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, Reykjavík, Iceland {mpegia22, bjorn}@ru.is
[4] Universitat Politècnica de Catalunya, Barcelona, Spain josep.ll.berral@upc.edu

Abstract. Nowadays, large quantities of multimedia data are generated by various applications on smartphones, drones and other devices. Facilitating retrieval from these multimedia collections requires (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 the first systematic evaluation of the time-quality tradeoff that arises when MuseHash media representation is combined with state-of-the-art approximate nearest-neighbour indexes along multi-core and GPU processing.

Keywords: MuseHash · Query processing performance · Approximate nearest neighbour indexes · High-Performance Computing.

1 Introduction

Joint representations of media modalities have recently come into focus within the multimedia community. The most common and successful approaches are based on supervised learning of hash functions, where a model is trained to encode the different modalities into a joint representation, typically a hash code

[§]The first two authors contributed equally to the research reported in this paper.

based in Hamming space. At query time, the given query modalities are then hashed into the same representation, and the media collection is ranked based on similarity to the hash code of the query. So far, in this work, the focus of the evaluation has been on the *effectiveness* of the hash codes, as represented by retrieval accuracy over relatively small benchmark collections. As media collections become increasingly large and complex, however, we must consider the *efficiency* of query processing with these representations. This paper represents the first step in this direction.

Query processing using such hash codes is one instance of the nearest neighbour problem, which is a well-studied problem in the literature. Challenges for nearest-neighbour queries arise when dealing with large-scale high-dimensional collections. This has been termed "curse of dimensionality" since, as dimensionality or dataset size increases, using indexes to return exact nearest neighbour results becomes intractable and a brute-force scan of the collection is necessary [24]. To make retrieval feasible at large scale, a multitude of approximate indexing methods have been proposed, based on a variety of approaches, including tree-based structures, graph-based structures, and hashing-based structures. Such approximate indexes yield a *time-quality tradeoff* between efficiency and effectiveness, more precisely between query processing performance and result quality, that must be evaluated. For some tasks, approximate indexes can even improve quality compared to the brute-force scan. Furthermore, with the increase in available processing power, multi-core and GPU processing can be applied to facilitate hash-code creation and also, depending on the index structure, similarity computations.

In this paper, we present an analysis of query processing performance with the MuseHash approach [22], a recent state-of-the-art multimodal hashing approach for media representation. The main contributions of this paper are:

– We have combined MuseHash media representations with a large set of approximate indexes implemented in the ANN Benchmarks system.
– We have explored the impact of multi-core and GPU processing, using High-Performance Computing (HPC) infrastructures.
– We present experiments for two benchmark collections, a small aerial dataset and a much larger lifelog collection, as well as synthetic collections.

The results indicate that MuseHash can be combined with approximate indexes for efficient query processing, in particular the graph-based HNSW structure, which is considered state-of-the-art in the indexing community, outperforming a full sequential scan in terms of result quality. We also show that while multi-core and GPU processing can improve the performance of the evaluated approach, this impact alone is smaller than the impact of the indexing structure.

The remainder of this paper is organised as follows. Section 2 presents the state-of-the-art in hash-based media representation, including the MuseHash approach. Section 3 then presents the relevant state-of-the-art approximate indexes, while Section 4 presents multi-core processing and HPC. Section 5 then offers our analysis of the experimental results. Finally, Section 6 concludes.

2 Hash-Based Representation and MuseHash

Hash-based representations play a crucial role in efficiently indexing and retrieving multimedia data. They condense complex feature vectors into compact hash codes, streamlining storage and retrieval processes.

MuseHash [22] is a recent multimodal hashing algorithm which excels in handling multimedia data with diverse modalities, leveraging a combination of modalities in its queries, such as visual and temporal aspects, to deliver highly relevant results. In brief, MuseHash extracts features independently for each modality, utilizing Bayesian ridge regression to learn hash functions that map features to the Hamming space. This separate computation for each modality enables support for both unimodal and multimodal queries.

