Experiment Log on CacheGen

1. Paper review

KV cache is widely used to reduce prefill recompuatation. The problem occurs when KV cache need to be loaded from another machine especially when the network is less than ideal causing innegligible network delay. CacheGen proposed by this paper aimed to reduce the network transfer overhead.

CacheGen design consists of 2 parts. First part been encoding part where the author provide a custom KV cache codec to minimize the size of KV bit stream. Second part is streaming where CacheGen would adapt to the bandwidth dynamically.

Starting with the encoding part of KV cache, the author analysis 3 key insights to KV cache that sparks the idea to CacheGen encoding:

1. In the same layer+channel, tokens locate closer together has higher KV tensor value proximity. Most likely due to self-attention mechanism.
2. Loss in shallow layer KV cache value has impact output quality more than loss in deeper layer KV cache value. Intuitively, loss in shallow layer propagate and thus the impact.
3. Tokens grouped by layer and channel has more similarity with each other than token grouped by position in the context.

Given the 3 insights above, the CacheGen encoding the author came up with has the 3 key steps:

1. Calculate delta tensor. Form groups of consecutive tokens, calcualate delta tensor between the first tensor in the group and the rest and store only the first KV tensor and the delta tensor of the rest. This is compact and can compress and decompress in parallel.
2. Apply different level of quantization to delta tensor at different level with vector-wise quantization. 8-bit high precision quantization for anchor tokens.
3. Use a arithmetic coder to encode quantized delta tensor into bitstream while compressing values in each layer and channel separately. Use arithmetic coding, the more homogeneous the source is, the shorter the compressed output would be.

For the streaming part of CacheGen, contexts are divided into different consecutive chunks. Each chunk with be stream at different quantization level, to high quantization level to plain text, based on the bandwidth of the previous sent chunk. The bandwidth for a chunk quantization to adapt to would base on the actual bandwidth faced by the previous chunk so the delay of variation in the bandwidth would only affect at most 1 chunk delivery thus making it an effective adaptive strategy.

Together, CacheGen keep the low computation delay and minimize network delay under volatile network conditions.

2. Environment setup

(1) System

1. WSL

Haven't use Conda on this computer, not going to try Conda on Windows, too many foreseeable problems. First try Windows subsystem for Linux(WSL). Encounter many problem mainly due to:

- unsupported feature in WSL
- mainland network issues, had to get support from Microsoft store

2. Local Virtual Machine

Thus abandon WSL, start using VMware Workstation Pro. The step for getting VMware Workstation has got a lot more complicate since last time I use it. Thankfully, there is a good tutorial [5].

The build and experiment require GPU but do not have them locally, thus abandon this plan.

3.Online GPU Environment.

Next, try AutoDL. Rent 2 A40 graphic card. With efforts put into it, this one finally works! But it is not capable with docker environment thus cannot run the example architecture. My suspect the AutoDL environment is also built based on docker thus cannot handle docker within docker.

3. Potential Improvement from Network Point of View

(1) Separate Transmission of Delta Tensor

As mentioned in the streaming part of CacheGen in the paper, CacheGen will send chunks at different encoding levels or text format due to change in bandwidth. According to the paper, this would help the transmission.

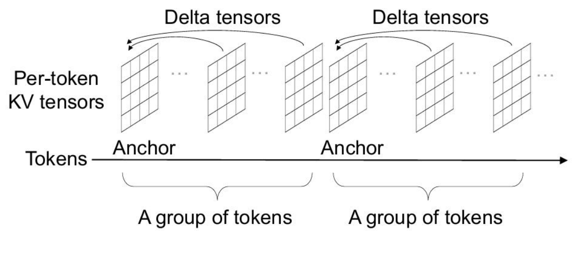


Figure 6: *Within a token group, CacheGen computes delta tensors between KV tensors of the anchor token and those of remaining tokens.*

As we can see from the figure from the paper, ach group consisted of an anchor token and the others tokens. The anchor token is quantized with higher precision quantization. We compute the delta tensor between the rest of the tensors and the anchor tensor and store them. When sending them, the current CacheGen send a group of anchor and delta tensors all together and when facing a lower bandwidth, CacheGen send text format which create more computational overhead and undermined the effort to reduce Time-To-First-Token(TTFT).

