TAGC: Optimizing Gradient Communication in Distributed Transformer Training

Igor Polyakov
iv.polyakov@vk.team
VK, ITMO University
Russia

Alexey Dukhanov
dukhanov@itmo.ru
ITMO University
Russia

Egor Spirin
egor.spirin@vk.team
VK Lab
Russia

Abstract

The increasing complexity of large language models (LLMs) necessitates efficient training strategies to mitigate the high computational costs associated with distributed training. A significant bottleneck in this process is gradient synchronization across multiple GPUs, particularly in the zero-redundancy parallelism mode. In this paper, we introduce Transformer-Aware Gradient Compression (TAGC), an optimized gradient compression algorithm designed specifically for transformer-based models. TAGC extends the lossless homomorphic compression method by adapting it for sharded models and incorporating transformer-specific optimizations, such as layer-selective compression and dynamic sparsification. Our experimental results demonstrate that TAGC accelerates training by up to 15% compared to the standard Fully Sharded Data Parallel (FSDP) approach, with minimal impact on model quality. We integrate TAGC into the PyTorch FSDP framework, the implementation is publicly available at https://github.com/ipolyakov/TAGC.

CCS Concepts: • **Computing methodologies → Machine learning**; **Distributed computing methodologies**.

Keywords: Distributed Learning, Gradient Compression, Language Model, Transformer, Inter-Node Communication

1 Introduction

With the rapid advancement of large language models (LLMs), their applications have expanded across various domains, including natural language processing (NLP) and computer vision (CV) [32], audio processing [12], and healthcare [17]. However, training a state-of-the-art LLMs requires immense computational resources [27, 29, 33]. The trend toward increasingly larger model sizes is evident in models such as DeepSeek [22], LLaMA [10], and GPT [4]. This growth in model complexity significantly impacts computational costs, particularly in terms of training time, which can span several weeks or even months. Furthermore, incremental improvements in large models are driven by extensive experimentation, making the execution time of such experiments a critical factor in accelerating advancements in the field.

Multiple methods have been suggested to optimize the LLM training, allowing to scale models faster by improving GPU utilization. One possible direction for optimization involves the development of more efficient operators that uses GPU-specific computation, such as FlashAttention [8], XFormers [18], and LigerKernels [15]. On the other hand, training large-scale neural networks typically requires hundreds of GPUs. This is achieved through distributed training strategies, including pipeline [14, 23], tensor [16, 24] or zero-redundancy [26, 28] parallelism, which breaks the training process into parts that are executed in parallel on different computing units. The model is broken down into a number of shards, each of them is executed on a separate device with periodic synchronization, primarily of gradients.

The synchronization between devices is a component of an overall training performance, and it can be a bottleneck in large models training because the compute amount grows faster than network bandwidth [11]. The existing techniques optimize different stages of synchronization between devices. For example, tensor quantization both for forward activations [6] and gradients [1, 9, 30] or gradient sparsification [3, 30]. This also includes lossless compression methods [19] for gradients.

Although there are various communication optimization techniques, it is not always possible to integrate them into existing tools for training large neural networks. One widely used tool for optimizing distributed training is NCCL [25], a library that provides efficient collective communication

operations for GPU clusters and is considered the industry standard. However, NCCL requires that operations have the distributive property, namely $f(grad_1) + f(grad_2) = f(grad_1 + grad_2)$ for some operation f. This is a constraint that many quantization and sparsification algorithms do not satisfy. This way, compression algorithms become a more viable approach to improving computational cluster utilization.

One existing method, lossless homomorphic compression [19] (LHC), is an efficient cross-device communication method using NCCL that allows training neural networks in Data Parallelism setup. The main idea is that the gradients of some models are sparse, which allows them to be effectively compressed during synchronization and reduce the load on the network. Experiments have shown more than 3x speed up for models with $\approx 95\%$ sparsity and only 1.2x speed up for the BERT model that uses the transformer [31] architecture and had very dense gradients with $\approx 20\%$ sparsity. The lack of model sharding support and ignoring specifics of transformer architecture make it unsuitable for use with LLM training pipelines.

