Distributed Neural Networks

Ethan Harness

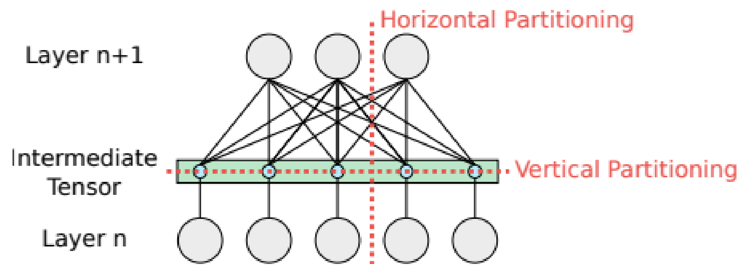
Overview

The need for parallel training

- Modern AI uses massive models with trillions of parameters.
- Training large models like this using standard programming methods is inefficient.
- As a result, frameworks and libraries have been developed to enable end users to train these large models across many machines.
- These tools allow users to more easily distribute their model onto their computing infrastructure for training and inference to meet their needs.

Model Parallelism

- In model parallelism, the model is split onto multiple machines, so that no single machine holds the entire model.
- These splits can be done either vertically or horizontally.
  - Horizontally splitting a model is done layer by layer such that all individual layers are housed on a single machine.
  - Vertically splitting a model is done between layers and are shared amongst the computing infrastructure.



https://doi.org/10.48550/arXiv.2007.03970

Model Parallelism

- To inference or train the model, data must be communicated between the individual splits.
- It can reduce the memory footprint of the model itself since the entire model isn’t housed on one machine.
- However, since communication must be done on every sample for every layer result, the overall overhead makes this a type of parallelism slow and as a result it isn’t widely used.

Data Parallelism

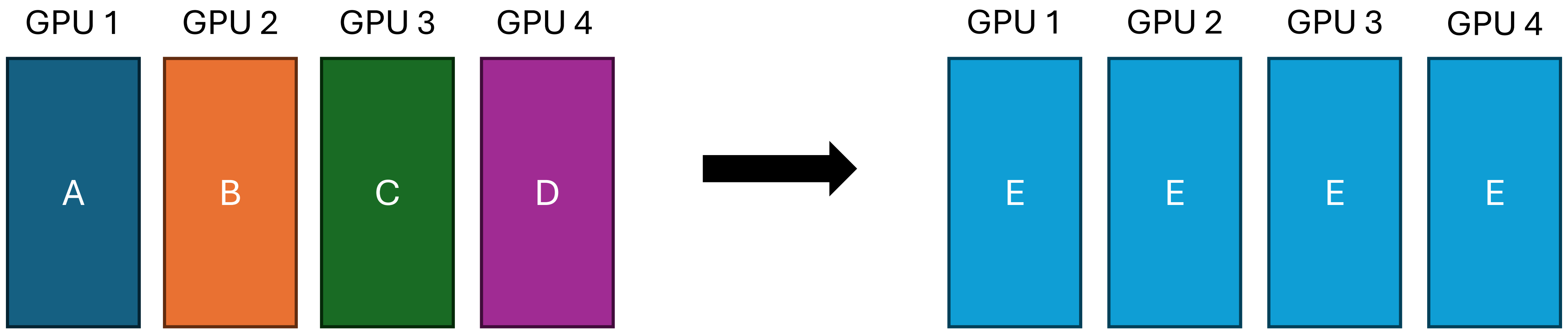
- While model parallelism splits parts of the model onto multiple machines, data parallelism houses the entire model on many machines.
- Instead, batches of the training data are given to each machine which processes them independently.
- This approach also uses communication to synchronize gradients and intermediate results.
- This approach increases throughput while minimizing overall communication time, so it is more widely used.
- With large models, it suffers from the memory overhead needed to store the entire model across the computing infrastructure.

Communication Patterns

- Underpinning all distributed training methods are the various ways data is communicated between machines.
- These techniques are central to ensuring training is occurring efficiently.
- These operations are implemented in a variety of frameworks and the distributed training libraries we will look at support a number of these frameworks. Some of the most common are…
  - NVIDIA’s NCCL
  - Facebook’s Gloo
  - MPI

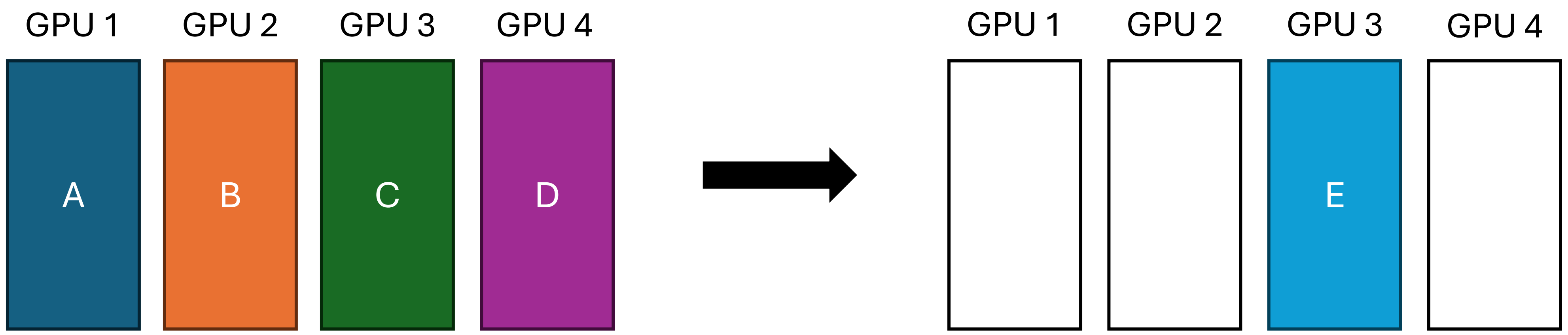
AllReduce

- In AllReduce, data across the GPU’s get combined and some operation is performed to those data to produce a new dataset.
- For example, E may be an array where E[i] = A[i] + B[i] + C[i] + D[i]
- This is one of the most important operations in distributed training



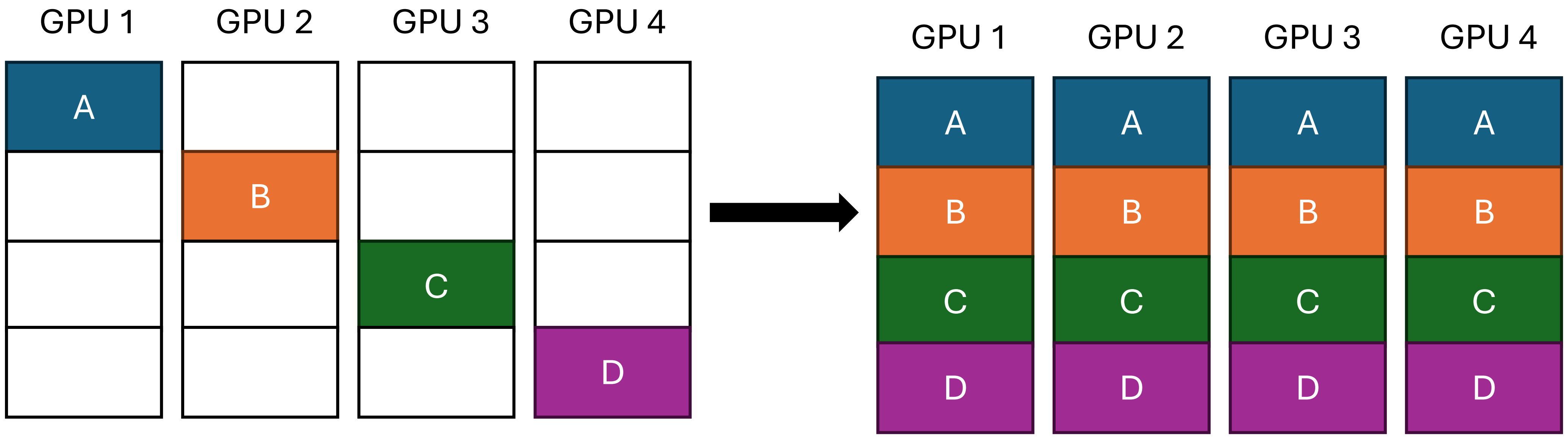
Reduce

- AllReduce except only 1 GPU receives the result.



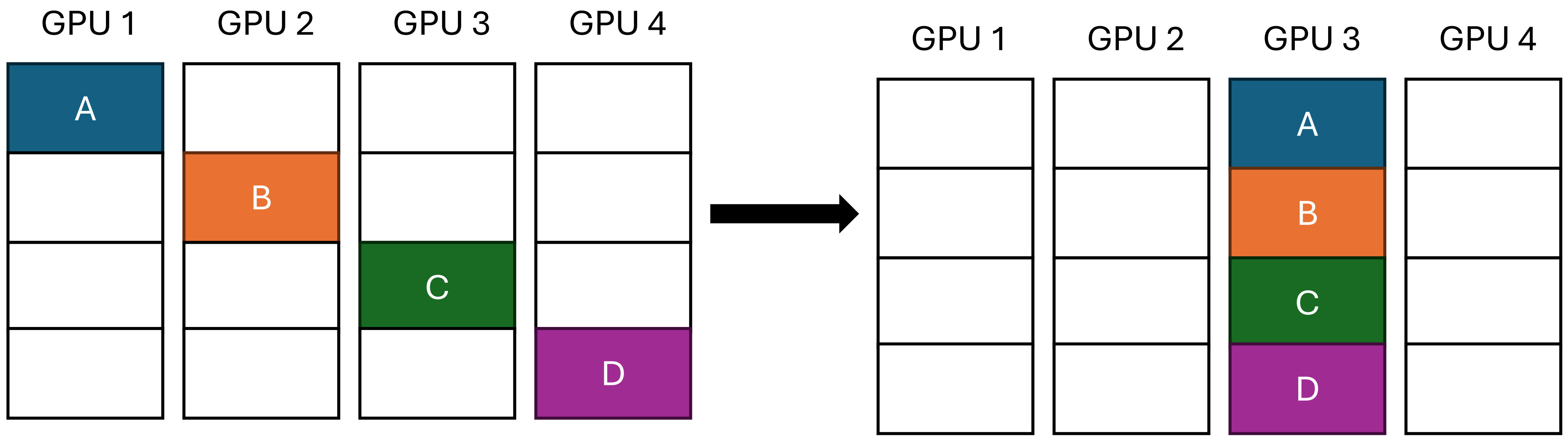
AllGather

- Partitioned data is collected and synchronized between every GPU.



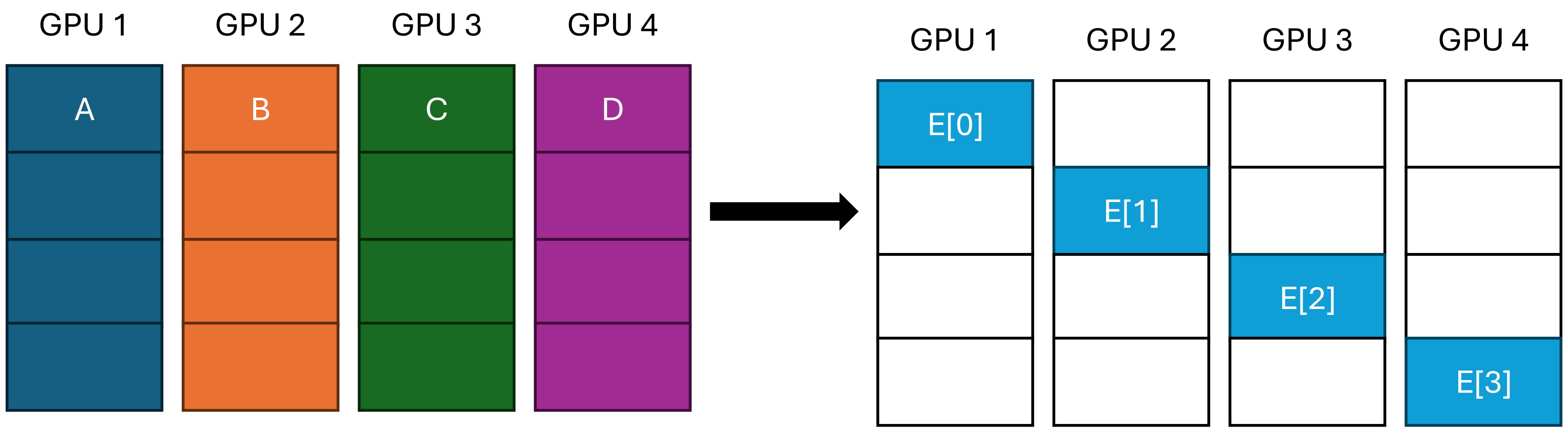
Gather

- Data is gathered across GPUs onto one GPU. Like AllGather but with only 1 destination.



