Comparison of scalable distributed algorithms for assessing the *k*NNG in multi-GPU

Gabriel Orlando[1], **Hermes Senger** [1], **Murilo Naldi**[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFScar)
Caixa Postal 676 – 13565-905 – São Carlos – SP – Brazil

gmcorlando@estudante.ufscar.br, {hermes,naldi}@ufscar.br

Abstract. *Many applications, require finding a dataset's k-Nearest Neighbors Graph (kNNG), crucial for many Machine Learning tasks like clustering and anomaly detection. However, its computation can be costly due to the complexity of finding all kNN for every data point. To address this issue, scalable approximated algorithms have been proposed to speed up the kNNG and maintain its quality. This paper presents an adaption of NNDescent using multi-GPU and an experimental comparison of distributed and parallel approximate kNNG algorithms in GPUs, assessing their scalability, computational cost, and solution quality. Our goal is to identify the most efficient method without significant accuracy loss, enabling faster techniques and handling large datasets.*

1. Introduction

Given an integer number *k*, the process of finding the *k* nearest neighbors - *k*NN of all the dataset's objects can be accomplished by building the *k* nearest neighbor graph - *k*NNG, i.e., a directed graph structure that connects the *k*NN of each object under a specific distance metric, where each edge contains the distance between connected objects. The *k*NNG is crucial for applications in various areas like Machine Learning, including clustering (e.g., Watershed Clustering [Xia et al. 2022], classification (enabling textual classification [Wang and Liu 2010]), and outlier detection (e.g., Local Outlier Factor - LOF [Kang et al. 2012]). The fast construction of the *k*NNG is paramount for many tasks.

Building the *k*NNG is challenging due to its asymptotic computational complexity of $O(n^2)$ for n points in the dataset, which arises from the pairwise comparison of points [Dong et al. 2011]. It may be computationally unfeasible for large datasets due to computational effort and memory limitations of most computing systems [Xing and Bei 2020].

Heuristics were proposed for faster approximate *k*NNG with very similar results to the exact ones [Turan et al. 2020]. For example, NNDescent exploits neighbor relations for significant speedup [Dong et al. 2011, Shimomura et al. 2021]. Another example is using an Inverted File to reduce similarity calculations and maintain fast and high-quality *k*NNG construction [Amato and Savino 2010, Zhang et al. 2013].

Nowadays, many tasks are running on Graphical Process Unit - GPU, like in classification and clustering areas, where many algorithms benefit from the acceleration provided by the GPU [Kuttranont et al. 2017, Andrade et al. 2013]. Additionally, this accelerator has an architecture that supports massive data parallel processing. This means that a single GPU can process an exceptionally large number of data instances in parallel, resulting in faster applications compared to those running on CPUs [Asano et al. 2009].

As datasets increase and GPUs face physical limitations, finding the *k*NNG becomes more challenging [Pereira et al. 2012]. Despite GPUs' inherent parallelism, memory limitations hinder exact *k*NNG construction for large datasets, leading to memory overflow issues. Besides the physical constraint, the data transfer between primary and GPU memory can become a burden due to bus limitations [Croix and Khatri 2009].

The utilization of multiple GPUs can considerably accelerate computational processes, offering scalability—a term that denotes the ability to maintain or enhance performance as the problem size increases. Scalability becomes especially important when dealing with large datasets. A scalable method should be capable of efficiently constructing the approximate *k*NNG, even for large datasets [Jogalekar and Woodside 2000].

Due to the variety of algorithms, there is an increasing need for analysis regarding the approximation quality, construction time, and scalability. Initially, we present an adaption of NNDescent compatible with multi-GPU systems. Then, we focus on our primary goal, evaluating the use of approximate heuristics for estimating the *k*NNG in multi-GPU systems, especially for offering a higher amount of overall memory compared to a single-GPU system, which allows us to deal with larger datasets [Fogal et al. 2010]. By comparing their performances, we will assess which ones are faster and more accurate, their benefits, and drawbacks. Additionally, we review GPU architectures to understand the principal issues the compared algorithms have when dealing with different types of parallel/distributed systems. Finally, we assess the scalability of these algorithms to understand the computational boundaries of each method and check the resulted *k*NNG, considering memory usage, building time, quality, and dataset size. We consider two scenarios: when the dataset fits one GPU memory (Small scenario) and when we must divide the dataset across multiple GPUs (Large scenario). We hope our work sheds light on new ways to enhance and provide scalability for Machine Learning tasks based on the *k*NNG.

