Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs

Background and Motivation

Previous lectures in the class focussed on methods for retrieval augmented generation e.g. RETRO where tokens from the training set were augmented with their k nearest chunks from a frozen LM database (this assumes some distance metric defined between data elements and finds database elements which minimize this metric to a new query). However, exact k-NN search is prohibitive for high-dimensional data space, therefore approximate nearest neighbour algorithms are usually employed. Paper 1 (Malkov and Yashunin) focuses on a method for hierarchical navigable small world (HNSW) which has logarithmic complexity in scaling compared to power law or linear relationships. They use a heuristic for neighbour selection and a greedy search procedure that traverses the graph from the top layer, where the layers further down have smaller distance between the query point and its neighbours.

Related Works

Navigable World Models: The simplest setting for a navigable graph (with logarithmic or polylogarithmic scaling) is a superposition of a structured subgraph and a smaller random subgraph with a few long-range connections following a pre-specified distribution e.g. (Watts and Strogatz, Kleinberg). Another setting studies graphs in a geometric sense (Boguna et al.) by formalizing heterogeneity and clustering in terms of hyperbolic geometry. However, scalable graph traversal in these settings require prior knowledge of data-distribution which may be infeasible.

Moreover, scale-free (power-law degree distribution) networks constructed, for eg, using the method described in Boguna et al. (https://arxiv.org/pdf/0709.0303.pdf) show a polylogarithmic increase in average path length between nodes as the network size increases.

Methods for Approximate NN Search

Some of these methods iteratively traverse the graph, and choose the next base node which minimizes the distance to the query. These may have poor results on highly clustered data. k-nearest neighbors graph construction, as outlined in [Dong et al.] performs well on datasets with different measures and shows good recall compared to LSH ; however it has power law scaling (empirical complexity of n^1.14) which this work aims to better.

Key details about Methodology

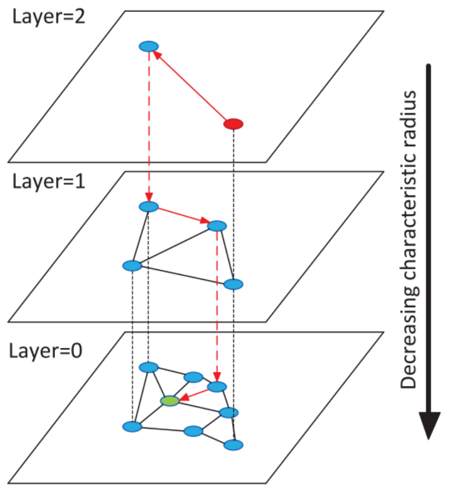


Fig. 1. Illustration of the Hierarchical NSW idea. The search starts from an element from the top layer (shown red). Red arrows show direction of the greedy algorithm from the entry point to the query (shown green).

3 main steps:

a. Graph construction (one shot)

Dynamic neighbour update:

a. Gradually increase the search space / node degree considered (distance of query point to neighbours decreases)
b. Propose heuristic to select neighbours so as to retain some global information about connectivity

Details / Pseudocode:

1. -ln (U(0, 1)) x m_L) : Set bottommost layer l for every query q
2. For topmost layer, find single nearest neighbour. This is the entry point for the subsequent layer.
3. for lc -> (min(L, l) .....0): W: dynamic list of found neighbors, C: candidate neighbors, ef: Number of elements to return
    a. If $c \in C$ (nearest to q) is farther from $f \in W$ (farthest to q), W is not updated based on neighbours of c.
    b. Else update C, update W and remove furthest element f. Repeat till |C|>0
    c. Neighbour selection heuristic: initialize R, add e from W to R if $d(e, q) < d(r \in R, q)$. Return M neighbours
4. Shink connections if needed based on maximum number of connections per layer M_max.

