Gradient-Compressed HBFP for Distributed Deep Learning Workloads

Jinay Dagli
jinay.dagli@iitgn.ac.in

***Abstract*—With the massive increase in the computational resources required for the training of Deep Neural Networks (DNNs), low-precision training is shown to be a promising alternative. One such low-precision training involves the use of Hybrid Block Floating Point (HBFP) as an alternate numerical encoding for training neural networks with as low as 6 bits [1]. Another major bottleneck in the current distributed deep learning setup is the time required by the nodes to communicate gradients to other nodes. One way of reducing this communication time is by compressing the gradients before communication takes place [2]. In this project, we first analyse the benefits of HBFP over other low-precision formats, such as bfloat16. We then incorporate gradient compression techniques, such as PowerSGD [3]. Finally, we try to improve the gradient compression ratio by coming up with a novel gradient compression scheme that merges the benefits of MSTopK [4] and PowerSGD, while maintaining good accuracy.**

I. Introduction

Deep Neural Networks (DNNs) have become prevalent in many areas. The constant improvements in the DNN models and algorithms have led to an unparalleled growth in the model complexity and the area and power requirements for training such models. With such a large growth in the computational resources required for these applications and with the decline of the Moore's Law, low-precision training is one of the important alternatives. Several techniques have been proposed for enabling low-precision training, such as [5], [6], [7]. HBFP, in particular, shows promise as it enables training even the large neural networks, with just 6 bits, while maintaining accuracy. HBFP involves sharing a single exponent across a block of mantissas.
Current DNNs offer the benefit of data parallelism, such as while computing the gradient descent during training. But this leads to another major bottleneck in the current DL workloads, that is, the communication time required for communicating the gradients evaluated by each worker to the other nodes. Previous works have shown how this communication time dominates the total time taken for each iteration in the training [4]. Several works show that compressing gradients before communicating those could help alleviate this problem with a low cost of accuracy loss.
In this project, we target both the problems (arithmetic density and gradient communication) by incorporating gradient compression (GC) techniques into HBFP training. We then try to improve the compression ratio with a novel technique built upon two important gradient compression techniques (PowerSGD (pSGD) and MSTopK).

A. Hybrid Block Floating Point (HBFP)

The shortcoming of using floating-point arithmetic is the energy consumption required by the hardware. On the other hand, fixed-point representation cuts down the representable range of numbers. BFP, or Block Floating Point, lies at the intersection of both these formats. BFP uses a shared exponent for a block of mantissas (Figure 1). The exponent enables a dynamic range that could be represented.

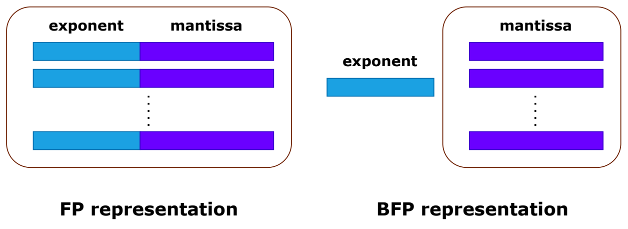

Fig. 1. (a) FP representation (Each number has an exponent and a mantissa) and (b) BFP representation (An exponent shared across a block of mantissas)

The drawback of the BFP representation is the storage of numbers in the BFP format, and performing some arithmetic operations in BFP might lead to accuracy loss (because of the wide distribution of values). To alleviate these issues, HBFP [5] was proposed to use BFP in all operations based on dot-products, and FP representation for all other operations.

B. Gradient Compression (GC)

In recent times, synchronous data-parallel SGD is a common way to accelerate the training of neural networks. But because of the large gradient vectors of models, communication time amongst the nodes creates a hindrance in the scalability of training. Lossy gradient compression acts as a promising solution to this problem.
Several GC techniques have been proposed over the years. These can be broadly classified into three types. The first way to compress gradients is sparsification (dropping the unimportant gradient enties). MSTopK is an example of sparsification of gradients that keeps only the top-K elements of the gradient vector. The second GC technique is quantization. Examples include signSGD, which just communicates the sign of the gradients. The third (and most efficient) way is by decomposing the large gradient vector into two low-rank matrices. The most promising work in this case is PowerSGD. More details regarding the algorithm and results of the MSTopK and PowerSGD scheme can be seen from the original papers [4], [3].

