A Memory Mapped Data Loader for Deep Learning

Chen Zhu Peng Ding
ShanghaiTech Univeristy
{zhuchen, dingpeng}@shanghaitech.edu.cn

I. Introduction

Data loading, which refers to the procedure of accessing and decoding the data stored on the disk, is an important part of any machine learning system. The datasets are usually larger than the memory of the machine, so the process will constantly read the disk. If not handled with caution, especially when the data is stored as individual files on HDD, data loading will consume an unignorable amount of time. To reduce time consumption, a direct approach is to reduce the "buffer miss rate" of the main thread, which increases the the requested data's chance to be in the memory and can be achieved by prefetching data into the memory with other threads.

To make prefetching efficient, we need to know the order in which the main thread accesses the data. Almost all deep neural networks are trained with stochaistic gradient descent (SGD), which requires accessing the data in random and different orders between epoches to prevent overfitting when the dataset is not large enough[1]. Under such a requirement, we either provide the prefetching process with the exact accessing order of the main thread, or we add some constraints to the randomness of the accessing order so that the prefetching thread knows the approximate accessing order without knowing which epoch the main thread is in.

When we need to tune the hypterparameters, we can run multiple processes on a single machine on different GPUs. To save memory and maximize the potential memory buffer size, the processes can share the memory used to store the prefetched data. However, if the processes are at different epoches, even when the random seed is fixed, the processes may access different pieces data for a long period, which may result in a 50% buffer miss rate in the worst case for 2 processes. To deal with such a situation, we can put some constraints on the randomness of the accessing order to increase the locality. One approach is to divide the dataset into blocks, and shuffle the order of accessing only inside that block. In this way, when the 2 processes are accessing the data inside the same block, there should be no conflict since the blocks are usually smaller than the memory size. When 2 processes are accessing different data blocks, their speed difference may finally enforce them to access the same blocks.

Some deep learning frameworks have already provided efficient data loaders. For example, MXNet [1] provides a parallelized data loader for images, which utilizes multiple threads to prefetch the dataset stored as a whole compact file into memory. However, the memory is not shared for different processes, which wastes the memory space. Its shuffling is confined inside the blocks, which should only serve to simplify the design of prefetching threads.

[1]To the best of our knowledge, almost all datasets for research, including large datasets such as ILSVRC2012 (1.2 million training images), are traversed many times before the networks converge.

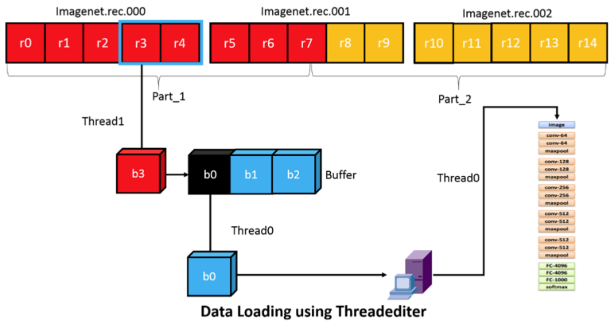

Fig. 1. The MXNet data loader.

Inspired by the design of MXNet data loader and the motivation to improve space efficiency, we also divide the dataset into "blocks", shuffling the accessing order within a certain number of data records, and use memory mapped files to share the memory across processes. Apart from sharing memory buffer, memory mapped files also saves an extra copying from the kernel space to the user space. For implementation, we build upon LMDB, writing a generalizable Python wrapper for C++ to provide data to the main thread and start another thread to prefetch data into the mapped memory.

II. Methods

A. Comparison of potential methods

1. Direct File Accessing:

Storing the data as individual files is the most flexible way for accessing the data. It has no restriction on the file format and as long as there is sufficient storage, the amount of data is limitless. However it is not memory and I/O friendly. If the training set consists of corpus is stored as plaintext file, the I/O request will be very often and it causes huge overhead for training. Even thought programmer may access the files as memory mapped file, since they are distributed in a non-consecutive manner on the filesystem, the I/O will still be an overhead.

2. Database:

Usually, data are stored within database. Data within the database is stored consecutively on disk. That will help the I/O and memory perform better. And key value pair provides a very convenient way for accessing. But when one wants to access the data in a perticular fashion, he should spend time getting familiar with the database language.

