Time-Quality Tradeoff of MuseHash Query Processing Performance

Maria Pegia[1,3,§ [0000−0003−2643−0028]], Ferran Agullo Lopez[2,§ [0000−0002−7276−2472]], Anastasia Moumtzidou[1 [0000−0001−7615−8400]], Alberto Gutierrez-Torre[2 [0000−0002−5548−3359]], Björn Þór Jónsson[3 [0000−0003−0889−3491]], Josep Lluís Berral García[4,2 [0000−0003−3037−3580]], Ilias Gialampoukidis[1 [0000−0002−5234−9795]], Stefanos Vrochidis[1 [0000−0002−2505−9178]], and Ioannis Kompatsiaris[1 [0000−0001−6447−9020]]

[1] Information Technologies Institute - Centre for Research and Technology Hellas, Thessaloniki, Greece {mpegia,moumtzid,heliasgj,stefanos,ikom}@iti.gr
[2] Barcelona Supercomputing Center, Barcelona, Spain {alberto.gutierrez,ferran.agullo}@bsc.es
[3] Reykjavik University, Reykjavik, Iceland {mpegia22, bjorn}@ru.is
[4] Universitat Politècnica de Catalunya, Barcelona, Spain josep.ll.berral@upc.edu

Abstract. Nowadays, massive quantities of multimedia data, such as videos, images, text and audio, are generated by various applications on smartphones, drones and other devices. To facilitate efficient retrieval from these multimedia collections, we need (a) effective media representation and (b) efficient indexing and query processing approaches. Recently, the MuseHash approach was proposed, which can effectively represent a variety of modalities, improving on previous hashing-based approaches. However, the interaction of the MuseHash approach with existing indexing and query processing approaches has not been considered. This paper provides a systematic evaluation of a set of state-of-the-art approximate nearest neighbor search algorithms for image retrieval, when applied to the MuseHash approach, providing quantitative comparison results and evaluating the use of High-Performance Computing (HPC) infrastructures. An extensive set of experiments on a benchmark aerial dataset and on a real life-log dataset demonstrates the effectiveness of employing hashing and ANN techniques with HPC, resulting in reduced computational time.

Keywords: Approximate nearest neighbor · Information Search · Optimization · High-Performance Computing.

1 Introduction

The nearest neighbor problem is fundamental problem in data mining and machine learning aiming to identify data points that are closest to a specific query

§Equal contribution

point, using a defined distance metric. It is widely employed in retrieval tasks to find the most similar data points to a given query, such as image retrieval.

Challenges in nearest neigbour method applications arise when dealing with large-scale high-dimensional datasets, given the *curse of dimensionality*. As dimensionality or dataset size increases, the *exact* nearest neighbor search becomes intractable. As a tractable alternative, *Approximate* Nearest Neighbor (ANN) methods provide approximate solutions to the same problem with reduced computational complexity. A popular approach to ANN search is hashing, by mapping data points to compact binary codes so that similar points have similar codes. This allows for efficient retrieval by indexing and searching in the hash code space. Hashing-based ANN methods, such as Locality-Sensitive Hashing (LSH) [6] or Deep Hashing [13], are examples of efficient solutions when managing large-scale datasets. However, challenges remain in designing effective hash functions, optimizing search efficiency, and balancing retrieval accuracy and computational cost.

Researchers have proposed various solutions to these challenges, including the design of effective hash functions tailored to specific data distributions, learning-based methods for generating discriminative hash codes, and the utilization of parallel computing or distributed systems for efficient search. Nevertheless, existing hashing methods often tend to balance storage and solution effectiveness, whereas ANN methods focus on achieving a balance between efficiency and result accuracy. Combining these approaches can potentially provide a solution that optimizes performance in terms of time, storage, and retrieval.

In this paper, we present a comprehensive analysis of query processing performance of the MuseHash approach, a recent multimodal hashing approach to media representation. Moreover, to the best of our knowledge, no study has explored the performance of multi-modal hashing methods as base for several ANN approaches or their performance on HPC resources.

The main contributions of this paper are summarized as follows:

- Thorough examination of the advantages and limitations of existing ANN methods as applied to MuseHash.
- Evaluation of the performance of classic methods and ANN with hashing methods when utilizing HPC (multi-threading and GPUs).
- Analysis of the impact of HPC (multi-threading and GPUs) on retrieval speed and efficiency through experimental evaluations.

After this study, we conclude that the MuseHash offers an effective and compact representation to retrieve the data. Moreover, longer hash lengths provide better results but are slower on CPU. When using GPUs, however, this drawback can be addressed reaching speeds similar to using shorter hash lengths.