2. Related Work

Selecting suitable algorithms for this research was challenging. It was crucial to consider specific properties during the evaluation. First and foremost, the compatibility of the algorithms with a multi-GPU system, as the research aims to investigate the scalability of parallel and distributed algorithms. Another critical property was the requirement for the algorithm to be approximate, given that the exhaustive search becomes impractical when dealing with large datasets due to its aforementioned computational complexity.

The authors in [Johnson et al. 2019] introduce data quantization for constructing the kNNG, including *Inverted File* algorithms. They partition the dataset into clusters and the search process is limited to relevant cells and avoid exhaustive search, making them promising for research. This idea results in the Inverted File Flat - IVFFlat algorithm, multi-GPU compatible and provide an approximate *k*NNG, which is compared here.

[Johnson et al. 2019] presents a comparison of *Inverted File* methods, focusing on querying approximate *k*NN for datasets across multiple GPUs. Their objective is not to construct the approximate *k*NNG itself but rather to facilitate efficient querying. By evaluating the performance and characteristics of these methods, the research aims to provide valuable insights and comparisons in the context of multi-GPU systems.

In [Dong et al. 2011], a method called NNDescent is proposed for constructing neighborhood graphs using the heuristic that the nearest neighbors of a given data point

are likely to be neighbors amongst themselves. By leveraging this heuristic, the algorithm generates a high-quality approximate kNNG avoiding the need for an exact search. However, this method has a memory overhead, challenging large-scale dataset processing.

[Wang et al. 2021b] introduces a single GPU kNNG merging algorithm using NNDescent for incremental construction. While the approach faces memory limitations due to the merging process being confined to a single GPU, they present a solution to mitigate these challenges by incorporating disk input/output - I/O operations. Our paper adapted their algorithm for multi-GPU systems, further enhancing efficiency and scalability, and allowing the construction of the approximate kNNG for large-scale datasets.

[Shimomura et al. 2021] present a survey on exact/approximate kNNG construction on CPU for small-scale datasets. The survey delves into an exploration of algorithms' performance over various features of datasets, e.g., dimensionality and cardinality. Differently from [Shimomura et al. 2021], we present a comparative analysis focused on multi-GPU systems towards datasets of large-scale size. We also conduct empirical experiments and analyze scalability, considering datasets that fit single and multiple GPUs. We aim to provide a more practical understanding of the algorithms' performances.

3. Theoretical background

3.1. Quantization techniques

Quantization is a fundamental method that maps continuous infinite values to a smaller set of discrete finite values. By employing techniques such as *Product Quantization* [Jégou et al. 2011] or *Standard Quantization* [Zhou et al. 2012], memory usage can be reduced while enabling the processing of larger datasets. Additionally, combining quantization techniques with the *Inverted File* heuristic presents an excellent approach to decrease memory consumption further and facilitate the efficient handling of extensive datasets.

One specific combination of techniques is the utilization of *Product Quantization* - PQ alongside *Inverted File to minimize memory usage*, resulting in Inverted File with Product Quantization - IVFPQ [Jégou et al. 2011]. PQ compresses high-dimensional vectors for efficient nearest-neighbor searches, decomposing vectors into low-dimensional subspaces quantized independently, reducing the vector size for handling large datasets.

Similarly, the combination of *Standard Quantization* - SQ with *Inverted File* gives rise to Inverted File with Standard Quantization - IVFSQ [Zhou et al. 2012]. IVFSQ aims to decrease memory usage without compromising the quality of the approximation. Unlike PQ, SQ is not commonly employed alongside the *Inverted File* algorithm. However, it offers a unique approach by transforming vectors in each partition into bits vectors.

3.2. *Inverted File* parameters influence

Inverted File depends on two parameters: **"nList"** (number of partitions created) and **"nprobe"** (number of partitions searched). A partition denotes a unique space of multiple cells where dataset vectors are mapped to a cell based on their characteristics. The parameter "nList" impacts search efficiency and accuracy, as few partitions lead to slow searches, while more cells reduce search time but lower approximation quality and increase mapping time. On the other hand, overestimating **"nprobe"** can result in slower performance, as more cells need to be examined. Conversely, small values are not recommended as they may lead to reduced approximation quality, as only a few partitions

are searched. Thus, the implicit trade-off between speed and quality using Inverted File algorithms is explored in the present paper for a variety of scenarios.

3.3. Multi-GPU parallelism

Multi-GPU systems have become increasingly frequent in the architecture of HPC systems and servers, accelerating tasks in scientific simulations [Wang et al. 2020, Spampinato et al. 2010]. Despite GPU's remarkable processing capabilities, limited memory poses challenges, necessitating careful management. Therefore, tasks on multi-GPU systems can be handled via **Replication** and **Sharding** [Johnson et al. 2019].