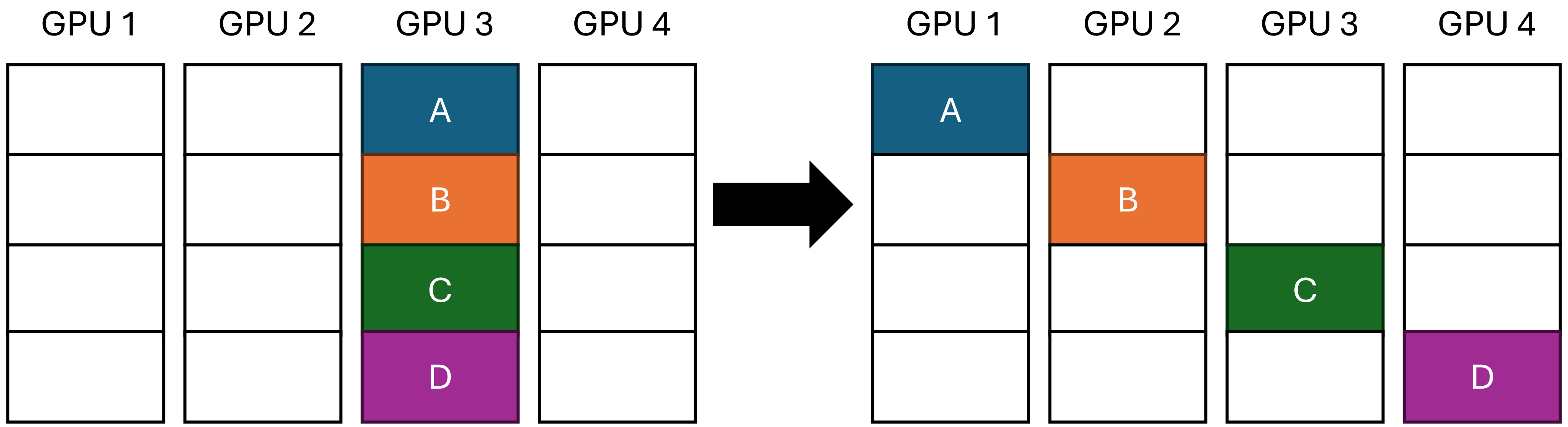
ReduceScatter

- Data is reduced, but only part of it is stored on each GPU.



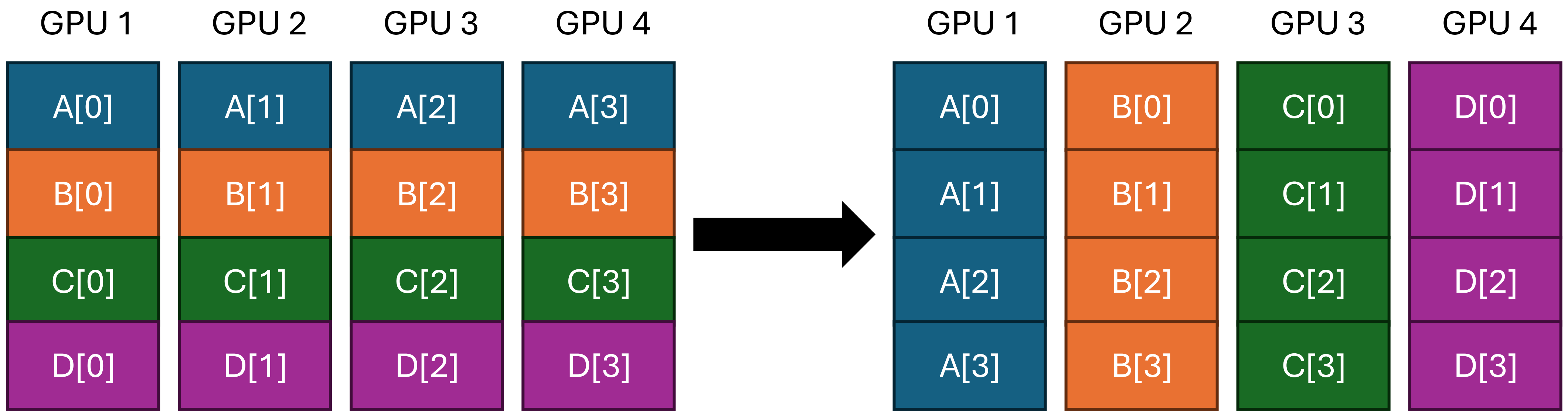
Scatter

- The reverse of a gather. Takes data from 1 GPU and partitions it across many GPU’s.



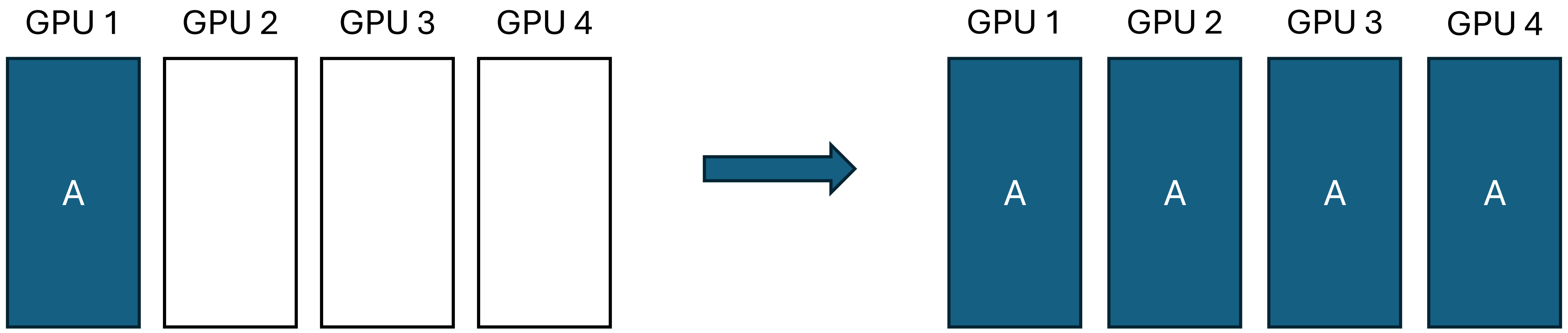
AllToAll

- Data is partitioned into chunks and that chunk is sent to its corresponding GPU.



Broadcast

- Data within 1 GPU gets transmitted to all other GPU’s.



PyTorch DDP

PyTorch DDP Introduction

- Training ML models with PyTorch has 3 main steps
  - Forward Pass which computes cost based on model architecture and input
  - Backward Pass which computes the gradients
  - Optimizer step which updates model
- Parallelizing this process can be done simply
  - Replicate model across many threads/processes and portion training data accordingly
  - Compute forward and backward passes independently
  - Synchronize results and update
  - Repeat until model is trained
- More efficient parallelization techniques though require more granular control of communication.
- PyTorch DDP handles this communication in a way that involves little work to go from a single threaded application to distributed application.

PyTorch DDP Background

- Multiple techniques are within PyTorch to parallelize training
  - PyTorch Data Parallel offers a single process multi-threaded solution to train on multiple GPUs in parallel but is limited to a single machine.
  - PyTorch Distributed Data Parallel offers multi-process solution to train onmultiple GPUs across multiple machines.
  - PyTorch RPC handles more advanced distributed training scenarios.
- Multiple ways to facilitate distributed training
  - Can train many models in parallel, then average model parameters together. This limits communication overhead but is not mathematically equivalent to single threaded training.
  - Can synchronize gradients between replicas. This has more communication overhead but is equivalent to single threaded training. This is also the approach that PyTorch DDP uses.

PyTorch DDP Background Continued

- Communication done through AllReduce.
- This procedure can be implemented in a few ways to speed up naïve implementations
  - Ring-based
  - Tree-based
- AllReduce is a synchronous procedure since every process needs to communicate a tensor for the later operation to be performed.

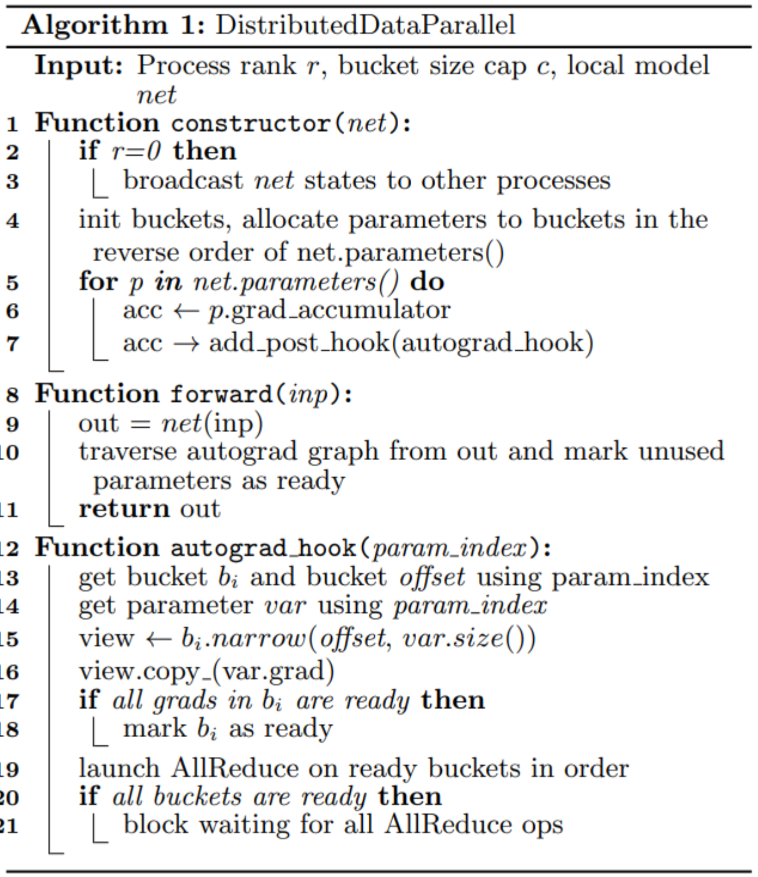
PyTorch DDP System Design

- PyTorch DDP starts by replicating model across all processes.
- During backward pass, we synchronize gradients before optimizer.
- Afterwards, we can run the optimizer knowing all models are synchronized before and after.
- DDP uses auto grad hooks to enable communication without any end user modifications. When triggered it fires AllReduce to synchronize.
- DDP also uses gradient bucketing. This strategy allows for overlap between gradient computation and gradient synchronization.

PyTorch DDP System Design Continued

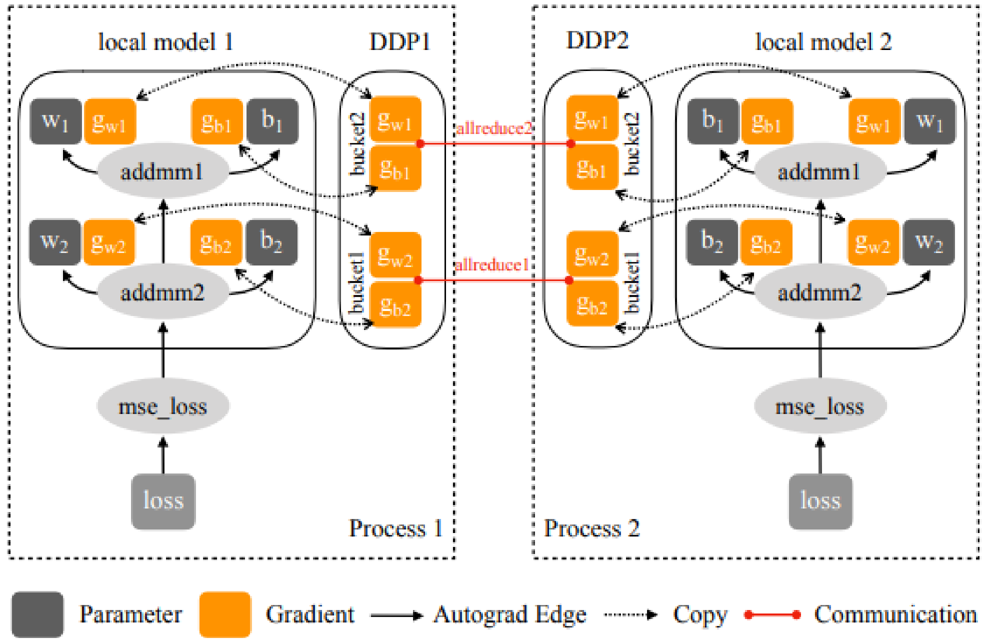
- Gradient bucketing splits model parameters into buckets.
- Every bucket has a corresponding set of gradients that get computed in the backward pass.
- When buckets finish across models, they can synchronize immediately. This allows gradient synchronization of 1 bucket to overlap with the gradient computations in other buckets
- To ensure accuracy bucket order is the same across all processes. (DDP just splits by using the reverse order of the model parameters)
- DDP also has to mark gradients not participating as ready during the end of the forward pass.

