Survey on Large Scale Neural Network Training

Julia Gusak[1,*], Daria Cherniuk[1,*], Alena Shilova[2,*], Alexander Katrutsa[1], Daniel Bershatsky[1], Xunyi Zhao[3], Lionel Eyraud-Dubois[3], Oleg Shlyazhko[5], Denis Dimitrov[4], Ivan Oseledets[1], Olivier Beaumont[3]

[1]Skolkovo Institute of Science and Technology, Russia
[2]Inria, University of Lille - CRIStAL, France
[3]Inria, University of Bordeaux, France
[4]AIRI, Russia
[5]Independent researcher, Russia
[*]equal contribution
{y.gusak, daria.cherniuk, a.katrutsa, d.bershatsky2, i.oseledets}@skoltech.ru,
{alena.shilova, xunyi.zhao, lionel.eyraud-dubois, olivier.beaumont}@inria.fr,
olehshliazhko@gmail.com,
den.dimitrov@gmail.com

Abstract

Modern Deep Neural Networks (DNNs) require significant memory to store weight, activations, and other intermediate tensors during training. Hence, many models don't fit one GPU device or can be trained using only a small per-GPU batch size. This survey provides a systematic overview of the approaches that enable more efficient DNNs training. We analyze techniques that save memory and make good use of computation and communication resources on architectures with a single or several GPUs. We summarize the main categories of strategies and compare strategies within and across categories. Along with approaches proposed in the literature, we discuss available implementations.

1 Introduction

Modern trends in the development of Deep Learning (DL) and Artificial Intelligence (AI) technologies involve the use of Deep Neural Networks (DNNs) to solve various problems of image, video, audio, natural language processing, content generation in the form of images or text in a given style and subject, etc.

The question we address in this survey is the following: given your model (which you do not want to rewrite) and your computation platform (which you do not want to change), what are the generic approaches that can allow you to perform the training efficiently? For training to be efficient, it must be feasible (the data must fit in memory), it must exploit the computational power of the resources well (the arithmetic intensity of the operations must be sufficient) and, in the parallel case, it must not be limited by too large data exchanges between the nodes. The efficiency of the training depends fundamentally on the efficient implementation of computational kernels on the computational resources (CPU, TPU, GPU) and on the efficient implementation of communications between GPUs and between different memories. In both cases, a lot of work has been done on the optimization of the arithmetic intensity of the computational kernels and on the efficient realization of collective communication operations over the hardware network. For the user, powerful profiling tools have been developed to identify hardware bottlenecks and can be used to decide which strategies described in this survey can be used to solve the problems of arithmetic intensity , memory and by controlling the volume of data exchanged.

The present survey covers generic techniques to cope with these limitations. If the computation cannot be performed a priori because the model, the optimizer states, and the activations do not fit in memory, there are techniques to trade memory for computation (re-materialization) or for data movements (activation and weight offloading), and it is also possible to compress the memory footprint by approximating optimizer states and gradients (compression, pruning, quantization). The use of parallelism (data parallelism, model parallelism, pipelined model parallelism) can also make it possible to distribute the memory requirements over several resources. If the arithmetic intensity of the computations is not sufficient to fully exploit the GPUs and TPUs, it is generally because the size of the mini-batch is too small, and then the above techniques can also enable to increase the size of the mini-batch. Finally, if the communications, typically induced by the use of data parallelism, are too expensive and slow down the computation, then other forms of parallelism can be used (model parallelism, pipelined model parallelism) and the compression of the gradients can allow to limit the volumes of exchanged data.

In this survey, we explain how these different techniques work, we describe the literature to evaluate and compare the proposed approaches and we also analyze the frameworks that allow to implement these techniques (almost) transpar-

ently. The different techniques that we consider, and their influence on communications, memory and computing efficiency are depicted in Table 1.

According to our taxonomy, we distinguish the following methods based on their purpose: reducing the memory footprint on a GPU is discussed in Section 2, the use of parallel training for models that do not fit on a GPU is considered in Section 3, and the design of optimizers developed to train models stored on multiple devices is addressed in Section 4.

2 Memory Usage Reduction on a Single GPU

During the forward pass, neural networks store the activations necessary to perform backpropagation. In some cases, these activations can consume a substantial amount of memory, making training infeasible. There are two main approaches to reducing that memory footprint: rematerialization (also called checkpointing) and offloading. Let us introduce the following notations: F_i/B_i is a computation of forward/backward pass on module i; F_i^n is an operation that computes the output of F_i and forgets its input; F_i^c computes the output of F_i and keeps the input; F_i^e computes the output of F_i, keeps the input, and all intermediate activations needed to later compute B_i. These different operations can easily be implemented in frameworks like PyTorch or Tensorflow.

2.1 Rematerialization of Activations

Rematerialization is a strategy that only stores a fraction of the activations during the forward pass and recomputes the rest during the backward pass. Rematerialization methods can be distinguished by what computational graphs they are dealing with. The first group comes from Automatic Differentiation (AD), they find optimal schedules for homogeneous sequential networks (DNNs whose layers are executed sequentially and have the same computational and memory costs). The second group concentrates on transition model such as heterogeneous sequential networks (DNNs may be any sequential neural networks consisting of arbitrarily complex modules, e.g. CNNs, ResNet, some transformers), which adjust solutions from AD to heterogeneous settings. The final group concentrates on general graphs, but this problem becomes NP-complete in the strong sense and thus can be solved optimally only with ILP, otherwise approximately with various heuristics.

For homogeneous sequential networks, the binomial approach was proven to be optimal in [Grimm et al., 1996] and implemented in REVOLVE [Griewank and Walther, 2000]. Compiler-level techniques have been proposed in [Siskind and Pearlmutter, 2018] to make it applicable to fully arbitrary programs. This results in a divide-and-conquer strategy, which however assumes that computations can be interrupted at arbitrary points, making it unsuitable for GPU computations. In the case of heterogeneous computation times and homogeneous memory costs the optimal strategy can be found with dynamic programming [Walther and Griewank, 2008]. A direct adaptation of results for homogeneous chains was applied to RNNs in [Gruslys et al., 2016].