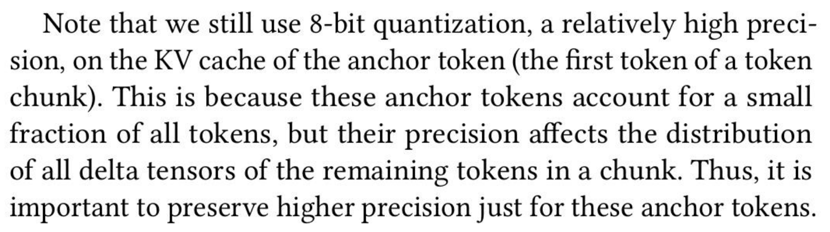
It came to my attention that sending plain text would request recomputation of KV cache and thus create giant computation overhead and eliminate the advantage of using CacheGen. Thus I propose potential solutions to this problem: Transmit delta tensor and anchor tensor separately.

I have two solutions I have in mind both transmit delta tensor and anchor tensor separately.

The first one is simple, I would send the anchor tensor and then send the delta tensor creating the two phase sending procedure. One big chunk of data would be split in to 2 smaller portions. This would be used when facing low bandwidth where sending the whole chunk would hinder the transmission process. However, this would still have the problem as when facing even lower bandwidth, entire chunk of delta tensor might be too big and sending text would be a better solution, thus the problem of computation overhead remain.

The second idea I have is when during sending phase, we will first calculate delta tensor for all groups and depends on the bandwidth:

1. During high bandwidth, CacheGen would send the anchor tensors first. Only when all anchor tensors are sent, we would send the delta tensors.
2. During low bandwidth, CacheGen would send delta tensors. What special about this idea is that would would send delta tensor not in groups, but send each delta tensor individually.


Note that we still use 8-bit quantization, a relatively high precision, on the KV cache of the anchor token (the first token of a token chunk). This is because these anchor tokens account for a small fraction of all tokens, but their precision affects the distribution of all delta tensors of the remaining tokens in a chunk. Thus, it is important to preserve higher precision just for these anchor tokens.

According to section 5.2 of the paper (as shown above), the anchor tensor would have a higher precision compare to delta tensor. Each individual delta tensor shall not be too much larger than individual text token (need further experiments) and thus we do not need to send text format.

The anchor tensor and delta tensors would reunite at the server side and when a whole group is reunited, the server would put them into storage backend.

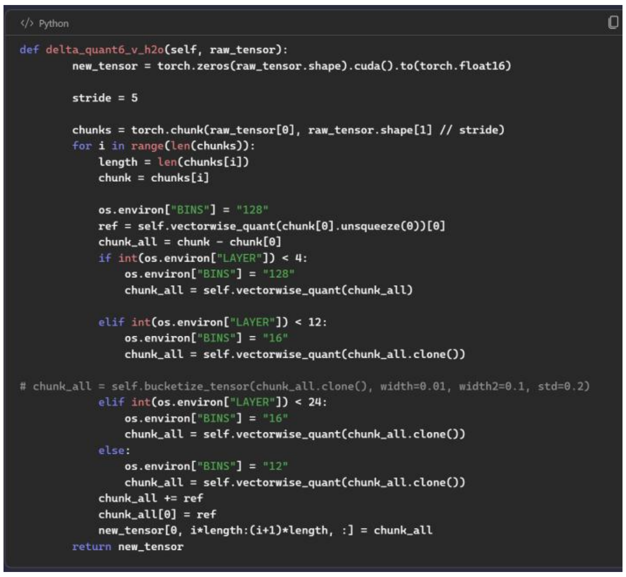
We would however, need to implement identifier for each delta tensor so the server side know which anchor tensor group the delta tensor is in and what is the position of the delta tensor within the group.

