A. Quasi-Randomness vs. Webdataset Sharding

In Section 3.2, we mentioned that, while `FFCV`'s quasi-random loading strategy is similar to WebDataset's sharding, the two strategies differ in key aspects. Here we discuss in more detail why this is the case.

In systems like Webdataset, sharding is done to reduce the number of files and relieve some strain on the file system. However, each file system has a block size after which random reads become sequential reads. In almost all system, this lies below 2MB. So, if Webdataset was to create shards of 2MB, it would create a dataset with way too many files that it overwhelms the file system.

On the other hand, `FFCV`'s .beton format (by design) decouples this phenomenon by *sharding internally a single large file*. This means that our dataset is a single file the is structured to imitate sharding, as shown in Figure 3. This enables us to increase the quality of randomness as our shards are typically orders of magnitude smaller than WebDataset's. At the same time, we avoid overwhelming the filesystem as long as we pick the minimum shard size that will satisfy the file system while retaining a single file.

B. Why does FFCV Use Multithreading Instead of Multiprocessing?

In this section, we discuss in more detail (than Section 3.4) why `FFCV` relies to threads instead of sub-processes.

While many libraries, such as PyTorch, use multiprocessing during data loading to avoid the GIL, this comes with a large performance drop. In such strategies, all workers have to report back to the main process (running on a single thread). This often leads to lower performance with very large worker pools than with smaller as the main thread becomes overwhelmed. To avoid this, we resorted in `FFCV` to multi-threading instead, thus enabling each thread to write an individual sample directly in place in RAM without any inter-process communication. This also enables multiple threads to work simultaneously on the same batch, and avoids situations like when PyTorch's main thread could be sitting idle waiting for any of its workers to have their batches ready, although if these workers could accumlate their work (which is what we do in `FFCV`), the main thread would have not have been idle.

C. Comparison of JIT Compilers Alternatives

We motivate our decision to use Numba to compile FFCV's pipelines by elaborating on the pros and cons of potential alternatives:

`torch.script`: This JIT compiler, included as part of `Pytorch` was designed to optimize inference speed of deep neural networks and ease the transition between research code and deployment environments. In this context, it makes sense for it to only support the most basic python features and the Pytorch library itself. Relying on this solution for FFCV would have mean giving up on data augmentations written in `numpy` and interoperability with other deep learning frameworks. While `torch.scrip` can generate very fast code for some ML oriented operations (e.g matrix multiplications), it lacks some optimizations (`for` loops) and doesn't release the `GIL` which makes would have made it particularly unsuited for FFCV.

It is still important to note that it still possible to leverage `torch.script` within FFCV: users are allowed to introduce in their pipeline functions compiled with it as long as they operate on the GPU. Indeed, FFCV since doesn't compile those, it does not interacting with Numba and could even bring additional performance improvements in some scenarios.

`TVM`: TVM[4] shares many properties with Numba. They both are capable to generate LLVM IR, and can leverage the same optimization passes. Their performance should in theory be very similar. TVM has other advantages like being able to target other environments, this would not be useful for FFCV. The main differentiating factor remaining is the API. The familiar numpy augmented python seemed to be the most approachable for potential users. On the other end, TVM which was designed to represent and optimize complex neural network architectures would have been much more cumbersome to work with in this scenario.

D. Comparison with WebDataset

In Figure 7, we plot loader throughput compared with number of cores used. We find that standard `FFCV` configurations dominate WebDataset in performance.