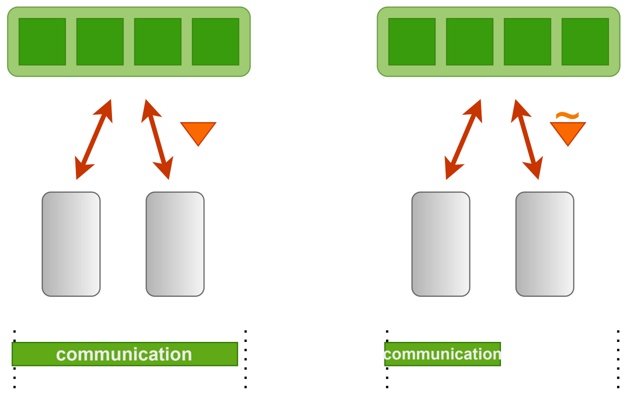

Fig. 2. Gradient Compression can reduce the communication time; and hence improve scalability of distributed training.

II. HBFP vs bfloat16

We compare the implementation of HBFP against bfloat16 in terms of the number of operations required to perform one complete step of update, and in terms of throughput. In order to show the relative overhead in terms of extra operations, we estimate the total number of operations required for one complete step in training of Resnet50. For HBFP8, we require 193.4M operations, as compared to 123.8M for bfloat16.
It is shown that for a systolic-array based accelerator, throughput as a function of its dimensions m, n, w, and frequency of operation f, is given by

$$T = 2mn^2wf$$

It is also shown that the maximum value of n for HBFP8 can be as large as 400, while that for bfloat16 is just 220. We observe that the maximum throughput for HBFP8 is nearly 6.3x that of bfloat16.

III. HBFP+GC Training Results

Table I and Table II show the accuracy results for training of two networks Resnet20 (on Cifar10) and Resnet50 (on Cifar100) respectively. [8] has shown that PowerSGD works best for gradient compression amongst the wide range of techniques available. We therefore choose PowerSGD as the SOTA GC technique for this project. The algorithm for PowerSGD is given as Algorithm 1. We analyse HBFP+GC for various mantissa bits, block sizes and PowerSGD ranks. HBFPm denotes HBFP representation with a mantissa bit width of m and exponent size of 10. r denotes the rank of PowerSGD used.
Using the insight that towards the last epochs, the gradient is towards the minima, we also experiment with using lower ranks towards the end of training (in the last 10% of the epochs), and observe a slight improvement in the accuracy for most cases.

Algorithm 1 Rank-r PowerSGD Compression

```
1: function POWERSGD($G \in \mathbb{R}^{n\times m}$, $r$)
2:     $Q \in \mathbb{R}^{m\times r}$ initialized from i.i.d std. normal distr.
3:     $P \leftarrow G \cdot Q$
4:     $P \leftarrow \frac{1}{W} \cdot (G_1 + \ldots + G_W) \cdot Q$        ▷ All Reduce Mean
5:     $\hat{P} \leftarrow$ ORTHOGONALIZE($P$)
6:     $Q \leftarrow G^\top \cdot \hat{P}$
7:     $Q \leftarrow \frac{1}{W} \cdot (G_1 + \ldots + G_W)^\top \cdot \hat{P}$ ▷ All Reduce Mean
8:     return $\hat{P} \cdot Q^\top$
9: end function
```

