Lion Cub: Minimizing Communication Overhead in Distributed Lion

Satoki Ishikawa [1] **Tal Ben-Nun** [2] **Brian Van Essen** [2] **Rio Yokota** [1] **Nikoli Dryden** [2]

Abstract

Communication overhead is a key challenge in distributed deep learning, especially on lower-bandwidth interconnects, and given current hardware trends, communication is likely to become a major bottleneck. While gradient compression techniques have been explored for SGD and Adam, the Lion optimizer has the distinct advantage that its update vectors are the output of a sign operation, enabling straightforward quantization. However, simply compressing updates for communication and using techniques like majority voting fails to lead to end-to-end speedups due to inefficient communication algorithms and reduced convergence. We analyze three factors critical to distributed learning with Lion: optimizing communication methods, identifying effective quantization methods, and assessing the necessity of momentum synchronization. Our findings show that quantization techniques adapted to Lion and selective momentum synchronization can reduce communication costs while maintaining convergence. We combine these into Lion Cub, which enables speedups in training compared to Lion especially with low-bandwidth environments. This highlights Lion's potential as a communication-efficient solution for distributed training.

1. Introduction

Recent progress in deep learning has largely been driven by improvements in computational power (Kaplan et al., 2020; Hoffmann et al., 2022). Training large-scale models on distributed systems using hundreds or thousands of GPUs has become standard practice. This shift has increased the demand for scalable and communication-efficient distributed training algorithms capable of handling large-scale systems. Thus, a key challenge to train large-scale models is minimizing communication overhead, which frequently limits the scalability of distributed training.

To date, communication has typically been optimized via efficient allreduce algorithms and frameworks (e.g., Thakur et al., 2005; NVIDIA, 2024; Cai et al., 2021), communication/computation overlap, or through improved network hardware and increased numbers of NICs per compute node. However, current trends show computational throughput growing much faster than communication bandwidth. For example, the InfiniBand roadmap indicates 2× bandwidth growth every four years, whereas Nvidia has delivered 2–3× flop increases every two years (InfiniBand, 2024; Morgan, 2024). While technologies such as NVLink allow for higher-bandwidth cliques, these do not scale to entire clusters. Further, the growth in model size far outpaces the growth in GPU memory capacity, requiring smaller per-GPU batch sizes. As communication volume is proportional to the model size and computation to the batch size, this leads to further imbalance. The imbalance is even larger in loosely coupled grid systems, where workers may be separated by wide-area links, and in edge or collaborative learning settings that rely on wireless channels with modest throughput and variable latency (Zhang et al., 2020b; Diskin et al., 2021; Ryabinin et al., 2021; Hilmkil et al., 2021; Borzunov et al., 2022). Consequently, the long-term trend is toward communication costs dominating computation.

Many algorithmic approaches have been developed to reduce communication volume, typically via gradient compression (Tang et al., 2020) in the context of SGD. These include sign-based algorithms (e.g., Seide et al., 2014; Bernstein et al., 2018; 2019) that communicate only gradient sign; sparsification (e.g., Ström, 2015; Dryden et al., 2016; Lin et al., 2018; Renggli et al., 2019); and randomized compression operators (e.g., Alistarh et al., 2017; Wen et al., 2017; Wang et al., 2018). Similar ideas have been applied to more complex optimizers, such as Adam or LAMB (Tang et al., 2021; Li et al., 2022; Lu et al., 2023). However, these methods often slow down convergence due to information loss during compression, although recent papers have mitigated this effect.

In contrast to these approaches, we focus on the Lion optimizer (Chen et al., 2024), which directly applies the sign operator to its updates while maintaining performance com-

[*]Equal contribution [1]Institute of Science Tokyo [2]Lawrence Livermore National Laboratory. Correspondence to: Satoki Ishikawa <ishikawa@rio.scrc.iir.isct.ac.jp>.

42nd International Conference on Machine Learning Workshop TTODLer-FM. Copyright 2025 by the author(s).

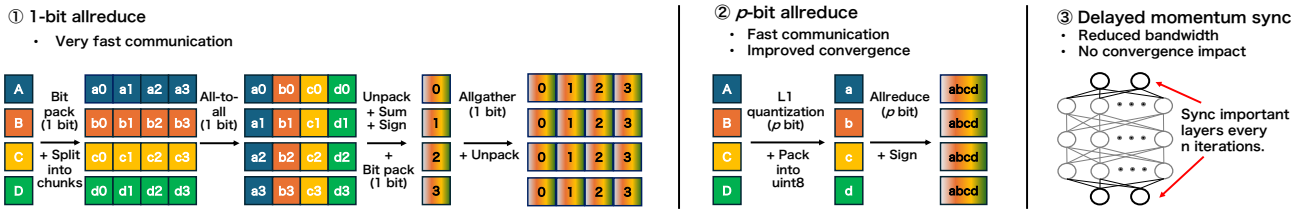


Figure 1: **Overview of our communication algorithm.** We propose two synchronization methods for Lion's update vector: 1-bit allreduce and p-bit allreduce. 1-bit allreduce provides fast communication but may compromise convergence. Additionally, the frequent packing and unpacking of 1-bit data can introduce overhead. In contrast, p-bit allreduce requires more communication time than 1-bit allreduce but typically converges more quickly. As a result, p-bit allreduce can sometimes achieve faster convergence of the training loss relative to training time. Synchronizing momentum can also help maintain convergence stability.

parable to AdamW on a wide range of tasks. Hence, Lion is a promising candidate for gradient compression to reduce communication overhead. Recently, Liu et al. (2024) analyzed distributed Lion and theoretically demonstrated that it converges with a "majority vote" update rule, and validated this experimentally. However, their communication implementation is based on parameter servers, which do not scale, and they do not demonstrate that distributed Lion can achieve a real-time speedup for training.

Figure 1 is an overview of our key contributions. We first study the communication primitives required for distributed Lion, and implement a 1-bit allreduce similar to that of Tang et al. (2021), which significantly reduces bandwidth requirements. We then analyze the distribution of gradient updates across layers in Lion and the impact of the 1-bit allreduce, and identify key limitations of the 1-bit representation, including occasional loss spikes during training. To remedy this, we develop a novel L1 quantization method for Lion, which can be efficiently implemented with a p-bit allreduce while significantly improving convergence. We also study the impact of distributed and compressed communication on momentum in Lion, and find that local estimates of momentum are sufficient except for certain layers, which may require occasional momentum synchronization to ensure convergence. Finally, we experimentally evaluate our implementation and show that distributed Lion can deliver improved wall-time training on ResNet and transformer models without compromising model quality.

We summarize our contributions as follows:

- We propose optimized communication primitives to implement both 1-bit and p-bit allreduces specialized for distributed Lion, which outperform parameter servers.
- We analyze the distribution of gradient updates in Lion, and show that standard quantization methods are ineffective due to outliers. We propose an L1 quantization method which offers superior performance while still enabling efficient communication.
- We analyze the necessity of momentum synchronization in Lion and find that it can often be eliminated or performed only periodically on key layers.
- We experimentally evaluate our distributed Lion implementation and improve training wall-time by up to 5.1× without impacting quality for large models.

2. Background and Related Work

Lion The Lion optimizer (Chen et al., 2024) is a recent memory-efficient optimizer with performance comparable to AdamW. It has the following update rule:

$$m_{t+1} = \beta_2 m_t - (1-\beta_2)\nabla f(\theta_t)$$
$$\theta_{t+1} = \theta_t + \eta_t \left(\operatorname{sign}\left(\beta_1 m_t - (1-\beta_1)\nabla f(\theta_t)\right)\right),$$

where η_t is the learning rate, β_1 and β_2 are hyperparameters , and m_t the momentum. When $\beta_1 = \beta_2$, Lion matches sign descent. However, when $\beta_2 > \beta_1$, the importance of the current gradient ∇f_t is increased compared to sign descent with momentum (Liu et al., 2024). Recently, Liu et al. (2024) proposed Distributed Lion, which combines Lion with majority voting, and demonstrated its convergence both theoretically and experimentally.

1-bit Communication There has also been research aimed at reducing communication costs by applying 1-bit communication to optimizers other than SGD. 1-bit Adam and 1-bit LAMB (Tang et al., 2021; Li et al., 2022) propose methods for compressing update (momentum) vectors for Adam and LAMB. As these optimizers do not use a sign or similar operation, directly compressing the updates results in accuracy degradation, and incorporate methods such as error feedback (Seide et al., 2014) to compensate. Other systems, such as QSGD (Alistarh et al., 2017), have generalized 1-bit communication to arbitrary bitwidths.

Quantization When quantizing to n bits, it is essential to ensure the full range of the representation is used. This is typically achieved by normalizing and rescaling the input data by the absolute maximum of the input elements, e.g.: $Q_\infty(x) = \operatorname{sround}\left(\frac{2^{n-1}-1}{\|x\|_\infty}x\right)$ where $\|\cdot\|_\infty$ is the maximum norm and sround is a stochastic rounding operator (Al-

Implementation	Latency cost	Bandwidth cost
Parameter server (naïve)	$2(P-1)\,\alpha$	$2P\,N\,b\,\beta$
Parameter server (efficient)	$2\log_2(P)\,\alpha$	$3\frac{P-1}{P}N\,b\,\beta$
Direct allreduce	$2\log_2(P)\,\alpha$	$2\frac{P-1}{P}N(\log_2(P)+1)\,\beta$
1-bit compressed allreduce	$(P-1+\log_2(P))\,\alpha$	$(1+\frac{P-1}{P})N\,\beta$

Table 1: Comparison of communication bandwidth costs for implementing majority voting with P workers, N parameters, b bits per standard word, latency α, and inverse bandwidth β (s/bit).