In more detail, the method involves 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.

Overall, MuseHash as a hash-based approach gives compact representations of data and uses less memory for data storage. However, the fast similarity search for the ranking procedure is too slow, when you use brute-force and you have to query large collections. Thus the motivation to use optimization techniques in the ranking procedure drive us to decide approximate nearest neighbors approaches.

3 Approximate Indexes and ANN Benchmarks

The nearest neighbors problem is crucial in computer science, involving finding the closest data points to a given query within a dataset [1]. It is applied in pattern recognition, data mining, image retrieval, and recommendation systems. Exact solutions to this problem are computationally challenging, particularly with large datasets. Approximate nearest neighbors algorithms [3] offer a practical compromise, providing reasonably accurate matches while significantly improving computational efficiency, making them ideal for large-scale applications. This involves balancing retrieval accuracy and faster query processing, ideal for large-scale datasets. Evaluating approximate nearest neighbor algorithms includes assessing accuracy and efficiency using metrics such as precision, recall, query time, and index construction time.

Previous works, such as [14] and [2], use pretrained models to extract features for each modality on diverse datasets. They apply methods from [3], evaluating performance based on exact kNN points rather than provided dataset labels. However, our emphasis is on utilizing dataset labels for measuring method performance. We selected the following approximate nearest neighbors similar to the cited works on the current research [3]: tree-based structures, graph-based structures, pruning techniques, brute-force approaches and baseline methods.

Tree-Based Methods: BallTree [4] uses hyper-spheres to create a tree hierarchy, while CKDTree [19] extends the KD-trees for multiple dimensions with hyper-rectangles. Random Projection Tree (RPT) methods, like Annoy (Approximate Nearest Neighbors Oh Yeah) [16], utilize random projections to split data points and build index structures for fast approximate nearest neighbor search. On the contrary, PyNNDescent [9] 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)[17] arranges the dataset into small-world graphs for efficient approximate nearest neighbor search with minimal memory usage. SW-graph (Small World Graph) [18] combines small-world graph and locality-sensitive hashing. It balances retrieval accuracy and efficiency by leveraging the data's local and global structures.

Pruning Methods: SCANN (Scalable Nearest Neighbors) [11] uses locality-sensitive hashing for fast approximate nearest neighbor search with single-threaded and multi-threaded implementations for large-scale datasets.

Brute-Force Methods: Ball, KD-Tree, BruteForce, and BruteForce-BLAS [7] are simple methods for solving the nearest neighbor search. They calculate the distances between the query point and all data points to find the closest neighbors. Although ensuring accuracy, these methods can be computationally expensive, especially for large datasets. BruteForce-BLAS improves efficiency using the BLAS library for faster distance calculations, serving as a baseline for calculating advanced approximate indexes.

Baseline Methods: Dummy-Algo-MT [10] and Dummy-Algo-ST offer simpler and generic implementations. Dummy-Algo-MT is a multi-threaded brute-force search implementation with parallel processing for improved performance. Dummy-Algo-ST is a single-threaded version, provides a basic implementation for comparison. These baseline methods serve as reference points for evaluating advanced approximate indexes.

4 Multi-Core and GPU processing

Current hardware capabilities can be exploited to improve the speed of a variety of tasks, including feature retrieval. For example, when dealing with data that cannot be accommodated in memory, methods such as DiskANN [13] or SPANN [6] utilise data locality and fast SSD storage. Also, multi-core processing has been exploited in retrieval, with examples such as SCANN [11], or through parallelism at the level of query and data processing. This allows an algorithm to make use of all the available hardware resources, drastically improving performance, even though a linear speed-up may not be achievable. On the other hand, making use of Graphic Processing Units (GPUs) has been popularized, specially in the image processing field and in intensively parallel problems. In the case of the retrieval task, the SONG [25] algorithm, for example, has been co-designed to make use of GPUs, beating similar algorithms that are CPU-based. This highlights that when developing new indexing structures, it is advantageous to keep in mind the capabilities of the underlying hardware.