Algorithm 2 Proposed Algorithm

```
1: function PROPOSED($G$, $r$, $k$, $N$)
2:     $x \leftarrow G$[coordinate] for coordinate in coordinates
3:     $a \leftarrow$ abs($x$)
4:     $a, u \leftarrow$ mean($a$), max($a$)
5:     $l, g \leftarrow 0, 1$
6:     $t1, t2 \leftarrow 0, 0$                        ▷ Initialize thresholds
7:     $k1, k2 \leftarrow 0$, len($x$)     ▷ Initialize indices for thresholds
8:     for $i = 1$ to $N$ do
9:         buf $\leftarrow l + \frac{(g-l)}{2}$
10:        thres $\leftarrow a +$ buf $\times (u - a)$
11:        nz $\leftarrow$ count nonzero($a \geq$ thres)
12:        if nz $\leq k$ then
13:            $g \leftarrow$ buf
14:            if nz $> k1$ then
15:                $k1 \leftarrow$ nz
16:                t1 $\leftarrow$ thres
17:            end if
18:        else if nz $> k$ then
19:            $l \leftarrow$ buf
20:            if nz $< k2$ then
21:                $k2 \leftarrow$ nz
22:                t2 $\leftarrow$ thres
23:            end if
24:        end if
25:    end for
26:    $p1 \leftarrow$ nonzero indices($a \geq$ thres1)
27:    $p2 \leftarrow$ nonzero indices(($a <$ thres1) and ($a \geq$ thres2))
28:    rand $\leftarrow$ random(0, len($p2$) $- (k - k1) + 1$)
29:    $G_s \leftarrow G$[concat($p1, p2$[rand : rand $+ k - k1$])]
30:    $Q$ initialized from i.i.d std. normal distr.
31:    $P \leftarrow G_s \cdot Q$                      ▷ POWERSGD($G_s, r$)
32:    $P \leftarrow \frac{1}{W} \cdot (G_{s1} + \ldots + G_{sW}) \cdot Q$
33:    $\hat{P} \leftarrow$ ORTHOGONALIZE($P$)
34:    $Q \leftarrow G_s^\top \cdot \hat{P}$
35:    $Q \leftarrow \frac{1}{W} \cdot (G_{s1} + \ldots + G_{sW})^\top \cdot \hat{P}$
36:    return $\hat{P} \cdot Q^\top$
37: end function
```

TABLE I
ACCURACY RESULTS FOR RESNET20 TRAINED ON CIFAR10

Number Format	Block Size	w/o pSGD	r-1 pSGD	r-2 pSGD	r-4 pSGD	r-2 in last 30 epochs; r-4 in rest	r-1 in last 30 epochs; r-4 in rest
FP32	-	91.74%	92.04%	92.14%	92.13%	92.17%	92.01%
HBFP8	64	92.05%	92.06%	92.11%	92.21%	92.23%	92.03%
	576	92.29%	91.92%	91.92%	92.12%	92.31%	92.09%
HBFP6	64	91.70%	91.23%	91.34%	91.70%	91.86%	91.66%
	576	91.35%	91.35%	91.55%	91.63%	91.1%	91.29%
HBFP5	64	90.32%	90.35%	90.33%	90.23%	90.30%	89.85%
	576	89.34%	89.44%	89.43%	89.54%	89.57%	89.74%
HBFP4	64	81.85%	80.35%	81.28%	82.22%	82.95%	82.33%
Per-tensor Compression	**-**	**-**	**∼243x**	**∼122x**	**∼61x**	**∼122x, ∼61x**	**∼243x, ∼61x**

TABLE II
ACCURACY RESULTS FOR RESNET50 TRAINED ON CIFAR100

Number Format	Block Size	w/o pSGD	r-1 pSGD	r-2 pSGD	r-4 pSGD
FP32	-	73.23%	72.97%	73.56%	74.65%
HBFP8	64	73.02%	72.88%	73.45%	73.88%
	576	74.80%	73.91%	73.47%	73.16%
HBFP6	576	71.41%	71.75%	71.52%	71.48%
HBFP5	64	66.87%	65.38%	65.64%	66.42%
	576	63.88%	64.19%	64.29%	63.57%
Per-tensor Compression	**-**	**-**	**∼268x**	**∼134x**	**∼67x**

We also experiment by training of large language models along with the Gradient-Compressed HBFP. As shown in [**?**], we would require a larger rank for training language models. Similar to the original PowerSGD implementation, we use rank-32 for training of a transformer on the IWSLT14 German-English dataset. We obtained a BLEU score of 27.8 when using rank-32 PowerSGD, as compared to 28.37 in the case when PowerSGD was not used. A compression of 14x is achieved when using 32x compression. In both implementations, we used HBFP8 with a moderate block size of 64.

IV. OPTIMIZING GC: POWERSGD+MSTOPK

As could be observed in Section III, the accuracy of even large neural networks does not drop on using PowerSGD GC (even with very low ranks). This motivated us to compress the gradients even further, in order to get even higher compression ratios, while tolerating some loss in accuracy.

Previous works have proposed that the accuracy of neural networks remains maintained even when we drop 99.9% of the gradients while communicating (MSTopK GC technique). The major disadvantage of this technique where PowerSGD provides benefits, is the inability to use the All-Reduce technique in the former techniques. We propose to integrate sparsification and low-rank decomposition, such that we benefit from the use of All-Reduce technique, as well as get very low drops in accuracy.