Replication copies the dataset over the GPUs, which implies that the dataset must fit into one GPU memory; otherwise, the task can not be realized. The process can query points simultaneously in each GPU by splitting the queries over the GPUs, which can make the whole process faster. However, a considerable amount of memory is required.

Sharding is a technique where we divide the dataset into sub-datasets called shards, that has the same columns but contains different vectors. The vectors held in each partition are unique and independent from the vectors from other shards. It is slower than Replication but manages more memory simultaneously to deal with larger datasets.

4. Multi-GPU NNDescent

We present an adapted version of [Wang et al. 2021b] for multi-GPU and large datasets. The algorithm divides data into shards, merging the resulting kNNG of each fragment using I/O disk operations to prevent GPU memory overflow. The algorithm's structure is easily parallelized over all the GPUs, guaranteeing that the processes are synchronized and avoiding two or more GPUs merging the same shards simultaneously to construct a satisfactory approximate *k*NNG. We divide the algorithm into three steps: *Shard division, kNNG construction with NNDescent for each shard, and sub-graph merging.*

4.1. Proposed approach

At first, we divide the dataset into *S* subsets called shards. We must ensure that each GPU in the system can merge 2 *k*NNG simultaneously, such that each *k*NNG corresponds to a specific shard S_i.[1] After that, we construct the *k*NNG for each shard S_i, as illustrated in Algorithm 1. Consequently, as we use a multi-GPU system and threads to perform this task, we build the *k*NNG for num shards simultaneously, using NNDescent, where num is the number of GPUs in the system. The number of shards should be iterable by the number of GPUs, i.e., the shards should be equally divided over the GPUs.

After constructing the *k*NNG for all the shards, we need to merge these partial graphs to obtain the final *k*NNG. The process is described in Algorithm 2 and loading the $kNNG_0$ into GPU_0, merging it with the other $kNNG$ graphs, and storing the result. Completing the process, all *S* $kNNG$ graphs would have been merged, potentially containing new neighbors from previous merges. To construct the final $kNNG$ for each shard, we need to merge all $kNNG$ graphs. We achieve this by running the merging process for all *S* $kNNG$ graphs. At each iteration, we merge the current $kNNG_i$ with $S - i$ $kNNG$ graphs since $kNNG_i$ has already been merged. To ensure synchronization, a mutex is used to allow merging only when the $kNNG_i$ is ready. Our approach enables parallelization across multiple GPUs, resulting in a better performance than a single GPU.

[1] In our experiments, our shards had at most 3.7 GB to fulfill this requirement.

Algorithm 1 Simultaneously construct the kNNG for each shard S_i

Require: $Path$: dataset path on disk; k: number of neighbors; S: number of shards; num: number of GPUs;
Ensure: Each GPU can construct two kNNG at once
  $Iters \leftarrow \frac{S}{num}$
  for i in [0,1,...,$Iters - 1$] **do**
    for g in [0,1,...,$num - 1$] **do in parallel** ▷ Each device computes shards in parallel
      $shardID \leftarrow i \times num + g.$
      $kNNG_{shardID} \leftarrow NNDescent(shardID, k)$ ▷ Runs in parallel within a GPU
      $save\_to\_disk(kNNG_{shardID})$;
    end for
  end for

Algorithm 2 Simultaneously merge each shard's kNNG

Require: $Path$: dataset path on disk; S: number of shards; num: number of GPUs;
Ensure: Each GPU can merge two kNNG at once
  $Iters \leftarrow \frac{S}{num}$
  for i in [0,1,...,$Iters - 1$] **do**
    for g in [0,1,...,$num - 1$] **do in parallel** ▷ Each device computes merge in parallel
      $graphID \leftarrow i \times num + g.$
      $merged\_kNNG_{graphID} \leftarrow MergeKNNGs(graphID)$ ▷ Merge
  $kNNG_{graphID}$ with $S - graphID$ kNNGs, managed by a mutex for synchronization.
      $save\_to\_disk(merged\_kNNG_{graphID})$
    end for
  end for

5. Experiments

Our study is accomplished in a multi-GPU server with 3 **NVIDIA GeForce RTX 2080 Super** with 8192 MiB of memory each, 2 Intel(R) Xeon(R) Silver 4208 with 16 CPU cores, and 128GB of primary memory. The algorithms were compiled with CUDA 12.0 over Ubuntu 22.04. The code for the experiments is presented in a Github repository.[2].

