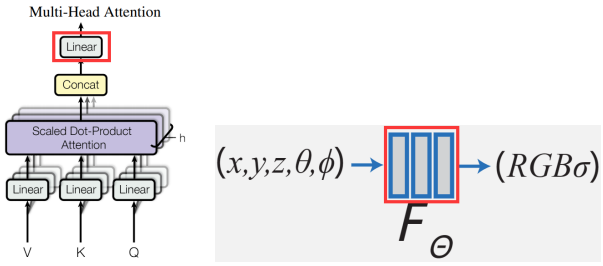
A Workflow for High Performance and High Fidelity Neural Network Compression

Yifan Chen, Letong Han, Xuanhang Diao
Shanghaitech University
Shanghai, China
{chenyf2022, hanlt, diaoxh2022}@shanghaitech.edu.cn

***Abstract*—Deep neural networks have demonstrated impressive capabilities in many domains. However, as these models become increasingly complex, they require hardware with increasingly demanding performance and specifications. While various deep learning model compression methods have been widely discussed, recent developments in deep learning, particularly in neural rendering and large language models, have raised the bar for the effectiveness, speed, and robustness of compression methods. To address these new challenges, we propose a deep learning model compression workflow that combines multiple compression strategies with Graphics Processing Unit (GPU)-based parallel acceleration for downstream tasks in neural rendering. Our test results demonstrate the effectiveness of our workflow on raw neural rendering tasks, and theoretically our workflow can be extended to other deep learning models as well.**

I. INTRODUCTION

Multi-layer perceptron (MLP) is a type of neural network that consists of multiple fully connected layers (FC). It is one of the simplest neural network structures but has powerful expressive ability. Theoretically, it can approximate any expression infinitely. Additionally, MLP has good generalization performance and is suitable for solving various problems. Hence, it is widely used in the design of deep neural networks. For instance, as shown below1, the Transformer [1] structure used in large language models relies on a two-layer MLP to aggregate multi-head attention scores, while the Neural Radiance Field (NeRF [2]) uses MLP to continuously describe objects and generate new perspectives.



(a) Transformer (b) NeRF

Fig. 1. As shown in the red box in the figure, MLP exists widely in various neural network structures

The drawback of MLP is its large parameter scale, which makes training difficult and hinders its inference speed and application deployment. For instance, the parameter of single-layer FC1 is about $H \times W/K^2$ times that of single-layer convolutional neural network (CNN)2 [3]. Here, H and W are the length and width of the input image, and K is the size of the convolution kernel selected in CNN. Since smaller convolution kernels are usually preferred in practice, and even the simplest MNIST dataset has an image size of 28×28, FC has a significant disadvantage in the number of parameters. Similarly, NeRF requires separate rendering for each scene and also needs to consider deployment capabilities on low-storage and low-computing devices, making the original MLP unsuitable. Hence, it is crucial to reduce the model's size to ensure its inference speed and deployment ability.

$$FLOPs_{MLP} \sim (H \times W)^2 \times C_{in} \times C_{out} \quad (1)$$

$$FLOPs_{CNN} \sim H \times W \times C_{in} \times K^2 \times C_{out} \quad (2)$$

Neural network compression methods, such as Pruning, Quantization, and Tensor Decomposition, have been widely discussed [4]. Quantization has even been integrated into popular neural learning frameworks like Pytorch. To test its effectiveness, we applied INT8 quantization to an 8-layer MLP used in an excavator scene in NeRF. The resulting model size decreased to $1/3.87$ of the original, and the inference speed increased by 2.47 times on the same hardware. However, the reconstruction quality index was only $2/3$ of the original, as measured by Peak Signal-to-Noise Ratio (PSNR) for image quality evaluation.

It is worth noting that MLP compression methods, such as low-bit representation or removing lower weights, may not be universally applicable. For instance, the winner-take-all feature of classification networks can make the weight of many parameters close to 0. However, this does not apply to implicit neural expressions, and simply destroying the neural network structure can adversely affect its performance. Therefore, certain constraints must be imposed on the neural network during compression to ensure its optimal performance.