In this section we define how the MuseHash algorithm is optimized to use the available hardware as efficiently as possible and how it is transformed to increase its time performance when more resources are available (scalability). Concretely, we are focusing on the offline and querying phases of the algorithm. Our approach is driven by the usage of parallel computation using multi-core and GPU processing.

In the offline phase, the focus is on feature extraction. As this implies extracting features from the latent space of a Neural Network, the most viable optimization is to use GPUs alongside specialized accelerated software when performing the required forward pass (CUDA [20] and cuDNN [8]). Moreover, the task can be done simultaneously by multiple GPUs as there are no dependencies between samples. Notice that the proposed optimization can also be applied in the querying phase if the incoming sample has no precomputed features.

In the querying phase, however, we propose to improve query processing performance through two different strategies: query parallelism and data parallelism. Both strategies are designed for high-capability environments, enabling the scalability of algorithms with increased resources. Nevertheless, they are also applicable to smaller devices with multi-core possibilities.

On one hand, the **query parallelism** strategy divides the incoming queries into a set of processes, henceforth query processors, where each of them has a different copy of the overall data. The query processors can be located in different machines and can work independently due to the absence of dependencies between queries. On the other hand, the **data parallelism** strategy splits the workload of a single query into multiple processes that divide the full set of data in equal parts to perform the subsequent similarity comparison. These processes need to synchronize to produce a common result.

The proposed strategies are not contradictory and can be integrated together for an increased performance, making use of multiple machines where different query processors are located (query parallelism) and utilizing all the available resources in each of them (data parallelism). Similarly to query parallelism, the data parallelism strategy can be applied to processes located in different machines, but its implementation is much harder and the subsequent communication latency could be detrimental. In addition, we propose using GPU processing to speed up the similarity comparison, as this accelerated hardware can increase the performance of the internal computations of the algorithms.

5 Experiments

In this section, we report results from a set of experiments that explore query processing performance of state-of-the-art methods with the MuseHash media representations. In the offline phase, the focus is on feature extraction and the implementation options are relatively straight-forward: using 4 GPUs simultaneously resulted in 20-fold speed-up compared to the CPU-only version. In the querying phase, however, we have a choice between approximate indexing and hardware-based implementations, so due to space limitations the focus here is on a detailed analysis of the querying phase. We start by investigating the impact of applying approximate high-dimensional indexes from the ANN Benchmarks collection. In the remaining four experiments, we study the impact of hardware on the brute-force scan and one particular approximate indexing strategy.

5.1 Datasets

We utilize two benchmark collections and three synthetically generated collections. The benchmark collections provide us with diverse modalities and rich content for realistic analysis of results quality. While the relevant properties and modalities are summarized in Table 1, they are:

AU-AIR The AU-AIR dataset [5] consists of eight aerial traffic surveillance video clips at an intersection in Aarhus, Denmark. Captured on windless days, the videos depict varied lighting conditions due to the time of day and weather. With a resolution of 1920x1080 pixels, it comprises 32,823 frames extracted at five frames per second to avoid redundancies.

LSC'23 This dataset was generated by an active lifelogger over the course of 18 months [12]. The primary resource of the collection is an image dataset featuring fully redacted and anonymized wearable camera images. Captured using a Narrative Clip device, these images have a resolution of 1024x768 pixels. Due to huge imbalance on the dataset, we filtered the original dataset with using only the data that include label information with label frequency greater than 257. This preprocessed dataset used in our experiments comprises 40926 images.

To evaluate scaling of the various 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 (**small synthetic**), 112000 (**medium synthetic**) and 448000 (**large synthetic**) samples with different hash lengths (32, 128, 512 and 2048) and 450 samples for testing (querying).

