Pipelined Model Parallelism: Complexity Results and Memory Considerations

Olivier Beaumont, Lionel Eyraud-Dubois, Alena Shilova

Pipelined Model Parallelism: Complexity Results and Memory Considerations

Olivier Beaumont, Lionel Eyraud-Dubois and Alena Shilova
Inria Bordeaux – Sud-Ouest and Université de Bordeaux
Bordeaux, France
E-mail: `firstname.lastname@inria.fr`

October 16, 2020

Abstract

The training phase in Deep Neural Networks has become an important source of computing resource usage and because of the resulting volume of computation, it is crucial to perform it efficiently on parallel architectures. Even today, data parallelism is the most widely used method, but the associated requirement to replicate all the weights on the totality of computation resources poses problems of memory at the level of each node and of collective communications at the level of the platform. In this context, the model parallelism, which consists in distributing the different layers of the network over the computing nodes, is an attractive alternative. Indeed, it is expected to better distribute weights (to cope with memory problems) and it does not imply large collective communications since only forward activations are communicated. However, to be efficient, it must be combined with a pipelined / streaming approach, which leads in turn to new memory costs. The goal of this paper is to model these memory costs in detail, to analyze the complexity of the associated throughput optimization problem under memory constraints and to show that it is possible to formalize this optimization problem as an Integer Linear Program (ILP).

1 Introduction

Deep Neural Network (DNN) training is a long and memory-intensive operation. Indeed, supervised DNN training requires performing numerous forward and backward computations, each on a subset of input data called a mini-batch. In turn, each forward and backward phase involves complex data dependences and induces memory issues. In practice, parallel training is performed both on small groups of GPU machines and on large HPC infrastructures [20, 41], especially because HPC machines offer high-bandwidth and low-latency networks [23, 31, 12].

The first approach to use parallelism at the level of the node is to make the best use of the available multi-core by optimizing the individual compute kernels, which usually consist of tensor computations. This approach has been widely used in the context of GPUs and TPUs and has made the success of frameworks such as TensorFlow [4], PyTorch [36] or NGraph [1].

At a larger scale, the best known approach to parallel DNN training is the so-called data parallel approach. Using data parallelism [44, 35], the model and all associated weights are replicated on all participating nodes. Then, different mini batches are trained in parallel on different nodes. In this framework, all participating nodes execute forward and backward phases in parallel, and thus all

compute a gradient for all weights in the network. Synchronization between the nodes takes place at the end of the backward step, and all partial gradients are collected and aggregated through a collective communication such as `AllReduce`, and then broadcast to all participating nodes by a `Broadcast` type operation.

The above approach is very possible and scalable as long as two conditions are fulfilled.

The first condition is related to communication resources and states that the network must be able to support the collective communications of the weights, without inducing too much idle computing time. The second condition is related to memory and states that each participating node must be able to store all network weights and activations corresponding to the processing of a mini-batch.

In order to limit network issues and lower the size of the collective communication, data parallelism can be combined with compression to limit the size of messages. When used at a very large scale, this approach nevertheless leads to poor performance due to synchronization and communication costs and requires the use of huge mini-batches, which could also have a negative impact on the performance of the training phase [32].

To deal with limitations induced by memory issues, several approaches have been proposed because it has been observed that in many cases, large and heavy models are required to reach good classification accuracy. In general, the memory consumption during the training phase is composed of two main parts [18].

The first source of memory consumption is due to the storage of all the forward activations on each node (i.e. all the outputs of the different stages of the network) until the associated backward operation. This part is directly proportional to the size of the mini-batch itself. The second part is related to the storage of the network parameters (weights) and is directly connected to the size of the model [28]. Using the data parallel approach, these weights must be replicated on each node, and must even be aggregated and disseminated on the network after each parallel training phase in mini-batches.

To reduce the memory requirements related to the storage of forward activations, several approaches have been proposed. Rematerialization [21, 7, 16, 29, 27, 30] consists in deleting from the memory certain activations computed during the forward phase, which then must be recomputed during the backward phase. Another approach [39, 38, 6] is to offload some of the activations from the GPU memory to the CPU memory during the forward phase and then to bring them back in GPU memory during the backward phase when they are actually required.

In order to limit the memory requirements resulting from the storage of network weights, a natural approach is to distribute the different layers of the network over several computation resources. This approach, denoted as model parallelism, has also been advocated in many papers [13, 25, 32, 42]. In this context, each mini-batch is processed by a sequence of processors, and only activations are communicated between processors. This approach is orthogonal to data parallelism and can naturally be combined with it, using several processors to manage the same sub-part of the DNN.

Unfortunately, if images are processed entirely and in sequence, model parallelism can reduce memory requirements, but it does not accelerate computations because of the shape of the dependencies introduced by back-propagation, as shown in [25, 32]. To obtain some speedup using this approach, it is necessary to process several images in parallel, using a pipelined (or streaming) approach. On the other hand, such approach requires the storage of several models (to ensure that forward and backward computations for the same image use the same weights) and several activations of each type, corresponding to different images being processed in the pipeline.

2

Most of the literature on this approach combining pipelining and model parallelism [25, 32, 33, 42] focuses on the problem of finding an efficient partitioning of the network, that balances the load between processors and minimizes data exchanges between resources. However, in these approaches, memory is processed in a second stage, and typically, the solution found without memory constraints is then degraded, in terms of throughput, to fulfill these constraints.

To the best of our knowledge, the analysis of the induced memory requirements proposed in the present paper is much tighter than what has been proposed in the literature.[40] shares several characteristics with the present paper. As in the present paper, the authors propose an ILP based solution for computing non-contiguous allocations of layers. The main advantage of [40] over our approach is that it does not require graph linearization and can be directly applied to general DAGs. On the other hand, the paper addresses both the inference and training phase, but the specificities of the memory issues in the case of training are not addressed. These specificities are in practice crucial in this context, and the careful modeling of memory requirements is the main contribution of the present paper. Finally, since both [40] and our approach are based on integer linear programming, their use is in practice limited to medium size networks (Resnet-50 is the deepest network considered in both papers).

Overall, in this paper, we perform the theoretical analysis of the pipelined model parallelism under memory constraint. Our theoretical contributions are a very careful modeling of induced memory costs, an analysis of the potential benefits of non-contiguous allocations, a complexity analysis of the problem of finding the best periodic schedule under memory limitation and an Integer Linear Program to compute a non-contiguous solution, that provides significant improvement over the literature for reasonable size networks.

The rest of the document is organized as follows. The related work is presented in Section 2. We introduce the notations and the computational model we use in Section 3. The strong and weak points of the current state-of-the-art are discussed in Section 4. In the framework of this model, we establish several complexity results for the search of the optimal allocation of layers to resources in Section 5. We then propose an Integer Linear Programming (ILP) formulation of the optimization problem, which takes into account all sources of memory consumption in Section 6. The performance of the ILP-based solution, both in terms of solution quality and running time are analyzed in Section 7. Finally, conclusions and perspectives of this work are proposed in Section 8.

2   Related Works

Memory consumption is now becoming an important issue in deep learning and encompasses many different aspects. Unfortunately, the memory limitations of current hardware often prevent data scientists from considering larger models, larger images, or larger batches of images. In practice, training is performed automatically and transparently for the user thanks to autograd tools for backpropagation, such as `tf.GradientTape` in TensorFlow or `torch.autograd.backward` in PyTorch.

In order to reduce memory needs, one line of research for this purpose is to design and train memory efficient architectures and to aim at the same performance as state-of-the-art networks. Reversible Neural Networks [19, 9] (RevNet), for example, are designed to allow the backpropagation algorithm to be performed without storing forward activations until the computation of the associated backward operation. Quantification [37, 26] and pruning [22] rather reduce memory consumption at the time of inference by changing weights and/or network activations into binary or quantized

3

variables. Other *ad-hoc* architectures such as MobileNets [24] or ShuffleNet [43] finally try to sparsify the network architecture in order to reduce the size of the model.

Rematerialization is being increasingly employed to reduce memory usage. The use of rematerialization strategies has recently been advocated for DNN in several papers. A direct adaptation of the results on homogeneous chains has been proposed for the case of Recurrent Neural Networks (RNN) in [21]. A further generalization of the results on homogeneous chains enabled the derivation of optimal rematerialization strategies for joint-networks [7], which consist of several homogeneous chains joined together at the end. Some research has attempted to adapt rematerialization strategies to arbitrary computation graphs (ACG) in [16, 29, 27, 30]. For practical use, an implementation of rematerialization exists in PyTorch [2], based on a simple strategy of periodic and one-pass rematerialization that exploits the ideas presented in [11]. Another implementation [3] based on [5] has also been proposed.

Another alternative [39, 38] is to offload some of the activations from the GPU memory to the CPU memory and to bring them back when required during the backward phase. Finally, domain decomposition or spatial parallelism techniques can be used to limit the memory needed to store the forward activations. In [14], dividing large images into smaller ones makes it possible to train the network in parallel on the smaller images (enhanced by a halo), at the cost of additional communications to synchronize parameter updates. A similar strategy has been proposed for the use of channel and filter parallelism in [15]. Approaches like rematerialization, activation offloading, spatial parallelism and channel parallelism are orthogonal to our approach and could be used in combination to achieve even greater memory savings, although it is out of the scope of the present paper.

An important issue related to memory and the pipelined approach proposed in [32] is the need to keep many copies of the weights. To solve this issue, a strategy has been proposed in [33] to keep only two models in memory. In the present paper, we use the strategy of [33] in a more general framework. Another strategy to keep fewer models in memory is to allow more asynchronous updates. This strategy has been explored in particular in [10], which proposes an intermediate approach to avoid the phenomenon of gradient staleness, though it requires 2 versions of the model weights and 2 version of the gradients, which does not save memory more. We propose in Section 4 a detailed and critical study of the hypotheses underlying the solution provided in PipeDream

3 Model and Notations

3.1 Notations

In what follows we represent each DNNs as a chain so that the task graph corresponding to forward and backward propagation is depicted in Figure 1. This assumption on the shape of the network might require linearization and is used in most of the related literature [11, 32, 33, 10, 42].

Let us model the network as a chain of L layers (that could consist in a single layer or a block of layers in case of prior linearization), numbered from 1 to L. Each layer l is associated both to a forward operation F_l and a backward operation B_l. During training, each mini-batch undergoes a forward pass followed by a backward pass through the whole network and the dependences between tasks are depicted in Figure 1.

As we are interested in solutions in which the different layers of the network are allocated to potentially different computation resources, we also introduce communication tasks $T_{l.l+1}^c$. These

communication tasks correspond to sending forward activations or gradients from one computation resource to another, and their cost will be 0 between two successive layers allocated to the same resource. We denote by $a^{(l)}$ the activation tensor produced by $T_l, l \leq L$ and by $a^{(2L-l)} = b^{(l)} = \frac{\partial \mathcal{L}}{\partial a^{(l)}}$ the back-propagated intermediate gradient value provided as input of the backward operation T_{2L-l}.

The sequence of tasks associated with the processing of a minibatch is therefore given by $T_1, T_{1.2}^c, T_2 \ldots, T_{L-1.L}^c, T_L, T_{L+1}, T_{L+1.L+2}^c, T_{L+2}, \ldots, T_{2L-1.2L}^c, T_{2L}$, where if $l < L$ then $T_l = F_l$ corresponds to the forward task performed on the layer l, and $T_{2L-l+1} = B_l$ is the associated backward task.

During the training operation, we assume that the set of input data (typically images) is split into mini-batches of size $\mathcal{B}$. In this context, we will denote by

- $d_l = u_{F_l}$ the duration of the forward task on the layer l with a mini-batch of size $\mathcal{B}$

- $d_l = u_{B_l}$ the duration of the backward task on the layer l with a mini-batch of size $\mathcal{B}$

- a_l the size (in bytes) of the tensor $a^{(l)}$ produced by $T_l, l \leq L$ when applied to a mini-batch of size $\mathcal{B}$ and

- $a_{l'}$ the size (in bytes) of the tensor $a^{(l')} = b^{(2L-l')}$ produced by $T_{l'}, l' \geq L+1$ when applied to a mini-batch of size $\mathcal{B}$. In general we will assume that tensors $a^{(l)}$ and $b^{(l)}$ have the same size, i.e. $\forall l \leq L, a_{2L-l} = a_l$.

