Hashing Table Prefixing Implementation Report

Ziyuan Lin

2024.3.17

1 Basic Idea of Hash Prefix Caching

In this small project, I implement the prefixing caching using hash table. With the modern hash function, the collision rate will be extremely small. Hence any block will be totally determined by its prefix and own tokens. Thus $hash(prefix, own)$ uniquely defines a block. Using this prefix technique, all sequences in the entire life cycle of the engine can share the prefix. Reducing the time for allocation and computation. When a request finishes, we do not immediately remove the cache. Instead, we fix a pool of prefix cache with maximum size K as a hyperparameter. When a block is freed, the block is added into the pool. When the pool is full, the cache pool starts to evict the block from the pool. The eviction policy is to remove block first based on the last utilized time, then on the prefix length of the block. The later the block being used implies that the block is frequently used, hence last to remove. For the blocks with the same last touch time, remove the block with the longest prefix to prevent from removing the token in the middle.

2 Details of Design

The initial thought was to implement only a hash table. It records "all" possible prefixs. Let's talk about the insertion mechanism first. When a new prompt comes, say it consists of 4 blocks, 3 full blocks and 1 partial block, then all 4 blocks are added to the hash table as prefixs. When the sequence grows, two cases happens. In the first case, a partial block gets longer, then I need to update the prefix in the hash table. In the second case, the original partial block is now full, and a new partial block(with only one entry) is now generated. In this case, I need to add a new prefix for this new partial block. Now let's discuss about the eviction mechanism. In this mechanism, the first thing to care about is the reference count attribute of the block. If the reference count is larger or equal to 1, that means we are currently generating the sequence corresponding to this block. If we evict this block, then we can no longer generate the sequence. Hence we can only remove those blocks with reference count 0. The reference count decreases to 0 when the sequence of the corresponding block finishes, and the block is ready to be freed. Hence we have no guilty to remove it. But why

not keep them even if they have reference count 0? Because if a future sequence is the same as it, we can reuse it, which could largely increase the performance and make full use of hash table. However, this causes some problem: memory. If we still maintain all prefixes even if the reference count decreases to 0, then we may run out of cpu memory. The bad thing to run out of cpu is we are no longer able to switch the block from gpu to cpu. Hence, a trade off exists.

What I decided to do is setting a maximum number of blocks(prefixes) with reference count 0 maintained in the hash table. When I think like this, I decided to build another structure to store the information for those blocks with reference count 0 that are ready to be freed, but still inside hash table. Note that this new structure is only responsible for storing the information of blocks, like hash of the block(Also last access time and prefix length). The block itself is still inside the hash table. I call this structure cache pool. So when we notice a block's reference count decreases to 0, we decide to insert its information into the cache pool. There are two cases. In the first case, the cache pool is not full yet, then simple insertion is enough. In the second case, the cache pool is full. Then we need to insert and remove. The block I decided to remove based on two criterions, namely, last accessed time and prefix length. I use a nested heap to implement these.(Basically a heap of heap). At the same time, I need to really remove the block in the hash table now. Finally, when a new prompt comes, and we find one of its block is in the hash table. In this case, we need to check whether or not this block is in the cache pool, that is whether it is one of the block that has reference 0(Ready to be freed because the corresponding sequence finishes). If this is the case, I remove this block's information in the cache pool, because its reference count is no longer 0 – some sequence needs it! If do not remove it, a bad senario happens. When a new block is ready to free, then it might squeeze out this block with positive reference count which is way too bad. That's why I need that remove code there.

3 Experimental Results

The experiment shows that adding hashing prefix cache can indeed improve the performance. However, this can only be achieved in some extreme scenario. In this section, I am going to present some results and explain the reasons behind these results.

3.1 Random Prompt Generator

I create a random prompt generator. The generator has a pool of frequent sequences. For each loop, it has probability p to select a request from the pool, and $1 - p$ probability to generate a random request with length 15. I generate 1000 requests to form the prompt.

3.2 Hit rate

Range p from 0.1 to 1.0. Here is how hit rate changes. In this work, the hit rate is counted only once for the initial prompt.

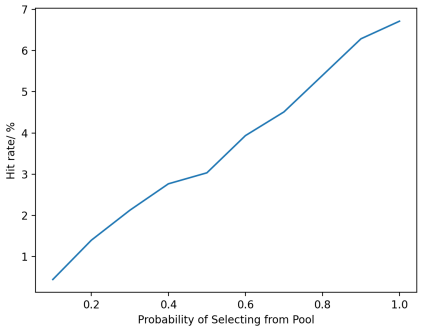

Figure 1: Hit rate vs Sampling probability

As we can see, hit rate increases almost linearly with respect to probability of sampling from the pool. I also test the case with only one type request. For instance, the hit rate of sequence "What is your name?" is 8.14%. This totally makes sense since as p increases, matched or closed matched sequences increases. For identical sequences, 8.14% is not that high because the generated sequences are different even though they have the same prompt tokens.

3.3 Performance

I use the items per second as the metric for performance. Figure 2 shows how the items per second changes with respect to p. Note that the average items per second for non-prefixing version is 416.92.

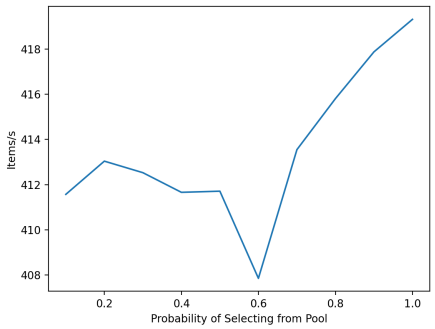

Figure 2: Items per second vs Sampling probability

We note that the performance is not quite stable compared to the hit rate. This is due to two factors. On the one hand, the new scheme needs extra work on calculating the hashing value and managing the cache pool. On the other hand, the randomness causes large difference in requests, hence quite different final sequence length. However, we can see that even with extra work, the performance of the hashing prefix version still outperforms original no prefixing case for large enough p. For the extreme case of identical prompts, the performance is even better. For instance, "What is your name?" achieves 459.22 items/s.

4 Future Work

Here I list 3 potential future work: (1) Change the direct hashing to hierarchical hashing. That is assume that each block is identified by $hash(prev\_hash, own)$. This assumption should holds since even with hierarchy, the hit rate is still small. (2) Change the eviction policy. We may change the eviction policy. For example, the first criterion may be based on the frequency touched by gpu. (3) Further experiment. I think current experiment is not enough to understand the whole story. More prompts can be tried. Also I need to find a better performance criterion.