OPT2024: 16th Annual Workshop on Optimization for Machine Learning

Cyclic Data Parallelism for Efficient Parallelism of Deep Neural Networks

Louis Fournier LOUIS.FOURNIER@ISIR.UPMC.FR
ISIR - Sorbonne Université, Paris - France

Edouard Oyallon
Center for Computational Mathematics, Flatiron Institute, New York, USA

Abstract

Training large deep learning models requires parallelization techniques to scale. In existing methods such as Data Parallelism or ZeRO-DP, micro-batches of data are processed in parallel, which creates two drawbacks: the total memory required to store the model's activations peaks at the end of the forward pass, and gradients must be simultaneously averaged at the end of the backpropagation step. We propose Cyclic Data Parallelism, a novel paradigm shifting the execution of the micro-batches from simultaneous to sequential, with a uniform delay. At the cost of a slight gradient delay, the total memory taken by activations is constant, and the gradient communications are balanced during the training step. With Model Parallelism, our technique reduces the number of GPUs needed, by sharing GPUs across micro-batches. Within the ZeRO-DP framework, our technique allows communication of the model states with point-to-point operations rather than a collective broadcast operation. We illustrate the strength of our approach on the CIFAR-10 and ImageNet datasets.

1. Introduction

Deep learning models have grown significantly in size in recent years, reaching hundreds of billions of parameters and requiring increasingly expensive training [12]. As the cost of training these models continues to rise, there is an increasing need for parallelization strategies. In particular, Data Parallelism (DP) [6, 17] remains the dominant method for training DNNs at scale. In DP, the model to be trained is first replicated on multiple workers. Then, each worker locally computes gradients. They are then communicated and averaged across all workers, and the averaged gradient is used to locally update models, typically following SGD.

However, DP has significant drawbacks. First, the communication step between workers is synchronous, and waiting for the slowest workers to complete their gradient computations results in idle workers [11]. Second, gradients are communicated globally using an all-reduce operation, which means that the communication step becomes a challenge as the number of workers increases [2]. Finally, the total memory used by all workers grows linearly with the number of workers since the model is fully replicated on each worker [23]. This requirement can be impractical since modern model sizes exceed the memory capacity of a single device (e.g., a GPU).

In this work, we tackle the memory and communication drawbacks of DP by proposing to change the execution time of workers in DP from simultaneous to sequential. We refer to this process as Cyclic Data Parallelism (CDP). Our modification of standard DP aims to balance both the communication costs and the overall memory usage, by relying heavily on the sequential nature of the execution steps of DNNs during training. More specifically, CDP reduces the cost of gradient

© L. Fournier & E. Oyallon.

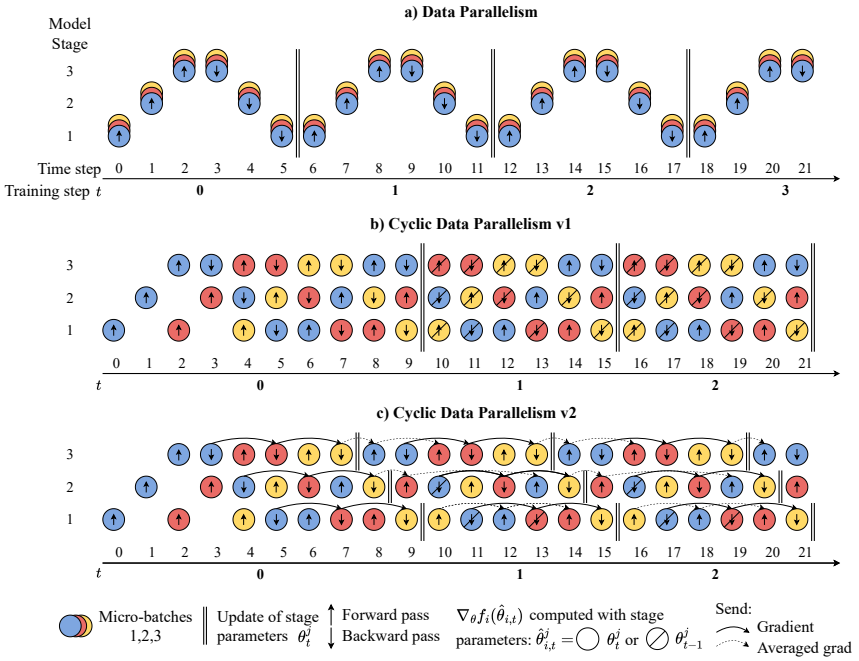

Figure 1: **Timeline of executions for DP and the two versions of CDP, for N=3 workers. (a) DP.** The workers begin executing their forward pass simultaneously, maintaining synchronization throughout the entire forward-backward pass. **(b) CDP-v1.** The workers begin executing with an equal delay between them (2 time steps). The model's parameters are updated with a delay constant equal to one training step. **(c) CDP-v2.** This delay is halved in CDP-v2, by allowing the stages to update and send gradients independently. The novel balanced communication scheme is indicated. The total complexity of a training step does not change, but activation memory does not peak in CDP.

communications in DP from collective communications at the end of the training step to point-to-point communications balanced over the entire training step. This balances the total memory used by all workers, at the cost of a small gradient delay. CDP can be combined with standard parallelization implementations for further improvements.

Contributions. Our contributions are as follows: **(a)** We propose CDP, an alternative to DP that balances gradient communications and overall memory usage across training. **(b)** We particularize the CDP paradigm to approaches such as Model Parallelism (MP) and Zero Redundancy Optimizer powered DP (ZeRO-DP) [23], showing improvements in all cases. **(c)** Empirically, we show that the gradient delay of CDP does not affect training on the CIFAR-10 and ImageNet datasets.