In this paper, we propose an approach reducing gradient synchronization time during transformer-based model training in the zero-redundancy parallelism mode. Our method, Transformer-Aware Gradient Compression (TAGC), is based on LHC using all its efficiency, but with adaptation for sharded models and optimized specifically for the transformer architecture. We also provide several hyperparameters, making it possible to select a balance between accelerating the training speed and maintaining model quality.

2 Background

2.1 Distributed Training

Distributed training enables scaling by partitioning computations across multiple devices, such as GPUs or TPUs, allowing for more efficient training of large-scale models. There are several strategies for distributed training, among which Data Parallelism [20] and its extension Sharded Data Parallelism [26, 33] are commonly used.

In Data Parallelism (DP), the dataset is divided into smaller batches, and each computing device processes a separate batch independently. Each device holds a complete copy of the model and performs forward and backward passes on its local batch. After computing local gradients, all devices are synchronized using the All-Reduce operation, resulting in an average gradient on each of them. At the end of the step, optimizer updates model parameters with the same gradients on each device.

The Sharded Data Parallelism or Zero-Redundancy improves DP by dividing the model, its gradients and optimizer states into a number of shards, and stores each shard on a separate device. This makes it possible to train bigger models at the cost of increased communication, as there are multiple

Table 1. Compression level relative to the percentage of sparseness in the LHC algorithm. $\theta\%$ means that the synchronization matrix must contain at least $\theta\%$ of zeros, and can be lossless compressed-decompressed with the corresponding compression ratio.

Compression level	Sparsification (θ), %
2x	80%
4x	90%
10x	98.75%

synchronisations of shards during each forward and backward step. Main operations are All-Gather (to collect shards from all devices) and Reduce-Scatter (to distribute the shards back on separate devices).

NVIDIA Collective Communications Library (NCCL) [25] is commonly used to optimize multi-GPU communication, providing efficient implementations of collective operations such as Reduce, All-Reduce, All-Gather, and Reduce-Scatter.

2.2 Lossless Homomorphic Compression

The lossless homomorphic compression method [19] is a gradient compression technique with efficient in-network communication. The method introduces two data structures to transfer the gradient: Index and Count Sketch. Index tracks the indexes of non-zeros gradient values, and Count Sketch stores those values in a compressed manner. LHC uses the All-Reduce collective communication operation and integrates into Data Parallelism pipelines, allowing for efficient and lossless gradient synchronization.

LHC allows applying different levels of compression for Count Sketch with respect to its sparseness percentage. See Table 1 for details. Thus, the sparser the data, the higher the level of compression that can be applied. To decompress, LHC uses two algorithms: peeling and estimation. The peeling algorithm provides a substantially better quality, but works effectively only when the gradients are sparse and have 80-95% of zeros. The estimation works for any gradient sparsity, but introduces noise, which is generally better to avoid.

Due to limitations in NCCL, which does not support the bitwise OR operation needed to correctly synchronize Index between devices, it is simulated using an addition operation. There are two options to do so: store a 1-bit Index that is faster, but with a higher amount of collisions, or a 4-bit that is stable, but 4 times bigger in terms of communication amount.

Direct application of LHC to transformer-based architectures is feasible only with an estimation strategy due to the high density of gradient values [2], which leads to a noticeable degradation in quality. Furthermore, modern large language models (LLMs) require sharding during training,

---

Algorithm 1: THLC Communication Hook

Input: shard gradient g_{shard}, sparsfication threshold θ, shard gradient accumulator acc, shard master device d

Output: total gradient on master device g, updated shard gradient accumulator acc_{next}