3. Built-in Loaders

Nowadays, many frameworks provides specialized data loaders for their users. MXNet RecordIO files and its iterator is one of them. The training dataset is stored as a sequence of MXNet RecordIO files, which help to collect data and store them in a consecutive manner. Even thought there can still be several pieces if the dataset is very large, the collectiveness is enhanced. When read from the file, there will be two threads corporating in a producer-consumer manner, one for prefetching and one for loading. The loading thread (t0) will send message to the prefetching thread (t1) for accessing the targeted data. t1 divides the dataset into several blocks and locates which the target lies in. Then it will walk through the targeted block and the block of data will be stored in the local buffer of prefetching thread (t1). Loading thread (t0) will load the data from t1's local buffer. This Producer-consumer model helps save the time for loading the data lying in the same data block, since the targeted block is already loaded into memory. The loading thread shuffles the order of accessing only inside the block. However, when trying to use multi-process for training, accessing the same data block concurrently will end up with several copies, as each process own its memory space. And the loader can only support image files, so it lacks flexibility.

B. Our Choice

Since this loader is primarily designed for training the neural network, we aim to avoid overfitting. By analyzing the existing loader types, the producer-consumer model of prefetching the data is appreciated. And we want to train the network in a multi-process fashion. Meanwhile, memory performance and efficiency should be guaranteed. This can be achieved by using shared memory, but we want a lock-free structure. At the same time, we want to ease the burden of I/O, and avoid replication of data. Generalizing the usage of our data loader is also essential, which means we need to have a universal applicable storage format.

Considering all of these requirements, we make the following decisions:

1. Block-wise shuffle, with batch-wise loading
2. Producer-Consumer Model of prefetching
3. Memory Mapped File Database

As for the database, we choose LMDB for 3 reasons. First, it accesses data records more efficiently as its storage is organized as a B+ tree. Second, it has already enabled memory map which simplifies the design process. Third, it is also used by some deep learning frameworks such as Caffe, and our code should be readily for existing datasets stored in LMDB.

C. performance analysis

The direct disk read and write can be modeled with Figure 2. We can see there is two memory copy. One happens when the interrupt handler shipping the data from the disk to the kernel space, and then the kernel copy the data from kernel to the user space. That's two data copy. With memomry mapped file, the process reads the file by accesssing the pointer, and if the accessed data is not in the memory, interrupt handler will break in and copy the data from the disk to user space. There

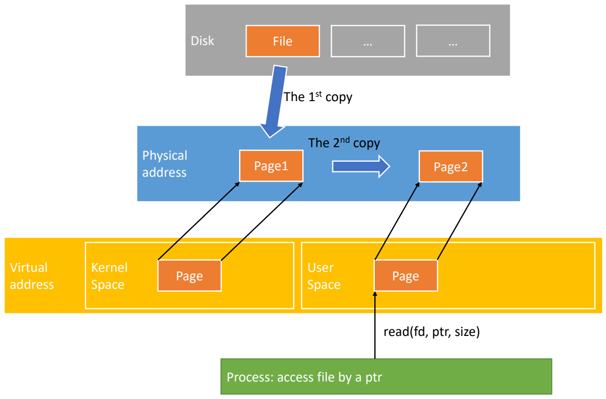

Fig. 2. Direct disk I/O.

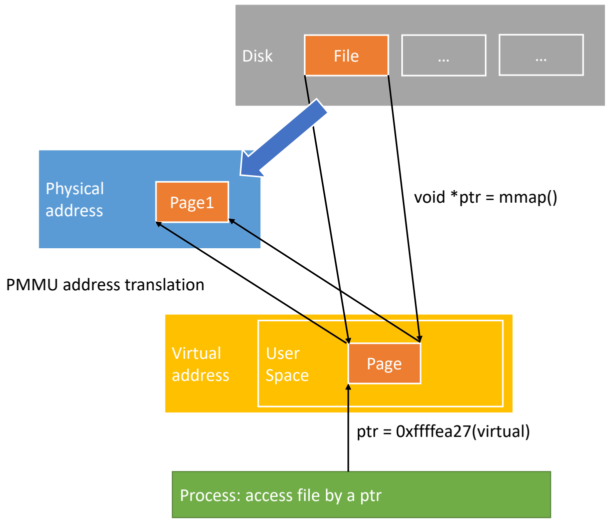

Fig. 3. Memory mapped file saves one copy.

is one less data copy, as shown in Figure 3 and this is one of the reasons why LMDB is faster than other database.

