., Karasek, J., Uher, V. & Dutta, M. K. (2015), Multi-GPU implementation of k-nearest neighbor algorithm, in ‘2015 38th International Conference on Telecommunications and Signal Processing (TSP)’, pp. 764–767.

URL: <https://ieeexplore.ieee.org/abstract/document/7296368>

Mussabayev, R., Mladenovic, N., Mussabayev, R. & Jarboui, B. (2023), ‘How to Use K-means for Big Data Clustering?’, *Pattern Recognition* **137**, 109269.

URL: <https://www.sciencedirect.com/science/article/pii/S0031320322007488>

Mussabayev, R. & Mussabayev, R. (2024), Superior Parallel Big Data Clustering Through Competitive Stochastic Sample Size Optimization in Big-Means, in N. T. Nguyen, R. Chbeir, Y. Manolopoulos, H. Fujita, T.-P. Hong, L. M. Nguyen & K. Wojtkiewicz, eds, ‘Intelligent Information and Database Systems’, Springer Nature, Singapore, pp. 224–236.

OpenMP (2025), ‘OpenMP Specifications’.

URL: <https://www.openmp.org/specifications/>

Prasath, V. B. S., Alfeilat, H. A. A., Hassanat, A. B., Lasassmeh, O., Tarawneh, A. S., Alhasanat, M. B. & Salman, H. S. E. (2019), ‘Effects of Distance Measure Choice on K-Nearest Neighbor Classifier Performance: A Review’, *Big Data* **7**(4), 221–248. Publisher: Mary Ann Liebert, Inc., publishers.

URL: <https://www.liebertpub.com/doi/abs/10.1089/big.2018.0175>

Ramasubramanian, V. & Paliwal, K. K. (1992), ‘An efficient approximation-elimination algorithm for fast nearest-neighbour search based on a spherical distance coordinate formulation’, *Pattern Recognition Letters* **13**(7), 471–480.

URL: <https://www.sciencedirect.com/science/article/pii/0167865592900647>

Sadílek, V. & Vořechovský, M. (2018), ‘Evaluation of pairwise distances among points forming a regular orthogonal grid in a hypercube’, *Journal of Civil Engineering and Management* **24**(5), 410–423. Number: 5.

URL: <https://transport.vilniustech.lt/index.php/JCEM/article/view/5189>

Van Zee, F. G. & van de Geijn, R. A. (2015), ‘BLIS: A Framework for Rapidly Instantiating BLAS Functionality’, *ACM Trans. Math. Softw.* **41**(3), 14:1–14:33.

URL: <https://doi.org/10.1145/2764454>

APPENDIX

1 Euclidean

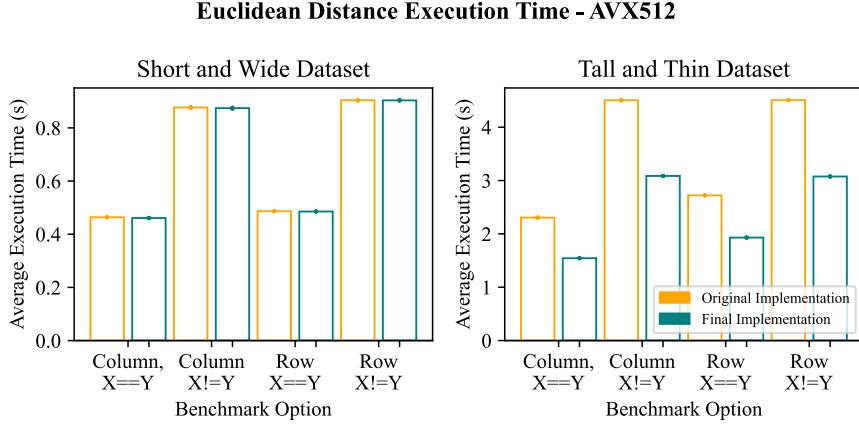


Figure 16: Comparison of the original Euclidean implementation with the final implementation’s execution time on the AMD EPYC 9654 (AVX512) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots.

