Towards Fast Crash-Consistent Cluster Checkpointing

Andrew Wood
Boston University
aewood@bu.edu

Moshik Hershcovitch
IBM Research
moshikh@il.ibm.com

Ilias Ennmouri
IBM
ilias.ennmouri@ibm.com

Weiyu Zong
Boston University
willzong@bu.edu

Saurav Chennuri
Boston University
saurav07@bu.edu

Sarel Cohen
The Academic College of Tel Aviv-Yaffo &
Hasso Plattner Institute, Germany
sarelco@mta.ac.il

Swaminathan Sundararaman
IBM Research
swami@ibm.com

Daniel Waddington
IBM Research
daniel.waddington@ibm.com

Peter Chin
Dartmouth College
pc@dartmouth.edu

***Abstract*—Machine Learning models are expensive to train: they require expensive high-compute hardware and have long training times. Therefore, models are extra sensitive to program faults or unexpected system crashes, which can erase hours if not days worth of work. While there are plenty of strategies designed to mitigate the risk of unexpected system downtime, the most popular strategy in machine learning is called checkpointing: periodically saving the state of the model to persistent storage. Checkpointing is an effective strategy, however, it requires carefully balancing two operations: how often a checkpoint is made (the checkpointing *schedule*), and the cost of creating a checkpoint itself.**

In this paper, we leverage Python Memory Manager (PyMM), which provides Python support for Persistent Memory and emerging Persistent Memory technology (Optane DC) to accelerate the checkpointing operation while maintaining crash consistency. We first show that when checkpointing models, PyMM with persistent memory can save from minutes to days of checkpointing runtime. We then further optimize the checkpointing operation with PyMM and demonstrate our approach with the KMeans and Gaussian Mixture Model algorithms on two real-world datasets: MNIST and MusicNet. Through evaluation, we show that these two algorithms achieve a checkpointing speedup of a factor between 10 and 75x for KMeans and over 3x for GMM against the current state-of-the-art checkpointing approaches. We also verify that our solution recovers from crashes, while traditional approaches cannot.

I. Introduction

Machine learning programs extensively use Volatile Memory to accelerate model training by storing and updating their models in DRAM. Unfortunately, when a program faults or the machine is power cycled during training, the state of that training is irreversibly lost. Such program faults or unexpected system shutdowns can result in large financial loss as hours if not days of progress is lost and must be redone.

To safeguard the training state from faults or inadvertent restarts/shutdowns, numerous strategies have been developed. One of the most popular strategies, called *checkpointing* [1], [2], is where the model state is periodically saved to persistent storage. Upon a program fault or shutdown, the state of the training can be "rolled back" (or restarted) from the most recent model version saved. How often the program state is checkpointed (called the checkpoint *schedule*) must be balanced against the penalty for any progress lost between checkpoints, as the checkpointing operation is not instantaneous. To create a checkpoint, the training must suspend updating the model so it's state can be consistently written to storage.

Machine learning and Deep Learning models can contain billions of parameters, meaning that the checkpointing operation must often persist tens to hundreds of GBs of information at a time which is expensive. Additionally, models can train for thousands or hundreds of thousands of iterations, leading to hours if not days of time aggregated by checkpointing alone [3], [4]. Since each iteration is prohibitively expensive to calculate, data scientists usually checkpoint with a fixed period (i.e. every X number of iterations) to amortize the cost of checkpointing. This results in more work that needs to be recomputed upon a crash and is proportional to the periodicity of the checkpointing schedule.

In this paper, we advocate for the use Persistent Memory (PM), which acts as persistent storage while operating close to DRAM speeds to accelerate the checkpointing operation. By using persistent memory we eliminate the additional slowdown with traditional checkpoint on SSDs or HDDs. Since checkpointing is faster with our approach, it can occur more often and thus reduces the time to recreate the lost state during a crash. Of course, using PM alone does not guarantee crash-consistency of checkpoints on faults. Users would still need to handle crash consistency at the program level.

To checkpoint models in a crash-consistent way, we use Python Memory Management (PyMM), an open-source python library [5], [6]. PyMM automatically provides crash-consistent updates of python objects on PM. As a result, PyMM significantly simplifies application development and also eliminates any programmer errors while checkpointing models. Additionally, PyMM is already optimized to work on persistent memory (such as Optane DC). By using PyMM with PM, we observe the checkpointing process can be significantly accelerated while still providing stronger crash consistency guarantees as compared to the traditional checkpointing strategies (such as pickle, NumPy.save, and NumPy.memmap). More specifically, we evaluated our solution on KMeans and Gaussian Mixture Model algorithms on MNIST and Musicnet datasets and show 10x to 75x, and 3x checkpoint speedup, respectively.