The remainder of this paper is organized as follows. Section 2 provides an overview of the relevant state-of-the-art ANN research in the field. Section 3

presents in short our approach, some ANN techniques used in this research and the HPC approach. Section 4 showcases the experimental results, and finally, Section 5 offers a concise summary with the conclusions.

2 Related Work

The nearest neighbor problem has been extensively studied in the field of computer science and information retrieval for several decades [20]. It plays a crucial role in various applications, including pattern recognition, data mining, image retrieval, and recommendation systems. The goal is to efficiently find the most similar data point(s) to a given query point in a dataset. On the other hand, methods for finding the exact nearest neighbors seek the closest data point with precision, ensuring accuracy but incurring significant computational costs, particularly when dealing with sizable datasets.

In recent years, there has been significant progress in developing Approximate Nearest Neighbor (ANN) search algorithms [1], with approximate solutions while expecting to maintain accuracy. This is a trade off between retrieval accuracy versus faster query processing, making it suitable for large-scale datasets with improved efficiency. The evaluation of ANN algorithms typically involves measuring their retrieval accuracy and efficiency. Performance is commonly evaluated using metrics like precision, recall, query time, and index construction time. Moreover, benchmark datasets like MIRFlick25K [8] and NUSWIDE [5] have been established to facilitate fair comparisons among different approaches [19].

Overall, the research on approximate nearest neighbor search algorithms has made significant strides in balancing retrieval accuracy and computational efficiency [3], [18], [2], [16], [15], [14], [10], [7], [9]. These advancements enable efficient similarity search in large datasets, making them applicable to a wide range of domains and applications. However, there are still challenges to address, such as improving the trade-off between accuracy and efficiency, managing high-dimensional data, and developing robust solutions for dynamic datasets.

Furthermore, the exploration of hashing techniques to enhance existing ANN methods has aimed to tackle challenges related to improving the trade-off between accuracy and efficiency, effectively handling high-dimensional data, and creating resilient solutions for dynamic datasets [12] . By incorporating such hashing techniques, ANN methods can improve the trade-off between accuracy and efficiency, because they can handle high-dimensional data more effectively and with less computational cost.

Moreover, work has been done to leverage current hardware capabilities, aside from algorithmic improvements. For example, when dealing with data that cannot be accommodated in memory, ANN methods like DiskANN [22] or SPANN [23] propose to use data locality and fast disk storage (Solid State Disks (SSD)). Also, multi-threading has been exploited in ANN, with examples such as SCANN [10], or through threading parallelism at the level of query processing. This allows a

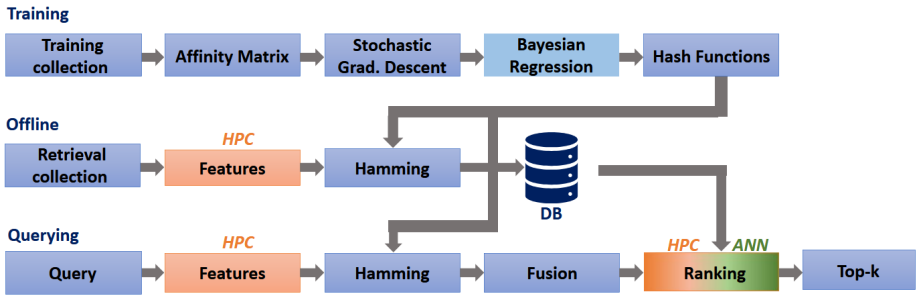

Fig. 1. Overview of this research.

machine to make use of all the available parallelism in the installed CPU, drastically rising performance even though linear speed up is not warranted. On the other hand, using Graphics Processing Units (GPUs) for computing has been popularized specially in mathematical processing like Neural Networks. This is also the case of ANN. SONG [24] algorithm has been specifically co-designed to make use of GPUs, beating similar algorithms that are CPU-based.This highlights that when developing new ANN algorithms, it is advantageous to implement them efficiently while being mindful of the capabilities of the underlying hardware.

3 Methodology

In the subsequent subsections, we present concise descriptions of the three main components of our study (Figure 1): the MuseHash method (Section 3.1) (blue blocks), the state-of-the-art ANN methods (green block) that have been integrated into our research (Section 3.2), and the optimization techniques (orange blocks) utilized for HPC infrastructure (Section 3.3). The synergy of these three elements is pivotal in our research, with MuseHash providing a robust foundation for multi-modal data management, state-of-the-art ANN methods endowing us with advanced learning capabilities, and HPC optimization amplifying our computational prowess. This amalgamation enables us to achieve exceptional efficiency and performance, effectively addressing the challenge of longer hash codes in shorter time frames while optimizing overall performance.