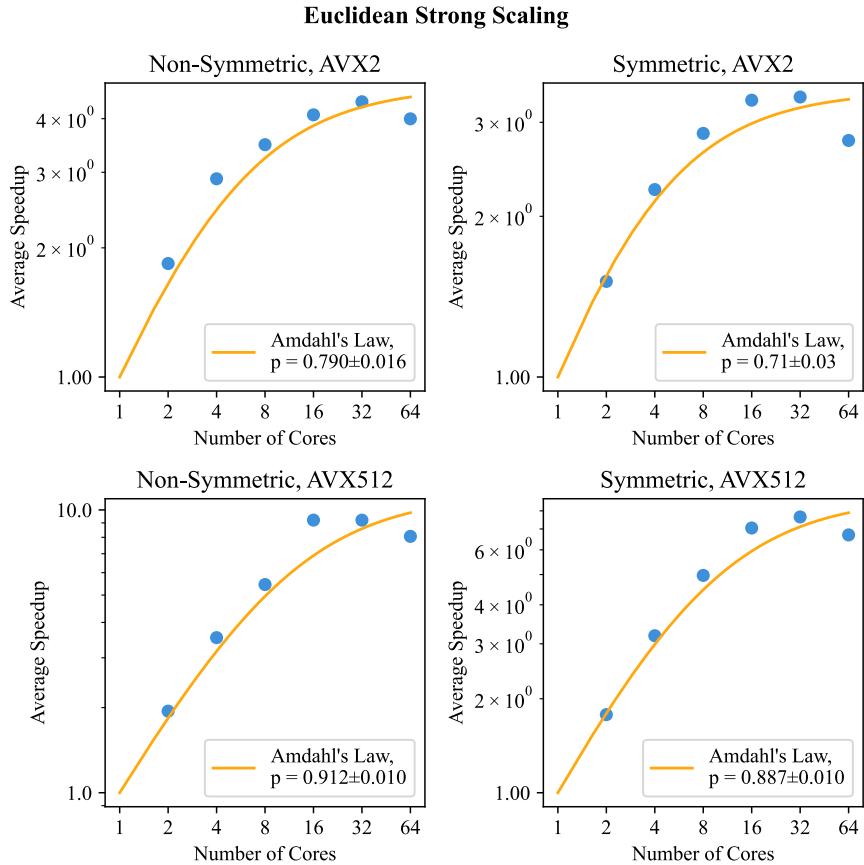


Figure 17: A collection of log-log strong scaling graphs for Euclidean distances. Strong scaling tests were conducted with row-major data ordering and tall and thin datasets due to consistently being the slowest benchmark option and ideal for showing the best case strong scaling. These four graphs were chosen to display how the scaling changes between SIMD register width and with or without matrix symmetry. Each test was fitted against Amdahl’s law using the damped least-squares method to approximate the parallel portion, p .

2 Manhattan

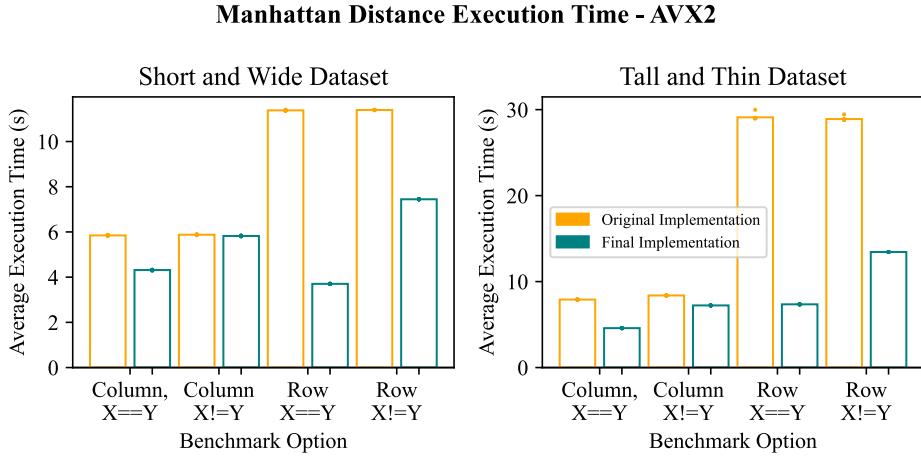


Figure 18: Comparison of the original Manhattan implementation with the final implementation’s execution time on the AMD EPYC 7702 (AVX2) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots.