The rest of this paper is organized as follows: We first describe the background on checkpointing strategies and persis-

tent memory. Then we describe our checkpointing solution with PyMM, provide experimental results (i.e., setup, methodology, and results) that demonstrates the benefits of our checkpointing strategy. We then discuss the related work and finally conclude.

II. Background

This section provides a brief background on checkpointing, how persistent memory works as compared to traditional memory, and how to use persistent memory to checkpoint.

A. Current Checkpointing Strategies

Checkpointing is defined as the process of writing out the model parameters and any other information (such as architecture, data iterator state, etc.) necessary to load the model and continue training. Currently, checkpointing is performed on persistent storage devices such as SSDs by writing objects (i.e., models) via files. More specifically, when a model is checkpointed, the parameters of the model are traditionally organized in matrices which are then written to disk either as separate files (one per collection of parameters), or in a compressed archive. Unfortunately, this solution is slow, synchronous, and the training process is stalled during checkpointing.

Memory mapping is an alternative, asynchronous technique to checkpointing using files [7]. In this approach, a user creates a memory map of a file on disk and updates the model in memory (just like updating it in DRAM). Periodically or explicitly (i.e., via a `msync()` call), the contents of the mapped memory region is persisted to the mapped device. For example, the NumPy memory mapping API provides functionality that mimics an `ndarray` object and a user can perform the same operations on the memory mapped `ndarray` as a DRAM backed `ndarray` with little to no code modification. When a write occurs in memory mapped region, the data first makes its way from the cache to DRAM, where it is eventually flushed to the device.

Memory mapping, while more elegant than writing to a file, is also slow, volatile, and not crash consistent. The flush operation still runs at the speed of the device, which is orders of magnitude slower than DRAM. Additionally, unless the user manually invokes eager flushing, the data is still volatile until it is lazily written in the background. Therefore, memory mapped checkpointing is only faster than file checkpointing when the user relies on lazy flushing, meaning the data is not guaranteed to be persistent when the data is written with the memory mapping API.

B. Persistent Memory

Persistent Memory (PM) provides high capacity and persistence without sacrificing speed [8]–[14]. The most popular version of PM is offered by Intel called *Optane DC*) [15], [16]. The key is that the latency of the devices, while three to five times slower than DRAM, is orders of magnitude faster than the latency of existing persistent storage. Therefore, users can have devices with low latency, high throughput, and persistence. Similar to DRAM, Optane DC utilizes load/store instructions and is connected to the memory bus. Another advantage of Optane DC is its large capacity with a maximum of 512GB DIMMs. Optane DC uses slots similar to DRAM, meaning that a system can be equipped with multiple DIMMs. Currently, the maximum possible capacity for a commodity 2U server is 12TB, significantly more than DRAM.

Optane DC can be configured in a variety of ways. First, Optane DC can be configured to always persistent data, called *App Direct Mode*, or to be volatile, called *Memory Mode*. When in Memory Mode, Optane DC can only be accessed through DRAM cache misses, and essentially provides a layer between DRAM and lower layers in the memory hierarchy. In this paper, we do not make use of Memory Mode, and always configure Optane DC to be persistent. In App Direct Mode, Optane DC can further be accessed through two settings: Device Direct Access (DevDax) or Filesystem Direct Access (FS-DAX). When in FS-DAX, Optane DC can be mounted like a normal filesystem. However, when in DevDax, Optane DC can only be accessed through the kernel as an object store. While DevDAX is slightly faster than FS-DAX, DevDAX is significantly more difficult to debug, so it is often preferable to access Optane DC in FS-DAX . An additional benefit to FS-DAX is that since it is mounted with a filesystem, interfacing with FS-DAX requires no additional code changes and can be accessed by normal I/O operations.

C. Checkpointing Models on Persistent Memory

The traditional methods of checkpointing (i.e. writing to files and memory mapping) is supported on Optane DC, albeit with a significantly faster device. More specifically, as mentioned earlier, Optane DC communicates directly with the CPU cache through the memory bus and does not need data to pass through DRAM. Therefore, when data is written from the cache to Optane DC, it is written at the speed of the Optane DC device, which is orders of magnitude faster than its NVMe SSD counterpart. Optane DC can also be mounted using a filesystem via FS-DAX and files can be written to the Optane DC device without any change to I/O libraries.

III. Checkpointing Models on Persistent Memory with PyMM

Python Memory Management (PyMM) is an open source Python library [5], [6] and is functionally similar to memory mapping but on an Optane DC device II-B. PyMM can interact with Optane DC in App Direct Mode [5], [16], and it can interact with other devices that are mapped via FS-DAX, including NVMe SSDs and other devices (through emulation). PyMM internally utilizes libraries from Memory-Centric Active Storage (MCAS) [17], to handle all metadata operations. MCAS uses a key-value store to support PyMM, and uses `libpmem` to flush data to the device.