The drawback of this idea would be if bandwidth continue to be low when we send all the delta tensors, we might need to send anchor tensors in low bandwidth which would increase the transmission overhead.

Implementation

To add the separation of anchor tensor and delta tensor, I would first tried to locate the part in the source code where the author implement the dynamic adaption to the bandwidth. After spending a lot of effort diving in the source code, I could not find it. Finally, I reach out to the author via email to see if I was missing something. However, the author confirm the dynamic adaption to bandwidth was not implement in the code available, she might add in for future updates. This implementation would take time much larger than the rest and and would leave the implementation after her updates.

I then tried to find where the delta tensor was produced through tracing the source of KV Tensor but the result was unfruitful. After some discussion with the author, this part of the code is also not publicly available yet and they would push it latter. What she could provide is a snippet of how delta tensor is produced:



```python
def delta_quant6_v_h2o(self, raw_tensor):
        new_tensor = torch.zeros(raw_tensor.shape).cuda().to(torch.float16)

        stride = 5

        chunks = torch.chunk(raw_tensor[0], raw_tensor.shape[1] // stride)
        for i in range(len(chunks)):
            length = len(chunks[i])
            chunk = chunks[i]

            os.environ["BINS"] = "128"
            ref = self.vectorwise_quant(chunk[0].unsqueeze(0))[0]
            chunk_all = chunk - chunk[0]
            if int(os.environ["LAYER"]) < 4:
                os.environ["BINS"] = "128"
                chunk_all = self.vectorwise_quant(chunk_all)

            elif int(os.environ["LAYER"]) < 12:
                os.environ["BINS"] = "16"
                chunk_all = self.vectorwise_quant(chunk_all.clone())

# chunk_all = self.bucketize_tensor(chunk_all.clone(), width=0.01, width2=0.1, std=0.2)
            elif int(os.environ["LAYER"]) < 24:
                os.environ["BINS"] = "16"
                chunk_all = self.vectorwise_quant(chunk_all.clone())
            else:
                os.environ["BINS"] = "12"
                chunk_all = self.vectorwise_quant(chunk_all.clone())
            chunk_all += ref
            chunk_all[0] = ref
            new_tensor[0, i*length:(i+1)*length, :] = chunk_all
        return new_tensor
```

This part of the experiment would not have a demo as the backbone is unfinished in the public repo available. The pseudo code I implement complete:

1. The separation of individual anchor and delta tensor.
2. Header (identifier) to be used for reunification at server side.

Special thanks to the two authors of the paper, Yuhan Liu (main contributor) and Yihua Chen (CUDA implementation). Through unable to access the full CacheGen with anchor/delta tensor encoding and dynamic bandwidth adaption, Yuhan provided some more insights to the idea of the paper and provide more information on how would the implantation of the encoding would look like. Yuhan discuss more information on planned future update of CacheGen and happy for collaboration on CacheGen and future related work in the field.

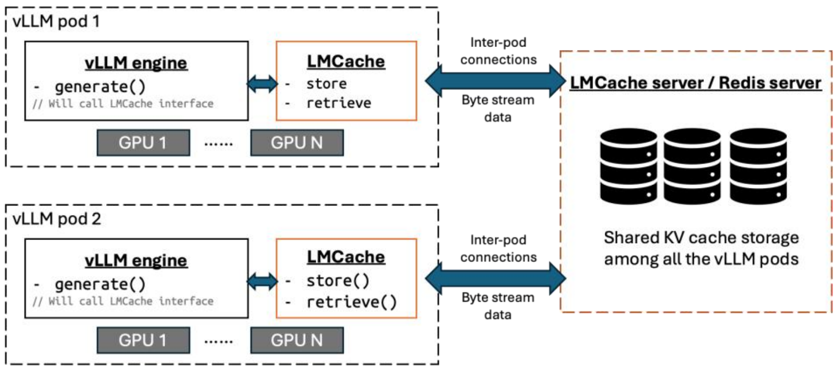
Future Work

After I my research, I earned basic understanding of KV cache and quantization, but the how quantization affects the precision of the output of LLM is still unclear to me. Dive deep in to the architecture of KV cache and quantization would be interesting and facilitate new ideas on network applications on AI.