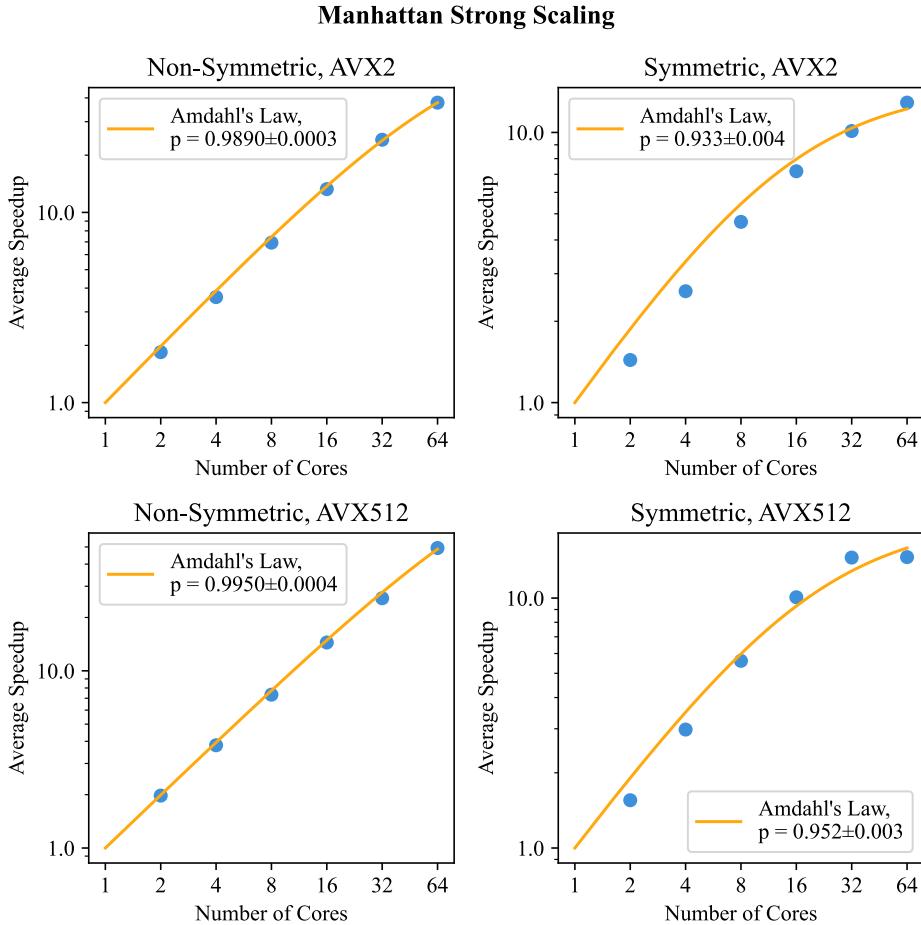


Figure 19: A collection of log-log strong scaling graphs for Manhattan distances. Strong scaling tests were conducted with row-major data ordering and tall and thin datasets due to consistently being the slowest benchmark option and ideal for showing the best case strong scaling. These four graphs were chosen to display how the scaling changes between SIMD register width and with or without matrix symmetry. Each test was fitted against Amdahl’s law using the damped least-squares method to approximate the parallel portion, p .