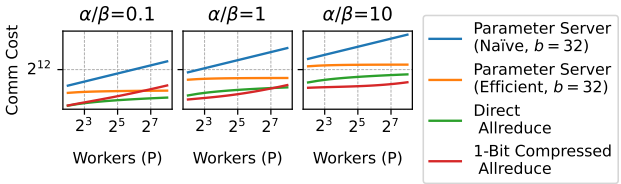


Figure 2: **Communication costs with our performance model.** We illustrate the communication costs from Table 1 for different cases of relative latency and bandwidth.

istarh et al., 2017). This simple quantization method performs well with a uniform distribution, but frequently fails when applied to data following other distributions (Dettmers et al., 2022; 2023). Empirical evidence shows that gradients in deep learning frequently follow a Laplace or lognormal distribution, leading to methods specifically tailored to these distributions (e.g., Yu et al., 2020; Chmiel et al., 2021) but which are computationally intensive.

3. Communication Design

The communication for majority voting can be implemented in several ways. We compare three implementations: parameter servers, a direct allreduce, and a variant of a 1-bit compressed allreduce. We use the α-β model, where α is the latency in seconds and β the inverse bandwidth in seconds per bit. Let there be P workers and N parameters, and assume there are b bits in a word. See, e.g., Thakur et al. (2005); Chan et al. (2007) for derivations of the communication costs we use. We focus on bandwidth in the text and summarize both latency and bandwidth terms in Table 1. For simplicity, we ignore computational overheads.

In a parameter server model, as assumed by distributed Lion (Liu et al., 2024), each worker sends its update to a centralized parameter server via a gather operation. The server then aggregates the updates and broadcasts the final update to the workers. A naïve implementation has each worker send to the server for cost $PNb\beta$. A similar implementation can be used for the broadcast, again requiring $PNb\beta$, for a total cost of $2PNb\beta$. A more efficient implementation of the gather and broadcast could instead be used, with cost $3\frac{P-1}{P}Nb\beta$. However, allreduce-based approaches are more efficient, and further, do not risk communication being bottlenecked at a single server.

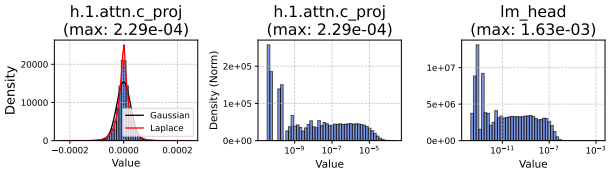


Figure 3: **The distribution of the update vector is not uniform.** The distribution represents the values of the update vector before quantization (sign). Left: The values do not follow a uniform distribution; rather, they approximate a Laplace distribution (not a Gaussian distribution). Right: The norm of the update vector is heavy-tailed, with the maximum significantly exceeding the mean.

In a direct allreduce implementation, we first convert signs to 0/1 (forcing 0 to be one or the other; see the sequel for a discussion) and then performing a standard allreduce with summation to obtain the majority vote. As the sum is between 0 and P, we require $\log_2(P)+1$ bits per parameter. This allreduce requires time $2\frac{P-1}{P}N(\log_2(P)+1)\beta$, a savings of $\frac{b}{\log_2(P)+1}$ by using a more concise representation. In practice, we typically use 8-bit words and pack entries.

Lastly, we can adapt the 1-bit compressed allreduce of Tang et al. (2021) to support majority voting. The update vector is first represented to 1 bit (again forcing 0 to be either -1 or 1). This is then split into P equal chunks, and the ith chunk is sent to the ith worker via an all-to-all communication. Each worker locally sums the contributions, takes the sign, and recompresses the result to 1 bit. Finally, an allgather is performed to disseminate the complete update to every worker. (See Figure 1 for an illustration.) We assume the all-to-all is implemented as a pairwise exchange with cost $N\beta$, and the allgather has cost $\frac{P-1}{P}N\beta$, yielding the lowest bandwidth cost, although care must be taken to efficiently implement the intermediate packing and unpacking.

Figure 2 provides an illustrative comparison of these algorithms for different scales and latency/bandwidth ratios. The naïve parameter server scales poorly in all situations. The direct allreduce offers the best performance in high-latency settings and at large scale, where the cost of the all-to-all makes the 1-bit compressed allreduce less efficient. Conversely, in bandwidth-bound regimes, the lower data transfer requirements of the 1-bit allreduce are more advantageous.

4. Quantization Design

4.1. The distribution of update vectors

The commonly used quantization function $Q_\infty(x)$ is information theoretically optimal when x follows a uniform distribution. However, it can introduce significant error when the distribution of x deviates too far from uniform. In particular, it is sensitive to outliers: when the maximum value is excessively large, most values are quantized to zero.

Table 2: **Validation loss on the Fineweb dataset with different quantization function.**

Optim	32 bit	1 bit	8 bit (L_0)	8 bit (L_∞)	8 bit (L_1)
Loss	2.94	3.22	3.07	4.64	2.97

Figure 3 illustrates the distribution of the update vector. This distribution is not uniform and is more closely aligned with a Laplace distribution than a Gaussian distribution. Consequently, when standard stochastic quantization is applied, the presence of outlier values results in most values being quantized to zero. In fact, for the last layer (lm_head), the values of the update vector mostly range from 1×10^{-12} to 1×10^{-7}. However, due to the maximum value of 1.63×10^{-3}, quantizing with 5 bits leads to any value below 1×10^{-4} being represented as zero, thus most values are quantized into 0. This indicates the limitation of applying standard stochastic quantization to Lion.

4.2. L1 Quantization algorithm

We consider a new type of quantization function that normalizes by the L_p norm instead of the L_∞ norm, making it robust against outliers:

$$Q_p(x)_i = \text{clamp}\left(\text{round}\left(\frac{2^{n-1}-1}{2M_p(x)}\, x_i\right),\, 2^{n-1}-1\right), \quad (1)$$

where $M_p(x) = \left(\frac{1}{d}\sum_{j=1}^{d} |x_j|^p\right)^{1/p}$ and $\text{clamp}(\cdot, \alpha)$ is to truncate the values into $[-\alpha, \alpha]$. We call this quantization as L_p quantization where $p = \infty$ corresponds to the standard quantization methods. As shown in Table 2, when training GPT on the FineWeb dataset, Lion converges poorly under Q_∞ and 1-bit quantization, whereas L_0 and L_1 quantizations converge more effectively. Based on these findings, we will adopt Q_1 which uses the L_1 norm and is henceforth referred to as "L1 quantization" for all subsequent experiments. This quantization method is simple yet sufficiently effective for practical applications, as will be demonstrated later. Its simplicity also results in low computational overhead.

Figure 4 shows the accuracy of quantizing Lion updates with different methods in terms of whether the signs of the updates match or flip (introducing error). Standard Q_∞ stochastic quantization clips most values to zero, resulting in a low match and flip rate. One approach to address this ("w/o zero") is to stop rounding to zero and instead quantize sufficiently small values to $+1$ or -1. While this improves the match rate, it does not match distributed Lion. We also consider transforming values to a logarithmic scale before quantizing them, then reverting, which also does not match distributed Lion. In contrast, our L1 quantization significantly improves the sign match rate and reduces the flip rate compared to distributed Lion. We note that it may

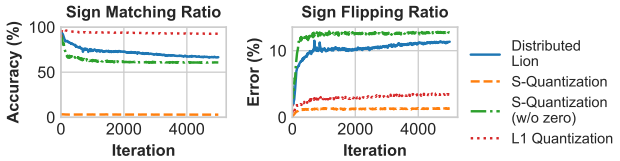


Figure 4: **L1 quantization approximates Lion well.** Top: Percent of update elements whose sign after quantization matches the standard Lion. Bottom: Percent of update elements whose sign after quantization reverses compared to standard Lion, changing the update direction. (This excludes elements mapped to 0.)

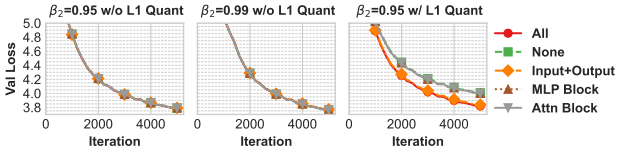


Figure 5: **(Left) Momentum synchronization or L1 quantization is necessary when β_2 is small.** Momentum is never synchronized for "None"; all other settings synchronize momentum every 10 iterations. When β_2 is large, gradients are stable across workers, and synchronization is not needed. When β_2 is small, synchronization is necessary for key layers. In addition, unlike standard distributed Lion, L1 quantization achieves convergence comparable to synchronized momentum, even without synchronization ("None").

be surprising that distributed Lion is able to converge when its sign matching rate is 60–70%, only a little better than random. However, most of its errors are from returning zero, and consequently, its sign flip rate is around 10%. Further, the parameters that result in zeros often exhibit significant inter-worker variability, suggesting the need for updates in these cases is inherently low.