Another advantage of using memory mapped file is that the computation can start immediately after the first batch of data arrived. There is no waiting time as in conventional I/O read. So the efficiency of data loading and network training can increase a lot.

D. Producer-consumer model

Here we give two views of the producer-consumer model in our program. From a programmer's point of view, as shown in Figure 4, the entire dataset is considered as divided into several blocks with a predefined block size. And the block size should be bounded by the physical memory size of the programmers' computer. The prefetching thread walks through the targeted block and the memory mapped file trick guarantees that the touched data will be loaded into the memory. Then the loader thread query the batch of data belongs to that data block, which is already stored in the memory. So the hit rate can be much higher, and the accessing time can be cut down massively. The loader thread conduct shuffle on the list of pointers to the data batches.

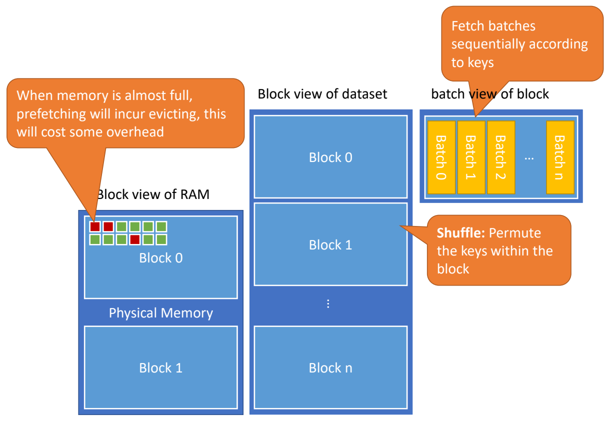
Fetch batches sequentially according to keys
When memory is almost full, prefetching will incur evicting, this will cost some overhead
Block view of dataset
batch view of block
Block view of RAM
Block 0
Block 1
Block n
Batch 0
Batch 1
Batch 2
Batch n
Shuffle: Permute the keys within the block
Physical Memory

Fig. 4. Programmer's view of producer consumer model.

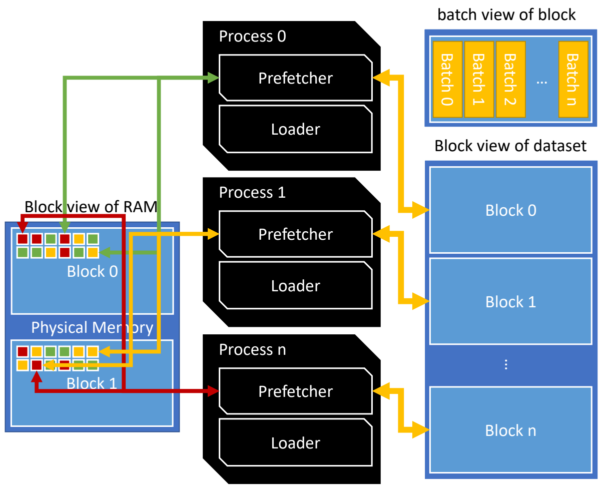

Fig. 5. System's view of the producer consumer model.

From the system point of view, as shown in Figure 5, the entire dataset is a mmap address space. And the actual data is stored as LMDB pages on the permanent storage. The LMDB database use key value pair for data retrieving. The data on each block will be simultaneously loaded into memory by the prefetching thread on different training processes. Normally, the filesystem should take care of the location where to write each data entry, and there should be low collision rate and relatively higher hit rate. But when the total size of all the targeted blocks exceeds the maximum buffer size on the machine, the page replacement policy and the data accessing pattern will have very important influences on the hit rate. If they collide with each other, then the hit rate will be greatly impacted. But fortunately, we only read from the LMDB in training phase. Then there is no dirty page, so the replacement can directly prune the entry without worrying about writing back to the disk. There is zero overhead.

