Distributed Gradient Preconditioning for Training Large-Scale Models

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Gradient Preconditioning for Training Large-Scale Models

Master Thesis

N. A. Baumann

May 2, 2023

Supervisors: Prof. Dr. T. Hoefler, Dr. K. Osawa, Dr. S. Li
Department of Computer Science, ETH Zürich

Acknowledgements

First of all, thank you Kazuki and Shigang for supervising me on this project. It was a great project where I learned a lot about using deep learning and distributed training, which is, in today's world, a major skill to have. Special thanks to Kazuki and Shigang, your knowledge on this topic is outstanding and thank you for guiding me through it to the point where I am today. A very big thank you goes to Professor Torsten Hoefler for giving constructive feedback. Another thanks goes to my girlfriend Julia for reading my thesis. Thanks to Grammarly for helping me write clear and correct sentences. Lastly to my family, a big thank you for standing next to me and being there for me when I needed you.

Abstract

Neural Networks (NNs) are getting deeper and more complicated to the point where single accelerator training is no longer an option. Training today's state-of-the-art NNs is done in parallel over thousands of GPUs. Preconditioning-based optimizers are getting more attention in distributed training as well. We conduct a literature review of existing distributed second-order methods in training NNs. We thoroughly look at two famous preconditioning methods called K-FAC and Shampoo and describe the approaches on how to distribute additional computations across multiple GPUs. We implement distributed K-FAC (distr. K-FAC) and distributed Shampoo (distr. Shampoo) in PyTorch. Based on our analysis of the performance of both algorithms, we introduce *3D-Shampoo*, an extension of Shampoo to training in 3D parallelism settings (i.e. a combination of data, operator, and pipeline parallelism). 3D-Shampoo works with 3D parallelism from the DeepSpeed library (*Rasley et al.* ,2020), a modified version of the Shampoo optimizer (*Gupta et al.* ,2018), and is designed for very big language models such as GPT-2 which support operator parallelism like Megatron-LM's GPT-2 (*Narayanan et al.* ,2021). The final part of this thesis consists of a description of the 3D-Shampoo algorithm, how it works, and the results of its performance on Megatron-LM's GPT-2 for different levels of parallelism. Further, our 3D-Shampoo has shown a competitive throughput (number of tokens processed per second) with the SGD optimizer for all kinds of parallelism (data parallelism, operator parallelism, pipeline parallelism, and a combination of them) training GPT-2-like Transformer models. The code used for our experiment is publicly available[1].

[1] `https://github.com/noabauma/3d-shampoo`

Contents

Chapter 1

Introduction

Deep neural networks (DNNs) are getting bigger and have revolutionized the field of machine learning by achieving state-of-the-art performance on a wide range of tasks, from image and speech recognition to natural language processing, such as ChatGPT from OpenAI[1]. We have gotten to a point where neural networks are the size of hundreds (or even thousands) of Gigabytes (GBs) with current GPUs have memory capacity in the range of 16 GB up to 80 GB with the new Nvidia A100[2]. Hence, distributed training over multiple GPUs is a must-do to train the newest generations of neural networks. Training those neural networks is computationally expensive, particularly for large-scale problems with high-dimensional data. This has motivated the development of more efficient optimization methods to accelerate training while maintaining or improving performance.

One approach to improving optimization is to use second-order methods (or preconditioning methods in particular), which take into account the curvature of the loss function when updating the network parameters. Preconditioning methods can potentially converge faster and generalize better than widely-used first-order methods, but they can be computationally expensive, high in memory cost, and difficult to scale to large datasets and deep neural networks.

In recent years, there has been increasing interest in distributed optimization methods that can leverage the power of multiple processors or GPUs to accelerate training. However, existing distributed optimization methods for deep neural networks have mostly focused on first-order methods, and the use of second-order methods in a distributed environment remains an open research field.

Our goal is to identify new research directions for improving the distributed

[1] https://chat.openai.com/

[2] https://www.nvidia.com/en-us/data-center/a100/

training of deep neural networks and to gain a deeper understanding of the advantages and limitations of preconditioning methods in a distributed setting. We also want to show the potential of preconditioning methods in large-scale models on high-performance computer clusters having multiple GPUs. In this thesis, we investigate distributed preconditioning methods for DNNs, focusing on the K-FAC [23] and Shampoo [15] algorithms. We also propose a new method called 3D-Shampoo, which combines Shampoo with DeepSpeed's (ZeRO-style) data and pipeline parallelism [32] and Megatron-LM's language models [34, 25] supporting operator/tensor parallelism to further accelerate training on multiple GPUs.

This thesis is organized as follows: Chapter 2 is a literature review of existing distributed preconditioning methods. We discuss there what has been done already and why this thesis came to be. Chapter 3 is about the distributed Kronecker-Factored Approximate Curvature (distr. K-FAC) algorithm and its computational and communications cost. In chapter 4 is about the distributed Shampoo (distr. Shampoo) algorithm and its computational and communications cost. Chapter 5 we analyse the results of both the distr. K-FAC and distr. Shampoo algorithm for different numbers of GPUs, NNs, Batch sizes and more on the supercomputer Piz Daint from the Swiss National Supercomputing Centre (CSCS) [3]. Chapter 6 is about 3D-Shampoo's algorithm, implementation, and throughput results on Piz Daint against the SGD optimizer for various configurations.

Chapter 2

Existing Distributed Preconditioning Methods

For this chapter, we conducted a literature review of all known distributed preconditioning methods that have been tested on multiple GPUs (see table 2.1). We have included every known paper that demonstrates a 2nd-order method (or preconditioning) for updating the parameters of a network, including some sort of distributed/multi-GPU setting. The goal of this literature review is to show what distributed preconditioning methods are currently doing and how many GPUs their algorithms have been tested on. Some algorithms, such as K-FAC and its similar counterparts (e.g. M-FAC, HyLo), have been shown to work well on multi-GPU training, with *Osawa et al.* [28] being the largest with results on 1024 GPUs. We see potential in *Anil et al.* Shampoo optimizer [5] for its simplicity and scalability (see chapter 4 for more information on Shampoo) to work on very large models such as GPT-based language models. *Anil et al.* only demonstrated their concept for heterogeneous-based learning, where the p-root inverse computation of the preconditioning matrices was done on a CPU because of its double precision and the CPU being mostly stale during training. We see the potential of using Shampoo in a distributed layer-wise fashion and the p-root inverse computation done on GPUs and with some modification to work on 3D parallelism (see chapter 6).

Year	Category	Cited	Authors	Ref	Algorithm Name	Type of Parallelism	#GPUs
2022	NG	0	Mu+	[24]	HyLo	DP & layer-wise MP	64
2022	NG	1	Yang+	[42]	SENG	DP & layer-wise MP	32
2021	NG	9	Pauloski+	[29]	KAISA	DP & layer-wise MP	128
2021	Newton	26	Islamov+	[19]	NEWTON-STAR/-LEARN	MP	0 (142 CPU cores)
2021	NG	5	Chen+	[8]	THOR	layer-wise MP	Ascend 910 × 256
2021	NG / GN	5	Haider+	[16]	NGHF	DP	4
2020	adaptive grad	32	Anil+	[5]	Shampoo	Heterogenous training	1 TPU
2020	NG	26	Osawa+	[28]	K-FAC	DP & layer-wise MP	1024
2021	newton	5	Li+	[21]	DN-ADMM	MP	0
2020	newton	6	Fang+	[12]	Newton-ADMM	MP	30
2019	quasi-newton	4	Adya+	[2]	Distributed NLCG	DP	64
2019	quasi-newton	0	Liu+	[22]	Distributed L-BFGS	DP	1
2018	newton	31	Dunner+	[11]	ADN	DP & MP	0
2018	newton	113	Wang+	[41]	GIANT	own kind of DP	0 (480 CPU cores)
2021	NG	10	Tang+	[38]	SKFAC	DP	4
2018	newton	41	Yao+	[43]	ABSA	DP (in theory)	1
2021	NG	17	Frantar+	[13]	M-FAC	MP (in theory)	1
2020	NG	79	Singh+	[36]	WoodFisher	DP	4
2020	NG	19	Pauloski+	[30]	Distributed K-FAC	DP & layer-wise MP	256

Table 2.1: Table of existing papers on preconditioning distributed methods. "Year" is the year of publication. "Category" indicates the type of algorithm it belongs to. NG stands for Natural Gradient and GN for Gauss-Newton. "Cited" is how many times the paper has been cited at the time this thesis was written."Type of Parallelism" indicates the type of distributed training method used in the paper. "DP" stands for data parallelism and "MP" for model parallelism. "#GPUs" indicates how many GPUs were used in their experimental results. If 0 or 1 is given, they only have theoretical assumptions on how to run their algorithm in a distributed way. Some papers only used CPUs, still in a parallel fashion.

Chapter 3

Distributed Kronecker-Factored Approximate Curvature

This chapter explains our distributed K-FAC (distr. K-FAC) algorithm in detail with the computational and communication cost. Kronecker-Factored Approximate Curvature (K-FAC) [23] is a powerful deep-learning optimization technique that uses the Kronecker factors of matrices to approximate the curvature of the loss function. The main idea behind K-FAC is to use a low-rank approximation of the inverse of the Fisher Information Matrix (FIM), which captures the second-order derivatives of the loss function with respect to the model parameters.

K-FAC can be implemented for different Fisher types and shapes. The different types of Fisher can either be: exact, Monte-Carlo, or empirical, and the possible shapes for Fisher are full, layer-wise, unit-wise, or element-wise. In this project, we only consider computing the layer-wise computational and communication costs of K-FAC because distributed gradient preconditioning will be implemented for layer-wise K-FAC. When the FIM is approximated with the layer-wise K-FAC, it has the shape of a block diagonal matrix. A matrix that is of the shape of a block diagonal matrix can be inverted by the independent blocks individually. Computing the inverse of the FIM is the most expensive part of one training step, thanks to the block-diagonal shape of the FIM with their Kronecker factored blocks. We can compute the FIM in a parallel fashion over multiple GPUs. Depending on the level of parallelism (i.e. number of GPUs), we split the workload evenly such that only a certain GPU will compute the preconditioning matrix and precondition the gradients of the given layers. This method of computing the FIM in a parallel fashion falls into the category of the so-called model parallelism (MP). Any method/algorithm that distributes a single model across multiple devices can be referred to as MP. When implementing distributed K-FAC, we use the concept from *Osawa et al.* [28], which is a mixture of data parallelism

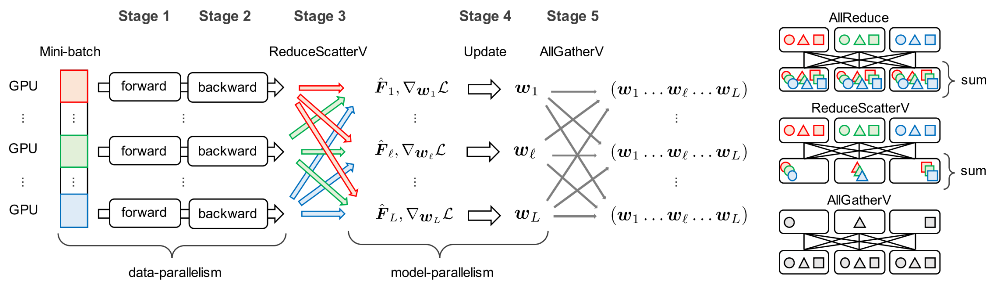


Figure 3.1: An overview of one training step of the distributed K-FAC from *Osawa et al.* [28], which they call Scalable and Practical Natural Gradient Descent (SP-NGD). To the right is a reminder of the collective communications `AllReduce`, `ReduceScatterV`, and `AllGatherV`. The figure is adopted from *Osawa et al.* [28].

(DP) and MP, named Scalable and Practical Natural Gradient Descent (SP-NGD). For information about DP see section 6.2. To achieve great accuracy of the empirical FIM, the SP-NGD algorithm from *Osawa et al.* [28] uses multiple train batches (called mini-batches) in a DP fashion to approximate the Kronecker factors. After the forward and backward pass, the individual Kronecker factors and gradients from each mini-batch will be averaged and distributed to the individual GPUs in a `reduce_scatter()` fashion. After computing the preconditioning matrices and preconditioned gradients, we `all_gather()` the gradients back to all the ranks to start over the training again. Figure 3.1 gives a great overview of the distributed K-FAC algorithm.

Before we move on to the next sections, we would like to mention that all of the notations in this chapter are adopted from the paper "Scalable and Practical Natural Gradient for Large-Scale Deep Learning" from *Osawa et al.* [28].

3.1 Concept and Computational Cost of K-FAC

In this section, we go through the algorithm of K-FAC in detail as well as what the computational cost for the different kinds of layers is: Fully-connected (section 3.1.1), convolutional (section 3.1.2), and unit-wise layers (section 3.1.3). Depending on the type of layer, K-FAC performs differently.

Consider a NN consisting of L-layers with $l \in \{1, 2, \dots, L\}$ being the layer index. At each layer, l and training step t, the updating step of K-FAC for the parameters w_l of the lth layer using the FIM preconditioned gradients are defined as

$$w_l^{(t+1)} \leftarrow w_l^{(t)} - \eta(\hat{F}_l^{(t)} + \lambda I)^{-1} \nabla_{w_l} \mathcal{L}^{(t)} \tag{3.1}$$

with $w_l \in \mathbb{R}^{N_l}$ being the set of parameters of a given layer l, N_l being the number of parameters of the lth layer, $\nu \in \mathbb{R}$ being the learning rate, $\hat{F}_l \in \mathbb{R}^{N_l \times N_l}$ being the empirical FIM, $\lambda \in \mathbb{R}_+$ being a dampening value, $I \in \mathbb{R}^{N_l \times N_l}$ being the identity matrix, and $\nabla_{w_l}\mathcal{L} \in \mathbb{R}^{N_l}$ being the gradient of a scalar loss function.

This is done in an MP way, where each GPU updates only a certain number of layers. We discuss the distribution of the layers over the available GPUs in section 5.1.

At each layer l, we compute the inverse layer-wise FIM

$$(\hat{F}_l + \lambda I)^{-1} \approx (G_l + \frac{1}{\pi_l}\sqrt{\lambda} I)^{-1} \otimes (A_{l-1} + \pi_l \sqrt{\lambda} I)^{-1} \tag{3.2}$$

with G_l, A_{l-1} being the so-called Kronecker factors, λ a dampening factor, and π_l^2 being the average eigenvalue of A_{l-1} divided by the average eigenvalue of G_l- $\pi_l > 0$ because both G_l and A_{l-1} are PSD.

Preconditioning is the step where we compute the matrix-vector product of the gradient $\nabla_{w_l}\mathcal{L}$ with the preconditioning matrix $(\hat{F}_l + \lambda I)^{-1}$.

For K-FAC, we have to compute the inverse of both the left-hand-side (LHS) and the right-hand-side (RHS) of the Kronecker product $\otimes$ using `torch.cholesky_inverse`.

To avoid computing the Kronecker product before preconditioning, we used the identity

$$\mathrm{vec}(BXA^T) = (A \otimes B)\,\mathrm{vec}(X), \tag{3.3}$$

with

$A = (G_l + \frac{1}{\pi_l}\sqrt{\lambda} I)^{-1}$,

$B = (A_{l-1} + \pi_l \sqrt{\lambda} I)^{-1}$,

$X = \nabla_{w_l}\mathcal{L}$ (non-vectorized).

In the next sections, we show how we calculated the computational cost for computing the Kronecker factors and the inverse FIM for the three different kinds of NN layer types: Fully-connected (FC), convolutional (Conv) layers, and unit-wise layers such as batch normalization (unitBN).

3.1.1 K-FAC for Fully-Connected Layers

For a fully-connected (FC) layer of a NN, the output s_l is defined as

$$s_l \leftarrow W_l a_{l-1}, \tag{3.4}$$

where $a_{l-1} \in \mathbb{R}^{d_{l-1}}$ is the input to this layer (the activation of the previous layer), and $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ is the weight matrix. For simplicity, we ignored the bias. The Kronecker factors for FC layers are defined as follows

$$\begin{aligned} G_l &:= \mathbb{E}[\nabla_{s_l} \log p_\theta(y|x) \nabla_{s_l} \log p_\theta(y|x)^T], \\ A_{l-1} &:= \mathbb{E}[a_{l-1} a_{l-1}^T], \end{aligned} \tag{3.5}$$

with $G_l \in \mathbb{R}^{d_l \times d_l}$, $A_{l-1} \in \mathbb{R}^{d_{l-1} \times d_{l-1}}$.

Computing G_l and A_{l-1} of each mini-batch of size M costs $\mathcal{O}(Md_l^2)$ and $\mathcal{O}(Md_{l-1}^2)$, respectively. We assume a computing cost of $\nabla_{s_l} \log p_\theta(y|x)$ being equal to $\mathcal{O}(d_l)$.

Each layer has to compute the eigenvalues of both matrix G_l and A_{l-1} to calculate π_l which costs: $\mathcal{O}(d_l^3 + d_{l-1}^3 + d_l + d_{l-1})$. The linear terms come from averaging the eigenvalues of the matrices G_l and A_{l-1}.

Computing the inverse of both the LHS and the RHS of the Kronecker product costs: $\mathcal{O}(d_l^3 + d_l)$ and $\mathcal{O}(d_{l-1}^3 + d_{l-1})$, respectively. The linear term comes from adding the identity matrices.

Preconditioning the gradients will be done with the use of eq. (3.3) with

$A = (G_l + \frac{1}{\pi_l}\sqrt{\lambda} I)^{-1} \in \mathbb{R}^{d_l \times d_l}$,

$B = (A_{l-1} + \pi_l \sqrt{\lambda} I)^{-1} \in \mathbb{R}^{d_{l-1} \times d_{l-1}}$,

$X = \nabla_{w_l} \mathcal{L}$ (non-vectorized) $\in \mathbb{R}^{d_{l-1} \times d_l}$.

Hence, computing the preconditioning only costs $\mathcal{O}(d_{l-1}^2 d_l + d_{l-1} d_l^2)$.

Updating the parameters of layer l costs $\mathcal{O}(N_l) = \mathcal{O}(d_{l-1} d_l)$, if we consider $\nabla_{w_l} \mathcal{L} = \mathcal{O}(d_{l-1} d_l)$.

Normally, computing the Kronecker product directly and multiplying it with the gradient would have a cost of $\mathcal{O}(d_l^2 d_{l-1}^2)$, which would have been much more expensive than using the trick with eq. (3.3).

In total, computing the fully-connected layer-wise inverse FIM costs:

$$\mathcal{O}(\underbrace{d_l^3 + d_l + d_{l-1}^3 + d_{l-1}}_{\text{comp. of LHS \& RHS of } \otimes} + \underbrace{d_l^3 + d_{l-1}^3 + d_l + d_{l-1}}_{\text{comp. } \pi_l} + \underbrace{M(d_l^2 + d_{l-1}^2)}_{\text{comp. } G_l \text{ \& } A_{l-1}})$$

$$= \mathcal{O}(d_l^3 + d_{l-1}^3 + M(d_l^2 + d_{l-1}^2) + d_l + d_{l-1}),$$

with updating the parameters:

$$\mathcal{O}(d_l^3 + d_{l-1}^3 + M(d_l^2 + d_{l-1}^2) + d_l + d_{l-1} + \underbrace{d_{l-1}^2 d_l + d_{l-1} d_l^2}_{\text{preconditioning}} + \underbrace{d_{l-1} d_l}_{\text{upd. parameters}}).$$

Normally we don't have to update the FIM for each layer after every training step. It is only needed for approximately every 100th step or so. Hence we can store the LHS and RHS of $\otimes$ and use them with the next gradient update. This will reduce the computational cost of such updates to

$$\mathcal{O}(\underbrace{d_{l-1}^2 d_l + d_{l-1} d_l^2}_{\text{preconditioning}} + \underbrace{d_{l-1} d_l}_{\text{upd. parameters}}).$$

3.1.2 K-FAC for Convolutional Layers

Convolutional layers are a type of layer used in convolutional neural networks that apply small filters to the input data. The filters slide over the input tensor and compute dot products with local patches of the data. The output of the convolution operation is a set of feature maps that capture the local features and the spatial relationships between them.