[Beaumont et al., 2019] considered the most general case of heterogeneous sequential DNNs. This paper proposed a new modeling of the problem directly inspired by the data dependencies induced by the PyTorch framework. A dynamic programming approach is used that finds, for linear or linearized chains, the optimal strategy.

Some methods can perform rematerialization for general graphs, though the exact approaches are exponentially expensive (see Table 2). For example, [Jain et al., 2020b] proposed an Integer Linear Program (ILP) to find the optimal rematerialization strategy suitable for an arbitrary Directed Acyclic Graph (DAG) structure. [Kirisame et al., 2020] presented a cheap dynamic heuristic DTR that relies on scores encouraging to discard (i) heavy tensors (ii) with a long lifetime and (iii) that can be easily recomputed.

Support in popular open-source frameworks. Machine learning framework PyTorch [1] provides two options for checkpointing: the user explicitly defines which activations to store or uses a periodic strategy based on [Chen et al., 2016].Checkmate[2], written in TensorFlow[3], accepts user-defined models expressed via the high-level Keras interface. Framework Rotor[4] allows the algorithm from [Beaumont et al., 2019] to be used with any PyTorch DNN implemented with the `nn.Sequential` container.

2.2 Offloading of Activations

Offloading (also called *Memory Swapping*) is a technique that saves GPU memory by offloading activations to CPU memory during forward pass and prefetching them back into GPU memory for the corresponding backward computation.

Due to the limited bandwidth of the PCI bus between the CPU and the GPU, the choice of which activations to transfer and when to do it must be optimized. Authors of vDNN [Rhu et al., 2016] followed a heuristic effective for CNNs by offloading only the inputs of convolutional layers, though it does not generalize well to general DNNs. [Le et al., 2018] considered the activations life-cycle to choose the candidates for offloading and used graph search methods to identify the instants when to insert offload/prefetch operations. AutoSwap [Zhang et al., 2019] decides which activations to offload by assigning each variable a priority score. SwapAdviser [Huang et al., 2020] used a Genetic Algorithm (GA) to find the best schedule (execution order of the modules) and memory allocation; it relied on Swap Planner to decide which tensors to offload (based on their life cycle) and when to perform offload/prefetch (as soon as possible). Authors in [Beaumont et al., 2020] made a thorough theoretical analysis of the problem. They proposed optimal solutions and extended them in [Beaumont et al., 2021a] to jointly optimize activation offloading and rematerialization.

Support in popular open-source frameworks. Framework vDNN++[5] implemented a technically improved version of vDNN. TFLMS [Le et al., 2018] was initially released as a TensorFlow pull request but later got its own repository[6]. A

[1] https://pytorch.org
[2] https://github.com/parasj/checkmate
[3] https://www.tensorflow.org
[4] https://gitlab.inria.fr/hiepacs/rotor
[5] https://github.com/shriramsb/vdnn-plus-plus
[6] https://github.com/IBM/tensorflow-large-model-support

Method	# of GPUs	Approx. computations	Communication costs per iteration activation values / weight values / activation grads / weight grads	Batch size per GPU increase?	# of FLOP per iteration
No data parallelism	1	baseline	baseline	baseline	baseline
Rematerialization	≥ 1	✗	= / = = / =	✓	↑
Offloading:					
activations	≥ 1	✗	↑ / = = / =	✓	=
weights	≥ 1	✗	= / ↑ = / ↑ or =	✓	=
tensors in GPU cache	≥ 1	✗	↑ or = / ↑ or = = / ↑ or =	✓	=
Approx. gradients:					
lower-bit activation grad.**	≥ 1	✓	↓ / = = / =	✓	↓ or =
approx. matmul**	≥ 1	✓	↓ / = = / =	✓	↕
lower-bit weight grad**	≥ 1	✓	= / = = / ↓	✓	↓ or =
Data parallelism*	> 1	✗	baseline	✗	=
Partitioning:					
optim. state	> 1	✗	= / ↑ = / =	✓	=
+ gradients	> 1	✗	= / ↑ = / =	✓	=
+ parameters	> 1	✗	= / ↑ = / =	✓	=
Model parallelism*	> 1	✗	↑ / = ↑ / ↓	✓	=
Pipeline parallelism	> 1	✓/ ✗	↑ / = ↑ / ↕	✓	=

*We assume updates performed in synchronous way. If updates are asynchronous than for both data and model parallelism less gradients contribute to epoch update and number of epochs till convergence might be increased.

**Communication channel in this case is a bus between a processing unit and a memory bank.

Table 1: Methods to train large neural networks. ↑ and ↓ correspond to the increase and decrease comparing to the baseline above. FLOP is floating-point operations.

Paper	Approach	Scope	Guarantees	Complexity	Implementation
[Griewank and Walther, 2000]	dynprog	hom. seq.	optimal	$O(L^2M)$	REVOLVE
[Grimm et al., 1996]	closed-form				
[Walther and Griewank, 2008]	dynprog	het. time, hom. memory	optimal	$O(L^3M)$	-
[Siskind and Pearlmutter, 2018]	divide & conquer	compiler	-	-	checkpointVLAD
[Chen et al., 2016]	periodic	het. seq	heuristic	-	PyTorch
[Gruslys et al., 2016]	dynprog	RNN	-	$O(L^2M)$	BPTT
[Beaumont et al., 2019]	dynprog	het. seq	optimal w/ assumptions	$O(L^3M)$	Rotor
[Jain et al., 2020b]	ILP	any	optimal	NP-hard	CheckMate
	rational LP		heuristic	$O(EL)$ vars & constraints	
[Kusumoto et al., 2019]	exact dynprog	any	-	$O(T2^{2L}))$	Recompute
	approx. dynprog		heuristic	$O(TL^2)$	
[Kumar et al., 2019]	tree-width decomposition	any	bounded	$2^{O(w)}L + O(wL \log L)$	-
[Kirisame et al., 2020]	greedy (priority scores)	any	heuristic	-	DTR

Table 2: Comparison of rematerialization strategies. The complexity is expressed with respect to number of modules L, memory on GPU M, no-checkpoint execution time T, E is the number of dependencies between layers, i.e. number of activations ($E = \Theta(L)$ in linear case, $E = O(L^2)$ in general) and w is a treewidth of the computational graph.

branch of the Rotor framework[7] provides the implementation of the combined offloading and rematerialization algorithms from [Beaumont et al., 2021a].