We propose a high-performance and high-fidelity neural network compression workflow that utilizes GPU-based quantization and low-rank constraint strategies to achieve efficient and performant neural network compression. Section II provides background information on neural network compression and optimization strategies, while Section III describes our algorithm design in detail. Sections IV and V present and analyze our experimental design and results, respectively, and

in the final section, we summarize the full text and discuss the possibility of future expansion work.

II. Background

A. Model Compression Techniques

Pruning is a powerful technique to reduce the number of deep neural networks parameters. In DNNs, many parameters are redundant because they do not contribute much during training. So, after training, such parameters can be removed from the network with little effect on accuracy2.

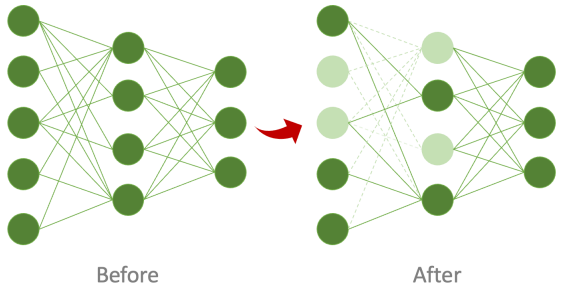


Fig. 2. Weight connections that are below some predefined thresholds are pruned, or prune the neurons

In DNNs, weights are stored as 32-bit floating point numbers. Quantization compresses the original network by reducing the number of bits required to represent each weight. For example, the weights can be quantized to 16-bit, 8-bit, 4-bit and even 1-bit. By reducing the number of bits used, the size of the DNN can be significantly reduced3.

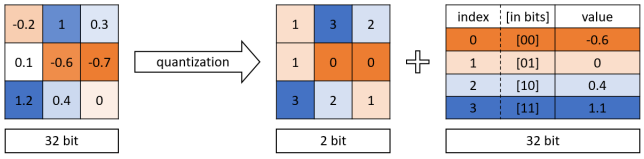


Fig. 3. A (little extreme) example. The representation of the weight parameter is changed from 32bit to 2bit

Low-rank factorization identifies redundant parameters of deep neural networks by employing the matrix and tensor decomposition. When reducing the model size is necessary, a low-rank factorization technique helps by decomposing a large matrix into smaller matrices, the main challenge is that the decomposition process results in harder implementations and is computationally intensive4.

B. Distributed Alternating Direction Method of Multipliers(Distributed ADMM)

ADMM [5] is a computational framework for solving optimization problems, suitable for solving distributed convex optimization problems, especially statistical learning problems. ADMM decomposes a large global problem into multiple smaller and easier-to-solve local sub-problems through the Decomposition-Coordination process, and obtains the solution to the large global problem by coordinating the solutions of the

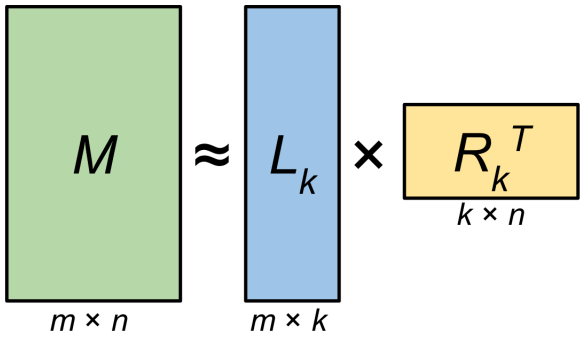


Fig. 4. A weight matrix M with $m \times n$ dimension and having a rank r can be decomposed into smaller matrices.

sub-problems. Most of the problems that meet the following conditions can be optimized using ADMM without concern of convergence issues

- Two optimization variables
- Only equality constraints

Consider a problem of the form

$$\min f(x) + g(z)$$
$$s.t.\ Ax + Bz = c$$

Turn Primal problem to Lagrangian function

$$L_\rho(x, z, u) = f(x) + g(z) + u^T(Ax + Bz - c) \tag{3}$$

define augmented Lagrangian, for a parameter $\rho > 0$

$$L_\rho(x, z, u) = f(x)+g(z)+u^T(Ax+Bz-c)+\frac{\rho}{2}\|Ax+Bz-c\|_2^2 \tag{4}$$

