Large-Scale AI Engineering: PCCheck Integration

George Manos • Luca Renna

Check out PCCheck »

Introduction

Training large-scale machine learning models requires both a lot of resources utilized for several weeks. As the scale of such deployments increases, the probability of failures also increases along with it. In our case, on the Alps cluster, there could be disruptions due to machine failures, server maintenance which takes place weekly, or even time limitations that are assigned to each quota. Therefore, having a checkpointing mechanism to pick-up the model training from where it stopped is essential in order to safely finalize a training process.

Standard checkpointing mechanisms, such as the one presented below, usually pause the training process and checkpoint the parameters to persistent storage. This, however, induces large overhead, especially when applied in high frequency, yet frequent checkpoints are essential to avoid long recovery times (i.e. the time since the process restarts until it reaches the state it was when the partition took place).

Implementations

In this project we implement the intra-node version of the checkpointing mechanism outlined in [1], and compare it against a naive baseline checkpointing implementation that uses `torch.save()`. However, related work [2] has shown that `torch.save()` isn't enough to make sure that your changes have been persisted to disk and need to run `fsync()` right after.
While we adapted PCCheck such that it works on a single node, the original system supports checkpointing mechanism for distributed training, and can be adapted properly in the future. But, we will show below the performance benefits of using PCCheck over naive checkpointing only on a single node.

In both cases, the user can use the `--load-checkpoint` flag to load the latest persisted checkpoint and resume the training process.

You can find our GitHub repo here: https://github.com/WatchmeDoc/LSAI-Checkpointing.git

Baseline

We implement the baseline using a standard pytorch.save, where we checkpoint the current training step, the model state, optimizer state and the learning rate scheduler state. We also checkpoint the dataloader state by storing the random states of the relevant libraries (python, numpy, torch, cuda), as well as the step number (`train_step`) we are currently at.

To load the checkpoint, we perform a standard pytorch.load to load the aforementioned states. Then, when we need to restore the dataloader state, we iterate `train_step` times through it without any actions.

The complete code implementation can be found under `/training/train_checkp.py` . To run it, the user can use:

```
python -m training.train_checkp.py --seed 4 --training-steps 151 --checkpoint-freq 50 -
```

PCcheck Implementation

The above naive checkpointing mechanism induces high overhead in DNN training jobs, especially when applied in high frequency. PCCheck [1] is a framework for DNN training that supports multiple concurrent checkpoints in parallel. Concurrent checkpoints can help reduce idle GPU time and increase checkpoint frequency, since training does not have to wait for the previous checkpoint to finish, before initiating a new one. However, naively issuing concurrent checkpoints can increase CPU memory and storage overheads, as well as PCIe and storage bandwidth contention, which could degrade training throughput.

We use PCCheck to checkpoint model and optimizer state, whereas the learning rate scheduler and the random states are persisted to disk through regular `pickle` serialization. Although PCCheck is fault-tolerant, this persisting isn't, but can be easily made by using 2 separate files to write on and select the proper one when loading.

How it works

In short, PCCheck uses a separate GPU buffer that keeps track of the optimizer and model states. When a checkpoint is issued, the model state and optimizer state dictionaries are copied to DRAM using the copy mechanism described on their paper, which in turn is asynchronously persisted to disk while the training loop proceeds unaffected. In order to further reduce stalls, PCCheck supports pipelining by splitting the data into multiple chunks and using multiple threads. The system also supports overlapping checkpoints (e.g. issuing a new checkpoint before the previous one has completed) through its `max_async` parameter, as well as out of the box support for failures during checkpointing, meaning that if the machine crashes during checkpoint, the last persisted checkpoint will be unaffected and the training process can resume from it.

The reader may refer to the original paper for further implementation details, as well as guidance towards how to configure this system (number of threads, checkpoint frequency etc) in Section 3.4.

To run the training process with PCCheck, the user can use:

```
python -m training.train_pccheck --seed 4 --training-steps 151 --checkpoint-freq 50 --m
```

Configurations

In this section, we list some possible configurations for PCCheck checkpointing mechanism.