Let us denote by P the total number of available GPUs and let us assume (as in PipeDream) that all pairs of GPUs are connected through a direct link of capacity β. Moreover, we assume that each GPU is equipped with an available memory of size M. This memory will be used to store all data required to perform the training operation. More precisely, this memory requirements can have different origins:

- Model weights. Since we are considering model parallelism, we assume that the L layers of the network will be split across the P GPUs. If GPU P_k is in charge of layer l, then it has to store the corresponding weight denoted as W_l. As we will see it, in order to update the weights and to use consistent weights during both the forward and the backward phases, P_k will in practice store several copies of the weights. In what follows, in order not to add more weight and activation copies, we will assume that the processor in charge of a layer is in charge of processing both the forward and backward tasks associated to this layer.

- Activations. Let us now concentrate on activations $a^{(l)}$. As depicted in Figure 1, activation $a^{(l)}$ is produced by T_l and it must be kept in memory until task T_{2L-l} consumes it to produce
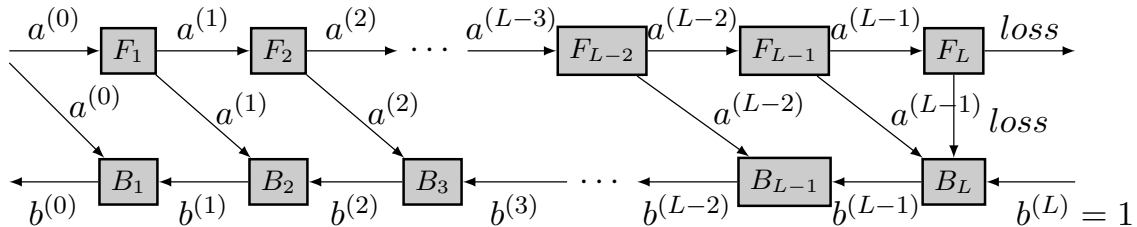

Figure 1: Graph for the Backward Propagation

5

$b^{(l)}$. Therefore, a memory of size a_l must be reserved to store an activation between tasks T_l and task T_{2L-l+1}. As we will see it, in general, to keep processors busy, we will process several batches in parallel. Therefore, several activations corresponding to layer l will be kept in memory simultaneously.

- Gradients. Let us now concentrate on gradients $b^{(l)}$. As depicted in Figure 1, the gradient $b^{(l)}$ produced by task T_{2L-l} is consumed by T_{2L-l+1} to produce $b^{(l-1)}$. Therefore, a memory of size $a_{2L-l}(= a_l)$ must be reserved to store an activation between tasks T_{2L-l} and T_{2L-l+1} to produce $b^{(l-1)}$. Thus, gradients are kept in memory for a much shorter time than activations.

- Communication buffers. When tasks T_l and T_{l+1} are assigned to different GPUs, a communication will take place, and we assume that for convenience some memory is reserved as a buffer to store $a^{(l)}$ while it is sent or received. This requires a storage of size a_l on both GPUs.

3.2 Periodic Schedules and Valid Patterns

In general, throughout this paper, we are interested in finding optimal task allocation (load balancing) and optimal periodic schedules. Both problems could be solved either simultaneously or separately *i.e.* solving the load balancing problem first and then finding an optimal schedule for it. Considering some fixed allocation, it is possible to build various schedules, but we restrict the search to periodic schedules that follow the Pattern conditions defined in Definition 1 (see Figure 2 for an example). In Definition 1, a valid pattern contains exactly one task T_l of each type l, for $l \leq 2L$. The pattern defines both the sequence of tasks that will be performed on each computing resource, and the index shifts between the tasks that are processed in the pattern. For example, if T_2 takes place before T_1 in the pattern, then T_2 must necessarily apply to a batch for which T_1 has been processed in the previous copy of the pattern. In practice, we will denote by $s(l)$ the shift associated to T_l, so that previous condition becomes $s(1) \geq s(2) + 1$. Hence, during the $i-$th copy of the pattern in the periodic schedule, T_l will operate on mini-batch $i + s(l)$.

Definition 1 (Valid Pattern). A valid pattern is defined by the following constraints:

- $\forall l \leq 2L, T_l$ is present exactly once in the pattern, on one of the GPUs and it has an exclusive access to the GPU during d_l time units.

- $\forall l \leq 2L - 1, T_{l,l+1}^c$ is present exactly once in the pattern, on one of the communication links. Its duration is 0 if T_l and T_{l+1} are located on the same resource. Otherwise, it gets an exclusive access to the link between the GPUs that process T_l and T_{l+1} during a_l/β time units.

- task shifts are valid, *i.e.* if the shift for T_l is denoted by $s(l)$, then $\forall l' > l$, if task $T_{l'}$ starts before the end of T_l, then the shift for $T_{l'}$ must satisfy $s(l') < s(l)$.

- In the pattern, on each GPU and on each communication resource, the length of the period is smaller than T

- the starting dates of all tasks and communications should be within the time range of size T

Definition 2 (1F1B). We call all periodical schedules satisfying the pattern described in Definition 1 as 1F1B schedules (1-Forward-1-Backward).
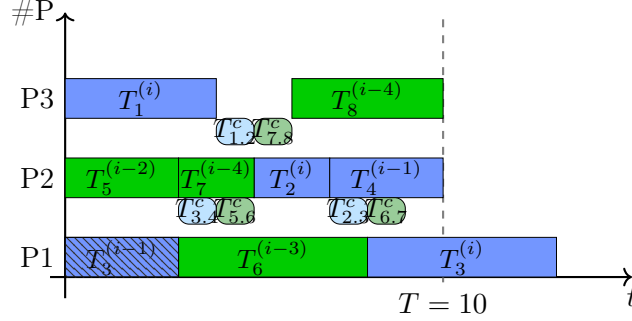
Figure 2: Example of valid pattern. The batch indices are in the superscipts near task names

Lemma 1. *We can restrict the search of optimal periodic schedules to the search of optimal patterns.*

Proof. Clearly, it is easy to build the periodic pattern from a valid periodic schedule.

Let us now suppose that we have a valid periodic pattern and let us prove that it can be turned into a valid periodic schedule. To do so, we need to specify, for each task, the index of the mini-batch on which this task operates. In order to determine these indexes, we can use a basic greedy algorithm. Let us suppose that T_{2L} processes the i-th mini-batch during a given period. Then, we can always assume that during this period, $T^c_{2L-1.2L}$ is performed on mini-batch $i+2$. Indeed let us assume that T_{2L} is performed on P_k and T_{2L-1} on $P_{k'}$, $k' \neq k$ (the case where both tasks are scheduled on the same resource is less interesting since the duration of $T^c_{2L-1.2L}$ is 0 in this case). Since there could a small shift in the periodic pattern between P_k and $P_{k'}$, it might happen that T_{2L} has already started on P_k at the time when $T^c_{2L-1.2L}$ ends. On the other hand, since this shift is always smaller than T by construction, the communicated activation during the current period will be available on P_k 2 periods later, so that P_k will indeed be able to process mini-batch $i+2$. Using the same reasoning and the same shift of two mini-batches between any two consecutive processing and communication tasks, it is possible to transform any valid periodic pattern into a valid schedule.

Note that our greedy algorithm induces a very large span for the indexes of the mini-batches performed in the same period. We will see later that this large span induces a large memory costs, but that given a periodic pattern, it is possible to build a schedule with minimal memory need (this will be done in Lemma 10).

$\square$

Similarly to PipeDream, we use the notion of 1F1B schedules from Definition 2. As mentioned earlier, we are interested only in 1F1B schedules, and therefore we look for 1F1B solutions that minimize the period T given a batch-size $\mathcal{B}$, or equivalently that maximize the number of trained images per time unit in steady-state, given by $\frac{\mathcal{B}}{T}$.

4 PipeDream

4.1 PipeDream Overview and Positioning of Present Paper

In this section, we present in detail the results of PipeDream [32], which has been proposed to automate the phase of grouping neural network layers into stages executed by different computational and communication resources. It should be noted that PipeDream offers the possibility to combine

data parallelism and model parallelism, which is an attractive feature in practice. In the present paper, we focus on the model parallelism part, but we discussed in Section 1 how to combine data parallelism and model parallelism a posteriori. With respect to model parallelism, we propose several improvements compared to PipeDream:

- **Memory.** We propose a much more precise and accurate modeling of memory constraints and we propose that these constraints be directly integrated into the process of finding the best solution, contrary to what is done in PipeDream. Indeed, PipeDream computes an efficient solution for the operations, and the only way to make it fit into memory later is to adjust the size of the mini-batches and to introduce delays. PipeDream is therefore not able to optimize the overall solution, taking into account both computation and memory requirements simultaneously. In PipeDream two-phase optimization process (i) the allocation problem is handled first without taking memory constraints into account (ii) the scheduling problem is then solved and memory constraints are considered. This approach leads to non-optimal solutions.

- **Communications.** We also propose a more sophisticated communication modeling, which solves the incorrect scheduling issues sometimes observed in [42] with high communication costs with PipeDream.

- **Shape of solutions.** A in [40], we propose to consider more general solutions, in which a processor can receive any set of layers, whereas PipeDream restricts itself to solutions in which each processor receives a set of contiguous layers. This assumption on the DNN slicing simplifies the optimization process, allowing in particular the use of dynamic programming to find the optimal solution under this constraint, but it does not allow to maximize the use of computational resources.

The general idea of PipeDream is to seek the solutions in a particular shape, in which each processor receives a set of consecutive layers. In terms of work partitioning, if we forget about data parallelism, it is essentially a 1D chain load balancing problem. Each processor is associated with the time to process the workload that corresponds to the layers assigned to it. We associate to each communication resource, assuming a complete graph between all computing resources, the time needed to push all the activations (and gradients) that it is responsible for. It is then possible, using dynamic programming, to find the best allocation since the packing problem can be solved in polynomial time under the assumption of contiguous allocations. We discuss the consequences of this assumption in Section 4.2

Once layers have been assigned to processing resources, PipeDream proposes to rely on a 1F1B scheduling scheme for the tasks. More specifically, the authors advocate the use of very regular solutions, in which each processor alternates between a forward propagation step (on the oldest batch it has received and not yet processed) and a backward propagation step (on the oldest batch it has received and not yet processed). A strong statement of [32] is that it is possible, by correctly initializing the process, i.e. by injecting the right number of mini-batches at the beginning of the training phase, to reach the steady state with the help of this simple schedule which in addition is easy to implement in practice. We will come back to this point in more detail and to the generality of this result under different models in Section 4.3.

The number of mini-batches injected in the system at the beginning is crucial. Indeed, it has a direct influence on memory requirements and on the achieved throughput. In PipeDream, the

authors impose that the model weights used for backward propagation must be the same as the weights used in the forward phase. This is a very sensible assumption and it allows us to interpret the computed gradients in a straightforward way. In the present paper, we retain the PipeDream model, but similarly to its extension PipeDream-2BW [33], we keep fewer models in memory than what is proposed in PipeDream. More specifically, in [32], memory constraints are not taken into account to obtain the allocation, and they are not even inputs of the optimization problem. Once an allocation is obtained, if memory constraints are not satisfied, then the mini-batch size is adapted to fit into the memory constraint. Finally, the code of PipeDream [33] includes some memory considerations when computing the solution, but it remains only approximative as it will be then shown by our experiments.

4.2   Contiguous vs General Allocations

In PipeDream [32], the authors restrict the search to contiguous allocations, where a set of contiguous layers is allocated to each processor. In this section, our goal is to study in two different settings the theoretical impact on the expected throughput of this constraint. In Section 4.2.1, we consider the case where we assume that processors have an infinite memory, or equivalently that we first compute the allocation and then modify the size of the mini-batch so that it fits into the memory as done in [32]. In this case, we prove that the throughput can be divided by 2 if we restrict the search to contiguous allocations. Even worse, we prove in Section 4.2.2 that if we consider the memory constraint, imposing a contiguous allocation might lead to infeasible results. Even in the case where it is possible to find a valid contiguous allocation, the ratio between the throughput achieved by the best contiguous allocation and the throughput achieved by the best general solution can be arbitrarily large.