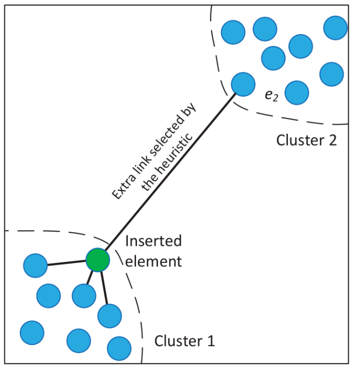


Figure 2 from paper illustrating how the heuristic can find neighbours from a different cluster

Observations:

Algorithm shows good performance on datasets with different metrics and different effective dimensionality compared to the baseline NSW.

Key Hyperparameters: m_L, M_max (number of layers, search depth)

Setting appropriate non-zero values for these leads to search complexity of O(log(N)) rather than power law or polylogarithmic complexity. A good choice for m_L is found to be 1 / ln(M). The average memory consumption is capped by the choice for Mmax0 and Mmax for each subsequent layer.

Limitations
Distributed search is not possible out-of-the box on Hierarchical NSW, since the search always starts from the top layer, however simple workarounds are possible to make its performance comparable to NSW.

Class Discussion
Question related to the construction of dynamic graphs - this work currently does not focus on updating the graph, but some follow-up works do e.g. https://arxiv.org/abs/2307.10479 (not about hierarchical NSW) discusses how old neighborhoods may be updated and how good performance can be ensured in exploratory search.

One method to dynamically update graph: Build up a static taxonomy / classification (hierarchical) for all subjects. For every new query q, search first in taxonomy level, and then search within all instances belonging to that taxonomy. Some questions arise regarding the resolution of such a hierarchy. A better solution would be online clustering approaches - where some clusters may be merged together and cluster labels may be updated because the taxonomy / class labels may not be trivial to construct for high-dimensional data.

MemGPT: Towards LLMs as Operating Systems

Background and Motivation

The second paper proposes an alternative to long-context models (which traditionally incur a quadratic increase in compute time and memory cost) in the form of virtual memory paging that enables LLMs to conduct extended dialogues and analyze lengthy documents. Alternate approaches have been proposed to circumvent the issues surrounding limited context window size. However, such approaches all fundamentally have some finite restrictions, despite improvements in that upper limit. This work, therefore, proposed a method that could theoretically extend indefinitely, bounded only by the storage capacity of inference systems.

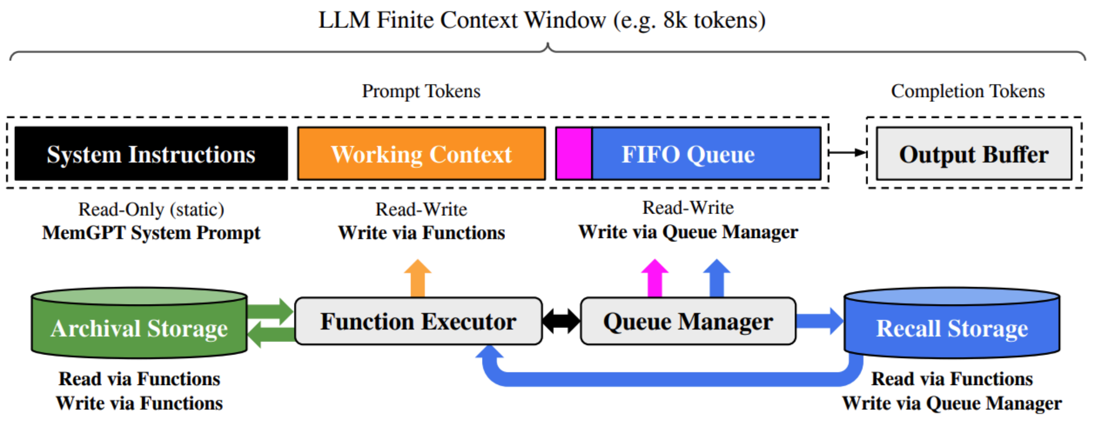
Related Works

As mentioned, the main relevant alternatives are those in the space of improving context window length. Towards that end, several were highlighted in the work itself, specifically focusing on “sparsifying the attention, low-rank approximations, and neural memory.” These approaches fundamentally differ from that proposed by MemGPT, since the former assume a *fixed* memory capacity, as set by the system RAM, only seeking to then maximize the amount of information that can then be fit into that limited space.