PyTorch DDP System Design Continued



```
Algorithm 1: DistributedDataParallel
Input: Process rank $r$, bucket size cap $c$, local model $net$
Function constructor($net$):
    if $r$=0 then
        broadcast $net$ states to other processes
    init buckets, allocate parameters to buckets in the reverse order of net.parameters()
    for $p$ in $net.parameters()$ do
        acc ← $p$.grad_accumulator
        acc → add_post_hook(autograd_hook)
Function forward($inp$):
    out = $net$(inp)
    traverse autograd graph from out and mark unused parameters as ready
    return out
Function autograd_hook($param\_index$):
    get bucket $b_i$ and bucket $offset$ using param_index
    get parameter $var$ using $param\_index$
    view ← $b_i.narrow(offset, var.size())$
    view.copy_(var.grad)
    if $all$ $grads$ $in$ $b_i$ $are$ $ready$ then
        mark $b_i$ as ready
    launch AllReduce on ready buckets in order
    if $all$ $buckets$ $are$ $ready$ then
        block waiting for all AllReduce ops
```

PyTorch DDP pseudocode



Distributed gradient reduction

PyTorch DDP System Design Continued

- PyTorch supports NCCL, Gloo, and MPI as its communication backends.
- PyTorch wraps these in its ProcessGroup API.
- Users can choose which communication backend to use by modifying the process_group argument in the DDP constructor.

PyTorch DDP Implementation

- DDP contains a few arguments that can affect how training is done
  - process_group enables the user to specify the communication backend
  - bucket_cap_mb enables a user to control the bucket size
  - find_unused_parameters specifies if DDP should attempt to find unused parameters at the end of the forward pass
- If a device spans multiple devices, this can also affect DDP behavior.
- DDP uses model buffers when a layer needs to running information. DDP broadcasts buffer values from process of rank 0 to all other processes before forward pass.

DeepSpeed

DeepSpeed

- DeepSpeed is a lightweight wrapper for PyTorch.
- Provides an easy way to use state of the art training techniques like activation checkpointing, mixed precision training, distributed training, and more so users can focus more on constructing individual models.
- Shown to be 3.75 times faster than NVIDIA Megatron when training GPT2.

Deep Speed Memory Efficiency

- Provides memory efficient data parallelism and enables training of larger models on the order of trillions of parameters.
- Can train larger models on a single GPU when compared to PyTorch DDP.
- This is thanks to a training technique called ZeRO (Zero Redundancy Optimizer) which partitions the model across devices.

ZeRO

- As models have continued to grow, parameter counts have also grown. Originally models were on the scale of millions. For example, a Resnet50 model has by default 25 million parameters in PyTorch. GPT-2 has 1.5 billion. Now they are on the scale of trillions of parameters.
- As a result, fitting entire models onto a single device became unfeasible.
- Model parallelism was a way to alleviate these issues. This still requires a substantial amount of communication between devices though. Also causes inefficiencies since individual computations are a lot smaller.
- Model parallelism is also worse than data parallelism in terms of speed, but data parallelism normally requires the entire model to be on a device.
- ZeRO was developed to enable normal data parallelism with the benefits of reduced memory sizes like in model parallelism.

ZeRO and normal model memory scaling

- Most memory for an ML model comes from the model states.
  - Model states refers to the model parameters, the gradients, and the optimizer states
  - If you assume 16-bit floating point values, both the parameter size and the gradient size take up twice the number of parameters. For GPT2 with 1.5B parameters, requires 6 GB for parameters and gradients.
  - Optimizer states, memory used by a model's optimizer, also require a large amount of memory. For the ADAM optimizer, it requires 2 sets of states for the momentum and variance of gradients. In total model states can be 8x parameter count.
  - In traditional mixed precision training, initial model states and grads are in 16-bit precision with optimizer states in 32-bit precision plus an additional 32-bit copy of the parameters. In total optimizer states can go from 4x to 12x using mixed precision training.
  - Suddenly a 1.5 billion parameter model requires 24 GB of memory. As a result, scaling beyond the order of a few billion parameters isn't feasible this way.

ZeRO and normal model memory scaling

- Residual memory can also eat up memory
    - Residual memory are all other forms of memory used in training that are not in the model states.
    - Activations, which are the intermediate layer results, can also eat up memory. These are needed in the backwards pass. For the GPT-2 example, you can easily need 60 GB for activations alone. Through activation checkpointing which saves a portion of the activations then recomputes others as needed, you can get to about a size of 8 GB for activations with a 33% computation overhead to recompute needed activations.
    - Temporary buffers for communication in distributed data parallelism environments can also eat up memory. Some libraries fuse all gradients together in one All-Reduce and enlarge the gradients from 16-bit to 32-bit. For GPT-2 you can use 6 GB for these buffers.
    - Memory fragmentation also causes issues. Especially for large models where you allocate and free lots of tensors, you can easily cause situations where you run out of memory even when lots of memory is still available. In the paper, it found situations where OOM exceptions would occur with over 30% still available.
- As a result of all of these, scaling to trillions of parameters isn't feasible using traditional means.

The ZeRO solution

- ZeRO solves memory problems by partitioning the model states on multiple devices and uses communication strategies to get the required states as needed (ZeRO-DP). Also includes optimizations for minimizing residual memory (ZeRO-R).
- Memory optimizations in ZeRO-DP result in a linear drop in memory consumption with devices. ZeRO-DP also implements this in a way that only sees a 1.5x increase in communication volume at worst with no increase at best.

ZeRO-DP

- ZeRO-DP has 3 partitioning schemes. Optimizer state partitioning, optimizer state plus gradient partitioning, and optimizer state plus gradient plus parameter partitioning.
- Optimizer state partitioning splits optimizer states among each device. Every device only needs to update its optimizer states on its device. After every training step, we need to synchronize devices through an all-gather. For N devices, this partitioning reduces optimizer state memory by N on every device.
- Gradient Partitioning splits gradients onto devices in the same way optimizer state partitioning splits optimizer states. Since processes only update corresponding parameters, each device only needs reduced gradients corresponding to its parameters. This does require a reduce scatter. To improve this, you can use bucketization to overlap computation and communication. Achieves same memory savings as with optimizer partitioning.

ZeRO-DP

- With optimization state and gradient partitioning, you achieve a proportional reduction in corresponding memory with devices. Combined, the communication overhead is also equivalent to traditional data parallelism. This is because both the communications required operate in half the time as 1 AllReduce.
- Finally, parameter partitioning has only a subset of the parameters on each device. This again gives us a proportional memory reduction with a model's parameters as devices increase. This does require additional communication when a parameter on a device is needed for some computation.
- Since all parameters are needed for the forward and backwards passes, to avoid excessive memory being used we can pipeline communication and only receive the immediately required parameters before using them then discard them after.

ZeRO-DP Partitioning Schemes

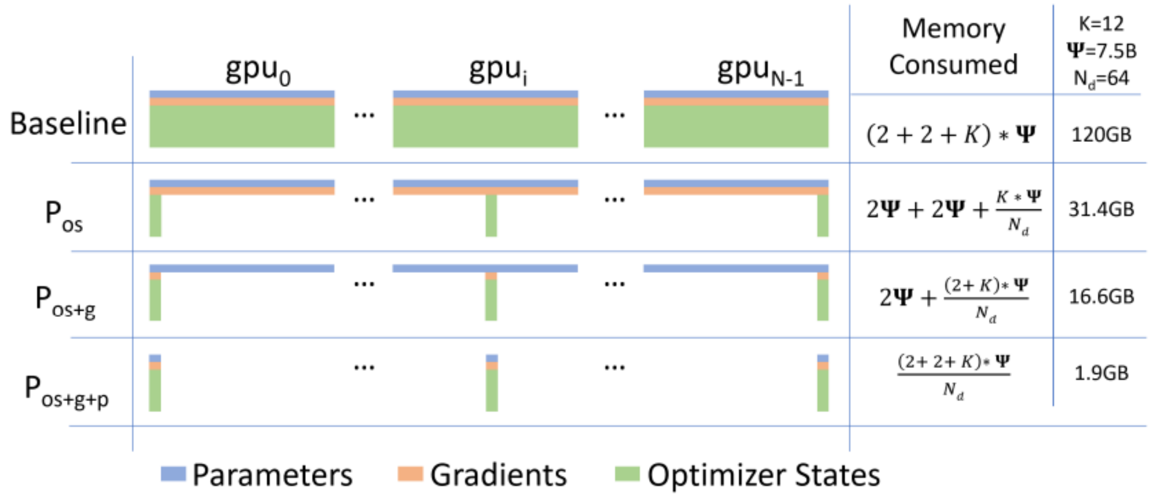


Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

DeepSpeed Continued

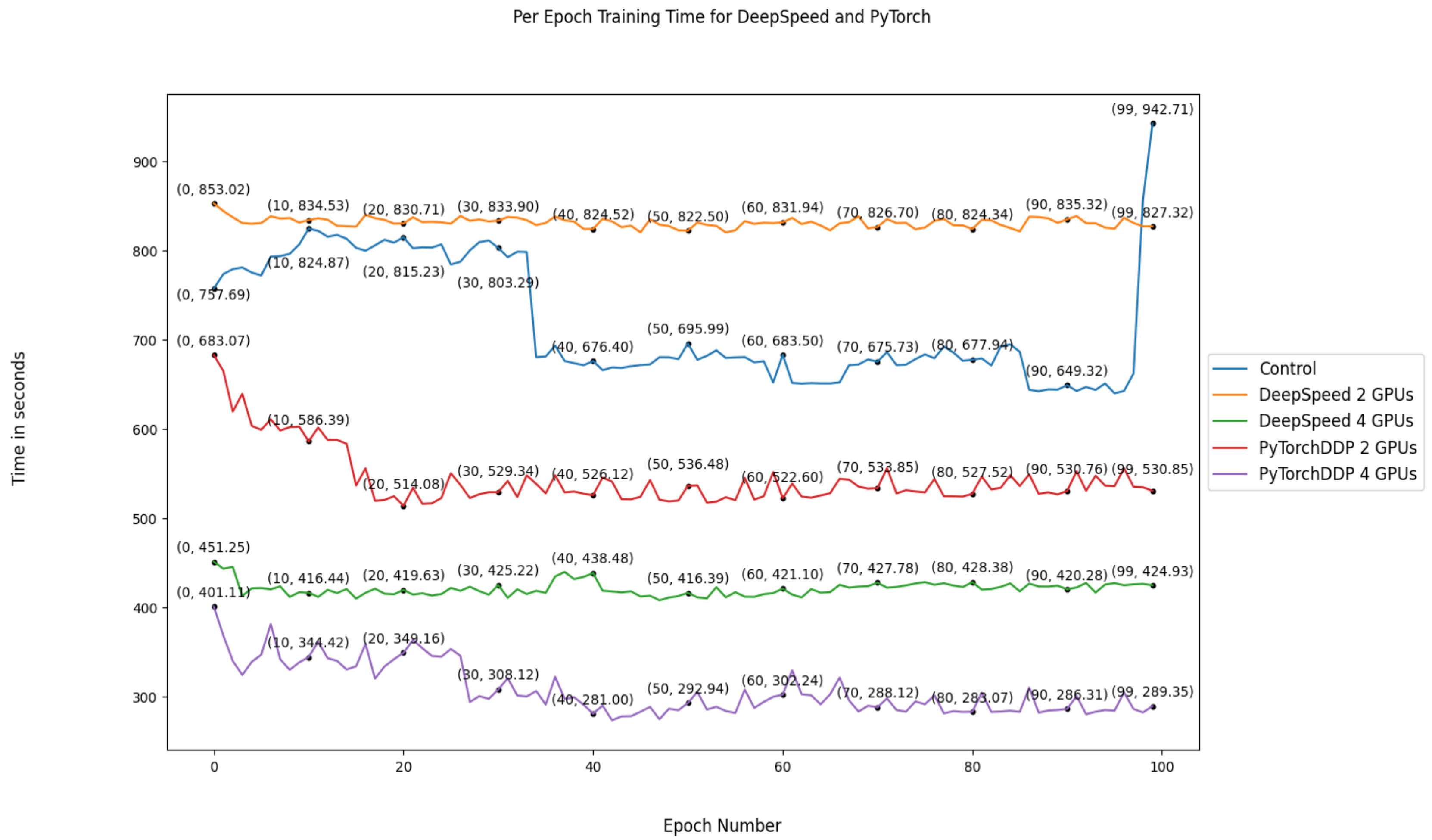
- As a result of ZeRO, DeepSpeed makes it easier to scale models.
- DeepSpeed also provides even more optimization techniques to make model training more efficient.
- DeepSpeed was also designed to be very easy to port existing PyTorch models to.