4.2.1   Without memory constraints

As a simple starting example, we can consider the case of a chain of length 3 to be processed on 2 processors, where the computing costs are $(u_{F_i} + u_{B_i}) = (1, 2, 1)$, and where memory is large enough to not be a constraint. It is clear that the smallest period achievable by a contiguous allocation is 3: either the first or the last processor gets the layer of cost 2. On the other hand, a non-contiguous allocation allows to run the first and last layers on one processor, and the layer of cost 2 on the other processor, resulting in a period of 2 and no idle time on any processor. The overhead of the contiguous constraint is thus $\frac{3}{2}$ in this case.

The following theorem shows that the exact ratio is actually 2 in the worst case:

Theorem 1. *On any chain, the period of the best contiguous allocation is at most twice the period of the best non contiguous allocation. Furthermore, for any $k \geq 1$, there exists a chain for which the period of the best contiguous allocation is $2 - \frac{1}{k}$ times larger than the best allocation.*

Proof. To prove the first result, consider any chain C, and denote by T^* the period of the best non constrained allocation for this chain. Clearly $T^* \geq \frac{\sum_l u_{F_l} + u_{B_l}}{P}$, and $T^* \geq \max_l (u_{F_l} + u_{B_l})$. We can build a contiguous allocation with period at most $2T^*$ with a greedy Next Fit procedure: add layers to the first processor as long as the total load is below $2T^*$, move to the next processor and repeat. Since no layer has cost more than T^*, each processor except maybe the last one has load at least T^*. This shows that this procedure ends before running out of processors.

Let us now prove the second statement, with an example inspired from [8]. For any $k \geq 1$, let us set $\epsilon = \frac{1}{2k+1}$. Let $P = 2k + 1$, and let us build the chain C_k with $k + 1$ parts: the first k parts contain 4 layers with computation costs $(k, \epsilon, k - 1, \epsilon)$; the last part contains one layer of cost k, $(k - 2)(2k + 1) + 1$ layers of cost ϵ, and one layer of cost 1. Note that the total number of layers of cost ϵ is $2k + (k - 2)(2k + 1) + 1 = (k - 1)(2k + 1)$.

There exists an allocation with period $T^* = k$ for chain C_k: $k + 1$ processors process a layer of cost k, 1 processor processes a layer of cost $k - 1$ and the layer of cost 1, and $k - 1$ processors process a layer of cost $k - 1$ and $2k + 1$ layers of cost ϵ. In this allocation, no processor has any idle time.

Chain C_k contains $2k + 2$ layers with cost at least 1. On any contiguous allocation on $2k + 1$ processors, at least one processor p processes two such layers. If it processes one layer of cost k and one of cost $k - 1$, it also processes the layer of cost ϵ between them, and thus its load is at least as $2k - 1 + \epsilon$. If it processes the layer of cost 1 and the last layer of cost k, it also processes all layers of cost ϵ in between, for a total load at least $k + ((k - 2)(2k + 1) + 1)\,\epsilon + 1 = 2k - 1 + \epsilon$. This shows that there is no contiguous allocation with period $2k - 1$ or less, which concludes the proof. □

4.2.2 With memory constraints

Let us prove that the situation is worse when we explicitly take memory into account. For the sake of simplicity, we will not consider activation sizes but model weights only.

Let us first prove that sometimes, non-contiguous allocations are necessary to obtain a valid solution.

Lemma 2. *Non contiguous allocations are sometimes required in order to process training under memory constraints.*

Proof. In order to prove this result, let us consider the basic network consisting of 3 layers. For this network the time necessary to process any layer is 1 and the weights of the layers are respectively 1, 2 and 1, *i.e.* $(u_{F_i} + u_{B_i}) = (1, 1, 1)$ and $W_i = (1, 2, 1)$. In addition, let us assume that the platform is made of 2 identical GPUs whose memory is exactly 2 (remember that we assume in this section that all activation sizes are 0). Then, in any contiguous allocation on the 2 processors, layer 2 must be co-located with either layer 1 or layer 3, thus inducing a memory need of 1+2=3 that exceeds the memory limit. On the other hand, if non-contiguous allocations are allowed, then layers 1 and 3 can be co-located on one of the GPUs whereas layer 2 is alone on the other GPU, what provides a valid allocation. □

Lemma 3. *If there exist both a valid contiguous allocation and a valid non-contiguous allocation given a memory constraint, then the ratio between achieved throughputs in the non contiguous and contiguous settings can be arbitrarily large.*

Proof. In order to prove this result, let us consider the basic network depicted in Figure 3. In this network, there is a sequence of P layers with processing cost 1 and model weight $M - 1$, followed by a layer with processing cost A and model weight M and followed by P layers with processing cost A and model weight 1. In addition, let us assume that the platform is made of $P + 2$ GPUs whose memory limit is M. Here P, M, A are arbitrary values where $P \leq M$ and processing costs are comprised of both forward and backward computation times.

Then, a valid solution consists in grouping, $\forall i \leq P$ layer i and layer $P + i + 1$ on processor P_i, to dedicate processor $P + 1$ to layer $P + 1$ and to leave processor $P + 2$ idle. The required memory $M - 1 + 1$ can fit into the memory and the processing time on each ressource is $A + 1$ per mini-batch
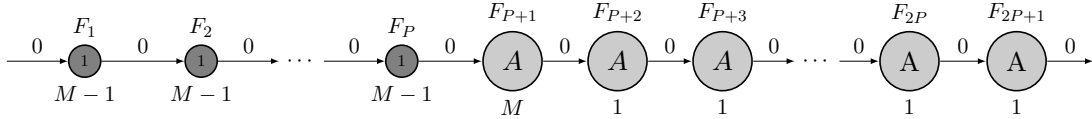
Figure 3: Bad Ratio for Contiguous Allocations and Memory Constraint

(accounting for both forward and backward phases). Then, we can build a periodic schedule whose throughput is $\frac{1}{A+1}$ in steady-state.

If we use contiguous allocation, the first $P+1$ layers must be on separate processors, because of the memory contraint. Then, the last P layers must be on the only remaining processor $P+2$, that can accommodate them since each of these layers only require 1 unit of memory. On the other hand, the load on the last processor is $P \times A$, so that the ratio between the throughputs achieved by the best solution and the best contiguous solution is given by $P\frac{A}{A+1}$, which can be made arbitrarily close to P.

□

4.3 Limitations of 1F1B Schedules

PipeDream 1F1B schedule can be suboptimal for some fixed task allocation when memory limits are taken into account, as shown in the following two examples.

In the first example (Figure 4), let us consider a simple network with only four layers where

- computational times are equal respectively to $3, 2, 2$ and 2;

- activation sizes are $a_0 = 1$, $a_1 = 2$, $a_2 = 2$ and $a_3 = 4$;

- all communication times are negligibly small;

- all weights occupy negligibly small amount of memory;

- training is done with mini-batch of size $\mathcal{B} = 1$

- 4 GPUs are available and each of them has a memory size 4.

Due to memory limit, it is not possible to store on a GPU more than 4 activations of the first layer, 2 activations of the second layer, 2 activations of the third layer and 1 activation of the last layer. In order to train such network with 1F1B strategy a period $T \geq 8$ should be selected yielding throughput of 1/8. However, as it is shown in Figure 4 there exists another periodic schedule with period $T' = 12$ that processes two forward and two backward operations of each layer during the period. Let us denote such schedules as 2F2B. This 2F2B schedule does not exceed memory budget and achieves throughput of $2/T' = 1/6$, which is larger than the one achieved by the best 1F1B schedule.

The same result holds for the general non-contiguous allocation schemes. Our second example (Figure 5) is based on another network with three layers where

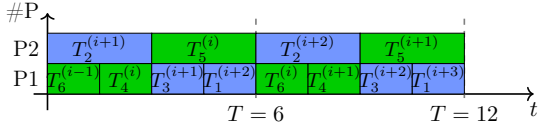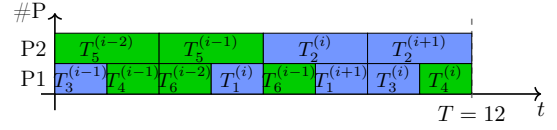- computational times are equal respectively to $1, 2$ and 1;

11
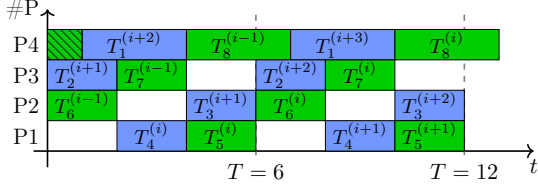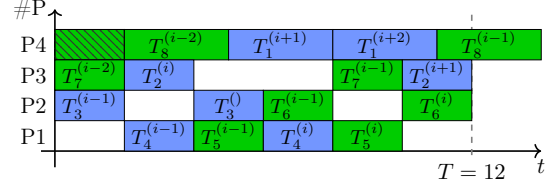
(a) 1F1B schedule

(b) 2F2B schedule

Figure 4: Non optimality of 1F1B with memory constraints: both schedules process the network (3,2,2,2) with the same throughput, but the 1F1B schedule requires more memory as it keeps 4 copies of the first layer, 3 of the second, 2 of the third and 1 of the last one, while the 2F2B schedule has lower memory consumption as it keeps 4, 2, 2 and 1 copies respectively for each layer.

(a) 1F1B schedule

(b) 2F2B schedule

Figure 5: Non optimality of 1F1B with memory constraints for non-contiguous allocations: both schedules process the network (1,2,1) with the same throughput, but the 1F1B schedule requires more memory as it keeps 3 copies of the first layer, 2 of the second and 1 of the third one, while the 2F2B schedule has lower memory consumption as it keeps 2, 2 and 1 copies respectively for each layer.

- activation sizes are $a_0 = 2$, $a_1 = 3$ and $a_2 = 2$;

- all communication times are negligibly small;

- network weights occupy negligibly small amount of memory;

- training is performed with $\mathcal{B} = 1$

- 2 processors are available and each of them has a memory size 6.

In this case, the memory limit does not allow a GPU to store more than 2 activations at the same time for the second layer and more than 3 activations stored at the same time of the first and third layer. 2F2B schedule helps to achieve the optimal throughput while the memory constraints are satisfied, whereas 1F1B schedules with optimal throughput requires higher memory for the second GPU.

5 Complexity Results

5.1 General Problem

In this section, we prove that the general problem that consist in finding an optimal periodic pattern, given a platform and a network, is NP Complete in the strong sense. This result is easy to establish
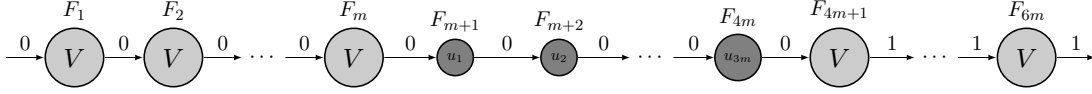
Figure 6: Network Instance for the Proof of Theorem 3

and does not even require to take memory or communication constraints into account, and only relies on the basic underlying load balancing problem.

We prove that this problem is strongly NP-hard by a reduction from the 3-partition problem [17]: given a set of integers $\{u_1, u_2, \ldots, u_{3m}\}$ such that $\sum_i u_i = mV$, is it possible to partition it into m parts $\{S_1, \ldots, S_m\}$ so that for any $j \leq m$, $|S_j| = 3$ and $\sum_{i \in S_j} u_i = V$. This problem is known to be NP-hard in the strong sense.

Given an instance of 3-partition, we consider the following instance of our problem, where the network is depicted in Figure 1 and the processing resources are defined as follows

- $L = 3m$, $P = m$

- $\forall l,\ a_l = W_l = 0$, so that memory and communications are not a concern

- $\forall l \leq L, d_{\lceil}l] = u_l, \forall l \geq L+1, d_{\lceil}l] = 0$

and the decision problem is to determine if there exists a periodic schedule with period $T = V$.

Theorem 2. *The associated decision problem of determining if there exists a periodic schedule of period at most T is strongly NP-Complete*

Proof. Let us assume that there exists a solution to the 3-Partition instance. Then, we build a pattern where each group S_i is scheduled (in any order) on a different GPU. There always exists a task ordering such that the schedule is valid. Since there is no memory issues (all sizes are set to 0), we obtain a valid periodic schedule.