The important command line arguments we added for PCCheck are the following:

- `max_async` , default value = 1. Maximum allowed asynchronous checkpoints at a time. For example, if `max_async=1` and another checkpoint tries to take place before the previous one has completed, the process will block
- `non_blocking` , we also added a non-blocking copy functionality within the transfer from one buffer to the other. Generally, copying on GPU invokes cudaMemcpyAsync, and if `non_blocking` is set to `False` (default), a call to `cudaStreamSynchronize` will be called after each `cudaMemcpyAsync` [3]. In our system, we provide the option to set `non_blocking` to `True` to parallelize the `state_dict` serialization and do a `cucaStreamSynchronize` at the end.
- `checkpoint-freq` , checkpoint frequency. Should be tuned according to the probability of failures [1]. Higher frequency means larger overhead to the training loop, lower frequency means more time to fully recover after failure.

We also added a `pccheck_config.json` in `checkpointing/pccheck` directory that allows to configure how pccheck works. The important parameters there are the following:

- `basic_file` : filepath to the checkpoint file
- `num_threads` : number of threads used for pipelining the copying process. Higher number of threads generally means each thread handles less workload but increases the probability of stragglers. Also, they are bottlenecked by the disk I/O and thus one gets diminishing benefits after 2 or 4 threads.
- `save_memory` : Skips the use of some auxiliary arrays, which in turn may save some DRAM memory but can slow things down when using multiple threads. No effect whatsoever for 1 thread.

Issues

While attempting to make the system run on the Alps cluster, we encountered several issues that we had to work on, as the system was built and tested on x86 machines, while GH200 nodes run on Arm64 architecture.

First of all, just like any academic research paper, PCCheck codebase included various baselines, as well as support for other disk types such as Intel Optane Memory, features which interrupted our installation process as we didn't have root privileges on the cluster to install all the required tools. Thus, we had to figure out which parts were actually needed and which weren't to strip down the codebase as much as possible such that we can install it on our machines.

Secondly, as the system uses C++ on the background and some very low level primitives, such as barriers in assembly instructions, we had to make several changes and translate the x86 instructions into Arm64. We also had to profile the GH200 node thoroughly to find how it allocates mmap regions in order to fix the configured file offsets that PCCheck uses to find the data it stores.

Third, as the system was tested on much smaller models and machines, we faced several integer/float overflows in the C++ level, issues which required thorough profiling as they didn't just pop. However, this also presented an opportunity; given that 1 GH200 node holds a whooping 288 CPU cores, the system could further be optimized to take advantage of those massive computational resources.

The above issue brings us to the next issue we faced, regarding multithreading. In x86 machines, using multiple threads brought significant speedup on PCCheck, but this was not the case in Arm64, as it induced some undefined behavior (e.g. large increased overhead, straggling threads, but at some times it would indeed bring the desired speedup). To overcome this, we profiled the CPU usage of each core using `htop` and found that on the first few checkpoints, where the `mmap` took place for the first time, there was a significant delay for multiple threads, while the processor that the main process was bound on would get congested. Thus, we made the initialization of PCCheck slower by

pre-allocating the memory within the `mmap` region, which mitigated the aforementioned effect and afterwards we saw a 1.8x performance boost in checkpointing.

Finally, to add support for other data types, as C++ only holds support for fp32 and fp64, we set the buffer to always be on fp32 and just convert the data transferred to it into fp32. This could further be optimized as C++ treats the buffer as bytes and thus the representation won't shouldn't be an issue. However, we still need to change some pre-computed offsets on the background such that the file is loaded then properly.

Results

Experiments Setup

We modified on the provided codebase, implementing our new features and changing the model configuration such that its dimension matches the sequence length. As described above, PCCheck uses an auxiliary GPU buffer of the same size as the model + optimizer parameters and thus the model couldn't fit with sequence length of 4096 in a single GPU. Thus, we compare with Transformer dimension = sequence length with values of 1024 and 2048, as well as FP32 and BF16.