2. The Cyclic Data Parallelism paradigm

2.1. Mini-batch SGD with Data Parallelism

Consider the DP paradigm for a mini-batch SGD step. First, the model is replicated over N workers, each fed with mini-batch slices (i.e., micro-batches [13]). At training step t, each replica $n \in [1, N]$ uses its micro-batch to compute a training objective f_n parameterized by θ_t, and its corresponding

gradient $\nabla f_n(\theta_t)$ via a forward and backward pass. We assume that the model can be partitioned into N sequential stages, and write $\theta_t = \{\theta_t^j\}_{j\in[1,N]}$ to emphasize the dependency in the stage j. The parameter gradients are averaged over all workers, typically using an all-reduce operation [2]. Finally, each worker locally updates the parameters θ_t with the averaged gradient. With the learning rate γ_t, the standard update rule [27] which results can be written as

$$\theta_{t+1} = \theta_t - \frac{\gamma_t}{N}\sum_{n=1}^{N} \nabla f_n(\theta_t). \tag{DP}$$

The execution timeline of DP is shown in Fig. 1a, where one time step corresponds to the execution of a forward or backward pass of a stage (we assume a homogeneous execution time between all workers), and the training step t equal to $2N$ time steps.

This paradigm has several inherent problems. The total number of retained activations peaks every $2N$ time steps, as all workers simultaneously store all intermediate activations before the backward pass. This can be seen at step 2 in Fig. 1a. Then, the gradient synchronization barrier every $2N$ steps introduces latency and workers idling as they wait for the communication to start but also finish, as the communication overhead grows linearly with the number of workers N.

2.2. Towards delayed mini-batch SGD with Cyclic Data Parallelism

Algorithmic description. Contrary to the previous DP approach, our main idea is to break the synchrony between the forward and backward passes of the N micro-batches. Instead of each step being computed simultaneously, we assume that each micro-batch computation is delayed by 2 time steps between one worker and a previous worker. More specifically, the computations of ∇f_n are delayed by a delay of 2 time steps compared to ∇f_{n-1} (where n is taken modulo N). This results in a cyclic pattern, illustrated in Fig. 1b and c. Furthermore, each stage constantly performs either a forward or a backward pass on a single and distinct micro-batch: in particular, this implies that the maximum number of activations stored at a given time step is nearly constant during training, and in this case, smaller compared to DP. We call this concept **Cyclic Data Parallelism** (CDP).

Update rules. While the forward and backward passes are fully defined assuming an available parameter θ_t, one has to define the corresponding update rule to generate θ_{t+1}, since breaking the synchrony between the backward passes of the micro-batches makes it impossible to follow Eq. (DP). Similar to [3, 37], at the training step t we introduce an auxiliary variable $\{\hat{\theta}_{n,t}\}_n$ representing the parameter available for gradient computation. Formally, our algorithm can be written as

$$\theta_{t+1} = \theta_t - \frac{\gamma_t}{N}\sum_{n=1}^{N} \nabla f_n(\hat{\theta}_{n,t}) \;\text{ with } \hat{\theta}_{n,t}^j = u_{n,j}(\theta_t^j, \theta_{t-1}^j)\,, \tag{CDP}$$

where $u_{n,j}(a,b) \in \{a,b\}$ is a rule on the parameters, that depends on both the micro-batch n and the stage j. Thus, $u_{n,j}$ implicitly depends on the time step of the algorithm but not on the training step in our paradigm. An alternative that would is possible, and would easily allow more complex asynchronous or randomized variants. Note that some rules $u_{n,j}$ are not possible due to the delay between workers, preventing for example the update rule of DP. In particular, we propose two update rules with specific rules $u_{n,j}$ that can be considered as the edge cases of the update rule, with maximum and minimum delay, respectively, with all other rules $u_{n,j}$ being an intermediate between

them. First, we use a simultaneous barrier after the Nth micro-batch has finished its computation, which is illustrated in Fig. 1b with a barrier at the time step 9. In this specific case, a consistent update consists of the rule $u_{n,j}(a,b) = b$, using the stored $\hat{\theta}_{n,t} = \theta_{t-1}$. This results in

$$\theta_{t+1} = \theta_t - \frac{\gamma_t}{N} \sum_{n=1}^{N} \nabla f_n(\theta_{t-1}) \,. \tag{CDP-v1}$$

We refer to this update rule as CDP-v1. This one step delay rule also corresponds to the ones of [20, 26]. We now introduce a more elaborate rule referred to as CDP-v2 and illustrated in Fig. 1c. We fix the rule $u_{n,j}$ so that after the Nth micro-batch has finished computing the gradient of a stage, it directly transmits the updated stage parameters to be computed on a micro-batch (with a predefined order of communication). This gives the rule $u_{n,j}(a,b) = a$ if $j \geq N - n + 1$, and b otherwise, reducing by half the number of delayed parameters compared to CDP-v1. Note that the parameters in the workers' memory are always the freshest available. This update is written as

$$\theta_{t+1} = \theta_t - \frac{\gamma_t}{N} \sum_{n=1}^{N} \nabla f_n(\theta_{t-1}^1, ..., \theta_{t-1}^{N-n}, \theta_t^{N-n+1}, ..., \theta_t^N) \,. \tag{CDP-v2}$$

Note that in both cases the gradient communication does not have to be *simultaneous*, instead, the results can be communicated at intermediate steps, benefiting the overall communication bandwidth. We propose a possible communication scheme for CDP-v2 in Fig. 1c. We describe in Alg. 1 of the Appendix the computation and communication done by one worker.