The work also highlighted the related line of work exploring “retrieval-augmented models.” Unlike those works highlighted above, however, these works are related in that only their *approach* parallels that of MemGPT; the problem they seek to solve, namely that of improving recall quality, is distinct from that of MemGPT, which seeks to solve the limited context window problem. Interestingly, however, the solving of the latter does in fact improve the former, yet it is not explicitly stated as the end goal, unlike in these related lines of inquiry.

Key details about Methodology

The structure of the approach parallels that of classical OS design, where applications are given the illusion of unlimited memory with the help of paging to storage, shown in the overview below.

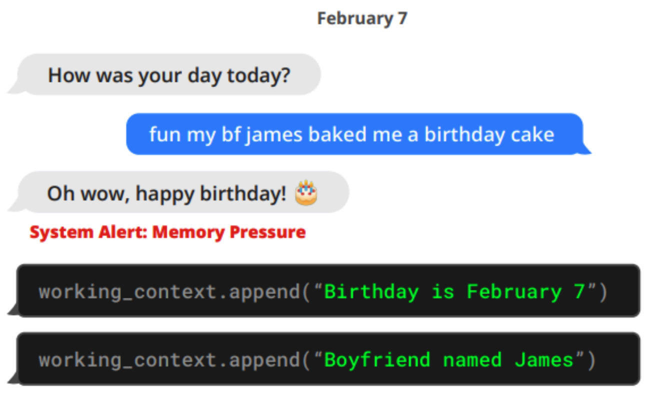


The flow of computation now differs from the traditional structure of LLM inference, where, rather than having prompts immediately be fed into an inference system, they are instead stored in a separate **FIFO Queue**. The advantage of doing so is that a **queue manager** can then allow for the **paging in** of relevant pieces of context from “memory” to append to the context based on the queried prompt via a **function executor**. That is, rather than simply relying on the relevant context being present in the most recent *N* tokens, where *N* is the finite capacity of the LLM context, a queue manager can ensure this is the case through appropriate context fetching. In this way, contexts are effectively “indefinitely long,” as the fetching can be done arbitrarily far back into the past, so long as it is stored somewhere in the database.

More in-depth, the FIFO queue holds the rolling history of messages between the agent and user and also the system messages, discussed subsequently. The Queue manager is then responsible for fetching from this queue the prompt to be handled by the LLM inference system.

LLM inference then proceeds as normal, with the primary difference being that output tokens may now include bespoke function calls, such as working_context.append(...). Upon being produced, this output buffer is then handled by the function executor. Crucially, this function executor is a hand-crafted program, meaning the calls queried by the LLM must be made aware of how to make valid API invocations to this program. New code is added automatically through the separate function executor program, which takes as input a YAML file including the desired function, corresponding implementation, and description specified in some prespecified format. In doing so, the function executor then produces a corresponding prompt for the LLM to perform in-context learning to learn to invoke this new function.

Such parsed function calls, therefore, are commonly related to the writing or reading of messages from storage, which itself bifurcates into two pieces: the **archival storage** used for holding long-term data that will be placed in main context and **recall storage** used for storing messages that the LLM processed. In this way, the MemGPT system augments an existing LLM with the ability to determine whether additional context is required to properly respond to an input query, delaying returning a response to the ultimate end user until such an "inner monologue" is completed. An example of this is shown below


February 7

How was your day today?