Note that FP16 didn't work out of the box in the current training loop, as there were some exploding values (i.e. norm computation would crash due to non-finite values).

To test correctness, we launch two runs using a random seed (e.g. 4), with 150 training steps and checkpoints at 50-steps intervals. We then launch a third run with the same parameters but with a failure happening after step 50. Finally, we recover from the checkpoint and we resume training. If everything is set correctly, the 3 plotted lines should overlap 100%.

Validation

Applying the aforementioned experiment for both `checkp` (baseline that uses `torch.save()`) and `pccheck`, we get the following plots.

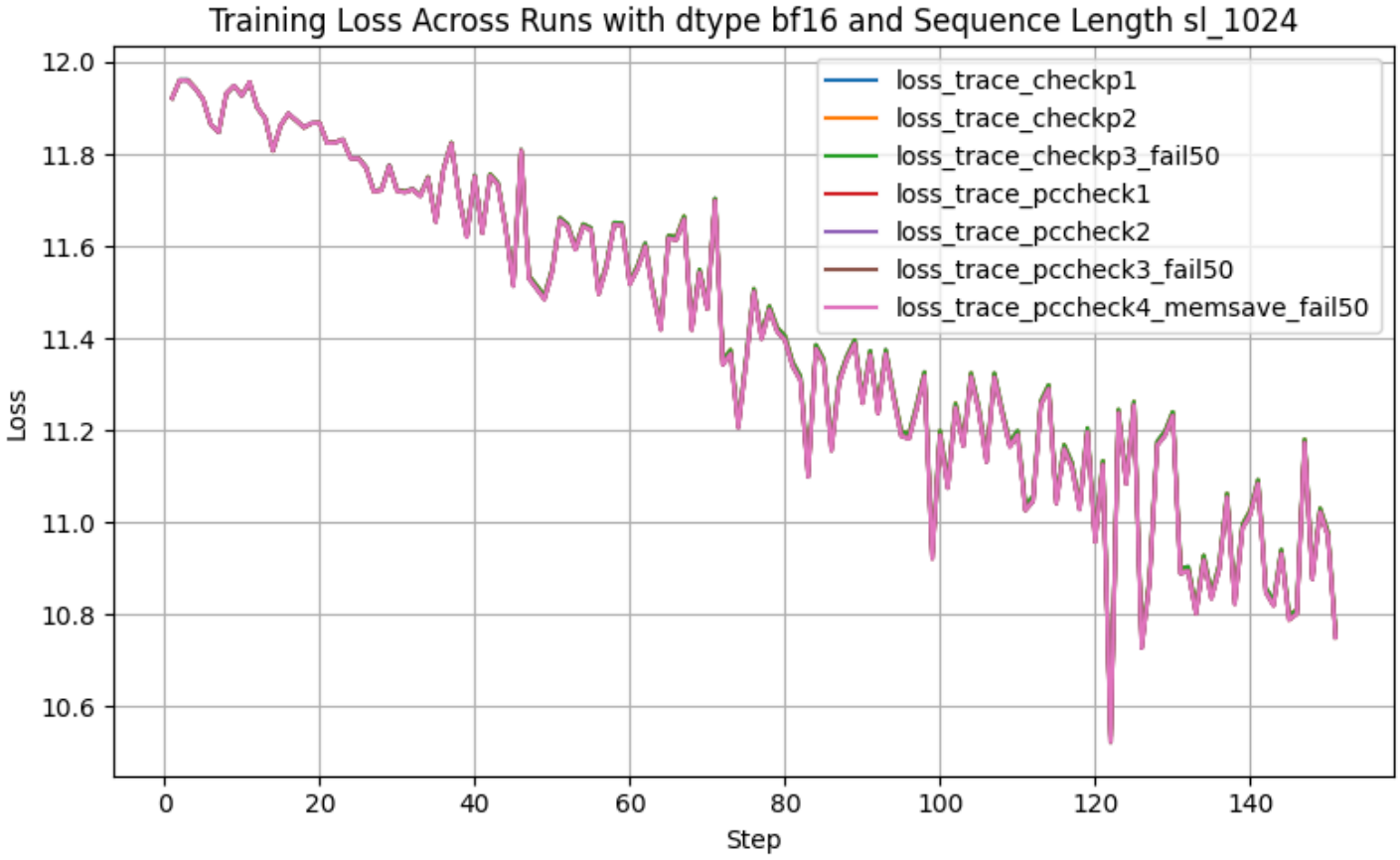

Figure 1: Loss trace for dtype bf16 and Sequence Length 1024. Here, we also confirm that `mem_save` set to `True` works.

