# **Experiement Log on CacheGen**

## Paper review

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

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 tokens. 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.

## Environment setup

### System

#### 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

#### 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.

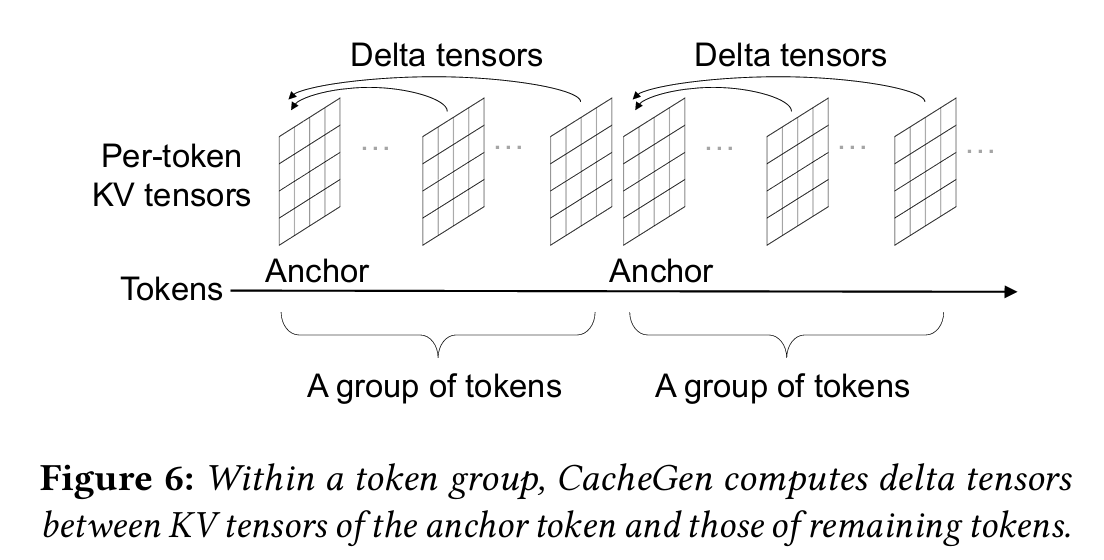
#### 3.Online GPU Environment.

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.

## Potential Improvement from Network Point of View

### 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.



As we can see from the figure from the paper, the quantization is applied on each group of token. Each group consisted of an anchor token and the others. The anchor token is quantized with higher . 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 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 the send the delta tensor creating the two phase sending procedure. 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, sending text would be a better solution and 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 all of the anchor tensor 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. 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 client side and when a whole group is united, the client and start decode the group. Or when client receive a delta tensor, it would determine if it has received the anchor tensor.

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

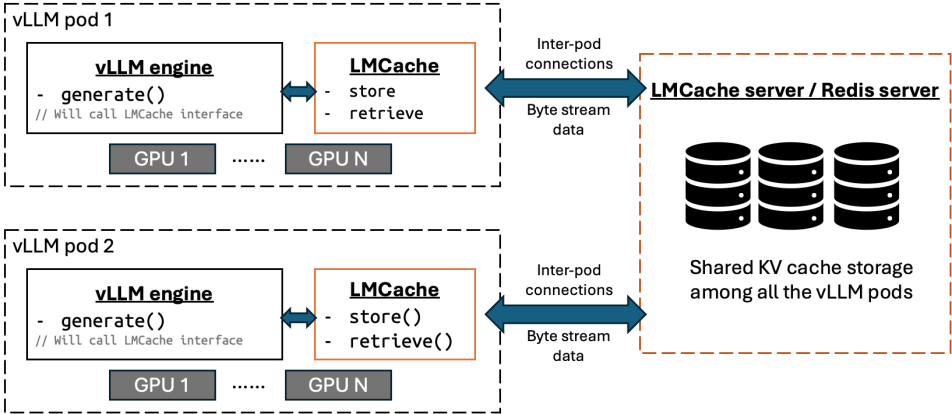
The drawback of this idea would be if bandwidth continue to be low when we send all the delta tensors, we might need to send anchor tensors in low bandwidth which would increase the transmission overhead.

#### Implementation

To add the separation of anchor tensor and delta tensor, I would first need 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. I reach out to the author via email to see if I was missing something. To my surprise, the author confirm it 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 decided to work on the separation of anchor token and delta token. TODO

### 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 it has preexisting infrastructural lay down by the author and it could also adapt the author’s future development on the engine.

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.

The implementation is in ditributedCacheGen branch of CacheGen repository, link: https://github.com/troyyxk/CacheGen/tree/ditributedCacheGen

#### 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.

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

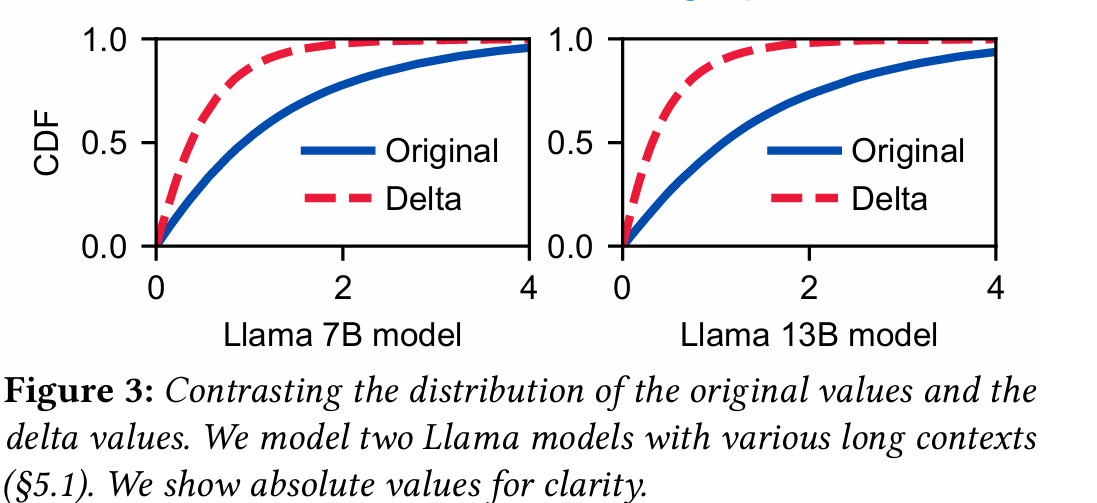
### Server queue for multiple call to backend

Inspired by operating system

## Questions

- What is a layer and channel here?

- Dont really understand this one:



- Isn’t the first insight contradict the third insight?

## References

1. 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).
2. 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).
3. 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
4. [Mathematicalmonk]. (2011, September 25). (IC 5.1) Arithmetic coding - introduction [Video]. YouTube. <https://www.youtube.com/watch?v=ouYV3rBtrTI>
5. [KilObit]. (2024, May 23). VMware Workstation Pro is Now FREE (How to get it) [Video]. YouTube. <https://www.youtube.com/watch?v=66qMLGCGP5s>
6. [JJC]. (2017, February 7). How to hash a string into 8 digits? [Video]. Stack Overflow. <https://stackoverflow.com/a/42089311>

#### Soruce code:

<https://github.com/LMCache>

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