Let us now assume that there exists a pattern of period T. Then, since one task cannot be split between two resources, then each GPU is allocated to different tasks for a total length smaller than T. Since the overall load is mT, the load on each GPU must be exactly T. $\qquad\square$

5.2 Fixed Allocation Problem

We just proved that the problem is in general NP-Complete in the strong sense. Let us now assume that the allocation is given, *i.e.* that we know which layer is assigned to which processor. In this context, the main problem becomes rather a scheduling problem: given a memory limit M on each GPU and task allocation on P resources, how to schedule tasks in periodical manner so that the resulting period is minimal.

We prove below in Theorem 3 that even once the resource allocation problem is solved, the problem remains NP-hard in the strong sense, what shows that both scheduling and resource allocation are difficult.

We prove again that this problem is strongly NP-hard by a reduction from the 3-partition problem [17] Given an instance of 3-partition, we consider the following instance of our problem, where the network is depicted in Figure 7 and the processing resources are defined as follows
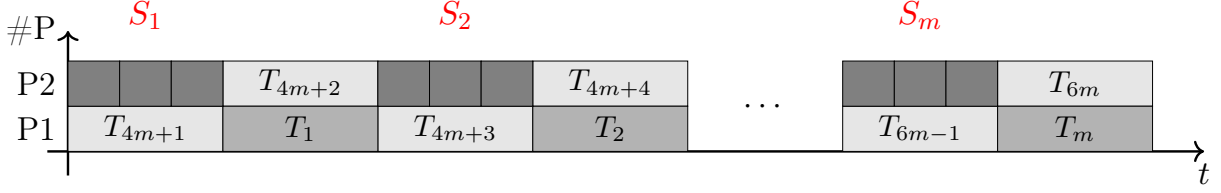
- $L = 6m$, $P = 2$, $M = m$

Figure 7: 3P Instance Schedule for the Proof of Theorem 3

- $\forall l \leq 4m$, $a_l = 0$, $\forall 4m+1 \leq l \leq 6m$, $a_l = 1$ and the bandwidth between the 2 GPUs is assumed to be infinite, so that there are no communication costs. We also assume that $\forall l, W_l = 0$.

- $\forall l \leq m$, $d_{[l]} = V$, $\forall m + 1 \leq l \leq 4m$, $d_{[l]} = u_{l-m}$, $\forall 4m+1 \leq l \leq 6m$, $d_{[l]} = V$

- all memory and computational costs for the backward phase are 0, $i.e.$ $\forall l \geq 6m + 1, d_{[l]} = 0, a_l = 0$.

- P_1 is allocated to tasks T_{4m+2}, T_{4m+4}, T_{4m+6}, ..., T_{6m-2}, T_{6m} and T_l, $\forall m + 1 \leq l \leq 4m$

- P_2 is allocated to tasks T_{4m+1}, T_{4m+3}, T_{4m+5}, ..., T_{6m-3}, T_{6m-1} and T_l, $l \leq m$

and the decision problem is to determine if there exists a periodic schedule with a period $T = 2mV$.

Theorem 3. *The associated decision problem of determining if there exists a periodic schedule of period at most $T = 2mV$ is NP-Complete in the strong sense*

Proof. Let us assume that there exists a solution to the 3-Partition instance. Then, we build a pattern where each group S_i is scheduled as depicted in Figure 7. Since $\forall l, W_l = 0$ and $\forall l \geq 6m + 1, a_l = 0$, the memory costs come from the storage of activation. Moreover, since communication costs are 0, all tasks $T_l, l \geq 4m + 1$ in Figure 7 can process the same mini-batch. The other tasks can be associated with any mini-batch number compliant with Definition 1, as activations sizes are zero, they do not affect memory. Therefore, using the schedule depicted in Figure 7, the peak memory is reached at the instant when T_{6m} is processed on P_1 and when T_{6m-1} is processed on P_2 and the value of the peaks is m on each resource. Thus, we built a schedule of optimal throughput where all resource constraints are satisfied.

Let us now assume that there exists a valid pattern of period T. Let us first prove that tasks $T_{4m+1}, \ldots, T_{6m}$ are scheduled as depicted in Figure 7. Let us suppose that $\exists j : 4m+1 \leq j \leq 6m-1$, T_j is scheduled after T_{j+1}. Then, by construction, T_j and T_{j+1} cannot process the same mini-batch. Therefore, at the time when T_j is processed, there are necessarily two activations produced by T_j in memory, the one produced by this instance of T_j and the one used by the instance of T_{j+1}, that must be kept in memory until the associated backward task is executed. Therefore, the memory need would be at least $m + 1$, $i.e.$ one for each $T_l, l \neq j$ and 2 for T_j, which is too large for the memory of a GPU. Therefore, tasks $T_{4m+1}, \ldots, T_{6m}$ are scheduled as depicted in Figure 7.

Then,

- There are exactly m holes of size V on P_2 to accommodate m tasks of size V $T_1, \ldots, T_m$, which does not induce memory needs since the activations they produce have size 0.

14

- There are exactly m holes of size V on P_1 to accommodate $3m$ tasks of size u_i $T_{m+1}, \ldots, T_{4m}$, which does not induce memory needs since the activations they produce have size 0. On the other hand, the packing on these tasks into the holes creates a solution to the initial 3-partition instance, what completes the NP-Completeness proof.

$\square$

6 Integer Linear Program

We present in this section an Integer Linear Program to find a valid pattern with minimum period length. We concentrate on scheduling issues on both computational and communication resources in Section 6.1, then we consider memory related issues in Section 6.2.

We first present the main variables used in this ILP (other variables are introduced later):

- T denotes the period considered in the 1F1B schedule;

- $z_{l,l'}$ is equal to 1 if task T_l and task $T_{l'}$ are processed on the same resource, and 0 otherwise (it is implied that $z_{l,l} = 1$ and $z_{l,l'} = z_{l',l}$);

- τ_l is the starting time of task T_l in the period ;

- $\widetilde{\tau}_l$ is the starting time in the considered period of the communication of the output of T_l.

We also use d_l to denote the duration of the task T_l, *i.e.* u_{F_l} if $l \leq L$ and u_{B_l} if $l \geq L+1$, and $\widetilde{d_l}$ to denote the time needed to communicate the activation produced by T_l, equal to $\frac{a_l}{B}$. In several places, we use a large constant K which needs to be larger than the period, for example we can use $K = \sum_l d_l + \frac{a_l}{B}$.

6.1 Communication and Computation Constraints

6.1.1 Limit on the number of resources

In order to provide a limit on the number of resources used, we introduce variable f_l which is equal to 1 if and only if task T_l is the lowest-index task processed on its resource. To this end, we consider the following set of constraints:

$$\forall l < l' < l'', \quad z_{l',l''} \geq z_{l,l'} + z_{l,l''} - 1 \tag{1}$$

$$\forall l, \quad f_l \geq 1 - \sum_{l' < l} z_{l,l'} \tag{2}$$

$$\sum_l f_l \leq P, \tag{3}$$

and we show that they are enough to obtain the following:

Lemma 4. *Constraints* (1)-(3) *ensure that at most P resources are used in the schedule.*

Proof. Constraint (1) ensures the consistency of the $z_{l,l'}$ variables: for any l, l', l'', if $z_{l,l} = 1$ and $z_{l,l''} = 1$, then $z_{l',l''} = 1$. They can thus be used to define an equivalence relation between tasks, where each class contains tasks which are processed on the same resource. Then, Constraint (2)

15

ensures that f_l is exactly 1 for the task with the smallest index among all the tasks processed on a given resource, and is trivially satisfied for all other tasks. Therefore, the sum of f_l provides the total number of allocated resources, and constraint (3) enforces that no more than P resources are used in the schedule. Reciprocally, in any valid solution that uses no more than P resources, there exists an assignment of f_l variables such that $\sum_l f_l \leq P$, *i.e.* the assignment where $f_l = 1$ for the task with the smallest index processed on the resource and 0 for all other tasks. $\square$

In addition, we consider only schedules where forward tasks T_{L-l} for $l \leq L$ are placed on the same resource as their respective backward tasks T_{L+l+1}, so we add the following equation to the Linear Program

$$\forall l < L, \quad z_{L-l,L+l+1} = 1. \tag{4}$$

6.1.2 Ordering of Computational Tasks

Let us now consider tasks that are processed on the same resource. In order to enforce that two tasks processed on the same resource cannot overlap, we introduce a set of variables $w_{l,l'}$ and the following equations, valid for all $l \neq l'$:

$$\tau_l - \tau_{l'} + K(1 - z_{l,l'} + w_{l,l'}) \geq d_{l'}, \tag{5}$$
$$\tau_{l'} - \tau_l + K(2 - z_{l,l'} - w_{l,l'}) \geq d_l, \tag{6}$$
$$w_{l,l'} \leq z_{l,l'}. \tag{7}$$

Throughout the proofs, K is used to define a condition that must be valid as soon as a boolean variable x is equal to 1. The general idea is to use $(1-x) * K$ in the equation as follows: if $x = 1$, then $(1-x)K = 0$ and the rest of the condition must be satisfied. Conversely, if $x = 0$ then $(1-x)K$ is significantly larger than the other terms and the condition is automatically satisfied, regardless of the value of the other variables. An example of the use of this technique can be found below in Lemma 5.

Lemma 5. *Constraints* (5)-(7) *ensure the following:*

- *If T_l and $T_{l'}$ are assigned to the same resource, then either $T_{l'}$ starts after the end of T_l (and $w_{l,l'} = 1$), or T_l starts after the end of $T_{l'}$ (and $w_{l,l'} = 0$).*

- *If T_l and $T_{l'}$ are not assigned to the same resource, then $w_{l,l'} = 0$.*

Proof. Let us assume that T_l and $T_{l'}$ are assigned to the same resource. Then, by definition, $z_{l,l'} = 1$ and we obtain

$$\tau_l - \tau_{l'} + Kw_{l,l'} \geq d_{l'},$$
$$\tau_{l'} - \tau_l + K(1 - w_{l,l'}) \geq d_l$$
$$\text{and } w_{l,l'} \leq 1.$$

In turn, $w_{l,l'} = 1$ implies $\tau_l - \tau_{l'} + K \geq d_{l'}$ and $\tau_{l'} \geq \tau_l + d_l$. The first constraint is always true since K is large and the second constraint implies that $T_{l'}$ starts after the end of T_l. The proof for $w_{l,l'} = 0$ is symmetric and is omitted here.

16

Then, let us assume that T_l and $T_{l'}$ are not assigned to the same resource. Then, by definition, $z_{l,l'} = 0$ and we obtain

$$\tau_l - \tau_{l'} + K(1 + w_{l,l'}) \geq d_{l'},$$
$$\tau_{l'} - \tau_l + K(2 - w_{l,l'}) \geq d_l$$
and $w_{l,l'} \leq 0$.

These three constraints are compatible since the first two are always true independently of the value of $w_{l,l'}$, by definition of K, and the last one enforces $w_{l,l'} = 0$. $\qquad\square$

6.1.3 Ordering of Communication Tasks

If tasks T_l and T_{l+1} are not processed on the same resource, a communication needs to take place for the output of T_l. We recall that $a^{(l)}$ denotes the data (either an activation or a gradient) computed by T_l and needed by T_{l+1}. In order to schedule these communications, we define a new set of variables $\widetilde{z}_{l,l'}$, which play an analogous role to $z_{l,l'}$ for communication tasks. More precisely, we want $\widetilde{z}_{l,l'} = 1$ if the communications of $a^{(l)}$ and $a^{(l')}$ share the same communication link, and $\widetilde{z}_{l,l'} = 0$ otherwise. We prove that this property is enforced by the following equations, for all $l \neq l'$:

$$\widetilde{z}_{l,l'} \geq z_{l,l'} + z_{l+1,l'+1} - z_{l,l+1} - 1 \tag{8}$$
$$\widetilde{z}_{l,l'} \geq z_{l,l'+1} + z_{l+1,l'} - z_{l,l+1} - 1, \tag{9}$$
$$\widetilde{z}_{l,l'} \leq 1 - z_{l,l+1} \tag{10}$$
$$\widetilde{z}_{l,l'} \leq 1 - z_{l',l'+1} \tag{11}$$
$$\widetilde{z}_{l,l'} \leq z_{l,l'} + z_{l+1,l'} \tag{12}$$
$$\widetilde{z}_{l,l'} \leq z_{l,l'+1} + z_{l+1,l'+1} \tag{13}$$

Proof. Constraints (10) and (11) ensure that $\widetilde{z}_{l,l'} = 0$ if any of $a^{(l)}$ or $a^{(l')}$ does not require a communication (because the corresponding tasks are processed on the same resource). In the following, we assume that both $z_{l,l+1}$ and $z_{l',l'+1}$ are 0, and we denote by P_i the processor that runs T_l and by P_j the processor which runs T_{l+1}. We consider several cases:
First case: $a^{(l)}$ and $a^{(l')}$ share the same communication link.