1   $g_{shard} \leftarrow g_{shard} + acc$;
2   $g_{shard}, acc_{next} \leftarrow \text{sparsify}(g, \theta)$;
3   $local\_index \leftarrow \text{create\_index}(g)$;
4   $index \leftarrow \text{All-Reduce}(local\_index)$;
5   $local\_count\_sketch \leftarrow \text{compress}(g_{shard}, index)$;
6   $\text{Reduce}(local\_count\_sketch, d)$;
7   // Applying gradient only on Master device
8   **if** $LOCAL\_RANK = d$ **then**
9      $count\_sketch \leftarrow$ reduced local count sketches;
10      $g \leftarrow \text{peeling\_decompress}(index, count\_sketch)$;
11   **return** g, acc_{next};
12   **return** None, acc_{next};

Figure 1. Communication amount per parameter per rank in bits for LHC and TAGC algorithms for 1-bit and 4-bit Index. Count Sketch stores compressed parameters, 2x compression leads to 16-bit per parameter. (a) 4-bit Index for LHC and TAGC algorithms. (b) 1-bit Index for LHC and TAGC algorithms.

necessitating the use of the Reduce, Reduce-Scatter, and All-Gather operators for collective communication, which are not supported by LHC.

3 Method

Our method, Transformer-Aware Gradient Compression (TAGC), consists of two parts: integrating lossless homomorphic compression into Zero-Redundancy parallelism, where a model is sharded across devices, and its adaptation to transformer architecture for fast and stable communication. We also provide technical details of integrating and configuring TAGC into PyTorch FSDP [33], Zero-Redundancy parallelism implementation in one of the most popular deep learning frameworks.

3.1 Gradient Compression in Sharded Setup

Our method is based on the LHC [19] compression method, but with respect to collective communications for sharded models. The original LHC implementation relies on Data Parallelism pipeline, synchronizing both Index and Count Sketch using All-Reduce at the end of the backward pass. Meanwhile, TAGC only uses All-Reduce for Index synchronization to determine gradients to reduce, while Count Sketches are synchronized by the Reduce operator, which communicates twice less data.

Figure 1 demonstrates the communication amount for the both Index sizes and compressed count sketch with LHC and TAGC algorithms per parameter. Significant improvement in network load can be observed. Since LHC relies on All-Reduce and TAGC integrates the Reduce operation, our approach provides a 2x advantage in the amount of communication for Count Sketch, that is generally larger than Index
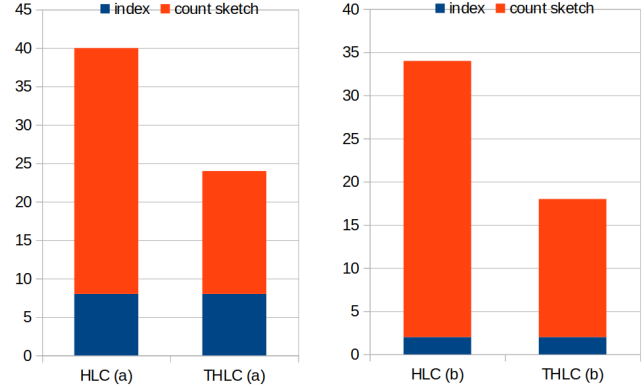
per parameter. Compared to no compression, where each parameter is synchronized as is, TAGC may achieve a 6.15x increase in speed for the 32-bit number representation. Full compression is presented in Figure 2.
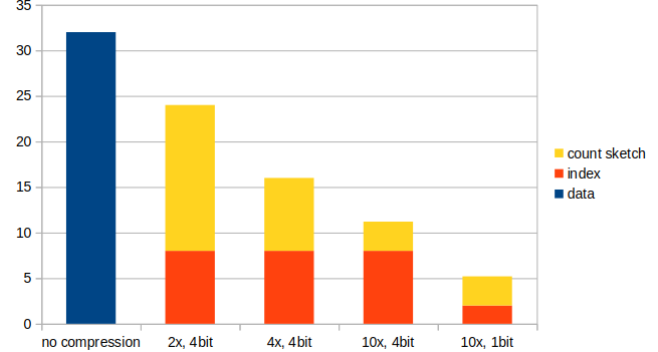

Figure 2. Communication amount per parameter per rank in bits for various compression configurations in TAGC.