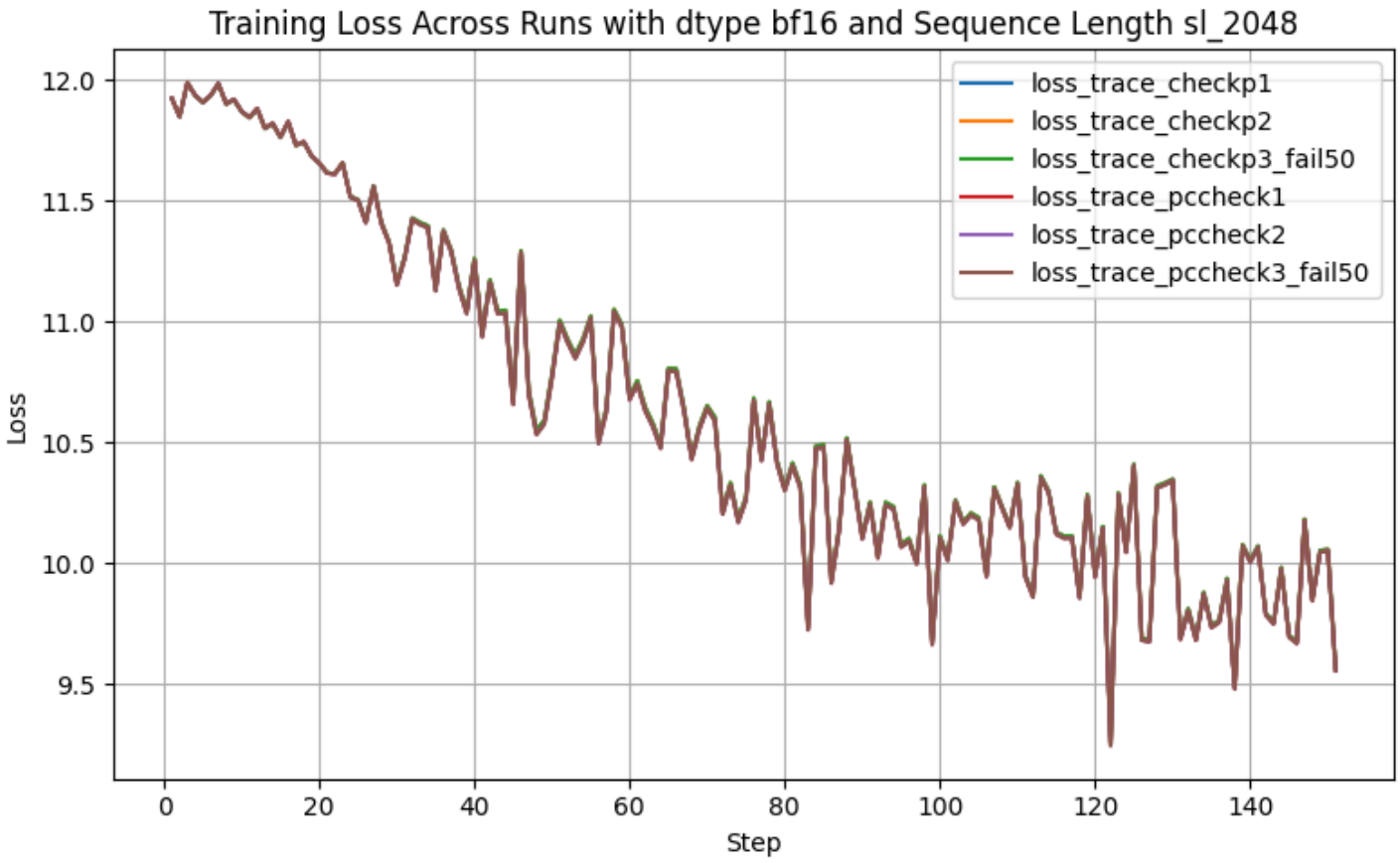

Figure 2: Loss trace for dtype bf16 and Sequence Length 2048.