3.1 MuseHash

MuseHash [19] is an efficient indexing and retrieval algorithm designed for multimedia data. It leverages a combination of modalities in its queries, such as visual and temporal aspects, to deliver highly relevant results. The method comprises three main phases: training, offline, and querying. In the **training phase**, hash functions are generated from the training collection using Bayesian ridge regression. These functions map feature vectors from each modality to the Hamming

space. Affinity matrices are created based on ground truth labels, and semantic probabilities are derived from these matrices. During the **offline phase**, features are extracted from the retrieval set for each modality. Using the learned hash functions, hash codes are computed and stored in a database, ensuring efficient storage and retrieval of multimedia data. Finally, in the **querying phase**, hash functions learned in the previous steps are applied to a given query. Unified hash codes are generated from query-specific hash codes using the XOR operation. The database is queried using Hamming distances, leading to the retrieval of top-k relevant results. Overall, MuseHash combines supervised learning, Bayesian regression, and Hamming distance-based retrieval to significantly enhance the accuracy and efficiency of multimedia data retrieval.

3.2 ANN Methods

From the different distinguished state-of-art ANN methods, we select the following ones because of their diverse and complementary characteristics based on the work of Aumúller et al. [1]: tree-based structures, graph-based structures, pruning techniques, brute-force approaches and baseline methods.

Tree-based methods BallTree [3] is a tree-based data structure that uses hyper-spheres to partition the data space and construct a tree hierarchy. CKDTree [18] extends the KD-tree algorithm to support nearest neighbor search in multiple dimensions by using hyper-rectangles. Random Projection Tree (RPT) methods, like Annoy (Approximate Nearest Neighbors Oh Yeah) [2], utilize random projections to split data points and build index structures for fast approximate nearest neighbor search. Annoy is particularly renowned for its simplicity and efficiency in handling high-dimensional data. On the contrary, PyNNDescent [16] employs randomized k-d trees, combining randomized partitioning and nearest neighbor search to efficiently navigate the tree structure and find approximate nearest neighbors.

Graph-based methods The HNSW (Hierarchical Navigable Small World)[15] organizes arranges the dataset into a hierarchical structure comprising small-world graphs, enabling efficient approximate nearest neighbor search while maintaining minimal memory usage. SW-graph (Small World Graph) [14] combines the properties of a small-world graph and locality-sensitive hashing. It balances retrieval accuracy and efficiency by exploiting the local and global structures of the data.

Pruning methods SCANN (Scalable Nearest Neighbors) [10] utilizes locality-sensitive hashing techniques for efficient approximate nearest neighbor search. It offers both single-threaded and multi-threaded implementations, making it suitable for large-scale datasets.

Brute-force methods Ball, KD-Tree, BruteForce, and BruteForce-BLAS [7], which are simple and straightforward methods for solving the nearest neighbor search problem. They involve calculating the distances between the query point and all data points to find the closest neighbors. While these methods guarantee

accurate results, they can be computationally expensive, especially for large datasets. BruteForce-BLAS improves efficiency by leveraging the BLAS library for faster distance calculations. These brute-force approaches serve as baseline algorithms for evaluating the performance of more advanced ANN methods.

Baseline methods Dummy Algorithm Multi-Threaded (Dummy-Algo-MT) [9] and Dummy Algorithm Single-Threaded (Du-mmy-Algo-ST), offer simpler and more generic implementations. Dummy-Algo-MT performs a multi-threaded brute-force search, leveraging parallel processing for improved performance. Dummy-Algo-ST is a single-threaded version that provides a basic implementation for comparison. These baseline approaches serve as reference points to evaluate the efficiency and effectiveness of more advanced ANN algorithms.

3.3 Performance and HPC scalability

In this subsection we define how the code is optimized to use the available hardware as efficiently as possible. Our approach is driven by the usage of parallel computation using multithreading and GPUs. This approach can also be extended to use multiple machines if the infrastructure is available.

Feature extraction The method to extract the features is used in the offline and querying phases as show in Figure 1. For this particular step the most viable optimization is to use GPUs with NVIDIA CUDA code as this part implies a forward pass of a Neural Network. Moreover, multiple GPUs were used at the same time as the feature extraction from each sample is independent. Including 4 GPUs implied a speed-up of 3.1x compared to just using 1 GPU. This is also effective for use in the querying phase.

Retrieval In the retrieval phase, hashes have a high impact on performance. On one side, the hashes provide a compact description of the multimodal information from each data point. This compact code can be then used to find the similarities to other data points. The computations needed to compare these hashes versus the complete multimodal information is lower as information is compressed, therefore providing a faster way to find similar data points. Similarly, the length of this hash also has an impact on performance as longer hashes require more computation to be compared. Therefore there is a trade-off between the representation accuracy of the data points given by the length of the hashes and the actual performance in terms of queries per second.