Repeat for $k = 1, 2, 3, ...$

$$x^{(k)} = argmin_x L_\rho(x, z^{(k-1)}, u^{(k-1)})$$
$$z^{(k)} = argmin_z L_\rho(x^{(k)}, z, u^{(k-1)})$$
$$u^{(k)} = u^{(k-1)} + \rho(Ax^{(k)} + Bz^{(k)} - c)$$

and we achieve Transform a multi-objective problem into multiple single-objective optimization problems, then for single-objective optimization, refer to common single-objective optimization algorithms such as gradient descent (GD).

For the distributed ADMM, consider to compensate matirx A and b as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \dots & \mathbf{A}_{1N} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \dots & \mathbf{A}_{2N} \\ & & \dots & \\ \mathbf{A}_{M1} & \mathbf{A}_{M2} & \dots & \mathbf{A}_{MN} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \dots \\ \mathbf{b}_M \end{bmatrix} \tag{5}$$

then problem transfer to

$$\min \sum_j f_j(x_j) + \sum_i g_i(z_i)$$
$$s.t.\ \sum_j P_{i,j}x_j + z_i = b_i, \forall i$$
$$P_{i,j} = A_{i,j}X_j, \forall i, j$$
$$x_j = x_{ij}, \forall i, j$$

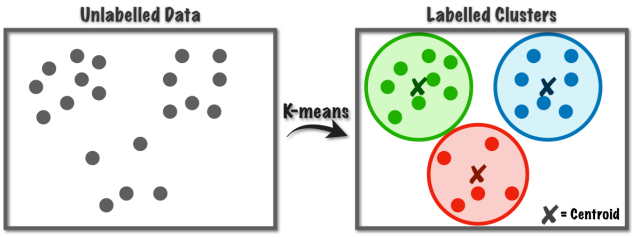


Fig. 5. K-Means aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

C. K-Means algorithm and its parallelization

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. A cluster refers to a collection of data points aggregated together because of certain similarities. Define a target number k, which refers to the number of centroids in the dataset. A centroid is the imaginary or real location representing the center of the cluster. In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible5.

The clustering algorithm requires massive computations, with distance between each data point and each centroid being calculated. Since calculation of the appropriate centroid for each data point is independent of the others, the algorithm provides a good scope for parallelism. Note that here we describe the possibility of K-Means parallelization. Parallelism at different levels of abstraction, such as instruction-level parallelism and task-level parallelism, is not distinguished. Therefore, the unit in the following text does not refer to a single thread or node in a narrow sense, but the smallest processing unit in parallelization.

For parallelization of the k-means algorithm [6], a data-parallelism approach was adopted. The first k data points are chosen as the initial, each processing unit assigned N/num_{unit} number of points for Data-parallelism. Each unit runs a loop, in every iteration of which it computes the closest cluster centroid for every point assigned to it, and then assigns the point to that cluster. After every point is assigned to a cluster, the global cluster centroids are updated with the values computed for their coordinates from the points that were assigned to that cluster. This is again an instance of data-parallelism centroids. The L2-norm is computed for every cluster centroid coordinates by comparing against corresponding values in the previous iteration. The norms are then summed and compared against the threshold value. All units break from the iterations loop as soon as the delta value goes below the threshold.

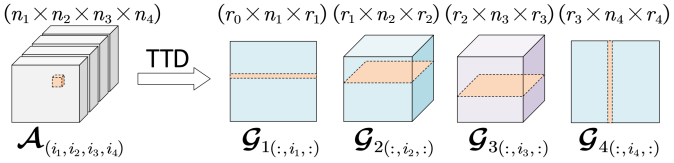


Fig. 6. An attractive property of the TT-decomposition is the ability to efficiently perform several types of operations on tensors if they are in the TT-format: basic linear algebra operations, such as the addition of a constant and the multiplication by a constant, the summation and the entrywise product of tensors (the results of these operations are tensors in the TT-format generally with the increased ranks); computation of global characteristics of a tensor, such as the sum of all elements and the Frobenius norm.