III. Experiments

In the experiments, we train a neural network for visual question answering. The network has 59.6 million parameters. The input to the network includes extracted image features and tokenized questions. The compressed image features are 300 KB on average and the total size is about 29.5 GB. Each

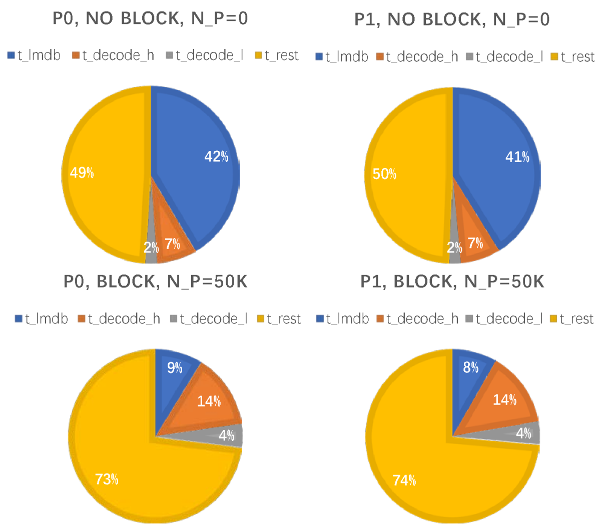

Fig. 8. Portions of each procedure.

image has up to 5 related questions. Before each experiment, we clear the memory buffer to ensure fair comparisons. We use n_p to represent number of questions in each block. Since the time for accessing and decoding the feature vectors are significantly larger than that of the questions, and multiple questions may share the same image feature, the questions are sorted according to their related questions and divided into blocks following that order. In this way, the prefetcher accesses unique image features and becomes more efficient.

A. A single process

We evaluate the influence of prefetching on a single training process on both SSD and HDD. We report the average time consumption for training 100 batches during an epoch. The results are listed in Figure 6

B. Two processes with limited memory

To study the case when memory is smaller than the data size, we use the uncompressed image features which will inflate into 200GB. The memory of the machine is 64GB. Read speed for disk and memory are 510.09MB/s and 7826.62MB/s respectively, which means we can expect a maximum speed multiplication of 15.3 when all data are in the memory. We use $n_p = 50k$, which results in a block size of 9GB. The 2 processes start at the same time in the 2 experiments. We compare the training time for $n_p = 0$ and $n_p = 50k$. The total training time for the 2 processes when $n_p = 0$ are 250.74s and 248.93s respectively, which is reduced to 129.36s and 127.74s with $n_p = 50k$. From Figure 8, we can see the 2 processes achieved approximately 5 times acceleration for data accessing with prefetching.

C. Four processes with abundant memory

In this experiment, we use the compressed image feature, so the memory is abundant. We run 4 processes simultaneously to test the pressure on CPU caused by prefetching. The detailed times for each procedure are listed in Table I. Although t_{LMDB} decreased due to prefetching, t_{others} increased, which is probably caused by a heavier CPU burden.