5.1. Algorithms

In our study, we consider two distinct scenarios for comparing the algorithms. The Small scenario involves datasets that can fit entirely into the memory of a single GPU. We leverage the GPU techniques discussed in Section 3.3 for this scenario. By examining the algorithms under these conditions, we can evaluate their performance and effectiveness when the entire dataset can fit the memory constraints of a single GPU. The Large scenario involves datasets that are too large to be stored in the memory of a single GPU and must be divided into shards to distribute the workload across multiple GPUs. As described in Section 3.3, only the Sharding technique is applicable in this scenario. By comparing the algorithms under these conditions, we can assess their scalability and efficiency when dealing with datasets that exceed the memory capacity of a single GPU. Table 1 presents the compared algorithms, the strategies adopted, and which scenarios will be considered.

[2]All the scripts used in this paper are available at `https://github.com/gorlando04/Scalable-distributed-algorithms-for-approximating-the-kNNG`

Table 1. Strategies and algorithms used on the comparison proposed

	GPU parallelism technique		Scenario	
Algorithms	**Replication**	**Sharding**	**Small**	**Large**
IVFFlat [Amato and Savino 2010]	YES	YES	YES	NO
IVFPQ [Jégou et al. 2011]	YES	YES	YES	YES
IVFSQ [Zhou et al. 2012]	YES	NO	YES	NO
NNDescent [Wang et al. 2021b]	YES	NO	YES	NO
NNDescent I/O	NO	YES	YES	YES
Exact-Construction	YES	YES	YES	YES

5.2. Datasets

Artificial datasets were generated using probabilistic distributions to build a controlled environment with scalable properties (e.g. dimensionality), which is crucial when working with GPU processing and *k*NN algorithms. We used the Python-based platform `Scikit-Learn` [Pedregosa et al. 2011] that encompassed various data distributions and sizes to ensure a diverse range of test cases. All the datasets used in our experiments were real-valued and created using two principal distributions. The Gaussian distribution generated isotropic blobs, while the *make_biclusters* method produced objects exhibiting biclustering characteristics [Dhillon 2001]. The shape of each artificial dataset generated for the experiments in the Small scenario can be found in Table 2.

Table 2. Articial datasets used for the Small scenario

Name	Number of samples	Dimensions	Size(GB)
SK-1M-12d	1 million	12	0.048
SK-10M-12d	10 million	12	0.48
SK-20M-12d	20 million	12	0.96
SK-30M-12d	30 million	12	1.44
SK-40M-12d	40 million	12	1.92
SK-50M-12d	50 million	12	2.4

Moreover, for the Large scenario, the dataset exceeds the memory capacity of a single GPU and necessitates partitioning across multiple GPUs, as described in Table 3.

Table 3. Articial datasets used for the Large scenario

Name	Number of samples	Dimensions	Size(GB)
SK-100M-12d	100 million	12	4.8
SK-150M-12d	150 million	12	7.2
SK-200M-12d	200 million	12	9.6
SK-250M-12d	250 million	12	12
SK-300M-12d	300 million	12	14.4
SK-350M-12d	350 million	12	16.8
SK-400M-12d	400 million	12	19.2

5.3. Evaluation method

One satisfactory way to measure an approximate algorithm performance is to compare the approximate result with the exact one. For example, `Recall@k` can be used to measure the quality of the constructed *k*NNG and is a great way to verify the quality of the constructed *k*NNG [Patel et al. 2022, Wang et al. 2021a]. We calculate `Recall@k` using Equation 1, where $R(i, k)$ represents the number of truth-positive first *k* neighbors for the object *i*. Here, the value of *k* for Recall was defined as 10, as `Recall@10` has been constantly used in other research work [Johnson et al. 2019, Wang et al. 2021a].

$$Recall@k = \frac{\sum_{i=1}^{n} R(i,k)}{n * k} \quad (1)$$

`Recall@10` of the whole dataset will be used to measure the approximate *k*NNG quality for the datasets described in Table 2. However, the datasets in Table 3 will have the quality assessed by `Recall@10` for the first 10.000 nearest neighbors because our system cannot handle building of the full *k*NNG for datasets with over 200 million points.

5.4. Parameters settings

We defined (*nList*) for *Inverted File* methods as $nList = 2^{\log_{10} n^2}$, where n is the dataset's number of samples, and probe varied from 5 to 100. *Inverted File* algorithms limit shards to the number of GPUs, while NNDescent I/O needs more due to disk operations and memory overhead. However, as discussed, NNDescent I/O does disk operations and has a significant memory overhead as explained in Section 2, so we would not be able to construct the approximate *k*NNG using three shards, and then we need to split the dataset in more shards, which can explain the difference in the number of shards for *Inverted File* algorithms and NNDescent I/O. Therefore, the number of shards for NNDescent I/O follows the proposal in Section 4 to optimize GPU usage.