Fig. 3 shows the different ways in which sparsification could be integrated with low-rank decomposition. In the first method, the gradient matrix is first decomposed, and the resultant matrices are then sparsified. Similarly, another way is to first sparsify the gradient matrix, and then decompose this sparsified matrix into two low-rank matrices. A third way is to simultaneously decompose and sparsify the gradient matrix; and then add the resultant matrices. One of the important aims of this integration is that it should be compatible with the All-Reduce technique. This constraint directs us to use the second way of integrating sparsification and low-rank decomposition. As mentioned earlier, we use the MSTopK-based approach for sparsification, and then, use the PowerSGD update.

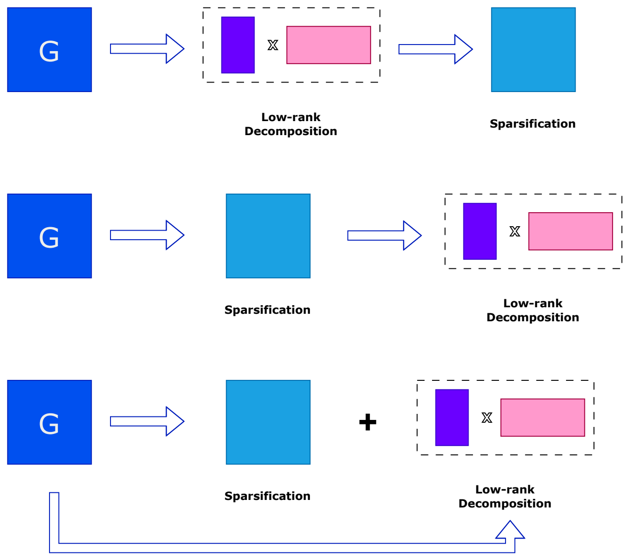

Fig. 3. Three ways to merge sparsification and low-rank decomposition

Algorithm 2 shows the approach of integrating the two techniques. Like the MSTopK technique, we first use binary search to find the optimal threshold. N denotes the number of searches for finding the threshold. Suppose we want to select k elements from the input gradient vector. The threshold is initialised as the mean of the absolute numbers in the vector. The numbers greater than or equal to this are kept, and the rest are dropped. If the number of these elements are greater (or lesser) than k, the threshold is halved (or doubled). The number of such searches of finding the thresholds is limited by N. As shown in [4], this approximate topK method is quite efficient on GPUs because it doesn't have any expensive memory access operations. Once we get the sparsified gradient vector, we could plug it into the PowerSGD algorithm (shown in Algorithm 1) to get the low-rank decomposition of this sparsified matrix.

To show the benefits of the technique in terms of compression ratio, we show below the calculation of per-tensor compression for the proposed technique in terms of the PowerSGD rank, r and the topK value, k.

As shown in [4], the compression ratio for a particular value of k is given by $(\frac{1}{k})$. For Resnet20, the per-tensor compression as a function of r is shown to be $(\frac{243}{r})$x.
Plugging in the two values, we could easily show the per-tensor compression for the proposed technique to be as follows:

Per-tensor Compression $= \frac{243}{r.k}$ (For Resnet20)

As an example, if we consider dropping 80% gradients and decomposing gradients with rank-1, we could get nearly 1215x compression, as compared to 243x when only using PowerSGD. Similarly, the compression ratio for Resnet50 for the same parameters is nearly 1340x.

A. Results (CNNs)

We validate the technique by training Resnet20 on Cifar10, and then, use the best-case parameters for training Resnet50 on Cifar100. Table III shows the results for the training of Resnet20.

TABLE III
ACCURACY RESULTS FOR RESNET20 TRAINED ON CIFAR10 (USING PROPOSED GC)

Number Format	**Block Size**	***(k, r)***					
		(-,-)	**(0.2, 1)**	**(0.2, 2)**	**(0.1, 1)**	**(0.1, 2)**	**(0.01, 2)**
FP32	-	91.74%	90.72%	91.05%	89.47%	89.56%	83.05%
HBFP8	64	92.05%	90.51%	90.59%	89.39%	89.93%	82.67%
	576	92.29%	90.53%	91.10%	89.28%	89.80%	
HBFP6	64	91.70%	90.10%	90.02%	88.43%	89.02%	80.39%
HBFP5	64	90.32%	87.83%	88.25%	86.04%	86.07%	
Compression	-	**-**	**∼1215x**	**∼608x**	**∼2430x**	**∼1215x**	**∼12150x**