For convolutional (Conv) layers, the Kronecker factors are defined as

$$G_l := \mathbb{E}[\nabla_{M_{Sl}} \log p_\theta(y|x) \nabla_{M_{Sl}} \log p_\theta(y|x)^T], \tag{3.6}$$

$$A_{l-1} := \frac{1}{h_l w_l} \mathbb{E}[M_{A_{l-1}} M_{A_{l-1}}^T], \tag{3.7}$$

and $G_l \in \mathbb{R}^{c_l \times c_l}$, $A_{l-1} \in \mathbb{R}^{c_{l-1}k_l^2 \times c_{l-1}k_l^2}$. c_l, c_{l-1}, and k_l are the number of output channels, input channels, and kernel size (assuming square kernels for simplicity), respectively. This means G_l and A_{l-1} are still square matrices and will be the same operations of computing the LHS and RHS of the $\otimes$ like for FC layers. This gives us the computational cost for Conv layer layer-wise inverse FIM and updating the parameters:

$$\mathcal{O}(c_l^3 + c_{l-1}^3 k_l^6 + M(c_l^2 + c_{l-1}^2 k_l^4) + c_l + c_{l-1}k_l^2 + \underbrace{c_{l-1}^2 k_l^4 c_l + c_{l-1} k_l^2 c_l^2}_{\text{preconditioning}} + \underbrace{c_{l-1} k_l^2 c_l}_{\text{upd. parameters}}).$$

3.1.3 K-FAC Unit-Wise Natural Gradient

For unit-wise layers such as batch normalisation (unitBN), we don't have to calculate the Kronecker factor as for the other type of layers. We directly compute the FIM of those layers.

$$\begin{aligned} F_l &\approx \hat{F}_l && (3.8)\\ &= F_{l,unitBN} && (3.9)\\ &:= \operatorname{diag}(F_l^{(1)} \dots F_l^{(i)} \dots F_l^{(c_{l-1})}) \in \mathbb{R}^{2c_{l-1} \times 2c_{l-1}}, && (3.10) \end{aligned}$$

where

$$F_l^{(i)} = \mathbb{E}\begin{bmatrix} \nabla_{\gamma_l}^{(i)^2} & \nabla_{\gamma_l}^{(i)} \nabla_{\beta_l}^{(i)} \\ \nabla_{\beta_l}^{(i)} \nabla_{\gamma_l}^{(i)} & \nabla_{\gamma_l}^{(i)^2} \end{bmatrix} \in \mathbb{R}^{2\times 2}. \tag{3.11}$$

$\nabla_{\gamma_l}^{(i)}$, $\nabla_{\beta_l}^{(i)}$ are the ith element of $\nabla_{\gamma_l} \log p_\theta(y|x)$, $\nabla_{\beta_l} \log p_\theta(y|x)$, respectively. $F_{l,unitBN}$ being a c_{l-1} times 2×2 block diagonal matrix. Computing the inverse $(F_{l,unitBN} + \lambda I)^{-1}$ can be done with little to no effort using the inverse matrix formula

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{3.12}$$

Hence the computational cost of $F_{l,unitBN}$ is $6c_{l-1}$.

3.2 Communication Cost of Distributed K-FAC

In this section, we describe the communication cost of the SP-NGD algorithm with the given Hardware properties of Piz Daint of the Swiss National Supercomputing Center (CSCS). First, we go through the amount of data we have to distribute after the data-parallel (DP) computation. Secondly, we go through the calculation of the communication cost after the model parallelism (MP). A great schematic overview of the different types of communication happing in distr. K-FAC is shown in fig. 3.1.

3.2.1 Communication Cost after the DP Part

For distr. K-FAC, we need to communicate between different GPUs in order to distribute and collect parameters, gradients and FIMs. The first communication is done after the forward and backward pass. We need to communicate A_{l-1}, G_l, $\nabla_{w_l}\mathcal{L}$ and $F_{l,unitBN}$ for all layers $l \in \{1, \dots, L\}$ using `reduce_scatter()`. Table 3.1 shows the number of parameters for the different layers that we have to communicate. A_{l-1}, G_l and $F_{l,unitBN}$ are all symmetric matrices. We could exploit this property to reduce the amount of communication without loss of information, but it has been shown to have

Table 3.1: Number of parameters to distribute for different kinds of layers for distributed K-FAC after the forward and backward pass.

type of layer	non-symmetry-aware	symmetry-aware
fully-connected	$d_{l-1}^2 + d_l^2 + d_l d_{l-1}$	$\frac{d_{l-1}(d_{l-1}+1)}{2} + \frac{d_l(d_l+1)}{2} + d_l d_{l-1}$
conv layer	$c_{l-1}^2 k_l^4 + c_l^2 + c_l c_{l-1} k_l^2$	$\frac{c_{l-1}k_l^2(c_{l-1}k_l^2+1)}{2} + \frac{c_l(c_l+1)}{2} + c_l c_{l-1} k_l^2$
unitBN	$4c_{l-1}$	$3c_{l-1}$

varying effectiveness depending on the type of DNNs [29]. If symmetry-aware communication was active, there would be a bigger overhead due to packing and unpacking those matrices. One parameter can either be a 16-bit or 32-bit float. For simplicity, we don't consider mixed precision in our calculations. In the case of a network with L fully-connected layers with non-symmetry-aware communication, we would have to communicate $L(d_{l-1}^2 + d_l^2 + d_l d_{l-1})$ parameters. Table 3.2 consists of different known NN types and the costs to communicate their parameters for both symmetry- and non-symmetry-aware communication of distr. K-FAC. Some networks are smaller and lead to smaller communication costs than others. One can use Visual Geometry Group (VGG's) [35] with or without Batch normalization (BN). Without BN, there is no communication cost for $F_{l,unitBN}$, which is noticeable in table 3.2. WideResNet's [44] has a very big communication cost for its very big Kronecker factors due to its wide layers.

3.2.2 Communication Cost after the MD Part

The second communication is done after the parameters of each layer have been updated using the `all_gather()` function. Here we have to communicate all of the preconditioned gradients $\nabla_{w_l}\mathcal{L}$ over all GPUs before the wanted optimizer updates the parameters of the model and starts over again with the DP mini-batch forward and backward pass of the next training step. This means we have a communication cost of $d_l d_{l-1}$ for FC layers, $c_l c_{l-1} k_l^2$ for Conv layers and $2c_{l-1}$ for BN. Regarding from a perspective on the communication cost for different types of DNNs, these are displayed in the column $\nabla_{w_l}\mathcal{L}$ of table 3.2.

3.3 Computational and Communication Overlap of Distributed K-FAC

For distr. K-FAC, computation and communication can, at some times, overlap. Let's consider having a single step of training, where we update the preconditioning matrix and use the SP-NGD method from *Osawa et al.* [28]. In this case, the first-ever communication already happens after the forward pass in the DP part. There, we compute the Kronecker factor A_{l-1} of

DNN type	A_{l-1}	G_l	$F_{l,unitBN}$	$\nabla_{w_l}\mathcal{L}$	$\sum$	$\sum$ in [MB]
ResNet152	320.5M	69.0M	0.6M	60.1M	450.2M	1800.9
	160.4M	34.5M	0.5M	60.1M	255.5M	1021.8
ResNet101	230.6M	52.5M	0.4M	44.5M	328.0M	1312.0
	115.4M	26.3M	0.3M	44.5M	186.4M	745.8
ResNet50	121.4M	32.4M	0.2M	25.5M	179.6M	718.4
	60.7M	16.2M	0.2M	25.5M	102.7M	410.6
ResNet18	93.0M	2.0M	38400	11.7M	106.7M	426.9
	46.5M	1.0M	28800	11.7M	59.2M	236.9
WideResNet101	814.0M	66.7M	0.6M	126.8M	1008.1M	4032.4
	407.1M	33.4M	0.4M	126.8M	567.7M	2270.9
WideResNet50	430.8M	39.9M	0.3M	68.8M	539.9M	2159.4
	215.4M	20.0M	0.2M	68.8M	304.5M	1217.9
EfficientNet-b4	77.1M	119.9M	0.5M	19.2M	216.8M	867.1
	38.6M	60.0M	0.4M	19.2M	118.2M	472.8
EfficientNet-b0	16.4M	26.2M	0.2M	5.3M	48.0M	191.9
	8.2M	13.1M	0.1M	5.3M	26.7M	106.7
VGG19-BN	207.7M	665.4M	44032	143.7M	1016.8M	4067.2
	103.9M	332.7M	33024	143.7M	580.3M	2321.2
VGG19	207.7M	665.4M	0	143.7M	1016.8M	4067.0
	103.9M	332.7M	0	143.7M	580.3M	2321.0
VGG16	160.0M	664.8M	0	138.3M	963.1M	3852.3
	80.0M	332.4M	0	138.3M	550.8M	2203.0
VGG13	112.2M	664.2M	0	133.0M	909.4M	3637.6
	56.1M	332.1M	0	133.0M	521.3M	2085.0
VGG11	110.5M	664.2M	0	132.9M	907.5M	3630.2
	55.3M	332.1M	0	132.9M	520.2M	2080.9
MLP-Mixer	124.1M	123.7M	0.2M	59.8M	307.8M	1231.4
	62.1M	61.9M	0.1M	59.8M	183.9M	735.7
ViT-Huge	681.1M	480.2M	0.4M	303.9M	1465.6M	5862.3
	340.6M	240.2M	0.3M	303.9M	885.0M	3540.0
ViT-Base	192.7M	135.7M	0.2M	86.3M	414.8M	1659.3
	96.4M	67.9M	0.1M	86.3M	250.7M	1002.7
ViT-Tiny	13.5M	8.5M	38400	5.7M	27.7M	110.8
	6.8M	4.2M	28800	5.7M	16.7M	66.8

Table 3.2: Number of parameters to distribute after the forward and backward pass for K-FAC for different kinds of NN architectures. The last column considers parameters of 32-bit precision. Each DNN type has two rows. The first row shows the number of parameters for non-symmetry-aware communication, and the second row for symmetry-aware communication. A_{l-1} and G_l are the sums for all the Kronecker factors of each layer of layer type: `Linear` and `Conv2d`. $F_{l,unitBN}$ is unit-wise Fisher for the layers: `BatchNorm1d`, `BatchNorm2d` and `LayerNorm`. ResNet [17], WideResNet [44], EfficientNet [25], VGG [35], MLP-Mixer [40], and ViT [9] are all examples of different well known NNs.

each Conv and FC layer for each mini-batch on each GPU. While distributing all $A_{0:L-1}$ using `reduce_scatter()`, the backward pass can already happen. Unfortunately, this is the only overlap which we could do if we update the preconditioner at the same training step as it is being computed. All other communications have to be done in a blocking manner. If the update of the preconditioning matrix is not part of the same training step (maybe the updated preconditioning matrix will be used for the next timestep), we could overlap the communication for G_l, $\nabla_{w_l}\mathcal{L}$ and $F_{l,unitBN}$ as well as for `reduce_scatter()`. In the end, we did not implement the communication in an overlapping fashion with the computational parts. The Kronecker factors and the FIM are all communicated to all the desired GPU ranks via a `reduce_scatter()`, and the `all_gather()` is done after the preconditioning step. There is no big benefit in doing the communication of the Kronecker factors right away after the forward (i.e. backward) pass as shown in the results chapter 5. This would also have been out of the scope of this thesis.

Chapter 4

Distributed Shampoo

This chapter explains our distributed Shampoo algorithm, how it works, and its computational and communication cost calculation. Shampoo from *Gupta et al.* [15] is an optimization algorithm that improves the convergence and stability of first-order optimization using preconditioning matrices. Shampoo's curvature-based preconditioner takes into account the second-order information of the loss function, similar to the idea behind the popular quasi-Newton methods [6]. However, unlike quasi-Newton methods, which store and update a dense approximation of the Hessian matrix, Shampoo approximates the curvature matrix using a low-rank factorization (i.e. the Kronecker factors of a given layer), reducing computational and memory requirements. Shampoo is considered to be an adaptive gradient-based method and is closely related to AdaGrad [10]. As stated by *Gupta et al.* [15]: Shampoo can be viewed as an efficient, practical and provable optimizer using the full AdaGrad preconditioner without falling back to diagonal matrices. Shampoo is also similar to K-FAC [23] due to also having (Kronecker) factored preconditioning matrices. Overall, Shampoo has been shown to outperform standard first-order methods in a variety of deep learning tasks, including image classification, language modelling, and reinforcement learning [5, 15, 27]. Let's discuss the three different kinds of Shampoo algorithms:

1. Shampoo for matrix case,
2. Shampoo for general tensor case,
3. Diagonal version of Shampoo for matrix case.

The first and third algorithm only works with 2D layers, such as FC (i.e. Linear) layers. The second algorithm is meant for any k-dimensional tensors ($k \geq 2$), as well as for FC ones. We only implemented the distributed preconditioning method of Shampoo in ASDL for the second algorithm due to its generalized form for any tensor shape ($k \geq 2$). Any one-dimensional layers will not be preconditioned. But they will still be updated with any wanted optimizer (e.g. SGD). For very big layers which don't fit into the memory of one GPU, we have two options. The first option is to use the given `BlockPartitioner` method to further reduce the layer into smaller blocks. The second option is to ignore layers of certain sizes. For example, embedding layers like GPT-2, which can have vocabulary sizes of 50'000 or more, we just ignore the preconditioning step on those types of layers. In the following section, we calculate the computational cost of all three different kinds of Shampoo algorithms to demonstrate the cost-effectiveness of each one. In the following section, whenever we mention the different algorithms and their computational and communicational cost, the notations are adopted from *Gupta et al.* [15].

4.1 Computational Cost of Shampoo

In this section, we explain the computational cost of the three different available Shampoo algorithms as well as the computational cost of computing the p-th root and inverse of a positive semi-definite (PSD) matrix. In the original paper, algorithms 1 and 2 used the Singular Value Decomposition (SVD) to compute the p-th root and inverse of the PSD matrix [15]. *Anil et al.* [5] discovered a faster method to compute the p-th root and inverse using the so-called coupled Newton iteration, which is based on the Schur–Newton method from *Guo et al.* [14].

4.1.1 Shampoo Algorithm: Matrix Case

Algorithm 1 Shampoo Algorithm for the matrix case for one layer

```
Initialize: $W_1 = \mathbb{R}^{m\times n}$ ; $L_0 = \varepsilon I_m$ ; $R_0 = \varepsilon I_n$
for $t = 1, \dots, T$ do
    Receive loss function $f_t : \mathbb{R}^{m\times n} \to \mathbb{R}$
    Compute gradient $G_t = \nabla f_t(W_t)$ $\{G_t \in \mathbb{R}^{m\times n}\}$
    Update preconditioners:
        $L_t = L_{t-1} + G_t G_t^T$
        $R_t = R_{t-1} + G_t^T G_t$
    Precondition the gradient:
        $\tilde{G}_t = L_t^{-1/4} G_t R_t^{-1/4}$
    Update parameters:
        $W_{t+1} = W_t - \eta \tilde{G}_t$
end for
```

In this section, we discuss the Shampoo algorithm for the layers that must be updated, those layers with parameters in the shape of a matrix (i.e. $k = 2$), and those which include linear or embedding layers. Algorithm 1 shows the pseudo-code of the Shampoo algorithm for matrix-shaped layers. $W_t\mathbb{R}^{m\times n}$ is the weight matrix of a given layer at the training step t that we want to precondition. At time step $t = 1$, W_1 should be initialized according to the user's desire (e.g. RNG, Zeros, or save State). L_0 and R_0 are the two preconditioning matrices of a given layer. They are initialised by a diagonal matrix filled with ε for some stabilizing. The difference between the Shampoo algorithm and any first-order method lies in the "Update preconditioners" and the "Precondition the gradient" steps. In the "Update parameter" step, the preconditioned gradients $\tilde{G}_t$ are used to update the parameters of the given layer. In the following pseudo-code example, we demonstrate the update of a simple SGD method. One could use a momentum-based method to update the gradients like the current/improved version of Shampoo from *Anil et al.* [5] shown in algorithm 5. Now, let's get into the computational cost of this algorithm.

Updating the preconditioners left $L_t \in \mathbb{R}^{m\times m}$ and right $R_t \in \mathbb{R}^{n\times n}$ preconditioners costs $\mathcal{O}(m^2n + m^2)$ and $\mathcal{O}(n^2m + n^2)$, respectively. Computing the $-1/4$ power of both the preconditioners costs $\mathcal{O}(m^3 + n^3)$. Updating the parameters costs $\mathcal{O}(m^2n + mn^2 + mn)$, if we consider computing the gradient as $G_t = \mathcal{O}(mn)$.

Hence, in total, we have the following computational cost:

$$\begin{aligned}\mathcal{O}(\underbrace{m^2n + n^2m + m^2 + n^2}_{\text{upd. } L_t \text{ \& } R_t} + \underbrace{m^3 + n^3}_{\text{comp. } L^{-1/4} \text{ \& } R^{-1/4}} + \underbrace{m^2n + mn^2 + mn}_{\text{upd. parameters}}) \\ = \mathcal{O}(m^3 + n^3 + m^2n + n^2m + m^2 + n^2 + mn).\end{aligned} \tag{4.1}$$

If we don't consider updating L_t and R_t at every training step, we only have the computational cost of updating the parameters.

4.1.2 Shampoo Algorithm: General Tensor Case

In this section, we are going to show the actual Shampoo algorithm meant for any layers with parameters of shape $k \geq 2$ which also includes the matrix shape layers. Algorithm 2 shows the pseudo-code for the Shampoo algorithm for any tensor-shaped gradients. One can see new math notations like $G_t^{(i)}$ and $\times_i$. Those are newly defined tensor operations which are well described in the original Shampoo paper of *Gupta et al.* [15]. We adopted the notations from *Gupta et al.* and use them when we state the computational cost of this algorithm.

For the Shampoo algorithm 2, the $G_t \in \mathbb{R}^{n_1\times\cdots\times n_k}$ is the gradient at training step t as a tensor of order k. n_i is the size of i-th dimension with $i \in \{1, \ldots, k\}$. We also denote $\mathbf{n} = \prod_{i=1}^{k} n_i$ and $n_{-i} = \prod_{j\neq i} n_j$ where n_{-i} is the product of

Algorithm 2 Shampoo Algorithm for a general tensor case for one layer

```
Initialize: $W_1 = \mathbb{R}^{n_1 \times \cdots \times n_k}$ ; $\forall i \in [k] : H_0^i = \varepsilon I_{n_i}$
for $t = 1, \dots, T$ do
    Receive loss function $f_t : \mathbb{R}^{n_1 \times \cdots \times n_k} \to \mathbb{R}$
    Compute gradient $G_t = \nabla f_t(W_t)$ $\{G_t \in \mathbb{R}^{n_1 \times \cdots \times n_k}\}$
    $\tilde{G}_t \leftarrow G_t$ $\{\tilde{G}_t$ is the preconditioned gradient$\}$
    for $i = 1, \dots, k$ do
        $H_t^i = H_{t-1}^i + G_t^{(i)}$
        $\tilde{G}_t \leftarrow \tilde{G}_t \times_i (H_t^i)^{-\frac{1}{2k}}$
    end for
    Update: $W_{t+1} = W_t - \eta \tilde{G}_t$
end for
```

all of the other dimensions except the i-th dimension. Note that $\mathbf{n} = n_i n_{-i}$. For each dimension k, we have to compute the *contraction* of the gradient $G_t^{(i)}$ which costs $\mathcal{O}(n_i^2 n_{-i} + n_i^2)$. The *contraction* can be seen as a slice of the ith dimension of the gradient multiplied with itself (for example $G_t G_t^T$ and $G_t^T G_t$ from algorithm 1). After that, we compute $2k$-th root and inverse of preconditioning matrix $(H_t^i)^{-1/2k}$. Computing this will cost

$$\begin{aligned}\mathcal{O}(\texttt{max\_iter}((\lceil \log_2(2k) \rceil + 2) n_i^3 + n_i^2 + n_i) + \texttt{max\_iter}(n_i^2 + n_i)) \\ = \mathcal{O}(n_i^3).\end{aligned}$$

We calculated the computational cost of the matrix p-root and its inverse in section 4.1.4. The next step is to compute the `tensordot()` of the preconditioned gradient $\tilde{G}_t$ with preconditioning matrix $(H_t^i)^{-1/2k}$ which is marked in algorithm 2 with $\times_i$. One can think of the `tensordot()` operation for any dimensional tensor as a matrix-matrix multiplication in a certain dimension in the gradient $\tilde{G}_t$ with the matrix $(H_t^i)^{-1/2k}$. This operation costs $\mathcal{O}(n_i^2 n_{-i})$.

Hence, for each dimension k, we compute:

$$\mathcal{O}(\underbrace{n_i^2 n_{-i}}_{\text{upd. } H_t^i} + \underbrace{n_i^3}_{\text{comp. } (H_t^i)^{-1/2k}} + \underbrace{n_i^2 n_{-i}}_{\text{upd. } \tilde{G}_t}).$$

When including updating the parameters of the layer for each training step, we have the following computational cost for algorithm 2:

$$\mathcal{O}(\sum_{i=1}^{k} (n_i^2 n_{-i} + n_i^3 + n_i^2 n_{-i}) + \underbrace{\prod_{i=1}^{k} (n_i)}_{\text{upd. parameters}}). \tag{4.2}$$

Note that for $k = 2$, the operational intensity is the same as for eq. (4.1) from section 4.1.1 which is the operational intensity for two-dimensional (matrix) layers.

Usually, we don't update the curvature and preconditioner at every training step. The preconditioning matrices $(H_t^i)^{-1/2k}$ are usually staled for $\sim$100 steps. This gives us a new computational cost of

$$\mathcal{O}(\underbrace{\sum_{i=1}^{k}(n_i^2 n_{-i})}_{\text{preconditioning}} + \underbrace{\prod_{i=1}^{k}(n_i)}_{\text{upd. parameters}})$$

$$= \mathcal{O}(\underbrace{\mathbf{n}\sum_{i=1}^{k} n_i}_{\text{preconditioning}} + \underbrace{\mathbf{n}}_{\text{upd. parameters}}).$$

We still use the staled preconditioners to precondition the gradients at every training step.