In that case, $T_{l'}$ must be processed either on P_i or P_j, otherwise the communication of $a^{(l')}$ occupies another link than (P_i, P_j). Therefore, constraint (12) simply becomes $\widetilde{z}_{l,l'} \leq 1$. Similarly, $T_{l'+1}$ must be processed either on P_i or on P_j, so that constraint (13) simply becomes $\widetilde{z}_{l,l'} \leq 1$.
Since $z_{l,l+1} = 0$, constraints (8) and (9) become

$$\widetilde{z}_{l,l'} \geq z_{l,l'} + z_{l+1,l'+1} - 1$$
and $\widetilde{z}_{l,l'} \geq z_{l,l'+1} + z_{l+1,l'} - 1$.

Since the communication of $a^{(l')}$ use the link between P_i and P_j, $T_{l'}$ and $T_{l'+1}$ must be processed on these processors. Hence, either $T_{l'}$ is on P_i (and $T_{l'+1}$ is on P_j) or $T_{l'}$ is on P_j (and $T_{l'+1}$ is on P_i) and therefore, one of the conditions above enforces $\widetilde{z}_{l,l'} = 1$.
Second case: $a^{(l)}$ and $a^{(l')}$ do not share the same communication link.

Let us first focus on constraint (8). We claim that both $z_{l,l'}$ and $z_{l+1,l'+1}$ can not be 1 at the same time. Indeed, this would imply that T_l and $T_{l'}$ are processed on P_i, and also that T_{l+1} and $T_{l'+1}$

17

are processed on P_j, and thus that $a^{(l)}$ and $a^{(l')}$ share the same communication link. Since $z_{l,l+1} = 0$, the right hand side of constraint (8) is at most 0. Using the same analysis for constraint (9), we prove that the first two constraints simply become $\widetilde{z}_{l,l'} \geq 0$.

We now prove by contradiction that at least one among constraints (12) and (13) enforces $\widetilde{z}_{l,l'} = 0$. Indeed, $z_{l,l'} + z_{l+1,l'} \geq 1$ implies that $T_{l'}$ is processed either on P_i or P_j, and similarly $z_{l,l'+1} + z_{l+1,l'+1} \geq 1$ implies that $T_{l'+1}$ is processed either on P_i or P_j. Since we assume that $z_{l',l'+1} = 0$, having both $z_{l,l'} + z_{l+1,l'} \geq 1$ and $z_{l,l'+1} + z_{l+1,l'+1} \geq 1$ is in contradiction with the fact that $a^{(l)}$ and $a^{(l')}$ do not use the same link.

Therefore, in this second case, the system of constraints enforces $\widetilde{z}_{l,l'} = 0$. $\qquad\square$

Finally, we can ensure a correct ordering of the communications without overlap in a way similar to Lemma 5. We introduce binary variables $\widetilde{w}_{l,l'}$ together with the following equations, for all $l \neq l'$:

$$\widetilde{\tau}_l - \widetilde{\tau}_{l'} + K(1 - \widetilde{z}_{l,l'} + \widetilde{w}_{l,l'}) \geq \widetilde{d}_{l'} \tag{14}$$

$$\widetilde{\tau}_{l'} - \widetilde{\tau}_l + K(2 - \widetilde{z}_{l,l'} - \widetilde{w}_{l,l'}) \geq \widetilde{d}_l \tag{15}$$

$$\widetilde{w}_{l,l'} \leq \widetilde{z}_{l,l'} \tag{16}$$

Lemma 6. *Constraints* (8)-(16) *ensure that:*

- $\widetilde{z}_{l,l'} = 1$ *if and only if both $a^{(l)}$ and $a^{(l')}$ need to be communicated and their communications are assigned to the same link.*

- *In that case, either the communication of $a^{(l')}$ starts after the end of the communication of $a^{(l)}$ (and $\widetilde{w}_{l,l'} = 1$), or the communication of $a^{(l)}$ starts after the end of the communication of $a^{(l')}$ (and $\widetilde{w}_{l,l'} = 0$).*

- *In the opposite case, $\widetilde{w}_{l,l'} = 0$.*

Proof. The proof is similar to the proof of Lemma 5, replacing $w_{l,l'}$ by $\widetilde{w}_{l,l'}$ and $z_{l,l'}$ by $\widetilde{z}_{l,l'}$, and is therefore omitted here. $\qquad\square$

6.1.4 Period Length

In order to obtain a valid pattern from the variables defined so far, we use without loss of generality the following conventions: the ending times of all tasks and communications are between 0 and T, and task T_1 starts at time 0:

$$\forall l, \quad 0 \leq \tau_l + d_l \leq T \tag{17}$$

$$\forall l, \quad 0 \leq \widetilde{\tau}_l + \widetilde{d}_l \leq T \tag{18}$$

$$\tau_1 = 0 \tag{19}$$

We cannot specify that all *starting* times should be non-negative: as can be seen on Figure 2, in general the patterns on different processors are not aligned to start at the same time. So in order to ensure that each resource is occupied for a duration at most T, we include the following constraints which state that the distance between the ending time and starting time of two tasks assigned to the same resource is at most T:

$$\forall l \neq l', \quad T \geq \tau_l + d_l - \tau_{l'} - K(1 - z_{l,l'}) \tag{20}$$

$$\forall l \neq l', \quad T \geq \widetilde{\tau}_l + \widetilde{d}_l - \widetilde{\tau}_{l'} - K(1 - \widetilde{z}_{l,l'}) \tag{21}$$

18

Lemma 7. *Without considering memory constraints, from any valid pattern according to Definition 1, we can obtain values for all variables T, τ_l, $z_{l,l'}$, f_l, $\widetilde{\tau}_l$, $\widetilde{z}_{l,l'}$, $w_{l,'}$ and $\widetilde{w}_{l,l'}$ which respect equations (1)-(21), and vice-versa.*

Proof. If all variables respect the constraints (1)-(21), then Lemmas 4, 5 and 6 ensure that the pattern built from the values of τ_l and $z_{l,l'}$ is a valid pattern. Furthermore, constraint (20) ensures that for any l and l' such that $z_{l,l'} = 1$, T is no smaller than $\tau_l + d_l - \tau_{l'}$ and $\tau_{l'} + d_{l'} - \tau_l$, depending on which task starts first. Since a forward task is always allocated to the same resource as the respective backward task (Constraint (4)), all used resources process at least two tasks. The same can be said for communication tasks, which ensures that T is a valid period for the constructed pattern.

Reciprocally, let us consider any valid pattern, and assign values to all the variables according to this pattern. As discussed above, this can be done in a way that respects constraints (17)-(19) without loss of generality. The above lemmas ensure that constraints (1)-(16) are satisfied. Since T is a valid period, constraints (20), (21) and (18) are satisfied for any l and l' such that $z_{l,l'} = 1$. Finally, if $z_{l,l'} = 0$, these constraints are automatically satisfied since K is large. □

6.2 Memory Constraints

In this section, we focus on the memory usage induced by the pattern described in previous section. The memory needs have different origins:

- If two successive tasks T_l and T_{l+1} are processed on the same resource, the output of T_l needs to be stored in memory until it is processed by T_{l+1}. This is addressed in Section 6.2.1.

- The main point of pipelining is that during one period, the forward task T_{L-l} and its associated backward task T_{L+l+1} do not operate on the same mini-batch. This implies that the processor in charge of these operations must store several activations produced by the forward task and not yet consumed by the corresponding backward task. This will be addressed in Section 6.2.2.

- Processors need to store the weights of the layers that they process. This will be addressed in Section 6.2.3.

- When $T_{l'-1}$ and $T_{l'}$ are not processed on the same resource, either a forward activation (if $l' < L$) or a gradient (if $l' > L$) are received by the resource in charge of $T_{l'}$ and are kept in memory until the next $T_{l'}$ is performed. Similarly, when $T_{l'}$ and $T_{l'+1}$ are not processed on the same resource, either a forward activation (if $l' < L$) or a gradient (if $l' > L$) must be sent by the resource in charge of $T_{l'}$ and will be kept in memory until the associated communication. This will be addressed in Section 6.2.4.

In order to avoid symmetries in the formulation of the Integer Linear Program, we provide a formulation based on tasks and task collocations rather than on processing resources. We will therefore compute, for each task T_l, the amount of memory required *at the instant when T_l is performed* respectively by the storage of models $M_l^{(\text{MOD})}$, by direct dependencies $M_l^{(\text{DIR})}$, by local activations $M_l^{(\text{ACT})}$ and by external activations and gradients $M_l^{(\text{EXT})}$.
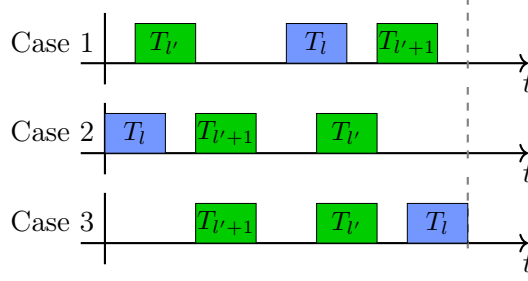
Figure 8: Different Cases for direct dependencies, where T_l, $T_{l'}$ and $T_{l'+1}$ are on the same processor.

6.2.1 Memory for Direct Dependencies

As depicted on Figure 2, the output $a^{(l')}$ of a forward task $T_{l'}$ is used twice: first by the next forward task $T_{l'+1}$, then by the corresponding backward task $T_{2L-l'-1}$. In this section, we account for the memory consumption of $a^{(l')}$ from $T_{l'}$ until $T_{l'+1}$; the memory consumption until the backward task will be accounted for in Section 6.2.2.

To evaluate $M_l^{(\mathrm{DIR})}$, let us assume that tasks l, l' and $l'+1$ are processed on the same resource. We are interested in the following event: the output produced by $T_{l'}$ occupies the memory of the resource when task T_l is performed. This event occurs in three possible situations (see Figure 8):

- $T_{l'}$ is processed before T_l, and T_l before $T_{l'+1}$;

- T_l is processed before $T_{l'+1}$, and $T_{l'+1}$ before $T_{l'}$;

- $T_{l'+1}$ is processed before $T_{l'}$, and $T_{l'}$ before T_l.

We therefore need to consider three variables: $w_{l',l}$, $w_{l,l'+1}$ and $w_{l'+1,l'}$. Since the ordering of execution on the resource is a total order, it is clear that all three variables can not be equal to one, and the list above shows that the event occurs if and only if exactly two of these variables are equal to one. We thus introduce a binary variable $o_{l,l'}$ for all l and l' with $l \neq l'$ and $l \neq l'+1$, with the following constraint:

$$o_{l,l'} \geq w_{l',l} + w_{l,l'+1} + w_{l'+1,l'} - 1 \tag{22}$$

We obtain the following lemma:

Lemma 8. *Consider any valid pattern according to Lemma 7, and assume that variables $o_{l,l'}$ satisfy Constraint (22).*

If the output produced by $T_{l'}$ is present in memory as a direct dependency when task T_l is performed, then $o_{l,l'} \geq 1$. The total amount of memory that is occupied by direct dependencies is at most $M_l^{(\mathrm{DIR})} = \sum_{l'=1}^{2L} o_{l,l'} a_{l'}$.