III. Method

A. Low-rank Representation

Fully-connected layers apply a linear transformation to an N-dimensional input vector x

$$y = Wx + b \tag{6}$$

A TT-layer transforms a d-dimensional tensor χ (formed from the corresponding vector x) to the d-dimensional tensor Γ (which correspond to the output vector y). We assume that the weight matrix W is represented in the TT-format with the cores $G_k[i_k, j_k]$. Finally the linear transformation of a fully-connected layer can be expressed in the tensor form [7]

$$\gamma(i_1, ..., i_d) = \sum_{j_1,...,j_d} G_1[i_1, j_1]...G_d[i_d, j_d]\chi(j_1, ..., j_d) \tag{7}$$

$$+\beta(i_1, ..., i_d) \tag{8}$$

The representation of a tensor A6 via the explicit enumeration of all its elements requires to store $\Pi_{k=1}^{d} n_k$ numbers compared with $\sum_{k=1}^{d} n_k r_{k-1} r_k$ numbers if the tensor is stored in the TT-format. Thus, the TT-format is very efficient in terms of memory if the ranks are small.

The weight matrix W can be decomposed into the product of multiple Tensors

$$W_i = Q_i^1 \cdot Q_i^2 \tag{9}$$

where $Q_i^1 \in \mathbb{R}^{1 \times n_1 \times r}$ and $Q_i^2 \in \mathbb{R}^{r \times n_2 \times 1}$ are the TT cores, and r is the target rank controlling the compression ratio. As we discussed in IIA, straightforwardly decomposing the full-rank tensor W_i into a low-rank TT format inevitably causes a large approximation error, so we adopt this ADMM-based low rank approximation to finetune the learned NeRF.

$$argmin_{\{W_i,b_i\}} ||c' - c_{gt}||_2^2$$
$$s.t. \;\; rank(W_i) < r,$$

where r is the desired rank of W_i. This non-convex optimization with constraints can be solved using the ADMM optimization method by introducing an auxiliary variable Z and an indicator function $g(\cdot)$

$$g(W_i) = \begin{cases} 0, rank(W_i) < r \\ +\infty, otherwise. \end{cases} \tag{10}$$

The original optimization problem can then be rewritten

$$argmin_{\{W_i,Z_i\}}||c' - c_{gt}||_2^2 + g(Z_i)$$
$$s.t.\ \ W_i = Z_i$$

Refer to the general steps of ADMM, we get two subproblems

- W-subproblem

$$\min_{W} \quad l(W_i) + \frac{\rho}{2}||W_i - Z_i + U_i|| \tag{11}$$

  Can be solved by Stochastic Gradient Descent (SGD)
- Z-subproblem

$$\min_{Z} \quad g(Z_i) + \frac{\rho}{2}||W_{i+1} - Z_i + U_i|| \tag{12}$$

updating Z can be performed as:

$$Z_{i+1} = \Pi_S(W_{i+1} + U_i) \tag{13}$$

Where $\Pi_S(\cdot)$ is the projection of singular values onto S, which is done by truncating ranks to target ranks r^*. Singular Value Decomposition (SVD) for real matrices

$$M = U\Sigma V^T \tag{14}$$

Here M is a $m \times n$ matrix, while U is $m \times m$ orthogonal unitary matrix, Σ is $m \times n$ diagonal matrix, and V is $n \times n$ orthogonal unitary matrix.

Finally, update U can be performed as

$$U_{i+1} = U_i + \rho(W_{i+1} - Z_{i+1}) \tag{15}$$

B. Quantization

The concept behind model quantization is to represent deep neural network weights using a smaller scale [8]. This can be achieved by two methods: first, by representing weights with similar values using the same value, and second, by using a lower number of bits to represent weights with less precision. For example, instead of using $float32$, 3-bit, 2-bit, or even 1-bit can be used.

In addition, after analyzing the weight distribution in the NeRF multi-layer perceptron, it was observed that the weights are mostly centered around 0.

Based on these observations, we specify the following quantization strategy

- All bias items are represented using single-precision floating point numbers (float32).
- Other weights are quantized and represented using 3-bit quantization.