Finally, we need to define the main parameter of the *k*NNG algorithms, the value of the hyperparameter k. This step is crucial because we need to choose a value for k that is sufficient to test the GPU memory usage, but on the other hand, the value must not be enormous because we could overflow the GPU memory for most datasets. Therefore we define k as being 32 for our experiments because is a value "rule of thumb" adopted in similar work [Johnson et al. 2019, Wang et al. 2021b] and was a great value to test the GPU usage without overflowing the GPU memory for the datasets.

5.5. Small scenario results

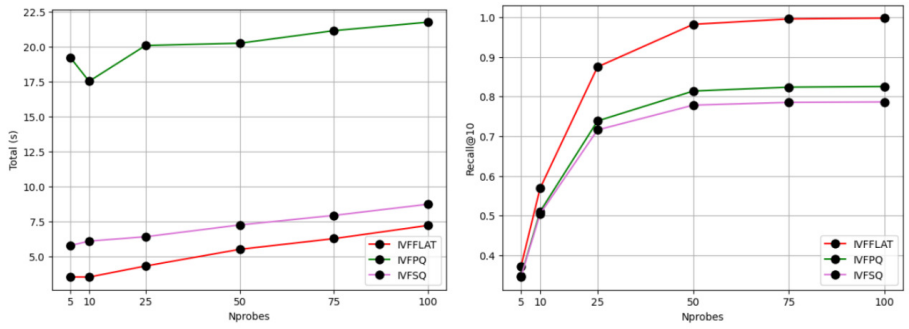
In this section, datasets fit entirely in one GPU memory and can be replicated over the GPUs' memories. Although replication is an appropriate solution for this scenario, we compare both techniques, as sharding is also a possibility. In Fig. 1, the time to construct the approximate *k*NNG and the quality of the approximate *k*NNG using Replication is shown. Additionally, all the dataset's results are presented in Table 4.[3]

Table 4. Results of the approximate *k*NNG construction using Replication for datasets described in Table 2. For *Inverted File* methods, nprobe = 50 was chosen. Exact-Construction method will also be shown. Speedup is relative to the Exact method

	IVFFLat			IVFPQ			IVFSQ			NNDescent			Exact
Datasets	**Time**	**Rec@10**	**Speedup**	**Time**	**Rec@10**	**Speedup**	**Time**	**Rec@10**	**Speedup**	**Time**	**Rec@10**	**Speedup**	**Time**
SK-1M-12d	5.51s	0.98	1.86x	20.26s	0.81	-	7.62s	0.778	1.35x	4.37s	0.973	2.35x	10.29s
SK-10M-12d	-	-	-	72.33s	0.786	13x	60.22s	0.686	15.64x	45.90s	0.862	20.52x	941.96s
SK-20M-12d	-	-	-	115.47s	0.782	32.55x	-	-	-	-	-	-	3767.88s
SK-30M-12d	-	-	-	290.16s	0.779	29.15x	-	-	-	-	-	-	8460.95s
SK-40M-12d	-	-	-	649.45s	0.746	23.16x	-	-	-	-	-	-	15045.11s
SK-50M-12d	-	-	-	564.14s	0.744	41.63x	-	-	-	-	-	-	23486.75s

With the results, we could check that Replication has a restriction in terms of memory, shown in Table 4, where it is possible to observe how scalable each method is.

[3] In some cases, the method could not complete the *k*NNG construction for some settings due to memory overflow, and were indicated with a "-".

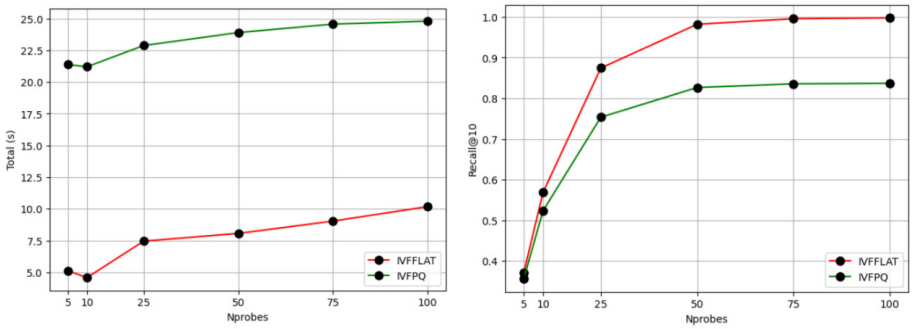


(a) SK-1M-12d - Time (b) SK-1M-12d - Rec@10

Figure 1. Running time and `Recall@10` of approximate *k*NNG construction using Replication for 1M(Fig. 1a, Fig. 1b) dataset with *Inverted File* methods. In this experiment, nprobe parameter was varied from 5 up to 100, to produce the curves.