5.2 Experimental Settings

We have run a large-set of experiments, but due to space limitations we report the results of a representative sample in this section. For example, since the

Table 1: Summary of 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

ANN Benchmark collection contains a large set of algorithm implementations, we have chosen to omit algorithms that (a) are alternative implementations of the same approach or (b) perform poorly, leaving 5 indexing strategies and one brute-force approach.

In our benchmark experiments, we have assessed the retrieval performance of MuseHash using several evaluation metrics, including mean Average Precision (mAP), precision, recall, and F-score, but due to space limitations we focus on F-score in our presentation. We have assessed query performance using latency (milliseconds per query) and throughput (reported as thousand queries per second). In the following we report on throughput, sometimes represented as speed-up over a baseline implementation.

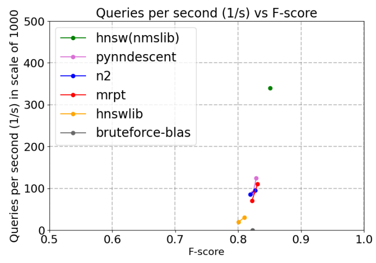
For the benchmark collections, the following feature vectors from each modality are used as input representations for our evaluation:

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 [22]. The first four coordinates represent the year, the next 12 are for month (one-hot-encoded), the next 31 for day (one-hot-encoded), the next 24 for hours (one-hot-encoded), the next 60 for minutes (one-hot-encoded), and the next 60 for seconds (one-hot-encoded). AU-AIR has an additional 12 digits for microseconds (binary encoded).
Spatial 3-D vector with values (altitude, longitude, altitude).

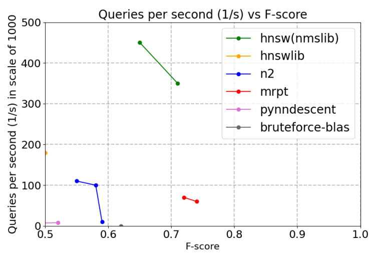
We compute hash codes using MuseHash for different bit lengths $d_c = 16, 32, 64, 128, 256, 512, 1024, 2048$. Moreover, we conduct experiments using both single modalities and a combination of all modalities.

The first experiment with ANN Benchmarks was performed using a server with Intel(r) Core(TM) i9-10920X CPU @ 3.50GHz (12 cores, 2 threads/core) with 134.25 GB of RAM. The remaining experiments were performed on the MareNostrum IV accelerated clusters from the Barcelona Supercomputing Center, where each machine is an IBM Power9 8335-GTH @ 2.4GHz (20 cores, 4 threads/core) with 512 GB of RAM and 4 NVIDIA V100 GPU with 16GB RAM. In each case, 5 executions are averaged, with error margins calculated with the standard deviation.

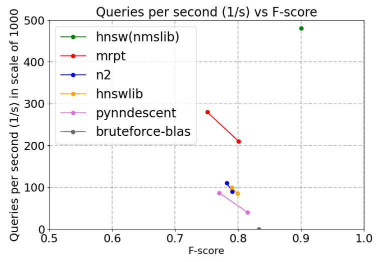
[5]https://github.com/Leo-xxx/pytorch-notebooks/blob/master/Torn-shirt-classifier/VGG16-transfer-learning.ipy
[6]https://github.com/maknotavailable/pytorch-pretrained-BERT