5. On the necessity of momentum synchronization

In standard Lion, while the final update vectors are 1-bit, the momentum is computed with full-precision gradients. In contrast, in distributed Lion (Liu et al., 2024), only the 1-bit updates are communicated, and the momentum term cannot be recovered, which can lead to momentum divergence between workers. Further, communicating the momentum would add overhead during training. We now investigate the extent to which momentum diverges and how to efficiently mitigate any divergence.

5.1. Layerwise variance of momentum

Figure 5 compares multiple synchronization strategies for momentum: no synchronization, synchronizing every 10 iterations, and selective layer synchronization (only MLP or attention blocks). When β_2 is large (0.99), we see that training converges similarly with all strategies; hence momentum

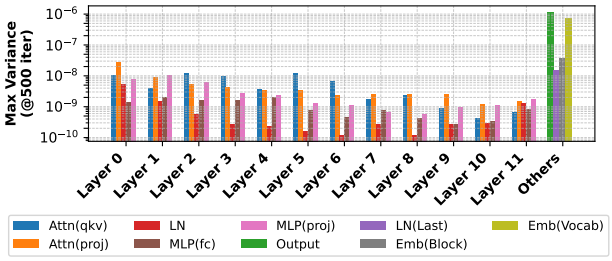


Figure 6: **There is significant variance between workers for the first (embedding) and last layers.** We plot the maximum standard deviation between workers for each layer, observed for each element, while training GPT (730M) on OpenWebText.

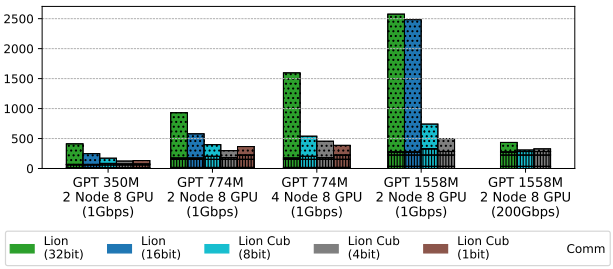


Figure 7: **Breakdown of runtime performance when training GPT.** Lion Cub significantly reduces communication volume, leading to reduced communication time and faster end-to-end iterations. "Comm" includes both communication time and quantization overheads.

synchronization is not required to maintain convergence. In contrast, with a smaller β_2 (0.95), convergence deteriorates without synchronization. Intuitively, this is due to a larger β_2 ensuring more stable gradients across iterations, which likely compensates for any differences in updates across workers. Figure 17 validates this, showing that momentum variance between workers decreases as β_2 increases. Hence, the need for momentum synchronization depends primarily on the value of β_2 rather than on specific synchronization timing during training.

We can also see from Figure 5 that synchronizing the momentum of only the input and final layers is sufficient to achieve convergence similar to that of synchronizing the momentum across all layers at $\beta_2 = 0.95$. Figure 6 compares the variance in momentum between workers, showing, for each layer, the maximum standard deviation measured for the element with the highest deviation across workers. The standard deviation for the input embedding layer and the output layer is nearly 100 times larger than that of other layers. This indicates that the momentum in these layers varies significantly between workers, whereas there is almost no variation in the other layers. Given that the input and output layers are the layers where data directly enters, this result may be expected, considering that the data differs across workers. In vision models, it is only the input layer that exhibits such a large standard deviation (see Figure 16). We leave a more detailed analysis of this to future research.

5.2. Impact of quantization on momentum variance

Our L1 quantization enables the communication of richer information than simple 1-bit updates, which we can also use to reduce the divergence in momentum between workers. Figure 5 shows the results of training with 8 workers using L1 quantization to compress updates to the range $[-15, 15]$. We are able to achieve similar convergence curves regardless of whether or not momentum is synchronized.

These findings suggest two approaches for good convergence: 1-bit communication with momentum synchronization or L1 quantization with no momentum synchronization. The best choice depends on the specific setting. As a general guideline, when there are few workers, using L1 quantization is recommended. However, with a larger number of workers, the increased bitwidths required to correctly represent the majority vote limit potential speedups in training and quantization information. For example, with 125 workers, a uint8 can can only support 0/1 summation and it becomes necessary to synchronize momentum occasionally.

6. Lion Cub

We have presented a set of methods for accelerating communication and maintaining convergence while using the Lion optimizer, including 1-bit allreduces, p-bit allreduces with L1 quantization, and delayed momentum synchronization. We refer to this collection of techniques for reducing Lion's communication as "Lion Cub", indicating a smaller, more communication-efficient version of Lion.

6.1. Performance results

We first consider the runtime performance of training. Figure 7 presents a breakdown of the runtime for training 350M, 774M, and 1558M parameter GPT models with standard Lion and Lion Cub in multiple precisions. As communication precision decreases, communication time also decreases, in line with expectations. This delivers up to 5.1× speedups to end-to-end training over standard Lion in the lower-bandwidth setting. Unfortunately, when training the 730M parameter model on 2 nodes, 1-bit Lion Cub in fact increases communication time; this is due to the overhead of packing and unpacking the compressed representation outweighing the decreased communication time. However, if we increase the number of nodes, this only increases the communication time, not the pack/unpack time, resulting in further communication reductions. As model size increases from 350M into 1558M, the communication bottleneck intensifies, yielding greater performance improvements for Lion Cub over standard Lion. Even in the higher-bandwidth

setting, Lion Cub reduces iteration time compared to standard Lion.

6.2. Training results

We now validate that Lion Cub preserves convergence and model quality. Figure 8 presents training curves for GPT (774M) trained on the OpenWebText dataset and Fineweb dataset with 8 workers on 2 nodes. Lion Cub (8-bit) maintains convergence comparable to the 32-bit Lion, while Lion Cub (4-bit) shows a decline in convergence. Consequently, Lion Cub (8-bit) with L1 quantization achieves faster real-time convergence. The convergence quality of Lion Cub (4-bit) can be recovered by synchronizing the momentum of the head layer every ten iterations, without significantly impacting training time. On the other hand, synchronizing momentum across all layers creates a communication bottleneck, which limits the speedup in real-time training performance. We trained on the Fineweb dataset with more aggressive hyperparameter settings than those used for the OpenWebText dataset. Under these conditions, LionCub(1-bit) failed to converge, forcing us to adopt LionCub(4-bit). Because this setup is highly sensitive to the numerical precision, compressing gradients from 32-bit to 16-bit (bfloat16) prior to communication (the 16-bit Lion) resulted in unstable training. Overall, Lion Cub, employing L1 quantization, provides substantial benefits in both training stability and communication speed.

We also evaluate Lion Cub in two additional contexts for vision tasks in Figure 9. The first task is training ResNet-18 on CIFAR100 for 100 epochs. Second, we fine-tune a DeiT-Tiny model, pretrained on ImageNet, on CIFAR100 for 20 epochs. In both cases, Lion Cub achieves comparable convergence to Lion in less overall time. Further, this is achieved even when not using L1 quantization or momentum synchronization. Hence, it is possible to lower the communication volume significantly to reduce overall training time in these cases.

7. Discussion and Conclusion

Limitations Distributed training methods, especially ones incorporating compression, are notoriously fickle, and their quality is often sensitive to the scale of the model and cluster being used. While we have demonstrated Lion Cub's strong performance at a relatively small scale, computational resource limitations have so far prevented confirmation of its benefits at larger scales. We also note that we have focused on communication in a data-parallel setting, as opposed to model-parallelism techniques. While we believe Lion Cub could be extended to these approaches, we leave this to future work.

Another significant limitation is that while our method performs very well in lower-bandwidth networks, the improvements are less dramatic in low-latency, high-bandwidth networks. Despite this, we believe that distributed training in lower-bandwidth environments remains essential. We expect communication reduction to become increasingly important in the future. As deep learning becomes democratized, many more people will be training models on clusters with relatively low-end interconnects and will be communication-bound without improved techniques like Lion Cub. But even on high-end clusters, trends in hardware indicate that the balance of communication and computation will continue to worsen as GPU performance improves faster than network bandwidth, necessitating reduced communication for scalability and efficiency.

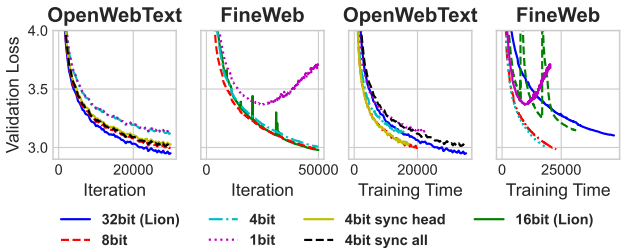


Figure 8: **Validation loss with GPT (774M)** when training with 8 workers on 2 nodes in our Ethernet configuration. Lion Cub reduces training time compared to standard Lion and can maintain comparable convergence. Note in 4-bit Lion Cub, workers only send 1/-1 values, whereas in 8-bit Lion Cub, we use L1 quantization and send values in $[-15, 15]$. In 4-bit, sync head Lion Cub, we synchronize the momentum of the final layer every 10 iterations. Sync all synchronizes the momentum of all layers.

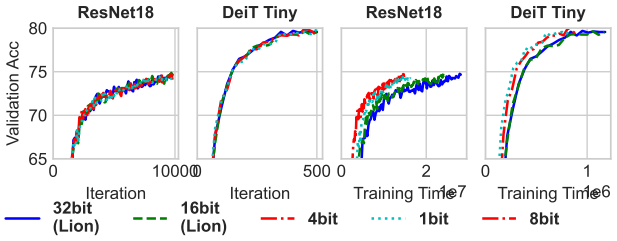


Figure 9: **Validation accuracy for training ResNet-18 on CIFAR100 and fine-tuning DeiT-Tiny on CIFAR100** using 8 workers on 2 nodes in our lower-bandwidth configuration. Interestingly, if you look at the training curve per iteration, even 1-bit allreduce closely mirrors Lion with 32-bit precision.