To evaluate the methods we propose to use two different approaches: data parallelism and query parallelism. The first approach is based on creating partitions of the data so that it can be distributed to different processes to search. The second approach is based in having the data accessible by a process pool and whenever a new query is done, this query is assigned to one of these processes.

To evaluate the scalability of hash lengths with different methods the ANN benchmark [1] was used and adapted to be used in a HPC SLURM based cluster.

In particular, modifications were added to execute several queries in parallel and the cuML Python package was used to include the CUDA *bruteforce* approach.

4 Experiments

4.1 Datasets

In our multimodal retrieval case, we leverage the use of two distinct datasets: the AU-AIR dataset [4] and the LSC'23 dataset [11]. These datasets provide us with diverse modalities and rich content for performing comprehensive and accurate retrieval. Table 1 present the way of splitting the dataset and its available modalities.

Table 1. Two benchmark datasets used in experiments.

	Ground Truth	Modalities				Collection Sizes			
Dataset	Labels	Image	Text	Time	Location	Whole	Retrieval	Training	Test
AU-AIR	8	✓	✗	✓	✓	32283	32183	2000	100
LSC'23	135	✓	✓	✓	✓	40926	40676	4000	250

AU-AIR The AU-AIR dataset consists of eight video clips recorded for aerial traffic surveillance at a specific intersection in Aarhus, Denmark. The videos were captured on windless days and exhibit varying lighting conditions due to the time of day and weather. The dataset has a resolution of 1920x1080 pixels and comprises 32,823 frames extracted from the raw videos at a rate of five frames per second to avoid redundancies.

LSC'23 The LSC'23 dataset was generated by an active lifelogger and spans a duration of 18 months. This dataset is composed of three password-protected files. The first file contains the core image dataset, consisting of wearable camera images in fully redacted and anonymized form. These images, captured using a Narrative Clip device, have a resolution of 1024 $\times$ 768 pixels. To respect privacy requirements, all faces, readable text, and certain scenes and activities have been manually removed. Due to huge imbalance on the dataset, we filtered the original dataset with using only the data that include label information and with label frequency greater than 257. Therefore, out of a total of 41,100, the preprocessed dataset used in our experiments comprises 40,926 data points.

In order to evaluate scaling of the ANN approaches, three other synthetic datasets are randomly generated using the uniform distribution (i.e., each hash possible has equal probability of being present) over all the space defined by the hash length, simulating the MuseHash enconding. The training sizes are 28000, 112000 and 448000 samples with different hash lengths (32, 128, 512 and 2048) and 450 samples for testing (querying).

By evaluating our method separately on these distinct datasets, we can provide a comprehensive analysis of its performance in different retrieval scenarios. This approach allows us to tailor our evaluation metrics and techniques to the specific characteristics and challenges posed by each dataset. Additionally, it enables us to highlight the strengths and limitations of our method in different multimodal retrieval settings, contributing to a more thorough understanding of its capabilities.

4.2 Experimental Settings

In our experiments, we assess the performance of MuseHash and ANN methods using several evaluation metrics, including mean Average Precision (mAP), precision, recall, and f-score. The following feature vectors from each modality are used as input representations for our evaluation:

Image 4096-D vector from the fc-7 layer of pre-trained VGG16 network[5].

Textual 768-D vector from pre-trained BERT model[6].

Temporal 191-D vector representation for LSC'23 and 203-D vector for AU-AIR based on the work [19]. The first four coordinates of the temporal feature belong to the 4 digits of the year, the next 12 digits to the one-hot-encoding for month, the next 31 digits to the one-hot encoding for day, the next 24 to the one-hot-encoding for hours, the next 60 to the one-hot encoding for minutes, the next 60 to the one-hot-encoding for seconds. It should be noted that only the AU-AIR dataset has 12 last additionally digits, which correspond to a binary encoding of microseconds.

Spatial 3-D vector representation. Each location corresponds to a 3-D vector with values (altitude, longitude, altitude).

We compute hash codes using MuseHash for different bit lengths $d_c = 16, 32, 64, 128, 256, 512, 1024, 2048$. Moreover, we conduct experiments using the features of each modality as well as for different combination of hash codes.

4.3 Retrieval results using CPU

In our CPU experiments, we explore retrieval results using two datasets: AU-AIR and LSC'23, which some of them shown in Figure 2, due to page limitation.