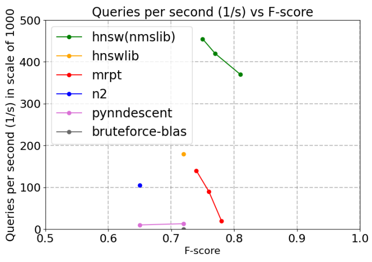
(1) AU-AIR: Visual mod./16b codes



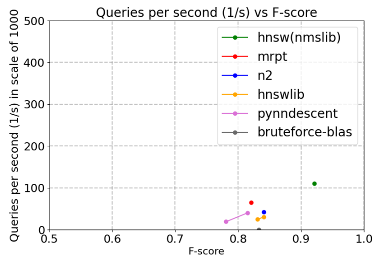
(2) LSC'23: Visual mod./16b codes



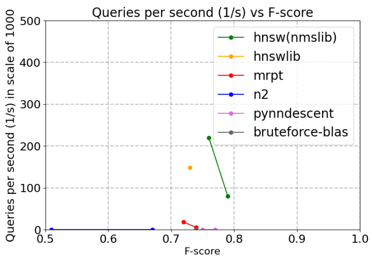
(3) AU-AIR: All mod/16b codes



(4) LSC'23: All mod./16b codes



(5) AU-AIR: All mod./2048b codes



(6) LSC'23: All mod/2048b codes

Fig. 1: CPU experiment results: AU-AIR (1st column), LSC'23 (2nd column). Rows show visual modality with 16-bit codes, all modalities with 16-bit codes, and all modalities with 2048-bit codes.

5.3 Experiment 1: Impact of Approximate Indexes

In this first experiment, we evaluate retrieval results on the AU-AIR and LSC'23 datasets. Figure 1 shows a representative sample of the results; similar observations hold with other settings. The first column of Figure 1 displays results for the AU-AIR collection, covering from top to bottom (1) visual modality with 16-bit codes, (3) all modalities with 16-bit codes, and (5) all modalities with 2048-bit codes. The second column presents results for the LSC'23 dataset, following the same structure of modalities and hash code lengths. In each case, the x-axis represents the F-score value (to make the graphs more readable, we focus on the range from 0.5 to 1.0), while the y-axis represents throughput (thousand

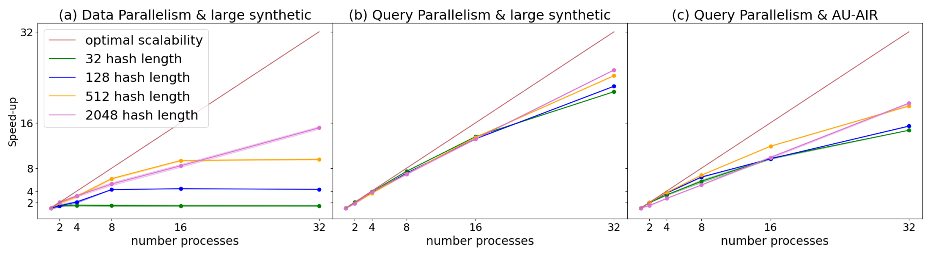


Fig. 2: Speed-up of Data Parallelism vs Query parallelism for different hash lengths and datasets.

queries per second). As outlined above, while the ANN Benchmark contains a large number of indexing algorithms, we report only the six best-performing implementations here. As seen in the figure, each such approach offers some different parameters, which lead to different tradeoffs between quality and time.

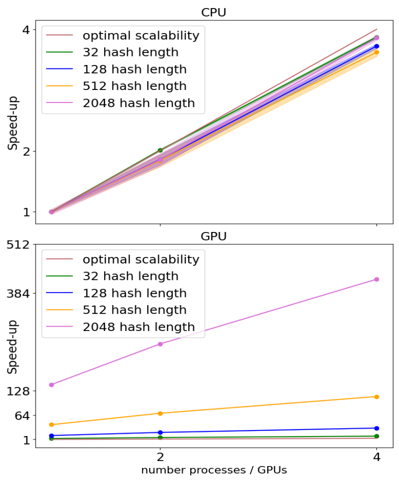
Overall, the figures show that the Hnswlib algorithm outperforms all other approaches across both collections and all settings, both in terms of throughput and result quality. This confirms results from recent studies with different settings. The brute-force algorithm, unsurprisingly, performs worst, with throughput below 1,000 queries per second in all cases.