Source Code

- https://github.com/troyyxk/CacheGen/tree/seperate_delta_tensor

(2) Distributed CacheGen



As we can see from the about architecture, the current CacheGen requires the a separate machine for a centralized server. One question arise is whether we can have the LMCache sever distributed.

There are several advantages of having distributed service vs a centralized server:

1. We can save 1 machine which was supposed to be used for a centralized server. In the distributed service, every machine could act like the server and thus getting rid of the needs for a separate server machine.

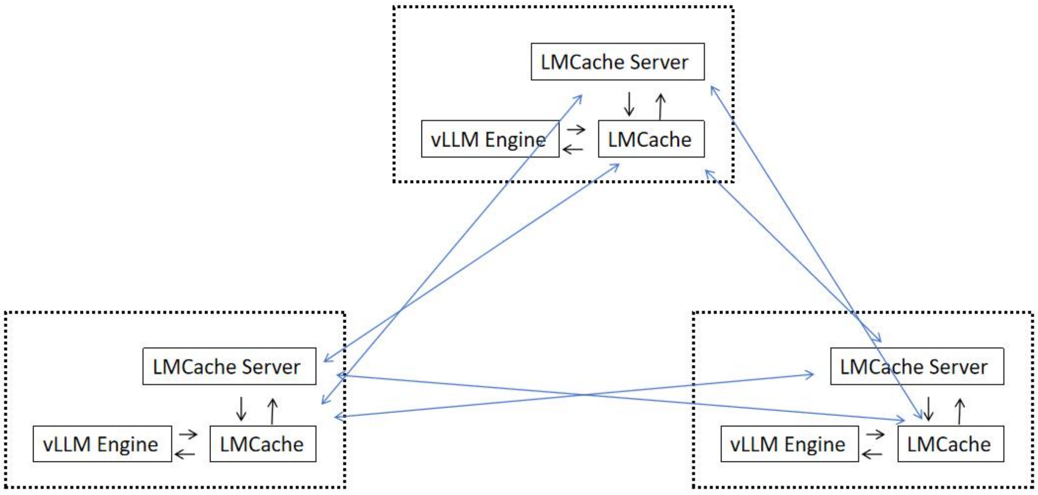
2. When all machines in the system acts like a server, there exits probability that the KV chunk request by the local vLLM engine is locate at the local machine and it is then able to fetch local which would eliminate transmission time.

3. Centralized server may have limited storing space in comparison to distributed service which has storing space of all machines combined.

Therefore, I would try to implement a distributed CacheGen.

Implementation

There are two ways of implementation I have in mind. The first one, is to have each LMCacheConnector acts like both a server and a connector. The second one is to have both LMCacheConnetor and LMCache server deploy on all machines. My implementation would go with the second one as mix use of client and server is not considered a good practice and there is preexisting infrastructural lay down by the author and it could also adapt the author's future development on the engine.



The above figure show case how a 3 machine cluster distributed CacheGen looks like.

One implementation detail we have to discuss here is how does a LMCacheConnetor know which server should it send a request from. My solution here is to have CacheEngineKey hash in to a sha256 int [5]. The new CacheGen would have configuration of how many servers are there in the cluster and would use integer representation of CacheEngineKey hash by the number of server to know which server contains the KV cache the key referring to. For instance, in a 3 server situation, a get call with a key of sha256 value 80262417 would points to 80262417 % 3 = 0, the first server.