Regarding AU-AIR, it's evident that incorporating additional modalities enhances precision and F-score. Additionally, as the hash code length increases, the methods exhibit improved performance, reaching their peak at 2048. However, it should be noted that the queries/s decrease when increasing the hash size. As far as LSC'23 is concerned, we observe that when textual information is

[5]https://github.com/Leo-xxx/pytorch-notebooks/blob/master/Torn-shirt-classifier/VGG16-transfer-learning.ipy

[6]https://github.com/maknotavailable/pytorch-pretrained-BERT

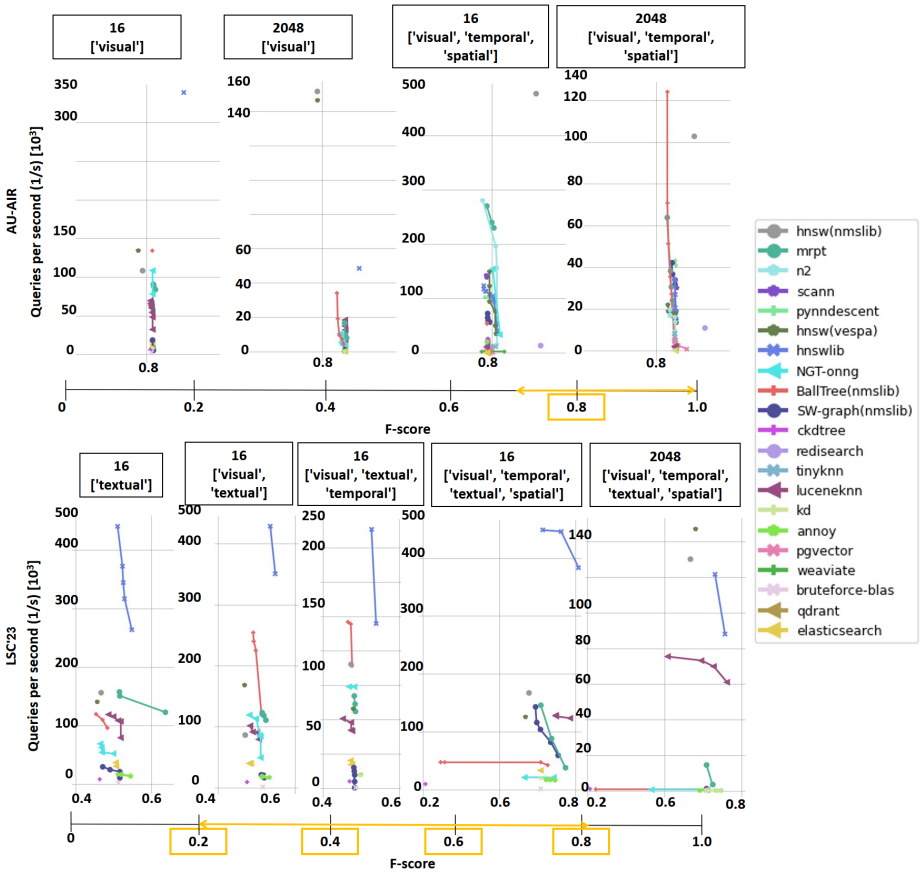

Fig. 2. F-score values for ANN and bruteforce methods on AU-AIR (first row) and LSC'23 (second row) datasets for 16 and 2048 bits considering only visual or textual or combination of modalities.

incorporated, the precision and recall are enhanced. This observation aligns with the success of textual queries using CLIP models in the VBS competition [17]. Using all available modalities also boosts retrieval results.

4.4 Scalability analysis of hashes and NN methods

In this section we experiment in different scenarios with the AU-AIR dataset using MuseHash encoding and the three synthetic datasets and different methods. First, we explore the scalability options of the bruteforce algorithm as a base reference and explore the trade-off of parallelizing the code by doing data parallelism and query parallelism and also using GPUs. Second, we explore one of the tree algorithms, ball trees, in order to understand their usability for the hashing methods as they are known to be greatly affected by the curse of di-