Remark on the convergence. Note that Eq. (CDP) can, in the worst case, be understood as SGD with delayed gradients with a fixed delay of 1. There is a large literature studying the convergence of delayed first-order methods [18, 33, 38], guaranteeing a convergence rate almost equal to that of SGD. In practice, very large DNNs using Transformer or GPT-2 architecture have been shown to converge similarly with or without a delay of 1 step [4, 8, 20, 26, 28, 35].

3. Variants of DP, MP and ZeRO-DP with CDP

Representation. We now present how CDP reduces the computational overhead of other parallelism frameworks (see the Appendix A for the Related Work). In particular, we study the integration of CDP with DP (on one or several GPUs), Model Parallelism (MP) [6, 29], and ZeRO-DP [23, 40]; as well as the link to Pipeline Paralellism (PP) [13, 19]. The implementations are schematically represented in Fig. 2. We first discuss the theoretical improvements of CDP.

Analytical comparisons. Note N the number of model stages and of micro-batches (of equal size B). We will refer to Ψ_P as the parameter memory of the model (including the optimizer states) and Ψ_A as the activations memory (for one data sample). In MP, activations are communicated accross stages, and this subset is noted by Ψ_A^{int} (with $\Psi_A^{\text{int}} \leq \Psi_A$). In Tab. 1, we summarize the memory requirements per device, the communication volume, and the maximum number of communication steps required between 2 time steps, for DP, MP, PP, and ZeRO-DP with and without CDP.

We find the following improvements with CDP compared to DP. CDP halves the memory required to train on DP with a single-GPU device. Multi-GPU DP with CDP doesn't require an all-reduce operation at the end of the training step but only point-to-point communications at each time step.

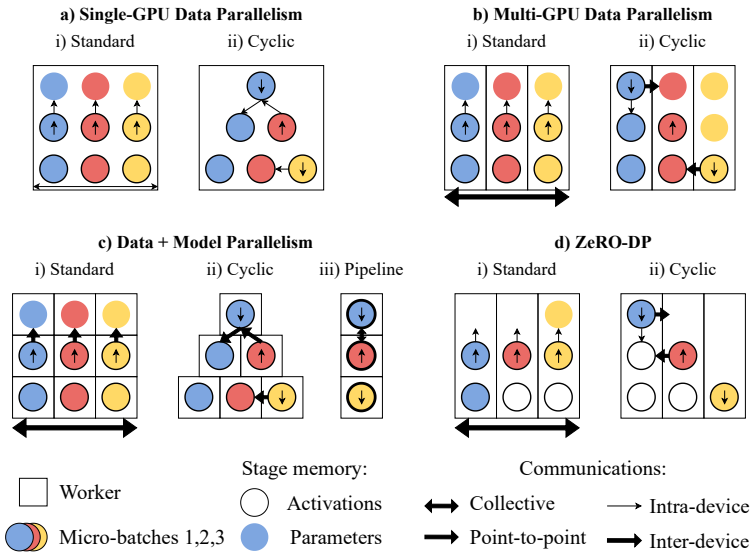

Figure 2: **Comparison between parallelism frameworks with and without using CDP**, for N=3. Devices (e.g., GPUs) are represented by rectangles and micro-batches by color. Model stages are represented by their activation (black circle) and parameter memories (disk). Communications are intra or inter-device (thin or thick arrow), collective or point-to-point (double-headed or single-headed arrow). **(a) Single-GPU DP.** On a single high-connectivity device, we can achieve a memory reduction of half. **(b) Multi-GPU DP.** Communications can be balanced when using multiple GPUs with CDP. **(c) DP+MP.** Both the number of required GPUs and the communications are reduced compared to a standard implementation of MP with DP. Only N GPUs are needed in PP, but they require more activation memory (thicker circle). **(d) ZeRO-DP.** The model states needs to be sent or received by only one worker at each time step, instead of the broadcast operation of ZeRO-DP.

This improvement is also found with MP and ZeRO-DP with CDP. CDP allows MP to halve the number of GPUs needed, thus halving the memory required, as well as the communication of gradients between GPUs. The number of GPUs can be further reduced from MP with CDP to using PP as in [19], which can be seen as a particular implementation of CDP. However, this requires the GPUs to be able to store the activations of the entire mini-batch. More generally, a disadvantage of MP and PP is that they need to communicate activations, which scale with the batch size B. We discuss each setting in more depth in the Appendix C.

4. Numerical analysis

Framework. To test that our update rules do not affect convergence, we implement them in a standard training pipeline on CIFAR-10 and ImageNet. Details are provided in the Appendix D.

Results. We report in Tab. 2 the test accuracy of our models for the three rules CDP-v1, CDP-v2, and DP, for 5 runs on CIFAR-10 and 3 runs on ImageNet respectively. CDP leads to similar or better performance compared to DP, consistent with the experimental findings of [20, 26]. For CIFAR-10, CDP-v2 significantly outperforms CDP-v1, demonstrating our improvement over PipeDream-2BW's rule. We also show in Fig. 3 how delay affects only the start of the training loss curves on ImageNet.

Table 1: **Theoretical cost of the parallelism implementations discussed in Sec. 3.** All implementations are improved (in bold) by using CDP (Cyclic) over DP, in terms of memory (per GPU or number of GPUs) or communication. Parameter and activation memories of a model (for 1 data sample) are noted as Ψ_P and Ψ_A (Ψ_A^{int} for the subset communicated in MP). N indicates both the number of stages and of micro-batches (of size B). The communication volume is the size of the communicated tensors. 'Max com. steps' is the maximum number of communication steps between 2 time steps for 2 workers, which scales as $\mathcal{O}(\log(N))$ for a collective operation compared to point-to-point communications. The parameter memory in Single-GPU DP may depend on the implementation.