3 Minkowski

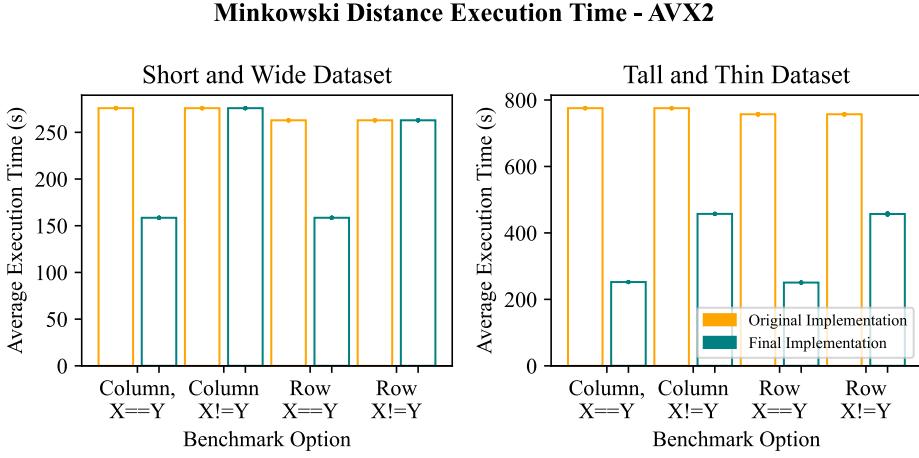


Figure 20: Comparison of the original Minkowski implementation with the final implementation’s execution time on the AMD EPYC 9654 (AVX2 forced) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots. Compiled with AOCC 5.0.0.

4 Cosine

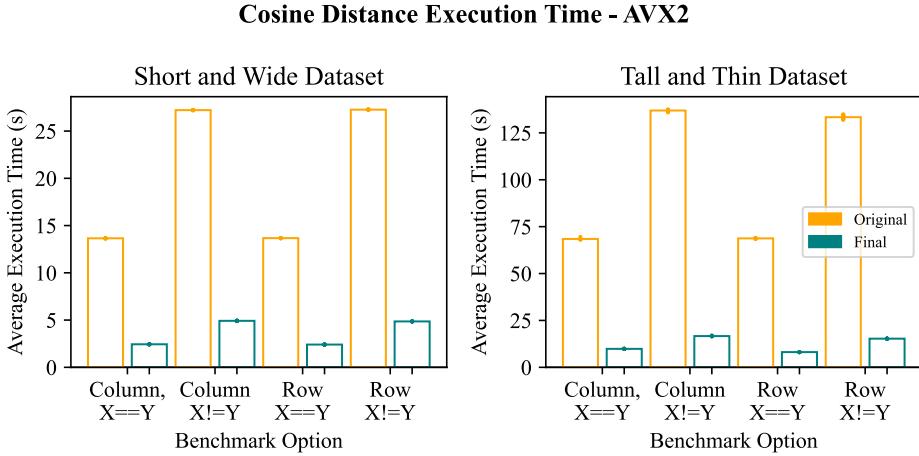


Figure 21: Comparison of the original Cosine implementation with the final implementation’s execution time on the AMD EPYC 7702 (AVX2) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots.

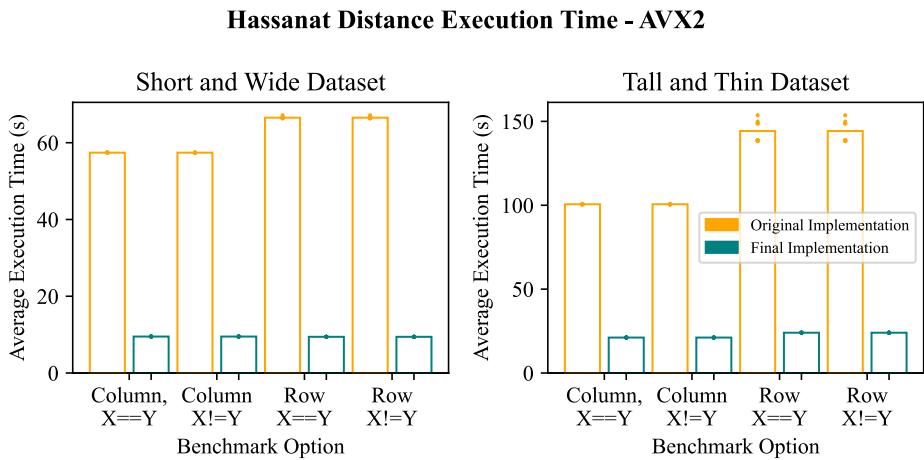


Figure 22: Comparison of the original Hassanat implementation with the final implementation's execution time on the AMD EPYC 7702 (AVX2) with both the short and wide, and tall and thin datasets. The average execution times are plotted as bars with each of the ten individual execution times plotted as dots.