Taking 3-bit quantization as an example. To compress the deep neural network weights, a codebook $\mathcal{C} = \{c_1, c_2, \cdots, c_K\}$ of size K is learned. Each weight W_i is represented using a binary variable $z_i \in \{0,1\}^K$, using a binary variable $z_i \in \{0,1\}^K$ with $\sum_{k=1}^{K} z_{i,k} = 1$ for each weight W_i, the compression goal is

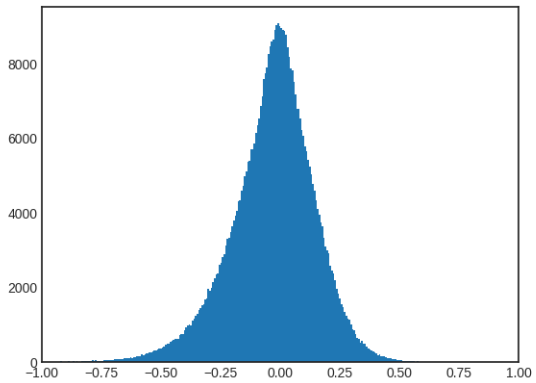


Fig. 7. The vertical axis is the number of current weights. The horizontal axis is the weight value

$$\min_{W,\mathcal{C},z} \quad l(W)$$
$$s.t.\ \ W_i = \sum_{k=1}^{K} c_k z_{i,k}$$

- W-subproblem

$$\min_{W} \quad l(W^{(i)}) + \frac{\rho}{2}||W^{(i)} - Z^{(i)} + U^{(i)}|| \tag{16}$$

  Can be solved by Stochastic Gradient Descent (SGD)
- Z-subproblem

$$\min_{Z} \quad \frac{\rho}{2}||W^{(i+1)} - Z^{(i)} + U^{(i)}|| \tag{17}$$

Furthermore, the Z-subproblem is essentially

$$\min_{z,\mathcal{C}} \quad \sum_{i=1}^{P}\sum_{k=1}^{K} z_k(W_i - c_k)^2 \tag{18}$$

which is known as the K-Means problem.

Relying on the K-Means parallelization idea mentioned in Section IIC, we can reduce time overhead through GPU acceleration.

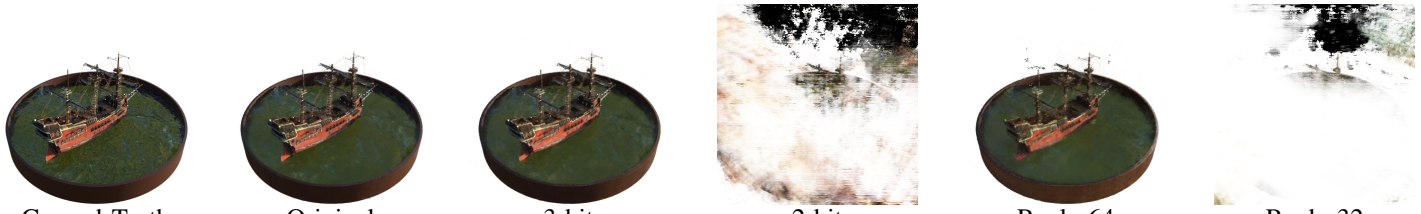
IV. RESULT

Dataset. We have evaluated our approach on two different scenes in Synthetic-NeRF dataset proposed in [2]. It is a popular dataset, containing 8 different realistic objects each, which are synthesized from NeRF (chair, drums, ficus, hotdog, lego, materials, mic and ship). The scenes we tested here is lego and ship. Due to the limited resources and time, the image resolution has been set up to 400×400 pixels (which is the half size of the original 800×800 pixels) , having 200 views for training and 200 for validation.

Training. To simplify the architecture of NeRF, we adopted a single MLP for inference with a depth of 8 and 256 channels. We set the sampling rate along the rays from 64 to 128

Ground Truth	Original	3-bit	2-bit	Rank=64	Rank=32
—	PSNR = 31.037dB	PSNR = 30.112dB	PSNR = 27.633dB	PSNR = 27.079dB	PSNR = 25.291dB
—	SIZE = 2.26MB	SIZE = 0.68MB	SIZE = 0.63MB	SIZE = 1.39MB	SIZE = 0.95MB

Fig. 8. The results of the Lego scene. The model size is significantly reduced, regardless of the compression strategy used.