Like memory mapping, by default, PyMM flushes data lazily to the device. It also provides an API for eager flushing of data (via `persist()`). The key datatype in PyMM is a `shelf` object. A `shelf` acts as a namespace. PyMM offers native support for NumPy and PyTorch tensors, meaning that programs

```
import numpy as np
import pymm

# create the shelf
shelf = pymm.shelf("myshelf",
                   pmem_path="/mnt/pmem0",
                   shelf_size_mb=10000)

# Gaussian Mixture Model w/ 2 clusters
model = GuassianMixtureModel(k=2, ...)

# preallocate variables on the shelf
shelf.means = model.means
shelf.covs = model.covs

while not m.converged():
    model.update(training_data)

    # Inefficient checkpoint w/ eager flushing
    # shelf.means = model.means

    # Efficient checkpoint w/ in-place update
    shelf.means[:,:] = model.means
    shelf.means.persist()

    # Inefficient checkpoint w/ eager flushing
    # shelf.covs = model.covs

    # Efficient in-place update
    shelf.covs[:,:,:] = model.covs
    shelf.covs.persist()
```

Fig. 1: A sample Python program using PyMM to checkpoint a Gaussian Mixture Model with 2 clusters to a 10GB shelf named "myshelf" on the Optane DC device mounted at `/mnt/pmem0`. After each `.persist()` call, the shelf variable is guaranteed to be persisted on the device.

can assign `ndarray` and `tensor` values to the shelf as shelf variables. Coupled with eager flushing, PyMM provides an API for models to checkpoint with the convenience of memory mapping, but with the speed of Optane DC.

To provide crash-consistency guarantees, each object contains two sets of checkpointing buffers, which are used in an alternating fashion to always ensure that model updates are atomic and persistent. This requires minimal bookkeeping information, with a single integer (that is atomically updated) required to determine which buffer contains the most recent checkpoint. This integer value is only updated once the entire checkpointing operation has succeeded: meaning that in the case of unexpected power loss we would only have to redo the work that happened after the last consistent checkpoint (i.e., less than an iteration's worth of work).

An example of checkpointing a model can be seen in Figure 1. A `shelf` can contain multiple data types as different `shelf` variables within the same namespace. In contrast, with memory mapping, multiple memory mapped files are required to save data of different data types, formats, etc or the application has to be significantly changed to deal with different data types within the same mapped region. With PyMM , all of these values can be stores on a single `shelf` , minimizing the amount of code needed to checkpoint on persistent memory.

We note that the code example in Figure 1 contains two types of `shelf` assignment operations. The commented out lines are *direct* shelf variable assignments. Direct assignment is slow when the `shelf` variable already exists. When a shelf variable already exists, PyMM does not have the luxury of evaluating the contents of the new value to determine if the old value's memory on the device can be recycled. Therefore, when assigning to an already existing `shelf` variable, PyMM must first delete the old shelf variable, and then re-create it with the new value. When the data is the same size and in the same format, like when training a model, this process is wasteful. A more efficient strategy is to assign in-place to the shelf variable, through the use of library supported APIs. Specifically, the most generic is to assign in-place to a `ndarray` or `tensor` through the `:` operator (i.e. `X[:,:] = Y`). However, each API has its own variation: NumPy uses the `numpy.copyto(dest, src))` function, while PyTorch uses the `tensor.copy_(other)` method.

IV. Experiments

This section describes our evaluation methodology, experimental setup, and results. The goal of our evaluation is to assess the performance benefits of using PM for checkpointing. We would like to answer the following questions through our evaluation:

- Q_1 How does checkpointing on Optane DC (without PyMM) compare to current checkpointing strategies?
- Q_2 How does checkpointing with PyMM compare to current checkpointing strategies (on NVMe SSD and on Optane DC)?
- Q_3 How does checkpoinging with PyMM hold up on real-world (i.e, large datasets and models) use cases?

The rationale behind Q_1 and Q_2 is to validate the need and usefulness of persistent memory and PyMM for checkpointing, respectively. Q_3 helps in understanding the benefits of using PyMM on popular real-world datasets.

A. Methodology

To answer Q_1 and Q_2, we chose to use multiple storage media and multiple checkpointing strategies. To determine a baseline, we evaluated checkpointing on traditional storage media (i.e. a gen-4 NVMe SSD) and chose the fastest solution. We then considered Optane DC on its own and mounted it as a filesystem in FS-DAX. We then re-ran our traditional checkpointing experiments where the files were written to the Optane DC device instead of the NVMe SSD. Finally, we used PyMM to checkpoint the same data to Optane DC both in FS-DAX and DevDax. In each of these experiments, we varied the size of the data being written from 1GB to 150GB. We repeated each experiment at least five times to reduce the chances of randomness interfering with our results.