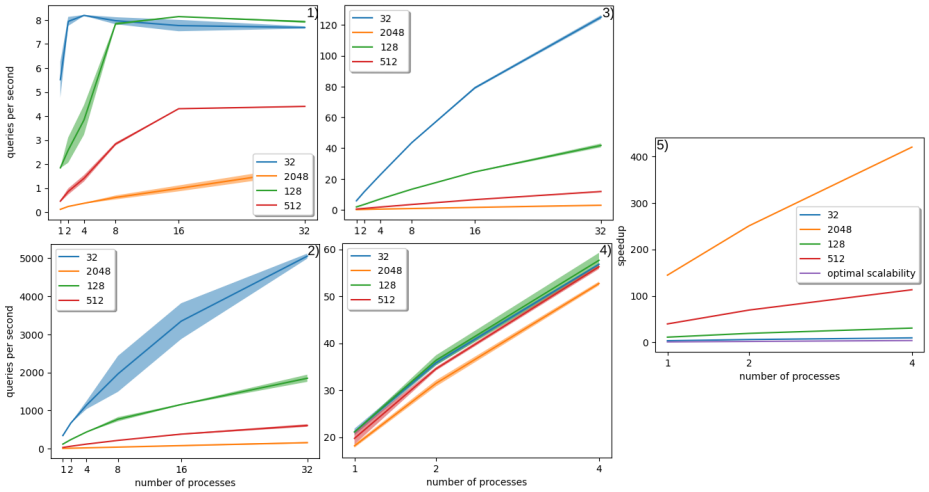

Fig. 3. Comparison of bruteforce approaches (dataset used in parenthesis): 1) Data parallelism (large), 2) Query Parallelism (AU-AIR), 3) Query parallelism (large), 4) Query parallelism GPU (large). 5) Query parallelism GPU scaling speed-up (large). Optimal scalability refers to the lineal scalability for CPU

mensionality [21]. Finally we experiment the scalability of PyNNDescent [16], another tree-based algorithm, with the different hash lengths to study the scalability of AAN methods with hashes and see if the curse of dimensionality affects it. Each experiment is performed 5 times. Then the queries/s are averaged and the error margins are calculated with the standard deviation. These experiments were performed with a machine with 2 IBM Power9 8335-GTH @ 2.4GHz (20 cores, 4 threads/core), 512 GB of RAM and 4 NVIDIA V100 GPU with 16GB RAM. For these experiments we have included GPU bruteforce from CuML and CPU bruteforce from scikit learn.

Experiment 1: Bruteforce comparison In this experiment we focus on comparing different bruteforce approaches. Notice that these approaches are the most computing intensive ones as they traverse all the dataset to find the exact match. In particular we are interested in comparing the parallelization by data (scanning the data with more than one process at the time) and query parallelization (one query per process, multiple queries at the same time). In Figure 3 it can be seen that data parallelism (Figure 3.1) scales poorly when compared to query parallelism (Figure 3.3) using the large dataset, even though bruteforce is easily parallelizable. On large datasets, query parallelism is 2x faster than data parallelism. Similar results are obtained with the other datasets as shown in Figure 3.2 with the AU-AIR dataset. Query parallelism scales almost linearly with respect to the number of processes for all the datasets.

Finally, GPU approach provides better results than the regular CPU multi-process approach. For example, with one GPU the queries/s is the same that

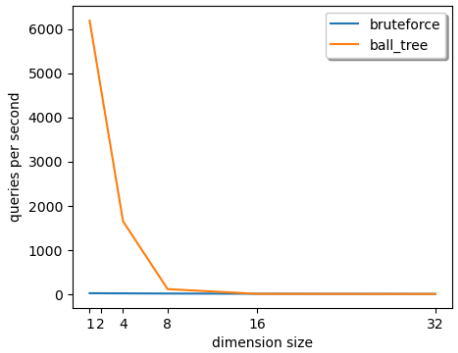

Fig. 4. Comparison of bruteforce and ball tree (leaf size 100) respect to the dimensions in the large synthetic dataset.

the best case with 32 processes. Moreover, GPUs provide almost the same speed for each hash length. Figure 3.5 shows the speed up compared to the baseline. Compared with the optimal scalability for CPU, the hash length 2048 shows speed ups from 150x to 400x, showing that GPUs are adequate in general and also desirable for long hash lengths.

Experiment 2: Ball tree vs Bruteforce Trees are a classic method to create an index to find similar data faster than comparing all the dataset, however they are limited. In the retrieval experiments, we saw that the ball tree can be performant, however it has drawbacks. Here we compare a ball tree with 100 elements per leaf and the bruteforce without any kind of parallelism using the large synthetic dataset. Ball trees suffer of the curse of dimensionality [21], rendering them to be similar to bruteforce when a high number of dimensions is used for the index, which is relevant when using longer hashes. As we can observe in Figure 4, after including 8 dimensions the performance in queries/s gets degraded to the level of the base bruteforce algorithm. This fact shows that this kind of approach is not doable for big data dimensions, as is in the case of the hashes.

Experiment 3: PyNNDescent Figure 5 shows the results of scaling PyNNDescent using multi-threading. It can be seen that for all the datasets the algorithm scales well up to 8 processes for 32 and 128 hash lengths, then the performance degrades. Longer hash sizes affect dramatically the performance of the algorithm, however scaling with 8 threads usually provides between a 3x and a 4x speed-up and after a given point the performance does not fall, therefore confirming that PyNNDescent is better suited than Ball tree when using a big number of dimensions.