4.1.3 Shampoo Algorithm: Diagonal Version of Shampoo for Matrix Case

Algorithm 3 Shampoo Algorithm for the matrix case but diagonal version

```
Initialize: $W_1 = \mathbb{R}^{m\times n}$ ; $L_0 = \varepsilon I_m$ ; $R_0 = \varepsilon I_n$
for $t = 1, \dots, T$ do
  Receive loss function $f_t : \mathbb{R}^{m\times n} \to \mathbb{R}$
  Compute gradient $G_t = \nabla f_t(W_t)$ $\{G_t \in \mathbb{R}^{m\times n}\}$
  Update preconditioners:
      $L_t = L_{t-1} + diag(G_t G_t^T)$
      $R_t = R_{t-1} + diag(G_t^T G_t)$
  Precondition the gradient:
      $\tilde{G}_t = L_t^{-1/4} G_t R_t^{-1/4}$
  Update parameters:
      $W_{t+1} = W_t - \eta \tilde{G}_t$
end for
```

The algorithm 3 is built to be used for very large dimensional layers when computing the p-th root of such matrices is too expensive or when the matrices are too big to store in memory. We never implemented and tested this algorithm. We only demonstrate the computational cost of this type of algorithm because it is also another type of Shampoo algorithm. If we had to deal with very big layers, another good method is to use the `Blockpartitioner()` method of the improved Shampoo algorithm from *Anil et al.* . This block partitioning further reduces the shapes of the preconditioning matrices into smaller ones, but it will also make more of them (see fig. 6.4). One could also just ignore preconditioning certain layers and only add a momentum-based optimizer to reduce memory usage. Algorithm 3 is the same as algorithm 1, the only difference is the $diag()$ functions in the "Update Preconditioners" step. In this case, L_t and R_t are only diagonal matrices. Computing the $-1/4$ power is done by taking the power over the element-wise diagonal entries. This computation is of order $\mathcal{O}(m)$ and $\mathcal{O}(n)$. Computing $\text{diag}(G_t G_t^T)$ costs $\mathcal{O}(nm)$ and $\text{diag}(G_t^T G_t) = \mathcal{O}(mn)$, respectively. Computing the matrix-matrix multiplication with two diagonal matrices $L_t^{-1/4} G_t R_t^{-1/4}$ costs $2mn$ multiplications, hence $\mathcal{O}(mn)$. This simplifies the computational cost to:

$$\begin{aligned}\mathcal{O}(&\underbrace{nm + mn + m + n}_{\text{upd. } L_t \text{ \& } R_t} + \underbrace{m + n}_{\text{comp. } L^{-1/4} \text{ \& } R^{-1/4}} + \underbrace{2mn + mn}_{\text{upd. parameters}}) \\ &= \mathcal{O}(4mn + 2m + 2n) = \mathcal{O}(mn + m + n).\end{aligned} \tag{4.3}$$

4.1.4 Matrix pth Root and its Inverse Algorithm

Anil et al. [5] uses a Schur-Newton iteration method to compute the p-th root and its inverse of a matrix made by *Guo et al.* [14], which is a follow-up to the Shampoo algorithm invented by *Gupta et al.* [15]. The p-root and inverse computation is the most time-consuming part of Shampoo (for results see fig. 6.11). *Anil et al.* improved the Shampoo algorithm by introducing it to *Guo et al.* p-root and inverse algorithm. Let's discuss this p-root and inverse algorithm of *Guo et al.* and have a look at it because it is a major part of the improved Shampoo algorithm from *Anil et al.* [5].

Assuming we have a preconditioning matrix $H \in \mathbb{R}^{n \times n}$, which is positive semi-definite (PSD). The goal is to compute $H^{-1/p}$. *Anil et al.* described the algorithm very well in their paper, but in short, the Newton method iteratively tries to solve $X^{-p} - H = 0$ for some matrix $X \in \mathbb{R}^{n \times n}$. It is satisfied when $X_{\texttt{iter}} \to H^{-1/p}$ as `iter` $\to \infty$. We also wrote down the pseudo algorithm in algorithm 4. This p-root inverse algorithm can be described in 3 steps.

The first step consists of computing the maximum eigenvalue of H_t^i using the $\lambda_{max} \leftarrow$ `PowerIter(`H_t^i`)` function. This function does at most $\mathcal{O}(\texttt{max\_iter}(n^2 + n))$ computations, where the algorithm runs either until convergence or until the maximum `max_iter` is reached. `max_iter` is normally kept at 100.

In the second step, X and M are computed with the calculated max eigenvalue from the first step. This costs $\mathcal{O}(n^2 + n)$ due to computing the Frobenius norm and adding a diagonal matrix to H.

Algorithm 4 pth root and its inverse algorithm from shampoo optimizer code of *Anil et al.* [5]. The `PowerIter` function is in the appendix algorithm 6.

```
Require: $H \leftarrow H_t^i$; $p = 2k$; $\varepsilon$; $\texttt{max\_iter} = 100$; $\texttt{tolerance} = 1e-12$
  $\alpha = -1/p$
  $\lambda_{max} = \texttt{PowerIter}(H, \texttt{max\_iter}, \texttt{tolerance})$
  $\varepsilon = \lambda_{max}\varepsilon$
  $H = H + \varepsilon I$
  $z = \frac{1+p}{2\|H\|_F}$
  $X = z^{1.0/p} I$
  $M = zH$
  $\texttt{error} = \max(|M - I|)$
  $\texttt{iter} = 0$
  while $\texttt{error} > \texttt{tolerance}$ and $\texttt{iter} < \texttt{max\_iter}$ do
    $\hat{M} = (1-\alpha)I + \alpha M$
    $\hat{X} = X\hat{M}$
    $M = \hat{M}^p M$
    $\texttt{error}_{new} = \max(|M - I|)$
    if $\texttt{error}_{new} > 1.2\texttt{error}$ then
      break
    end if
    $X = \hat{X}$
    $\texttt{error} = \texttt{error}_{new}$
    $iter = iter + 1$
  end while
  return X
```

In the third step, we do Newton iteration until convergence or the maximum `max_iter` reached. For each iteration, we have to compute p-th power of some matrix plus two other matrix-matrix multiplications, which gives us a computational cost of $\texttt{max\_iter}((\lceil \log_2(p) \rceil + 2)n^3 + 2n^2 + n)$ operations.

Hence, in total we have a computational cost of

$$\mathcal{O}(\texttt{max\_iter}((\lceil \log_2(p) \rceil + 2)n^3 + n^2 + n) + \texttt{max\_iter}(n^2 + n) + n^2 + n). \quad (4.4)$$

4.2 Communicational Cost of Shampoo

When implementing the DP and MP part for distr. Shampoo, we use the same parallelization approach as for distr. K-FAC, which is described in section 3.2. But in this case, we have to communicate less data than compared to K-FAC. Instead of `reduce_scatter()` for A_{l-1}, G_l, $\nabla_{w_l}\mathcal{L}$ and $F_{l,unitBN}$ after the forward and backward pass done in a DP fashion, we only have to `reduce_scatter()` $\nabla_{w_l}\mathcal{L}$ for each layer. Each GPU computes the Shampoo algorithm 2 for one layer l.

The MP part is the same as discussed in section 3.2.2. We only have to communicate the gradients $\nabla_{w_l}\mathcal{L}$ of each layer l that have been preconditioned back to all the available GPUs via `all_gather()`.

4.3 Computational and Communicational Overlap in Shampoo

Compared to distr. K-FAC, for distr. Shampoo there is less information that needs to be distributed. We only have to distribute $\nabla_{w_l}\mathcal{L}$ after the DP part using `reduce_scatter()` and `all_gather()` after updating all the weights of each layer. If the preconditioning matrix is updated at the same time as updating the weights for each layer, communication and computation cannot overlap when we want to calculate the preconditioning matrix in the same training step. If the preconditioning matrices are computed over multiple training steps and staled between each step, one can achieve overlapping computation and communication over a certain amount of time steps. This has been demonstrated by the improved Shampoo algorithm of *Anil et al.* [5]. In our case, we implemented distributed Shampoo without overlapping computation and communication but included the option of staling a certain amount of training steps. When a preconditioning step is needed, everything is done in one training step.

Chapter 5

Comparison of Distributed K-FAC and Distributed Shampoo

In this chapter, we go through our results of distr. K-FAC and distr. Shampoo which we both implemented in the library of ASDL. Further, in section 5.1 we discuss how we balance the workload over the available GPUs depending on the size of the model. Automatic Second-order Differentiation Library (ASDL) is a PyTorch library created by *Osawa et al.* [27] which lets users use the most known second-order methods (e.g. K-FAC, Shampoo, SENG [42], PSGD [20], . . .) by adding three more lines to an existing PyTorch script. The great thing about ASDL is its simplicity to use and easily switch between other second-order methods. Not every second-order (or preconditioning method in general) works in the same way. Some are better than others at optimizing different types of problems. That is why this library exists, to easily switch between the different types of second-order methods and to find the most suitable ones for a given problem. Distributed K-FAC and distributed Shampoo are both available on the ASDL library on the `dev-grad-maker` branch[1]. However, be aware that distr. K-FAC and distr. Shampoo are still prototypes.

5.1 Workload Balancing

In this section, we are going to talk about how we deal with balancing the workload of distr. K-FAC and distr. Shampoo depending on the number of available GPUs and the number of NN layers. Workload balance is a very important and difficult task which we have to solve because if the workload is very imbalanced, many GPUs would be idling, and one training step will also take longer than wanted. Idling should be minimized to improve throughput as well as reduce runtime for maximum efficiency. For both distr.

[1] https://github.com/kazukiosawa/asdl/tree/dev-grad-maker

K-FAC and distr. Shampoo, we implemented an algorithm which computes the distributed workload by the number of layers for all the available GPUs. However, we only use this approach when there are more layers than available GPUs. If we only have one GPU, no partitioning is needed and the single GPU has to compute all the preconditioning. If there is an equal amount of layers and GPUs, each GPU will be assigned one layer. In a situation where there are more GPUs than layers, every GPU gets one layer, and the remaining will be idling at the preconditioning step. The workload balancing algorithm is shown in the appendix listing A.1. The workload balancing (i.e. partitioning) algorithm only runs once at the initialization step of the preconditioning method and can be divided into two major steps. In the first step, the computational cost of each NN layer is estimated. For that, we use a cost function to estimate the cost for computing each layer. This cost function is not the same as the one we have estimated in section 3.2 for K-FAC and for Shampoo section 4.1 due to other important factors such as memory movement. But both the K-FAC and Shampoo cost function are similar and still have much in common; the more parameters a layer has, the longer it will take to precondition that layer. With a trial and error approach, we found out that the cost function $C_{K\text{-}FAC}(w_l)$ for K-FAC for one layer is well estimated with

$$C_{K\text{-}FAC}(w_l) = (\prod_{i=1}^{k} n_i)^{0.3} \tag{5.1}$$

where $w_l \in \mathbb{R}^{n_1 \times \cdots \times n_k}$ are the parameters if layer $l \in \{1, \ldots, L\}$. For Shampoo it is harder to predict the cost of computing certain layers due to having the p-root and inverse algorithm consisting of a while-loop (see algorithm 4). Depending on how ill-conditioned the preconditioning matrices are, the longer the p-root and inverse algorithm takes to compute. Still, we found the cost function for one layer $C_{Shampoo}(w_l)$ is appropriately estimated with

$$C_{Shampoo}(w_l) = \sum_{i=1}^{k} n_k^{0.4}. \tag{5.2}$$

Note that for both K-FAC and Shampoo, layers which don't need preconditioning (e.g. for K-FAC pooling layers and for Shampoo layers with $k < 2$) are effectively ignored by ASDL and hence not counted.

This raises the question, why even bother to estimate the cost of preconditioning each layer and not directly split the layers by the number of GPUs instead? This is because not every layer has the same amount of parameters or shapes and not every NN is the same type. Many NNs vary a lot in the number of parameters per layer, for example, VGG11 [35]. One can see

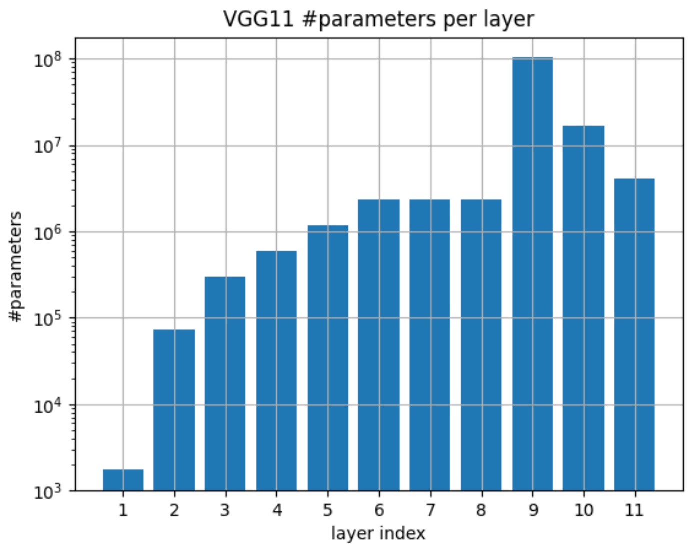


Figure 5.1: The Visual Geometry Group 11 layer (VGG11) network [35] without batch normalization represented in the number of parameters for each layer in a log-scale. The first eight layers $\{1, \ldots, 8\}$ are Conv2D layers, and the last three layers are linear layers. Layer 9 is a `nn.Linear(in=25088, out=4096, bias=True)` which has roughly $\approx 1.03e8$ parameters.

the strong difference in the number of parameters per layer of the VGG11 network (see fig. 5.1). When we compare the actual runtimes of computing the preconditioning matrices of each layer shown in fig. 5.2, one can see a similarity of the two preconditioning methods K-FAC and Shampoo. Notice the layers $\{6, 7, 8\}$ in the Shampoo plot; they all belong to the same type of layer `Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))`, still they have different runtimes. All the Conv2d layers from $\{1, \ldots, 8\}$ of the Shampoo plot differ in runtime, and some like layer 3 are even taking longer than the layers $\{6, 7, 8\}$ because the while-loop in the p-root inverse computation can finish computing earlier than for other layers. Whereas for the K-FAC fig. 5.2, one can see a much better resemblance to the number of parameters shown in fig. 5.1. Figure 5.3 shows our cost function estimate of VGG11 for K-FAC and Shampoo combined in one. For Shampoo, we don't classify certain layers as heavy compared to the cost function of K-FAC due to the uncertainty of the p-root inversion algorithm.

How we came up with those cost functions from eqs. (5.1) and (5.2)? Consider the exponent in the equations, which is $p = 0.3$ for K-FAC and $p = 0.4$ for Shampoo. The bigger the exponent is (e.g. $p \in [1, 3]$), the more it resembles the actual computational cost of K-FAC (resp. Shampoo). If it is relatively small $p \in (0, 1)$, then the cost function considers the cost of actual data movement on GPU memory. If $p = 0$, then the cost function splits the

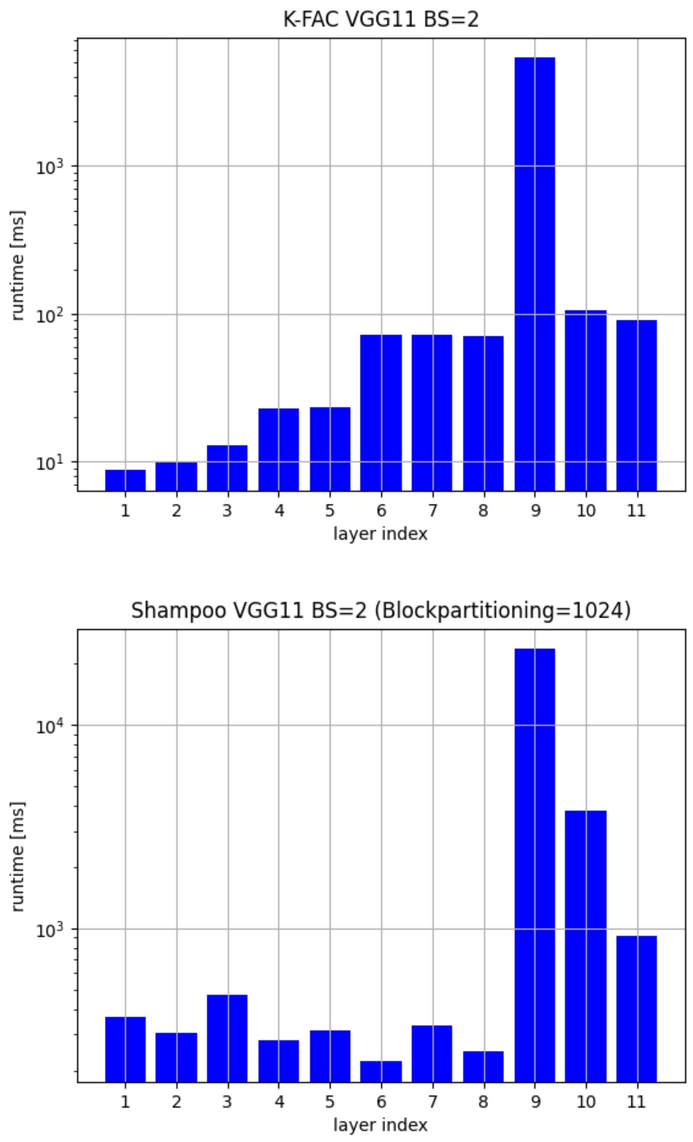


Figure 5.2: K-FAC and Shampoo algorithm on the VGG11 network runtime of computing the preconditioning matrices of each layer. Both were running on one Tesla P100 16GB GPU on Piz Daint and were averaged over three training steps with a batch size of 2. For Shampoo, we had to activate `Blockpartitioning=1024` to solve the memory capacity problems. The last three layers of VGG11 are too big to store on one P100 GPU. When `Blockpartitioning` is active, it further divides the Shampoo preconditioning shapes into smaller ones until they are at least the size 1024. One can compare the runtime with the actual size of the layers shown in fig. 5.1.

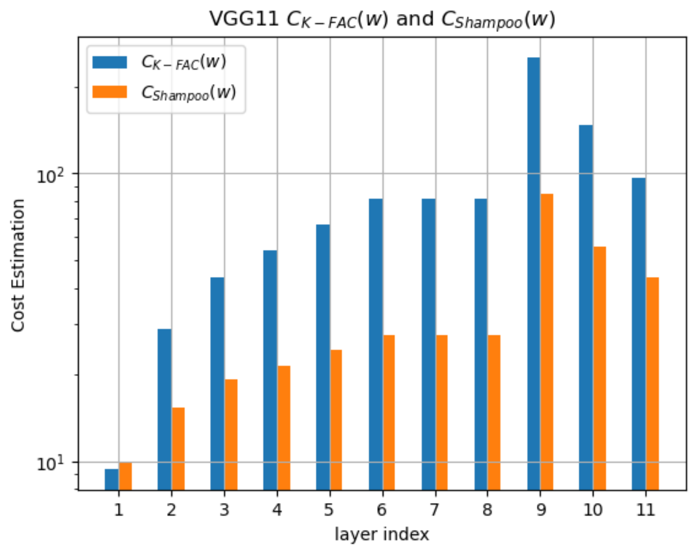


Figure 5.3: The estimated costs of preconditioning of VGG11 for each layer with the cost functions from eqs. (5.1) and (5.2).

workload by the number of preconditioning matrices. We ran tests with $p = 3$ for both distr. K-FAC and distr. Shampoo, which is the actual computational cost of computing the preconditioning matrix in $\mathcal{O}(n^3)$, but $p = 3$ leads to workload balance issues. By studying the performance of Piz Daints Tesla P100 GPUs using Nvidia Nsight, we came up with the respective exponent terms from eqs. (5.1) and (5.2). Note that VGG11 is an extreme case with a very imbalanced number of parameters per layer. For example, ResNet and DenseNet have better workload balances.

After we generated an approximated list of the cost of computing each of the layers, we now go to the second step of our workload balance algorithm, which is distributing the workload over all the available layers. We call this step partitioning, and it is a simple and reliable algorithm (see listing A.1) and works by setting the splits such that the sum of each partitioning is as evenly distributed as possible. This creates a partitioning list containing all the ranks of the GPUs in ascending order. This ascending order is important because we have to `Reduce_scatter` the gradients from the DP to the specified ranks from the partitioning and `All_Gather` them all back once the gradients have been preconditioned. Before those collective communications happen, we have to pack all the layers from the same partitioning rank into one contiguous flatten tensor. Sorting the partitioning list in an ascending order speeds up the packing of the layers into one flattened tensor.

We were not able to use `dist.reduce_scatter()` and `dist.all_gather()` from the PyTorch's distributed communication package because the tensors

	distr. K-FAC	distr. Shampoo
Main computation	Matrix Inversion	Matrix p-root and its inverse
Data to communicate	Kronecker Factors A_{l-1}, G_l, $F_{unit-wise}$ & Gradients $\nabla L(w)$	Gradients $\nabla L(w)$
Disadvantages	Higher communication cost, scales by the batch size, and extra computations in the forward and backward-pass due to the Kronecker factors.	Computing the p-root inverse is very expensive, and balancing the workload over multiple GPUs is difficult.
Advantages	K-FAC performs well on large mini-batch sizes and has been thoroughly tested in large-scale training.	Doesn't scale by batch size and is flexible with preconditioning shapes with the `BlockPartitioning` method.
Communication types	`Reduce_scatter`$(A_{l-1}, G_l, \nabla L(w))$ & `AllGather`$(\nabla L(w))$	`Reduce_scatter`$(\nabla L(w))$ & `AllGather`$(\nabla L(w))$

Table 5.1: This table is a summary of the chapters 3 and 4 and comparison of distr. K-FAC and distr. Shampoo what are the advantages and disadvantages of one and another.

have to be of the same size for those collective communications, which we can't guarantee with our partitioning algorithm. Some GPUs have more layers to precondition while there are smaller GPUs that have maybe only one big layer to precondition. We opted for multiple non-blocking `dist.reduce()` instead of `dist.reduce_scatter()`. Instead of `dist.all_gather()` we used multiple non-blocking `dist.broadcast()`. One could have used the mentioned collective communicators `dist.reduce_scatter()` and `dist.all_gather()` with padding in the flattened tensor to match the sizes of all the tensors which have to be communicated. This could improve the overall runtime of the communication parts. However, our results show that the communication parts clearly do not have such big of an impact compared to other factors, for example, computing the forward-/ and backward-pass or computing the preconditioning matrices (see section 5.2).