Additionally, we also ran Q_1 for a large number of iterations to understand the cumulative benefits of PyMM on PM for long training jobs. This scenario models iterative algorithms where the model (i.e. the parameters) is checkpointed after

Strategy	Optane DC Configs Tested	NVMe SSD Tested?	traditional?	crash consistent?
pickle	FS-DAX	Yes	Yes	No
NumPy.save	FS-DAX	Yes	Yes	No
NumPy.memmap	FS-DAX	Yes	Yes	No
PyMM	FS-DAX, DevDax	No	No	Yes

TABLE I: Checkpointing strategy and the media combinations we evaluated in Q_1, Q_2 and Q_3. Note that for strategies which write to files, Optane DC must be in FS-DAX.

being updated during each iteration. This scenario more closely resembles that of clustering algorithms and programs which must protect against faults in general.

To answer Q_3, we evaluated two popular clustering algorithms using NumPy and PyTorch. When choosing the clustering algorithms, we used the following criteria:

1) They must be popular.
2) They must be vectorizable (i.e., provide meaningful comparison across checkpointing strategies on a variety of media types).
3) Together they must cover the runtime complexity and program state size lattice.
4) Together they must cover CPU and GPU based implementations.

Using this criteria, we chose the KMeans [18]–[21] and Gausian Mixture Model (GMM) [18], [19], [21], [22] algorithms. Both algorithms solve the *k-clustering* problem: to find k representatives from a dataset that optimize some error function. The KMeans algorithm models a cluster as a centroid, and seeks to find the center of each centroid that minimizes the cost (i.e. l_2 distance) of assigning each point in the dataset to its closest cluster. The KMeans algorithm receives its name from the way it computes the centroid centers: once each data point is assigned to its closest cluster, the centroids are recomputed as the mean of the points assigned to that centroid. KMeans is popular because it is extremely fast and efficient, so we implemented it with NumPy and it is CPU bound. Our GMM implementation, on the other hand, is implemented on the GPU with PyTorch. GMMs take a more sophisticated approach to modeling a cluster. In a GMM, the probability of a point coming from a cluster is no longer binary; instead the probability follows a Gaussian distribution. The goal of the GMM is then to learn the Gaussian distributions that maximize the likelihood of generating the data.

Both algorithms take k, the number of clusters, as a hyperparameter; allowing us to control the size of the model state. More importantly, both algorithms are batch vectorizable, meaning that we can provide an apples to apples comparison between PyMM and traditional APIs (DRAM NumPy and PyTorch vs PyMM NumPy and PyMM PyTorch). When using a DRAM implementation, we shared the compute phase of each training iteration: we trained a single model and checkpointed that model multiple times after each iteration. We carefully recorded the timings of each checkpointing operation as well as the time in the compute phase. Therefore, we can construct the iteration/epoch time for a model using a particular checkpointing strategy by adding the runtime for the compute phase of the epoch and the runtime of saving the model using that checkpointing strategy. For DRAM implementations, we evaluated all of the checkpointing API/media combinations from Q_1, Q_2 and Q_3 except for memory mapping and `pickle`. We additionally added compressed saving (i.e. `numpy.savez_compressed(...)`) as an API method. We note that our GMM models checkpoint directly from GPU memory to PM. We chose to implement direct GPU memory to PM checkpointing after running a exploratory benchmark. In this benchmark, we observed that GPU memory to PM, while 20% slower than GPU memory to DRAM, is twice as fast than GPU memory to DRAM, then DRAM to PM.

When training our models, we considered two datasets of differing size and dimensionality. We chose the MNIST handwritten image dataset because of its immense popularity. Second, we took the MusicNet dataset, specifically the preprocessed version from Yu *et al.* [23]. We chose this dataset because it has a large number of dimensions and a large number of samples, meaning that the program state when training our models will be large. We believe that MusicNet creates a more realistic setting in our benchmarks, as clustering is often performed on massive datasets.

V. Experimental Setup

Our experiments were run on a server equipped with dual Intel Xeon Gold 6248 CPUs providing a total of 80 cores running at base frequency of 2.5GHz, 348GB of DDR4 DRAM, and 1.5TB of Optane DC. The server also has two Tesla M60 GPUs and two gen 4 NVMe m.2 drives. Each CPU socket is given a single Tesla M60 and a single NVMe drive.