Conclusion We presented Lion Cub, a suite of improvements enabling reduced communication overhead for distributed training with Lion. Lion Cub incorporates optimized communication primitives and a novel L1 quantization method, enabling it to converge comparably to full-precision Lion on a variety of models, including ResNets and GPT, while reducing overall runtime.

References

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. QSGD: Communication-efficient sgd via gradient quantization and encoding. *NeurIPS*, 2017.

Bernstein, J., Wang, Y.-X., Azizzadenesheli, K., and Anandkumar, A. signsgd: Compressed optimisation for non-convex problems. In *ICML*. PMLR, 2018.

Bernstein, J., Zhao, J., Azizzadenesheli, K., and Anandkumar, A. signSGD with majority vote is communication efficient and fault tolerant. In *ICML*, 2019.

Borzunov, A., Baranchuk, D., Dettmers, T., Ryabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative inference and fine-tuning of large models. *arXiv:2209.01188*, 2022.

Cai, Z., Liu, Z., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., and Saarikivi, O. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

Chan, E., Heimlich, M., Purkayastha, A., and Van De Geijn, R. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13), 2007.

Chen, X., Liang, C., Huang, D., Real, E., Wang, K., Pham, H., Dong, X., Luong, T., Hsieh, C.-J., Lu, Y., et al. Symbolic discovery of optimization algorithms. *NeurIPS*, 2024.

Chen, Y., Sun, X., and Jin, Y. Communication-efficient federated deep learning with layerwise asynchronous model update and temporally weighted aggregation. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

Chmiel, B., Ben-Uri, L., Shkolnik, M., Hoffer, E., Banner, R., and Soudry, D. Neural gradients are near-lognormal: improved quantized and sparse training. In *ICLR*, 2021.

Crawshaw, M., Bao, Y., and Liu, M. EPISODE: Episodic gradient clipping with periodic resampled corrections for federated learning with heterogeneous data. In *The Eleventh International Conference on Learning Representations*, 2023.

Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. 8-bit optimizers via block-wise quantization. In *ICLR*, 2022.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. QLoRA: Efficient finetuning of quantized LLMs. In *NeurIPS*, 2023.

DeVries, T. Improved regularization of convolutional neural networks with cutout. *arXiv preprint*, 2017.

Diskin, M., Bukhtiyarov, A., Ryabinin, M., Saulnier, L., Sinitsin, A., Popov, D., Pyrkin, D. V., Kashirin, M., Borzunov, A., Villanova del Moral, A., et al. Distributed deep learning in open collaborations. *Advances in Neural Information Processing Systems*, 2021.

Dryden, N., Moon, T., Jacobs, S. A., and Van Essen, B. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, 2016.

Gorbunov, E., Danilova, M., and Gasnikov, A. Stochastic optimization with heavy-tailed noise via accelerated gradient clipping. *Advances in Neural Information Processing Systems*, 2020.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, 2016.

Hilmkil, A., Callh, S., Barbieri, M., Sütfeld, L. R., Zec, E. L., and Mogren, O. Scaling federated learning for fine-tuning of large language models. In *International Conference on Applications of Natural Language to Information Systems*. Springer, 2021.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models. In *NeurIPS*, 2022.

InfiniBand. InfiniBand roadmap, 2024. URL https://www.infinibandta.org/infiniband-roadmap/.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

Kunstner, F., Chen, J., Lavington, J. W., and Schmidt, M. Noise is not the main factor behind the gap between sgd and adam on transformers, but sign descent might be. In *The Eleventh International Conference on Learning Representations*, 2023.

Lee, S., Zhang, T., and Avestimehr, A. S. Layer-wise adaptive model aggregation for scalable federated learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023.

Li, C., Awan, A. A., Tang, H., Rajbhandari, S., and He, Y. 1-bit lamb: communication efficient large-scale large-batch training with lamb's convergence speed. In *HiPC*, 2022.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, B. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *ICLR*, 2018.

Liu, B., Wu, L., Chen, L., Liang, K., Zhu, J., Liang, C., Krishnamoorthi, R., and Liu, Q. Communication efficient distributed training with distributed lion. *arXiv preprint*, 2024.

Lu, Y., Li, C., Zhang, M., Sa, C. D., and He, Y. Maximizing communication efficiency for large-scale training via 0/1 adam. In *ICLR*, 2023.

Morgan, T. P. NVIDIA unfolds GPU, interconnect roadmaps out to 2027, 2024. URL https://www.nextplatform.com/2024/06/02/nvidia-unfolds-gpu-interconnect-roadmaps-out-to-2027/

Nishino, R. and Loomis, S. H. C. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *31st conference on neural information processing systems*, 151 (7), 2017.

NVIDIA. NVIDIA collective communication library, 2024. URL https://developer.nvidia.com/nccl.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Ren, Y., Cao, Y., Ye, C., and Cheng, X. Two-layer accumulated quantized compression for communication-efficient federated learning: Tlaqc. *Scientific Reports*, 2023.

Renggli, C., Ashkboos, S., Aghagolzadeh, M., Alistarh, D., and Hoefler, T. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

Ryabinin, M., Gorbunov, E., Plokhotnyuk, V., and Pekhimenko, G. Moshpit sgd: Communication-efficient decentralized training on heterogeneous unreliable devices. *Advances in Neural Information Processing Systems*, 2021.

Safaryan, M. and Richtárik, P. Stochastic sign descent methods: New algorithms and better theory. In *ICML*, 2021.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Interspeech*, 2014.

Ström, N. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

Tang, H., Gan, S., Awan, A. A., Rajbhandari, S., Li, C., Lian, X., Liu, J., Zhang, C., and He, Y. 1-bit adam: Communication efficient large-scale training with adam's convergence speed. In *ICML*, Proceedings of Machine Learning Research. PMLR, Jul 2021.

Tang, Z., Shi, S., Wang, W., Li, B., and Chu, X. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1), 2005.

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., and Jégou, H. Training data-efficient image transformers & distillation through attention. In *ICML*, 2021.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint*, 2023.

Wang, H., Sievert, S., Liu, S., Charles, Z., Papailiopoulos, D., and Wright, S. Atomo: Communication-efficient learning via atomic sparsification. *NeurIPS*, 2018.

Wang, H., Liu, X., Niu, J., Guo, W., and Tang, S. Why go full? elevating federated learning through partial network updates. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *NeurIPS*, 2017.

Xiang, M. and Su, L. β-stochastic sign SGD: A byzantine resilient and differentially private gradient compressor for federated learning. *openreview*, 2023.

Yan, G., Li, T., Xiao, Y., Hou, H., and Song, L. Improved quantization strategies for managing heavy-tailed gradients in distributed learning. *arXiv preprint arXiv:2402.01798*, 2024.

Yang, H., Qiu, P., and Liu, J. Taming fat-tailed ("heavier-tailed" with potentially infinite variance) noise in federated learning. *Advances in Neural Information Processing Systems*, 2022.

Yu, H., Wen, T., Cheng, G., Sun, J., Han, Q., and Shi, J. Low-bit quantization needs good distribution. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2020.

Zhang, J., Karimireddy, S. P., Veit, A., Kim, S., Reddi, S., Kumar, S., and Sra, S. Why are adaptive methods good for attention models? *Advances in Neural Information Processing Systems*, 2020a.

Zhang, Z., Chang, C., Lin, H., Wang, Y., Arora, R., and Jin, X. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, 2020b.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-TR-871246). T.B.N., B.V.E., and N.D. were funded by LLNL LDRD #24-SI-008. This work used computational resources TSUBAME4.0 supercomputer provided by Institute of Science Tokyo through the HPCI System Research Project (Project ID: hp240170) This work is supported by JST CREST Grant Number JPMJCR2112.

A. Extended Related work

A.1. SignSGD

SignSGD is an optimization method that applies the sign operator to gradients, resulting in three possible values: +1, 0, and -1 (Seide et al., 2014; Bernstein et al., 2018; 2019). This approach has gained attention due to its similarity to Adam, particularly in terms of the normalized update (Kunstner et al., 2023), but also as a distributed learning method when combined with majority voting, as updates can be compressed (Bernstein et al., 2019). In majority voting, sign is first applied to the gradients, and then again to their sum: $\theta_{t+1} = \theta_t - \eta_t \,\mathrm{sign}\left[\sum_{i=1}^{N} \mathrm{sign}\, \nabla_\theta f_i(\theta_{t-1})\right]$. Note that sign is unbiased, which is why a stochastic sign is sometimes employed (Safaryan & Richtárik, 2021). Typically, discussions of majority voting assume a parameter server model for communication, where the signs computed on each worker are sent to a central server which computes the global sign and returns the result. As we will discuss, while straightforward, this approach does not scale.