5.2 Distributed K-FAC vs Distributed Shampoo Results

In this section, we are going to discuss the measured results of our implemented version of distributed K-FAC and distributed Shampoo in PyTorch which ran on Piz Daint for different models, batch sizes, and number of GPUs and compare distr. K-FAC and distr. Shampoo to each other. Before we go into the results of our measurements, we created this table 5.1 to show a summary of both distr. K-FAC and distr. Shampoo measurements side-by-side are also a summary of the chapters 3 and 4.

To measure our implementations of distr. K-FAC and distr. Shampoo, we used Nvidia Nsight [7] which is a suite of tools for profiling, debugging, and optimizing CUDA applications. It provides a range of features for analyzing performance metrics, inspecting memory usage, and identifying bottlenecks in GPU-accelerated code. We used the `nvtx.range()` method from `torch.cuda` to capture the runtime of specific functions such as: forward, backward, collective communication, preconditioning, and more.

In section 5.1, we discussed our approach to achieving workload balance. Now, let's have a look at some runtime plots of different NN and the number of GPUs.

One of the simplest cases we looked at first was when we had more GPUs available than the number of layers of a NN which have to be preconditioned. Figure 5.4 is a runtime plot for the case of 4 GPUs and a 3-layer multi-layer perception consisting of linear layers with ReLU activation in between. In this case, GPU 0 has the first linear layer, GPU 1 the second linear layer and GPU 2 the last linear layer. GPU 3 does not have any layers to precondition because this NN only consists of 3 layers. Hence GPU 3 will be idling at the time when GPU 0, 1 and 2 will do the preconditioning. Still, GPU 3 has to communicate its gradients from its training batch to the other available ranks. All the linear layers are the same shape of 4×4. Still, one can see the p-root inverse and its non-deterministic number of iterations shows that for any training step, sometimes it takes fewer steps to compute the preconditioning matrices than at other steps, and it is also depending on the deepness of the layer.

Another example we looked at was for a known NN such as ResNet18 [17] how the workload balance looks like and how it scales by the number of GPUs. In fig. 5.5, we show the runtime of ResNet18 for different numbers of GPUs and ran for three training steps. We are satisfied with our distributed workload for all three cases $\#GPUs = \{2, 4, 8\}$. For ResNet18, the workload is well-balanced.

But we also looked at other models to ensure that the workload balance makes sense for other NNs for a fixed number of GPUs. In fig. 5.6, we compared the workload balanced of 4 different known NNs, which are all different from each other: ResNet18, DenseNet121 [18], VGG11_BN, and MobileNetV2 [33]. For ResNet18 and DenseNet121, the workload balance is very well. But for VGG11 and MobileNetV2 it could be better. VGG11 has the problem of having very big linear layers at the end of the model, as we have discussed as an extreme case in section 5.1. MobileNetV2 on the other hand, is pretty similar to any other convolutional neural network. It does not have a very imbalanced number of parameters per layer. In our case, the distr. Shampoo cost function exponent $p = 0.5$ is too aggressive towards the computational cost. Using a smaller exponent $p \in [0.0, 0.3]$ would fix this problem. This shows our cost function doesn't work well for every case and requires hyperparameter tuning in the exponent for specific networks. It is

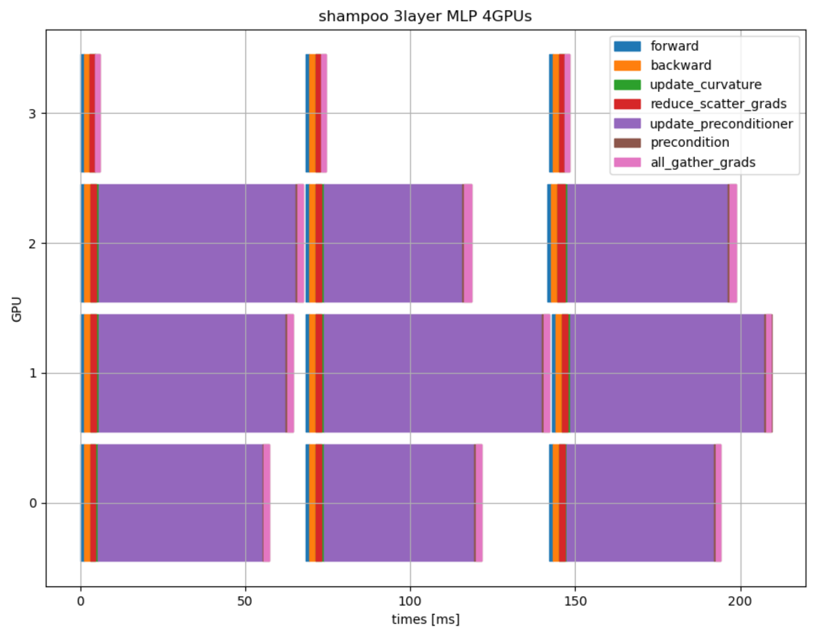


Figure 5.4: Runtime plot of the distributed Shampoo optimizer in ASDL running on 4 Tesla P100 GPUs on Piz Daint (i.e. 4 Nodes) a 3-layer multi-layer perception (MLP) with ReLU activations in between. The linear layers are of dimension 4×4 (4 in- and 4 output dimensions). We profiled three training steps after some warmup. `forward` is the forward pass and `backward` is the backward pass. `update_curvature` is the step from Shampoo updating matrix H_t^i (see algorithm 2 first line in second for-loop). `reduce_scatter_grads` is the step where each GPU (i.e. each DP dimension) reduces and scatters all the gradients to the specific GPU ranks, which have to compute the preconditioning matrices. `update_preconditioner` is the step when the p-root inverse of matrix H_t^i (see algorithm 4). `precondition` is the step when the preconditioning matrices are preconditioning the gradients (see algorithm 2 second line in second for-loop). Last, `all_gather` is the last step of one training step where the preconditioned gradients are communicated back to all the GPUs such the available optimizer updates the parameters of the NN. The 3-layer MLP was initialized randomly and the same was for the training batches.

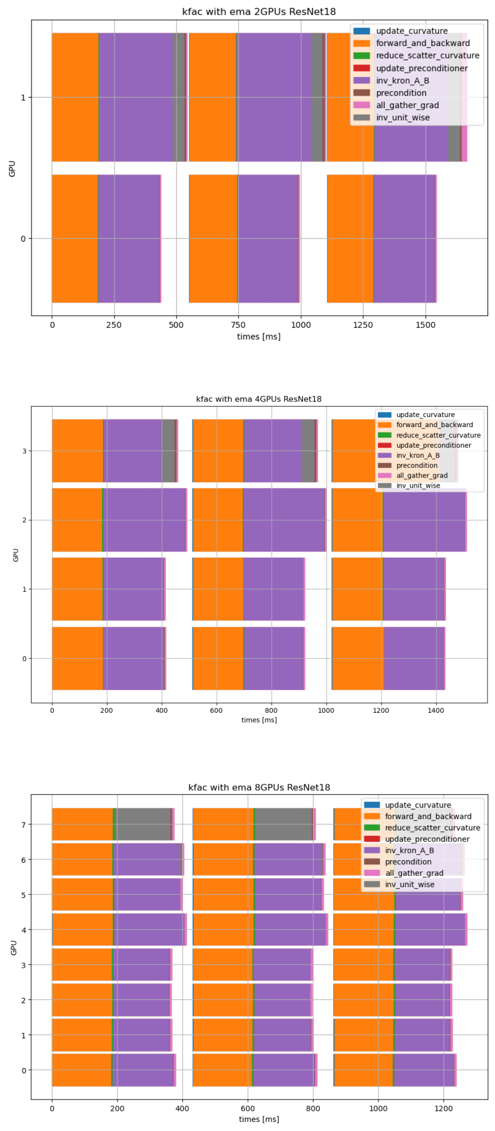


Figure 5.5: Three runtime plots of distributed K-FAC on the same ResNet18 NN with different numbers of GPUs trained on Piz Daint. The top figure is for 2 GPUs, the middle figure is for 4 GPUs, and the last figure is for 8 GPUs. `update_curvature` is the step where the Kronecker factures for linear and convolutional layers and for unit-wise layers the FIM are computed. `forward_and_backward` is the forward and backward pass. Note that the `forward_and_backward` is inside the `update_curvature` function call because the Kronecker factors are computed while doing the forward and backward pass via functorch extension. That is also why when using K-FAC, the forward and backward passes take longer than usual. `reduce_scatter_curvature` is the step where we communicate the Kronecker factors for linear and convolutional layers, FIM for unit-wise layers, and gradients to the GPU ranks, which have to compute their specific layers. `inv_kron_A_B` is the step where the inverse of the Kronecker factors A_{l-1} and G_l are computed. `precondition` is the preconditioning step of the gradients. `all_gather_grads` is the step where all the gradients which had been preconditioned are gathered back to all the GPUs. `inv_unit_wise` is the inverse computation for unit-wise layers such as batch normalization (see section 3.1.3 for information). `update_preconditioner` is actually the main function for all the inverse computation of the Kronecker factors and FIMs for unit-wise layers. This is also why it can't be seen on the plots because `inv_kron_A_B` and `inv_unit_wise` are overlapping it.

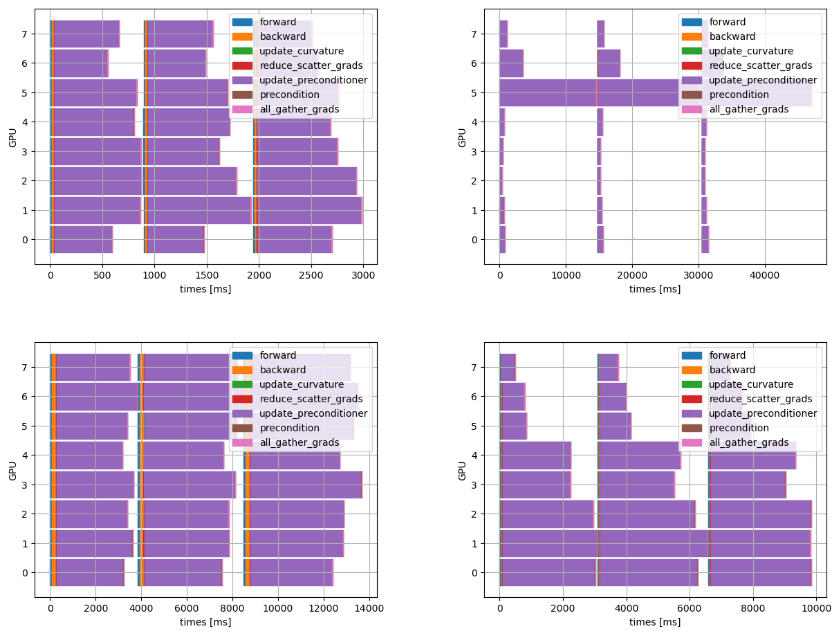


Figure 5.6: Runtime plots of distributed Shampoo on 4 known DNNs ResNet18 [17], VGG11 [35], DenseNet121 [18], and MobileNetV2 [33] with the Shampoo cost function eq. (5.2) and the exponent set to $p = 0.5$. All four DNNs were run on Piz Daint, with each having 8 GPUs. The runtime notations are described in fig. 5.4. The DNNs were measured with random numbers, hence no real dataset.

also why we ended up using $p = 0.4$ in the final implementation in ASDL, to be more conservative in balancing the workload.

One of the downsides of K-FAC is it scales by batch size. This is due to its empirical method of approximating the Kronecker factors. Hence we measured the performance of distr. K-FAC and distr. Shampoo on ResNet18 on a single GPU with different batch sizes and compare it to pure first-order optimization methods SGD and Adam (see fig. 5.7). Note that distr. K-FAC and distr. Shampoo are both implemented in ASDL, which means they can be combined with any wanted optimizer. In fig. 5.7, we used basic SGD for both distr. K-FAC and distr. Shampoo. One can see for all the methods except K-FAC, the throughput stays relatively constant, whereas K-FAC constantly increases, and the throughput stays the same. At a batch size of 512, K-FAC no longer fits on 16GB of GPU memory. Note the high runtime of forward and backward passes of K-FAC. This is due to the computation of the Kronecker factors whilst in the forward and backward pass via functorch. When compared to Shampoo, one notices the major impact of computing the Kronecker factors.

Finally, let's compare distr. K-FAC and distr. Shampoo side-by-side with a

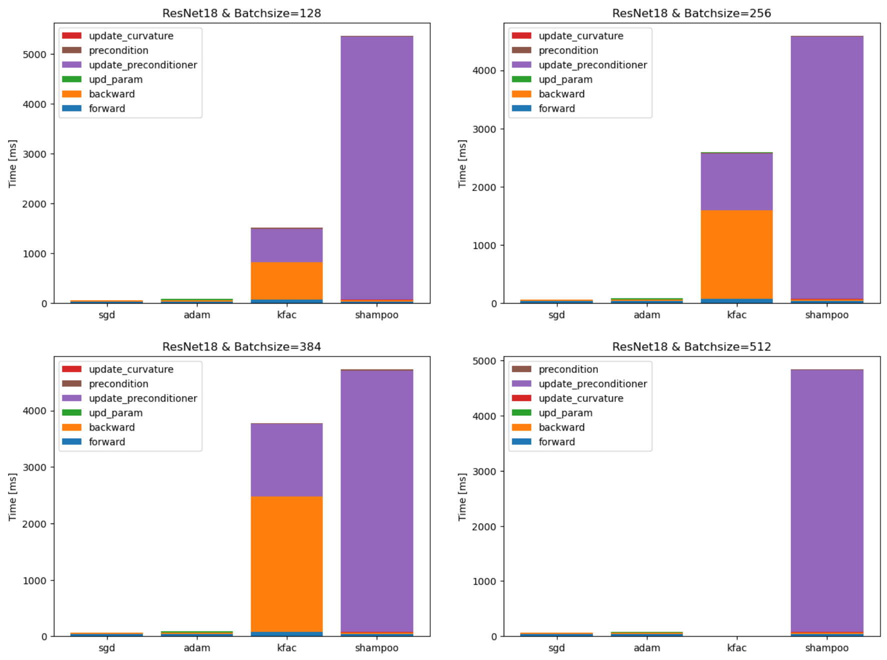


Figure 5.7: Bar plots of ResNet18 ran on the optimizers SGD and Adam and the distributed preconditioning methods K-FAC and Shampoo for different batch sizes of $128, 256, 384, 512$. For a batch size of 512, K-FAC no longer fit in 16GB of GPU memory. The preconditioning methods were combined with the SGD optimizer in ASDL. Less is better. We measured the runtime of each kernel for multiple training steps and averaged them. All were run on a single GPU. The model was trained on random numbers, no real dataset. The definitions of `update_curvature`, `precondition` and `update_preconditioner` for distr. K-FAC can be found in fig. 5.5 and for distr. Shampoo in fig. 5.4 respectively.

known NN, fixed batch size, and different numbers of GPU. The first model we tested was on DenseNet121 [18], which is a very deep NN consisting of convolutional and batch normalization layers having in total 242 layers when counting Batchnormalization and Convolutional layers individually. This makes preconditioning the DenseNet121 network well-distributed over many GPUs but very expensive for a small number of GPUs. In fig. 5.8 we plotted the runtime of one training step for distr. K-FAC (left) and distr. Shampoo (right) for different numbers of GPUs. Note that both y-axes are not on the same scale. Shampoo is around seven times as expensive as K-FAC for single GPU training. But when increasing the number of GPUs, we see a super linear scaling in runtime for distr. Shampoo and even surpassing in runtime at 16 GPUs (note the blue dashed line on the Shampoo plot). For distr. K-FAC more than 64 GPUs is no longer beneficial, and at this number of GPUs, one can already see the communication costs `all_gather_grad` and `reduce_scatter_curvature` getting thicker. Whereas for distr. Shampoo, we could have gone for even more GPUs. We also did

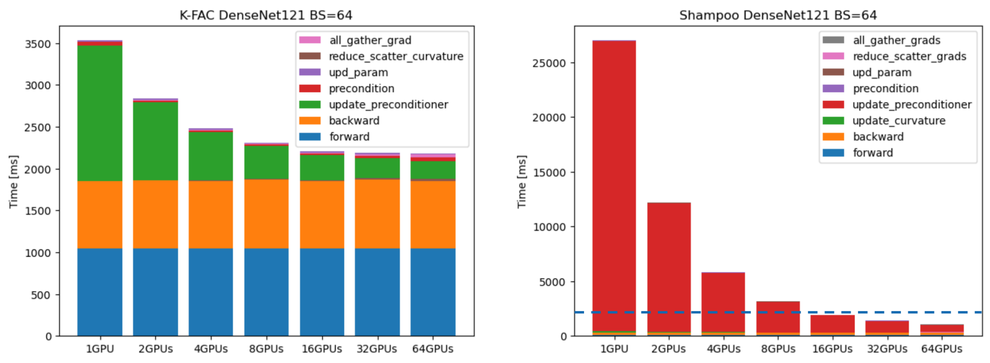


Figure 5.8: Two bar plots of both the distr. K-FAC (left) and distr. Shampoo (right) with the runtime of one training step averaged over multiple training steps. The y-axis is the runtime in milliseconds, and the x-axis shows the number of GPUs used. The model we used was DenseNet121 [18] with a batch size of 64. We tested the distributed preconditioning methods with up to 64 GPUs. The network was trained with random numbers. The definitions of `update_curvature`, `precondition`, `update_preconditioner`, and more for distr. K-FAC can be found in fig. 5.5 and for distr. Shampoo in fig. 5.4 respectively. The blue dashed line in the distr. Shampoo plot is the lowest measured runtime of distr. K-FAC $\approx 2300ms$ for 64 GPUs.

the same experiment on another NN called the WideResNet50 [44], which is the same as ResNet50 [17] but with wider layers which consist of more parameters (see fig. 5.9. WideResNet50 has 107 individual layers, which is twice as small as DenseNet121, which means less scalability. One can see that at 32 GPUs we no longer benefit in runtime performance by increasing the number of GPUs. Still, 32 GPUs training has double the throughput compared to 16 GPUs due to the same runtime performance.

Both figs. 5.8 and 5.9 show great scaling in the number of GPUs, especially when the NN is very deep. Distr. K-FAC is always capped at an already lower amount of GPUs due to its cost of computing the Kronecker factors during the forward and backward pass via functorch extensions. Runtime (i.e. throughput) is not the most important factor. It has been shown that for certain tasks, K-FAC performance is better than Shampoo in test accuracy with real datasets [27]. This means having a training step which leads to a greater step in accuracy can overcome the cost of being more expensive. This thesis does not go into the accuracy performance. There are many papers which compare the strength of second-order and first-order methods to one another (see chapter 2 or ASDL [27]). Now that we have seen the great distributed training performance of distr. Shampoo, in chapter 6, we are going to discuss our new optimizer which we call ”3D-Shampoo” which is distributed Shampoo for 3D parallelism.

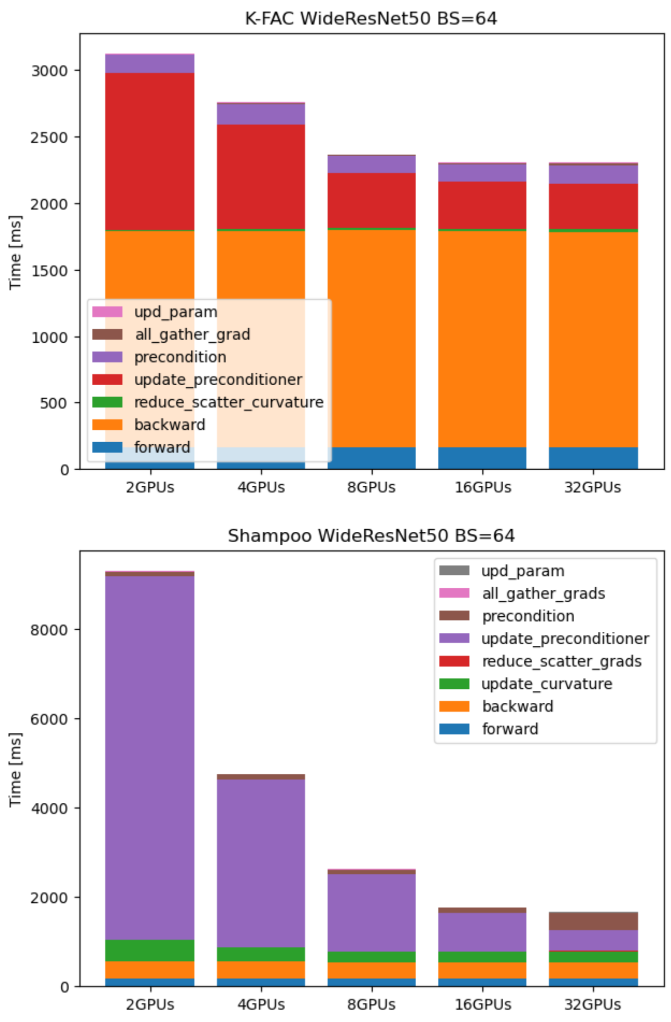


Figure 5.9: Two bar plots of both the distr. K-FAC (left) and distr. Shampoo (right) with the runtime of one training step averaged over multiple training steps. The y-axis is the runtime in milliseconds, and the x-axis shows the number of GPUs used. The model we used was WideResNet50 [44] with a batch size of 64. We tested the distributed preconditioning methods with up to 32 GPUs. The network was trained with random numbers. The definitions of `update_curvature`, `precondition`, `update_preconditioner`, and more for distr. K-FAC can be found in fig. 5.5 and for distr. Shampoo in fig. 5.4 respectively.