6.2.2 Memory Required for Local Activations

Let us now consider the memory required by the storage of local activations, between the instant when they are used by a forward task and the instant when they are consumed by the associated backward task. To achieve this, we need to analyze precisely the shifts in indices of the mini-batches that are processed during the same period.
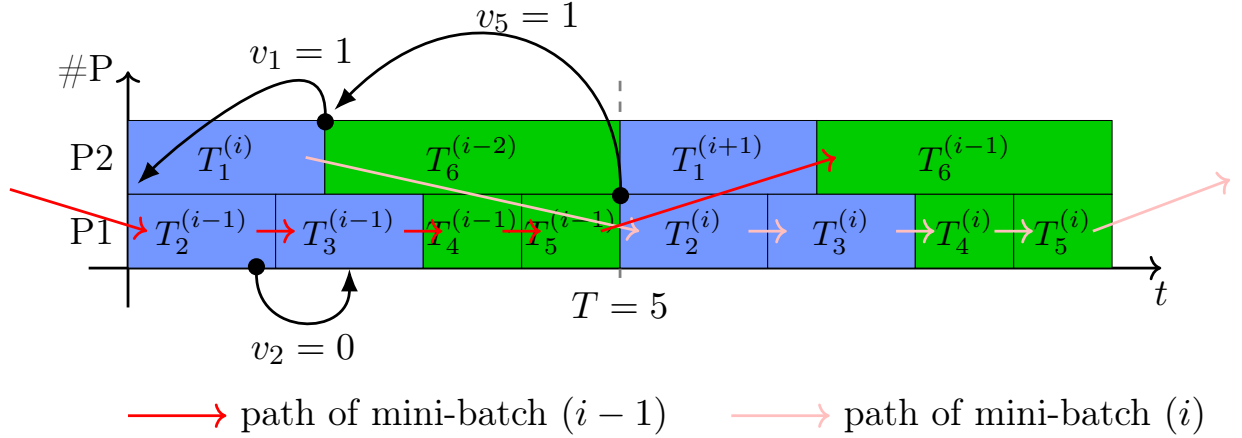
20

Figure 9: Example schedule (without communications) with paths of different mini-batches. Black arrows point from the end of a task T_l to the start of T_{l+1}, and show the value of the associated v_l variable.
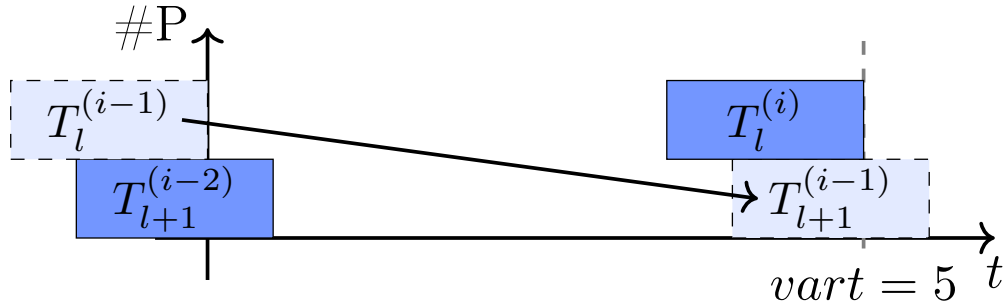

Figure 10: Example schedule where if task T_l processes mini-batch i during the period, T_{l+1} needs to process mini-batch $i-2$. Tasks indicated with dashed lines belong to different periods (the previous one for T_l, the next one for T_{l+1}).

As mentioned in the discussion of Definition 1, we observe that two consecutive tasks in the task graph, either $(T_l, T_{l.l+1}^c)$ or $(T_{l.l+1}^c, T_{l+1})$, can operate on the same mini-batch during a given period if they appear in the proper order, *i.e.* T_l before $T_{l.l+1}^c$ or $T_{l.l+1}^c$ before T_{l+1}. Otherwise, they must operate on different mini-batches. An example showing the path of different mini-batches for a simple case of two processors and no communication is shown on Figure 9.

When two successive tasks are too far apart in the pattern, it can even happen (in rare cases) that they have to process mini-batches with an index shift of two. Figure 10 shows such a case, which happens if and only if the difference between the end time of T_l and the start time of T_{l+1} is more than T.

To evaluate the shifts of indices in the pattern, we introduce new boolean variables associated to task l: v_l and v'_l are used to determine the shift between T_l and $T_{l.l+1}^c$, where v_l is 1 if the shift is at least 1, and v'_l is 1 if the shift is 2. Variables $\widetilde{v}_l$ and $\widetilde{v}'_l$ have the same meaning for the shift between

$T^c_{l.l+1}$ and T_{l+1}. For all l, we include the following constraints:

$$\tau_l + d_l - \widetilde{\tau}_l + K(1 - v_l) \geq 0 \tag{23}$$

$$\widetilde{\tau}_l - (\tau_l + d_l) + K v_l \geq 0 \tag{24}$$

$$\widetilde{\tau}_l + \widetilde{d}_l - \tau_{l+1} + K(1 - \widetilde{v}_l) \geq 0 \tag{25}$$

$$\tau_{l+1} - (\widetilde{\tau}_l + \widetilde{d}_l) + K\widetilde{v}_l \geq 0 \tag{26}$$

$$\widetilde{\tau}_l + T - (\tau_l + d_l) + 2K v'_l \geq 0 \tag{27}$$

$$\tau_l + d_l - (\widetilde{\tau}_l + T) + 2K(1 - v'_l) \geq 0 \tag{28}$$

$$\tau_{l+1} + T - (\widetilde{\tau}_l + \widetilde{d}_l) + 2K\widetilde{v}'_l \geq 0 \tag{29}$$

$$\widetilde{\tau}_l + \widetilde{d}_l - (\tau_{l+1} + T) + 2K(1 - \widetilde{v}'_l) \geq 0 \tag{30}$$

Lemma 9. *Consider any valid pattern according to Lemma 7, and assume that variables v_l, v'_l, $\widetilde{v}_l$ and $\widetilde{v}'_l$ satisfy Constraints (23)-(30).*

If i denotes the mini-batch performed by T_L, then for any $0 \leq j \leq L - 1$, the index of the mini-batch performed by T_{L-j} is at least $i + \sum_{l=L-j}^{L-1}(v_l + \widetilde{v}_l + v'_l + \widetilde{v}'_l)$, and the index of the mini-batch performed by T_{L+j+1} is at most $i - \sum_{l=L}^{L+j}(v_l + \widetilde{v}_l + v'_l + \widetilde{v}'_l)$.

Hence, the number of activations of type $a^{(L-j-1)}$ that needs to stored at the beginning of the period of this processor is $\sum_{l=L-j}^{L+j} v_l + \widetilde{v}_l + v'_l + \widetilde{v}'_l$.

Proof. Similarly to Lemma 5, we can show using the definition of K proposed above that constraints (23)-(26) ensure that for any l, $v_l = 0$ if $\tau_l + d_l \leq \widetilde{\tau}_l$, and $v_l = 1$ otherwise; likewise, $\widetilde{v}_l = 0$ if $\widetilde{\tau}_l + \widetilde{d}_l \leq \tau_{l+1}$, and $\widetilde{v}_l = 1$ otherwise. Additionally, constraints (27)-(30) ensure that if $\widetilde{\tau}_l + T < \tau_l + d_l$, then $\widetilde{v}_l = 1$, and $\widetilde{v}_l = 0$ otherwise; likewise if $\tau_{l+1} + T < \widetilde{\tau}_l + \widetilde{d}_l$, then $\widetilde{v}'_l = 1$, and $\widetilde{v}'_l = 0$ otherwise.

The claimed result can be proved by induction. For any $0 \leq j \leq L - 1$, let us set $k = L - j - 1$ and let us assume that the index of the mini-batch performed by $T_{k+1=L-j}$ is at least $I = i + \sum_{l=L-j}^{L-1} v_l + \widetilde{v}_l + v'_l + \widetilde{v}'_l$. Let us now consider $T_{k=L-j-1}$.

The ordering of $\widetilde{\tau}_k$ and τ_{k+1} can yield three possible cases:

- If $\widetilde{\tau}_k + \widetilde{d}_k \leq \tau_{k+1}$, then $\widetilde{v}_k = \widetilde{v}'_k = 0$ and both computation and communication tasks can process the same mini-batch in the same period: the communication can process mini-batch I.

- If $\widetilde{\tau}_k + \widetilde{d}_k > \tau_{k+1}$ and $\widetilde{\tau}_k + \widetilde{d}_k \leq \tau_{k+1} + T$, then $\widetilde{v}_k = 1$ and $\widetilde{v}'_k = 0$. In this case, similar to the one shown on Figure 9 with tasks T_1 and T_2, the communication cannot process mini-batch I: if it does, the result arrives too late and task T_{k+1} is not able to process mini-batch I. However the communication can process mini-batch $I + 1$.

- If $\widetilde{\tau}_k + \widetilde{d}_k > \tau_{k+1} + T$, then $\widetilde{v}_k = 1$ and $\widetilde{v}'_k = 1$. In that case, similar to the one shown on Figure 10, the communication can only process mini-batch $I + 2$.

Therefore, in all cases the index of the mini-batch processed by the communication is $I + \widetilde{v}_k + \widetilde{v}'_k$.

The same reasoning can be applied to the possible shift between the index of the mini-batch corresponding to T_k and the communication of this activation, this time involving v_k and v'_k. We thus prove that the index of the mini-batch performed by T_k during the current period is at least $I + \widetilde{v}_k + \widetilde{v}'_k + v_k + v'_k = i + \sum_{l=L-j-1}^{L-1}(v_l + \widetilde{v}_l + v'_l + \widetilde{v}'_l)$, which achieves the proof for forward tasks. The proof for backward tasks is very similar and is omitted here. $\square$

As mentioned above, we are interested in $M_l^{(\mathrm{ACT})}$, which is the memory consumed by the set of activations at the time when task T_l is performed, on the processor which computes T_l. We thus introduce integer variables $\sigma_{l',l}$, equal to the number of activations of type $a^{(l'-1)}$ stored on the processor that computes T_l, which satisfy the following constraints:

$$\forall l', l \quad \sigma_{l,l'} \leq 8L z_{l,l'} \tag{31}$$

$$\forall l', l \quad \sigma_{l,l'} \geq \sum_{m=l'}^{2L-l'} (v_m + v'_m + \widetilde{v}_m + \widetilde{v}'_m) - 8L(1 - z_{l,l'}) \tag{32}$$

Since Lemma 9 only provides the number of activations at the beginning of the period, we also need to account for the following events which may take place between the beginning of the period and instant τ_l (i) a forward task $T_{l'}$, $l' \leq L$ is computed, inducing an extra activation $a^{(l'-1)}$ in memory, and (ii) a backward task $T_{2L-l'+1}$, $l' \leq L$ is computed, removing an activation $a^{(l'-1)}$ from memory.

Lemma 10. *Consider any valid pattern according to Lemma 7, satisfying Constraints (23)-(32).*

The amount of memory occupied when task T_l is performed by activations required by future backward tasks is $M_l^{(\mathrm{ACT})}$:

$$M_l^{(\mathrm{ACT})} = a_{l-1} + a_l + \sum_{l'=1}^{L} (\sigma_{l,l'} + w_{l',l} - w_{2L-l'+1,l})a_{l'-1}.$$

Proof. We first show that $\sigma_{l,l'}$ is at least the number of replicas for a layer l' derived in Lemma 9 if l' is on the same processor as l, otherwise $\sigma_{l,l'} = 0$. Indeed, if T_l and $T_{l'}$ share the same resource then $z_{l',l} = 1$ and since the number of replicas is less than $8L$ for any layer, $\sigma_{l,l'} \geq \sum_{m=l'}^{2L-l'}(v_m + v'_m + \widetilde{v}_m + \widetilde{v}'_m)$. On the other hand, when T_l and $T_{l'}$ are on different resources $z_{l',l} = 0$ and $\sigma_{l',l} \leq 0$.