fun my bf james baked me a birthday cake

Oh wow, happy birthday! 🎂

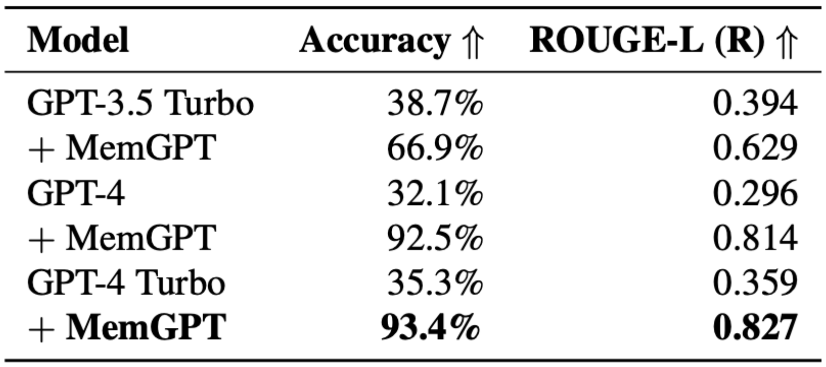
System Alert: Memory Pressure

```
working_context.append("Birthday is February 7")
```

```
working_context.append("Boyfriend named James")
```

Notably, in addition to function calls being appended in the manner above, system calls can also be invoked if detected, such as if memory capacity is nearly or completely filled.

Using this improved context window, there were several downstream implications highlighted in the experiments. One particular use was improved document retrieval, as seen below. Critically, these reported numbers are provided with the same baseline LLM when reported in the "+ MemGPT" rows.



Model	Accuracy $\Uparrow$	ROUGE-L (R) $\Uparrow$
GPT-3.5 Turbo	38.7%	0.394
+ MemGPT	66.9%	0.629
GPT-4	32.1%	0.296
+ MemGPT	92.5%	0.814
GPT-4 Turbo	35.3%	0.359
+ **MemGPT**	**93.4%**	**0.827**

Class Discussion

What type of database was used for document retrieval in their experiments? While a traditional database could be used in this scenario, they used a vector database in their experiments.

Why would one choose MemGPT over ChatGPT? To improve over the limited context window.

How would this paper be relevant for models that handle an already large number of tokens? Ultimately, a finite capacity for context windows limits what we can handle, such as in terms of documents that can be handled.

Can we have a caching mechanism in addition to the paging out idea? The recall storage already serves the role of the low-latency cache, where summaries of recent contexts are stored in this recall storage.

Is there any increase in inference latency? The inference latency is not significant enough to be noticeable for the user experience.

Future Research

There are a lot of works using multi-step LLMs, where the control flow is distinct from the workflow considered here. In particular, for static outsourcing, there is a downstream *known* function that the LLM outsources to, such as information lookup, whereas in other cases, the *thinking* may itself need to expand, where the LLM itself calls another LLM. It is not naively handled in the current paper and would require significant changes to determine how the function executor should now do the dispatching.

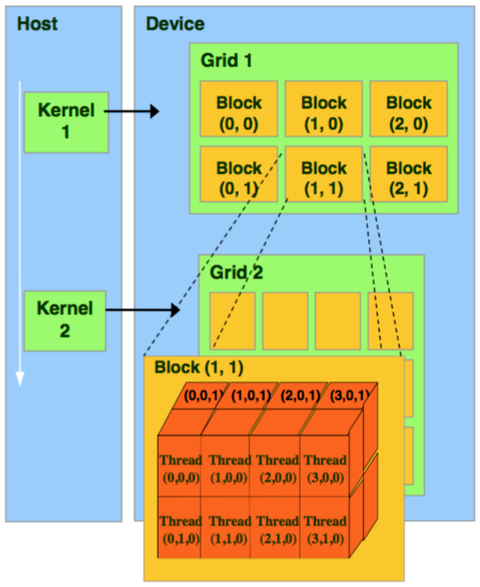
Billion-Scale Similarity Search with GPUs