Chapter 6

3D-Shampoo

Figure 6.1: Little fun demonstration of the fusion between the three different libraries, making it into 3D-Shampoo. Credits go to the available logos from Nvidia, Google, Microsoft, and DeepSpeed as well as the little Pixel Goku image from ClipartMax.com [1].

In this chapter, we will describe our new optimizer which we call ”3D-Shampoo”. This optimizer is a modified version of Google’s Shampoo optimizer [5] (see chapter 4 for information about Shampoo). This new optimizer adaptively works with the DeepSpeed library [32] and depending on the level of parallelism changes its approach to computing the preconditioning matrices of Shampoo. The ”3D” in the name comes from DeepSpeed, which supports up to 3D parallelism, hence 3D-Shampoo. 3D parallelism is the

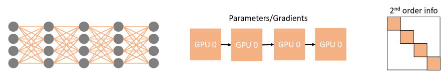
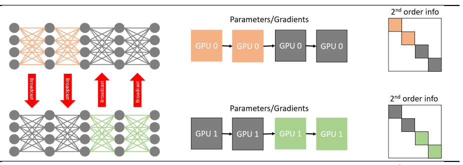
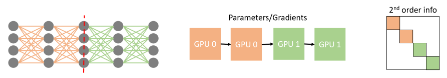
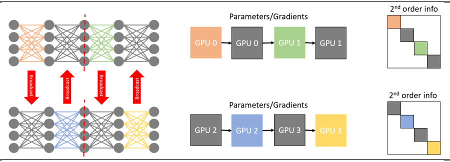
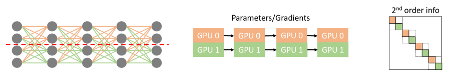
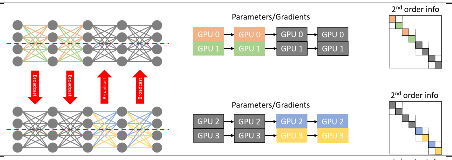
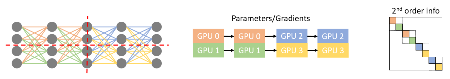
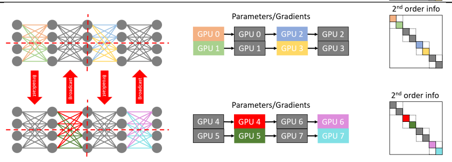
	None	ZeRO-DP
None		
PP		
OP		
PP + OP		

Table 6.1: Table of the different types of parallelism for NNs. (None) no parallelism. (ZeRO-DP) ZeRO-style Data parallelism with Optimizer States, Gradients, and Parameters. (PP) Pipeline parallelism. (OP) Operator parallelism. Combining all three parallelisms is known as 3D parallelism. The colours indicate which GPU stores which information from the NN. For ZeRO-DP, grey means not stored but dependent on which GPU it will be broadcasted and used as well. The dotted red line as well as the red arrows indicate the communication splits. Everything in those images is not for scale.

concept of combining the three different known deep learning parallelization (data parallelism, pipeline parallelism, and operator-/tensor parallelism) algorithms into one. We describe the concept of 3D parallelism in section 6.2. 3D-Shampoo is available on my GitHub reposetory[1] but use on your own risk. The code has been tested for some models and networks but not for all. It will probably be further improved/cleaned to be more user-friendly.

6.1 Collective Communication and the Piz Daint Network

In this section, we go through the communication cost of the Piz Daint network with the different kinds of MPI collective communications [39]. Because we are working on the Piz Daint supercomputer from CSCS [3], we need to understand its network for communicating between the GPUs for training large models with different levels of parallelism. We use the known measured communication cost of the Piz Daint network [4] to predict the

[1] https://github.com/noabauma/3d-shampoo

Type of Parallelism	Computational Cost	MLP
No Parallelism	$\mathcal{X}_{\{forw,back,upd\}} + \mathcal{X}_{prec}$	$6612Gflops$
ZeRO-DP	$\mathcal{X}_{\{forw,back,upd\}} + \mathcal{X}_{prec}$	$6612Gflops$
PP	$\frac{1}{N_{PP}}(\mathcal{X}_{\{forw,back,upd\}} + \mathcal{X}_{prec})$	$3306Gflops$
OP	$\frac{1}{N_{OP}}\mathcal{X}_{\{forw,back,upd\}} + \frac{1}{N_{OP}^3}\mathcal{X}_{prec}$	$3304Gflops$
ZeRO-DP + PP	$\frac{1}{N_{PP}}(\mathcal{X}_{\{forw,back,upd\}} + \mathcal{X}_{prec})$	$3306Gflops$
ZeRO-DP + OP	$\frac{1}{N_{OP}}\mathcal{X}_{\{forw,back,upd\}} + \frac{1}{N_{OP}^3}\mathcal{X}_{prec}$	$3304Gflops$
PP + OP	$\frac{1}{N_{PP}N_{OP}}\mathcal{X}_{\{forw,back,upd\}} + \frac{1}{N_{PP}N_{OP}^3}\mathcal{X}_{prec}$	$1652Gflops$
3D-Parallelism	$\frac{1}{N_{PP}N_{OP}}\mathcal{X}_{\{forw,back,upd\}} + \frac{1}{N_{PP}N_{OP}^3}\mathcal{X}_{prec}$	$1652Gflops$

Type of Parallelism	Communication Cost	MLP	
No Parallelism	0	0	0
ZeRO-DP	$LT_{broadcast}(N_{DP}, \Psi_{\{p,g\}}/L)$	$1151ms$	$8598MB$
PP	$T_{P2P_{uni}}(B(n_{in} + n_{out}))$	$0.2ms$	$2MB$
OP	$LT_{allgather}(N_{OP}, Bn_{out}) + LT_{(all)reduce}(N_{OP}, Bn_{in})$	$221ms$	$2147MB$
ZeRO-DP + PP	$LT_{broadcast}(N_{DP}, \Psi_{\{p,g\}}/L) + T_{P2P_{uni}}(B(n_{in} + n_{out}))$	$1151ms$	$8600MB$
ZeRO-DP + OP	$LT_{broadcast}(N_{DP}, \Psi_{\{p,g\}}/L) + LT_{allgather}(N_{OP}, Bn_{out}) + LT_{(all)reduce}(N_{OP}, Bn_{in})$	$1371ms$	$11GB$
PP + OP	$T_{P2P_{uni}}(B(n_{in} + n_{out})) + \frac{L}{N_{PP}}T_{allgather}(N_{OP}, Bn_{out}) + \frac{L}{N_{PP}}T_{(all)reduce}(N_{OP}, Bn_{in})$	$111ms$	$1076MB$
3D-Parallelism	$LT_{broadcast}(N_{DP}, \Psi_{\{p,g\}}/L) + T_{P2P_{uni}}(B(n_{in} + n_{out}))$ $+\frac{L}{N_{PP}}T_{allgather}(N_{OP}, Bn_{out}) + \frac{L}{N_{PP}}T_{(all)reduce}(N_{OP}, Bn_{in})$	$1261ms$	$9674MB$

Type of Parallelism	Memory Consumption	MLP
No Parallelism	$\Psi_{\{p,g,opt\}} + \Psi_{prec}$	$17.2GB$
ZeRO-DP	$\frac{1}{N_{DP}}(\Psi_{\{p,g,opt\}} + \Psi_{prec})$	$8.6GB$
PP	$\frac{1}{N_{PP}}(\Psi_{\{p,g,opt\}} + \Psi_{prec})$	$8.6GB$
OP	$\frac{1}{N_{OP}}\Psi_{\{p,g,opt\}} + \frac{1}{N_{OP}^2}\Psi_{prec}$	$6.4GB$
ZeRO-DP + PP	$\frac{1}{N_{DP}N_{PP}}(\Psi_{\{p,g,opt\}} + \Psi_{prec})$	$4.3GB$
ZeRO-DP + OP	$\frac{1}{N_{DP}N_{OP}}\Psi_{\{p,g,opt\}} + \frac{1}{N_{DP}N_{OP}^2}\Psi_{prec}$	$3.2GB$
PP + OP	$\frac{1}{N_{PP}N_{OP}}\Psi_{\{p,g,opt\}} + \frac{1}{N_{PP}N_{OP}^2}\Psi_{prec}$	$3.2GB$
3D-Parallelism	$\frac{1}{N_{DP}N_{PP}N_{OP}}\Psi_{\{p,g,opt\}} + \frac{1}{N_{DP}N_{PP}N_{OP}^2}\Psi_{prec}$	$1.6GB$

Table 6.2: Table of different types of NN parallelizations with their approximated computational cost, communication cost and memory consumption, everything based on per GPU. (ZeRO-DP) stands for ZeRO-style data parallelism and depending on the level, you can use it on parameters p, gradients g, optimizer states opt, and preconditioning matrices $prec$. Depending on the need, it can be used on all four or only on specific ones. B is the batch size. n_{in} and n_{out} are the input and output dimensions of some layers. L is the number of layers the whole NN has. Note that for an MLP consisting of L similar layers without bias: $\Psi_p = L(n_{out}n_{in})$. T_{xx} are the different types of communication costs for the different types of communications xx shown in table 6.3. (PP) stands for Pipeline Parallelism and (OP) for Operator Parallelism. $N = N_{DP}N_{PP}N_{OP}$ is the number of GPUs. N_{xx} is the number of GPUs in the given parallelization dimension. $\mathcal{X}_{xx}$ is the computational cost of a given task. $forw$ is the computational cost of computing the forward pass, the same for $back$ being the backward pass. upd is the cost of updating the parameters of a given optimizer. $prec$ is the cost of everything needed for the Shampoo algorithm, which includes computing the preconditioning matrix as well as preconditioning the gradients. Ψ_{xx} are the size in Bytes of a datatype xx. We also stated the communication cost in Bytes for the two examples models. We made some assumptions with an MLP example. MLP is a multilayer perceptron consisting of $n_{in} = n_{out} = L = B = 1024, dtype = float, N_{DP} = N_{PP} = N_{OP} = 2$ with the SGD optimizer and ran on Piz Daint.

Communication	Algorithm	function name	Communication Cost
P2P (bidirectional)	-	$T_{P2P}(n)$	$\alpha + n\beta$
P2P (unidirectional)	-	$T_{P2P_{uni}}(n)$	$\alpha_{uni} + n\beta_{uni}$
ALLGATHER	Recursive Doubling	$T_{allgather}(p, n)$	$\log(p)\alpha + (p-1)\frac{n}{p}\beta$
BROADCAST	van de Geijn	$T_{broadcast}(p, n)$	$(\log(p) + p - 1)\alpha + 2(p-1)\frac{n}{p}\beta$
ALL-TO-ALL	Bruck	$T_{all2all}(p, n)$	$\log(p)\alpha + \frac{n}{2}\log(p)\beta$
REDUCE-SCATTER	Long messages	$T_{red-sca}(p, n)$	$(p-1)\alpha + (p-1)\frac{n}{p}\beta + (p-1)\frac{n}{p}\gamma$
REDUCE & ALLREDUCE	Rabenseifner	$T_{(all)reduce}(p, n)$	$2\log(p)\alpha + 2(p-1)\frac{n}{p}\beta + (p-1)\frac{n}{p}\gamma$

Table 6.3: α is the latency (or startup time), β is the transfer time per byte, n is the number of bytes transferred, p is the number of processes, γ is the computation cost per byte. P2P stands for "Point-to-Point" communications, including block and non-blocking Send/Recv communications. The notations and the calculations are based on *Thakur et al.* [39]. In *Thakur et al.* paper, they demonstrated different types of algorithms for the same type of communication, depending on the size of bytes n or the number of processors p and other factors, one is better than the other. We chose the ones that are more suitable for distributed DNN training.

communication cost of our 3D-Shampoo algorithm for different NNs. The notations are adopted from the work of *Thakur et al.* [39]. Table 6.3 gives an overview of all the different types of communications with their respective cost.

Piz Daint of CSCS uses the Cray Aries routing and communications ASIC and Dragonfly network topology [4]. Given the table 6.3, we can construct the measured communication costs from [4] for the different types of communications. The following measurements are based on an Aries routing using Intel Xeon E5 CPUs. The measured latency is $\alpha \approx [1.3, 2.0]\mu m$, depending on message size and the network's quietness. Same goes for $\alpha_{uni} = \alpha$. The peak bandwidth for unidirectional traffic goes up to $10GB/s$ which is $\beta_{uni} \approx 0.067ns/B$. The peak bandwidth in bidirectional traffic is $7.5GB/s$ in both directions, which is $\beta \approx 0.133ns/B$. Note that the peak bandwidth for both uni- and bidirectional traffic is around $64KB$, which is not unusual for big DNNs with trillions of parameters. For example, a linear layer consisting of floats (4 Bytes) of dimensions $128 * 128$ has a size of $65'536B$. For the computational cost per Byte γ, we use the peak performance of an Nvidia Tesla P100 16GB GPU (single precision) of $9340GFLOPs$, which is $37360GB/s$ of computed Bytes per second. Hence $\gamma \approx 2.68e - 14s/B$.

6.2 3D Parallelism

In this section, we describe what 3D parallelism is. Deepspeed's 3D parallelism is a parallelization technique for training deep learning models that exploit three dimensions of parallelism: data parallelism (DP), pipeline parallelism (PP), and operator parallelism (OP).

Data Parallelism (DP) refers to the parallelization of distributing the training data over multiple GPUs. One GPU gets a micro-batch size to compute its gradients. Upon updating the parameters, the gradients of each micro-batch are then averaged via an `all_reduce()`. This parallelization is the simplest form of parallelization in deep learning. One has to copy the whole model on all the GPUs and only the data will be split across the GPUs. It only needs one collective communication if no ZeRO-style optimization [32] is happening to improve the memory consumption further if dealing with very big NNs. If ZeRO-style optimization is active, the communication cost will get very big quickly (see table 6.2). In the case of 3D parallelism, DP can be considered as the number of duplicates of the NN with the same set of OP and PP, but each copy works with another micro-batch.

Operator Parallelism (OP) refers to the practice of splitting the model layer weights. This is not a trivial task of splitting at a certain point of the model as in PP. Depending on the type of layers, the splits for OP have to be done individually for the specific layers such that they still achieve the desired functionality. This is also up to the model designer how to split the layer weights such as Megatron-LM's GPT-2. *Narayanan et al.* [25] have a great paper on how they achieve OP on GPT-2.

Pipeline Parallelism (PP) involves splitting the model into certain layers and calling them stages. Each stage will then be stored at a certain GPU. This technique is useful when dealing with big neural networks where the whole model can't fit into one GPU. An advantage of PP is having multiple splits and at multiple GPUs, like the name "pipeline" already indicates when passing micro-batches through the stages one micro-batch will be handled by one stage at a time in a pipeline fashion. The communication part happens as point-to-point communication when passing the output of one stage to another. This is done both independently for the forward pass as well as for computing the gradients for the backward pass. Due to the concept of the chain rule, the backward pass of the pipeline stage has to communicate the gradients of its first layer to the previous stage such that the previous stage can continue computing its gradients with the backward pass. PP is the method with the least amount of communication. In the case of 3D parallelism, PP works the same.

We created a simple visualization of the three different kind of parallelism and their combinations on a simple four-layer feed-forward NN shown in table 6.1. One can also see how the second-order information will be split across multiple GPUs. especially when OP is active, the shapes of the second-order information matrices will get smaller. Second-order information does not represent the true shapes of the preconditioning matrices of the Shampoo algorithm. They would be Kronecker factors of that second-order information, with each layer having two of them (input size and output size). Preconditioning matrices can't be well represented in a full Hessian matrix, which is why we opted for this visualization. Note the parameter matrices and second-order information are not for scale. The figure should only guide the viewer on how different types of parallelism are, which information it holds, and where the communication happens.

We came up with mathematical formulas on how to calculate the computational cost, communication cost, and memory consumption of the different kinds of parallelism and their combinations per GPU bases in table 6.2. It also includes an example model with the assumed costs. The table demonstrates the effect of increasing the number of parallelisms (i.e. number of GPUs) decreases the computational and memory consumption but with the cost of increasing the communication cost.

For more information about 3D parallelism, Microsoft Research has a great blog post on this topic on their webpage[2].

6.3 DeepSpeed Library

DeepSpeed is a popular open-source library developed by Microsoft for deep learning on large-scale models with massive computational requirements. It provides an efficient and scalable solution to the challenges of training deep learning models that require massive amounts of data and computing power. DeepSpeed optimizes training performance and memory efficiency by implementing a range of techniques such as gradient accumulation, tensor fusion, and memory optimization. DeepSpeed is known for the ZeRO-style optimization to reduce the memory footprint on a neural network training over multiple GPUs. It also supports distributed training on a wide range of hardware, from a single GPU to thousands of nodes in a cluster. The current version of the DeepSpeed library itself can handle data parallelism and pipeline parallelism for any given model. It also supports models which handle OP (e.g. Megatron-LM's GPT-2). Hence, if we give DeepSpeed a model which supports OP, we can achieve 3D parallelism. In our case, we use DeepSpeed for its DP and PP without ZeRO optimization due to actually updating and storing the preconditioning matrices. ZeRO-style optimization is obsolete when combined with 3D-Shampoo.

6.4 Megatron-LM's GPT-2

In this section, we go through the architecture of Megatron-LM's GPT-2. We do that to understand better how different levels of parallelism (DP, PP, OP, and up to 3D parallelism) will affect Megatron-LM's GPT-2 and what 3D-Shampoo can precondition. Megatron-LM's GPT-2 is a variant of the popular GPT-2 language model developed by OpenAI, which has been widely used for a variety of natural language processing tasks, including

[2]https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone

text generation, question answering, and language translation. However, Megatron-LM's GPT-2 model is characterized by its ability to scale to very large sizes, enabling it to process huge amounts of text and generate more accurate and refined responses.

One of the key differences between Megatron-LM's GPT-2 and other GPT-2 models is its use of parallelism and distributed computing to speed up training and improve performance. Designed to run on large-scale supercomputers with thousands of GPUs, Megatron-LM can train models with billions of parameters in a fraction of the time required by traditional training methods.

We use Megatron-LM's because of its ability to parallelize their models in operator parallelism style such that we can achieve 3D parallelism with DeepSpeed.

The main building block of Megatron-LM's GPT-2 is as follows:

1. Input embedding layer: This layer converts the input text into a sequence of vectors that can be processed by the model. Each token in the input sequence is represented as a vector in a high-dimensional space, where similar vectors correspond to similar tokens.
2. Positional coding layer: This layer adds information about the position of each token in the input sequence. This allows the model to differentiate between tokens that appear in different positions, even if they have the same representation in the input embedding layer.
3. Transformer decoder layers: These layers are the core of the GPT-2 model and consist of several transformer blocks that process the input sequence in parallel by the number of attention heads. Each transformer block consists of a self-attention mechanism that allows the model to focus on different parts of the input sequence and has an MLP with GeLU activations at the end. A graph is shown of those layers' parameters in fig. 6.3. This is where 3D-Shampoo will perform preconditioning because these layers have parameters of the shape of matrices.
4. Decoder Head Layer: This layer takes the output of the final transformer block and generates a probability distribution over the vocabulary of possible next tokens. This distribution is used to generate new text by sampling tokens from it one at a time.

The OP of Megatron-LM will take place in the embedding layer as well as in the transformer decoder layers of GPT-2. Megatron-LM will split those layers by the number of parameters to reduce the memory in GPU and accelerate training across multiple GPUs. The embedding layer is very huge due to its vocabulary size of 50'304. 3D-Shampoo does not consider preconditioning the embedding layer due to its sheer size. Still, the embedding layer will be updated via a momentum-based optimizer (see algorithm 5). But the transformer decoder layers are the ones in which 3D-Shampoo will perform preconditioning (with or without OP active). The transformer decoder layers consist of 4 matrix-shaped parameters: The query, key, and value parameters concatenated into one matrix-shaped tensor, the linear layer at the ending of the self-attention, and the two linear layers in the MLP. We visualized the transformer decoder layer in fig. 6.3. Information about how Megatron-LM performs OP on the transformer decoder layers can be read in *Narayanan et al.* [25]. One thing to note is that the number of attention heads depends on Megatron-LM's OP level. This means the number of attention heads has to be divisible by the level of OP. Every other layer which needs to be updated is flat and can't be preconditioned with Shampoo. Those layers will be updated normally via an SGD or Momentum-based optimizer. When building GPT-2, we consider using the number of transformer decoder layers to be the same as the number of GPUs (or at least being divisible by the number of GPUs), such that for any number of parallelism in DP, PP or OP, each GPU has the same number of transformers. E.g. if we have eight GPUs, each GPU would need one transformer decoder layer to perform optimization/preconditioning. Hence in total, we would need eight transformer decoder layers. We visualized such an example of Megatron-LM's GPT-2 consisting of eight transformer decoder layers for different levels of parallelism in fig. 6.2. With the paper of *Phuong et al*: "Formal Algorithms for Transformers" [31], we were able to create a better understanding of how the GPT-2 transformer is built and hence were able to visualize the GPT-2 for different parallel settings.