According to Lemma 9 the value $\sum_{m=l'}^{2L-l'}(v_m + v'_m + \widetilde{v}_m + \widetilde{v}'_m)$ represents the number of activations of layer $a^{(l'-1)}$ stored in the beginning of the period, but this number may vary within the period: it increases by one after each task $T_{l'}$ ($F_{l'}$) and it decreases by one after task $T_{2L-l'+1}$ ($B_{l'}$). Thus, the last term in the equation for $M_l^{(\mathrm{ACT})}$ corresponds to all activations that have been stored before task T_l and the rest represents the memory needed to perform task T_l. $\qquad \square$

6.2.3 Memory Required for the Models

As it was shown in [33], having just two models stored is enough to perform training. Besides, since the computed gradients have the same shape as model weights, they require the same amount of memory. Thus, the equivalent of three copies of the weights are necessary to perform all forward and backward computations with consistent weights. Thus

$$M_l^{(\mathrm{MOD})} = 3 \sum_{l',l' \leq L} z_{l,l'} W_{l'}. \tag{33}$$

6.2.4 Memory Buffer for Communications

Another type of memory usage on a resource P_i is the buffer memory to store activations or gradients: *incoming* ones, that were computed by another processor and then sent to P_i, or *outgoing* ones, that

were computed on P_i and need to be sent. We assume that a buffer is allocated to each of these data for the whole duration of the execution, but they are not shared between different data. We introduce binary variables $b_{l,l'}$ for all l and l', together with the following constraints:

$$b_{l,l'} \geq z_{l,l'} - z_{l,l'+1} \tag{34}$$

$$b_{l,l'} \geq z_{l,l'+1} - z_{l,l'} \tag{35}$$

Lemma 11. *Consider any valid pattern according to Lemma 7, satisfying Constraints* (34) *and* (35). *If a buffer is required for $a^{(l')}$ on the processor which computes T_l, then $b_{l,l'} = 1$. Hence, the memory reserved for buffers at the start of task T_l is at most $\mathrm{M}_l^{(\text{BUF})} = \sum_{l'=1}^{2L} b_{l,l'} a_{l'}$.*

6.2.5   Final Linear Program

We can now define the complete Linear Program for our problem: the objective is to minimize T, subject to Contraints (1)-(35), together with

$$\forall l, \quad \mathrm{M}_l^{(\text{MOD})} + \mathrm{M}_l^{(\text{ACT})} + \mathrm{M}_l^{(\text{DIR})} + \mathrm{M}_l^{(\text{BUF})} \leq M \tag{36}$$
$$\forall l, \quad T, \tau_l, \widetilde{\tau}_l \in \mathbb{R}$$
$$\forall l, l' \quad \sigma_{l,l'} \in \mathbb{Z}$$
$$\forall l, \quad f_l, v_l, v'_l, \widetilde{v}_l, \widetilde{v}'_l \in \{0,1\}$$
$$\forall l, l', \quad z_{l,l'}, w_{l,l'}, \widetilde{z}_{l,l'}, \widetilde{w}_{l,l'}, o_{l,l'}, b_{l,l'} \in \{0,1\}$$

Theorem 4. *The optimal solution of the above Integer Linear Program provides a valid pattern with minimum period, among those whose memory usage is at most M.*

7   Experimental Results

In this section, we present simulation results obtained for different state-of-the-art `ResNet` neural networks of size 18, 34 and 50, which are widely used for a large range of tasks. In order to perform these simulations, we first perform the profiling of the neural networks to measure the durations and memory costs of the different operations involved in the training. As mentioned in Section 3, this work only considers networks in the shape of adjoint chains as depicted in Figure 1. In the case of `ResNet` networks, a simple linearization approach is enough to transform the neural network computational graphs into chains, by applying a greedy procedure to obtain minimal groups of operations. In each group, the set of predecessors outside the group and the set of successors outside the group are disjoint. Overall, such groups form a sequence of operations with a straightforward order of execution. This approach was used as well in [5].

We implemented the ILP from Section 6 using the CPLEX solver [34]. In all our experiments, the execution time was limited to one hour. In case there is still a gap after one hour, we keep the best current solution computed by the solver. For `Resnet`-18 up to `Resnet`-50, the solutions produced were of very good quality (see the discussion below), though the solver was unable to prove its optimality. The results obtained are therefore heuristic in nature.

This time limit is reasonable, because the computed solution can be used during the entire training phase associated with a given image and mini-batch sizes, a given computing platform and a given network. It is common for the training to last several hours/days on a parallel platform, which makes this approach acceptable.
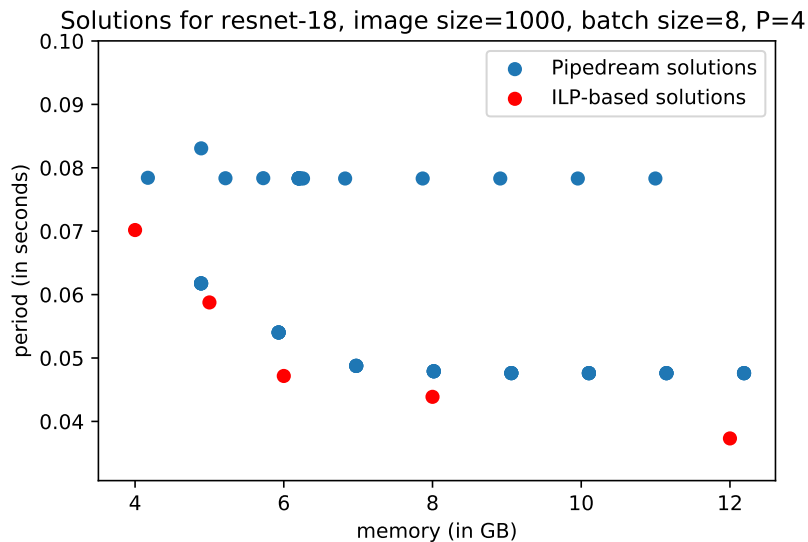
Figure 11: Results with `ResNet`-18 and 4 processors

To evaluate the quality of the solutions produced by the integer linear program, we compare the results obtained with those of PipeDream [32], which is the state of the art solution for pipelined model parallelism. In practice, as explained in Section 4, PipeDream takes as input the memory limit and the characteristics of the platform, computes the number of batches to be inserted in the pipeline, called NOAM, and finds a partition of the network that will be used for model parallelism. PipeDream then uses a greedy 1F1B strategy to schedule tasks. In practice, as we have pointed out, since the memory model used to compute the partitioning is over simplified, the calculated NOAM value is generally not optimal either.

Nevertheless, despite these limitations, PipeDream can be used to produce a large number of solutions from which one can then build valid solutions that fulfill the memory constraints. This is the approach we use in the following experiments. For a fairly large number of possible memory targets and possible NOAM values, we produce the allocations computed by PipeDream, simulate the execution of the eager scheduling strategy for 500 batches, and we evaluate a posteriori the actual memory consumption and the period (the inverse of the asymptotic throughput) that can be obtained. We thus obtain a set of (memory, period) pairs that correspond to feasible solutions. Our observations indicate that the solution produced by PipeDream generally consumes much more memory than the target value. Nevertheless, since the execution time of PipeDream is small, obtaining a set of good valid solutions through this "exhaustive" approach is still practical. In Figures 11 to 16, blue dots correspond to the actual memory consumption and observed period of solutions computed with this approach.

Figures 11 to 16 correspond respectively to the networks `ResNet`-18 (depth 18, with a batch size of 8 and image size 1000), `ResNet`-34 (depth 34, with a batch size of 32 and image size 224) and `ResNet`-50 (depth 50, with a batch size of 8 and image size 224), for the case of 4 or 8 GPUs. The red dots correspond to the best solutions found by CPLEX for our ILP after one hour, for different values of the memory limit.

First of all, it can be observed that the solutions returned by our ILP are almost always better
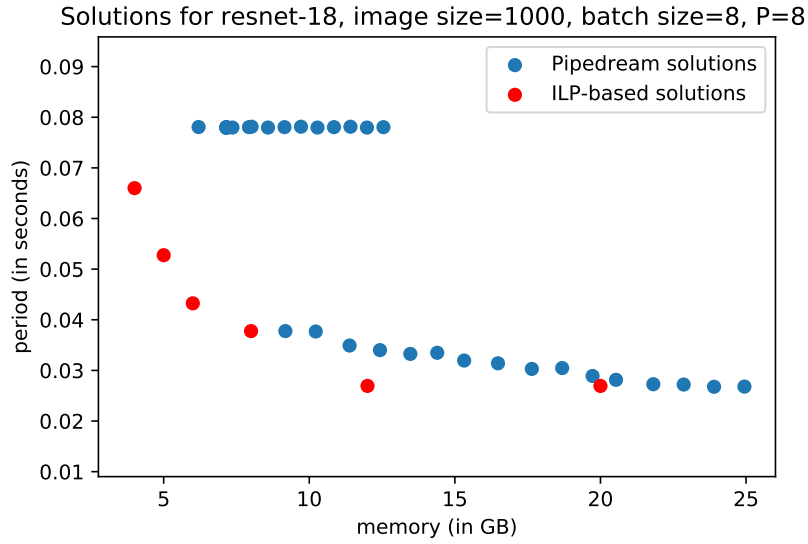
25

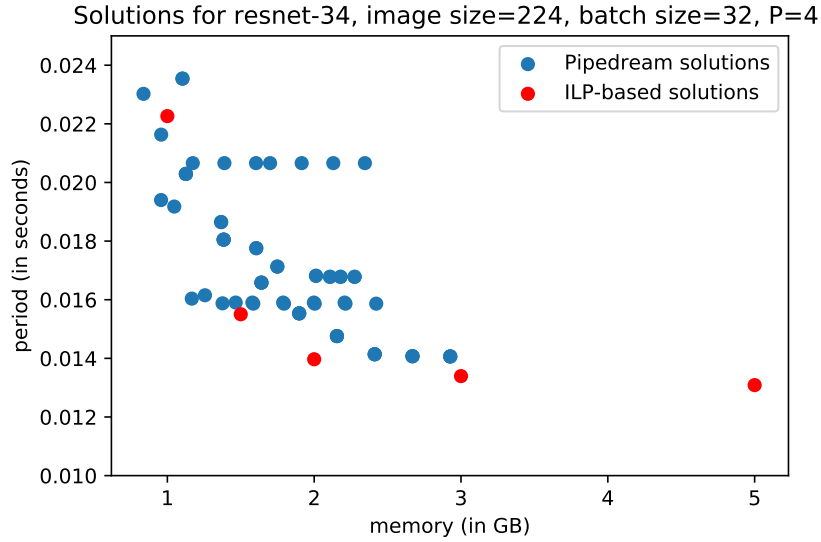Figure 12: Results with `ResNet`-18 and 8 processors

Figure 13: Results with `ResNet`-34 and 4 processors

than the solutions returned by the exhaustive approach based on PipeDream, even if optimality cannot be guaranteed. The only exception can be seen on Figure 13, where for instance in the case of a memory size of 1 GB, PipeDream finds solutions that strictly dominate the one returned by the linear program. It can also be observed that the ILP is able to find better solutions both in cases where memory is scarce (Figures 12 and 16) and where memory is abundant (Figures 11 and 14). When memory is abundant, the ability of the ILP to use non-contiguous partitionings of the networks allows to use this abundant memory to achieve better load-balancing. When memory is
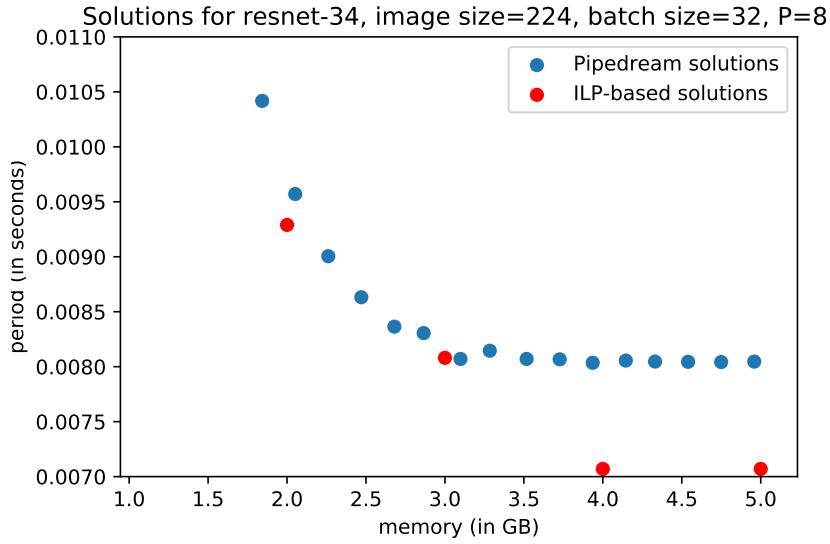
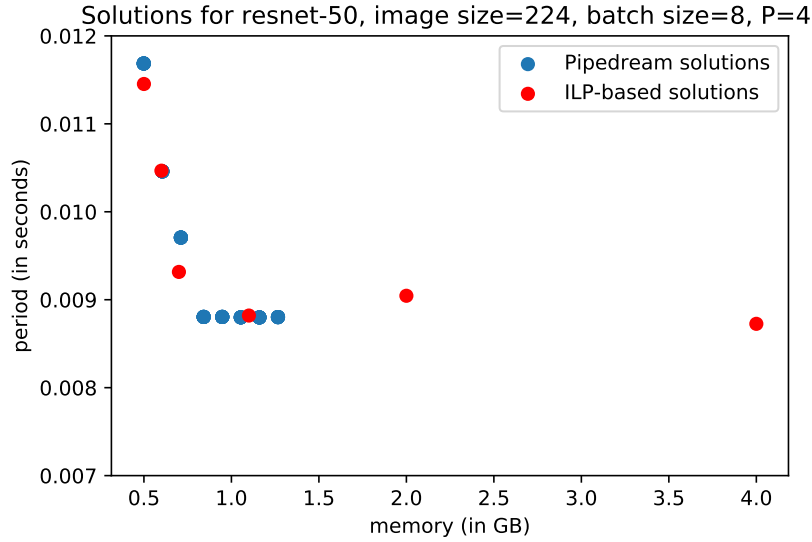Figure 14: Results with `ResNet`-34 and 8 processors