IVFPQ was the only one able to construct the approximate *k*NNG for all datasets, having more than 0.74 on `Recall@10` for most datasets. For vast GPU memory, IVFFLat and NNDescent with Replication technique could construct a higher quality approximate *k*NNG, as these methods build excellent *k*NNG approximations. Otherwise, IVFPQ is recommended, as this algorithm has a low GPU memory usage and can handle large-scale datasets.

In Fig. 2, the time to construct the approximate *k*NNG and the quality of the approximate *k*NNG using Sharding is shown. Also, it is essential to emphasize that in some cases, the method could not complete the *k*NNG construction for some settings due to insufficient GPU memory. Additionally, all the dataset's results are presented in Table 5. [4]



(a) SK-1M-12d - Time (b) SK-1M-12d - Rec@10

Figure 2. Running time and `Recall@10` of approximate *k*NNG construction using Sharding for 1M(Fig. 2a, Fig. 2b) dataset with *Inverted File* methods. In this experiment, nprobe parameter was varied from 5 up to 100,to produce the curves.

In Table 5, we could observe that, even though Sharding allows the larger datasets to be handled IVFFlat is an approximate method with a high GPU memory usage and was unsuccessful for some datasets. On the other hand, NNDescent I/O and IVFPQ

[4]As has been discussed in this paper, NNDescent I/O has different values of Shards for the datasets because we must ensure that our GPU could merge to *k*NNG at once, which explains larger datasets having greater number of shards.

Table 5. Results of the approximate *k*NNG construction using Sharding for datasets described in Table 2. For *Inverted File* methods, nprobe = 50 was chosen. Exact-Construction method will also be shown. Speedup is relative to the Exact method.

	IVFFLat				IVFPQ				NNDescent I/O				Exact	
Datasets	**Time**	**Rec@10**	**Shards**	**Speedup**	**Time**	**Rec@10**	**Shards**	**Speedup**	**Time**	**Rec@10**	**Shards**	**Speedup**	**Time**	**Shards**
SK-1M-12d	8.06s	0.981	3	1.27x	23.89s	0.826	3	-	12.19s	0.978	3	-	10.29s	3
SK-10M-12d	109.58s	0.970	3	8.59x	93.33s	0.802	3	10.09x	77.44s	0.866	3	12.16	941.96s	3
SK-20M-12d	305.45s	0.902	3	12.33x	155.64s	0.737	3	24.2x	156.80s	0.775	3	24.02x	3767.88s	3
SK-30M-12d	614.43s	0.768	3	13.77x	224.76s	0.614	3	37.64x	395.36s	0.897	6	21.40x	8460.95s	3
SK-40M-12d	-	-	-	-	531.92s	0.775	3	28.28x	522.26s	0.877	6	28.80x	15045.11s	3
SK-50M-12d	-	-	-	-	597.80s	0.788	3	39.28x	851.01s	0.927	9	27.59s	23486.75s	3

could construct the approximate *k*NNG for all the datasets presented in Table 2. Although NNDescent uses disk I/O operations, it performs well for large datasets, constructing the approximate *k*NNG at almost the same time as IVFPQ but with a finer recall. Therefore, even though IVFPQ is quicker than NNDescent I/O, regarding approximate *k*NNG quality, NNDescent I/O shows better results than IVFPQ, achieving more than 0.8 for most scenarios. However, NNDescent I/O's less satisfactory *k*NNG is due to the small number of shards, which results in fewer merging operations, decreasing the quality of the approximate *k*NNG.

5.6. Large scenario results

In this section, we analyze the Large scenario, where datasets don't fit in one GPU memory. Only Sharding compatible methods (IVFPQ and NNDescent I/O) are tested. We compare time and `Recall@10` for 10,000 objects due to computational constraints as described in Section 5.3. The parameters for both algorithms will follow the proposal presented in 5.5. Finally, IVFFlat will not be used in this scenario because this algorithm has an extensive memory overhead. We present the results in Table 6.[5]

Table 6. Results of the approximate *k*NNG construction using Sharding for datasets described in Table 3. For IVFPQ, nprobe = 100 was chosen for obtaining the max `Recall@10`. Exact-Construction method will also be shown. Speedup is relative to the Exact method