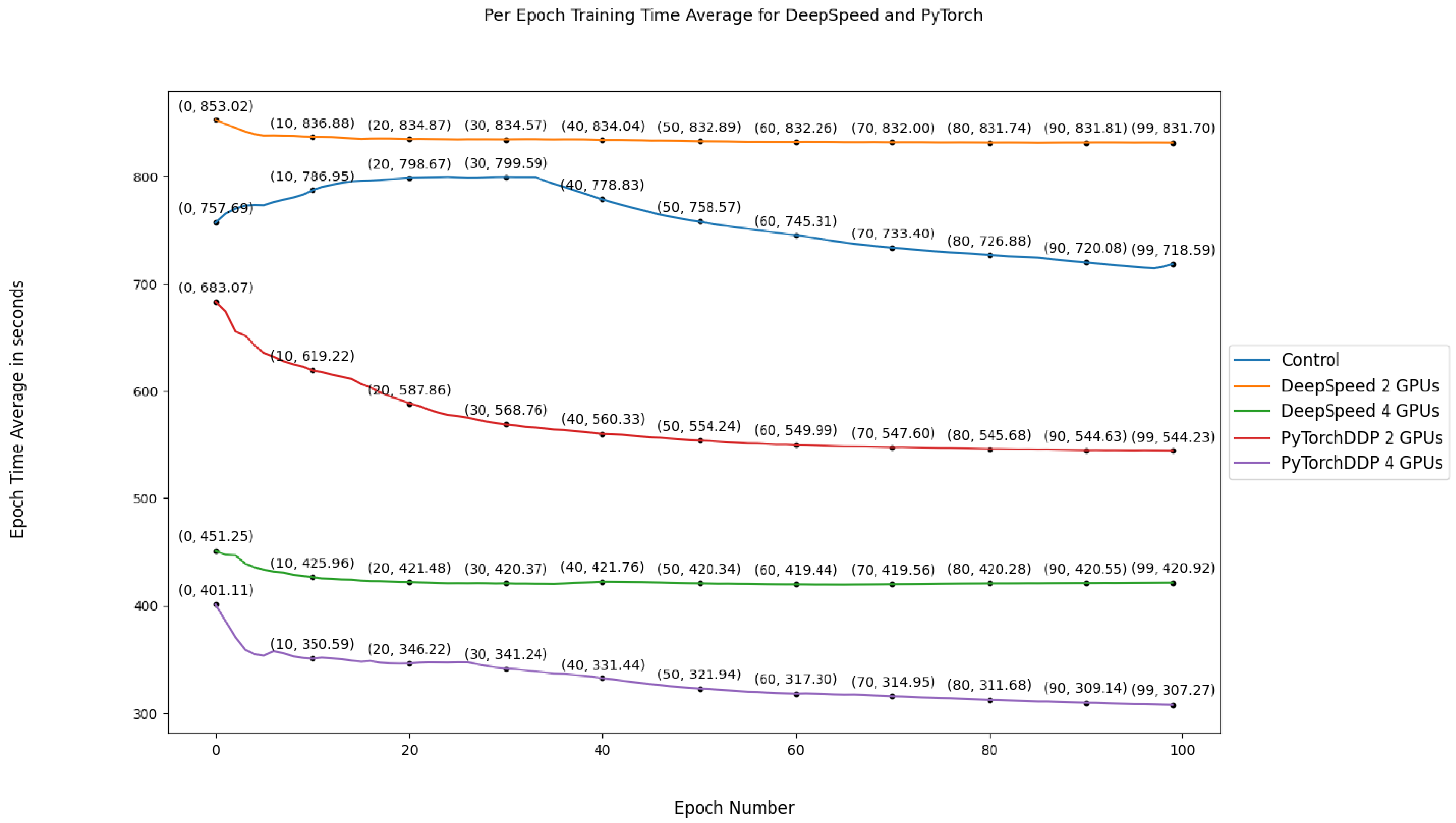
Results

Run Time Methodology

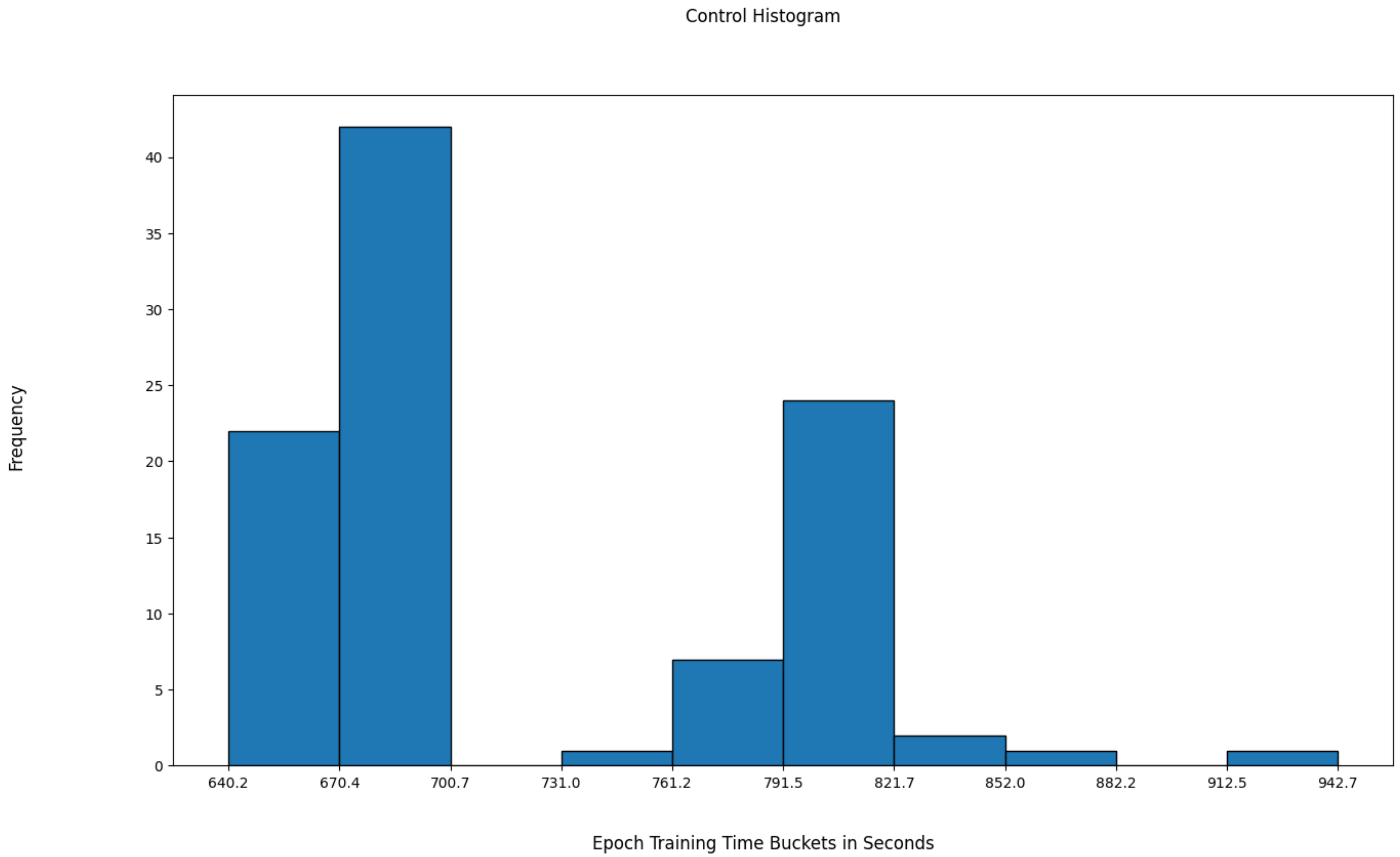
- We run a training session using the 2 packages on a nearly identical training script. Minor variations between the two, but this should only have at most a marginal difference on the runtime
- We compare a Control that uses 1 RTX 3090 that doesn't utilize PyTorch's DDP library, a PyTorch DDP script that uses 2 RTX 3090s, another PyTorch DDP that uses 4 RTX 3090s, a DeepSpeed script that uses 2 RTX 3090s, and another DeepSpeed script that uses 4 RTX 3090s.
- All run for 100 epochs on a training set of 131,072 images with a batch size of 64.
- The images used come from the imagenet-1k dataset. These images were also resized and transformed into tensors for use in our resnet101 model.
- All use a model that has activation checkpointing with 4 checkpoints.
- All run on a fixed seed to ensure they process the same images.
- The DeepSpeed models utilize ZERO level 3.