[7] https://gitlab.inria.fr/hiepacs/rotor/-/tree/offload-all-rl

2.3 Offloading of Weights

A lot of methods mentioned earlier are also suitable for offloading weights as they rely on universal techniques applicable to any tensors, for example, TFLMS, AutoSwap or SwapAdvisor. L2L (layer-to-layer) [Pudipeddi et al., 2020] keeps a single layer in GPU memory, which results in a significant reduction in the memory cost of the network. Granular CPU

Paper	What to offload	Scope	Features
vDNN [Rhu et al., 2016]	all/conv	seq, CNN	choice between memory/performance efficient kernels for conv
TFLMS [Le et al., 2018]	tensors with longer lifetime	any	rewriting graph with swap in/out; treesearch with bounds to schedule transfers
AutoSwap [Zhang et al., 2019]	tensors with highest priority scores	any	uses bayesian optmization to mix priority scores; optimizes memory allocation
SwapAdviser [Huang et al., 2020]	tensors with longer lifetime	any	memory allocation and scheduling operations done with Genetic Algorithm
[Beaumont et al., 2020]	chosen by greedy/dynprog	het. seq.	theoretical analysis, optimality proofs for relaxed models
rotor [Beaumont et al., 2021a]	chosen by dynprog	het. seq.	combination of offloading and rematerialization, optimal w/ assumptions

Table 3: Comparison of offloading strategies.

offloading [Lin et al., 2021] extends this approach and keeps a part of the network in GPU when there is enough memory. Their experiment shows that offloading only the first half of the network can significantly reduce the training time.ZeRO-Offload [Ren et al., 2021] managed to reduce the high communication cost by introducing the one-step Delayed Parameter Update (DPU) method. In ZeRO-Offload, only the gradients are offloaded to the CPU to update the weights and asynchronous updates are used to limit synchronizations.

Support in popular open-source frameworks. DeepSpeed [Rasley et al., 2020] implements the extremely aggressive memory management strategies proposed in ZeRO-Offload [Ren et al., 2021], which allows a single GPU to train models with more than 10 billion parameters.

3 Parallelism for Models that Don't Fit on a Single GPU

When using Model Parallelism [Dean et al., 2012], different layers of a network are allocated onto different resources, so that the storage of DNN weights and activations is shared between the resources. In Model Parallelism (MP), only activations have to be communicated and transfers only take place between successive layers assigned to different processors. Comparison of papers mentioned in this section is presented in Table 4.

The execution within Model Parallelism can be accelerated if several mini-batches are pipelined [Huang et al., 2019], and thus several training iterations are active at the same time. Once forward and backward phases have been computed on all these mini-batches, the weights are then updated. This approach is fairly simple to implement but it leaves computational resources largely idle. The PipeDream approach proposed in [Narayanan et al., 2019] improves this training process, by only enforcing that the forward and backward tasks use the same model weights for a given mini-batch. Such a weakened constraint on the training process allows PipeDream to achieve a much better utilization of the processing resources, but the asynchronous updates affect badly the overall convergence of the training in some cases [Li and Hoefler, 2021].

It has been shown that performing the updates less regularly [Narayanan et al., 2021a] helps limiting weight staleness as well. Alternatively, PipeMare [Yang et al., 2021] proposes to adapt the learning rate and the model weights for backward depending on the pipeline stage. The last method achieves the same convergence rate as GPipe, while having the same resource utilization as PipeDream without storing multiple copies of the weights.Another important issue related to PipeDream is the need to keep many copies of the model parameters, which can potentially cancel the benefit of using Model Parallelism. To address this issue, the methods to limit weight staleness can be used: in [Narayanan et al., 2021a] the updates are done so that it is possible to keep only two versions of the weights (Double-Buffering).

Modeling the storage cost induced by activations in pipeline approaches is a difficult task [Beaumont et al., 2021b]. Some pipelines (DAPPLE [Fan et al., 2021], Chimera [Li and Hoefler, 2021]) use the One-Forward-One-Backward scheduling (1F1B) to reduce memory consumption related to activations. It is a synchronous weight update technique that schedules backward passes of each micro-batch as early as possible to release the memory occupied by activations. Gems [Jain et al., 2020a] and Chimera [Li and Hoefler, 2021] implement bidirectional pipelines, where each GPU is used for two pipeline stages (i and $P-i$, P is the number of stages). The design of Gems is mostly concerned with activations memory: the forward pass of the next micro-batch starts after the first backward stage of the previous micro-batch is computed and activations memory is released. Chimera rather focuses on reducing the computational bubble by starting the forward passes of each pipeline direction simultaneously. A resembling approach was taken in [Narayanan et al., 2021b], where each GPU is assigned more than one pipeline stages (referred to as the Interleaved Pipeline).