Ground Truth	Original	3-bit	2-bit	Rank=64	Rank=32
—	PSNR = 29.729dB	PSNR = 28.515dB	(failed to synthesize)	PSNR = 24.763dB	(failed to synthesize)
—	SIZE = 2.26MB	SIZE = 0.68MB		SIZE = 1.39MB	

Fig. 9. The corsair(ship) scene is generally similar to the Lego scene. However, it should be noted that under certain parameter settings, the scene failed to synthesize.

and used normalized depth for rendering. During training, we followed the default configuration in the original NeRF and set the learning rate to 5×10^{-4}. The network was trained for 1.5×10^5 iterations.

We first trained a simplified NeRF with full rank. This initial model is then finetuned with two compression tasks:

- Quantization: 3-bit and 2-bit
- Low Rank: $r \in \{64, 32\}$

V. Discussion

All the results are reported in Fig 8 and Fig 9. Although our proposed method effectively compresses the implicit field, its performance is limited by factors that also limit the performance of NeRF, since our method is based on it. In particular, the compression performance is limited by complex scenes. We achieved better performance on the lego scene, which does not include many details.

The appropriate setting of parameters is crucial in various application scenarios. For example, in the context of implicit neural expression, inadequate compression of parameters may only have a marginal impact on the quality of reconstruction from certain perspectives. However, in conventional deep learning tasks like classification and regression, insufficient compression can render the network completely ineffective for certain inputs. Therefore, when extending the application of our approach to other scenarios, it is imperative to set reasonable parameters, conduct extensive tests, and incorporate layer-by-layer codebooks, layer-by-layer compression, and more refined adjustments to minimize performance loss.

a) Lego: For the Lego scene, our original model achieved a PSNR of 31.037dB. We observed that under the 3-bit quantization strategy, using a model size of 30.09% of the original model achieved a PSNR of 97.02%. However, when we attempted to quantize the model to 2 bits, the image quality sharply decreased, resulting in more "hollows" in the margin of the object. This was caused by the model learning the wrong parameter for σ. We also tried low-rank compression, which constrains the maximum rank of some layers' weight matrix. However, the results were not as good as quantization. With a maximum rank of 64, the model was compressed to 1.39 MB, but the PSNR decreased by almost 4dB to 27.079dB, which was outperformed by the 3 bit quantized model.

b) Ship: For the Ship scene, we achieved similar results to the Lego scene under both conditions of 3-bit quantization and rank=64 constraints. In quantized compression, we obtained a PSNR of 95.92% while using a model size of 30.09% compared to the original model. In low-rank constrained compression, we obtained a peak signal-to-noise ratio of 83.30% using a model size of 61.50%. However, we noticed that we were unable to complete the rebuild under certain conditions. This may be due to the lack of camera pose constraints in some perspectives of this scene [9]. We tested a number of different camera parameters and found that reconstruction was only possible for certain viewpoints.

VI. Conclusion & Future Works

In this project, we presented a compressing workflow that utilizes low-rank approximation and quantization to compress NeRF models. Our approach removes or clusters parameters from the model while ensuring that there is not a large drop in performance. Since this is a post-processing method, it is

easily deployable for any model, regardless of its architecture, training strategy, or objective function.

It is worth noting that although our experiments only verified the effectiveness of our method on the original NeRF, our compression strategy and acceleration method should be applicable to any other variants and applications of NeRF. In fact, our method can theoretically be applied to any type of neural network.

REFERENCES

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020.

[5] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

[6] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. A parallel implementation of k-means clustering on gpus. In *Pdpta*, volume 13, pages 212–312, 2008.

[7] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. *Advances in neural information processing systems*, 28, 2015.

[8] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[9] Yoonwoo Jeong, Seokjun Ahn, Christopher Choy, Anima Anandkumar, Minsu Cho, and Jaesik Park. Self-calibrating neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5846–5854, 2021.