	Memory per GPU		Communications inter-GPUs		Nb of	
Implementation	**Activations**	**Parameters**	**Volume**	**Max com. steps**	**GPUs**	**Rule**
Single-GPU DP	$NB\Psi_A$	$N\Psi_P$			1	(DP)
+ Cyclic	$\boldsymbol{\frac{N+1}{2}B\Psi_A}$	$\boldsymbol{\frac{N+1}{2}\Psi_P}$			1	(CDP)
Multi-GPU DP	$B\Psi_A$	Ψ_P	Ψ_P	$\mathcal{O}(\log(N))$	N	(DP)
+ Cyclic	$B\Psi_A$	Ψ_P	Ψ_P	$\boldsymbol{\mathcal{O}(1)}$	N	(CDP)
DP with MP	$B\Psi_A/N$	Ψ_P/N	$\Psi_P+B\Psi_A^{\text{int}}$	$\mathcal{O}(\log(N))$	N^2	(DP)
+ Cyclic	$B\Psi_A/N$	Ψ_P/N	$\boldsymbol{\frac{\Psi_P}{2}+B\Psi_A^{\text{int}}}$	$\boldsymbol{\mathcal{O}(1)}$	$\boldsymbol{\frac{N+1}{2}N}$	(CDP)
PP	$B\Psi_A$	Ψ_P/N	$B\Psi_A^{\text{int}}$	$\mathcal{O}(1)$	N	(CDP)
ZeRO-DP	$B\Psi_A$	Ψ_P/N	Ψ_P	$\mathcal{O}(\log(N))$	N	(DP)
+ Cyclic	$B\Psi_A$	Ψ_P/N	Ψ_P	$\boldsymbol{\mathcal{O}(1)}$	N	(CDP)

Table 2: **Top-1 test accuracy for the 3 learning rules** (CDP-v1), (CDP-v2) and (DP) on the CIFAR-10 and ImageNet datasets, by training a ResNet-18 or 50. Standard deviation over 5 runs is less than 0.08, indicating stable results. CDP-v2 in particular performs similarly to or better than DP.

	CIFAR-10			ImageNet		
Model	(CDP-v1)	(CDP-v2)	(DP)	(CDP-v1)	(CDP-v2)	(DP)
ResNet-18	94.1	**94.8**	94.7	69.9	70.0	**70.1**
ResNet-50	94.0	**94.5**	**94.5**	**75.8**	75.7	75.4

Activation memory tracking. To confirm the memory savings that CDP would provide, we track the memory used in a forward-backward pass of a ResNet-50 and a ViT-B/16 on ImageNet, to infer the mean memory usage per worker of DP and CDP. As predicted, the activation memory in CDP flattens as the number of workers increases, to 30% for the ResNet-50 and 42% ($= \frac{3.9-2.3\text{ GB}}{3.9\text{ GB}}$) for the ViT-B/16 of the memory used in DP. The ratio is close to theoretical having for the ViT, but lower for the ResNet due to the heterogeneity of its layers. More details are in the Appendix.

5. Conclusion

We introduced Cyclic Data Parallelism (CDP), an alternative paradigm to Data Parallelism. By executing the forward and backward passes of micro-batches of data cyclically rather than simultaneously, we balance gradient communication and the total memory occupied by activations. We particularize CDP in the context of Data, Model, and Pipeline Parallelism as well as ZeRO-DP, and demonstrate improvements in the total memory required to store activations or in the number of communication steps required between time steps during training. Our results are supported by existing theoretical guarantees and our numerical experiments. In future work, we would like to release an efficient implementation for the MP and ZeRO-DP frameworks.

References

[1] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Decoupled greedy learning of cnns, 2020.

[2] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: Theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19:1749–1783, 09 2007. doi: 10.1002/cpe.v19:13.

[3] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839*, 2018.

[4] Yangrui Chen, Cong Xie, Meng Ma, Juncheng Gu, Yanghua Peng, Haibin Lin, Chuan Wu, and Yibo Zhu. Sapipe: Staleness-aware pipeline for data parallel dnn training. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 17981–17993. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/725ce5f2b1a8e2e0ac66994e7fefe375-Paper-Conference.pdf.

[5] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021. doi: 10.1109/MM.2021.3061394.

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[8] Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Quentin Lhoest, Anton Sinitsin, Dmitry Popov, Dmitriy Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, Denis Mazur, Ilia Kobelev, Yacine Jernite, Thomas Wolf, and Gennady Pekhimenko. Distributed deep learning in open collaborations. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=FYHktcK-7v.

[9] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[10] Louis Fournier, Stephane Rivaud, Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Can forward gradient match backpropagation? In Andreas Krause, Emma Brunskill,

Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10249–10264. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/fournier23a.html.

[11] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, page 1223–1231, Red Hook, NY, USA, 2013. Curran Associates Inc.

[12] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[14] Arpan Jain, Ammar Ahmad Awan, Asmaa M. Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G. Anthony, Hari Subramoni, Dhableswar K. Panda, Raghu Machiraju, and Anil Parwani. Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020. doi: 10.1109/SC41405.2020.00049.

[15] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[16] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. URL https://api.semanticscholar.org/CorpusID:18268744.

[17] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, USA, 2014. USENIX Association. ISBN 9781931971164.

[18] Konstantin Mishchenko, Francis Bach, Mathieu Even, and Blake Woodworth. Asynchronous sgd beats minibatch sgd under arbitrary delays, 2023.

[19] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[20] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[21] Arild Nøkland and Lars Hiller Eidnes. Training neural networks with local error signals, 2019.