[4] https://tvm.apache.org/

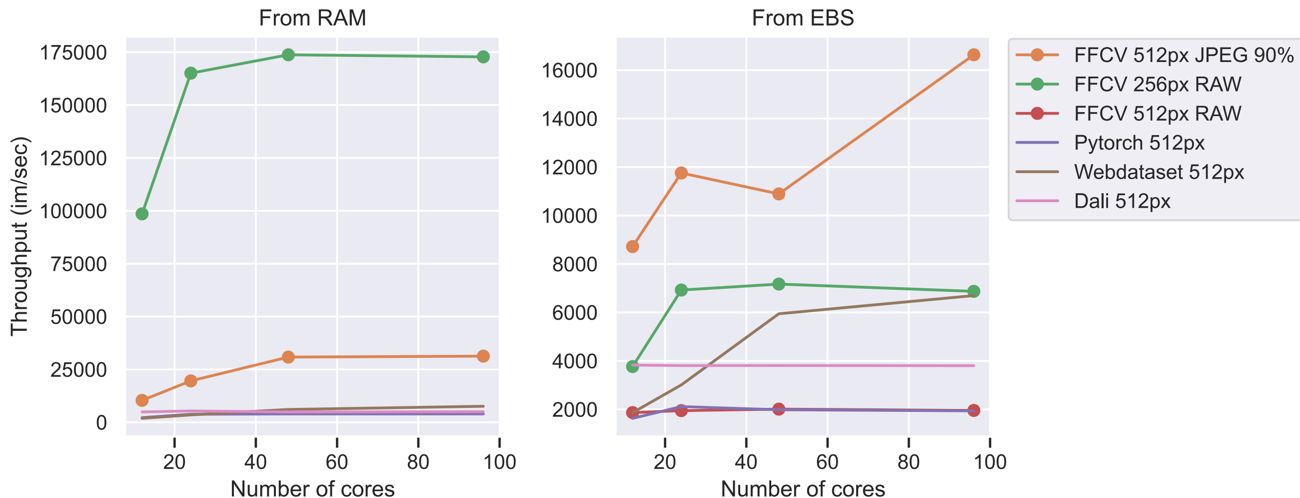

Figure 7. Comparison of FFCV with WebDataset, DALI, and PyTorch. We find that standard configurations of FFCV to outperform WebDataset either both when loading from RAM (left) and in network filesystem scenarios (right). We plot throughput of loading from the dataset (through a single epoch of loading) compared with number of cores utilized.

E. Video Dataset Loading

FFCV is also a high performance way to load video datasets. We investigate the use of FFCV when loading UCF101 [Soomro et al.(2012)], a standard video dataset of humans performing actions.

Setup. We consider clips of 50 frames per second, and implement a new custom data field to store small video clips inside of .beton files. In this format we compress the entire clip using JPEG formatting per frame. In our experiments we use the standard setting of 90 quality for the JPEG compression. We compare with the standard method for loading these video clips: a standard torchvision loader that (a) decodes the video then (b) extracts the relevant features. We benchmark loading on both a standard network file system (NFS) and also from memory; we find that all statistics below are nearly identical as we are mostly compute bound rather than storage bound (we use 48 cores/96 threads all around).

Benchmark results. We compare FFCV with the standard loader over four properties:

1. **Epoch loading time**: As our main concern, we measure the time required to iterate through the entire dataset (this is the same as throughput). It takes FFCV 24 seconds to iterate through the dataset, while it takes the torchvision dataset 19 minutes and 36 seconds.

2. **Loader startup time**: FFCV takes less than a second to instantiate the loader; on the other hand the torchvision loader requires 5 minutes and 2 seconds.

3. **Loader writing time**: FFCV takes 22 minutes to make, roughly the amount of time required to iterate through the torchvision dataset (which is called under the hood to generate the dataset).

4. **Storage size**: FFCV takes up 44GB versus 7GB for the original torchvision loader. This is because the torchvision loader stores the videos using a dedicated video codec, instead of using the JPEG compression.

We find that FFCV outperforms the standard torchvision loader across every axis except time to write the dataset (as the latter has no dataset to write) and storage size. FFCV is therefore the loader of choice for video datasets as long as storage is not at a high premium, and as long as the dataset will be iterated over more than once (the second pass will already save the time required to write the dataset).

F. Omitted Figures

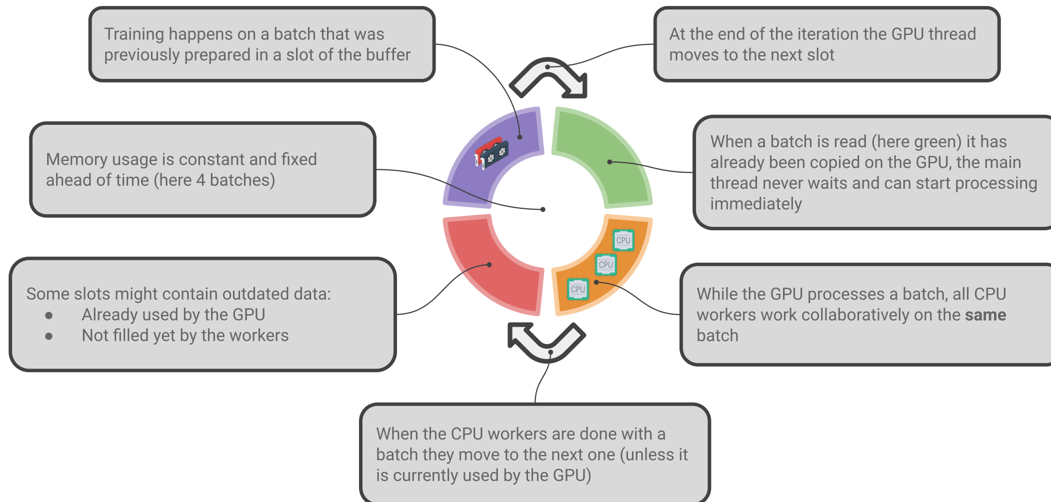

Figure 8. Illustration of `FFCV`'s circular buffer. The producer (data processing pipeline) works on an entry of the buffer while the consumer (user's code) uses a previously filled one. When done, they moving clockwise and pause when they encounter each other.