6.5 3D-Shampoo's Algorithm

In this section, we discuss 3D-Shampoo's algorithm and how it uses its distributed preconditioning method for any combination of parallelism up to 3D parallelism from DeepSpeed to achieve maximum throughput.

We have discussed the approach on distributed Shampoo in chapter 4 which works similarly for 3D-Shampoo when combined with DeepSpeed. DeepSpeed has a feature called `ProcessTopology`[3] to find out which GPU is assigned which DP, PP, OP stages, how many GPUs are in which parallelism N_{DP}, N_{PP}, N_{OP} and which GPU is from which parallelism group group_{DP}, group_{PP}, group_{OP}. This topology method is very handy for us in figuring out how to distribute the preconditioning method over all available GPUs. 3D-Shampoo does distributed preconditioning method when DP is active (i.e. $N_{DP} \geq 2$). For any other cases of parallelism and their combinations without DP, there is no need to do distributed preconditioning because

[3] https://deepspeed.readthedocs.io/en/latest/pipeline.html#deepspeed.runtime.pipe.ProcessTopology

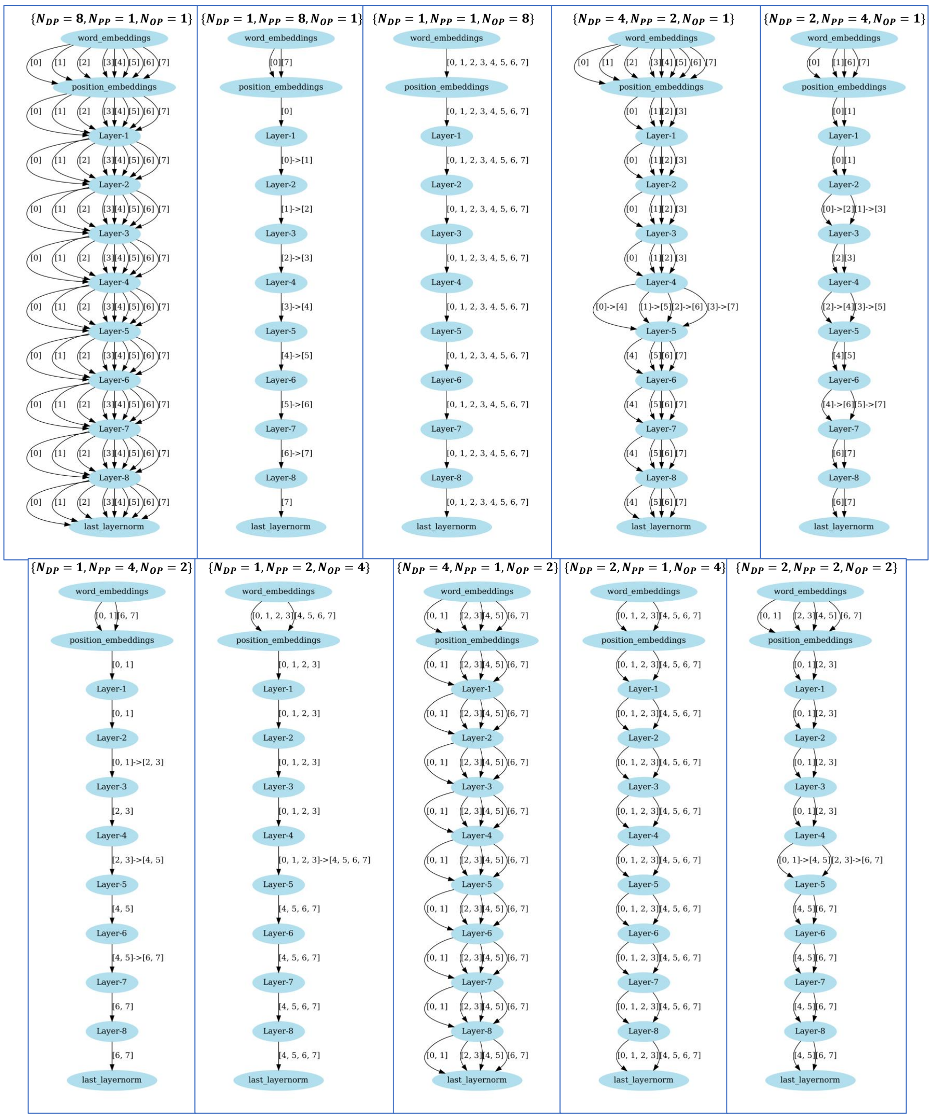

{$N_{DP} = 8, N_{PP} = 1, N_{OP} = 1$}
{$N_{DP} = 1, N_{PP} = 8, N_{OP} = 1$}
{$N_{DP} = 1, N_{PP} = 1, N_{OP} = 8$}
{$N_{DP} = 4, N_{PP} = 2, N_{OP} = 1$}
{$N_{DP} = 2, N_{PP} = 4, N_{OP} = 1$}
{$N_{DP} = 1, N_{PP} = 4, N_{OP} = 2$}
{$N_{DP} = 1, N_{PP} = 2, N_{OP} = 4$}
{$N_{DP} = 4, N_{PP} = 1, N_{OP} = 2$}
{$N_{DP} = 2, N_{PP} = 1, N_{OP} = 4$}
{$N_{DP} = 2, N_{PP} = 2, N_{OP} = 2$}
word_embeddings
position_embeddings
Layer-1
Layer-2
Layer-3
Layer-4
Layer-5
Layer-6
Layer-7
Layer-8
last_layernorm

Figure 6.2: Visual demonstration of Megatron-LM's GPT-2 model for different levels of 3D parallelism. N_{DP} states the number of data parallelism, N_{PP} number of pipeline parallelism, and N_{OP} operator parallelism. $N = N_{DP}N_{PP}N_{OP} = 8$ GPUs for all the graphs.The tokens start at the `word_embedding` layer and output at the last normalization layer. The Layer-X node is one of the transformer decoder layers of GPT-2 and fig. 6.3 shows one of the layers in detail. The labels of the edges show which rank passes through which node. When ranks are grouped into a list, they are of the same operator parallelism group. $[X] \to [Y]$ indicates the pass to the next pipeline stage from the next GPU Y. Multiple edges directing into the same layers indicate the number of data parallelism. For a certain level of parallelism, Megatron-LM uses multiple ranks to parallelize the embedding layers, even if they are not of the same group of the pipeline stage. The graphs were made with Graphviz.

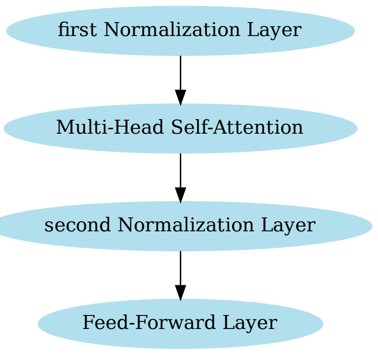

first Normalization Layer
Multi-Head Self-Attention
second Normalization Layer
Feed-Forward Layer

Figure 6.3: One of the transformer decoder layer of Megatron-LM's GPT-2. The first layer is a normalization to improve the stability and efficiency of the model. The second layer is the Multi-head self-attention layer. This layer allows the model to simultaneously attend to different parts of the input sequence. It works by projecting the input sequence into multiple attention heads, each of which computes an attention distribution over the entire sequence. The outputs of each attention head are concatenated and passed through a linear projection layer. The Multi-head self-attention layer holds the parameters of the Query, Key, and Value matrices concatenated into one matrix-shaped tensor. The third layer is again a normalization. The fourth and last layer is a feed-forward layer. This layer applies a non-linear transformation to the output of the self-attention layer. It consists of two linear layers with a GeLU activation function in between which are also preconditioned by 3D-Shampoo. This graph was made with Graphviz.

when 3D-Shampoo gets the set of parameters from the model to update, the given parameters are already split accordingly to PP and OP by DeepSpeed and Megatron-LM. When DP is active in DeepSpeed, we don't have to `reduce_scatter` the parameters because DeepSpeed already `all_reduce` the parameters after the backward pass. Hence, we only have to distribute the preconditioning across all the available GPUs of the given DP group. For a quick summary of how distributed preconditioning is handled by 3D-Shampoo, see table 6.4. Distributed Shampoo and 3D-Shampoo have minor changes in the final `all_gather` step. Distr. Shampoo does the `all_gather` step right after the given gradients have been preconditioned by the preconditioning matrices, such that any wanted optimizer (e.g. SGD, Adam) will update the parameters of the model, whereas 3D-Shampoo is an optimizer itself. Hence, it does the `all_gather` after it updates the parameters given by DeepSpeed and Megatron-LM's PP and OP splits. The `all_gather` is only done in the same DP groups. This means only the GPU ranks which have the same layers given by DeepSpeed and Megatron-LM's PP and OP have to communicate back their distributed and updated parameters. This means there are multiple `all_gather` communications happening for each DP group.

We made a pseudo-code of the 3D-Shampoo algorithm shown in algorithm 5.

Algorithm 5 3D-Shampoo one optimization step for a given set of weights

```
Initialize: rank $\in \{0, \dots, N-1\}$, partitioning, group$_{DP}$, group$_{PP}$, group$_{OP}$

Receive gradients $G_{1:L}$ and weights $W_{1:L} \in \{\text{group}_{PP} \cap \text{group}_{OP}\}$
for $1 = 1, \dots, L$ do
  if rank == group$_{DP}$[partitioning[$l-1$]] then
    $k \leftarrow dim(G_l)$
    if $k \geq 2$ then
      Compute preconditioning matrices and precondition gradients via algorithm 2:
          $\tilde{G}_l \leftarrow \texttt{Precondition}(G_l)$
    else
      Don't precondition:
          $\tilde{G}_l \leftarrow G_l$
    end if
    Weight decay (if wanted):
        $\tilde{G}_l \leftarrow \tilde{G}_l + \lambda W_l$
    Update gradient with momentum (if wanted):
        $\tilde{G}_l \leftarrow \tilde{G}_l + \mu B_l$
    Update parameters:
        $W_l \leftarrow W_l - \eta \tilde{G}_l$
  end if
end for
if $N_{DP} > 1$ then
  all_gather($W_{1:L}$, group$_{DP}$)
end if
```

At the initialization of 3D-Shampoo, we need the global rank of the GPU where $N = N_{DP}N_{PP}N_{OP}$ is the total number of GPUs. "partitioning" is a list of indexes which specifies which GPU from the DP group has to update the parameters of the given layer. We discussed the method of partitioning the workload in chapter 5 and the algorithm for that is shown in listing A.1. group$_{DP}$, group$_{PP}$, group$_{OP}$ are each list containing the ranks of the GPUs from the same group. E.g. every GPU which has the first PP stage are on the same list group$_{PP}$. When written $\{\text{group}_{PP} \cap \text{group}_{OP}\}$, we effectively mean only the intersection of these two groups. The gradients and parameters (weights) of this intersection are automatically given by DeepSpeed with the gradients all ready been `all_reduce` in the same DP group. The layer index $1, \dots, L$ is not necessarily the global index of the whole model. Our 3D-Shampoo algorithm has some additional steps after the preconditioning: "Weight Decay" and "Update gradient with momentum". Those are directly taken from the improved Shampoo algorithm from *Anil et al.* [5]. These extra methods are useful not only for preconditioned layers but also for layers

Type of parallelism	Shampoo distributed preconditioning method dependent on the types of parallelism.
None	None. We only have 1 GPU, which has to compute all the preconditioning matrices of all the layers.
DP	Split layers by number of GPUs.
PP	The Pipeline splits decide which GPU does compute which preconditioning matrices.
OP	OP split the weights accordingly to each GPU. No distribution is needed.
DP + PP	Each Pipeline split will be further split by N_{DP}.
DP + OP	Each OP split will be further split by N_{DP}.
PP + OP	The PP + OP splits decide which GPU does compute which preconditioning matrices.
3D Parallelism	Each PP + OP split will be further split by N_{DP}.

Table 6.4: Table of our approach of distributing the preconditioning method of Shampoo over the available GPUs to achieve maximum throughput with any given parallelism and their combinations. DP stands for (ZeRO-style) Data Parallelism. The level of ZeRO memory optimization does not impact our approach to computing a Shampoo optimization step. PP stands for Pipeline Parallelism and OP for Operator/Tensor Parallelism. N_{xx} is the number of parallelism in a given dimension $xx \in \{DP, PP, OP\}$. $N = N_{DP}N_{PP}N_{OP}$ is the total number of GPUs. Note that if $N_{DP} > 1$ (i.e. DP is active), we average all of the gradients of their parallel partner ranks in an `all_reduce` fashion before computing Shampoo's preconditioning matrices. Consider the example for DP + PP: $N_{PP} = 3$, $N_{DP} = 2 \rightarrow N = 6$: 3 pipeline stages with 2 GPUs in each pipeline stage do split their work of computing the preconditioning matrices.

which can't be preconditioned. Lastly when the parameters of the layers have been updated, when DP is active (i.e. $N_{DP} > 1$), we have to `all_gather` the gradients back to ranks that are in the same group$_{DP}$.

6.6 3D-Shampoo Implementation

In this section, we will discuss how we implemented 3D-Shampoo and how to run it on Piz Daint. The DeepSpeed library [32] provided by Microsoft is used to perform ZeRO-style data parallelism (ZeRO-DP) and Pipeline parallelism (PP). We combine DeepSpeed with Nvidia's Megatron-LM library [34] consisting of language models for Natural Language Processing (NLP) such as BERT and GPT-2. These support operator parallelism (OP) to achieve the desired 3D Parallelism. In the Megatron-LM library, OP is also known there as model parallelism. With Megatron-LM's GPT-2 is how DeepSpeed demonstrated the concept of 3D Parallelism [37]. Shampoo [5] can be directly taken from the Google research repository[4]. This Shampoo implementation

[4] https://github.com/google-research/google-research/tree/master/scalable_shampoo/pytorch

inherits the base class of PyTorch optimizer, which makes it easy to use with the DeepSpeed library. This is also the Shampoo version which 3D-Shampoo inherits.

The first step was to introduce the DeepSpeed library on Piz Daint. We had to minorly adapt the `deepspeed.init_distributed()` method, such that it can handle the TCP protocol system of Piz Daint's Slurm system. This fix was approved and added to the main DeepSpeed git repository as a pull request (PR) and is now for all users available[5]. Another small fix we had to do to the DeepSpeed library was the backward pass gradient of the pipeline engine of DeepSpeed to deal with the masks of Megatron's GPT-2. We fixed this bug by ignoring the mask during the backward pass when passing the gradients to the next PP stage. This bug probably happened due to some changes in Megatron-LM's GPT-2 model over the last 8 months. The current version of DeepSpeed (version 0.8.0) had a similar hacky solution, but we had to redo their hacky fix again. We had to install a PyTorch extension from Nvidia called Apex [26] to use any of Megatron's language models. To install it on Piz Daint, one has to use a matching CUDA version of PyTorch and CUDA itself on Piz Daint. In our case, we had to rely on PyTorch version `1.10.0+cu111` combined with Piz Daint's CUDA version 11.1. Finally, we were able to combine DeepSpeed ZeRO-DP and PP with Megatron's GPT-2 OP to achieve 3D parallelism on Piz Daint. To use Shampoo for any model, one has to add it as an optimizer.

6.7 3D-Shampoo Results

This section discusses our measured results of 3D-Shampoo on Piz Daint. We tested 3D-Shampoo with many different settings and variables to have a feel for how it behaves in different settings. As a note, every measurement of the runtime and throughput has been done on random tokens. Hence no real datasets. We ran measurements on the model of Megatron-LM GPT-2 due to being the model OP, which has been run and tested on DeepSpeed to achieve 3D parallelism. The hyperparameters of Megatron-LM's GPT-2 for all the different measurements have been: `--hidden-size=1024`, `--num-attention-heads=16`, `--seq-length=1024`, `--max-position-embeddings=1024`, `--fp32`, `--seed=42`, and `--lr=1.5e-4`. We used a reasonable learning rate to simulate normal optimizer steps. We also decided to fix the float size to 32bit because the preconditioning matrices are also of this datatype. Another tunable hyperparameter of Megatron-LM's GPT-2 is the `--um-layer=X`, which specifies how many transformer decoder layers we want to have (see section 6.4 for more information). For all the results, we did not precondition the embedding layer because they are too big. We only preconditioned the

[5]https://github.com/microsoft/DeepSpeed/pull/2905

transformer decoder feed-forward layers. We set up the model such that one GPU has to precondition at least one transformer decoder layer or the number of transformer decoder layers is divisible by the number total number of GPUs. This makes distributing the workload for preconditioning the layers simple. In 3D-Shampoo, each GPU has a certain amount of transformer decoder layers to precondition, depending on the total number of transformer decoder layers and the total number of GPUs. Some GPUs from the first PP stage also have to update the embedding layer with a momentum-based optimizer built already inside 3D-Shampoo and some GPUs at the end of the PP stages have an extra layer normalization at the ending. Those GPUs have to do some extra work as well as communicate the updated parameters of those extra layers back via `all_gather()` to all the GPUs from the same DP group. It is not that much work and is not as big an imbalance as the actual precondition of the given transformer decoder layers. Like stated in chapter 5, depending on how well-conditioned preconditioning matrices of certain layers are and even if they are of the same size of the shape, some take longer to compute the p-root and inverse. We used a vocabulary of size 50'304, and the tokens were generated with a uniform distribution in the range $[0, 5000]$. Every throughput and runtime measurement was performed after some warmup training steps and then at around 3-10 training steps were measured. Depending on the runtime, we used fewer training steps because some took at least 5 minutes. The shapes of the preconditioning matrices of one transformer decoder layer with 16 attention heads and hidden size of 1024 are: 2×4096^2, 1×3072^2, and 5×1024^2 which makes in total eight preconditioning matrices for one transformer decoder layer. This makes $48.2e7$ 32bit floats in total and multiplied by $2 * 4$ gives $385.9MB$ of preconditioning memory used per transformer decoder layer. We multiply by 2 because 3D-Shampoo stores the preconditioning matrices with and without the p-root inverse computation. This makes 3D-Shampoo kinda big per transformer decoder layer, but it the size can be further reduced by using the block partitioning method from *Anil et al.* [5]. Block partitioning shrinks the sizes of the preconditioning matrices into smaller desired shapes but for the cost of increasing the number of preconditioning matrices. When introducing the block partitioning method of 3D-Shampoo, we can further reduce the memory cost by setting the shapes to a size of 1024, which gives us a total of 24 preconditioning matrices per transformer decoder layer which is in a total of $201.3MB$. See fig. 6.4 for an overview of how the shapes of the preconditioning matrices are changing for one transformer decoder layer by changing the desired shapes in the block partitioning method.

The first thing we measured was to see how 3D-Shampoo performance compared to the most simple/lightweight/known optimizer, the Stochastic Gradient Descent (SGD) to give an expected runtime performance and memory consumption baseline. We compared both 3D-Shampoo and SGD to a

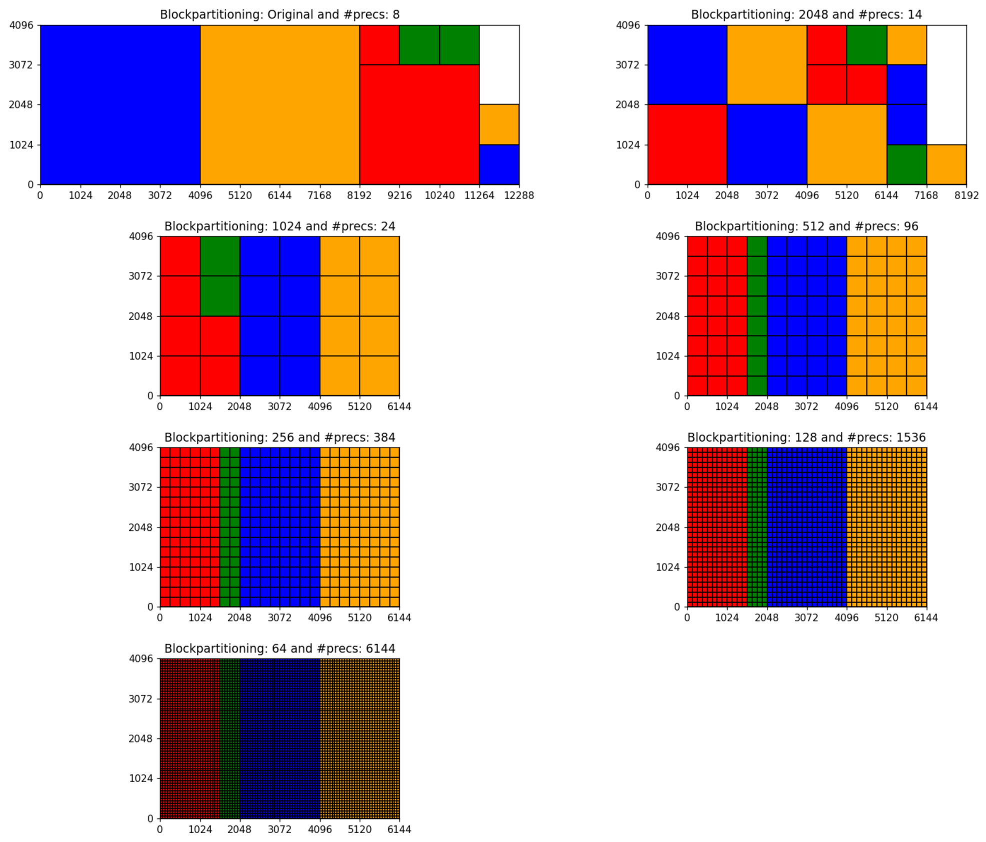


Figure 6.4: Figures of the different preconditioning matrices of one transformer decoder layer of GPT-2 with different levels of BlockPartitioning. The colours indicate which preconditioning matrices are for which layers. `attention.query_key_value` is red, `attention.dense` is green, `mlp.dense_h_to_4h` is blue, and `mlp.dense_4h_to_h` is orange. To pack all the squares tight into the rectangles we used a python package called `rectangle-packer`.

variety of different parallel configurations up to 3D parallelism. Figure 6.5 shows the different throughput and maximum memory allocation of the different parallel configurations. Pura DP has the lead of having the highest throughput and maximum memory consumption for both 3D-Shampoo and SGD. On the opposite, pure OP is the slowest and takes the least amount of memory. The only reason pure DP outperforms every other configuration is that the GPT-2 model is big enough to fit on one GPU. If the model is bigger, other configurations have to be considered. The throughput of SGD is always higher or at least the height of 3D-Shampoo. The same pattern occurs with the maximum memory consumption; 3D-Shampoo is on average 700MB higher than SGD. In my calculation stated in the figure description it should be around 312.5MB, but this higher deviation could be of some more allocation happening due to the p-root and inverse computation. In the end, the figure should demonstrate that 3D-Shampoo did not have as much of an impact on performance compared to SGD.