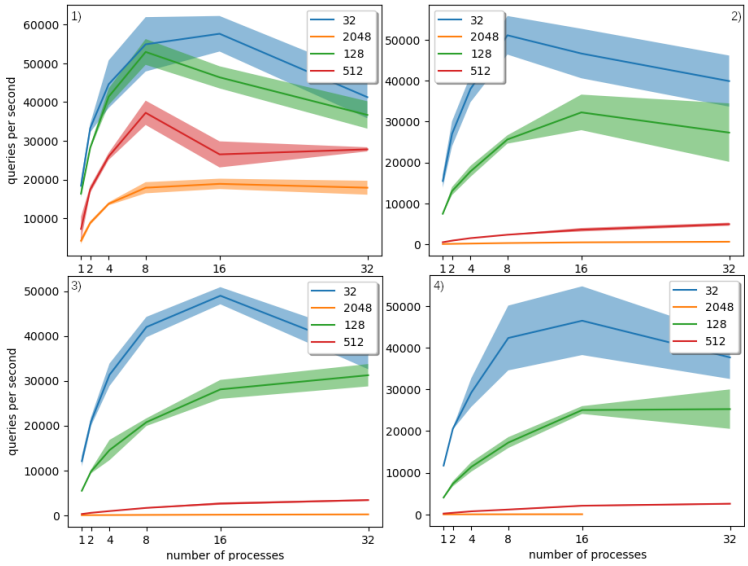

Fig. 5. Scaling of PyNNDescent using CPU parallelism for the different hash lengths. Average of queries/s with error margins for four different datasets: 1) AUAIR, 2) synthetic small, 3) synthetic medium, 4) synthetic large. Execution of 2048 hash length and 32 processes with the large dataset did not finish in a reasonable time (2 hours).

4.5 Discussion

In both datasets, we observed from Figure 2 that the Hnswlib, a fast approximate nearest neighbor search method, outperformed other methods. What's particularly intriguing is that ANN methods consistently outshine brute-force scenarios in terms of retrieval performance. This suggests the efficiency and effectiveness of these ANN techniques in enhancing retrieval results across both AU-AIR and LSC'23 datasets.

Regarding scalability, we have seen that longer hash lengths have a big impact on the performance, establishing a trade-off between precision and speed. The accuracy-performance trade-off is also seen in modern ANN methods, as shown in Figure 5. For long hashes, GPUs show a promising way to avoid this trade-off due to their capability of computing many data elements at the same time. Moreover, query parallelism was shown to be better than data parallelism in terms of performance and also in terms of adapting the retrieval procedure. In general, scalable solutions can improve drastically the performance but this scalability has to be studied for each approach applied as more hardware does not directly correlate with more performance, as seen in the experiments.

5 Conclusion

In this paper, we examine the multi-modal hashing codes produced by MuseHash as input for several state-of-the-art ANN methods and study their performance on HPC infrastructure. On one hand, the usage of hashing enables efficient searching similar data in a multi-modal space and using them over ANN algorithms improve their efficiency reducing computing time by doing an approximate search instead of an exhaustive one. Our experiments show the effectiveness of combining MuseHash with state-of-the-art ANN methods, showcasing Hnswlib as a top performer. Additionally, HPC resources can be used to reduce the execution time by exploiting their capabilities. The scalability depends on the approach and requires thoughtful consideration beyond hardware scaling as tailoring the application to the actual resources is usually required. Both CPU multithreading and GPUs can be leveraged to reduce execution time. Particularly, in case of longer hashes, GPUs are shown to be effective, opening a research path for future work by combining ANN with long enough hashes that provide accurate results and using GPUs to enhance their efficiency and response time.

Acknowledgment

This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreements H2020-101070250 CALLISTO, H2020-101070262 WATERVERSE, H2020-101080090 ALLIES and the Spanish Ministry of Science (MICINN), the Research State Agency (AEI) and European Regional Development Funds (ERDF/FEDER) under grant agreement PID2021-126248OB-I00, MCIN/AEI/10.13039/ 501100011033/FEDER, UE, Severo Ochoa Center of Excellence CEX2021-001148-S-20-3 and the Generalitat de Catalunya (AGAUR) 2021-SGR-00478.