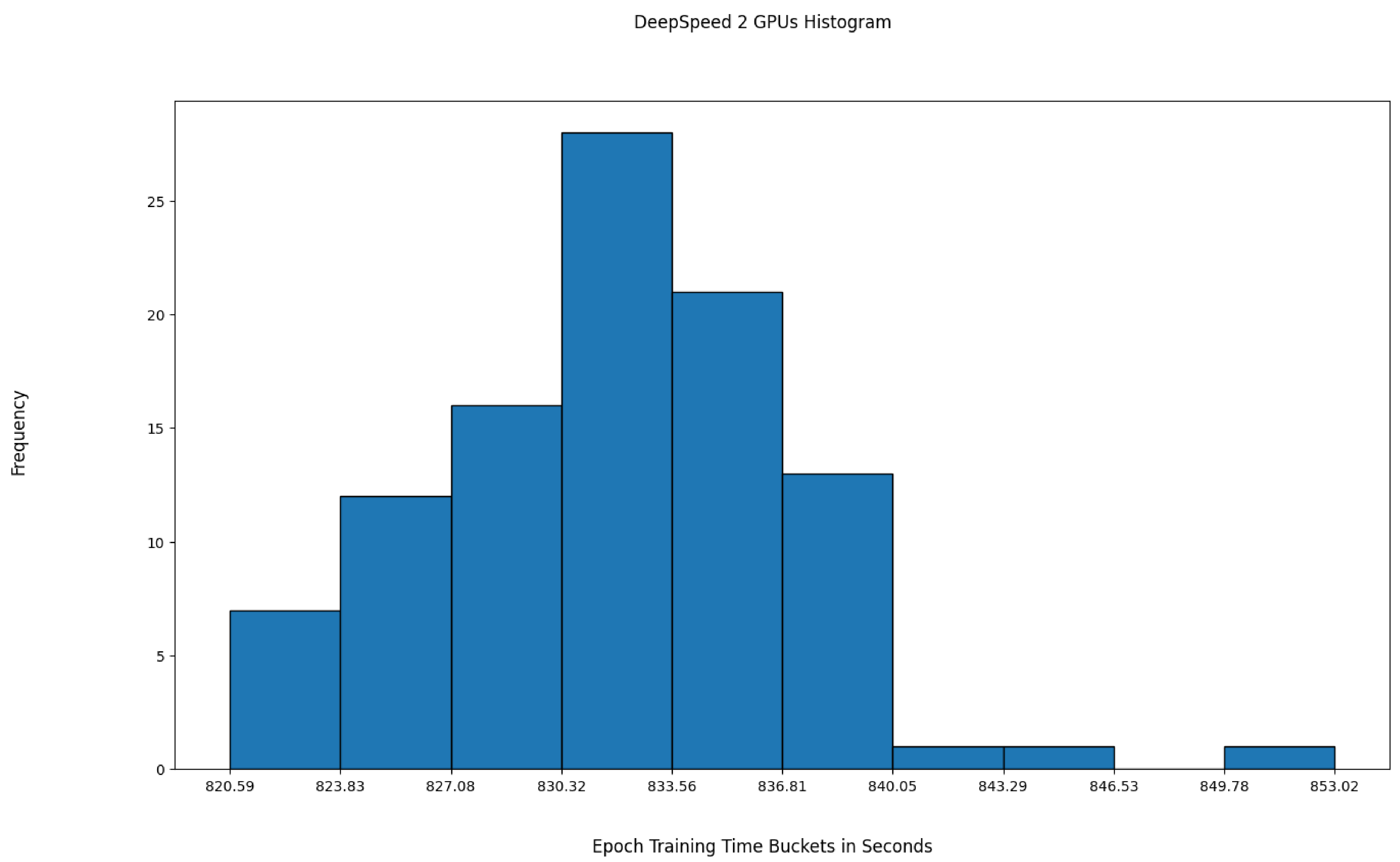
Run Time Comparison

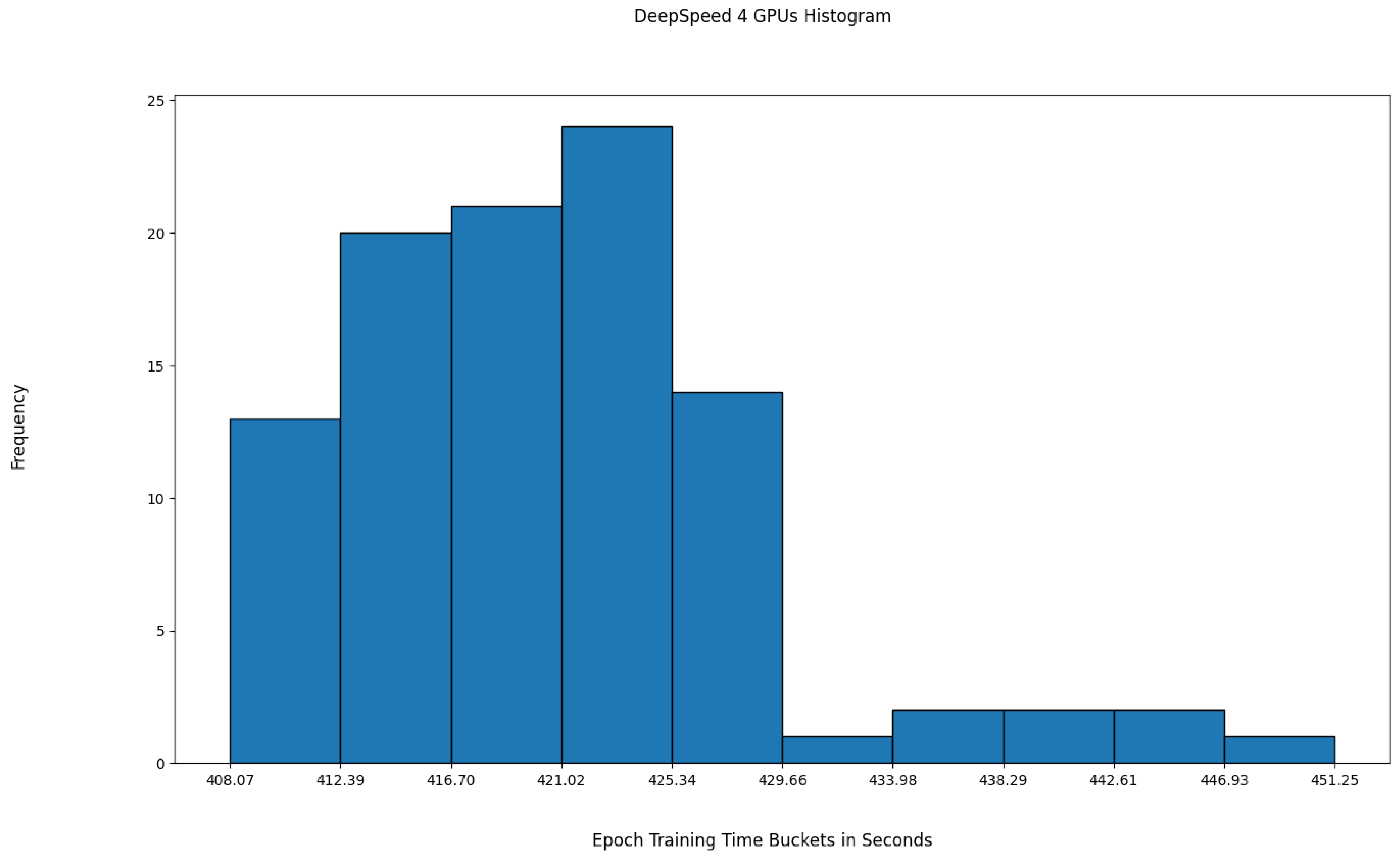


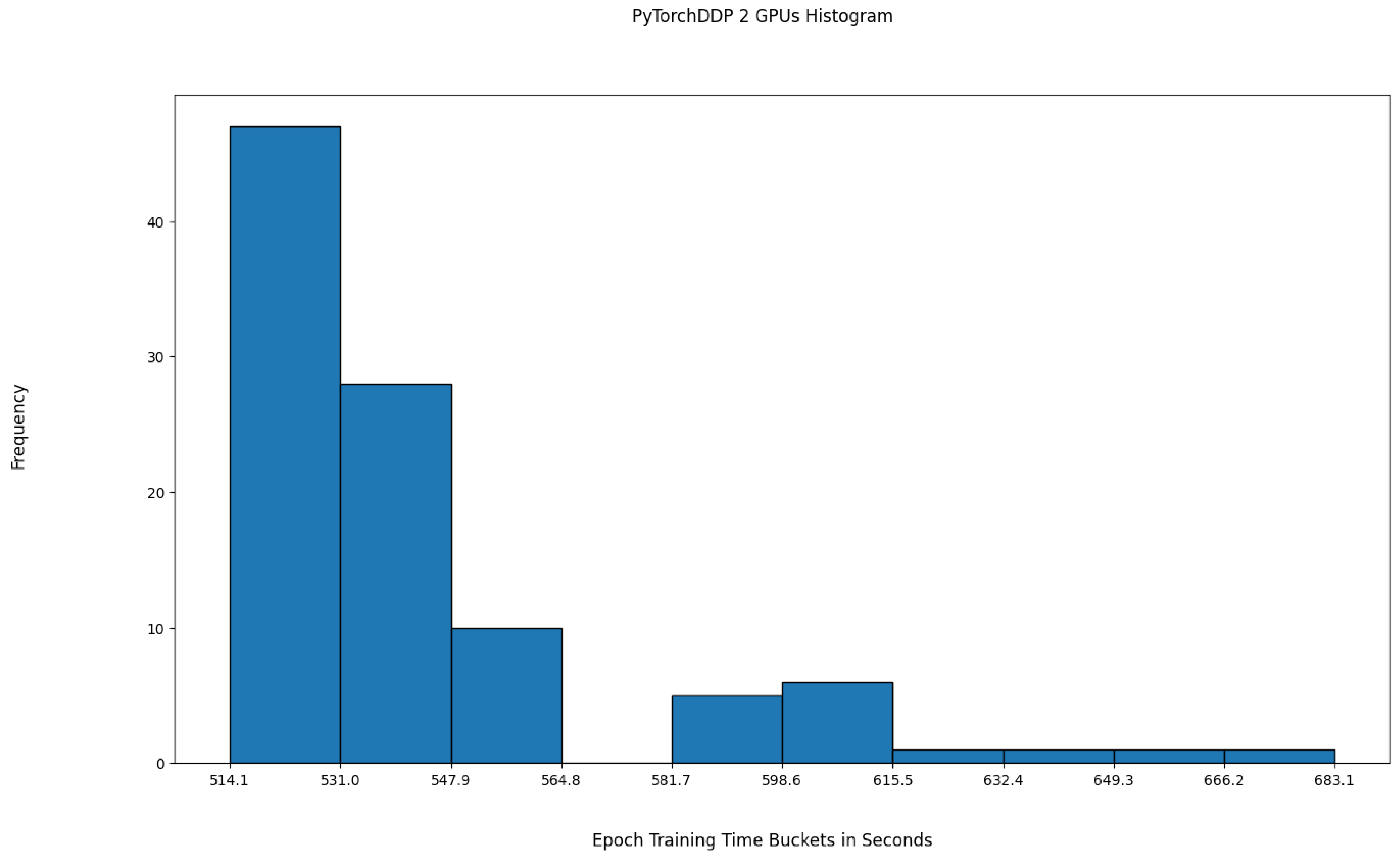

Per Epoch Training Time Average for DeepSpeed and PyTorch
Epoch Time Average in seconds
Epoch Number
Control
DeepSpeed 2 GPUs
DeepSpeed 4 GPUs
PyTorchDDP 2 GPUs
PyTorchDDP 4 GPUs
300
400
500
600
700
800
0
20
40
60
80
100
(0, 853.02)
(10, 836.88)
(20, 834.87)
(30, 834.57)
(40, 834.04)
(50, 832.89)
(60, 832.26)
(70, 832.00)
(80, 831.74)
(90, 831.81)
(99, 831.70)
(0, 757.69)
(10, 786.95)
(20, 798.67)
(30, 799.59)
(40, 778.83)
(50, 758.57)
(60, 745.31)
(70, 733.40)
(80, 726.88)
(90, 720.08)
(99, 718.59)
(0, 683.07)
(10, 619.22)
(20, 587.86)
(30, 568.76)
(40, 560.33)
(50, 554.24)
(60, 549.99)
(70, 547.60)
(80, 545.68)
(90, 544.63)
(99, 544.23)
(0, 451.25)
(10, 425.96)
(20, 421.48)
(30, 420.37)
(40, 421.76)
(50, 420.34)
(60, 419.44)
(70, 419.56)
(80, 420.28)
(90, 420.55)
(99, 420.92)
(0, 401.11)
(10, 350.59)
(20, 346.22)
(30, 341.24)
(40, 331.44)
(50, 321.94)
(60, 317.30)
(70, 314.95)
(80, 311.68)
(90, 309.14)
(99, 307.27)

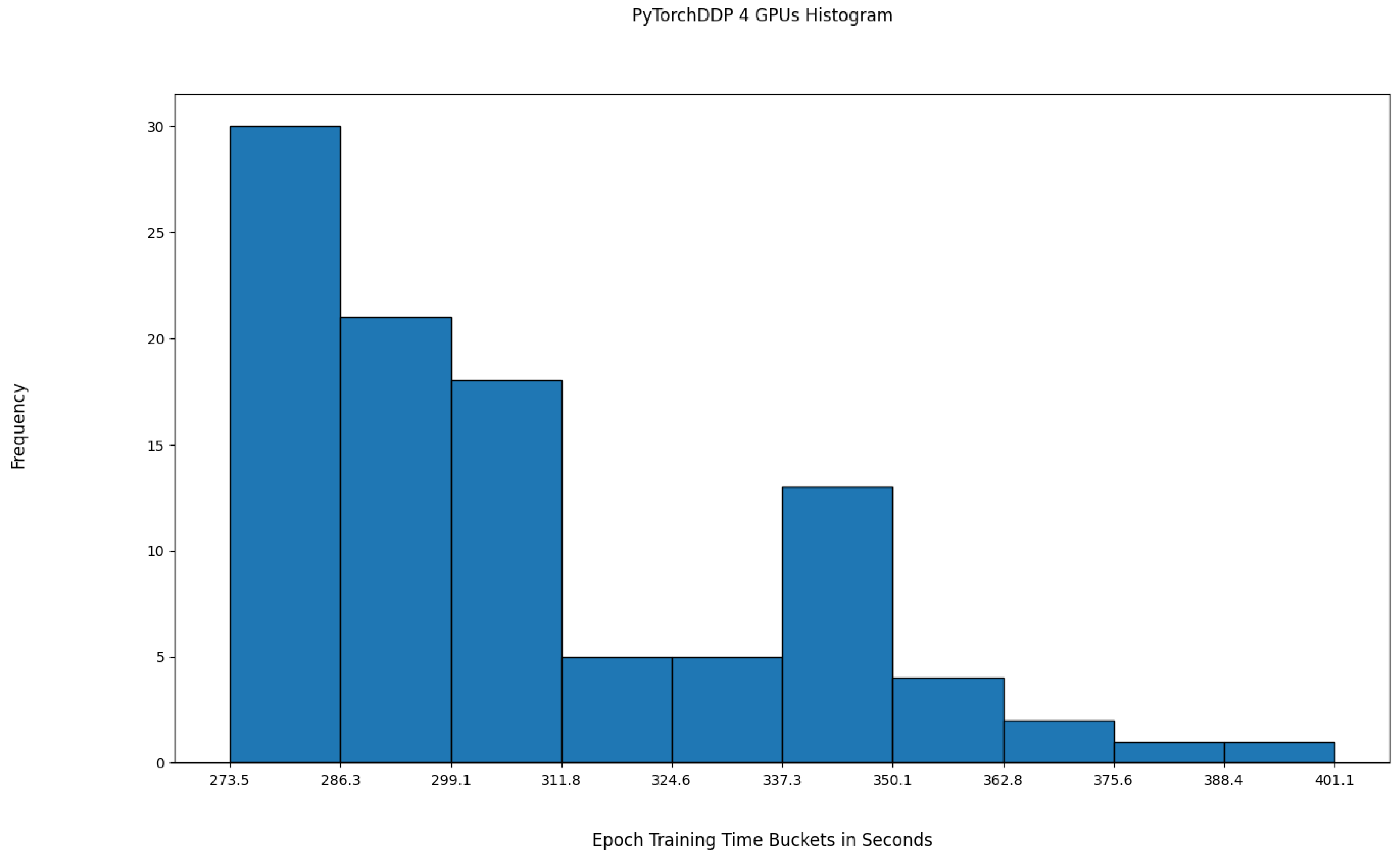
Run Time Distributions




DeepSpeed 2 GPUs Histogram
Frequency
0
5
10
15
20
25
820.59
823.83
827.08
830.32
833.56
836.81
840.05
843.29
846.53
849.78
853.02
Epoch Training Time Buckets in Seconds