[22] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2008.09.002. URL https://www.sciencedirect.com/science/article/pii/S0743731508001767.

[23] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[24] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning, 2021.

[25] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL https://proceedings.neurips.cc/paper_files/paper/2011/file/218a0aefd1d1a4be65601cc6ddc1520e-Paper.pdf.

[26] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021.

[27] Herbert E. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951. URL https://api.semanticscholar.org/CorpusID:16945044.

[28] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. Swarm parallelism: Training large models can be surprisingly communication-efficient, 2023. URL https://arxiv.org/abs/2301.11913.

[29] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers, 2018.

[30] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[31] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[32] Sebastian U. Stich. Local sgd converges fast and communicates little, 2019.

[33] Sebastian U. Stich and Sai Praneeth Karimireddy. The error-feedback framework: Sgd with delayed gradients. *Journal of Machine Learning Research*, 21(237):1–36, 2020. URL http://jmlr.org/papers/v21/19-748.html.

[34] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. Zero++: Extremely efficient collective communication for giant model training, 2023.

[35] Xiaofeng Wu, Jia Rao, and Wei Chen. Atom: Asynchronous training of massive models for deep learning in a decentralized environment, 2024. URL https://arxiv.org/abs/2403.10504.

[36] An Xu, Zhouyuan Huo, and Heng Huang. On the acceleration of deep learning model parallelism with staleness, 2022.

[37] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

[38] Haibo Yang, Xin Zhang, Prashant Khanduri, and Jia Liu. Anarchic federated learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 25331–25363. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/yang22r.html.

[39] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. Mics: Near-linear scaling for training gigantic model on public cloud, 2022.

[40] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.

[41] Huiping Zhuang, Zhenyu Weng, Fulin Luo, Toh Kar-Ann, Haizhou Li, and Zhiping Lin. Accumulated decoupled learning with gradient staleness mitigation for convolutional neural networks. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 12935–12944. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/zhuang21a.html.

Appendix A. Related work

Data Parallelism. For DP, the model is replicated on several workers, and each worker computes a micro-batch, a subset of the mini-batch. As noted above, this method suffers from a linear increase in memory as the number of workers increases, while communications costs increase either logarithmically or linearly [2]. To reduce communications, less strict synchronization steps have been introduced, at the cost of updating the parameters with stale gradients [17, 25, 32]. However, these techniques do not take into account the memory required to train DNNs, a point we improve on.

Two notable DP techniques aim to reduce the memory cost of DP: Zero Redundancy Optimizer powered DP (*ZeRO-DP*) [23] and *Fully-Sharded DP* [40]. Rather than fully replicating the entire model across the DP workers, each worker retains only a subset of the model states (i.e. optimizer states, gradients, and parameters) corresponding to one stage of the model. The memory required can be further reduced by storing model states on the CPU [24, 26], at the expense of communications between the CPU and the GPUs. Despite the memory gains, the volume of communication increases in ZeRO-DP, which has been addressed by compressing communications or striking a balance with the memory [34, 39]. Nevertheless, the simultaneous nature of the micro-batches executions remains the root cause of ZeRO-DP's issues, which we tackle in this paper.

Model Parallelism. Instead of parallelizing computations by slicing a mini-batch, MP [6, 29] aims to partition the components of the DNN. MP can be separated into two categories. First, Intra-layer MP [15, 30], also called tensor parallelism, partitions individual layers across workers but requires a high communication cost. Second, Inter-layer MP [6] splits a DNN into successive stages, and workers are solicited for only one stage, thus requiring less memory. A disadvantage of this method is that only one of the workers computes at a time since the stages are executed sequentially in the forward-backward pass. This limitation was addressed in several ways. Local learning [1, 10, 21] removes the need for a backward pass of the model with local auxiliary objectives at each stage. This allows stages to compute mini-batches simultaneously but at the expense of model performance. Delayed gradient methods [36, 41] allow stages to compute in parallel on micro-batches, by allowing updates with delayed gradients.

Of particular interest is *Pipeline Parallelism* (PP), a form of Inter-layer MP, introduced in GPipe [13], which also avoids this problem. In PP, mini-batches are divided into micro-batches which are passed sequentially to the successive stages. This allows each micro-batch to be propagated to the next stage as soon as it has completed its forward pass, allowing stages to compute micro-batches in parallel. GPipe thus reduces the number of idle workers, while still leaving a "bubble". PipeDream [19] proposes to reduce further reduce this bubble of computation by introducing the one-forward-one-backward (1F1B) pipeline schedule. Within this implementation, a stage alternates between the forward and the backward pass of one micro-batch. However, this comes at a cost as multiple versions of the parameters must be stored by a stage, and potentially storing the activations of the entire batch. Our method similarly introduces a cyclic delay of micro-batches. However, our framework is more general and notably applies to many other parallelism frameworks, not just PP. GEMS [14] is a MP framework that similarly reduces worker idleness by offsetting computations, like CDP and PP. Still, workers must store parameters of two stages, and GEMS does not extend to other frameworks either, unlike CDP. PipeDream-2BW [20] improves on PipeDream by keeping only two versions of the parameters and accumulating the gradients of the micro-batches to update them. This allows for continuous computations on each stage with no idle workers at the cost of a gradient

staleness of one mini-batch. We will show that our more general framework can recover and improve on the update rule of PipeDream-2BW when applied to the particular case of PP. PipeMare [37] keeps a single version of the weights and, like the previous delayed gradient methods, allows asynchronous updates. However, it requires additional hyperparameters, which we do not. DAPPLE [9] improves on PP by using stage replicas to handle micro-batches partitions. This can be combined with our CDP framework. Still, as in PP, GPUs need to store batch activations, not just micro-batches. Finally, note that the parallelization techniques we have discussed are often integrated together, with 3D Parallelism [31] being a notable example that combines DP, tensor parallelism, and PP.