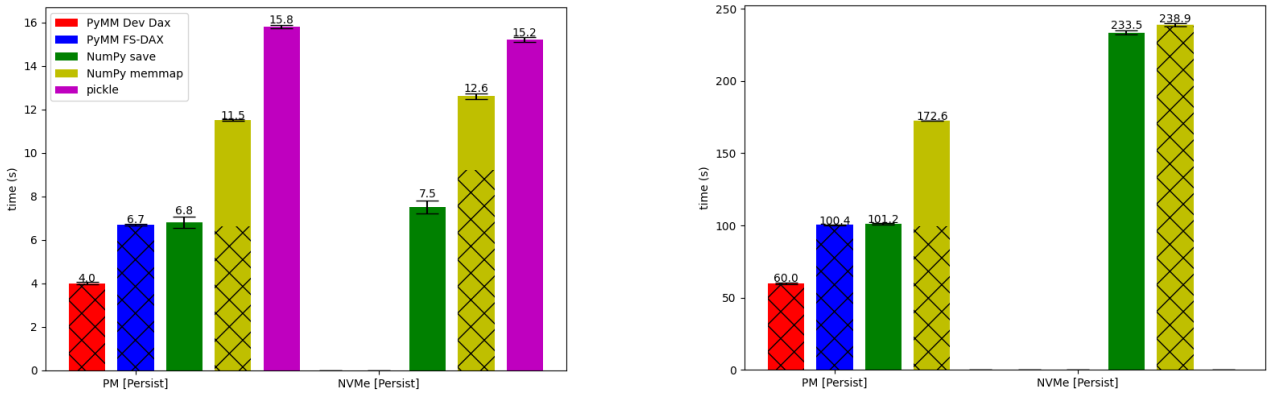
In our experiments, we train each model from scratch to account for random initialization. We note that for GMMs, random initialization does not affect the number of samples generated, as the data has enough features to where the Gaussians are spread out enough to be virtually nonexistent.

VI. Results and Discussions

In this section we present and describe the results of our experiments.

A. Checkpointing Fixed-Size Models (Q_1 and Q_2)

The results for our experiments on Q_1 and Q_2 can be found in Figure 2. The API method `numpy.save` serves as our benchmark as it is the fastest amongst the traditional checkpointing strategies. We observe that PyMM in DevDAX is always the fastest solution regardless of the size of the data. Interestingly, we observe that PyMM in FS-DAX is just as fast as our benchmark when making single writes, and also just as good as the first stage of memory mapping. When writing to



(a) 10GB

(b) 150GB

Fig. 2: Runtime (seconds) for checkpointing data from DRAM to different storage media. The x axis describes the device being written to, while the y axis shows the runtime of each checkpoint strategy writing to that device. We note that meshed areas correspond to the first step of checkpointing, where data is written to volatile memory. The rest of the bar (unmeshed) shows the remaining runtime for flushing the data to the device. We also note that `pickle` crashes when writing 150GB models due to a lack of memory capacity.

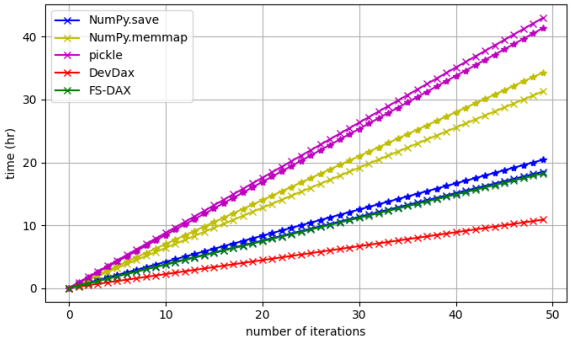


Fig. 3: Runtime (hours) for repeatedly checkpointing 10GB of data from DRAM to different storage media. The x axis describes the number of repetitions, while the y axis shows the runtime for writing the data repeatedly. 'x' markers are used for writing to Optane DC while '*' markers are used for writing to NVMe storage.

Optane DC, step 1 of FS-DAX takes almost the entire total runtime, suggesting that most of the work occurs in writing to the cache rather than flushing from the cache to the device. When writing to an NVMe device, PyMM is not available, and the superior option is our baseline of `numpy.save`. Interestingly, according to our results, if DevDAX is not available, saving to a NVMe device with `numpy.save` is a comparable solution to saving on Optane DC and FS-DAX with PyMM or `numpy.save`.

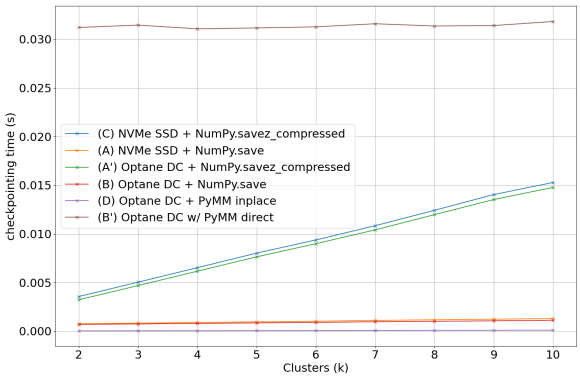
Therefore, we answer Q_1 and Q_2 with if Optane DC is present in the system, then PyMM with DevDAX is the fastest solution.