References

1. Aumúller, M., Bernhardsson, E., Faithfull, A.: : A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. Information Systems (2019). https://doi.org/10.1016/j.is.2019.02.006
2. Bernhardsson, E.: Annoy, `https://github.com/spotify/annoy`.
3. Boytsov, L., Naidan, B.: Engineering efficient and effective non-metric space library. In: Similarity Searching and Applications, (2013).
4. Bozcan, I., Kayaan, E.: AU-AIR: A Multi-modal Unmanned Aerial Vehicle Dataset for Low Altitude Traffic Surveillance. IEEE International Conference on Robotics and Automation (ICRA). https://doi.org/ https://doi.org/10.48550/arXiv.2001.11737 , (2020).
5. Chua, T.-S., Tang, J., Hong, R., Li, H., Luo, Z., Zheng, Y.: NUS-WIDE: a real-world web image database from National Uni- versity of Singapore. ACM International Conference on Image and Video Retrieval (ICMR), 1–9. https://doi.org/https://doi.org/10.1145/1646396.16464, (2009).
6. Durmaz, O., Bilge, H. S.: Fast image similarity search by distributed locality sensitive hashing. Pattern Recognition Letters. 361-369. https://doi.org/https://doi.org/10.1016/j.patrec.2019.09.025, (2019).
7. Francis, M., Durme, B.: Fast exact fixed-radius nearest neighbor search based on sorting. arXiv, https://doi.org/https://doi.org/10.48550/arXiv.1910.02478, (2019).
8. Huiskes, M. J., Lew, M.: The MIR flickr retrieval evaluation. Proceedings of the 1st ACM international conference on Multimedia information retrieval, 39–43. https://doi.org/https://doi.org/10.1145/1460096.1460, (2008).
9. Geiger, M. J.: A multi-threaded local search algorithm and computer implementation for the multi-mode, resource-constrained multi-project scheduling problem. In: European Journal of Operational Research, vol. 256, 729-741, https://doi.org/https://doi.org/10.1016/j.ejor.2016.07.024, (2017).
10. Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., Kumar, S.: Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In: International Conference on Machine Learning, `https://arxiv.org/abs/1908.10396`, (2020).
11. Gurrin, C., Jónsson, B. Þ., Nguyen, D. T. D, Healy, G., Lokoc, J., Zhou, L., Rossetto, L., Tran, M.-T., Hürst, W., Bailer, W., Schoeffmann, K.: Introduction to the Sixth Annual Lifelog Search Challenge, LSC'23. International Conference on Multimedia Retrieval (ICMR). 678–679. https://doi.org/https://doi.org/10.1145/3591106.3592304, (2023).
12. Li, W., Zhang, Y., Sun, Y., Wang, W., Li, M., Zhang, W., Lin, X.: Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement, IEEE Transactions on Knowledge and Data Engineering. https://doi.org/10.1109/TKDE.2019.2909204, (2019).
13. Lu, J., Liong, V. E., Zhou, J.: Deep Hashing for Scalable Image Search. IEEE Transactions on Image Processing. 2352 - 2367. https://doi.org/https://doi.org/10.1109/TIP.2017.2678163, (2017).
14. Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V.: Approximate nearest neighbor algorithm based on navigable small world graphs. Information Systems, vol. 45, 61–68, (2014)
15. Malkov, Y., Yashunin, D..:Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. arXiv, https://doi.org/https://doi.org/10.48550/arXiv.1603.09320.
16. McInnes, L.: PyNNDescent, `https://github.com/lmcinnes/pynndescent`.

17. Lokoč, J., Andreadis, S., Bailer, W. et al. Interactive video retrieval in the age of effective joint embedding deep models: lessons from the 11th VBS. Multimedia Systems. https://doi.org/https://doi.org/10.1007/s00530-023-01143-5, (2023).
18. Narasimhulu, Y., Suthar, A., Pasunuri, R., China Venkaiah, V.: Ckd-Tree: An Improved Kd-Tree Construction Algorithm. In: CEUR Workshop Proc., 2786, 211–218, (2021).
19. Pegia, M., Jónsson, B. Þ., Moumtzidou, A, Gialampoukidis, I., Vrochidis, S., Kompatsiaris, I.: MuseHash: Supervised Bayesian Hashing for Multimodal Image Representation, ACM International Conference on Multimedia Retrieval (ICMR), https://doi.org/https://doi.org/10.1145/3591106.3592228, (2023).
20. Reza, M., Ghahremani, B., Naderi, H.: A Survey on Nearest Neighbor Search Methods. In: International Conference on Very Large Data Bases. vol. 95, 0975-8887, (2014).
21. Weber, R. and Schek, H. and Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces, International Conference on Very Large Data Bases. vol. 98, 194-205, (1998)
22. Jayaram Subramanya, S., Devvrit, F., Simhadri, H. V., Krishnawamy, R., Kadekodi, R.: Diskann: Fast accurate billion-point nearest neighbor search on a single node, Advances in Neural Information Processing Systems. vol. 32 (2019)
23. Chen, Qi, Zhao, B., Wang, H., Li, M., Liu, C. and Li, Z., Yang, M.m Wang, J.: Spann: Highly-efficient billion-scale approximate nearest neighbor search. Advances in Neural Information Processing Systems. vol. 34, 5199-5212 (2021)
24. Zhao, W., Tan, S., Li, P: SONG: Approximate nearest neighbor search on GPU. 2020 IEEE 36th International Conference on Data Engineering (ICDE), 1033-1044 (2020)