The figure also shows that for AU-AIR, including more modalities improves F-score of Hnswlib, while it decreases quality for other approaches. Increasing the hash code length, up to 2048, enhances overall retrieval quality, but at the cost of significantly reduced throughput. Regarding LSC'23, the incorporation of textual information boosts precision and recall, aligning with the success of textual queries using CLIP models in the VBS competition [15]. Using all available modalities further enhances retrieval results, again at the cost of reduced throughput.

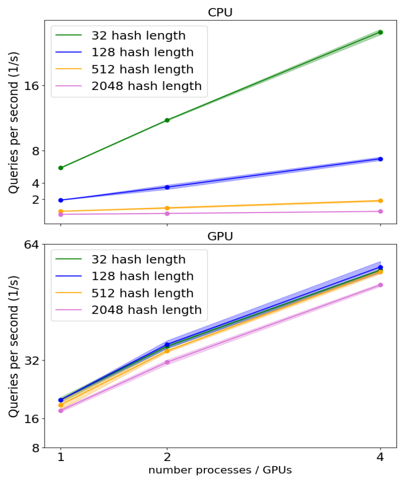
5.4 Experiment 2: Query Parallelism vs. Data parallelism

In this experiment, we focus on comparing data parallelism (different processes contribute to the computation of each query) and query parallelism (different processes answer different queries) strategies for scaling the brute-force algorithm (using an implementation taken from the scikit-learn package [21]) which traverses the full length of the dataset to find the exact match ($O(n)$ complexity). Figure 2 shows a representative sample of results, showing (1) data parallelism and (2) query paralleism for the large synthetic dataset, and (3) query parallelism for the AU-AIR dataset, for a variety of hash lengths In all graphs, the speed-up represents throughput with varying number of processors compared to the throughput of a single process; the optimal performance would be that with X processes, the speed-up should be a factor of X.

As Figure 2(a) shows, data parallelism can yield considerable increase in performance, but only when considering longer hash codes; with smaller hash-codes the speed-up reaches a plateau and the same presumably holds even for hash



(1) Speed-up CPU vs GPU



(2) QPS (1/s) CPU vs GPU

Fig. 3: CPU vs GPU parallelism comparison in the large synthetic dataset. Results shown with speed-up (1) and queries per second (2) metrics. Optimal scalability and GPU speed-up are computed taking as baseline the execution with just one simultaneous process from the CPU only version.

code lengths of 2048. Figure 2(b) indicates that query parallelism scales better than data parallelism, obtaining a speed-up factor of 25x, which is very close to the optimal of 32x, for the largest synthetic collection. Finally, Figure 2(c) shows that query parallelism results are somewhat worse for the AU-AIR dataset.

5.5 Experiment 3: GPU vs CPU comparison with brute-force

In these experiments we show the advantages of specialized hardware, concretely GPUs[7]. Figure 3 shows that this hardware provides better performance than the standard CPU multi-core processing. With the largest synthetic dataset and hash length, we obtain a speed-up of x144 with only one GPU in regard to the CPU sequential version. Moreover, if employing more GPUs simultaneously in a query parallelism scenario, where every query processor has a different GPU, we can accomplish an outstanding x419 speed-up. Lastly, as can be seen in Figure 3.2, it is important to point out that GPUs provide almost the same number of queries per second for each hash length in contrast to the CPU version, where the hash length heavily impacts the performance.