A.2. Heavy-tailed noise and Gradient Clipping

We can regard our L1 quantization as performing quantization after gradient clipping. In heavy-tailed centralized training, gradient clipping is known to be an effective approach to mitigate heavy-tailed noise (Gorbunov et al., 2020; Zhang et al., 2020a). In distributed optimization, several works have shown that clipping gradients before communication helps curb the influence of heavy-tailed noise (Yang et al., 2022; Xiang & Su, 2023; Yan et al., 2024). FAT-Clipping integrates gradient clipping into federated averaging, either once per round or at each local step, to tame fat-tailed noise and guarantee convergence (Yang et al., 2022). Similarly, EPISODE applies episodic gradient clipping, which uses the global gradient norm at the start of each round to decide whether to clip local updates, to adaptively bound heavy-tailed stochastic noise and ensure robust convergence under data heterogeneity (Crawshaw et al., 2023). Furthermore, TQSGD first applies gradient clipping and then quantizes the clipped gradients for communication, thereby compensating for heavy-tailed noise, and is the method most closely related to our L1 quantization (Yan et al., 2024). In contrast to these methods, L1 quantization automatically performs both scaling and clipping without any additional hyperparameters, and further combines this mechanism with the Lion optimizer. Although we do not provide formal convergence guarantees, our empirical evaluation offers a practical demonstration of the importance of clipping before quantization in large-scale training.

A.3. Layerwise Selective Communication

We synchronize the momentum of selected layers every few iterations. Although no prior work has performed layer-wise selective momentum synchronization, federated learning has long cut communication costs by transmitting only certain layers' weights or gradients more frequently or with higher precision. TWAFL is an asynchronous federated learning scheme in which the parameters of deep layers are updated less often than those of shallow layers (Chen et al., 2020). Likewise, FedLAMA reduces communication overhead by dynamically choosing each layer's optimal aggregation interval by evaluating model discrepancy and communication costs (Lee et al., 2023). TLAQC focuses on quantizing and communicating only two representative layers of the model, while all other layers merely accumulate updates locally (Ren et al., 2023). In FedPart, each client selectively updates and transmits only a subset of the network (for example, just the feature-extraction layers or only the classification layers) each round (Wang et al., 2024). Thus, in the federated-learning, it has long been known that different layers exhibit different degrees of importance to synchronize. Although our approach keeps weights and gradients fully synchronized across all GPUs, letting only the momentum terms diverge, we have found that the amount of momentum synchronization required varies from layer to layer.

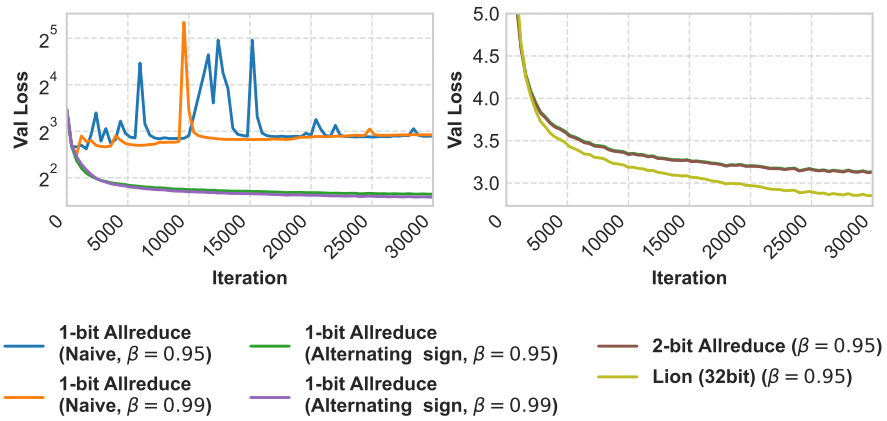


Figure 10: **1-bit allreduces, unable to represent exact zeros, diverge without adjustments.** The naive implementation of Lion using a 1-bit allreduce, which does not transmit exact zeros, leads to error accumulation and causes the loss to diverge. However, when zeros are converted to 1 or -1 on alternating iterations, the loss remains stable, regardless of the β_2 value.

		sign(majority vote)		
		1	**-1**	**0**
Standard Lion (FP32)	**1**	29.15%/32.08%	2.26%/4.34%	18.59%/13.54%
	-1	2.26%/4.34%	29.14%/32.13%	18.59%/13.56%
	0	0%/0%	0%/0%	0%/0%

Table 3: **Update values under majority vote vs standard Lion.** This shows the percent of update values with each sign under majority voting versus standard Lion from the 3000th iteration of training a GPT (730M) model.(Entries are for 4 / 8 workers.)

B. Communication Design

B.1. The impact of 1-bit allreduces

In both allreduce implementations, all updates are compressed to 1 bit. However, this loses information, since the sign operator is ternary and we no longer represent values which are exactly 0. We next examine the impact this has on training, focusing on the 1-bit compressed allreduce, which performs all communication in 1-bit precision.

The first allreduce stage uses an all-to-all to transmit update vectors. True zeros are rare in these, as a gradient must be consistently zero over many steps due to the momentum. One situation where this does occur is when training transformers, where certain parameters in the embedding layer could output zeros if the sequence length is always shorter than the model's maximum supported sequence length. The error introduced by 1-bit compression can accumulate over time in these parameters. However, this situation can be easily identified in advance and the updates masked.

In the second stage, an allgather disseminates aggregated updates, also with 1 bit. This is more challenging, because, in the typical case when there are an even number of workers, there can be an equal number of 1 and -1 updates for a parameter among the workers. Should this occur, the conversion to 1 bit can change the descent direction, impacting convergence. Standard Lion does not face this issue, as it applies the sign after summing updates. We analyzed the update vectors produced by standard Lion and the 1-bit allreduce when training a GPT model in Table 3. With 4 workers, over 30% of the parameters have an update where the majority vote is tied, leading to an erroneous update depending on how it is transmitted. While this rate decreases with 8 workers (as ties are less likely), it is still significant. Indeed, as Figure 10 demonstrates, these errors can lead to loss spikes and training divergence, and so are necessary to address, even if they may be relatively unlikely at scale. We find that a simple alternating sign method is sufficient to address this: we convert 0 to 1 on odd iterations and to -1 on even iterations before compressing to 1 bit. We considered a stochastic approach, but did not proceed with it as the overhead of random number generation was significant. Figure 10 shows that the alternating sign approach successfully stabilizes training and has minimal differences to using a 2-bit allreduce which transmits exact zeros.

C. Additional Explanation for Quantization

C.1. Toy Experiments

In Fig. 11, we implement a simple teacher–student MLP on synthetic data and introduce per-client heavy-tailed gradient noise sampled via the alpha stable Levy distribution (where $\alpha = 0.5$ recovers symmetric Levy noise and $\alpha = 2$ recovers

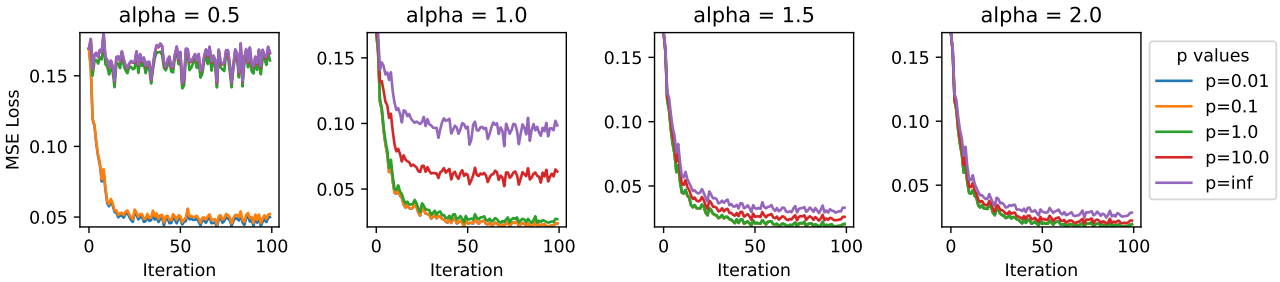


Figure 11: **Smaller p values enhance L_pQuantization under heavy-tailed noise.**

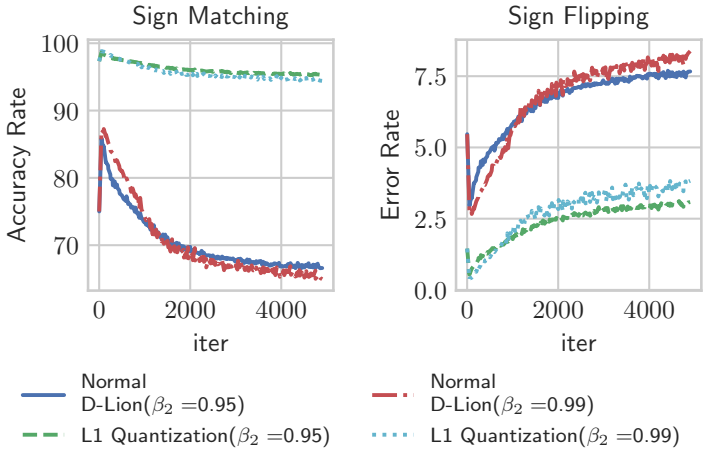


Figure 12: **Applying L1 quantization in vision models also improves the alignment rate.** We measured the same metrics as in Figure 4 for ResNet18 on CIFAR100, and found that L1 quantization similarly increases the alignment ratio in a vision model. This training was conducted with 8 GPUs. The variance in the input layer is significantly larger than that of other layers.