DeepSpeed 4 GPUs Histogram
Frequency
0
5
10
15
20
25
408.07
412.39
416.70
421.02
425.34
429.66
433.98
438.29
442.61
446.93
451.25
Epoch Training Time Buckets in Seconds


PyTorchDDP 2 GPUs Histogram
Frequency
0
10
20
30
40
514.1
531.0
547.9
564.8
581.7
598.6
615.5
632.4
649.3
666.2
683.1
Epoch Training Time Buckets in Seconds


PyTorchDDP 4 GPUs Histogram
Frequency
0
5
10
15
20
25
30
273.5
286.3
299.1
311.8
324.6
337.3
350.1
362.8
375.6
388.4
401.1
Epoch Training Time Buckets in Seconds

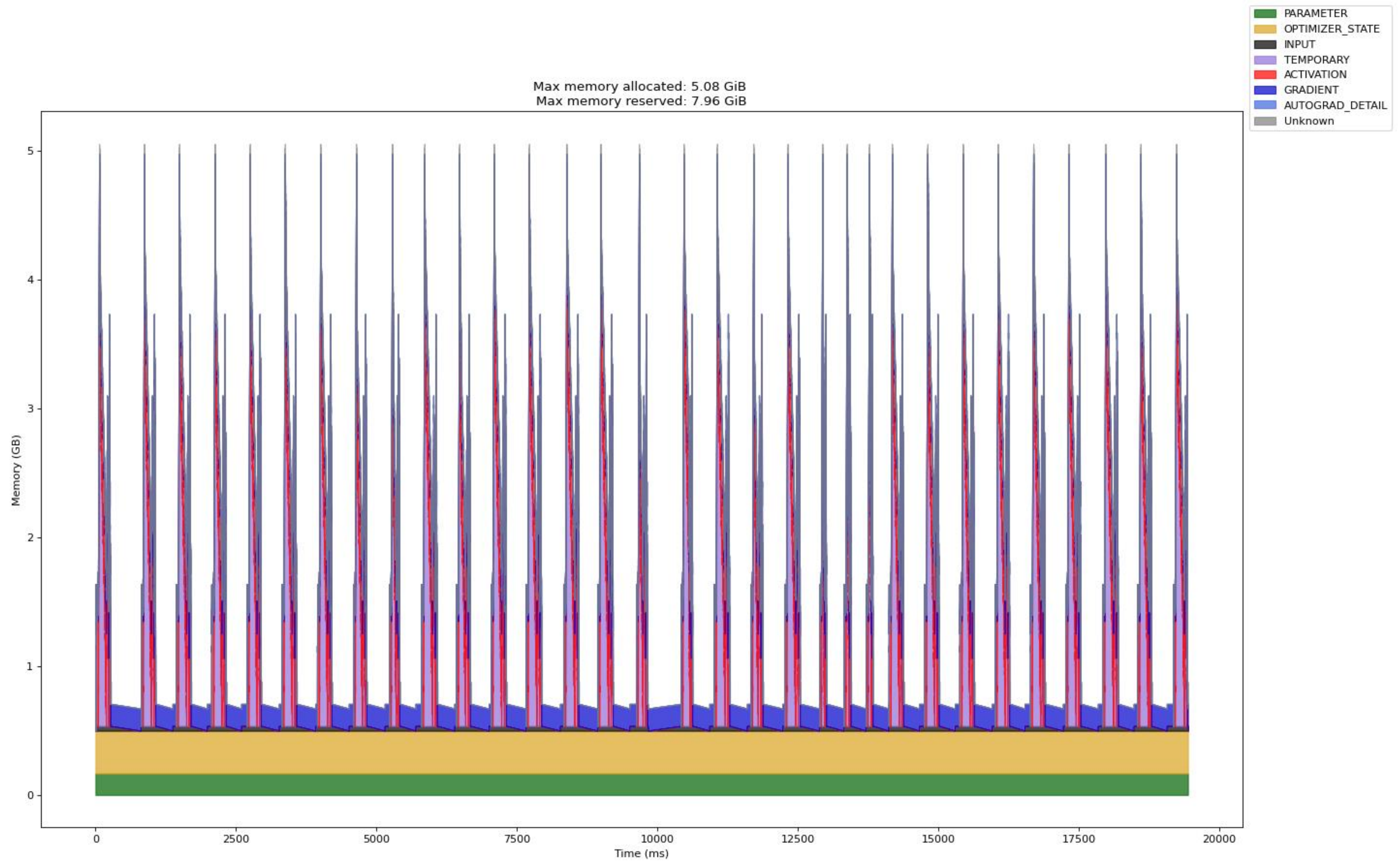
Run Time Analysis

- Despite DeepSpeed with 2 GPUs taking more time over the control, we can see a clear advantage over both the PyTorch DDP run with 2 GPUs and the control when utilizing 4 GPUs
- This supports the the claim that the communication overhead with ZeRO enabled decreases as more GPUs are added making it more effective in larger scale training scenarios
- Overall though PyTorch is clearly faster than DeepSpeed with ZeRO level 3 enabled which is to be expected as it has less overall communication that it needs to facilitate

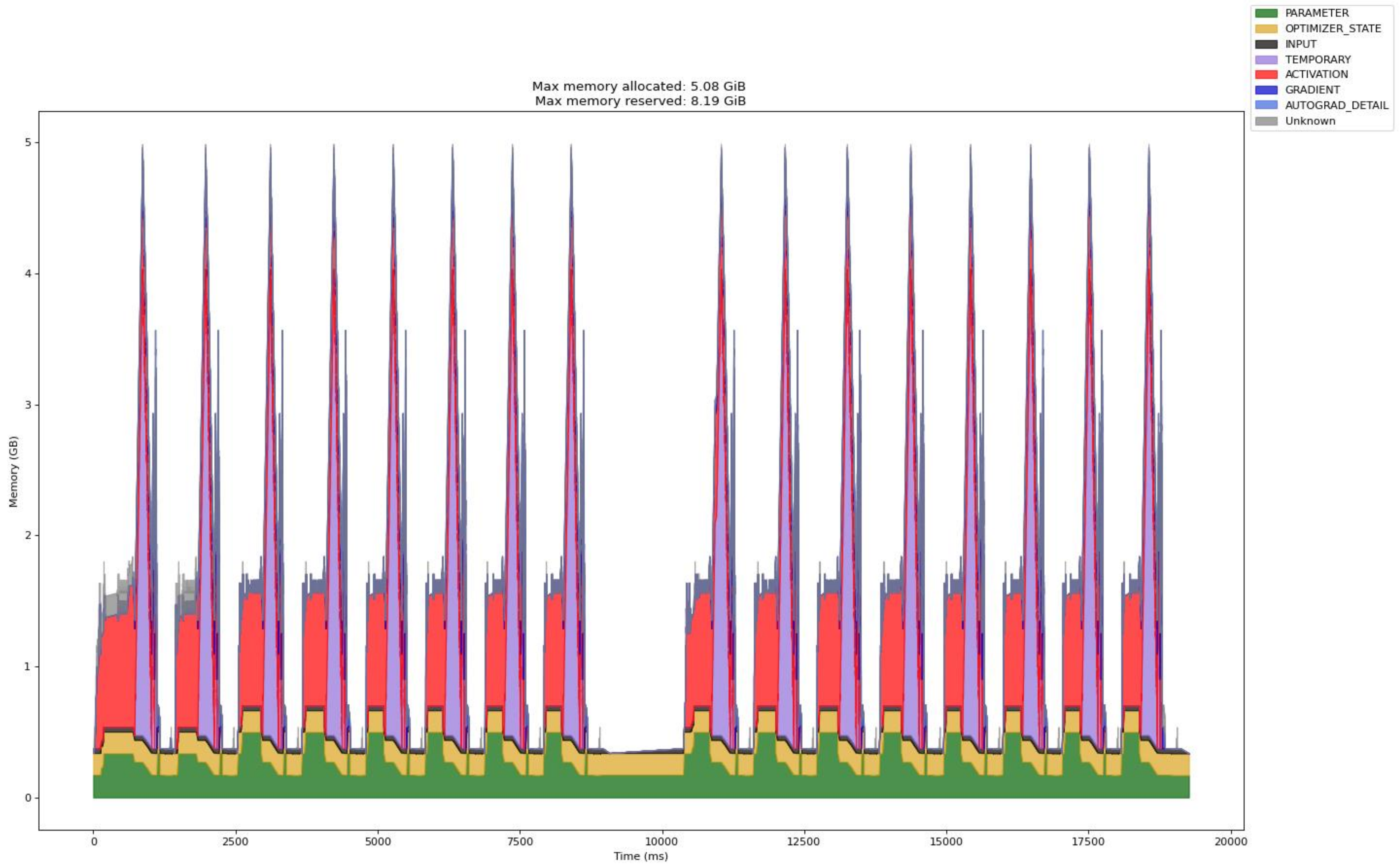
Memory Profiling Methodology

- The methodology to compare the memory overhead between the control and the 2 systems is the same except we scaled down the training as profiling collects a large amount of data causing it to be significantly slower then without profiling
- We run for 10 epochs on a batch size of 64 on a training set of 1024 images. (Note that the profiler only collects data for 2 of the 10 epochs as it uses the remaining to ‘warmup’ for more accurate collection.)
- Additionally, the memory snapshot is only for a single process in the training script.
- While the overall epochs are quite low, the results are consistent with testing I have done that isn’t shown.

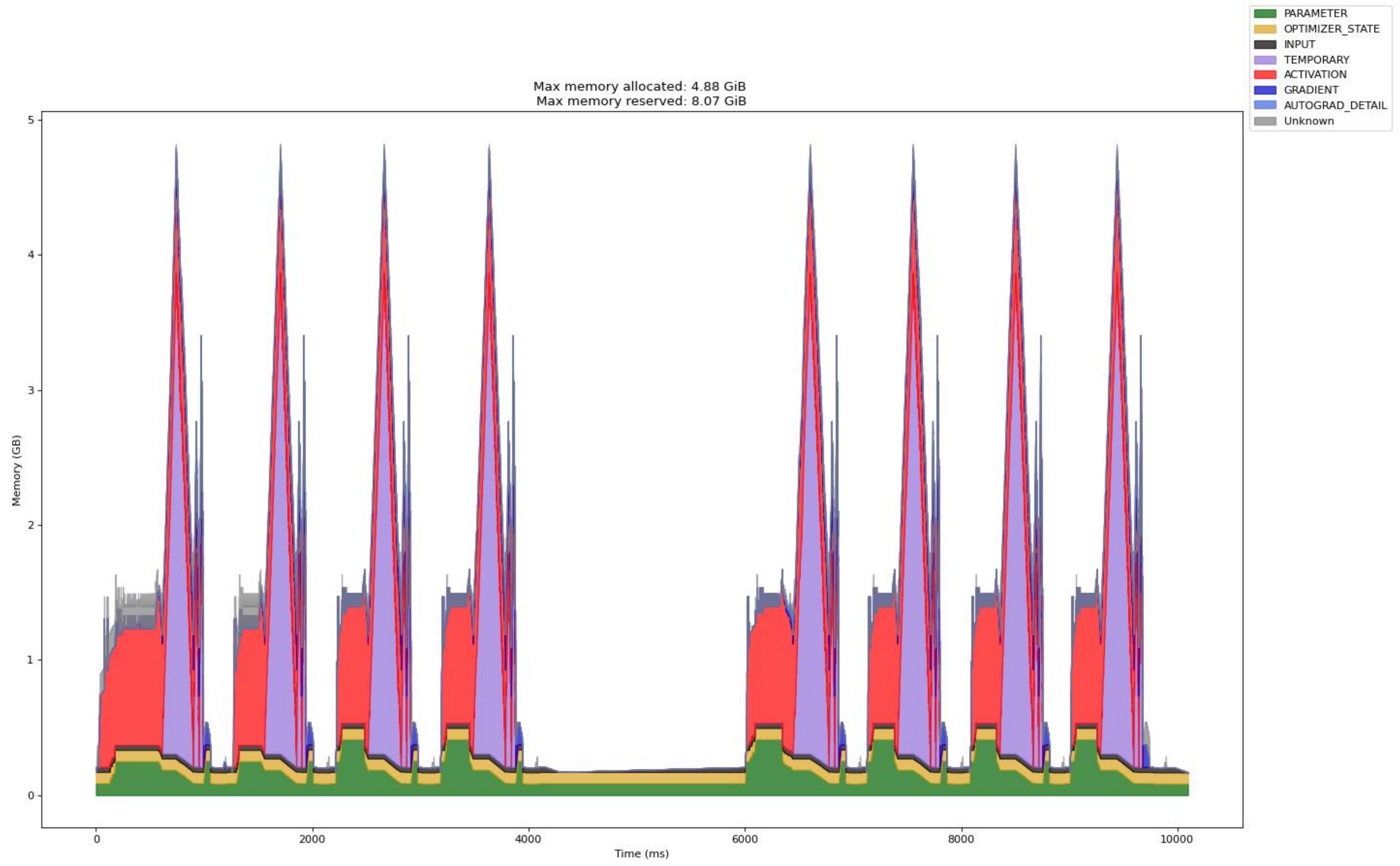
Memory Profiling Aggregated (Control)