Figure 6.6 is the exact same measurements of fig. 6.5 but rewritten in runtime per training step. This shows how 3D-Shampoo is slower than SGD, but for the configuration $N_{DP} = 1$, $N_{PP} = 2$, and $N_{OP} = 4$, 3D-Shampoo and SGD have the same runtime performance. For certain parallel configurations, the optimizer can be fully covered by the communication part of DeepSpeed.

But if we set the micro-batch size $|b|$ and batch size $|\mathcal{B}|$ in a way such that PP does only deal with one micro-batch, the runtime of the different parallel configurations is the same at around $\sim 1500ms$ like shown in fig. 6.7. Here again, the p-root inverse algorithm is fixed to 20 iterations, and each GPU has to deal with the same shaped 64 preconditioning matrices. Normally, one would not only use one micro-batch to train a model using PP.

We also made a case where we only changed to OP level from 1 to 16 to see how 3D-Shampoo against SGD the throughput and maximum memory allocation changes by the level of OP. We visualized it in fig. 6.8. One can see that the throughput for both 3D-Shampoo and SGD is increasing pretty much the same. The maximum memory allocation on the other hand, at $N_{OP} = 1$, 3D-Shampoo is $\sim 900MB$ higher than SGD but steadily decreases to $\sim 100MB$ higher than SGD. This was expected. The higher the OP, the smaller the layers are getting, and hence the preconditioning matrices are getting smaller as well.

A more interesting comparison is when we only change the DP level. This is where 3D-Shampoo should shine against any type of optimizer when compared to weak-scaling efficiency. As discussed in section 6.5, 3D-Shampoo will distribute the workload by the level of DP to maximize its efficiency and hence throughput of the model. We plotted the throughput and maximum memory of 3D-Shampoo and SGD as well as the weak-scaling speedup of both of them in fig. 6.9. This time, the GPT-2 consists of 16 transformer decoder layers such that when $N_{DP} = 16$, each GPU has to deal with one layer to precondition. One can see that both 3D-Shampoo and SGD are increasing in throughput by increasing the level of DP. But the maximum memory allocation of SGD stays constant at $7.3GB$. 3D-Shampoo has a very big memory consumption when $N_{DP} = 1$ but decreases linearly up to a point of having only $\sim 500MB$ more than SGD at $N_{DP} = 16$. The top figure is not really noticeable, but 3D-Shampoo scales in throughput better than SGD. That is why we created the weak-scaling plot to show the scaling of 3D-Shampoo compared to SGD. One can also expect for certain models super linear scaling for 3D-Shampoo which we already have demonstrated for distr. Shampoo on DenseNet121 shown in fig. 5.8. From these two figures, one can conclude that 3D-Shampoo is always performing better when increasing the number of GPUs.

We also wanted to conduct tests on how the BlockPartition function performs

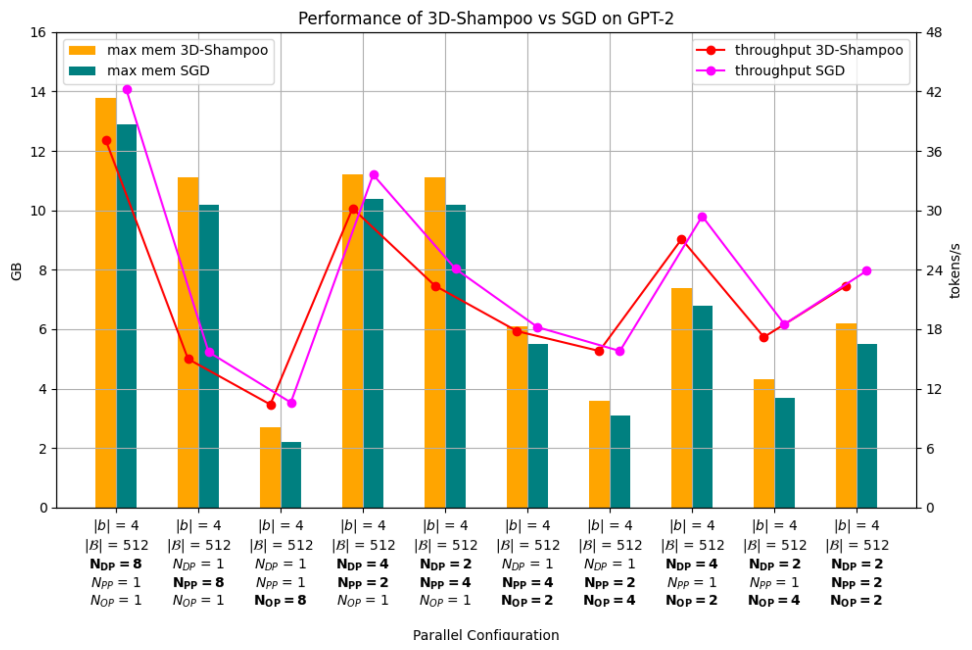


Figure 6.5: Figure of the throughput and maximum measured memory allocation of 3D-Shampoo and SGD on GPT-2 on 8 nodes (8 GPUs) on Piz Daint with different combinations of parallelism. The left y-axis is the maximum measured memory $[GB]$ occupation, and the right y-axis is the throughput in $[tokens/s]$. The x-axis indicates the type of parallelism and batch sizes. N_{XX} indicates the number of parallelisms in the given dimension XX, where $XX \in \{DP, PP, OP\}$. Bold font indicates the level of parallelism with up to 3D when all three are bold. DP stands for (ZeRO-style) Data-Parallelism. Note that in all measurements, we deactivate any ZeRO optimization due to actually being able to update the preconditioning matrices of Shampoo over multiple training steps. PP stands for Pipeline Parallelism and OP for Operator/Tensor Parallelism (For Megatron-LM, it is called Model Parallelism). Note that $N = N_{DP}N_{PP}N_{OP} = 8$ is applied for all the combinations of parallelism. $|b|$ is the micro-batch size and $|\mathcal{B}|$ is the batch size. The number of micro-batches for the PP is defined as $\#|b| = |\mathcal{B}|/(N_{DP}|b|)$. Depending on the level of DP (i.e. $N_{DP} > 1$), we have micro-batches trained on the model. We used the GPT-2 model from Nvidia's Megatron-LM with the hyperparameters: `--num-layers=8`, `--hidden-size=1024`, `--num-attention-heads=16`, `--seq-length=1024`, `--max-position-embeddings=1024`, `--fp32`, `--seed=42`, and `--lr=1.5e-4`. Even though the tokens were uniformly random in the range $[0, 5000]$, we still set a reasonable learning rate to simulate the optimization step. We averaged over 10 training steps with 3 warmup steps before. Depending on the level of parallelism in any dimension, Megatron-LM will build the model accordingly (see fig. 6.2). We used `--num-layers=8` such that we can achieve a balanced workload for all different kinds of parallelisms. The vocabulary size is 50304, which makes the embedding layer of GPT-2 very big. We ignore the embedding layer by not computing its preconditioning matrices. Still, the gradients of the embedding layer are updated with a momentum-based optimizer included by the *Anil et al.* Shampoo implementation [5]. In 3D-Shampoo, we fixed the number of iterations of the p-root inverse algorithm to 20 for a fair comparison between all the different parallel configurations. The same goes for the number and shapes of the preconditioning matrices which is 64 per GPU which are: 8×384^2, 8×128^2, 16×512^2, and 32×1024^2. This makes 3D-Shampoo at least 312.5MB bigger in memory usage compared to pure SGD due to the preconditioning matrices. "max mem" states the measured maximum allocated memory on a single GPU. For certain configuration of parallelism, we did not achieve maximum allocated memory on the NVIDIA® Tesla® P100 GPU which have 16GB of memory.

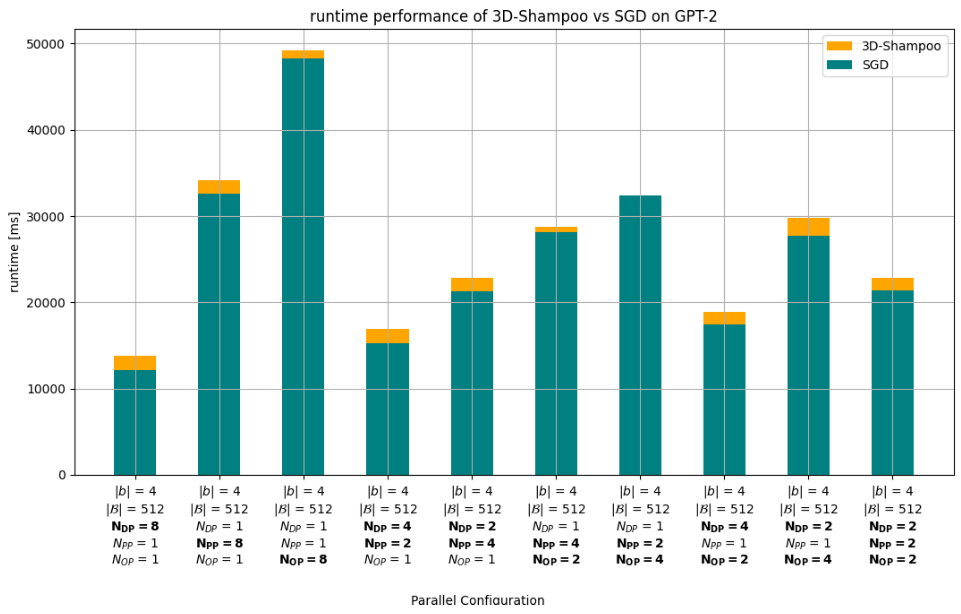


Figure 6.6: Figure of the runtime of one training step of 3D-Shampoo and SGD on GPT-2 on nodes (8 GPUs) on Piz Daint with different combinations of parallelism with the exact same settings as fig. 6.5. Runtime is shown in milliseconds and was measured over multiple training steps and averaged. $|b|$ is the micro-batch size and $|\mathcal{B}|$ is the batch size. N_{XX} indicates the number of parallelisms in the given dimension XX, where $XX \in \{DP, PP, OP\}$. Bold font indicates the level of parallelism with up to 3D when all three are bold.

for different shapes for GPT-2. Figure 6.10 shows our result of GPT-2 with one transformer decoder layer run on one GPU for different block sizes of the BlockPartition function. One can see a sweat spot of block shapes for the transformer and decoder layer which is at 1024. At this shape, there will be 1024 precondition matrices of this shape for one transformer decoder layer. Decreasing the block partitioning shapes further will also increase the number of preconditioning matrices (indicated in "precs"). Smaller preconditioning matrices are beneficial for the p-root inverse algorithm due to its high intensity of matrix-matrix operations, but having too many of those will make the whole 3D-Shampoo algorithm slower due to a lot of memory movement. We made a visual representation of all the preconditioning matrices with the different shapes of block partition in fig. 6.4. We measured the number of flops and runtime of computing the p-root inverse of one transformer decoder layer of GPT-2 for the different block shapes shown in table 6.5. We used Nvidia's `nvprof` to measure the flops. One can see when we use the original preconditioning shapes; we reach the highest flops per second performance at 7813 GigaFLOPS which is 84% of the theoretical peak performance of a Tesla P100 GPU (9.3 TeraFLOPS). We also stated the theoretical flops we got from our computational cost function from eq. (4.4)

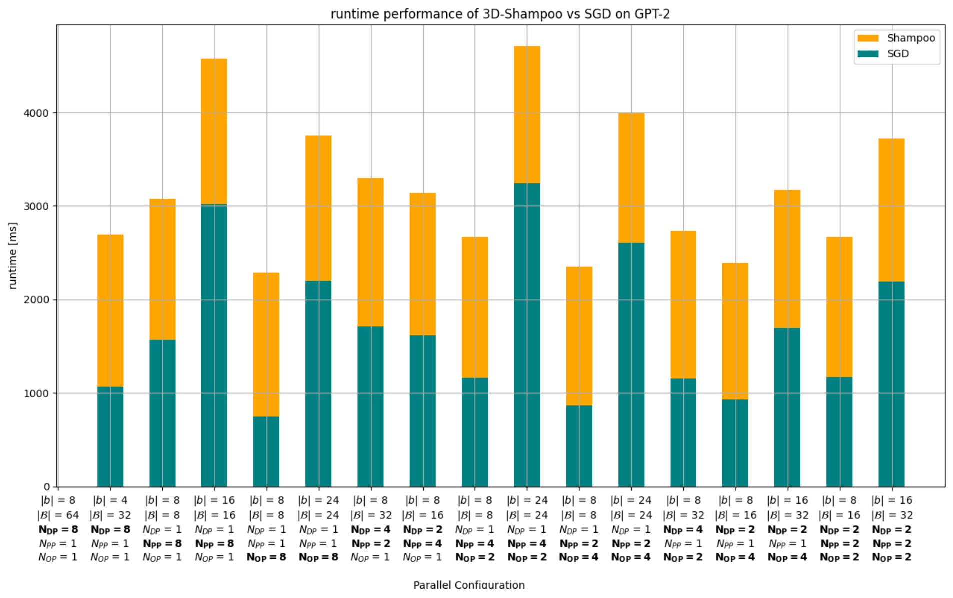


Figure 6.7: Figure of the runtime of one training step of 3D-Shampoo and SGD on GPT-2 on nodes (8 GPUs) on Piz Daint with different combinations of parallelism with the exact same settings as fig. 6.5 but with different micro-batch sizes $|b|$ and batch sizes $|\mathcal{B}|$. Here the PP only deals with one micro-batch instead of multiple micro-batches $(\#|b| = |\mathcal{B}|/(N_{DP}|b|) = 1)$. Hence, the runtime of 3D-Shampoo is always $\sim 1500ms$ slower than SGD runtime for any parallel configuration. Runtime is shown in milliseconds, measured over multiple training steps, and averaged. $|b|$ is the micro-batch size and $|\mathcal{B}|$ is the batch size. N_{XX} indicates the number of parallelisms in the given dimension XX, where $XX \in \{DP, PP, OP\}$. Bold font indicates the level of parallelism with up to 3D when all three are bold.

to show that our approximated theoretical flops are close to the measured one.

Lastly, we measured the performance of changing the number of fixed iterations for the p-root inverse algorithm with different block partitioning shapes shown in fig. 6.11. Normally, the number of iterations of the p-root algorithm is not fixed but capped at 100 iterations. It is noted that the expected runtime of this algorithm is roughly in the range of 10-30 iterations depending on how big and well-conditioned the preconditioning matrices are. But in fig. 6.4, we fixed the iteration to get a feeling of how it performs for different settings. For any block partitioning shapes, more iteration in the p-root inverse algorithm always makes the throughput smaller. The block partitioning shape of 1024 is always the fastest as we already have seen in fig. 6.10. From this figure, we want to show the impact of the number of iterations by the Shampoo algorithm. It is a major computational time when it comes to one training step.

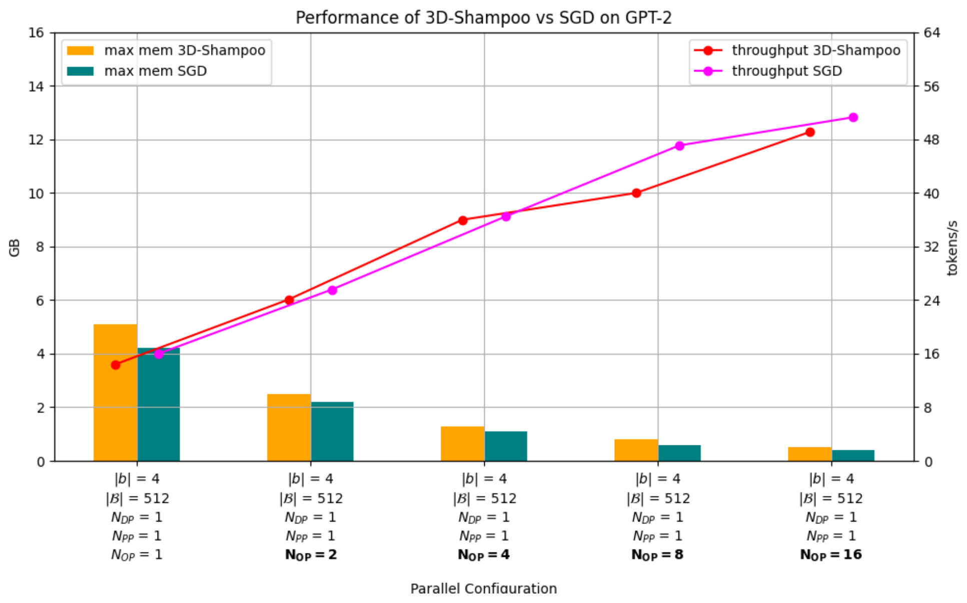
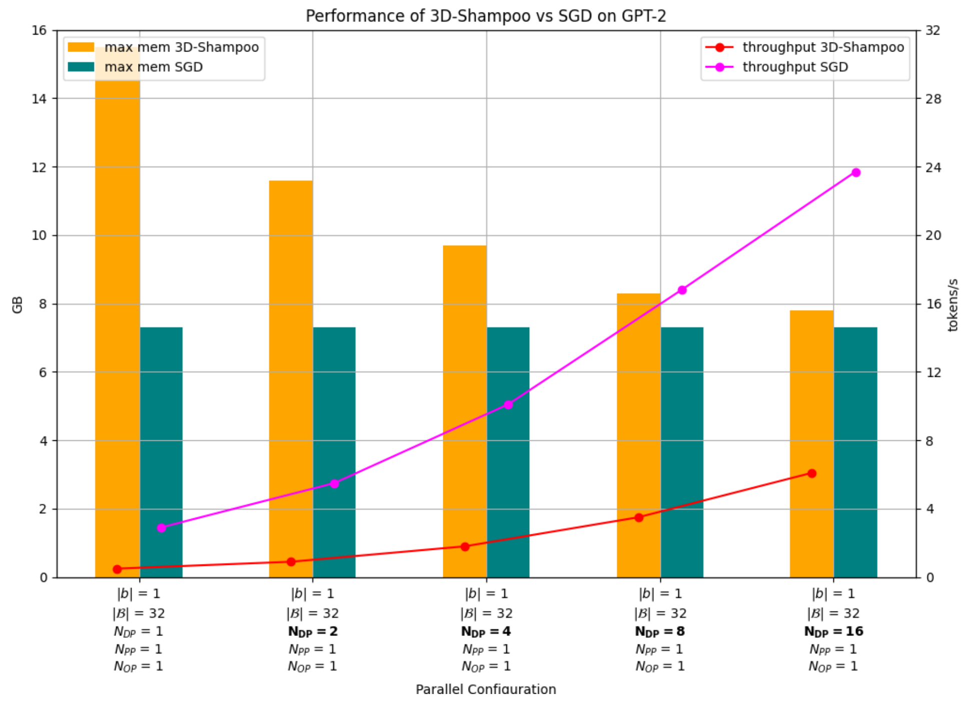


Figure 6.8: Figure of the throughput and maximum measured memory allocation of 3D-Shampoo and SGD on GPT-2 on Piz Daint with different levels of Operator Parallelism (OP) with the exact same settings as fig. 6.5 but changing the `--num-layers` from 8 to 1. The left y-axis is the memory allocation in Gigabytes, and the right y-axis is the throughput in the number of tokens per second. $|b|$ is the micro-batch size and $|\mathcal{B}|$ is the batch size. N_{XX} indicates the number of parallelisms in the given dimension XX, where $XX \in \{DP, PP, OP\}$. $N = N_{DP}N_{PP}N_{OP}$ is the number of GPUs in parallel training. Hence, the most left plot is single GPU performance, and the most right one is 16 GPUs in parallel.