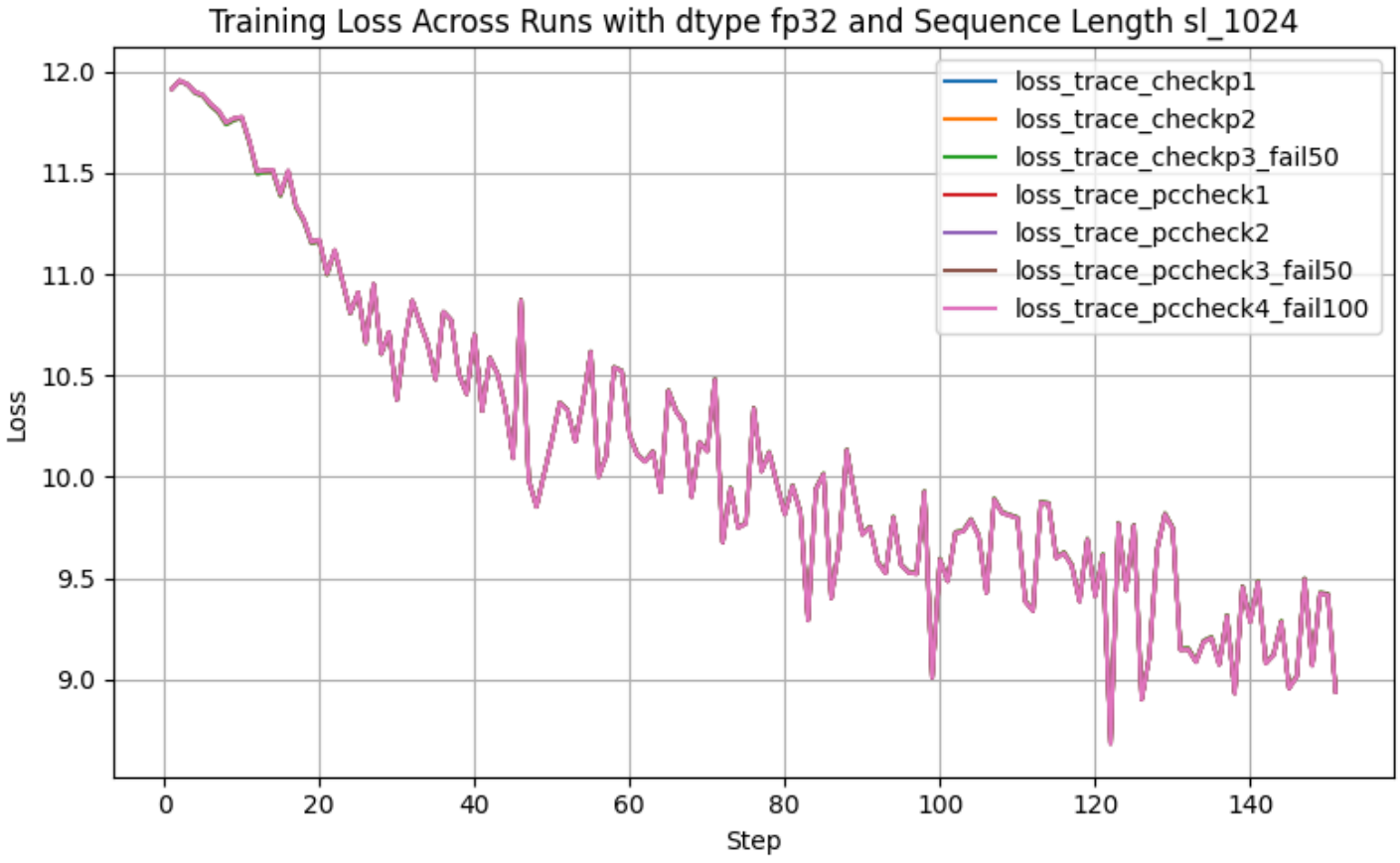

Figure 3: Loss trace for dtype fp32 and Sequence Length 1024.