Background and Motivation

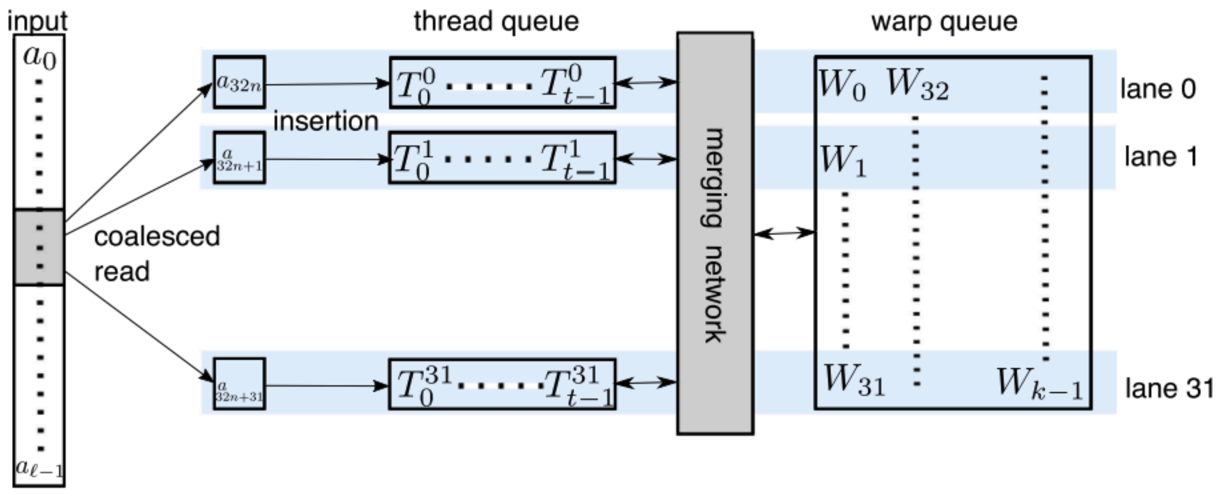
Vector databases are increasingly expanding the types of data they support with more complex, structured data becoming of interest for machine learning systems, such as images and videos. Such databases, however, fail to leverage the efficiency gains possible with the effective employment of GPUs. Such efficiency boils down to how to more effectively perform similarity search on GPUs when high-dimensional features are present.

Key details about Methodology

The contribution is a new k-selection algorithm that operates at up to 55% theoretical peak performance and has multi-GPU support. As a brief overview, GPU compute is organized hierarchically, as shown below, with clustering first into grids, then as blocks, and finally as threads. Notably, these distinctions are encoded in the *hardware*. In contrast, when a kernel from the CPU is dispatched for compute on the GPU, the program specifies execution in terms of a *thread warp*, which then gets mapped to the hardware thread blocks in some manner dictated by the GPU drivers. Part of GPU optimization deals with minimizing warp divergence overheads, where threads within a given warp have dramatically different runtimes.



In this vein, when a new search request comes in, the goal is to leverage thread-level parallelism to optimize the lookup. Each thread holds a local queue in its registers, against which the queries can be checked for insertion very efficiently. In this way, there is no need to traverse down the memory hierarchy or deal with cross-warp synchronization. The logic of insertion can easily be handled in parallel as well, with each thread independently checking whether insertion should occur or not. That is, since all but one will reject the insertion request, no synchronization is required to ensure correctness. A merging request is occasionally invoked if discrepancies arise between the local thread and warp queues, as highlighted in the figure below.


input
a_0
$a_{\ell-1}$
coalesced
read
a_{32n}
a_{32n+1}
a_{32n+31}
insertion
thread queue
$T_0^0 \cdots T_{t-1}^0$
$T_0^1 \cdots T_{t-1}^1$
$T_0^{31} \cdots T_{t-1}^{31}$
merging network
warp queue
W_0 W_{32}
W_1
W_{31}
W_{k-1}
lane 0
lane 1
lane 31