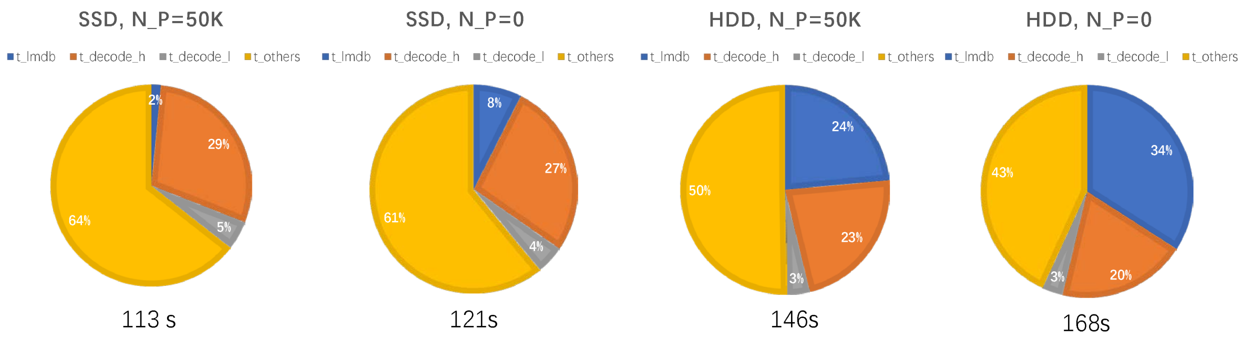

Fig. 6. Portion of data accessing, decoding and training in each case. t_{LMDB} represents the time for reading data from LMDB, t_{decode_h} represents the time of decoding the compressed image data, while t_{decode_l} stands for the time of decoding the queestion data, and t_{others} may include copying between host memory and device memory, as well as forward-backward and weight updating. For SSD, I/O is not a significant bottleneck, while for HDD, I/O is a bottleneck and prefetching reduces up to 13% of the average training time.

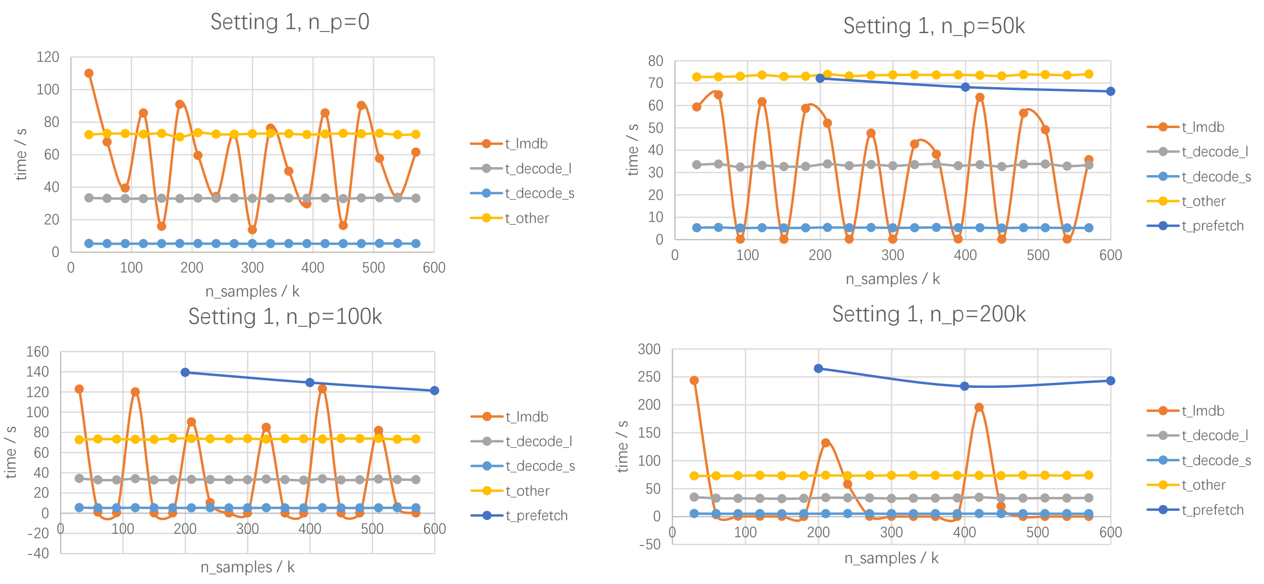

Fig. 7. Time consumption pattern of every 100 batches. When $n_p = 0$, no prefetching is performed. t_{LMDB} is more random when $n_p = 0$, and more periodic when $n_p > 0$. Other times remain constant for different n_p. For n_p=50k, 100k and 200k, the total time for 100 batches do not change much.

n_p	0	50k
t_{LMDB}	41.14	22.93
t_{decode_h}	20.97	20.41
t_{decode_l}	5.39	5.30
t_{others}	127.83	142.28
Total/s	195.33	190.92

TABLE I.

IV. Conclusion

Analysis and experiments demonstrate that the proposed method is able to reduce the time of data accessing by divide the dataset into blocks and prefetch data by blocks. A drawback of the methods is the additional prefetcher, which might slow down the CPU when multiple training processes are running. We may consider sharing prefetchers between processes in the future to further reduce the overhead of it.

References

[1] Chen T., Li M., Li Y., et al. *Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems*. arXiv preprint arXiv:1512.01274, 2015.