Figure 4: Loss trace for dtype fp32 and Sequence Length 2048. In this large scenario we also validate failing at 100th step (i.e. 2nd checkpoint), as well as using non_blocking copies and multiple threads.

From the figures above and the log files we include in our submission, we verify that the losses are exactly the same for a given seed, resulting in a successful recovery after failure. We also verified the loading of `pccheck` by performing in depth dictionary/tensor comparison of the model and optimizer state dictionaries against `torch.save()` and `torch.load()`. Finally, we confirm that the various parameters of PCCheck do not affect its correctness. The reason we made 3 runs in both PCCheck and checkp, 6 in total instead of 4, is that we use different files for each one, and had to confirm that both function as expected.

Performance

We also benchmarked the performance of PCCheck against the baseline. Here, we made 3 separate runs of each configuration and computed the average run time. In Fig. 5, we report the average time per checkpoint from the time the function is called until it has persisted to disk. However, as the checkpoint takes place asynchronously, to fully show-case the benefits of PCCheck we also have to

report the end-to-end training time for the same number of steps, which we do in Fig. 6, where we made a full training run for 800 steps, with checkpointing frequency of 40 steps. Nevertheless, it is also important to minimize the time it takes to persist a checkpoint to the disk, as it reduces the probability of a failure taking place during checkpointing.