	IVFPQ				NNDescent I/O				Exact	
Datasets	**Time**	**Rec@10**	**Shards**	**Speedup**	**Time**	**Rec@10**	**Shards**	**Speedup**	**Time**	**Shards**
SK-100M-12d	2266.92s	0.749	3	41.51x	2350.44s	0.973	15	40.03x	94106.28s	3
SK-150M-12d	3499.38s	0.748	3	60.37x	4804.50s	0.981	24	43.97x	211258.76s	3
SK-200M-12d	5041.69s	0.734	3	74.48	7639.18s	0.982	30	49.15x	375534.77	3
SK-250M-12d	6851.70s	0.737	3	-	10297.53s	0.983	36	-	-	3
SK-300M-12d	8638.30s	0.734	3	-	15915.68s	0.985	45	-	-	3
SK-350M-12d	11441.11s	0.763	3	-	21094.86	0.986	51	-	-	3
SK-400M-12d	12845.55s	0.754	3	-	27721.59s	0.987	60	-	-	3

The results in Table 6 reveal the differences between IVFPQ and NNDescent I/O for large-scale datasets. IVFPQ is fast and efficient in constructing the *k*NNG, making it suitable for quick approximations. However, its quality is lower, with `Recall@10` mostly below 0.76, which may be a concern if seeking higher-quality approximations. On the other hand, even though NNDescent I/O is slower than IVFPQ, the quality of the approximate *k*NNG is impressive, having more than 0.92 for `Recall@10`. Additionally, when comparing the construction time to the exact algorithm, NNDescent I/O provides significant acceleration, addressing the main challenge in *k*NNG building.

[5]Exact algorithm had a 5-day time limit (120h) for the experiments, noted as "-" in Table 6.

Therefore, IVFPQ and NNDescent I/O have proved to be a scalable approximate *k*NNG method that works fine with multi-GPU systems as it could construct the approximate *k*NNG for all the datasets with both techniques. Additionally, we could check for a trade-off between speed (IVFPQ) and approximation quality (NNDescent I/O).

6. Conclusion

The paper fulfilled its goals, introducing and studying multiple *k*NNG algorithms. We test scalability in two scenarios: replication, which faced memory limitation issues, and sharding, which could handle larger datasets.

In the Small scenario, IVFFlat, NNDescent, and IVFSQ had significant GPU memory usage, which made them unable to construct the *k*NNG for datasets over 20M samples. However, NNDescent and IVFFlat had high-quality approximations, having more than 0.95 of `Recall@10`. IVFPQ was rapid but lower quality than IVFFlat and NNDescent. NNDescent I/O performed well, constructing *k*NNG for all datasets in adequate time when compared to the exact construction, and had a high recall, having more than 0.86 for most datasets. In the Large scenario, IVFPQ constructed *k*NNG quickly for large-scale datasets with multi-GPU and Sharding benefits, 74x faster than exact construction. However, IVFPQ had lower quality, while NNDescent I/O provided good speed, compared to the exact-construction, 49x faster than exact-construction, and high-quality approximations with `Recall@10` over 0.92. Therefore, it indicates a trade-off between speed and quality when using IVFPQ and NNDescent I/O.

Finally, it is essential to note that the scope of our investigation is confined to single-node systems. While we speed up the construction of the kNNG through the utilization of multiple GPUs, our analysis does not extend to the evaluation of algorithms in the context of multi-node systems. The exploration of multi-node system scenarios represents a promising avenue for future research endeavors.

In the future, we plan to use the results of this research to enhance several Machine Learning techniques that rely on the construction of the kNNG. Our findings will guide our choices of algorithm and parameters based on the scenario of application and dataset characteristics.

Acknowledgment

G.O. and M.N. thanks FAPESP (contract number 2019/09817-6, 2022/04934-7 and 2023/00993-1) and CAPES for their support. H.S. thanks FAPESP (contract number 2019/26702-8, 2021/00199-8, and 2023/00566-6) for their support.

References

[Amato and Savino 2010] Amato, G. and Savino, P. (2010). Approximate similarity search in metric spaces using inverted files. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems*. ICST.

[Andrade et al. 2013] Andrade, G., Ramos, G., Madeira, D., Sachetto, R., Ferreira, R., and Rocha, L. (2013). G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia Computer Science*, 18:369–378. 2013 International Conference on Computational Science.

[Asano et al. 2009] Asano, S., Maruyama, T., and Yamaguchi, Y. (2009). Performance comparison of fpga, gpu and cpu in image processing. In *2009 international conference on field programmable logic and applications*, pages 126–131. IEEE.

[Croix and Khatri 2009] Croix, J. F. and Khatri, S. P. (2009). Introduction to gpu programming for eda. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 276–280.

[Dhillon 2001] Dhillon, I. S. (2001). Co-clustering documents and words using bipartite spectral graph partitioning. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, page 269–274, New York, NY, USA. Association for Computing Machinery.

[Dong et al. 2011] Dong, W., Charikar, M., and Li, K. (2011). Efficient k-nearest neighbor graph construction for generic similarity measures. In Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M. P., Bertino, E., and Kumar, R., editors, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 577–586. ACM.