Gaussian noise). At each iteration, the noisy gradients from $C = 8$ clients are quantized by an L_p quantizer for $p \in \{10^{-2}, 10^{-1}, 1, 10, \infty\}$ (the case $p = \infty$ corresponds to the standard direct uniform quantizer). Figure 11 compares the baseline direct quantizer ($p = \infty$) against the family of L_p quantizers under varying α. As these results demonstrate, the more heavy-tailed the gradient noise becomes, the more the performance of the commonly used L_∞ quantizer degrades, widening the gap with L_0 quantization. This suggests that if gradient noise is heavy-tailed such as Transformer training or language model training, an appropriate L_p quantization (with smaller p) may be especially important.

C.2. Additional Experiments with vision models

Error rate Figure 16 shows that the effects of L1 quantization observed in a vision setting, specifically during ResNet18 training on CIFAR100, are similar to those observed in Figure 4. The results indicate that L1 quantization enables a close approximation of Lion in the ResNet18 vision model. Interestingly, while the approximation to Lion worsens without L1 quantization, as shown in Figure 9, the 1-bit allreduce method still performs well. This suggests that in this particular setup, effective optimization is possible without closely approximating Lion. Further investigation of this observation is left to future work.

ImageNet As shown in Table 3, L_1 quantization yields the lowest training loss during GPT pre-training, while both L_0 and L_∞ quantizations incur higher losses. To verify whether this behavior extends to vision models, we trained ResNet-50 on ImageNet for 90 epochs (Figure 13). The results reveal that L_0 quantization achieves accuracy on par with, or slightly above, L_1 quantization, making it the best-performing quantization scheme overall. Although L_1 quantization remains a viable alternative due to its comparable accuracy, L_0 quantization demonstrates the highest top-1 performance. These observations are consistent with the toy experiments in Figure 11. A more detailed analysis of the accuracy–efficiency trade-off between L_0 and L_1 quantization is left for future work.

CIFAR5m Figure 14 presents an ablation study of quantization functions for ResNet-18 trained on CIFAR5m and CIFAR100. Note that CIFAR-5m consists of six million synthetic, CIFAR-10-style images generated by a DDPM-based model[1]. On CIFAR100, model accuracy is largely insensitive to the choice of quantizer: L_∞, L_0, and L_1 quantization achieve

[1] https://github.com/preetum/cifar5m

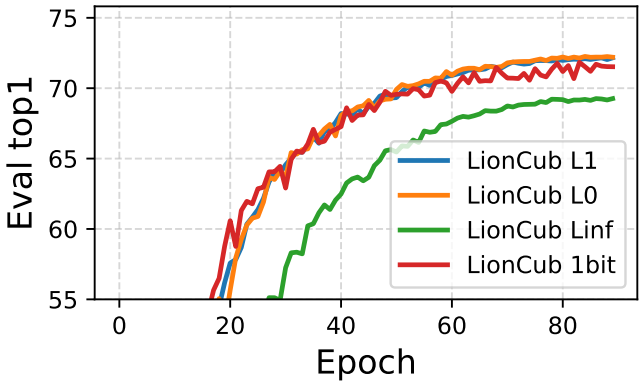


Figure 13: **ImageNet ablation study of quantization function.** We train ResNet-50 on ImageNet using different quantization functions. The training runs on 16 workers with a total batch size of 16,384. Because we use large-batch training, the final accuracy is lower than that of the standard 90-epoch ImageNet training.

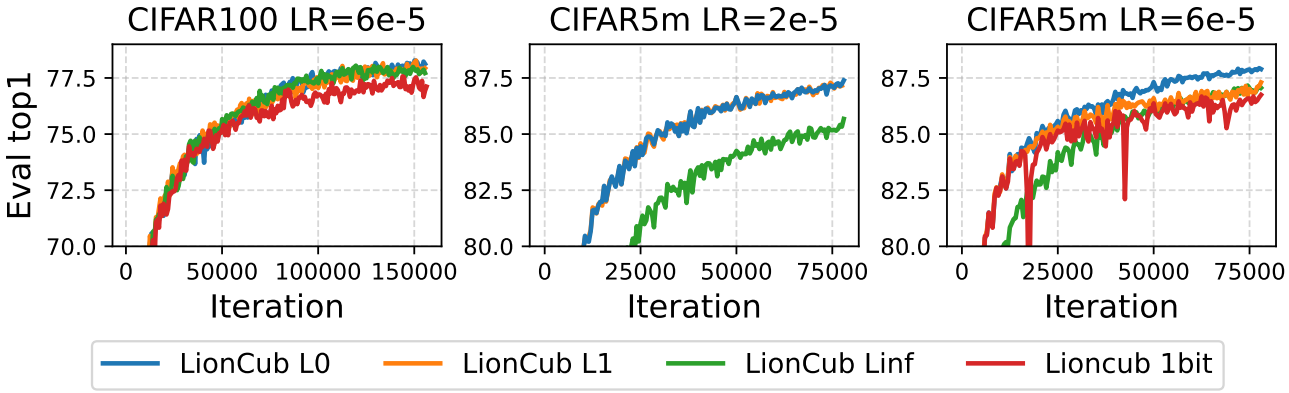


Figure 14: **CIFAR ablation study of quantization function.** Accuracy of ResNet-18 on CIFAR100 and CIFAR5m using L_∞, L_0, and L_1 quantization. While all three schemes yield nearly identical results on CIFAR100, the choice of quantization method produces clear performance differences on CIFAR5m.

nearly identical performance. Remarkably, this result mirrors the behavior observed in our toy experiment (Figure 11), where the weight distributions on CIFAR100 appeared not to be heavy-tailed but rather approximated Gaussian or uniform distributions. By contrast, when training on CIFAR5m with a learning rate of 2×10^{-5}, accuracy clearly depends on the quantization scheme: L_∞ quantization underperforms compared to L_0 and L_1. Intriguingly, increasing the learning rate to 6×10^{-5} causes L_0 quantization to surpass both the standard Lion optimizer and the other quantizers in accuracy. We hypothesize that the L_0 constraint acts as an implicit clipping mechanism, stabilizing weight updates under higher learning rates. These contrasting trends between the online-style CIFAR5m and the multi-epoch CIFAR100 highlight an important consideration: for developing and evaluating training methods for large-scale tasks—such as ImageNet classification or GPT language modeling—CIFAR5m may serve as a more representative benchmark than CIFAR100.

D. Additional Training Experiments

D.1. Additional Explanation for Momentum Synchronization

Vision Model The results of similar observations from Figure 6 are presented in Figure 16, showing the variance in momentum across layers for ResNet18 trained on CIFAR100. In the case of ResNet18, the variance was particularly high in the input layer. Unlike in GPT training, where both input and output layers exhibited high variance, it is interesting that only the input layer shows such variance in the vision model, with little variance observed in the output layer. In vision models, the smaller output dimension of the output layer may influence this variance, though further investigation is left for future work.

D.2. Additional experiments with language models

We conducted further experiments on GPT training by varying the number of nodes and model sizes, as shown in Figure hoge. In Figure 18, a 735M GPT model is trained across 4 nodes. Although 4-bit allreduce cannot be used in this case, we still observe a notable speedup over standard Lion. Additionally, Figure 19 presents training results for a 1.5B model. As

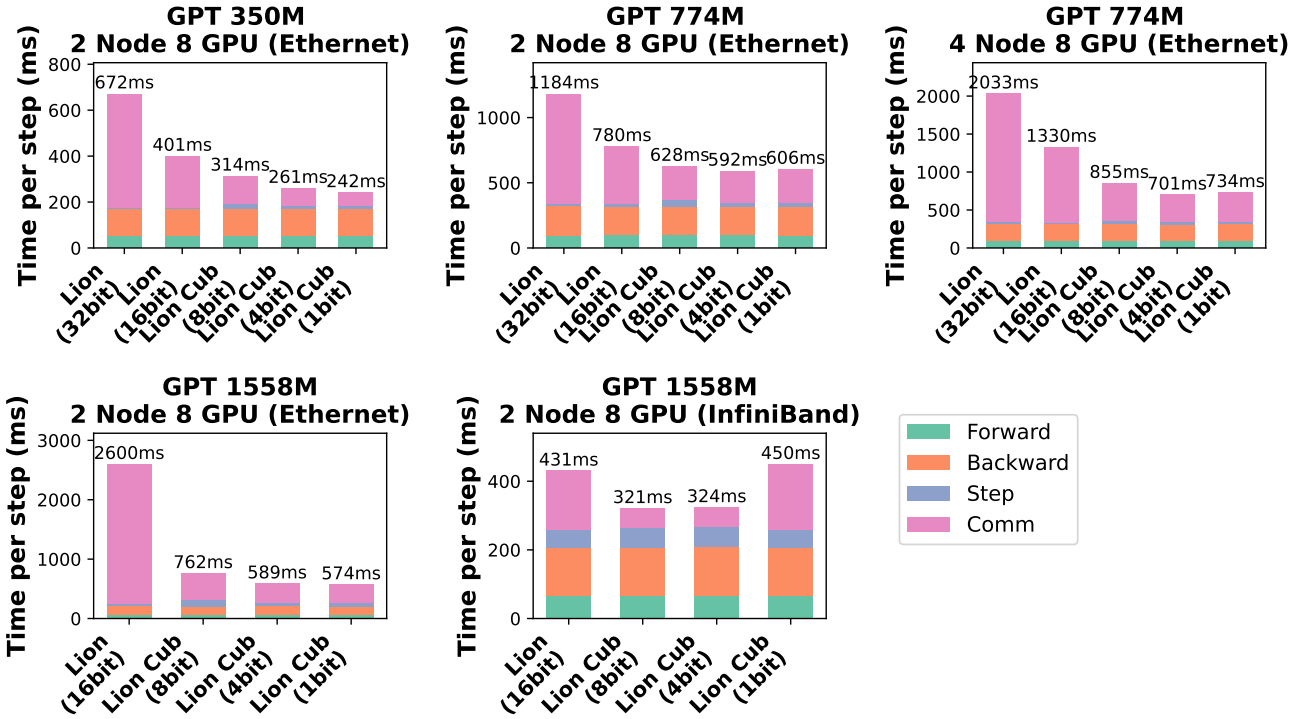


Figure 15: **Breakdown of runtime performance when training GPT.** Lion Cub significantly reduces communication volume, leading to reduced communication time and faster end-to-end iterations. "Comm" includes both communication time and quantization overheads.