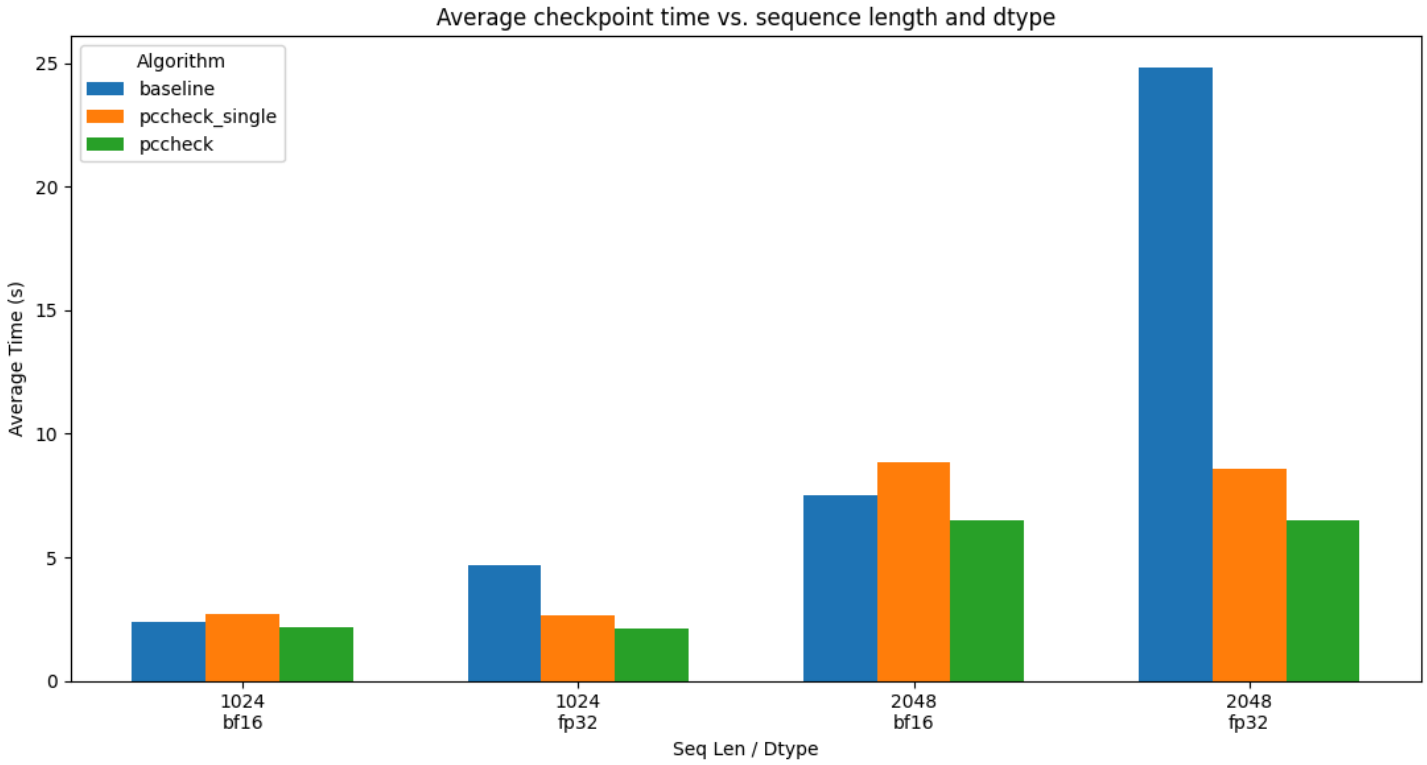

Figure 5: Average time per checkpoint, from the time it is invoked until it has persisted to disk.