B. Overall Checkpointing Benefits during Training

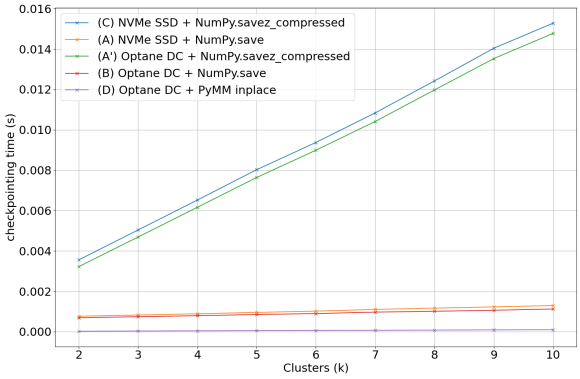
As seen in Figure 3, even small differences between runtimes can accumulate when the checkpointing occurs hundreds, if not thousands of times. Even with a small model size of 1GB, when training until convergence, using PyMM with DevDAX can result in saving over 400 seconds (about 7 minutes) from the next fastest solution (`numpy.save` on an NVMe SSD drive) if the model is checkpointed at least 2000 times. When writing to NVMe storage is more moderately sized ($\sim$10GB), the time saved between PyMM in DevDAX and the next closest solution (PyMM in FS-DAX) becomes over 5,800 seconds (over 1.6 hours). Therefore, when running iterative algorithms such as clustering, which repeatedly checkpoints the model being trained, the impact of PyMM on the overall checkpointing time is significant. We omit 1GB, 100GB, and 150GB from Figure 3 since the trend of the results does not change.

C. Checkpointing on Real-World Datasets (Q_3)

To answer Q_3, we first look at our KMeans implementation, the results of which can be seen in Figure 4. As mentioned in Section IV-A, direct assignment becomes expensive when PyMM must first delete the old value of the shelf variable and then write the new value: this is wasteful if the size and format of the data is unchanged. To implement in-place assignment, shelf variables are allocated once at the beginning of training and are never allocated again, allowing for a one-time upfront cost that writing to files (or direct assignment) must pay every iteration. We can clearly see from Figure 4(b) that after removing PyMM direct assignment, that PyMM in-place performs the best and has near constant runtime regardless of the number of clusters (and therefore the size of the data being written). This is supported from Q_1 and Q_2, where assignment on either PyMM DevDAX or PyMM FS-DAX was



(a) KMeans on Musicnet with all checkpointing operations



(b) KMeans on Musicnet removing PyMM direct assignment.

Fig. 4: Average checkpointing runtimes (seconds) from training KMeans to convergence on the Musicnet dataset.

almost entirely occupied by the time for allocation of memory from the device. From Figure 4(b) we can see that saving to an NVMe device is not that much worse than writing to Optane DC as files. However, we note that this solution is vastly inferior to PyMM in FS-DAX.

Turning to GMM, we can see the results of our experiments in Figure 5. First, we note that in this implementation, we accidentally found two ways of performing an in-place shelf assignment. This arose naturally from the implementation, which uses PyTorch to put the computation on a GPU. The first does an in-place operation using assignment (i.e. `shelf.X[:,:] = new_value`), while the second explicitly uses `torch.copy_(other)` to perform the assignment (i.e. `shelf.X.copy_(new_value)`). These operations have the potential to differ in two places: the shallow-copy tensor allocation used for indexing into a tensor (i.e. `shelf.X[:,:]` produces a shallow-copy of `shelf.X`), and the underlying implementation for the equality operator. From our results, we conclude that the runtime for creating the shallow copy is negligable, and the underlying implementation for the equality operator is the `.copy_(other)` method. Therefore, in our results, we observed no difference between GMM's inplace and direct assignment strategies.