3.2 Compressing Transformer Gradients

Gradient Sparsification. The main factor to successfully integrate TAGC into training pipeline is the sparseness of gradients. Although the transformer gradients are not sparse at all, its gradients follow an approximately log-normal distribution [5]. Based on this, we perform a preliminary sparsification of gradients before their synchronization. This improvement makes TAGC use a peeling algorithm to decompress and avoid estimation, which leads to a minimal quality drop.
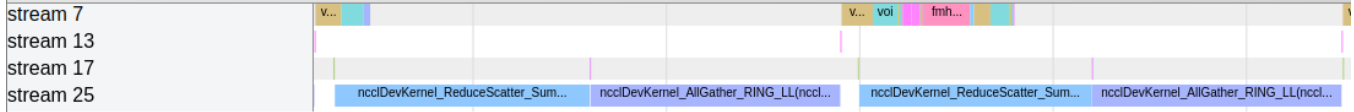
(a) Default FSDP workload without any compression. Computation (streams 7, 13, 17) takes only 22.4% of the time. Communication (stream 25) takes 96.1%. 47.5% is taken by the gradient exchange only, via Reduce-Scatter.
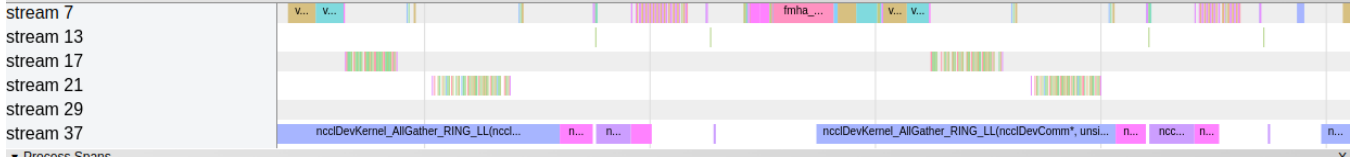
(b) TAGC compression, using 10x compression and 1-bit Index. Computation (streams 7, 13, 17, 21, 29) takes 64.6% of the time. Communication (stream 37) takes 80.4%. Gradient exchange takes 20.2% without gaps, or 42.1% with gaps.

Figure 3. CUDA stream profiles for backward step for 2 layers placed in consecutive FSDP units. Layers are attention projection and feed-forward up-projection. Each row represents a computation or communication stream in an execution timeline. For TAGC, communication time is 32.1% shorter than that of the baseline.

We sparsify each gradient fragment using a sparsity threshold, θ. The value for zeroing the gradient is chosen dynamically, so that at least θ% of the values become zero. θ is a hyperparameter of the algorithm.

Sparsifying gradients which are not zeros means losing information and leads to loss increase. We adapt local gradient accumulation [21] to avoid completely losing gradients, which we set to zero. We accumulate the sparsified values of the gradients locally, and we add them to associated gradients on the next iteration.

Selective Compressing. Another way to stabilize training is to selectively apply TAGC to model parameters. This approach allows only large layers with room for sparsity to be compressed and avoid overhead in the compression calculation. The transformer layer [31] consists of two main components: an attention mechanism for transferring information between text tokens, and a feed-forward mechanism, which is applied to each token individually. There are already numerous optimized attention kernels that fuse operations internally and provide fast computation and communication, e.g. FlashAttention [7, 8] or Torch SDPA. Furthermore, there are also normalization layers where parameters are stored, but they are too small to affect the overall communication time.

Therefore, we apply TAGC in two settings: i.e. for all layers, and for all non-attention linear layers. Non-attention linear layers in the transformer model include linear layers in a feed-forward block, an embedding matrix, a language modeling head, and a projection layer after the attention mechanism.