5.6 Experiment 4: Query parallelism with PyNNDescent

Figure 4 shows the speed-up results of scaling PyNNDescent using multi-core processing in a query parallelism scenario with the synthetic datasets. It can

[7]implementation taken from the cuML package [23]

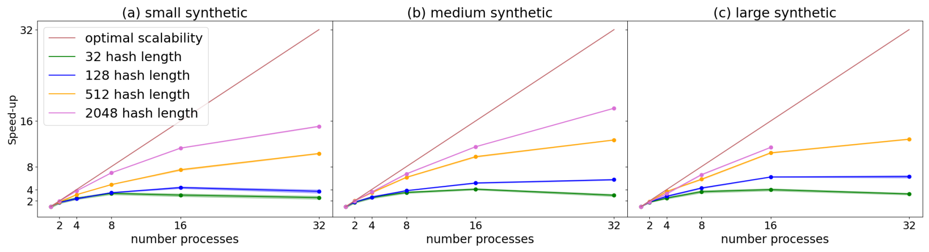


Fig. 4: Query parallelism with the PyNNDescent algorithm (with default parameters). Unfortunately, the executions of 32 processes with a hash length equal to 2048 with the synthetic large dataset did not finish in a reasonable time (2 hours).

be seen that the algorithm scales similarly in all of them, reasonably increasing the performance up to a speed-up between x3 and x8 when using 8 simultaneous processes. From this point on, employing more parallel processes does not substantially improve the final time performance, especially in the case of small hash lengths. In addition, these results show that the PyNNDescen algorithm is quite constant with regard to the number of dataset samples, obtaining similar scalability results with different sized datasets.

5.7 Experiment 4: Curse of dimensionality

Many ANN algorithms suffer from the curse of dimensionality problem, especially classic tree methods that create fast representations for speeding up the similarity comparisons [24]. In these experiments we compare the performance of a classical tree method (BallTree), a state-of-the-art ANN method (PyNNDescent) and the brute-force approaches when increasing the dimensionality through the size of the hash lengths. Figure 5 shows the results for the largest synthetic dataset. We can see that the BallTree algorithm already degrades to the brute-force performance when increasing the hash length to a size of only 8. In contrast, the PyNNDescent algorithm proves itself worth for tackling much larger dimension datasets, only matching the brute-force performance after increasing the length to 2048.

5.8 Discussion

In both benchmark datasets, we observed from Experiment 1 that the Hnswlib, a fast approximate nearest neighbor search method, outperformed all other methods, both in terms of througput and result quality. Overall, many indexing approaches performed well, yielding througput that was much better than that achieved by any hardware-based approaches. For the hardware-based approaches, the query parallelism strategy excellent speed-up capacity, especially with GPU processing, and also combined well with an approximate indexing approach, even though this combination was still outperformed by the best approximate indexing approach.

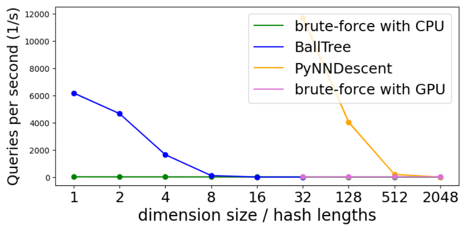


Fig. 5: Curse of dimensionality of brute-force, BallTree (with $leaf\_size = 100$) and PyNNDescent (with default parameters) algorithms in the large synthetic dataset with different hash lengths. Notice that the x-axis is not proportional.

6 Conclusion

In this paper, we have considered query performance for the multi-modal hashing codes produced by MuseHash, using both state-of-the-art approximate indexing approaches and hardware-based approaches. Our results indicate that these techniques improve performance to varying extent, but that they can also often be combined for further performance improvements. Future work includes applying a combination of the top-performing Hnswlib indexing algorithm, along with parallel hardware-based strategies.