Block sizes	#precs	theoretical flops	measured flops	avg runtime	performance	TPP%
Original	8	13746 Gflops	27510 Gflops	3.521s	7813 GFLOPS	84%
2048	14	4210 Gflops	8432 Gflops	1.324s	6369 GFLOPS	68%
1024	24	2063 Gflops	4136 Gflops	0.890s	4647 GFLOPS	50%
512	96	1032 Gflops	2102 Gflops	0.977s	2151 GFLOPS	23%
256	384	516 Gflops	1071 Gflops	3.004s	356 GFLOPS	4%
128	1536	259 Gflops	555 Gflops	12.060s	46 GFLOPS	0.5%
64	6144	130 Gflops	298 Gflops	48.416s	6 GFLOPS	0.06%

Table 6.5: Table of the theoretical and measured flops of computing the p-root and inverse of one transformer decoder layer of GPT-2 for different block partitioning shapes. The theoretical flops come from eq. (4.4) and the measured one from Nvidia's `nvprof`. "TPP%" stands for how close the measured "performance" is to the theoretical peak performance in per cent. "flops" stands for the number of floating point operations (single precision) and "FLOPS" flops per second. The theoretical peak performance of a Tesla P100 GPU is 9340 GFLOPS which makes the performance for "Original" shaped preconditioning matrices 84%. We used the exact same settings as fig. 6.5 for the GPT-2 model but changed the `--num-layers` from 8 to 1. The p-root inverse algorithm was fixed to `max_iter` = `num_iter` = 20 which means for while-loop breaking for any preconditioning matrices. The shapes of the preconditioning matrices of one transformer decoder layer for a different level of block partitioning are visualized in fig. 6.4.



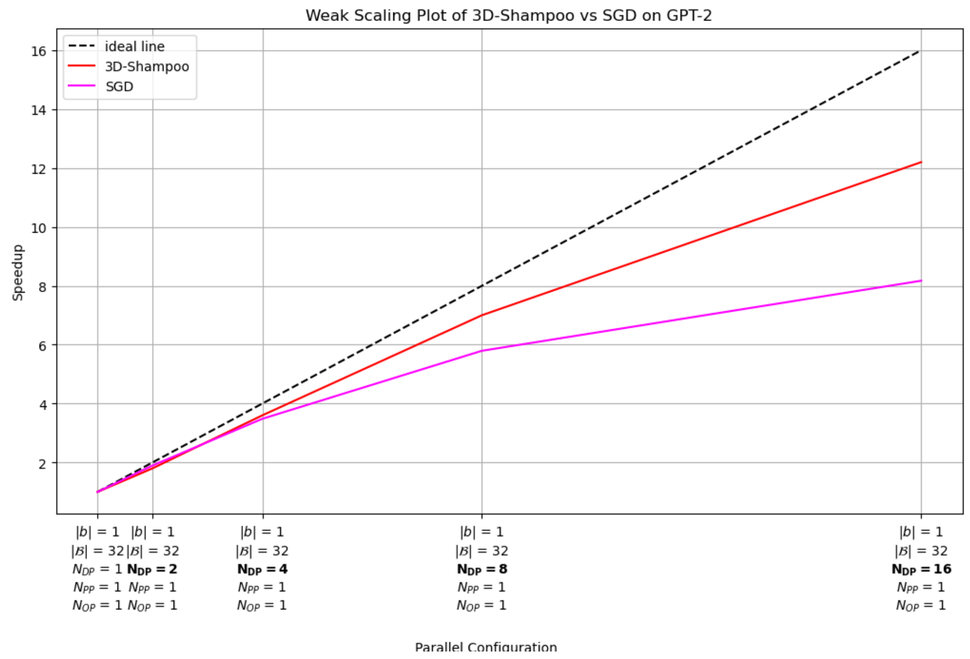


Figure 6.9: The top figure shows the throughput and maximum measured memory allocation of 3D-Shampoo and SGD on GPT-2 on Piz Daint with different levels of Data Parallelism (DP) with the exact same settings as fig. 6.5 but changing the `--num-layers` from 8 to 16. The bottom plot shows the weak-scaling speedup of the top figure. The left y-axis is the memory allocation in Gigabytes, and the right y-axis is the throughput in the number of tokens per second. $|b|$ is the micro-batch size and $|\mathcal{B}|$ is the batch size. N_{XX} indicates the number of parallelisms in the given dimension XX, where $XX \in \{DP, PP, OP\}$. $N = N_{DP}N_{PP}N_{OP}$ is the number of GPUs in parallel training. Hence, the most left plot is single GPU performance, and the most right one is 16 GPUs in parallel.

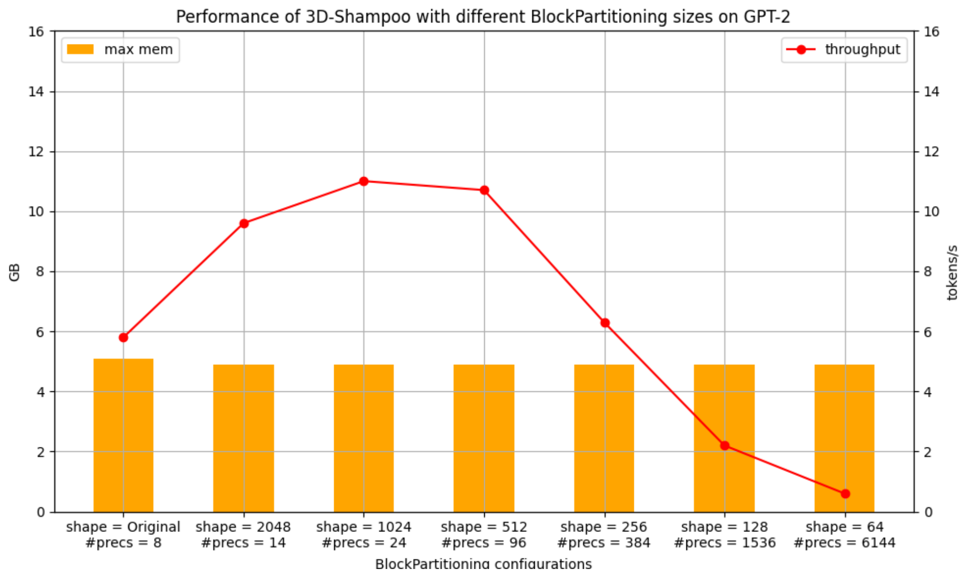


Figure 6.10: Figure of the throughput and maximum measured memory allocation of 3D-Shampoo on GPT-2 on Piz Daint with different settings of BlockPartitioning shapes with the exact same settings as fig. 6.5 but changing the `--num-layers` from 8 to 1. For all the different BlockPartitioning sizes, we fixed the number of iterations of the p-root and inverse algorithm to 20 iterations. We used micro-batch size $|b| = 4$ and batch size $|\mathcal{B}| = 32$. No parallelization is happening in this figure. Everything ran on a single GPU. "shape" indicates the level of BlockPartitioning level with "Original" meaning no BlockPartitioning with the original shapes of preconditioning matrices of one transformer decoder layer. "precs" shows the number of preconditioning matrices that have to be computed and used to precondition the gradients. The shapes of all the preconditioning matrices are shown in fig. 6.4.

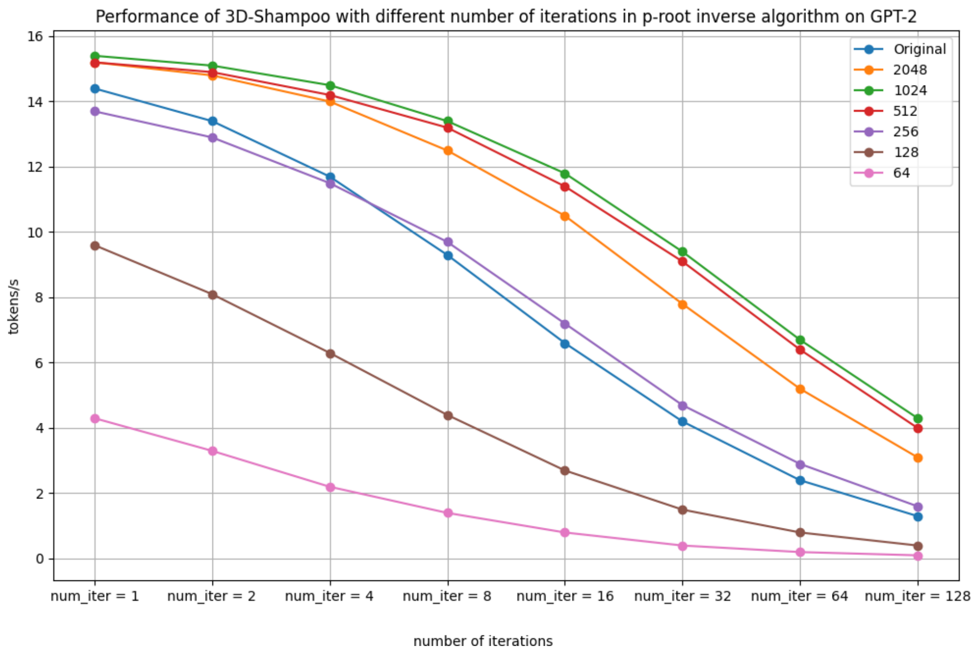


Figure 6.11: Figure of the throughput depending on the number of fixed iterations used to compute the p-root and inverse of the preconditioning matrices in 3D-Shampoo for on for different sizes of block partitioned preconditioning matrices. We used the same settings for GPT-2 as in fig. 6.5 but `--num-layers` from 8 to 1. We used micro-batch size $|b| = 4$ and batch size $|\mathcal{B}| = 32$. No parallelization is happening in this figure. Everything ran on a single GPU. "shape" indicates the level of BlockPartitioning level with "Original" meaning no BlockPartitioning with the original shapes of preconditioning matrices of one transformer decoder layer. Each line in the figure represents a different shape level of BlockPartitioning. The shapes of the block partitioned preconditioning matrices are visualized for one transformed decoder layered GPT-2 in fig. 6.4. Normally the p-root and inverse algorithm is not fixed by a number of iterations but capped at 100 iterations. It will end earlier when the error is small enough. See algorithm 4 for the pseudo-code of the p-root and inverse algorithm.

Chapter 7

Conclusion

In this thesis, our goals are to understand the strengths and weaknesses of the distributed preconditioning methods for training deep neural networks with parallel devices and identify promising directions to scale preconditioning up to today's largest-scale networks. To this end, we first conducted a literature review of the distributed preconditioning methods used in training deep neural networks. We then targeted the most popular methods, i.e., distributed (distr.) K-FAC and distr. Shampoo. We implemented them in PyTorch and analysed the performance on different numbers of GPUs and sizes of deep neural networks. Furthermore, based on our analysis, we introduced a new optimizer, *3D-Shampoo*, an extension of Shampoo for 3D-parallel (i.e., data-, operator-, and pipeline-) training. We observed that 3D-Shampoo achieved competitive throughput and memory usage with SGD when run on a multi-GPU environment for different sizes of GPT-2-like Transformer models and parallelism settings.

As for future work, it would be interesting to investigate the convergence speed and scaling of 3D-Shampoo in training large models (e.g., Transformers) on real-world tasks (e.g., language modelling) and for a higher number of GPUs.

Appendix A

Appendix

Algorithm 6 The PowerIter function to compute the maximum eigenvalue from algorithm 4

```
Require: $H \leftarrow H_t^i$; $\texttt{max\_iter} = 100$; $\texttt{tolerance} = 1e-6$
  $v \sim \mathcal{U}(-1,1) \in \mathbb{R}^{n_i}$
  $\texttt{error} = 1$
  $\texttt{iters} = 0$
  $\texttt{singular\_val} = 0$
  while $\texttt{error} > \texttt{tolerance}$ and $\texttt{iters} < \texttt{max\_iter}$ do
    $v = v / \|v\|_2$
    $\tilde{v} = Hv$
    $\texttt{sing\_}v = v^T\tilde{v}$
    $\texttt{error} = |\texttt{sing\_}v - \texttt{singular\_val}|$
    $v = \tilde{v}$
    $\texttt{singular\_val} = \texttt{sing\_}v$
    $\texttt{iters} = \texttt{iters} + 1$
  end while
  return $\texttt{singular\_val}$
```

Listing A.1: model partitioning function of shampoo

```python
def get_distr_prec_partition(self):
    """
    Distributes the workload by computational cost of each layer for
        ↪ total number of GPUs
    e.g.
    1 GPU for ResNet18:
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    3 GPUs for ResNet18:
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 2, 2]
    8 GPUs for ResNet18:
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 2, 3, 4, 5, 6, 7]
    21 or more GPUs for ResNet18:
```

```
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
    2 GPUs for 3 layers MLP (if first layer is bigger than 2nd and 3rd):
    [0,1,1]
    """

    total_comp_cost = 0
    comp_cost_layers = []
    shapes_list = []
    for p in self.model.parameters():
        if p.ndim > 1 and p.requires_grad:
            _transformed_shape = _merge_small_dims(p.shape, self.block_size)
            _partitioner = BlockPartitioner(_transformed_shape, self.block_size)
            shapes = _partitioner.kronecker_factor_shapes()

            shapes_list.append(_transformed_shape)
            comp_cost = self.computational_cost(shapes)
            total_comp_cost += comp_cost
            comp_cost_layers.append(comp_cost)

    num_layers = len(comp_cost_layers)

    partitions = [0]*num_layers
    if self.world_size == 1:
        return [], partitions
    elif num_layers > self.world_size:
        split_list = np.array([0])

        for rank in range(self.world_size-1):
            if rank == 0:
                split_list = np.append(split_list, self.next_split(comp_cost_layers))
            else:
                sub_sums = []
                for i in range(1, len(split_list)):

                    local_comp_cost = np.sum(comp_cost_layers[split_list[i-1]:split_list[i]])
                    sub_sums.append(local_comp_cost)

                    if i == len(split_list) - 1:
                        local_comp_cost = np.sum(comp_cost_layers[split_list[i]:])
                        sub_sums.append(local_comp_cost)

                while(True):
                    i = np.argmax(sub_sums)
                    if i == len(sub_sums) - 1:
                        sub_comp_cost_layers = comp_cost_layers[split_list[i]:]
                        shift = split_list[i]
                    else:
                        sub_comp_cost_layers = comp_cost_layers[split_list[i]:split_list[i+1]]
                        shift = split_list[i]

                    if len(sub_comp_cost_layers) > 1:
                        break
                    else:
                        sub_sums[i] = -1
```

```python


                    split_list = np.append(split_list, self.next_split(sub_comp_cost_layers) + shift)
                    split_list = np.sort(split_list)

            sub_sums = []
            for i in range(1, len(split_list)):

                local_comp_cost = np.sum(comp_cost_layers[split_list[i-1]:split_list[i]])
                sub_sums.append(local_comp_cost)

                if i == len(split_list) - 1:
                    local_comp_cost = np.sum(comp_cost_layers[split_list[i]:])
                    sub_sums.append(local_comp_cost)

            #if self.world_rank == 0:
            #    print(sub_sums, "\n")

            next_split = split_list[1]
            rank = 0
            for i in range(len(partitions)):
                if i == next_split:
                    rank += 1
                    if rank != self.world_size - 1:
                        next_split = split_list[rank+1]

                partitions[i] = rank
            return split_list[1:], partitions
        else: #atm, we do not support multiple gpus for one layer
            rank = 0
            for i in range(num_layers):
                partitions[i] = i

            return partitions[1:], partitions


    def computational_cost(self, shapes):
        """
        input: shape: [[x, x],[y, y],...] (Blockpartitioner.kronecker_factor_shape)
        output: returns the compuational cost of this Blockpartitioned layers
        """
        tmp_cost = 0
        for shape in shapes:
            assert len(shape) == 2
            assert shape[0] == shape[1]

            tmp_cost += shape[0]**0.4 # ATM simple O(n^3) assumption (maybe even less 0.4)

        return tmp_cost

    def next_split(self, subset_partitions):
        """
        deciding where the next split is happening

        input: subset_partitions: [] is a subset of comp_cost_layers
        output: index where to split (int)
    """
    assert len(subset_partitions) > 1

    x = np.array(subset_partitions)
    y = np.sum(subset_partitions)/2

    split_loc = len(x[np.cumsum(x) < y])

    split_loc += 1

    return split_loc
```

Bibliography

[1] Dragon ball-z - sangoku kamehameha pixel art. Clipartmax.com. Accessed: 17.4.2023.

[2] Saurabh Adya, Vinay Palakkode, and Oncel Tuzel. Nonlinear conjugate gradients for scaling synchronous distributed dnn training. *arXiv preprint arXiv:1812.02886*, 2018.

[3] Sadaf R Alam, Ladina Gilly, Colin J McMurtrie, and Thomas C Schulthess. Cscs and the piz daint system. In *Contemporary High Performance Computing*, pages 149–173. CRC Press, 2019.

[4] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray xc series network. *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[5] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

[6] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM review*, 60(2):223–311, 2018.

[7] Thomas Bradley. Gpu performance analysis and optimisation. *NVIDIA Corporation*, 2012.

[8] Mengyun Chen, Kaixin Gao, Xiaolei Liu, Zidong Wang, Ningxi Ni, Qian Zhang, Lei Chen, Chao Ding, Zhenghai Huang, Min Wang, et al. Thor, trace-based hardware-driven layer-oriented natural gradient descent computation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7046–7054, 2021.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani,

Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[11] Celestine Dünner, Aurelien Lucchi, Matilde Gargiani, An Bian, Thomas Hofmann, and Martin Jaggi. A distributed second-order algorithm you can trust. In *International Conference on Machine Learning*, pages 1358–1366. PMLR, 2018.

[12] Chih-Hao Fang, Sudhir B Kylasa, Fred Roosta, Michael W Mahoney, and Ananth Grama. Newton-admm: A distributed gpu-accelerated optimizer for multiclass classification problems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

[13] Elias Frantar, Eldar Kurtic, and Dan Alistarh. M-fac: Efficient matrix-free approximations of second-order information. *Advances in Neural Information Processing Systems*, 34:14873–14886, 2021.

[14] Chun-Hua Guo and Nicholas John Higham. A schur–newton method for the matrix pth root and its inverse. 2005.

[15] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

[16] Adnan Haider, Chao Zhang, Florian L Kreyssig, and Philip C Woodland. A distributed optimisation framework combining natural gradient with hessian-free for discriminative sequence training. *Neural Networks*, 143:537–549, 2021.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[18] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[19] Rustem Islamov, Xun Qian, and Peter Richtárik. Distributed second order methods with fast rates and compressed communication. In *International conference on machine learning*, pages 4617–4628. PMLR, 2021.

[20] Xi-Lin Li. Preconditioned stochastic gradient descent. *IEEE transactions on neural networks and learning systems*, 29(5):1454–1466, 2017.

[21] Yichuan Li, Nikolaos M Freris, Petros Voulgaris, and Dušan Stipanović. Dn-admm: Distributed newton admm for multi-agent optimization. In *2021 60th IEEE Conference on Decision and Control (CDC)*, pages 3343–3348. IEEE, 2021.

[22] Jie Liu, Yu Rong, Martin Takáč, and Junzhou Huang. Accelerating distributed stochastic l-bfgs by sampled 2nd order information. *Beyond First Order Methods in ML@ NeurIPS*, 2019.

[23] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[24] Baorun Mu, Saeed Soori, Bugra Can, Mert Gürbüzbalaban, and Maryam Mehri Dehnavi. Hylo: a hybrid low-rank natural gradient descent method. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2022.

[25] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 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*, pages 1–15, 2021.

[26] NVIDIA Corporation. NVIDIA Apex. `https://github.com/NVIDIA/apex`, 2019. A PyTorch Extension: Tools for easy mixed precision and distributed training in PyTorch.

[27] Kazuki Osawa, Satoki Ishikawa, Rio Yokota, Shigang Li, and Torsten Hoefler. Asdl: A unified interface for gradient preconditioning in pytorch, 2023.

[28] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Chuan-Sheng Foo, and Rio Yokota. Scalable and practical natural gradient for large-scale deep learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[29] J Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. Kaisa: an adaptive second-order optimizer framework for deep neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[30] J Gregory Pauloski, Zhao Zhang, Lei Huang, Weijia Xu, and Ian T Foster. Convolutional neural network training with distributed k-fac. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

[31] Mary Phuong and Marcus Hutter. Formal algorithms for transformers. *arXiv preprint arXiv:2207.09238*, 2022.

[32] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 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*, pages 3505–3506, 2020.

[33] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

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

[35] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[36] Sidak Pal Singh and Dan Alistarh. Woodfisher: Efficient second-order approximation for neural network compression. *Advances in Neural Information Processing Systems*, 33:18098–18109, 2020.

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

[38] Zedong Tang, Fenlong Jiang, Maoguo Gong, Hao Li, Yue Wu, Fan Yu, Zidong Wang, and Min Wang. Skfac: Training neural networks with faster kronecker-factored approximate curvature. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13479–13487, 2021.

[39] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[40] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, et al. Mlp-mixer: An all-mlp architecture for vision. *Advances in neural information processing systems*, 34:24261–24272, 2021.

[41] Shusen Wang, Fred Roosta, Peng Xu, and Michael W Mahoney. Giant: Globally improved approximate newton method for distributed optimization. *Advances in Neural Information Processing Systems*, 31, 2018.

[42] Minghan Yang, Dong Xu, Zaiwen Wen, Mengyun Chen, and Pengxiang Xu. Sketch-based empirical natural gradient methods for deep learning. *Journal of Scientific Computing*, 92(3):94, 2022.

[43] Zhewei Yao, Amir Gholami, Daiyaan Arfeen, Richard Liaw, Joseph Gonzalez, Kurt Keutzer, and Michael Mahoney. Large batch size training of neural networks with adversarial training and second-order information. *arXiv preprint arXiv:1810.01021*, 2018.

[44] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2017.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Distributed Gradient preconditioning for Training Large-Scale Models

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Baumann	Noah

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Sissach 2. 5. 2025	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.