3.3 Integrating into PyTorch FSDP

TAGC implementation relies on PyTorch FSDP [33], a Sharded Data Parallelism implementation in one of the most popular frameworks for training neural networks. The integration is based on various communication hooks implemented in the framework to compress and decompress gradients before and after cross-device communication.

To reduce iteration time, in addition to directly using gradient compression and exchanging the Index and Count Sketch instead of the full gradient, we maximized the overlap of computation and communication operations. This includes implementation of several callback functions (hooks) within TAGC that would allow parallel execution of operations for the next shard. For example, when performing a communication step for shard 0, a hook is called that allows computation for shard 1, such as sparsification, and Index creation.

For that, we have replaced the Reduce-Scatter operation in the data exchange stream with a set of compression and compressed data exchange operations in such a way that the data exchange and additional processing operations are performed in parallel where possible. New and existing operations are synchronized. The existing operations are performed in four CUDA streams: for performing the backward step, for copying data, auxiliary operations before and after the exchange (to avoid overflow of quantized data types, if they are used), and for communicating data between devices.

In addition, we also overlapped the sparsification and the All-Gather operation of the FSDP framework, which prefetches the parameters of the next block. The FSDP framework does not support overlapping the All-Gather operation with computation. This does not cause any problems in the default mode of operation of the FSDP framework, in which gradient accumulation is performed using Reduce-Scatter, since no computation is performed in this case. When adding the computation of our method, namely sparsification, Index computation, peeling, and so on, it was initially impossible to achieve parallelism between the computation and the All-Gather prefetching. The main problem is that FSDP places the memory copy operation in the CUDA communication

stream, completing the backward step. We solved this by placing the memory copy operation in the main computation stream. This allowed us to parallelize the sparsification, initialization, and Index creation with prefetching the next block of parameters.

Overall, TAGC allow to efficiently synchronize gradients for transformer-based model in sharded setup. This achieved by selective sparsification of gradients, proper selection of collective communication operations, and overlapping communication and computation in the target distributed learning framework. See Algorithm 1 with pseudocode for more details. Note that it does not include overlapping, since it heavily relies on PyTorch design.

4 Experiment

To demonstrate the effectiveness of TAGC, we provide multiple experiments with training a GPT-like [4] model on the language modeling problem in a distributed setup.

4.1 Experimental Setup

To show the impact of compressing gradients over synchronizing them as they are, we use a setup where communications become a bottleneck. We use 2 hosts, each of which is equipped with a single NVIDIA GeForce 3080 GPU and connected with a 10 Gbit/sec network. For the model configuration, we utilize the GPT-2 [4] architecture and use a small-size configuration with approximately 162 million parameters. All models in experiments are trained on the OpenWebText [13] dataset.

For selective compression, non-attention linear layers represent approximately 82% of all model parameters and, therefore, gradients.

4.2 TAGC Effectiveness

The efficiency of TAGC depends on speeding up training without losing quality. First, we profile the backward step for both the default FSDP implementation and compression with TAGC. Figure 3a provides information for the backward step without compression, in which the computation takes only 22.4% of the time. In addition, communication takes 96.1% of the time, 47.5% of which is spent exchanging gradients via Reduce-Scatter. After applying TAGC, as shown in the Figure 3b, the computation takes a higher percentage of total time due to the lower communication time. TAGC spent 32.1% less time for communication than default FSDP.

To demonstrate the effect of reduced communication time, we measured average iteration time during training models. See Table 2 for the results. Note that applying TAGC for all layers leads to increased iteration time. This is due to the large number of small layers, such as normalizations, which are faster to transmit as is, rather than compress, transmit and decompress. But when selectively compressing

only large linear layers, the iteration time was significantly reduced.

Table 2. Average iteration time of the default FSDP and TAGC for both selective compression options. TAGC uses 10x compression with 1-bit Index. FSDP with Non-Attention Linear layers compression stands for the wrapping strategy (whether small layers are stored in the default unit).