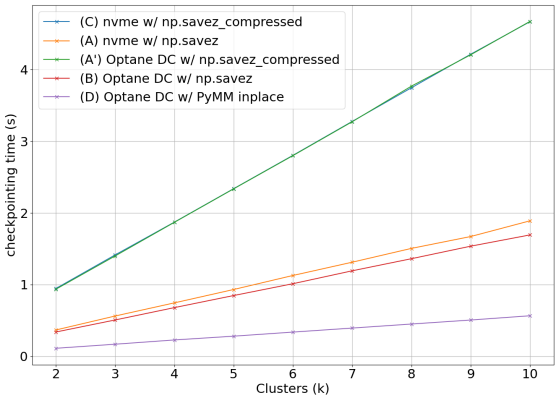


Fig. 5: Average checkpointing runtimes (seconds) from training GMM to convergence on the Musicnet datasets

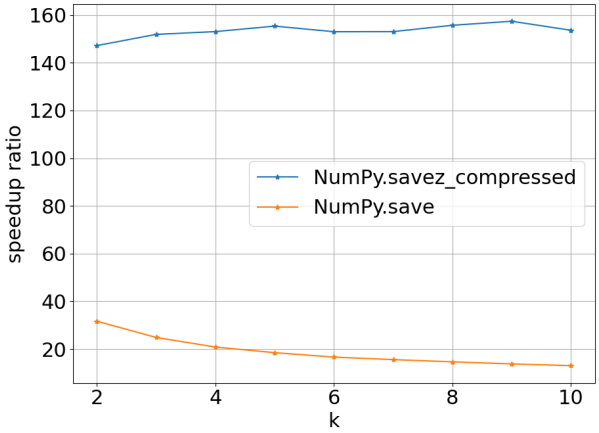


Fig. 6: Average speedup ratios of using PyMM inplace vs checkpointing to NVMe SSD for KMeans on the Musicnet dataset.

In Figure 5, we observe a larger linear coefficient for checkpointing than in Figure 4. This is because GMM allocates significantly more memory than KMeans. KMeans allocates one vector per cluster ($O(dk)$) memory where d is the dimensionality of the data and k is the number of clusters, while GMM allocates one mean vector and one covariance matrix per Gaussian ($O(dk + d^2k)$). We observe that, like with KMeans, PyMM is the fastest option regardless of the number of clusters (i.e. data size). Our results also match the results of KMeans, with compression resulting in larger runtimes (although smaller files). We observe a looser relationship between checkpointing to NVMe and Optane DC as files, although this is likely due to the slower speeds of the NVMe device.

Figure 6 shows the speedup ratios from the results of Figures 4 and 5. Speedup ratios are calculated as the ratio

Strategy	Configuration	One Ckpt. File		Two Ckpt. Files	
		Pass	Failed	Pass	Failed
pickle	FS-DAX	0	100	100	0
NumPy.save	FS-DAX	0	100	100	0
NumPy.memmap	FS-DAX	0	100	100	0
PyMM	FS-DAX	100	0	100	0
pyMM	DevDax	100	0	100	0

TABLE II: Checkpointing Strategy and the media combinations evaluated for crash consistency with 100 random crashes.

of an NVMe API's times to the in-place PyMM times. This figure shows the magnitude of difference between using an NVMe device and PyMM. While KMeans speedup trends are decreasing, we note that the the rate of decrease is also slowing, suggesting that they will stabilize above a speedup of 1x.

D. Crash-Consistent Checkpointing

We also evaluated the crash-consistent properties of the checkpointing strategies by injecting 100 random crashes during the checkpointing process. The results are described in Table II. Pickle and NumPy.save do not provide crash-consistency by default. More specifically, the file update during checkpointing (on persistent media) is not atomic. As a result, a fault or a crash during checkpointing results in partially updated file (i.e., an unusable checkpoint). To avoid this scenario, users have to explicitly write to different files and recover using the last complete checkpoint. NumPy.memmap cannot provide crash-consistent checkpoints as updates are directly to the mapped region in memory. It is important to note that, though the NumPy.memmap checkpoints can be remapped in memory after a crash, but it would not be a consistent (or up to date) checkpoint as it would contain updates from the previous checkpoint intermixed with updates from the current checkpoint. In contrast, updates via PyMM, can provide crash-consistent checkpoints as PyMM internally maintains two persistent buffers for each shelf object and alternates writing to these buffers to create a consistent checkpoint on every update.

VII. Related Work

There has been a gamut of work on checkpointing and reducing the checkpointing overhead [24]–[30]. To the best of our knowledge, we are the first to propose and demonstrate a checkpointing solution using persistent memory (PM) and a easy to use python library (PyMM), evaluate and quantify the overheads of different access modes in persistent memory, and provide guidance on which access modes to use to get the optimal performance.

Recent work such as CheckFreq [31], Check-N-Run [32], and DeepFreeze [33] have proposed solutions to reduce the overall checkpointing time. CheckFreq converts the synchronous checkpointing operation to a two phase asynchronous operation but the solution is still limited by the bandwidth and performance characteristics of the underlying NVMe SSDs [31]. Check-N-Run focuses on optimizing the overheads of distributed checkpointing of Deep Learning Recommendation Models via differential checkpointing (by writing a small portion of the model during checkpointing) [32]. DeepFreeze on the other hand, assume deep knowledge about the execution graph and does asynchronous sharding and copying of model parameters to speed up checkpointing [33].