Several papers specifically target challenging topologies. To solve the problem in the case of high communication costs and heterogeneous networking capabilities, the authors of Pipe-torch [Zhan and Zhang, 2019] propose an updated dynamic programming strategy which assumes no overlap between computations and communications. HetPipe [Park et al., 2020] addresses the additional problem of heterogeneous GPUs by grouping them into virtual workers and run-

Paper	Parallelism	Pipeline Feature	Partition Optimization
GPipe [Huang et al., 2019]	DP, PP	First introduced pipelining	-
Megatron-LM [Narayanan et al., 2021b]	TP, DP, PP	1F1B, Interleaved Pipeline	Heuristic
PipeDream [Narayanan et al., 2019]	DP, PP	Async Update	DynProg for DP, PP
PipeDream-2BW [Narayanan et al., 2021a]	DP, PP	Async Double-Buffered Update	DynProg for DP, PP, Checkpointing
DAPPLE [Fan et al., 2021]	DP, PP	1F1B	DynProg for DP, PP
PipeMare [Yang et al., 2021]	PP	Async Update, LR Rescheduling, Weight Discrepancy Correction	Splitting weights evenly between model partitions
Piper [Tarnawski et al., 2021]	TP, DP, PP	Async Update	DynProg for TP, DP, PP, Checkpointing
HetPipe [Park et al., 2020]	DP, PP	Parameter Server	LinProg for PP
Pipe-torch [Zhan and Zhang, 2019]	DP, PP	Async Update	DynProg for DP, PP, GPU allocation
Varuna [Athlur et al., 2021]	DP, PP	Opportunistic Backward Scheduling	Heuristic PP partition, Bruteforce for DP, PP depth
Gems [Jain et al., 2020a]	DP, PP	Bidirectional Pipeline	-
Chimera [Li and Hoefler, 2021]	DP,PP	1F1B, Bidirectional Pipeline	Greedy mini-batch size, Bruteforce for DP, PP depth

Table 4: Comparison of model parallelism strategies.

ning pipeline parallelism within each virtual worker, while relying on data parallelism between workers. Varuna [Athlur et al., 2021] focuses on "spot" (low-priority) VMs and builds a schedule that is robust to network jitter, by performing a pipelining technique that resembles 1F1B: activation recomputations and respective backward passes are scheduled opportunistically.

4 Optimizers for Cross-Device Model Training

4.1 Zero Redundancy Optimizer

The authors of [Rajbhandari et al., 2020] propose ZeRO (Zero Redundancy Optimizer) as an implementation of data-parallelism with reduced memory footprint. The algorithm has three versions depending on what tensors are partitioned across devices: Stage 1 (optimizer states), Stage 2 (optimizer states and gradients), and Stage 3 (optimizer states, gradients and model parameters). ZeRO works in a mixed precision regime to reduce the amount of data transferred between devices. Still, Stages 2 and 3 introduce a communication overhead. In [Ren et al., 2021] authors propose to unite ZeRO and CPU-side computation of parameter updates within ZeRO-Offload: gradients are transferred to CPU where copies of parameters are stored; the update is applied to the copies and the updated weights are transferred back to GPU. [Rajbhandari et al., 2021] further elaborates ZeRO-Offload with utilisation of NVMe memory, offload of activations, better computation-communication overlap and other improvements.

Support in popular open-source frameworks. An open source implementation of all ZeRO-* algorithms is available in the DeepSpeed[8] framework.

[8] https://github.com/microsoft/DeepSpeed

4.2 Low-Precision Optimizers

To further reduce memory footprint, low-precision optimizers can be used. These methods use low precision formats to represent the optimizers states and auxiliary vectors of states. Also, error compensation techniques are used to preserve the approximation accuracy of the tracking statistics. [Dettmers et al., 2021] proposes a method to store statistics of Adam optimizer in 8-bit while the overall performance remains the same as when the 32-bit format is used. The key technique to achieve such a result is blockwise dynamic quantization that efficiently handles both large and small magnitude elements. More aggressive precision reduction is presented in [Sun et al., 2020], where special routines to deal with 4-bit representation are developed. In particular, the adaptive Gradient Scaling (GradScale) method aims to mitigate the issue with insufficient range and resolution.

4.3 Acceleration of Convergence

Another way to accelerate training of large deep learning models is to reduce communication time between nodes and/or number of required epochs to converge at the appropriate local minimum.

Communication costs reduction. Different approaches have been proposed to compress gradients before transferring them between computational nodes. In particular, three classes of such methods are typically discussed: sparsification, quantization and low-rank methods. Sparsification methods only transfer some subset of complete gradient elements and update the corresponding elements in the parameter vector. This approximation significantly reduces the communication costs [Aji and Heafield, 2017; Alistarh et al., 2019] while the trained model performance is preserved. Another approach is based on quantization of transferred gradients, which consists of transferring only a limited number of bits, reconstructing the entire gradient vector from them,

and updating all elements of the parameter vector. This approach demonstrates promising results for some neural networks architectures and experimental settings [Alistarh et al., 2017]. In particular, recent results in sending only the signs of stochastic gradient elements [Stich et al., 2018] have been extended to more complicated Adam optimizer [Tang et al., 2021], where the non-linear effect of the optimizer states requires additional investigation of error compensation strategies. Another approach for communication costs reduction is the low-rank approach, in which a low-rank approximation of the gradient is constructed, transferred and used to recover the gradient in full format before updating the parameter vector. The low-rank approximation is constructed with the block power method [Vogels et al., 2019] or with the alternating minimization strategy [Cho et al., 2019]. The main difficulty here is to balance the gains from the reduction of communication costs with the additional costs induced by the construction of low-rank approximations. A comprehensive analysis and numerical comparison of many methods for communication overhead reduction from the aforementioned classes are presented in review [Xu et al., 2021].