Appendix B. Algorithm

In Alg. 1, we detail the execution of a worker in CDPv2, for a stage j on a micro-batch i, as well as its communications.

Appendix C. Detailled implementation of CDP in other parallelism frameworks

C.1. CDP implementation on a single-GPU device

We first explore the setting of a single-GPU device training with mini-batch SGD, assuming that communication within the GPU is cheap and fast, but memory is limited [5]. We illustrate the training of standard DP on a single-GPU device in Fig. 2a.i. Then, at the peak of the forward pass, the GPU must retain the activations of the entire model for N micro-batches, representing the total mini-batch, resulting in a total memory of activations of about $N\mathcal{B}\Psi_A$. Turning our attention to CDP, depicted in Fig. 2a.ii, we find that its implementation on a single-GPU device requires about half the memory compared to standard DP. Indeed, in this approach, each stage of the model processes one micro-batch at each time step. Consequently, the total memory occupied by activations remains nearly constant across time steps, approximately equal to $\frac{N+1}{2}\mathcal{B}\Psi_A$, reducing the total memory required for a DP implementation by half. Depending on the implementation, parameters may be shared on the GPU, resulting in no parameter memory improvement. The memory improvement is still significant in this case, as activation memory is generally much larger than parameter memory, peaking at 60GB for a GPT-2 model in [23], compared to 3GB for the fp16 parameters.

C.2. CDP implementation on a multi-GPU device

Here, one GPU processes a single micro-batch, but communication between GPUs may hinder the overall system efficiency. We consider the case where DP is implemented on N GPUs that process N micro-batches of data independently, as depicted in Fig. 2b.i. These GPUs communicate the gradients of the micro-batches with an all-reduce operation at the end of the training step before proceeding to the next time step. All-reduce is a collective operation that requires at least $\mathcal{O}(\log(N))$ communication steps [2] in a favorable setting, or $\mathcal{O}(N)$ steps for a bandwidth-optimal ring-based implementation [22]. CDP on N GPUs, as shown in Fig. 2b.ii, makes better use of GPU-to-GPU communication bandwidth by breaking the all-reduce operation across the training step into point-to-point operations between each time step. Indeed, half of the stages compute a backward pass at each time step, and then send the computed gradient to the next worker (modulo N) for the reduce operation. This communication scheme corresponds to the ring-based all-reduce operation [22], which has an optimal bandwidth usage.

Algorithm 1 Worker (i, j) in CDPv2, for a stage j on micro-batch i.

Input: Stage replica f_j, number of models/stages N, initial parameters θ_j^0, training steps T, dataset $\mathcal{D}$, optimizer OPT.
while $t < T$ **do**
  #### Forward pass
  # Propagate the new parameters
  $\theta_j \leftarrow$ **Wait and Receive** from worker $((i-1)\%N, j)$ if $i \neq (N+1-j)$ else (N, j)
  Send θ_j to worker $((i+1)\%N, j)$
  # Receive the activations
  if $j = 1$ **then**
    $x_j \leftarrow$ **Sample from dataset** $\mathcal{D}$
  else
    $x_j \leftarrow$ **Wait and Receive** from worker $(i, j-1)$
  # Compute the forward pass
  $x_{j+1} \leftarrow f_j(x_j, \theta_j)$
  # Propagate the activations
  if $j < N$ **then**
    Send x_{j+1} to worker $(i, j+1)$

  #### Backward pass
  # Receive the gradients
  if $i = 1$ **then**
    $\Delta_j \leftarrow 0$
  else
    $\Delta_j \leftarrow$ **Wait and Receive** from worker $(i-1, j)$
  if $j = N$ **then**
    $\delta_{j+1} \leftarrow \nabla\mathcal{L}(x_J)$
  else
    $\delta_{j+1} \leftarrow$ **Wait and Receive** from worker $(i, j+1)$
  # Compute the backward pass
  $\delta_j \leftarrow \frac{\partial f_j}{\partial x_j}^T \delta_{j+1}$
  $\Delta_j \leftarrow \Delta_j + \frac{1}{N}\frac{\partial f_j}{\partial \theta_j}^T \delta_{j+1}$
  # Propagate the gradients or the new parameters
  if j>1 **then**
    Send δ_j to worker $(i, j-1)$
  if i=N **then**
    $\theta_j \leftarrow \text{OPT}(\theta_j, \Delta_j)$
    Send θ_j to worker $(N+1-j, j)$
  else
    Send Δ_j to worker $(i+1, j)$

C.3. Implementation in a MP paradigm

Now, in MP, a GPU does not hold a full replica of a model but only of a single stage. During the forward and backward passes, activations and gradients must be communicated between GPUs holding successive stages. A naive implementation of DP and MP results in Fig. 2c.i, where the N micro-batches require N^2 GPUs to process each slice of the data and the model. Furthermore, note also that only N GPUs are busy at a time, so most GPUs are idle, waiting for the next micro-batch of data to pass through, which is a significant inefficiency of MP. Here, CDP with MP, which we present in Fig. 2c.ii, once again improves on the DP implementation. In particular, the number of GPUs required for CDP is reduced, and we can show that for each stage j, CDP requires only $N - j + 1$ GPUs to be shared among the N micro-batches. This leads to a "pyramidal" shape of the stage structure. Thus in total, only $\frac{N+1}{2}N$ GPUs are needed to hold a stage of the model, halving the number compared to DP, as well as the memory requirements. Compared to DP, a micro-batch doesn't send the activations of a stage to the same GPU each time for the computation of the next stage. Instead, a GPU in the previous time step will have released the activations stored in its memory after a backward pass. The micro-batch will send the activations to this GPU as its memory is available for computation. Since the number of devices is smaller in DP than in CDP, note also that the total number of gradient communications between devices is reduced. In fact, if a GPU can only able to hold the activation of one micro-batch, the number of GPUs used, $\frac{N+1}{2}N$, is optimal to compute N micro-batches. An in-depth discussion is proposed in Sec.E.