Our solution also shares the same goals of reducing the overall checkpointing time. But at the same time, our solution is orthogonal to these efforts and can be leveraged by these solutions to further reduce their overall checkpointing time.

VIII. Conclusion

In conclusion, we have shown that it is possible to provide fast crash-consistent checkpointing with the usage of PyMM and Optane DC. We first show that when performing a checkpoint, PyMM in DevDAX is always the fastest operation, and by repeating this operation over multiple iterations, models can save minutes to hours to days of checkpointing time. We then further optimize the checkpointing operation and show that algorithms such as KMeans and GMM can receive checkpointing speedups of a factor between 10 and 75x for KMeans and over 3x for GMM, respectively. We have also simulated random crashes during checkpointing and show that PyMM provides superior crash-consistency guarantees compared to the other traditional checkpointing strategies.

References

[1] A. C. Palaniswamy and P. A. Wilsey, "An analytical comparison of periodic checkpointing and incremental state saving," *ACM SIGSIM Simulation Digest*, vol. 23, no. 1, pp. 127–134, 1993.

[2] A. N. Tantawi and M. Ruschitzka, "Performance analysis of checkpointing strategies," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 2, pp. 123–144, 1984.

[3] E. Rojas, A. N. Kahira, E. Meneses, L. B. Gomez, and R. M. Badia, "A study of checkpointing in large scale training of deep neural networks," *arXiv preprint arXiv:2012.00825*, 2020.

[4] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta, "Performance implications of periodic checkpointing on large-scale cluster systems," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 8–pp.

[5] D. Waddington, M. Hershcovitch, and C. Dickey, "Pymm: Heterogeneous memory programming for python data science," in *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, 2021, pp. 31–37.

[6] "Python memory management," https://github.com/IBM/pymm, 2022.

[7] A. J. Smith, "A comparative study of set associative memory mapping algorithms and their use for cache and main memory," *IEEE Transactions on Software Engineering*, no. 2, pp. 121–130, 1978.

[8] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[9] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla *et al.*, "Phase change memory technology," *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.

[10] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 24–33.

[11] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[12] T. Kawahara, K. Ito, R. Takemura, and H. Ohno, "Spin-transfer torque ram technology: Review and prospect," *Microelectronics Reliability*, vol. 52, no. 4, pp. 613–627, 2012.

[13] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang *et al.*, "Advances and future prospects of spin-transfer torque random access memory," *IEEE Transactions on Magnetics*, vol. 46, no. 6, pp. 1873–1878, 2010.

[14] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[15] T. Mason, T. D. Doudali, M. Seltzer, and A. Gavrilovska, "Unexpected performance of intel® optane™ dc persistent memory," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 55–58, 2020.

[16] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[17] D. Waddington, C. Dickey, M. Hershcovitch, and S. Seshadri, "An architecture for memory centric active storage (mcas)," *arXiv preprint arXiv:2103.00007*, 2021.

[18] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[19] J. MacQueen, "Classification and analysis of multivariate observations," in *5th Berkeley Symp. Math. Statist. Probability*, 1967, pp. 281–297.

[20] P. E. Hart, D. G. Stork, and R. O. Duda, *Pattern classification*. Wiley Hoboken, 2000.

[21] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[22] G. J. McLachlan and K. E. Basford, *Mixture models: Inference and applications to clustering*. M. Dekker New York, 1988, vol. 38.

[23] R. Yu, A. Wood, S. Cohen, M. Hershcovitch, D. Waddington, and P. Chin, "Biologically plausible complex-valued neural networks and model optimization," 2022.

[24] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "{Heterogeneity-Aware} cluster scheduling policies for deep learning workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 481–498.

[25] F. Shahzad, M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, "An evaluation of different i/o techniques for checkpoint/restart," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1708–1716.

[26] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 25–36.

[27] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello, "Optimization of multi-level checkpoint model for large scale hpc applications," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1181–1190.

[28] Z. Lan and Y. Li, "Adaptive fault management of parallel applications for high-performance computing," *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1647–1660, 2008.

[29] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[30] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, 2014, pp. 571–582.

[31] J. Mohan, A. Phanishayee, and V. Chidambaram, "CheckFreq: Frequent, Fine-Grained DNN checkpointing," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 2021, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/mohan

[32] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram, "Check-N-Run: a checkpointing system for training deep learning recommendation models," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, 2022, pp. 929–943. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/eisenman

[33] B. Nicolae, J. Li, J. M. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*. IEEE, 2020, pp. 172–181. [Online]. Available: https://doi.org/10.1109/CCGrid49817.2020.00-76