Large batch training Another approach to speed up the convergence of the optimizer is to use a large number of samples per batch. This training setting leads to a reduction of the number of iterations in every epoch and a better utilization of GPU. In [Goyal et al., 2017] authors propose to use a linear scaling rule to update the learning rate along with the batch size. This setting stabilizes the optimization process and converges to the same final performance of the model. Note also that large batch training significantly reduces the variance in the stochastic gradient estimate. However, study [Keskar et al., 2016] observes that this feature of the training reduces the generalization ability of the trained model if other hyperparameters remain the same. Therefore, other alternatives to the linear scaling rule have been considered in further works.

In particular, Layer-wise Adaptive Rate Scaling is combined with SGD [You et al., 2017] (LARS scheme) and Adam optimizers [You et al., 2019] (LAMB scheme) to adjust the learning rate in every layer separately. These strategies are based on the observation of a significant difference in the magnitude of parameters and gradients in the different layers, which is natural for deep neural networks. Note that the memory saving techniques considered in Sections 2 typically allow the batch size to be increased with little overhead, even when exceeding the GPU memory capabilities.

5 Conclusion and Further Research

In the survey we have discussed methods that help to train larger models on a single GPU and do more efficient training on multiple GPUs. Such methods optimize both the training of known high-quality large models (e.g., GPT, CLIP, DALLE) from scratch and the fine-tuning of pre-trained models for specific and personalized tasks.

Several main factors influence the training of large DNNs: (i) memory required to store model parameters, activations, optimizer states(ii) time spent on data exchange (communication) between computing nodes and its impact on the computing time on a separate computing node, (iii) efficiency parallel computing (percentage of time when GPUs are not idle), (iv) the number of floating-point operations required to calculate the forward and backward passes for the given model architecture, dataset, and target functionality, (v) the number of iterations required to achieve the specified accuracy. The strategies discussed in Sections 2,3, and 4 of this survey are applied to reduce the influence of these factors.

Current research in rematerilization (Table 2) focus on finding the optimal checkpointing strategy if we have a homogeneous or heterogeneous sequential model. However, modern neural networks have a more complex structure - for example, due to a large number of residual connections. Currently, optimal rematerialization strategies for each architecture and input data size are heuristically searched. Further research can find theoretically optimal solutions for more general types of architectures. Rematerialization methods demonstrated their benefits in reducing memory on one GPU. Despite that, it is important to combine it with other methods to achieve significant decrease in memory consumption. For example, in [Beaumont et al., 2021a] considering the optimal combinations of offloading and rematerialization (Section 2) further pushed the performance of both methods. Considering other combinations, e.g. checkpointing and pipelining (Section 3), can be a promising further development of both methods.

In optimization methods (Section 4) there are three main research directions to adapt them for large model training: low-precision storage of states and gradients, batch size increasing with learning rate scheduling, compression of transferred gradients. They demonstrate promising results to train particular models but, at the same time, they are quite far from the complete technology. For example, the low precision approach requires extensive hardware support of the operations with numbers in a low-precision format, and the compression scheme for gradient transmission can be efficient only after the careful setting of the broadcasting environment. Thus, these approaches require additional research to make them robust and widespread.

Among the promising directions, we should mention computations with reduced precision, approximate methods [Novikov et al., 2022], randomized computations [Bershatsky et al., 2022], and structured NN layers [Hrinchuk et al., 2020], including those based on tensor factorizations. We should highlight the importance for these approximate methods to be additive in the sense that they can be combined and still provide sufficient enough performance with reasonable quality degradation.

Finally, it is worth emphasizing the importance of developing new promising computing architectures that can speed up elementary machine learning operations (for example, matrix-vector multiplication) and low-level optimization techniques.

Acknowledgments

The work was supported by the Analytical center under the RF Government (subsidy agreement 000000D730321P5Q0002, Grant No. 70-2021-00145 02.11.2021).

This work was supported by the Inria Molière International Associated Team between Skoltech and Inria.

6 Model performance and bottlenecks identification

Building high-performance computing systems requires a deep understanding of how the software works. In addition, it is necessary to understand how the software interacts with PC hardware resources, primarily with random access memory (RAM). Analyzing such interactions and calculating memory/time performance metrics is necessary to create highly efficient software.

Hardware and software. Deep learning requires high-performance computing systems and powerful hardware. The most popular hardware computers are the following elements: CPU (Central processing unit); GPU (Graphics processing unit); TPU (Tensor processing unit); neuromorphic computers; promising developments based on memristors (semiconductor elements capable of changing their electrical resistance depending on the current flowing through them) and quantum computers.

Despite a large variety of computational units, GPUs are most often used to train NNs. This is both because the GPU architecture is very well adapted for parallel computing, which includes operations such as matrix multiplication, operations on vectors, and because of the mass prevalence of such calculators, a large number of software and relatively low cost. Indeed, operations on vectors or matrices make up the majority of the calculations that take place in neural networks. In addition, the bandwidth of the GPU memory is much higher than the bandwidth of the RAM memory, which allows operations on a large array of input data to be performed more efficiently.

High-performance software are designed to make use of the hardware resources as efficiently as possible. To create such software, a low-level interaction with the main memory is required. Also, such software depends on the processor architecture and considers such characteristics as the amount of cache memory (L1 / L2 / L3 cache) and memory bandwidth. High-performance software also minimizes resource idle state, i.e., a state in which function calls in a serial scheme are executed at different speeds, as a result of which all resources involved in the calculation remain reserved until the slowest process completes calculations. For example, the slowest operation may be loading, reading, and decoding input data when training an NN. As a result, the learning rate will not be determined by the characteristics of the GPU or CPU, but by the data reading speed. The main software libraries for high-performance mathematical operations are such libraries as BLAS, LAPACK, MKL. These libraries support mathematical operations on vectors and matrices, are written in low-level programming languages, and efficiently use PC resources for calculations.