This would however have another problem. When the total number of servers in the cluster is not a prime number, due to common factor and pattern repetition, the distribution of the KV cache in each server would be less than ideal in comparison to the one with prime number of server. One optimization idea to solve the load balancing problem is to force prime number of server. For instance, when having 4-6 servers, we would only provide options for 3 LMCache server, or when having 8-12 servers, the only option would be 7 servers. This would however still waste many server so the ideal optimization would be left for future works.

Future Work

The first future work would be find a better solution for load balancing.

The current implementation has one major draw back that is assumes all server will stay alive which is generally not the case with most of the distributed systems. Applying consensus algorithm like raft would help in my opinion.

Finally, the current implementation does not allow dynamic add and delete server from the system. A future work on this topic would be necessary.

Source Code

- https://github.com/troyyxk/CacheGen/tree/ditributed_CacheGen

(3) Server scheduling for multiple call to backend

Inspired by CPU scheduling problem in operating system, I came up with the idea of request scheduling.

CPU scheduling arise from limited CPU resource. OS would try to make the system more efficient, faster and fairer with the limited resource. The current implementation of LMCache server sets no ceiling for the number of thread and the number of request it can sent at a time assumes unlimited resources. What would happen if there is not abundant network resource and there is a limit on how many package the LMCache server could send? How do we make sure the server works smoothly and fairly for all connecting LMCache connector?

Server scheduling came to help. It aims to provide efficient and fair server response for connectors given limited resource.

Implementation

The focus would be limited network resource where the bandwidth is only suitable to send a handful of resource at a time. Compare to the original input where all server messages are send when ready, the new implementation only a specific amount to be send at a time. This part is done by creating a specified amount of thread dedicated for request sending which we can call them sending threads. All send messages are store in the local data storage of the server allow put from receiving thread and pop from sending threads. Access to local storage is guarded by a thread lock to prevent race condition.

For ensuring the network fairness among clients (LMCache connectors) of this project, I took the idea from task priority in the CPU scheduling. In CPU scheduling, high priority are given to foreground (interactive) process with higher chance of execution compare to a background (batch) process. The scheduling want the more process interact the most with the user to be executed quickly so the user would not notice the existence of background waiting time. Similar idea can be apply here where a client with smaller amounts of request would be more sensitive to the network delay compare to the ones requesting large amount of data as the later one has the expectation for the data to take longer time to process. I implement it by getting the total number of requests waiting to be sent as a list, inverse each elements, and softmax the list. The resulting list would have a sum of 1 and the lower the amount of request waiting to be send for a client, the higher chance it would get pick.

Future Work

Network fairness is a really interesting topic. Diving deeper into how network fairness can benefit KV Cache and LLM in would be a topic to work on.

Source Code

- https://github.com/troyyxk/lmcache-server/tree/request_scheduling

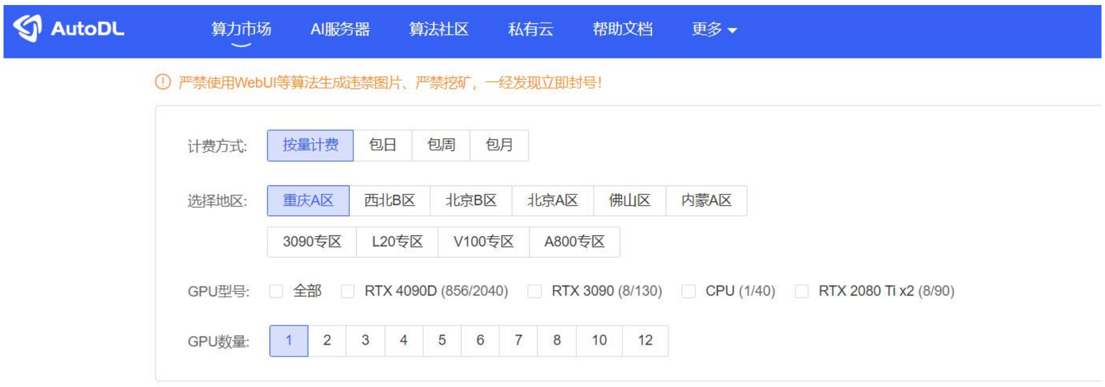
(4) Server Caching

This idea is inspired by the name of KV cache, it has cache in it and maybe apply cache policy? :) Maybe the lmcache server run out of storage and would request sweeping some kv cache out of the storage. The sweeping would be better decided with cache policy.