indicated by this graph, reducing training time with Lion Cub becomes increasingly crucial as model size grows to 1.5B. In Figure 19, "Default" refers to the standard 32-bit Lion implementation, where training was conducted using PyTorch's DDP module without modifications. We observed that combining gradients into a single tensor and performing a single allreduce operation was faster than PyTorch's default approach. Consequently, for all experiments presented in this paper, we consolidated gradients into one tensor and executed only one allreduce call per iteration.

In Figure 20, we compare LionCub with communication efficient variants of Adam. When training on OpenWebText under restricted bandwidth, the 1-bit Adam outperforms standard Adam in terms of total training time; however, because LionCub has even lower communication and compression overhead, it achieves the fastest overall training. On the FineWeb corpus, using the same hyperparameters that lead Adam to converge causes both 1-bit Adam and 0/1 Adam to diverge. It may be possible to restore stable convergence for those variants through further tuning of learning rates or other settings. Moreover, 1-bit Adam must run for roughly 23000 iterations before freezing its momentum term by default, at which point its communication cost drops. In training regimes with only a few thousand iterations, LionCub, which does not require any warm-up phase, can achive more stable and rapid training. However, a comprehensive assessment of Adam's communication-reduction techniques versus LionCub will require a more fine-grained analysis of overheads, convergence behavior, and hyperparameter sensitivity and it is left for future work.

E. Additional profiling results

E.1. Different network bandwidth

Using NCCL_IB_HCA, we measured at three different communication bandwidth levels within the same cluster. The results are presented in Figure 4. The 1-bit LionCub implementation uses an AlltoAll algorithm, which in some cases yields lower throughput than the 4-bit (and higher) LionCub variants that use AllReduce. Note that when a high-speed interconnect (e.g. 200 Gbps InfiniBand) is available, the cost of quantization and other computations can dominate overall performance. Conversely, when limited to 1 Gbps Ethernet, the 32-bit Lion implementation spends 74 % of its time on communication; using to the 4-bit LionCub cuts this share to nearly half.

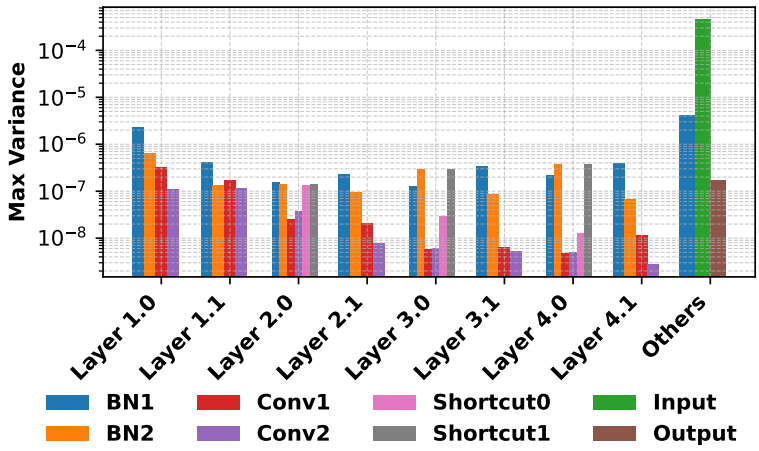


Figure 16: **The variance between workers is particularly large in the input layer.** We trained resnet18 on CIFAR100.

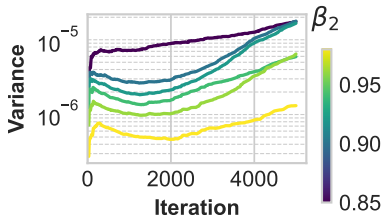


Figure 17: **The larger the β_2, the smaller the variance between workers.** We plot the progression of the maximum standard deviation between workers, which is observed for each element. The standard deviation decreases as β_2 increases. Additionally, the standard deviation tends to remain consistent throughout training.

E.2. Memory

We measured the memory consumption for training GPT with Lion Cub in Figure 21. For smaller model sizes, memory consumption is nearly the same across all methods. However, as model size increases, the memory usage of the Lion cub with 1-bit allreduce becomes higher than the others. This is due to the substantial, unavoidable temporary memory required for packing and unpacking data to a 1-bit format.

E.3. Vision model

We present results with ResNet-18 (He et al., 2016) and DeiT-Tiny, -Small, and -Base (Touvron et al., 2021) on CIFAR100 (Krizhevsky et al., 2009) in Figure 22, where Lion Cub similarly reduces both communication and training time. Here we also study the impact of the batch size: as batches grow, computation increases while communication remains constant. We see this effect in Figure 22, where the benefits of Lion Cub are more modest with large batch sizes. Hence, we expect that in large-batch training regimes with smaller models, as is common in vision tasks, reducing communication is of limited benefit. However, it has recently become a trend to train very large vision models, where communication reduction is more critical.

F. Experimental settings

F.1. Implementation Details

Algorithm 1 Distributed Lion with Quantization

Given $\beta_1, \beta_2, \lambda, \eta, f, Q$; **Initialize** $\theta_0, m_0 \leftarrow 0$
while θ_t not converged **do**
  $c_{i,t} \leftarrow \beta_1 m_{i,t-1} + (1-\beta_1)\nabla_\theta f_i(\theta_{t-1})$
  $c_t^* \leftarrow \sum_{i=1}^{N} Q(c_{i,t})$ (sync update vector)
  $\theta_t \leftarrow \theta_{t-1} - \eta_t \cdot \text{sign}(c_t^*) + \lambda\theta_{t-1}$
  $m_{i,t} \leftarrow \beta_2 m_{i,t-1} + (1-\beta_2)\nabla_\theta f_i(\theta_{t-1})$
end while

We implement Lion Cub in PyTorch (Paszke et al., 2019) 2.2.0 and use NCCL (NVIDIA, 2024) for communication. For our 1-bit allreduce, we built upon the DeepSpeed `compressed_allreduce` implementation (Tang et al., 2021) and

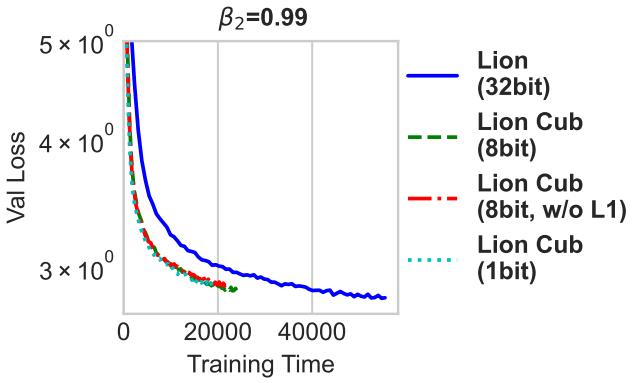
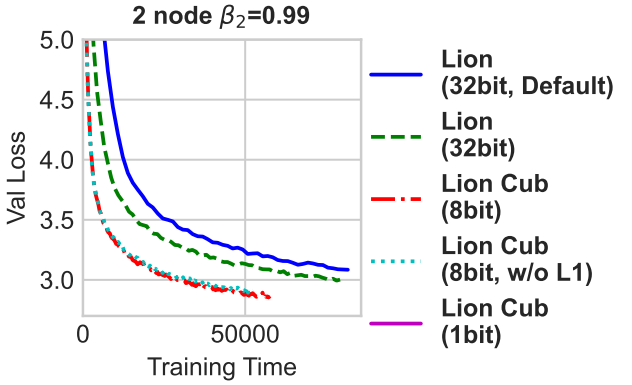
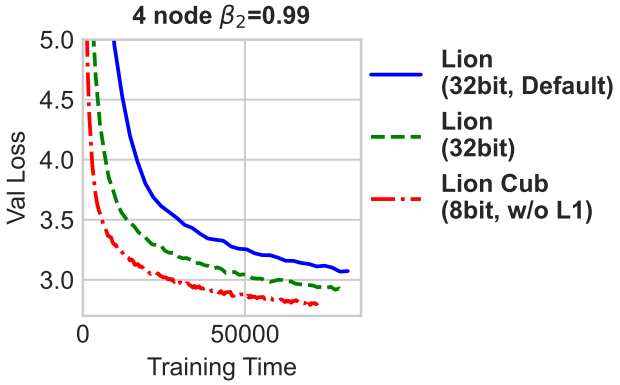


Figure 18: **Validation loss on OpenWebText with GPT-2 (735MB) when training with 4 nodes.** We trained a 735M parameter GPT model on OpenWebText with 16 GPUs using a cluster equipped with 4 H100 GPUs per node. In a 4-node training setup, we can not use the 4-bit version of Lion Cub because it exceeds the representable value range. Thus, we must use either the 8-bit or 1-bit variant of Lion Cub.