Algorithm	Compressed Layers	Iteration time, sec	Gradient exchange, ms/MB
FSDP	All	26.8	0.73
TAGC	All	28.5	0.64
FSDP	Non-attn	25.2	0.76
TAGC	Non-attn	**22.6**	**0.51**

To estimate quality drop, we trained multiple models in different TAGC setups. Figure 4 provides the validation curves during training. Application of TAGC leads to a minor loss increase. Difference between default FSDP training and TAGC in the most compressed setup, 10x compression and 1-bit Index, is 3.6%. TAGC with 2x compression and 1-bit index produces an unexpected result with the highest loss value, possibly due to the high number of collisions because of dense gradients together with the unstable index mode.
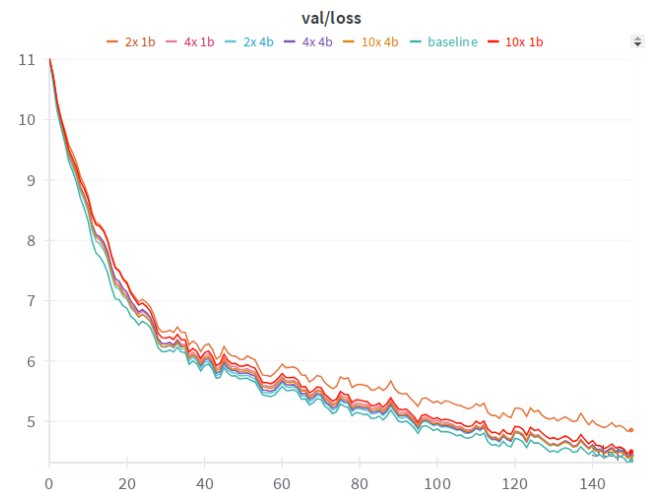

Figure 4. Validation loss by iteration number for various TAGC configurations and plain FSDP baseline.

TAGC reduced the communication time during gradient synchronization under low network bandwidth conditions by more than 30% resulting in a total communication speed up (together with forward step) of 15%, with a quality drop of only 3.6%. Moreover, TAGC has several hyperparameters, such as Index size and sparsity threshold, to balance

between increasing the speed of communication and the loss in quality.

5 Discussion and Future Work

TAGC is a possible optimization algorithm for computational clusters where a network or bus is a bottleneck. By compressing gradients, TAGC reduces the amount of data transferred at the cost of additional computation for compression and subsequent decompression. Since TAGC is based on LHC, it inherits the potential of lossless compression, resulting in negligible loss in quality.

Modern computing clusters contain hundreds of GPUs, the main approach to their arrangement is individual machines with 8 devices, which are then connected to a common network through a switch. Devices per machine are connected with very high bandwidth, intra-node communication and, therefore, TAGC will only slow down the training. TAGC is useful in inter-node communication, e.g. connecting 8 devices from one machine with 8 devices from others, especially when cluster is built without a specialised network, e.g. InfiniBand. One possible direction of future work includes adaption of a hybrid strategy where compression applies only for inter-node communication. Applying compression for other distributed strategies, e.g. Pipeline Parallelism or solely Tensor Parallelism, is also an important direction to enhance cluster utilization.

Another option to improve the method is to combine the gradient compression with quantization. The current LHC implementation provides solely float32 operations. A promising direction is to combine LHC compression together with quantization of gradients to float16, bfloat16, or int8 during communication to achieve even better compression.

6 Conclusion

In this paper, we introduced the TAGC algorithm for compressing gradients to accelerate communications during distributed training of neural networks. TAGC is specifically designed to work with the Sharded Data Parallelism training strategy, which is currently one of the most popular ways of training large neural networks. We also take into account the specifics of transformer architecture, applying layer selective compression with further dynamic sparsification.

As a result, TAGC speeds up training up to 15% compared to default FSDP in low-network bandwidth, with a loss degradation of only 3.6%. This is achieved with the maximum level of compression, where 98.75% of all gradients are sparsified. TAGC allows selecting the compression level to balance between speed and quality during training.