However I realized this is more of a server system optimization problem problem than a network problem and thus we will not dive deeper into this problem.

(5) Central and Regional Server

Before reading the paper, I thought GPU are usually locate closely with high-speed links connecting each other. The paper raise concern onto low and varying bandwidth and sparks my interests on how shall we deploy a cluster onto even lower bandwidth. What are such situations? The AutoDL online GPU platform I am using give me an idea.



严禁使用WebUI等算法生成违禁图片、严禁挖矿，一经发现立即封号!

计费方式: [✓ 按量计费] 包日 包周 包月

选择地区: [✓ 重庆A区] 西北B区 北京B区 北京A区 佛山区 内蒙A区 3090专区 L20专区 V100专区 A800专区

GPU型号: 全部 RTX 4090D (856/2040) RTX 3090 (8/130) CPU (1/40) RTX 2080 Ti x2 (8/90)

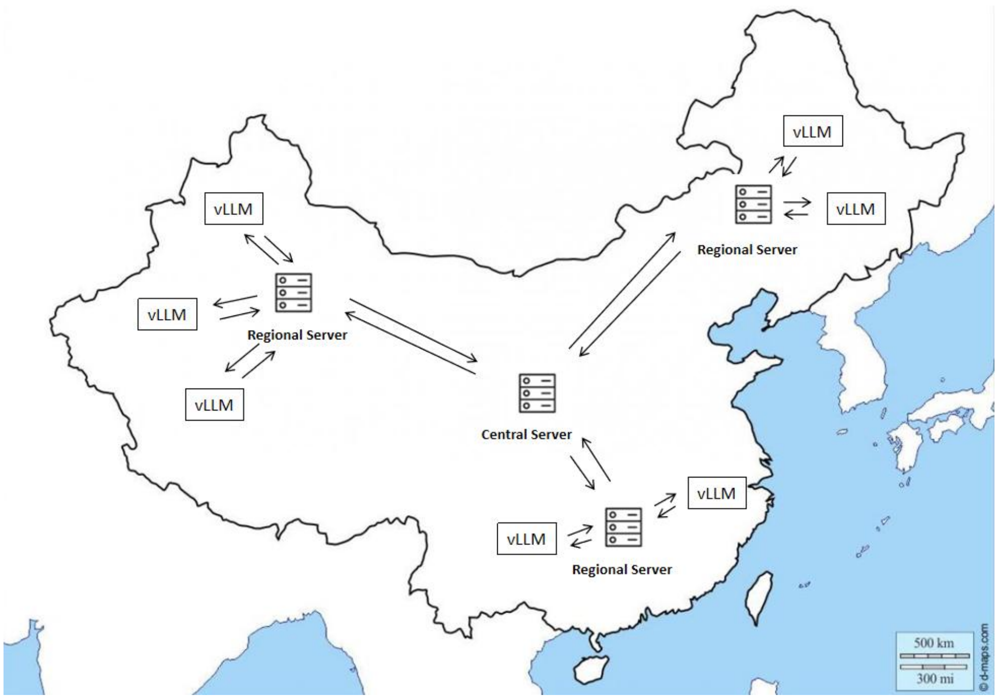
GPU数量: [✓ 1] 2 3 4 5 6 7 8 10 12

As we can see from the screenshot above, AutoDL has GPUs locate in different geographical physical locations across China. There may be situations where a person would gather GPUs from different regions to work together. The 2 problems arise from here are:

1. how shall we build such a cluster that minimize the network delay from GPU to server
2. how to enable cooperation (sharing KV cache) of GPUs that is such sparsely populated

The solution I proposed here is: central and regional server formation.