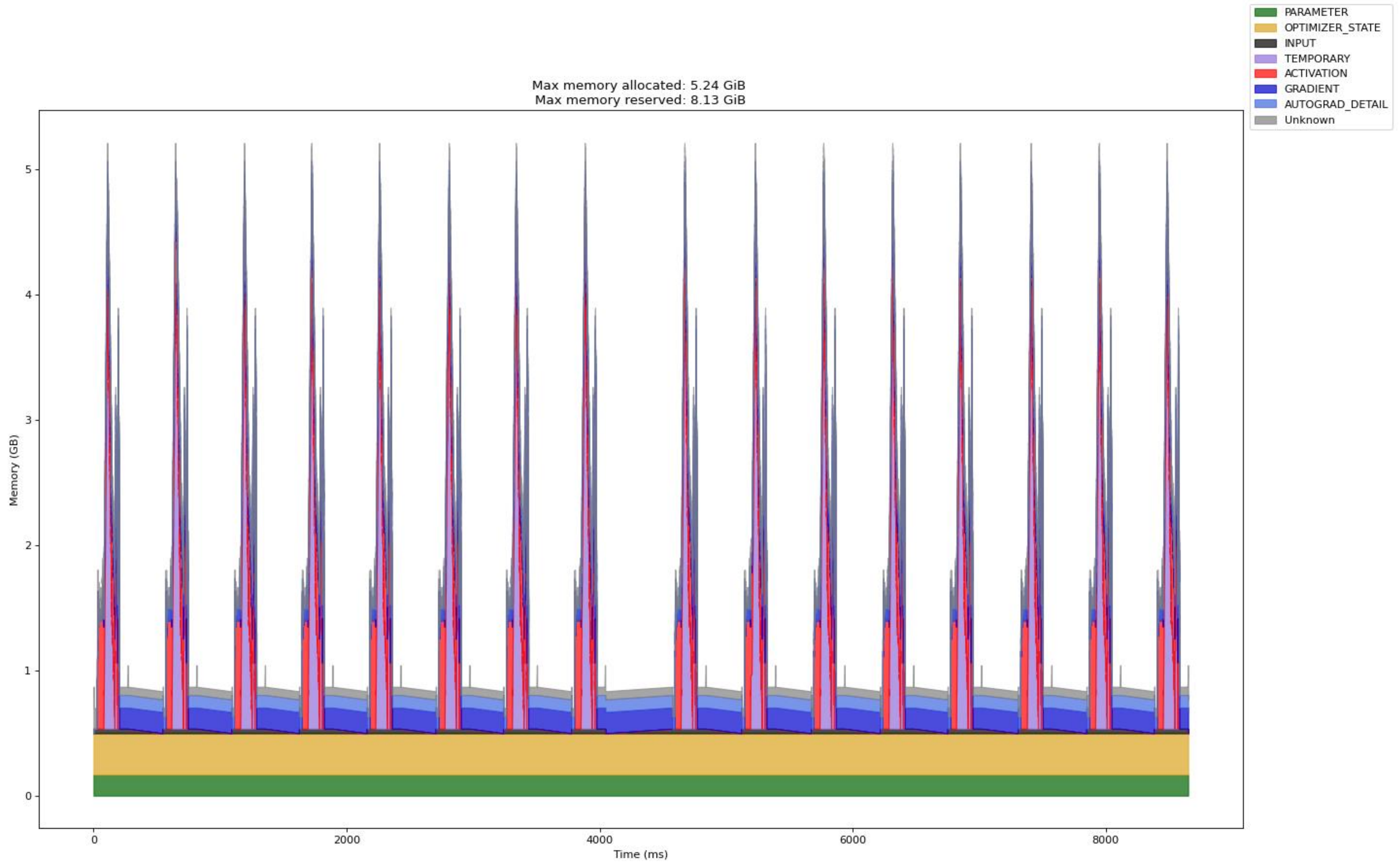
DeepSpeed 2 GPUs



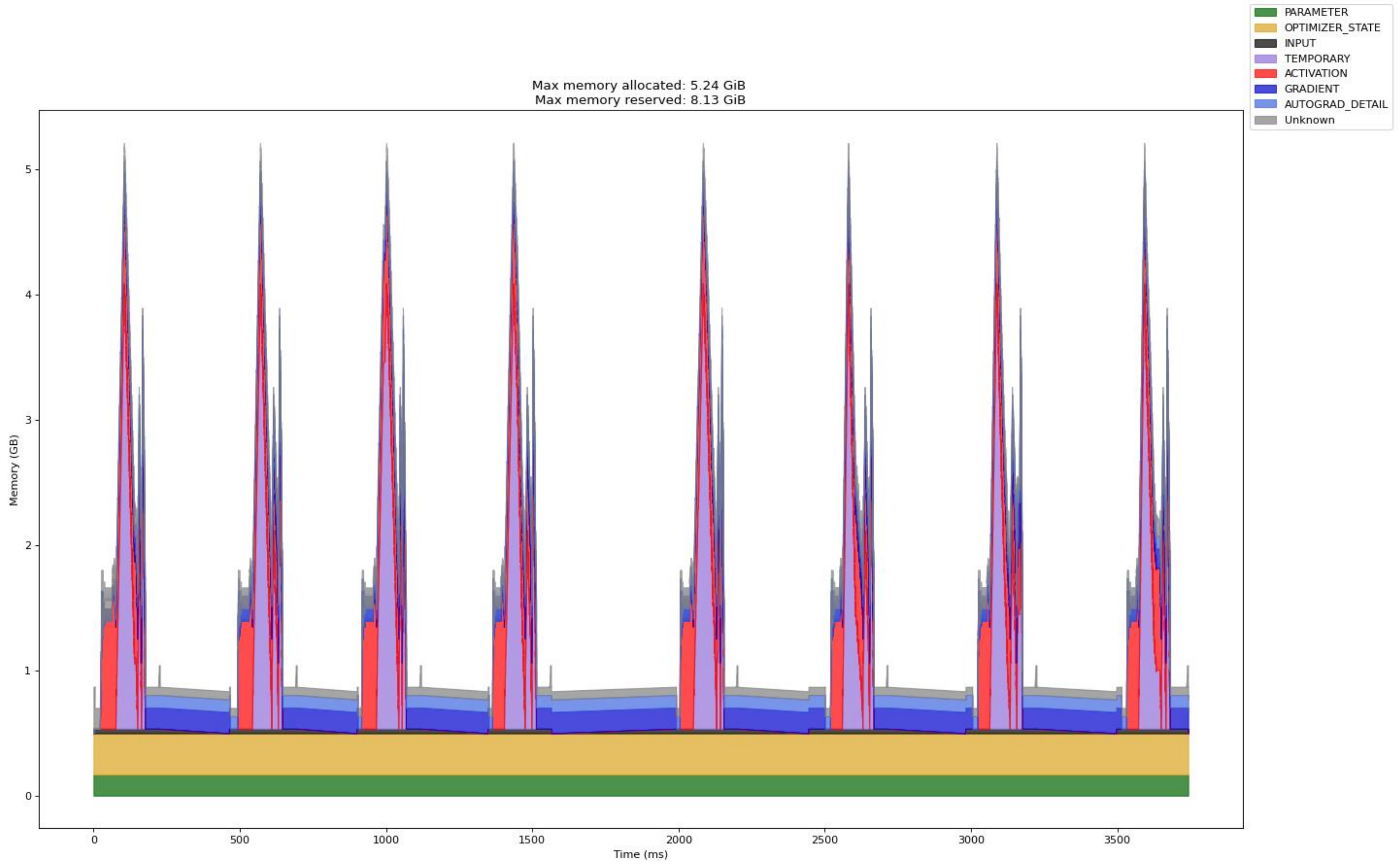
DeepSpeed 4 GPUs



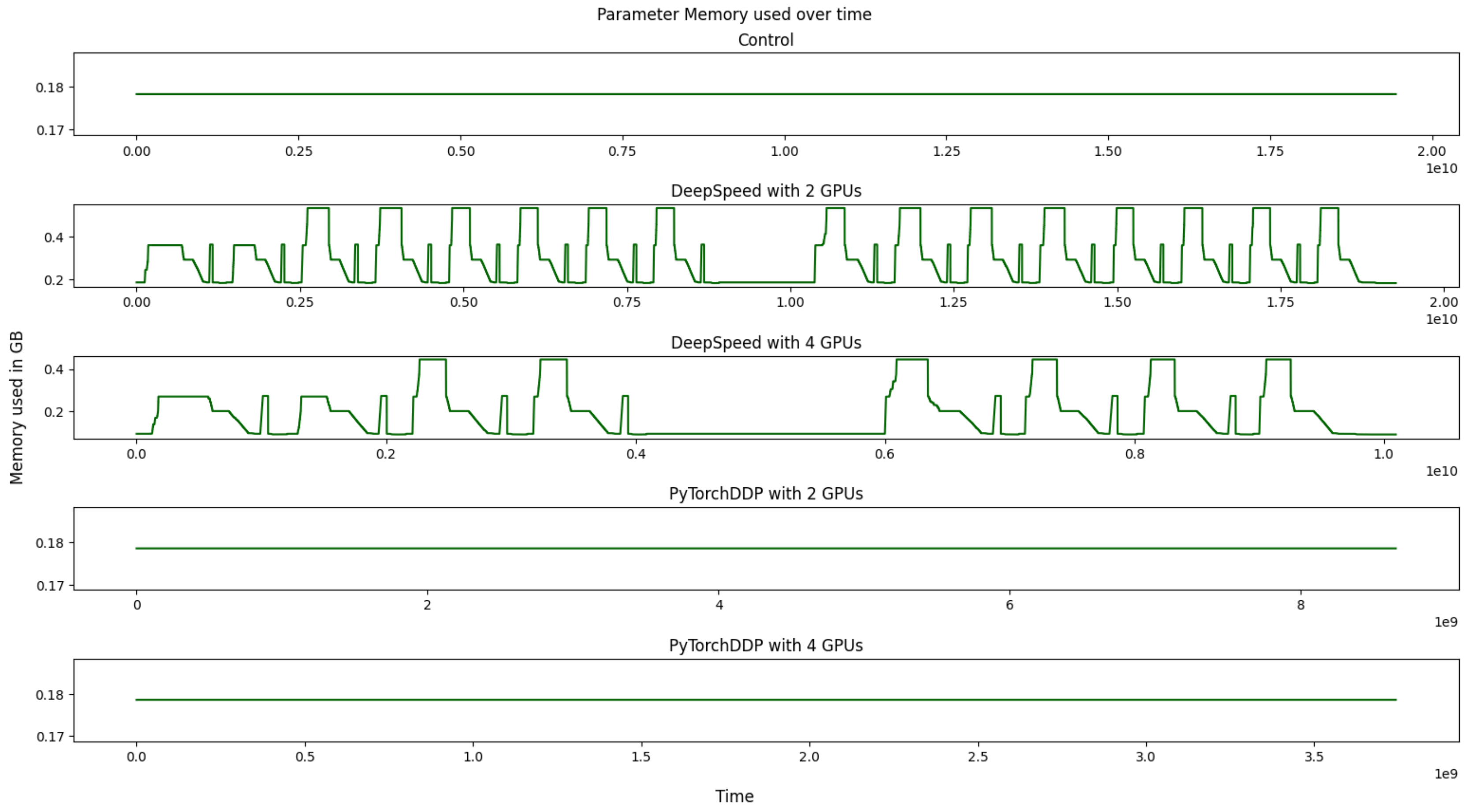
PyTorch 2 GPUs

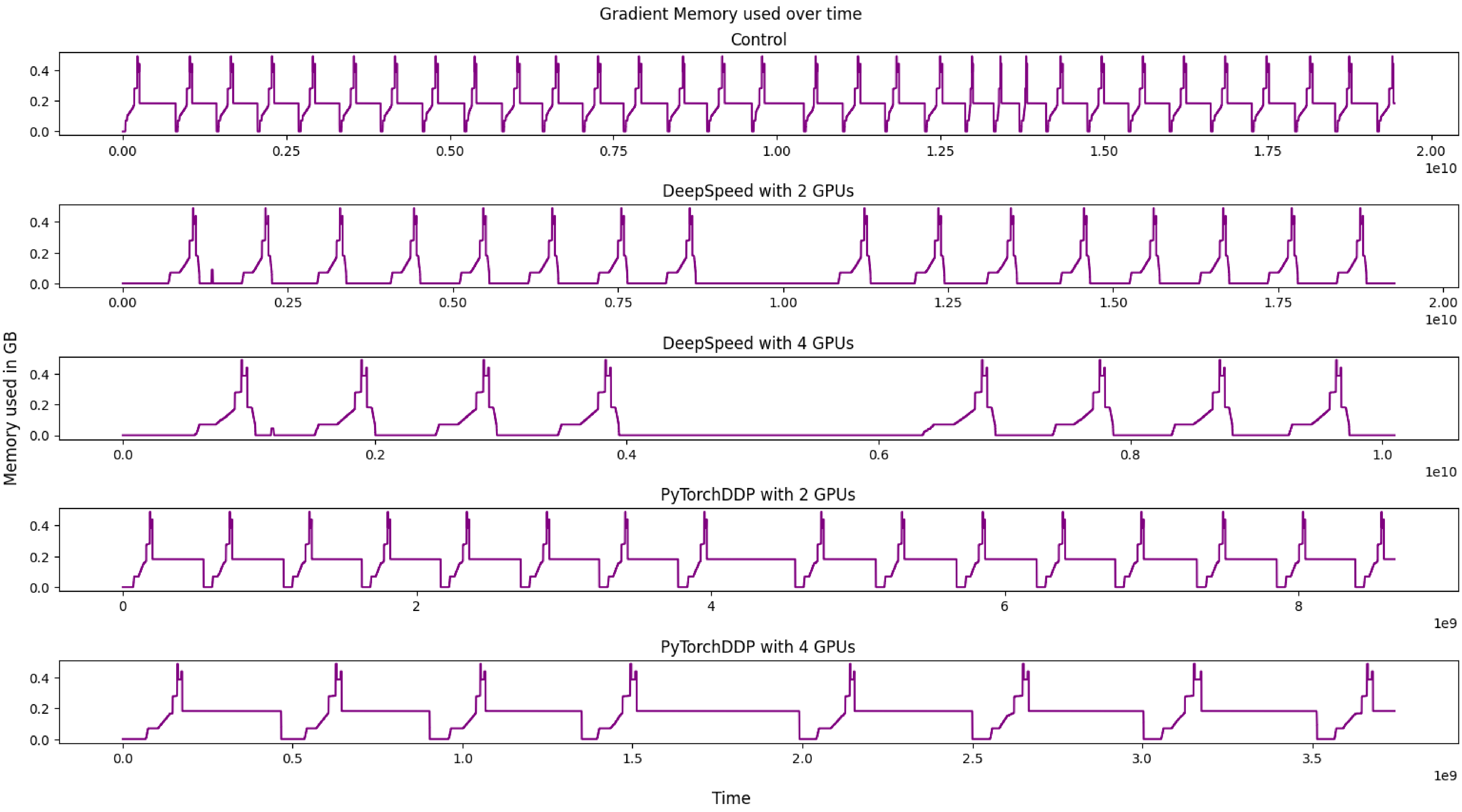


PyTorch 4 GPUs

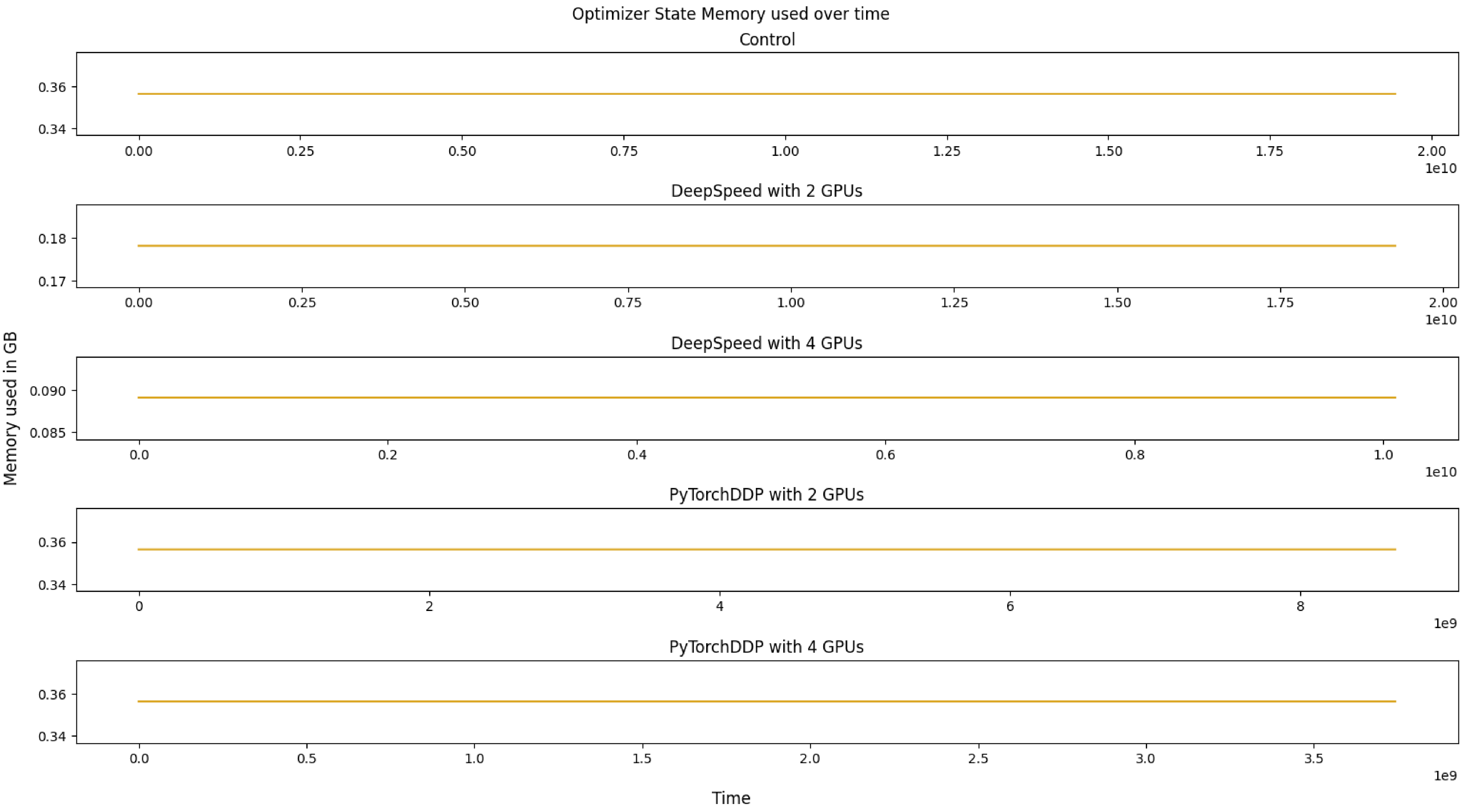


Important Memory Category Data Per System





Gradient Memory used over time
Control
DeepSpeed with 2 GPUs
DeepSpeed with 4 GPUs
PyTorchDDP with 2 GPUs
PyTorchDDP with 4 GPUs
Memory used in GB
Time
0.0
0.2
0.4
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
1e10
0.6
0.8
1.0
0
2
4
6
8
1e9
0.5
1.5
2.0
2.5
3.0
3.5


Optimizer State Memory used over time
Control
0.36
0.34
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
1e10
DeepSpeed with 2 GPUs
0.18
0.17
DeepSpeed with 4 GPUs
0.090
0.085
0.0
0.2
0.4
0.6
0.8
1.0
1e10
PyTorchDDP with 2 GPUs
0
2
4
6
8
1e9
PyTorchDDP with 4 GPUs
0.5
1.5
2.0
2.5
3.0
3.5
Memory used in GB
Time

Memory Profiling Analysis

- First each individual peak corresponds to a single batch being processed in the training step. In total there are 2 times the number of steps performed per epoch since we run for 2 epochs. Since we have a batch size of 64 on a sample of 1024 images, there are 32 peaks for the control and 16 for the 2 GPU runs and 8 for the 4 GPU runs.
- As we can see, overall, the memory utilized by DeepSpeed is slightly lower than the PyTorch runs.
- Contributing to this are mainly the optimizer state memory, the gradient memory, and the parameter memory.

Memory Profiling Analysis Continued

- Specifically, we see a linear decrease in optimization memory with each DeepSpeed run as the number of GPUs increases.
- Additionally, we can see that the gradient memory is almost nonexistent for most of the time and peaks similarly to the PyTorch runs. While we don't see an overall reduction in memory used for this category, we do see idle memory being saved.
- Finally, we see an interesting usage of parameter memory. While at peak times we see DeepSpeed using more parameter memory, this is in the forward pass of the training step where memory is the most abundant. As a result, when memory is the scarcest (i.e. during the backwards pass where activation memory dominates) we see this saving helping reduce the overall memory footprint.
- Overall, the memory utilization between PyTorch and the control are similar. Compared to DeepSpeed the overall memory used at peak times is also similar since this is dominated by activation memory which should be similar across all devices; however, we see clever optimizations that would help as we scale the model to more parameters and training sizes.

Final Thoughts and Conclusions