To execute a program on a GPU, it must be written in a particular programming language and compiled with an appropriate compiler. The CUDA C programming language allows you to develop and compile software that will run on a GPU. The CUDA C programming language is suitable for parallel computing. To date, libraries have been developed in the CUDA C language, which is the low-level basis for neural networks. These libraries include cuBLAS, cuDNN, Thrust.

optimizer	deepspeed optimization techniques											
	zero 0			zero 1			zero2			zero3		
	vRAM	Time	Loss	vRAM	Time	Loss	vRAM	Time	Loss	vRAM	Time	Loss
Adam	69.7	0.95	3.540	61.6	0.90	3.541	60.6	0.85	3.541	51.9	1.20	3.542
8bit Adam	59.7	0.88	3.539	60.8	0.89	3.540	60.2	**0.84**	3.540	51.3	1.18	3.541
CPU Adam	-	-	-	-	-	-	59.6	1.44	3.541	**50.1**	1.58	3.542
Fused Adam	69.7	0.90	3.540	61.6	0.89	3.541	60.6	0.86	3.541	51.9	1.18	3.542
1bit Adam	73.1	1.51	3.523	-	-	-	-	-	-	-	-	-

Table 5: It is the comparison of optimization techniques in finetuning of ruDALL-E Malevich. All experiments have the following setup: train size 2474, valid size 275, loss image weight 1000, batch size 3, "WarmupDecayLR" min lr 1.5e-8, max lr 7.5e-6, warmup steps 55, total num steps 550, weight decay 0.2, betas (0.9, 0.98), eps 1e-6, without deepspeed gradient checkpointing, trained using 2x8xA100. "vRAM" (MaxCacheGb) values in GB, "Time" values in seconds per training step, "Loss" values is weighted text and image CrossEntropy.

These libraries contain high-performance implementations of the operations used in the ANN. The Python programming language is predominantly used when writing methods for solving deep learning problems. To work with NNs, special libraries are used (PyTorch, TensorFlow, JAX, MXNet, etc.), the source code of which is written in Python and C++.

Busses and communications. For deep NNs training, the choice of bus and data transfer protocol is also important. Note two types of protocols: protocols for local communication on a single node (NVLink[9], PCIe[10]) and communication protocols between nodes (Infiniband[11]).

Libraries NCCL [12], MPI[13] and GLOO[14] can be used to organize computations on the GPU.

The NCCL library — is the best choice when training with multiple GPUs on the same node or if the GPUs on different nodes are connected by InfiniBand (NCCL does not support communication with the CPU). If nodes are connected by InfiniBand via CPU, either Gloo (if InfiniBand supports IP over IB) or MPI is used. We also note the NCCL Tests[15] library from NVIDIA for checking the correctness and speed of NCCL operations on available hardware, which can be used in the software.

Time complexity. In order to theoretically measure the computational time complexity of NNs, FLOP (floating point operations) or MAC (multiply - accumulate operations) are usually calculated. For example, for a fully connected layer with an input vector of size N_{in} and a weight matrix of size $N_{in} \times N_{out}$, MAC and FLOP are calculated by the formulas

$$\text{MAC} = N_{in}N_{out}, \quad \text{FLOP} = 2N_{in}N_{out} + N_{out}.$$

However, real computational complexity can differ from the theoretical one. Besides architecture type it is highly dependent on the device characteristics (CPU/GPU type, RAM memory, latency, etc.), which is used for time measurements, and on the framework, which is used for the implementation (PyTorch/TensorFlow/Caffe, etc.).

Memory complexity. When training a NN on a GPU, the memory on the device must be allocated not only to store the model parameters, but also to store the outputs of the intermediate network layers (activations), which are used to calculate gradients during error backpropagation. Therefore, the GPU memory requested during training iteration exceeds the memory required during model inference.

Arithmetic intensity and bottlenecks identification . In practice, the time complexity is limited by two factors: the time of mathematical operations T_{math} and the time of memory access T_{mem}. If memory access and mathematical operations can be performed on the device simultaneously for different subtasks, which is supported by modern GPUs, the total task execution time is limited by either memory access or computation time: $T = \max(T_{\text{mem}}, T_{\text{math}})$. Math speed, denoted BW_{math}, and memory access speed, denoted BW_{mem}, are measured in FLOPs per second (FLOP/s) and bytes per second (b/s) respectively.

The ratio of the number of memory accesses and mathematical operations, called the arithmetic intensity, is a parameter of a particular processor. For all modern GPUs, the speed of mathematical operations significantly exceeds the speed of reading from memory. The intensity of operations supported by the GPU can be calculated using the formula

$$\rho = \frac{T_{\text{math}}}{T_{\text{mem}}} = \frac{BW_{\text{math}}}{BW_{\text{mem}}} \tag{1}$$

This value can also be estimated for any algorithm as follows:

$$\rho_{\text{algo}} = \frac{N_{\text{FLOP}}}{N_{\text{mem}}} \tag{2}$$

where N_{FLOP} is the number of floating point operations performed by the algorithm, and N_{mem} is the number of memory accesses. If ρ_{algo} approximately coincides with the density

[9] https://www.nvidia.com/en-us/data-center/nvlink/

[10] https://pcisig.com /

[11] https://www.infinibandta.org/ibta-specification/

[12] NVIDIA Collective Communications Library, https://developer.nvidia.com/nccl

[13] Message Passing Interface, https://www.openmp.org/

[14] Facebook library, https://github.com/facebookincubator/gloo

[15] https://github.com/NVIDIA/nccl-tests

Operation	Bound	Arith. Int.
Linear (4096 x 1024, batch 512)	Comp	315
Linear (4096 x 1024, batch 1)	Mem	1
MaxPooling (kernel 3x3)	Mem	2.25
ReLU	Mem	0.25
LayerNorm	Mem	< 10

Table 6: Arithmetic intensity (in FLOPs/b) for common operations in half precision for Nvidia A100 (abbreviations Comp and Mem stand for compute-bound and memory-bound respectively).

for the ρ processor, then both the memory and the processor are loaded to the maximum during the execution of the algorithm. In the case of $\rho_{\text{algo}} < \rho$, the algorithm will be limited by the speed of memory access, and in the case of $\rho_{\text{algo}} > \rho$, it will be limited by the speed of arithmetic operations (faster).