Implementation



As we can see from the above image illustration I made, GPUs (encapsulate by vLLM) are located in different geographic regions on the map. For each region, there would be a regional server that has faster connection links between it and GPUs of the region. For each GPU, the regional serer would provide the same API as the lmcache server the author provide. The network delay between the regional server and the GPUs of the region would be minimized and thus problem 1 is solved.

Each regional serer would communicate with the central server and synchronize the kv cache they store periodically. This work behind the scenes for the vLLM and wouldn't add network delay between GPUs and regional server. This would partially solve problem 2 but still left questions around the timeliness of the KV Cache. To address the timeliness, I add an additional parameter to the client get message called force_lastest. If a regional server received a get message with force_latest=1, it would first synchronize the KV cache of the key in the client get message with the central, and then reply back to the client. A client get message with the force_latest=1 would take more time but would get the latest KV cache.

Future Work

This idea turns out to be more difficult to implement than I imagine it to be. There are several flaws with the current implementation. The major one been current implementation let regional server in charge of the synchronization process, and the regional servers has no idea the synchronization status of KV cache of other regional servers. The client get call with force_latest=1 would only guarantee the latest version of KV cache if the KV cache is from the region. One idea I have to address this problem would be moving the synchronization management from regional server to central server in the future.

Source Code

- https://github.com/troyyxk/lmcache-server/tree/central_regional_server

- https://github.com/troyyxk/CacheGen/tree/timestamp

(6) Wireless

I do think it is possible to deploy CacheGen in a wireless setting but there are currently no potential improvement ideas focusing on wireless. Problems I can think of with wireless CacheGen implementation including low and volatile bandwidth would hurt the quantization level of KV cache and thus do harm to the resulting accuracy of the output of the LLM. Mobility between regions in a wireless setting would cause gap in connection and change in server address require dynamic update from lmcache connector.

(7) Local Cache

Local cache for connectors! This idea is rather straight forward. I will put local cache onto the connector. The connector would first consult the local cache and then consult the remote server.

The difference between this idea and the 4th idea is that applying local cache would make use of edge device and affect network traffic compare to server cache which would only have affect on system side.

Implementation

There is an existing local storage backend in the source code and I will make use of it. Cache policy here would be LRU and the discarded KV cache would be send to remote server.

Future Work

Future work of this idea including applying different cache policy and examine their efficiency in CacheGen.

Source Code

- https://github.com/troyyxk/CacheGen/tree/local_cache

4. References

[0] YingSheng,LianminZheng,BinhangYuan,ZhuohanLi,MaxRyabinin,DanielY Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. 2023. High throughput generative inference of large language models with a single gpu. arXiv preprint arXiv:2303.06865 (2023).
[1] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. arXiv preprint arXiv:2208.07339 (2022).
[2] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic Coding for Data Compression. Commun. ACM 30, 6 (jun 1987), 520–540. https://doi.or g/10.1145/214762.214771
[3] [Mathematicalmonk]. (2011, September 25). (IC 5.1) Arithmetic coding - introduction [Video]. YouTube. https://www.youtube.com/watch?v=ouYV3rBtrTI
[4] [KilObit]. (2024, May 23). VMware Workstation Pro is Now FREE (How to get it) [Video]. YouTube. https://www.youtube.com/watch?v=66qMLGCGP5s

[5] [JJC]. (2017, February 7). How to hash a string into 8 digits? [Video]. Stack Overflow. https://stackoverflow.com/a/42089311

[6] Exploring collaborative distributed diffusion-based AI-generated content (AIGC) in wireless networks H Du, R Zhang, D Niyato, J Kang, Z Xiong, DI Kim... - Ieee network, 2023

5. Source Code form the author of CacheGen

https://github.com/LMCache
https://github.com/UChi-JCL/CacheGen?tab=readme-ov-file

Thanks for reviewing. Best of luck on future researches!