Figure 15: Results with `ResNet`-50 and 4 processors

scarce, non-contiguous partitionings are no longer beneficial, but the precise scheduling formulation of the ILP still allows to obtain better solutions. The solutions produced by the ILP are therefore generally of very high quality when the size of the network is not too large. On larger networks such as `ResNet`-101 or `DenseNet`-121 (of respective depths 101 and 121), one hour of execution is sometimes not enough for the ILP to find integral solutions of good quality. In this case, it would be necessary to consider other approaches, using fractional relaxations of the linear program or decoupling partition and scheduling phases. We leave these ideas for future works.
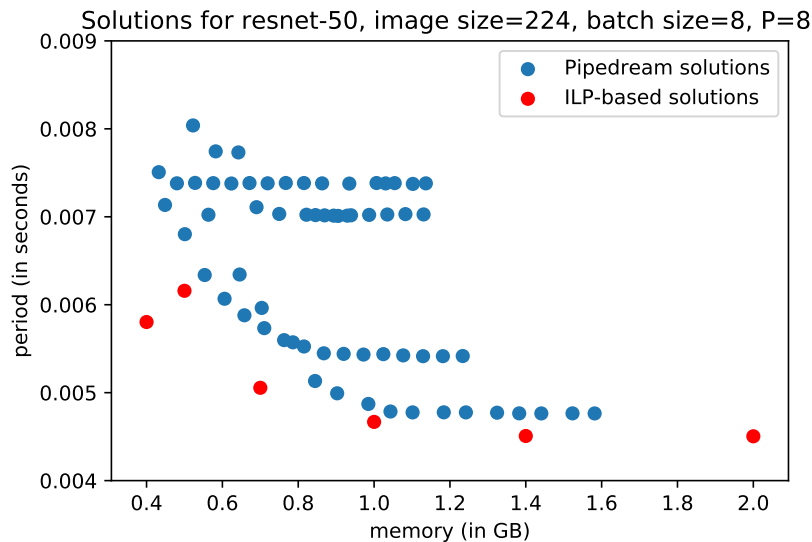
Figure 16: Results with `ResNet`-50 and 8 processors

8   Conclusion

In this paper, we consider the possibility of applying model parallelism, which is an attractive parallelization strategy that allows in particular not to replicate all the weights of the network on all the computation resources. Following the ideas proposed in PipeDream [32] we propose to combine pipelining and model parallelism, which allows to obtain a better resource utilization. Then, model parallelism can be combined with data parallelism to improve scalability.

Nevertheless, the combination of pipelining and model parallelism requires to store more activations at the nodes, which in turn causes memory consumption problems. We propose a very fine analysis of the memory costs induced by this combination. We use this modeling first to establish the computational complexity of the problems related to memory constrained pipelined model parallelism. This model also allows us to find a partition of the network that explicitly takes memory costs into account, contrary to what is done in PipeDream. We show that it is possible to formalize the problem of memory constrained throughput optimization as an integer linear program.

Through experiments on medium size networks (`Resnet`-18 to `ResNet`-50), we prove that the ILP is able to compute in reasonable time solutions that are better than those computed by PipeDream, by both providing good partitions of the networks and good scheduling strategies. Nevertheless, the computing cost induced by the integer programming approach becomes too large for very deep networks, and therefore, new heuristic solutions are required in this case, what opens interesting perspectives to this work.

References

[1] Ngraph compiler stack, 2018. http://ngraph.nervanasys.com/index.html/.

[2] Periodic checkpointing in pytorch, 2018. https://pytorch.org/docs/stable/checkpoint.html.

[3] Rotor. `https://gitlab.inria.fr/hiepacs/rotor`, 2019.

[4] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 265–283.

[5] BEAUMONT, O., EYRAUD-DUBOIS, L., HERRMANN, J., JOLY, A., AND SHILOVA, A. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. Research Report RR-9302, Inria Bordeaux Sud-Ouest, Nov. 2019.

[6] BEAUMONT, O., EYRAUD-DUBOIS, L., AND SHILOVA, A. Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training. In *Proceeding of EuroPar 2020* (2020).

[7] BEAUMONT, O., HERRMANN, J., PALLEZ, G., AND SHILOVA, A. Optimal Memory-aware Backpropagation of Deep Join Networks. Research Report RR-9273, Inria, May 2019.

[8] BOYAR, J., EPSTEIN, L., AND LEVIN, A. Tight results for next fit and worst fit with resource augmentation. *Theoretical Computer Science 411*, 26 (2010), 2572 – 2580.

[9] CHANG, B., MENG, L., HABER, E., RUTHOTTO, L., BEGERT, D., AND HOLTHAM, E. Reversible architectures for arbitrarily deep residual neural networks. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[10] CHEN, C.-C., YANG, C.-L., AND CHENG, H.-Y. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* (2018).

[11] CHEN, T., XU, B., ZHANG, C., AND GUESTRIN, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[12] CHU, C.-H., KOUSHA, P., AWAN, A. A., KHORASSANI, K. S., SUBRAMONI, H., AND PANDA, D. K. Nv-group: link-efficient reduction for distributed deep learning on modern dense gpu systems. In *Proceedings of the 34th ACM International Conference on Supercomputing* (2020), pp. 1–12.

[13] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., SENIOR, A., TUCKER, P., YANG, K., LE, Q. V., ET AL. Large scale distributed deep networks. In *Advances in neural information processing systems* (2012), pp. 1223–1231.

[14] DRYDEN, N., MARUYAMA, N., BENSON, T., MOON, T., SNIR, M., AND VAN ESSEN, B. Improving strong-scaling of cnn training by exploiting finer-grained parallelism. In *IEEE International Parallel and Distributed Processing Symposium* (2019), IEEE Press.

[15] DRYDEN, N., MARUYAMA, N., MOON, T., BENSON, T., SNIR, M., AND VAN ESSEN, B. Channel and filter parallelism for large-scale cnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), ACM, p. 10.

[16] FENG, J., AND HUANG, D. Optimal gradient checkpoint search for arbitrary computation graphs, 2018.

[17] GAREY, M. R., AND JOHNSON, D. S. *Computers and intractability*, vol. 174. freeman San Francisco, 1979.

[18] GLOROT, X., AND BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), pp. 249–256.

[19] GOMEZ, A. N., REN, M., URTASUN, R., AND GROSSE, R. B. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems* (2017), pp. 2214–2224.

[20] GOYAL, P., DOLLÁR, P., GIRSHICK, R., NOORDHUIS, P., WESOLOWSKI, L., KYROLA, A., TULLOCH, A., JIA, Y., AND HE, K. Accurate, large minibatch sgd: Training imagenet in 1 hour.

[21] GRUSLYS, A., MUNOS, R., DANIHELKA, I., LANCTOT, M., AND GRAVES, A. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems* (2016), pp. 4125–4133.

[22] HAN, S., MAO, H., AND DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[23] HEMENWAY, R. High bandwidth, low latency, burst-mode optical interconnect for high performance computing systems. In *Conference on Lasers and Electro-Optics, 2004. (CLEO).* (May 2004), vol. 1, pp. 4 pp. vol.1–.

[24] HOWARD, A. G., ZHU, M., CHEN, B., KALENICHENKO, D., WANG, W., WEYAND, T., ANDREETTO, M., AND ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[25] HUANG, Y., CHENG, Y., BAPNA, A., FIRAT, O., CHEN, D., CHEN, M., LEE, H., NGIAM, J., LE, Q. V., WU, Y., ET AL. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems* (2019), pp. 103–112.

[26] HUBARA, I., COURBARIAUX, M., SOUDRY, D., EL-YANIV, R., AND BENGIO, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research 18*, 1 (2017), 6869–6898.

[27] JAIN, P., JAIN, A., NRUSIMHA, A., GHOLAMI, A., ABBEEL, P., KEUTZER, K., STOICA, I., AND GONZALEZ, J. E. Checkmate: Breaking the memory wall with optimal tensor rematerialization, 2019.

[28] KUKREJA, N., SHILOVA, A., BEAUMONT, O., HUCKELHEIM, J., FERRIER, N., HOVLAND, P., AND GORMAN, G. Training on the edge: The why and the how. In *1st Workshop on Parallel AI and Systems for the Edge, Rio de Janeiro, Brazil* (2019).

[29] KUMAR, R., PUROHIT, M., SVITKINA, Z., VEE, E., AND WANG, J. Efficient rematerialization for deep networks. In *Advances in Neural Information Processing Systems* (2019), pp. 15146–15155.

[30] KUSUMOTO, M., INOUE, T., WATANABE, G., AKIBA, T., AND KOYAMA, M. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *arXiv preprint arXiv:1905.11722* (2019).

[31] LIU, J., , YU, W., WU, J., BUNTINAS, D., , PANDA, D. K., AND WYCKOFF, P. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro 24*, 1 (Jan 2004), 42–51.

[32] NARAYANAN, D., HARLAP, A., PHANISHAYEE, A., SESHADRI, V., DEVANUR, N. R., GANGER, G. R., GIBBONS, P. B., AND ZAHARIA, M. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 1–15.

[33] NARAYANAN, D., PHANISHAYEE, A., SHI, K., CHEN, X., AND ZAHARIA, M. Memory-efficient pipeline-parallel dnn training. *arXiv preprint arXiv:2006.09503* (2020).

[34] NICKEL, S., STEINHARDT, C., SCHLENKER, H., BURKART, W., AND REUTER-OPPERMANN, M. Ibm ilog cplex optimization studio. In *Angewandte Optimierung mit IBM ILOG CPLEX Optimization Studio.* Springer, 2020, pp. 9–23.

[35] PAINE, T., JIN, H., YANG, J., LIN, Z., AND HUANG, T. Gpu asynchronous stochastic gradient descent to speed up neural network training. *arXiv preprint arXiv:1312.6186* (2013).

[36] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch, 2017.

[37] RASTEGARI, M., ORDONEZ, V., REDMON, J., AND FARHADI, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision* (2016), Springer, pp. 525–542.

[38] RHU, M., GIMELSHEIN, N., CLEMONS, J., ZULFIQAR, A., AND KECKLER, S. W. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (2016), IEEE Press, p. 18.

[39] S B, S., GARG, A., AND KULKARNI, P. Dynamic memory management for gpu-based training of deep neural networks. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016), IEEE Press.

[40] TARNAWSKI, J., PHANISHAYEE, A., DEVANUR, N. R., MAHAJAN, D., AND PARAVECINO, F. N. Efficient algorithms for device placement of dnn graph operators. *arXiv preprint arXiv:2006.16423* (2020).

[41] YOU, Y., ZHANG, Z., DEMMEL, J., KEUTZER, K., AND HSIEH, C.-J. Imagenet training in 24 minutes.

[42] ZHAN, J., AND ZHANG, J. Pipe-torch: Pipeline-based distributed deep learning in a gpu cluster with heterogeneous networking. In *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)* (2019), IEEE, pp. 55–60.

[43] Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 6848–6856.

[44] Zinkevich, M., Weimer, M., Li, L., and Smola, A. J. Parallelized stochastic gradient descent. In *Advances in neural information processing systems* (2010), pp. 2595–2603.