For example, the maximum operation speed for the Nvidia A100 is 125 TFLOP/s in 16-bit arithmetic, and the data transfer rate is 900 GB/s from the main memory to the processor and 3.1 TB/s from the L2 cache on the processor. In this case, the optimal intensity of 16-bit operations is 139 and 40, depending on the initial location of the data (in the main memory of the GPU or in the cache, respectively). Table 6 lists the typical intensity for various operations in deep neural networks.

Profiling tools Profiling is a process of program analysis in runtime. Its main goal is to measure general performance metrics of whole program as well as performance counter for a specific part of a software (e.g. function or module). Ultimate goal of profiling tools is search and elimination of bottlenecks. Common usage pattern of such tooling is top-down analysis of a software from high-level to low-level objects until bottleneck is found. General purpose utilities for programm analysis as well as specialized ones are applicable to profiling of neural network training.

In general there are many directions for profiling. The first one lies between Python and native code and specific for CPU. The usefull utilities are cProfile[16] in Python, and perf[17] in the general case. They helps to identify issues in Python or native code as well as operating system issues.

Another profiling direction is analysis of GPU performance and interdevice communications among GPUs and CPU. Framework-independent but vendor-specific solution is NVIDIA Nsight[18] which enables one to watch specific kernel, performance counter, and communication operations. On the other hand, framework-dependent tools like PyTorch Profiler aka Kineto[19] extends NVIDIA Nsight with knowledge about high-level abstractions and low-level operations of PyTorch and visualizes with a plugin to TensorBoard[20]. The latter tool is irreplaceable in development of PyTorch extention modules.

[16] https://docs.python.org/3/library/profile.html
[17] https://perf.wiki.kernel.org/index.php/Main_Page
[18] https://docs.nvidia.com/nsight-systems/UserGuide/index.html
[19] https://github.com/pytorch/kineto
[20] https://github.com/tensorflow/tensorboard

References

[Aji and Heafield, 2017] Aji, A. F. and Heafield, K. (2017). Sparse Communication for Distributed Gradient Descent. In *EMNLP*, pages 440–445.

[Alistarh et al., 2017] Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. (2017). QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *NIPS*.

[Alistarh et al., 2019] Alistarh, D., Hoefler, T., Johansson, M., Kririrat, S., Konstantinov, N., and Renggli, C. (2019). The Convergence of Sparsified Gradient Methods. In *NeurIPS*, pages 5973–5983.

[Athlur et al., 2021] Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. (2021). Varuna: Scalable, Low-cost Training of Massive Deep Learning Models. *arXiv:2111.04007*.

[Beaumont et al., 2019] Beaumont, O., Eyraud-Dubois, L., Herrmann, J., Joly, A., and Shilova, A. (2019). Optimal Checkpointing for Heterogeneous Chains: How to Train Deep Neural Networks with Limited Memory. Research Report RR-9302, INRIA.

[Beaumont et al., 2020] Beaumont, O., Eyraud-Dubois, L., and Shilova, A. (2020). Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training. In *Euro-Par*.

[Beaumont et al., 2021a] Beaumont, O., Eyraud-Dubois, L., and Shilova, A. (2021a). Efficient Combination of Rematerialization and Offloading for Training DNNs. In *NeurIPS*.

[Beaumont et al., 2021b] Beaumont, O., Eyraud-Dubois, L., and Shilova, A. (2021b). Pipelined Model Parallelism: Complexity Results and Memory Considerations. In *Euro-Par*, pages 183–198.

[Bershatsky et al., 2022] Bershatsky, D., Mikhalev, A., Katrutsa, A., Gusak, J., Merkulov, D., and Oseledets, I. (2022). Memory-Efficient Backpropagation through Large Linear Layers. *arXiv:2201.13195*.

[Chen et al., 2016] Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training Deep Nets with Sublinear Memory Cost. *arXiv:1604.06174*.

[Cho et al., 2019] Cho, M., Muthusamy, V., Nemanich, B., and Puri, R. (2019). GradZip: Gradient Compression using Alternating Matrix Factorization for Large-scale Deep Learning. In *Workshop on Systems for ML at NeurIPS'19*.

[Dean et al., 2012] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M. A., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. (2012). Large Scale Distributed Deep Networks. In *NIPS*.

[Dettmers et al., 2021] Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. (2021). 8-bit Optimizers via Block-wise Quantization. *CoRR*.

[Fan et al., 2021] Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., et al. (2021). DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *PPOPP*.

[Goyal et al., 2017] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, Large Minibatch SGD. *arXiv:1706.02677*.

[Griewank and Walther, 2000] Griewank, A. and Walther, A. (2000). Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM TOMS*.

[Grimm et al., 1996] Grimm, J., Pottier, L., and Rostaing-Schmidt, N. (1996). Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs. Technical report, INRIA.

[Gruslys et al., 2016] Gruslys, A., Munos, R., Danihelka, I., Lanctot, M., and Graves, A. (2016). Memory-Efficient Backpropagation Through Time. In *NIPS*.

[Hrinchuk et al., 2020] Hrinchuk, O., Khrulkov, V., Mirvakhabova, L., Orlova, E., and Oseledets, I. (2020). Tensorized embedding layers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4847–4860, Online. Association for Computational Linguistics.

[Huang et al., 2020] Huang, C.-C., Jin, G., and Li, J. (2020). SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *ASPLOS*.

[Huang et al., 2019] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H. J., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. (2019). GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*.