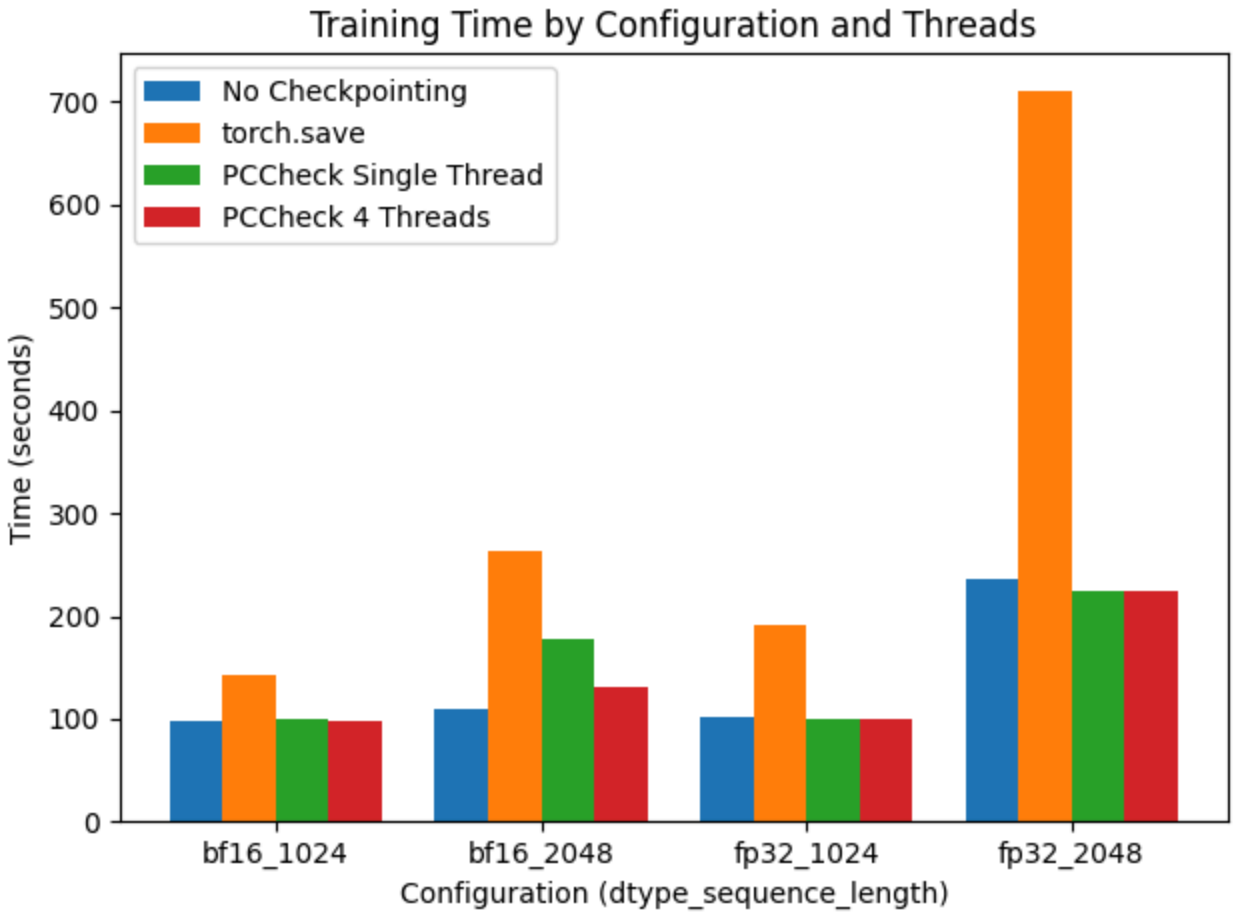

Figure 6: Total training runtimes for different configurations, from the beginning of the training loop until the end. Averaged out over 3 runs.

Our experiments showed that indeed we get a performance boost when using 4 threads in all cases, as the training time reduces by up to 3.2x and the average checkpointing time by a factor of 3.9x when using PCCheck instead of `torch.save`

From Fig.5, we can see that although the average checkpointing time of the baseline is faster on `bf16`, due to the model serialization and the type conversion overhead, it is much slower in both cases of `fp32`, especially on sequence-length of 2048 where PCCheck is 3.7x faster with 4 threads. We can also see that the checkpointing time for PCCheck doesn't change between `bf16` and `fp32` as it converts the first to the latter, which in turn induces a little overhead and has to handle the same amount of data.

From Fig. 6, we can see the benefits of using PCCheck more clearly. Now that we measure the full training loop and as PCCheck works asynchronously, the overhead added to the basic training loop with 4 threads is negligible in 3 out of 4 cases, and on the 4th one, which is bf16/sl 2048, the overhead is just 20% more time. In all cases however, PCCheck significantly outperforms `torch.save`.

Note that in this section, we don't report the time it takes to load from a checkpoint as they are approximately similar.

Future Work

There are still several things one may work on to extend this feature. The first and most important one is to profile deeper the use of multiple threads and CPU affinity, in order to leverage the large amount of CPU cores that are available in a single node.

The next important step is to also work on making the full checkpointing mechanism fault tolerant, meaning that if a failure occurs during a checkpoint, it will be able to recover afterwards.

Next, we also propose the further optimization of buffering to explicitly support the copying of fp16, bf16 and fp8 data types more efficiently, as this can help support larger model sizes and better showcase the performance boost of PCCheck vs `torch.save()`

Last but not least, adapting the distributed checkpointing library is also a significant step as using multiple GPUs is essential for large model training.

Acknowledgements

We express our gratitude to doctoral candidate Foteini Strati for her assistance to overcoming the aforementioned issues when adapting PCCheck to the new hardware. This project wouldn't be possible without her help, but it was also an excellent opportunity to dive deep into the Alps cluster system, explore some of its capabilities and limitations.

We also responsibly declare that we used help from Generative AI tools, including ChatGPT and GitHub Copilot.

References

[1] Foteini Strati, Michal Friedman, and Ana Klimovic. 2025. PCcheck: Persistent Concurrent Checkpointing for ML. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 811–827. https://doi.org/10.1145/3669940.3707255

[2] Mohan, J., Phanishayee, A., & Chidambaram, V. (2021). CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In M. K. Aguilera & G. Yadgar (Eds.), 19th USENIX Conference on File and Storage Technologies (FAST '21), 203–216. USENIX Association.

[3] PyTorch Docs, A guide on good usage of `non_blocking` and `pin_memory()` in PyTorch, https://docs.pytorch.org/tutorials/intermediate/pinmem_nonblock.html