Acknowledgment

The work reported here was supported by the EU's Horizon 2020 research and innovation programme under grant agreements H2020-101070250 CALLISTO, H2020-101070262 WATERVERSE, and H2020-101080090 ALLIES, and by the Spanish Ministry of Science (MICINN), the Research State Agency (AEI) and European Regional Development Funds (ERDF/FEDER) under grant agreements 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. Abbasifard, M.R., Ghahremani, B., Naderi, H.: A survey on nearest neighbor search methods. International Journal of Computer Applications **95**(25) (2014)
2. Arulmozhi, P., Abirami, S.: A comparative study of hash based approximate nearest neighbor learning and its application in image retrieval. Artificial Intelligence Review (2019)
3. Aumüller, M., Bernhardsson, E., Faithfull, A.: Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In: International conference on similarity search and applications. pp. 34–49. Springer (2017)
4. Boytsov, L., Naidan, B.: Engineering efficient and effective non-metric space library. In: Similarity Search and Applications: 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings 6. pp. 280–293. Springer (2013)
5. Bozcan, I., Kayacan, E.: Au-air: A multi-modal unmanned aerial vehicle dataset for low altitude traffic surveillance. In: 2020 IEEE International Conference on Robotics and Automation (ICRA). pp. 8504–8510. IEEE (2020)
6. Chen, Q., Zhao, B., Wang, H., Li, M., Liu, C., Li, Z., Yang, M., Wang, J.: Spann: Highly-efficient billion-scale approximate nearest neighborhood search. Advances in Neural Information Processing Systems **34**, 5199–5212 (2021)
7. Chen, X., Güttel, S.: Fast exact fixed-radius nearest neighbor search based on sorting. Preprint at arXiv. https://doi. org/10.48550/ARXIV **2212** (2022)
8. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning (2014)
9. Dong, W., Moses, C., Li, K.: Efficient k-nearest neighbor graph construction for generic similarity measures. In: Proceedings of the 20th international conference on World wide web. pp. 577–586 (2011)
10. Geiger, M.J.: A multi-threaded local search algorithm and computer implementation for the multi-mode, resource-constrained multi-project scheduling problem. European Journal of Operational Research **256**(3), 729–741 (2017)
11. 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. pp. 3887–3896. PMLR (2020)
12. Gurrin, C., Jónsson, B.Þ., Nguyen, D.T.D., Healy, G., Lokoc, J., Zhou, L., Rossetto, L., Tran, M.T., Hürst, W., Bailer, W., et al.: Introduction to the sixth annual lifelog search challenge, lsc'23. In: Proceedings of the 2023 ACM International Conference on Multimedia Retrieval. pp. 678–679 (2023)
13. 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 **32** (2019)
14. 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 (2019)
15. Lokoč, J., Andreadis, S., Bailer, W., Duane, A., Gurrin, C., Ma, Z., Messina, N., Nguyen, T.N., Peška, L., Rossetto, L., et al.: Interactive video retrieval in the age of effective joint embedding deep models: lessons from the 11th vbs. Multimedia Systems **29**(6), 3481–3504 (2023)
16. Luna, A.: Using annoy in package c++ code
17. Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE transactions on pattern analysis and machine intelligence **42**(4), 824–836 (2018)

18. Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V.: Approximate nearest neighbor algorithm based on navigable small world graphs. Information Systems **45**, 61–68 (2014)
19. Narasimhulu, Y., Suthar, A., Pasunuri, R., Venkaiah, V.C.: Ckd-tree: An improved kd-tree construction algorithm. In: ISIC. pp. 211–218 (2021)
20. NVIDIA, Vingelmann, P., Fitzek, F.H.: Cuda, release: 10.2.89 (2020), `https://developer.nvidia.com/cuda-toolkit`
21. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
22. Pegia, M., Jónsson, B.Þ., Moumtzidou, A., Gialampoukidis, I., Vrochidis, S., Kompatsiaris, I.: Musehash: Supervised bayesian hashing for multimodal image representation. In: Proceedings of the ACM International Conference on Multimedia Retrieval (ICMR). pp. 434–442 (2023)
23. Raschka, S., Patterson, J., Nolet, C.: Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. arXiv preprint arXiv:2002.04803 (2020)
24. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: VLDB. vol. 98, pp. 194–205 (1998)
25. Zhao, W., Tan, S., Li, P.: Song: Approximate nearest neighbor search on gpu. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE). pp. 1033–1044. IEEE (2020)