[Jain et al., 2020a] Jain, A., Awan, A. A., Aljuhani, A. M., Hashmi, J. M., Anthony, Q. G., Subramoni, H., Panda, D. K., Machiraju, R., and Parwani, A. (2020a). GEMS: GPU-Enabled Memory-Aware Model-Parallelism System for Distributed DNN Training. In *SC*.

[Jain et al., 2020b] Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. (2020b). Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *MLSys*.

[Keskar et al., 2016] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv:1609.04836*.

[Kirisame et al., 2020] Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. (2020). Dynamic Tensor Rematerialization. *arXiv:2006.09616*.

[Kumar et al., 2019] Kumar, R., Purohit, M., Svitkina, Z., Vee, E., and Wang, J. (2019). Efficient Rematerialization for Deep Networks. In *NeurIPS*.

[Kusumoto et al., 2019] Kusumoto, M., Inoue, T., Watanabe, G., Akiba, T., and Koyama, M. (2019). A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. In *NeurIPS*.

[Le et al., 2018] Le, T. D., Imai, H., Negishi, Y., and Kawachiya, K. (2018). TFLMS: Large Model Support in TensorFlow by Graph Rewriting. *arXiv:1807.02037*.

[Li and Hoefler, 2021] Li, S. and Hoefler, T. (2021). Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. In *SC*, pages 1–14.

[Lin et al., 2021] Lin, J., Yang, A., Bai, J., Zhou, C., Jiang, L., Jia, X., Wang, A., Zhang, J., Li, Y., Lin, W., et al. (2021). M6-10T: A Sharing-Delinking Paradigm for Efficient Multi-Trillion Parameter Pretraining. *arXiv:2110.03888*.

[Narayanan et al., 2019] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. (2019). Pipedream: Generalized Pipeline Parallelism for DNN Training. In *SOSP*.

[Narayanan et al., 2021a] Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. (2021a). Memory-Efficient Pipeline-Parallel DNN Training. In *ICML*.

[Narayanan et al., 2021b] Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. (2021b). Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *SC*, pages 1–15.

[Novikov et al., 2022] Novikov, G., Bershatsky, D., Gusak, J., Shonenkov, A., Dimitrov, D., and Oseledets, I. (2022). Few-Bit Backward: Quantized Gradients of Activation Functions for Memory Footprint Reduction. *arXiv:2202.00441*.

[Park et al., 2020] Park, J. H., Yun, G., Yi, C. M., Nguyen, N. T., Lee, S., Choi, J., Noh, S. H., and ri Choi, Y. (2020). HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *USENIX ATC*.

[Pudipeddi et al., 2020] Pudipeddi, B., Mesmakhosroshahi, M., Xi, J., and Bharadwaj, S. (2020). Training Large Neural Networks with Constant Memory using a New Execution Algorithm. *arXiv:2002.05645*.

[Rajbhandari et al., 2020] Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. (2020). ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *SC*, pages 1–16.

[Rajbhandari et al., 2021] Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. (2021). ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *SC*.

[Rasley et al., 2020] Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. (2020). DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *ACM SIGKDD*.

[Ren et al., 2021] Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. (2021). ZeRO-Offload: Democratizing Billion-Scale Model Training. *arXiv:2101.06840*.

[Rhu et al., 2016] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. (2016). vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *MICRO*.

[Siskind and Pearlmutter, 2018] Siskind, J. M. and Pearlmutter, B. A. (2018). Divide-and-Conquer Checkpointing for Arbitrary Programs with no User Annotation. *Optimization Methods and Software*.

[Stich et al., 2018] Stich, S. U., Cordonnier, J.-B., and Jaggi, M. (2018). Sparsified SGD with Memory. In *NIPS*, pages 4452–4463.

[Sun et al., 2020] Sun, X., Wang, N., Chen, C.-Y., Ni, J., Agrawal, A., Cui, X., Venkataramani, S., El Maghraoui, K., Srinivasan, V. V., and Gopalakrishnan, K. (2020). Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *NeurIPS*, volume 33, pages 1796–1807.

[Tang et al., 2021] Tang, H., Gan, S., Awan, A. A., Rajbhandari, S., Li, C., Lian, X., Liu, J., Zhang, C., and He, Y. (2021). 1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. In *ICML*.

[Tarnawski et al., 2021] Tarnawski, J. M., Narayanan, D., and Phanishayee, A. (2021). Piper: Multidimensional Planner for DNN Parallelization. In *NeurIPS*, volume 34.

[Vogels et al., 2019] Vogels, T., Karinireddy, S. P., and Jaggi, M. (2019). PowerSGD: Practical low-rank gradient compression for distributed optimization. In *NeurIPS*.

[Walther and Griewank, 2008] Walther, A. and Griewank, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM.

[Xu et al., 2021] Xu, H., Ho, C.-Y., Abdelmoniem, A. M., Dutta, A., Bergou, E. H., Karatsenidis, K., Canini, M., and Kalnis, P. (2021). GRACE: A compressed communication framework for distributed machine learning. In *ICDCS*, pages 561–572.

[Yang et al., 2021] Yang, B., Zhang, J., Li, J., Re, C., Aberger, C., and De Sa, C. (2021). PipeMare: Asynchronous Pipeline Parallel DNN Training. In *MLSys*.

[You et al., 2017] You, Y., Gitman, I., and Ginsburg, B. (2017). Large Batch Training of Convolutional Networks. *arXiv:1708.03888*.

[You et al., 2019] You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C.-J. (2019). Large Batch Optimization for Deep Learning. *arXiv:1904.00962*.

[Zhan and Zhang, 2019] Zhan, J. and Zhang, J. (2019). PipeTorch: Pipeline-Based Distributed Deep Learning in a GPU Cluster with Heterogeneous Networking. In *CBD*.

[Zhang et al., 2019] Zhang, J., Yeung, S. H., Shu, Y., He, B., and Wang, W. (2019). Efficient Memory Management for GPU-based Deep Learning Systems. *arXiv:1903.06631*.