As could be observed from Table (III), using r-1 pSGD after dropping 80% of the gradients compresses gradients by 1215 times, while losing 1% of the accuracy for FP32. A similar observation could be made for HBFP as well, where the accuracy drop is nearly 1.5%. The compression ratio is much larger as compared to PowerSGD (where the maximum compression is nearly 243x for Resnet20) and also as compared to MSTopK (where the compression after dropping 99.9% gradients is 1000x).
This technique could also provide compression as high as 2430x but with a greater trade-off in terms of accuracy (-2.5%).

A similar observation was made while training Resnet50 on Cifar100 where we observe an accuracy drop of 1.2% for HBFP8 and 1.5% for FP32 when using (k, r) = (0.2, 2) with a compression of 670x.

As could be seen, the proposed technique provides very high compression ratios with a tolerable loss in accuracy. Another advantage of the proposed technique is the ability to use the All-Reduce technique, which is a major hindrance in most of the other GC techniques.

B. Results (Language Models)

Table (IV) shows the results for the training of the transformer on the IWSLT14 German-English dataset. As could be observed, the BLEU score vanishes when using ¿50 % percentage of gradient drop. Results are reported for various percentages of gradient drops to identify the limit of the fraction of gradients that could be dropped when using the proposed implementation.

TABLE IV
BLEU SCORES FOR TRANSFORMER (USING PROPOSED GC AND HBFP8 WITH BLOCK SIZE 64)

	(-, -)	**(0.9, 32)**	**(0.75, 32)**	**(0.5, 32)**	**(0.3, 32)**
BLEU	28.37	28.2	24.68	11.13	6.56
Compression	-	∼15.55x	∼18.67x	∼28x	∼46.67x

As could be observed, the proposed implementation does not provide many benefits in terms of compression ratio when used along with PowerSGD.

V. CONCLUSION

In this project, we tried to alleviate the two major issues in distributed DL: arithmetic density (using HBFP) and communication bottleneck (using gradient compression). We first show the benefits of HBFP as compared to other low-precision number formats, particularly bfloat16. We then tested the viability of gradient-compressed HBFP by using PowerSGD with HBFP for both small and larger neural networks. We observed that it provides an improvement over the baseline accuracy in most cases, while reducing the communication time. Observing no drop in accuracy, we then propose a novel gradient compression technique that provides even greater compression ratio with a very low drop in accuracy. However, we still need to test the viability of this new technique for other larger networks, and for other important applications.

VI. ACKNOWLEDGEMENTS

I would like to thank Simla Harma for her guidance and help throughout the course of the project. I would also like to thank Ayan Chakraborty and Rafael Pizarro for their help during the initial phase of the project. Finally, I would also like to thank Prof. Babak Falsafi for providing me the opportunity to pursue this project.

REFERENCES

[1] S. B. Harma, A. Chakraborty, B. Falsafi, M. Jaggi, and Y. Oh, "Accuracy boosters: Epoch-driven mixed-mantissa block floating-point for dnn training," 2023.

[2] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," 2020.

[3] T. Vogels, S. P. Karimireddy, and M. Jaggi, "Powersgd: Practical low-rank gradient compression for distributed optimization," 2020.

[4] S. Shi, X. Zhou, S. Song, X. Wang, Z. Zhu, X. Huang, X. Jiang, F. Zhou, Z. Guo, L. Xie, R. Lan, X. Ouyang, Y. Zhang, J. Wei, J. Gong, W. Lin, P. Gao, P. Meng, X. Xu, C. Guo, B. Yang, Z. Chen, Y. Wu, and X. Chu, "Towards scalable distributed training of deep learning on public cloud clusters," 2020.

[5] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training dnns with hybrid block floating point," 2018.

[6] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pp. 88–91, 2019.

[7] G. Raposo, P. Tomas, and N. Roma, "Positnn: Training deep neural networks with mixed low-precision posit," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, jun 2021.

[8] S. Agarwal, H. Wang, S. Venkataraman, and D. S. Papailiopoulos, "On the utility of gradient compression in distributed training systems," *CoRR*, vol. abs/2103.00543, 2021.