If we assume that one GPU can store the activations of all N micro-batches, then we can further reduce the total number of devices needed to only N, one per stage. Indeed, this implementation of CDP with MP on N devices, depicted in Fig. 2c.ii, is equivalent to PP as presented in PipeDream [19]. If we follow our first update rule Eq. (CDP-v1), CDP follows the same update rule as PipeDream-2BW [20]. However, our second update rule Eq. (CDP-v2) improves on this update rule by reducing the gradient delay. PP requires fewer GPUs than MP, but only if the GPUs can store the activations of the entire mini-batch. This limits its scaling capacity compared to MP, which is 2 times more GPU efficient. Indeed, for GPUs with similar memory capacities, MP with CDP requires $\frac{N+1}{2}N$ GPUs to train a batch of size $\mathcal{B}$. Meanwhile, PP requires N GPUs to train a batch of size $\frac{\mathcal{B}}{N}$, so, combined with DP, N^2 GPUs for a batch of size $\mathcal{B}$.

C.4. Implementation in a ZeRO-DP paradigm

ZeRO-DP [23] is a training framework that aims to combine the advantages of DP and MP, which we represent in Fig. 2d.i. Instead of replicating the entire model on each of the N GPUs, ZeRO-DP replicates only the model states of a single stage across the GPUs. The model state (in stage 3 of ZeRO-DP) refers to the model parameters, gradients, and optimizer states. When the workers execute the forward or backward propagation through a stage, the model states of that stage are broadcast from the GPU that stores them to all GPUs. After the computation, the model states are deleted to free up the memory, so that a GPU only retains the model states of a maximum of two stages. The communication volume is similar to standard multi-GPU DP, with a maximum increase of 1.5. Note, however, that as with PP, the memory occupied by the activations on one GPU increases with N, since all stages of the model must be retained. CDP improves on ZeRO-DP by eliminating the need for collective communication of the model states, as shown in Fig. 2d.ii. Since one stage is computed by a single GPU at a time step, the model states need only to be held by that GPU, without

replication. Then, they only need to be communicated to a single GPU at the next time step. This reduces the communication overhead at each time step, similar to Multi-GPU DP.

Appendix D. Implementation details

Framework. To test our method, we propose to use the standard training pipeline on the CIFAR-10 [16] and ImageNet [7] datasets, trained on the ResNet-18 and ResNet-50 architectures, split into 4 stages with similar FLOPs. We simulate our delayed gradients to train with SGD following DP, CDP-v1, and CDP-v2. For ImageNet, following standard practice, we report the maximum validation accuracy over the last 10 epochs. Contrary to [37], we did not need any specific tuning of the other hyperparameters for our update rules. Our source code is provided with this submission and will be released at the time of publication.

Hyperparameters and FLOPs partition. We simulate the partitioning of the models into equal FLOPs stages using the `fvcore` library github.com/facebookresearch/fvcore/ to compute the FLOPs count of each module of the ResNet. For finer partitioning of the linear modules, we separate them into weight and bias modules by approximating the bias module's FLOPs as the square root of the linear module's FLOPs. For our training, we consider the SGD optimizer with momentum 0.9 and we simulate our delayed activations for DP, CDP-v1, and CDP-v2. We train over 100 epochs with batch size 128 and 90 epochs with batch size 256 respectively for CIFAR-10 and ImageNet. We consider an initial learning rate of 0.05 and 0.1, decreasing by a factor of 0.2 and 0.1 at epochs 30, 60, and 90 for CIFAR-10 and ImageNet, respectively. The weight decay is 10^{-4} for ImageNet and 5×10^{-4} or 10^{-3} for CIFAR-10, whether trained on a ResNet-18 or 50. To account for the smaller image size of CIFAR-10, we remove the first max pooling and reduce the kernel size of the first convolutional layer to 3 and the stride to 1. We need 8 A100 GPUs for 6 hours for our training runs on ImageNet.

Train loss curve. In Fig. 3, we report the training loss curve for the three learning rules while training a ResNet-50 on ImageNet. We observe that the effect of the delay, mostly visible for CDPv1, mainly affects the start of training.

Memory tracking details In Fig. 4, we report the mean memory extrapolated for the ResNet-50 and ViT-B/16, for $N = 4$, 8, and 32. As discussed, the ResNet reaches a lower memory improvement due to the heterogeneity of its layers, which require varying activation memory for the same execution time (as feature size decreases through depth). This is not an issue for a Transformer-based model, as feature size stays constant across depth. Although a heterogeneous environment will reduce this improvement, this confirms that CDP will significantly improve the memory used in real implementations when using homogeneous stages and workers.

Appendix E. Optimality of Model Parallelism with Cyclic Data Parallelism

We extend the discussion on the implementation of MP with CDP, specifically on the number of GPUs required. First, the following lemma shows the number of time steps required for a GPU to process one micro-batch.