(a) $n = 2$



(b) $n = 4$

Figure 19: **Validation loss on OpenwebText with GPT-2(1.5B).** We trained a 1.5B parameter GPT model on OpenWebText using a cluster equipped with 4 H100 GPUs per node. With low-latency environments, this approach significantly reduces training time.

adapted it to Lion. We identified the packing and unpacking of the 1-bit representation as a bottleneck that can dominate communication time, and implemented optimized routines using CuPy (Nishino & Loomis, 2017).

We conduct all experiments on a cluster where each compute node has four NVIDIA H100 GPUs with 94 GB of memory. Nodes are interconnected with 4× NDR200 200 Gbps InfiniBand NICs. To simulate a slower Ethernet network, we set the `NCCL_IB_DISABLE` environment variable to prevent NCCL from using the InfiniBand transport, and instead fall back to sockets with significantly lower bandwidth.

For our p-bit allreduce, we directly use NCCL's allreduce with its standard summation reduction. We select the datatype for communication based on p and the number of processes communicating. When fewer than 8 bits are needed, we pack entries into a uint8; however, we currently require the bitwidth to evenly divide the size of the datatype we use. To maximize efficiency, we map $\{-1, 1\} \mapsto \{0, 1\}$, which saves one bit per value. For example, when performing a 4-bit allreduce among 8 workers, we simply sum in 0/1 format. For an 8-bit allreduce among 8 workers, we apply L1 quantization to map values to $[0, 15]$ before allreducing.

We note that the 1-bit compressed allreduce uses an all-to-all to implement a reduce-scatter operation. However, this operation cannot be directly implemented with the standard reduce-scatter in NCCL or MPI due to the need to change bitwidths at intermediate reduction stages. A custom implementation doing this internally may yield further performance improvements; we leave this to future work.

F.2. GPT on OpenwebText

All our GPT training results use adaptations of the original GPT-2-style models (Radford et al., 2019), incorporating modern training techniques from LLaMA (Touvron et al., 2023), such as Gated Linear Unit activations and RMSNorm. These adjustments are aligned with the techniques in Liu et al. (2024). The learning rate was generally set to 6e-5, except for the 1558M model, which was trained with a learning rate of 1e-5. The value of β_1 was set to 0.9, and gradient clipping was

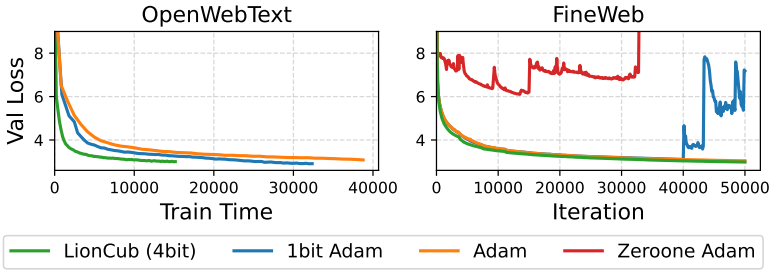


Figure 20: **Comparison with Adam.** We trained a 735M GPT model on OpenWebText and FineWeb with 8 GPUs.

Table 4: **Total training time and communication overhead ratio for different network speed.**

	1 Gbps		10 Gbps		200 Gbps	
Optimizer	**1step (s)**	**Comm. (%)**	**1step (s)**	**Comm. (%)**	**1step (s)**	**Comm. (%)**
Lion(32bit)	1.0	74.8	0.5	43.6	0.3	22.0
Lion(16bit)	0.6	60.3	0.4	30.3	0.3	14.7
LionCub(8bit)	0.5	41.8	0.3	20.8	0.3	9.7
LionCub(4bit)	0.3	32.2	0.3	19.3	0.3	11.8
LionCub(1bit)	0.4	41.1	0.4	28.1	0.3	25.0

applied at 1.0. Weight decay was also applied with a value of 1.0. The batch size was typically 12, but for the 1558M model, it was reduced to 8 due to memory constraints. Gradient accumulation was not used. The configurations for each model size are as follows:

- 124M GPT: $n_{\text{layer}} = 12$, $n_{\text{head}} = 12$, $n_{\text{embd}} = 768$
- 350M GPT: $n_{\text{layer}} = 24$, $n_{\text{head}} = 16$, $n_{\text{embd}} = 1024$
- 774M GPT: $n_{\text{layer}} = 36$, $n_{\text{head}} = 20$, $n_{\text{embd}} = 1280$
- 1558M GPT: $n_{\text{layer}} = 48$, $n_{\text{head}} = 25$, $n_{\text{embd}} = 1600$

The sequence length was set to 1024 for all models and dropout was not applied.

F.3. GPT on FineWeb

We trained a 774M-parameter GPT model (36 layers, 20 attention heads, embedding dimension 1 280) for 50,000 steps . We used the Lion optimizer with a base learning rate of 3e-5; in addition, we scaled the head, embedding, and scalar parameter groups to 3e-3, 3e-2, and 3e-3 respectively. The betas were set to ($\beta_1 = 0.8, \beta_2 = 0.95$), weight decay to 1e-2, and no gradient clipping was applied. We used a training sequence length of 12,288 tokens and a validation sequence length of 65,536 tokens with no dropout.

F.4. ResNet and DeiT

For both ResNet and DeiT models, the learning rate was generally set to 3e-4 and scheduled using a cosine scheduler. The value of β_1 was set to 0.9, with a weight decay applied at 0.0001. In addition to RandomCrop, Cutout, and RandomHorizontalFlip, we also applied auto-augmentation tailored for CIFAR. For ResNet18, we used the CIFAR-optimized version as described in the Cutout paper (DeVries, 2017).

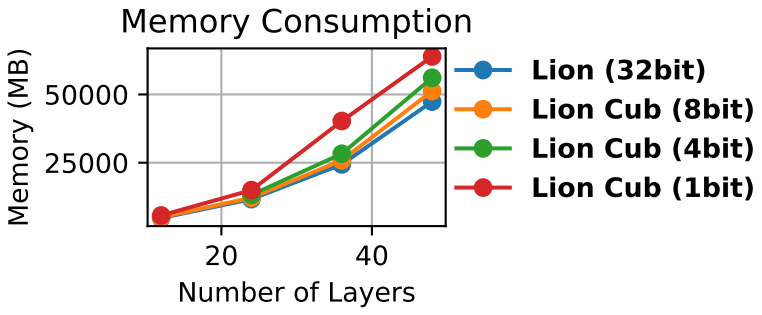


Figure 21: **Maximum memory consumption when training GPT using Lion Cub.** This figure illustrates the peak memory usage across different layers during GPT training with the Lion Cub optimizer.

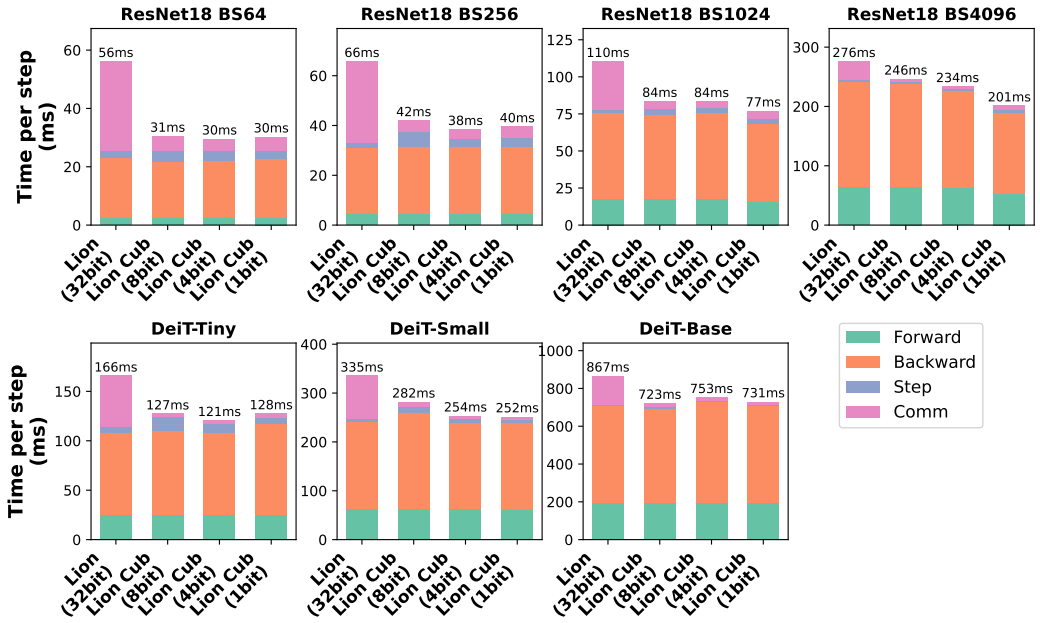


Figure 22: **Breakdown of runtime performance when training ResNet and DeiT.** All results use 2 nodes and our "Ethernet" configuration and "Comm" includes both communication time and quantization overheads. Lion Cub significantly reduces communication and overall iteration time compared to Lion.