DeepSpeed is easier to configure more complex training situations

- While I had trouble installing and setting up the DeepSpeed environment, I believe this was due to me learning how to work with the HPC environment.
- Other than this, configuring and using the 2 systems were both relatively easy. PyTorch DDP requires relatively little to setup and use, which is the same for DeepSpeed since its backend communication utilizes a lot from PyTorch DDP.
- However, DeepSpeed exposes a lot of configuration settings to the user with good documentation on how they work. As a result, I feel like it gives more granular control to the user while not forcing the user to be overloaded with configuring and setting things up. For a list of these settings see this site.
- DeepSpeed does this through a configuration file. In this file you can specify a host of different things that affect how you model is controlled. Some of these cannot be used out of the box though and require code changes. For instance, since ZERO 3 distributes parameters across GPUs it requires the user to modify how they might traditionally save their model.

DeepSpeed is the production system

- Between the 2 systems, I believe that DeepSpeed is the clear winner in a production environment.
- As modern models are much bigger than the one I trained, I can see the memory savings be a huge boon to help facilitate training.
- Especially given that modern training uses hundreds or even thousands of GPUs, the additional communication overhead that comes with using ZERO is negligible if this enables the entire model to fit on the GPU without needing to resort to traditional model parallelism.
- PyTorch does run faster in my testing, but it is hard to say how much faster it would be if we scaled the model to more GPUs.

References

- Langer, M., He, Z., Rahayu, W., & Xue, Y. (2020). Distributed Training of Deep Learning Models: A Taxonomic Perspective. *IEEE Transactions on Parallel and Distributed Systems, 31(12), 2802–2818*. (Slides 3-6)
- *Collective operations*. Collective Operations - NCCL 2.28.9 documentation. (n.d.). https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html (Slides 7-15)
- Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, & Soumith Chintala. (2020). PyTorch Distributed: Experiences on Accelerating Data Parallel Training. (Slides 17-24)
- *Training overview and features*. DeepSpeed. (n.d.). https://www.deepspeed.ai/training/ (Slides 26,27,36)
- Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, & Yuxiong He. (2020). ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. (Slides 28-35)
- Project code: https://github.com/EthanHarness/CloudComputingRepo