Lemma 1 *A GPU processing one micro-batch for the stage j is occupied (either computing or awaiting the backward pass while storing activations) for $2(N - j + 1)$ time steps.*

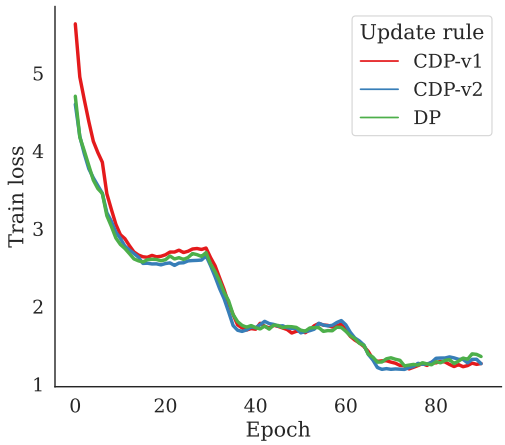

Figure 3: **Training loss of a ResNet-50 trained on ImageNet** following the learning rules (DP), (CDP-v1) and (CDP-v2). Values are averaged over a window of 7 epochs for the sake of readability. The loss of CDP-v1 is significantly higher at the beginning of training, which is not the case for CDP-v2. As parameters converge, the effect of the delay disappears and the three losses show a similar training curve, with a small advantage for both CDP-v1 and CDP-v2. This confirms that the delay in our update rules does not affect convergence, even in realistic settings.

Proof The final stage requires 2 time steps to compute a forward and a backward pass. Similarly, each stage takes 2 time steps to compute a forward and a backward pass and must also wait for the next stage to finish its execution. Thus, by recurrence, a GPU computing for the stage j will be occupied (for computation or activation retaining) for $2(N-j+1)$ time steps. ■

We will now show that MP with CDP can be implemented with only $N-j+1$ GPUs per stage j by showing that a GPU will always be available when a micro-batch is required to perform a forward propagation on the stage j.

Proposition 2 *MP with CDP can be implemented with $N-j+1$ GPUs for each stage $j \in [1, N]$.*

Proof Consider a worker n that computed the forward pass at stage $j-1$ and requires a GPU at stage j. The worker $n-1$ required a GPU at stage j, 2 time steps earlier. Denote this GPU, for instance, as the first GPU of stage j. The worker $n-2$ required a GPU at stage j, 4 time steps earlier, the second GPU of stage j. The final GPU at stage j, the $(N-j+1)$th, thus was required by a worker $2(N-j+1)$ time steps earlier. Since we showed in Lemma 1 that a worker occupies a GPU at stage j for $2(N-j+1)$ time steps, this proves that this GPU is now available for worker n to use. ■

Thus we have shown that MP with CDP requires $N-j+1$ GPUs per stage, or when summing all stages, a total of $\frac{1}{2}(N+1)N$ GPUs. The following result shows that this number of GPUs is optimal if we consider that a GPU can only hold the activations of one micro-batch. In other words, if a GPU is occupied either while computing or storing the activations of a micro-batch.

Proposition 3 *If a GPU is occupied when it is either computing or retaining the activations of a micro-batch, then MP requires a minimum of $\frac{N(N+1)}{2}$ GPUs to be occupied at all time steps.*

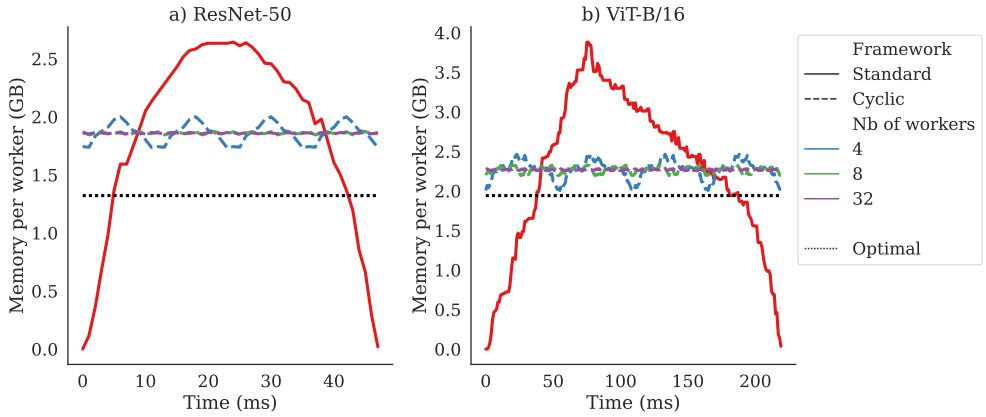

Figure 4: **Activation memory per worker, when training with N workers on ImageNet** with an efficient implementation of DP (full) and a CDP (dashed), on a ResNet-50 and a ViT-B/16. An optimal halving of the parameters is represented in black ('Optimal'). The memory required by a forward-backward pass for one work is first tracked, and parameter memory is removed. The figure is extrapolated by mimicking the total memory used by N workers training on DP (i.e. simultaneously) or CDP (i.e. cyclically), and dividing by N. As N increases, the memory required by CDP flattens, to a value lower than DP. This value reaches close to the reduction in half theorized for the ViT-B/16, with 42%. The heterogeneity of the layers of the ResNet reduces the effectiveness of CDP, only reaching 30% reduction.

Proof For one micro-batch, each stage j requires a GPU to be occupied during $2(N-j+1)$ time steps (see Lemma 1). Thus, N micro-batches require devices to be occupied during $2N(N-j+1)$ time steps. Since a training step, in which N micro-batches are processed, takes $2N$ time steps, a GPU occupied at all time steps will be occupied $2N$ time steps. Thus, a stage requires at least $\frac{2N(N-j+1)}{2N} = N-j+1$ devices occupied at all time steps to process N micro-batches. Summing over all stages gives the result. ■