[Fogal et al. 2010] Fogal, T., Childs, H., Shankar, S., Krüger, J. H., Bergeron, R. D., and Hatcher, P. J. (2010). Large data visualization on distributed memory multi-gpu clusters. *High Performance Graphics*, 3:57–66.

[Jogalekar and Woodside 2000] Jogalekar, P. and Woodside, M. (2000). Evaluating the scalability of distributed systems. *IEEE Transactions on parallel and distributed systems*, 11(6):589–603.

[Johnson et al. 2019] Johnson, J., Douze, M., and Jégou, H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

[Jégou et al. 2011] Jégou, H., Douze, M., and Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128.

[Kang et al. 2012] Kang, B., Kim, D., and Kang, S. (2012). Real-time business process monitoring method for prediction of abnormal termination using knni-based LOF prediction. *Expert Syst. Appl.*, 39(5):6061–6068.

[Kuttranont et al. 2017] Kuttranont, P., Boonprakob, K., Phaudphut, C., Permpol, S., Aimtongkhamand, P., KoKaew, U., Waikham, B., and So-In, C. (2017). Parallel knn and neighborhood classification implementations on gpu for network intrusion detection. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 9(2-2):29–33.

[Patel et al. 2022] Patel, Y., Tolias, G., and Matas, J. (2022). Recall@ k surrogate loss with large batches and similarity mixup. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7502–7511.

[Pedregosa et al. 2011] 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., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Pereira et al. 2012] Pereira, F., Theis, C., Moreira, A. J. C., and Ricardo, M. (2012). Performance and limits of knn-based positioning methods for GSM networks over leaky feeder in underground tunnels. *J. Locat. Based Serv.*, 6(2):117–133.

[Shimomura et al. 2021] Shimomura, L. C., Oyamada, R. S., Vieira, M. R., and Kaster, D. S. (2021). A survey on graph-based methods for similarity searches in metric spaces. *Information Systems*, 95:101507.

[Spampinato et al. 2010] Spampinato, D. G., Elster, A. C., and Natvig, T. (2010). Modelling multi-gpu systems. In *Parallel Computing: From Multicores and GPU's to Petascale*, pages 562–569. Ios Press.

[Turan et al. 2020] Turan, H. H., Sleptchenko, A., Pokharel, S., and ElMekkawy, T. Y. (2020). A sorting based efficient heuristic for pooled repair shop designs. *Comput. Oper. Res.*, 117:104887.

[Wang et al. 2021a] Wang, H., Zhao, W., Zeng, X., and Yang, J. (2021a). Fast k-nn graph construction by GPU based nn-descent. In Demartini, G., Zuccon, G., Culpepper, J. S., Huang, Z., and Tong, H., editors, *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, pages 1929–1938. ACM.

[Wang et al. 2021b] Wang, H., Zhao, W.-L., Zeng, X., and Yang, J. (2021b). Fast k-nn graph construction by gpu based nn-descent. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 1929–1938.

[Wang et al. 2020] Wang, X., Qiu, Y., Slattery, S. R., Fang, Y., Li, M., Zhu, S.-C., Zhu, Y., Tang, M., Manocha, D., and Jiang, C. (2020). A massively parallel and scalable multi-gpu material point method. *ACM Trans. Graph.*, 39(4).

[Wang and Liu 2010] Wang, Z. and Liu, Z. (2010). Graph-based KNN text classification. In Li, M., Liang, Q., Wang, L., and Song, Y., editors, *Seventh International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2010, 10-12 August 2010, Yantai, Shandong, China*, pages 2363–2366. IEEE.

[Xia et al. 2022] Xia, J., Zhang, J., Wang, Y., Han, L., and Yan, H. (2022). WC-KNNG-PC: watershed clustering based on k-nearest-neighbor graph and pauta criterion. *Pattern Recognit.*, 121:108177.

[Xing and Bei 2020] Xing, W. and Bei, Y. (2020). Medical health big data classification based on KNN classification algorithm. *IEEE Access*, 8:28808–28819.

[Zhang et al. 2013] Zhang, Y., Huang, K., Geng, G., and Liu, C. (2013). Fast knn graph construction with locality sensitive hashing. In *Procedings of Machine Learning and Knowledge Discovery in Databases, ECML PKDD 2013, Prague, Czech Republic, 2013*, volume 8189 of *Lecture Notes in Computer Science*, pages 660–674. Springer.

[Zhou et al. 2012] Zhou, W., Lu, Y., Li, H., and Tian, Q. (2012). Scalar quantization for large scale image search. In *Proceedings of the 20th ACM international conference on Multimedia*, pages 169–178.