We implemented TAGC as a part of PyTorch FSDP framework, making it easy to integrate the algorithm in existing pipelines. To facilitate further research, we made our code publicly available at https://github.com/ipolyakov/TAGC.

References

[1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. *Advances in neural information processing systems* 30 (2017).

[2] Abhimanyu Rajeshkumar Bambhaniya, Amir Yazdanbakhsh, Suvinay Subramanian, Sheng-Chun Kao, Shivani Agrawal, Utku Evci, and Tushar Krishna. 2024. Progressive Gradient Flow for Robust N: M Sparsity Training in Transformers. *arXiv preprint arXiv:2402.04744* (2024).

[3] Debraj Basu, Deepesh Data, Can Karakus, and Suhas Diggavi. 2019. Qsparse-local-SGD: Distributed SGD with quantization, sparsification and local computations. *Advances in Neural Information Processing Systems* 32 (2019).

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[5] Brian Chmiel, Liad Ben-Uri, Moran Shkolnik, Elad Hoffer, Ron Banner, and Daniel Soudry. 2020. Neural gradients are near-lognormal: improved quantized and sparse training. *arXiv preprint arXiv:2006.08173* (2020).

[6] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085* (2018).

[7] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[9] Yuqing Du, Sheng Yang, and Kaibin Huang. 2020. High-dimensional stochastic gradient quantization for communication-efficient edge learning. *IEEE transactions on signal processing* 68 (2020), 2128–2142.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[11] Amir Gholami. 2023. *Memory Footprint and FLOPs for SOTA Models in CV/NLP/Speech.* Retrieved January 4, 2025 from https://github.com/amirgholami/ai_and_memory_wall/blob/main/imgs/pdfs/ai_and_compute.pdf

[12] Deepanway Ghosal, Navonil Majumder, Ambuj Mehrish, and Soujanya Poria. 2023. Text-to-audio generation using instruction-tuned llm and latent diffusion model. *arXiv preprint arXiv:2304.13731* (2023).

[13] Aaron Gokaslan and Vanya Cohen. 2019. OpenWebText Corpus. http://Skylion007.github.io/OpenWebTextCorpus.

[14] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. 2021. Pipetransformer: Automated elastic pipelining for distributed training of transformers. *arXiv preprint arXiv:2102.03161* (2021).

[15] Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. 2024. Liger Kernel: Efficient Triton Kernels for LLM Training. *arXiv preprint arXiv:2410.10989* (2024).

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

[17] Yubin Kim, Xuhai Xu, Daniel McDuff, Cynthia Breazeal, and Hae Won Park. 2024. Health-llm: Large language models for health prediction via wearable sensor data. *arXiv preprint arXiv:2401.06866* (2024).

[18] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, et al. 2022. xformers: A modular and hackable transformer modelling library.

[19] Haoyu Li, Yuchen Xu, Jiayi Chen, Rohit Dwivedula, Wenfei Wu, Keqiang He, Aditya Akella, and Daehyeok Kim. 2024. Accelerating Distributed Deep Learning using Lossless Homomorphic Compression. *arXiv preprint arXiv:2402.07529* (2024).

[20] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[21] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[22] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[23] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.

[24] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[25] NVIDIA. 2025. *NCCL Library*. Retrieved February 2, 2025 from https://github.com/NVIDIA/nccl

[26] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[27] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[28] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

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

[30] Zhenheng Tang, Shaohuai Shi, Wei Wang, Bo Li, and Xiaowen Chu. 2020. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307* (2020).

[31] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[32] Wenhai Wang, Zhe Chen, Xiaokang Chen, Jiannan Wu, Xizhou Zhu, Gang Zeng, Ping Luo, Tong Lu, Jie Zhou, Yu Qiao, et al. 2024. Visionllm: Large language model is also an open-ended decoder for vision-centric tasks. *Advances in Neural Information Processing Systems* 36 (2024).

[33] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).