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
Université de Bordeaux
Bordeaux, France

February 18, 2021

Abstract

The training phase in Deep Neural Networks has become an important source of computing resource usage and the resulting volume of computation makes it crucial to perform efficiently on parallel architectures. Data parallelism is the most widely used method, but it requires to replicate the network weights on all processors, and to perform collective communications of the network weights. In this context, model parallelism is an attractive alternative, in which the different layers of the network are distributed over the computing processors. Indeed, it is expected to better distribute weights (to cope with memory problems) and it eliminates the need for large collective communications since only forward activations are communicated. However, to be efficient, it must be combined with a pipelined approach, which in turn induces new memory costs. In this paper, our goal is to formalize pipelined model parallelism as a scheduling problem, to establish its complexity, and to analyze the consequences of the assumptions that are typically performed in practical solutions such as Pipedream.

1 Introduction

Deep Neural Network (DNN) training is a long and memory-intensive operation. Indeed, DNN training requires performing numerous forward and backward computations, each on a subset of input data called a *batch*. In turn, each forward and backward phases involve complex data dependences and induce memory issues. In practice, parallel training is performed both on small groups of GPU machines and on large HPC infrastructures [19], especially because HPC machines offer high-bandwidth and low-latency networks [14, 5].

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 [1] or PyTorch [17]. At a larger scale, the best known approach to parallel DNN training is the so-called data parallel approach. Using data parallelism [21], the model weights are replicated on all participating nodes. Then, different mini-batches are trained in parallel on different nodes: all participating nodes execute forward and backward phases in parallel, and thus all compute gradients for all weights in the network. Synchronization between the nodes takes place at the end of the backward step, and all gradients are collected and aggregated through collective communications. The above approach is possible as long as two conditions are fulfilled: (i) the communication network infrastructure must be able to support the collective communications of the weights without inducing too much idle time and (ii) each participating node must be able to store all network (model) weights and activations corresponding to the processing of a mini-batch.

In many cases deep and heavy models bring better prediction quality, but they may induce memory issues, which makes the training impossible. Several approaches were proposed to deal with this problem. In general, the memory consumption during the training phase is composed of two main parts [10]: the storage of forward activations (i.e. the outputs of all internal operations of the neural network) until the associated backward operation and the storage of the network parameters (weights). 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, known as model parallelism, has been advocated in many papers [6, 11, 15, 20]. Each 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. Unfortunately, if batches are processed in sequence, model parallelism can actually reduce memory requirements, but not accelerate computations because of the shape of data dependencies imposed by back-propagation, as shown in [11, 15]. To obtain some speedup using this approach, it is necessary to process several batches in parallel, using a pipelined approach. As we will see, in turn, processing several mini-batches simultaneously induces extra memory requirements.

All known solutions for pipelined model parallelism [15, 16] rely on a certain number of assumptions, that make the problem tractable and derive practical solutions. In particular, they only consider (i) *contiguous* allocations, where each processor is assigned a contiguous set of layers from the network and (ii) 1-*periodic* greedy schedules, where all processors alternate between forward and backward computations. In this paper, our goal is to establish the complexity of both resource allocation and scheduling problems and to show that from a theoretical perspective, allowing more general solutions (non-contiguous allocations and k-periodic complex schedules with $k > 1$) provides significant improvement in terms of throughput. 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. We establish several complexity results for the search of the optimal allocation of layers to resources in Section 4 and consider general periodic patterns in Section 5 and non-contiguous allocations in Section 6. Finally, conclusions and perspectives of this work are proposed in Section 7.

2 Related Works

To reduce the memory requirements related to the storage of forward activations, several approaches have been proposed: re-materialization, based on discarding and recomputing activations on demand [12, 13] or offloading [3], based on moving 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 when needed. Re-materialization is being increasingly employed to reduce memory usage and or practical use, an implementation [1] of re-materialization based on [2] has been proposed for PyTorch.

Another way of saving memory is to distribute memory load over multiple processors. In this way the authors of [18] managed to implement data parallelism capable of training models with trillion parameters using distributed cache mechanisms. Domain decomposition or spatial parallelism techniques can be used to limit the memory needed to store the forward activations. In [7], dividing large images into smaller ones makes it possible to train the network in parallel on the smaller images and a similar strategy has been proposed for channel and filter parallelism in [8].

Many papers [11, 16] have recently explored the use of model parallelism, following the seminal contributions of [11] and [15]. In [20], the authors observe that the scheduling strategy proposed in Pipedream is not satisfactory to take communication costs into account and the number of models to be kept in memory has been improved in [16]. Performing pipelined model parallelism efficiently requires to solve two issues: an efficient allocation of layers of the network to the processors, and a schedule describing how to perform the corresponding operations over time. Most of the literature on this question [11, 15, 16, 20] solves these problems separately, and focuses largely on the first one. However, the actual memory usage depends strongly on the actual schedule, thus these solutions typically require to reduce the throughput to make sure that the

[1] `https://gitlab.inria.fr/hiepacs/rotor`

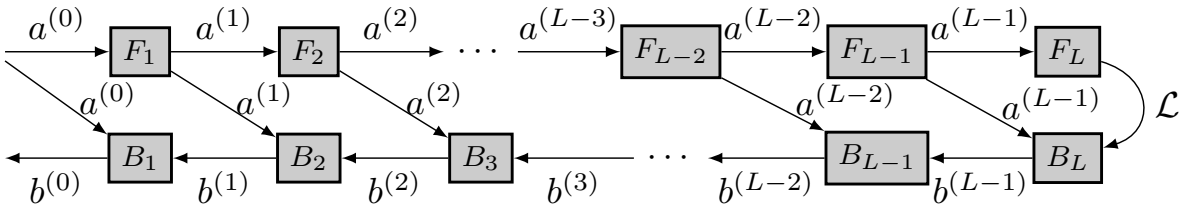


Figure 1: Dependency DAG for Forward-Backward Propagation

data fits in the processor memory.

3 Model and Notations

3.1 Notations

Like in the papers mentioned above [15, 16, 20], we consider linear (or linearized) DNNs, in which each forward operation depends only on the result of the previous operation, so that the network is a chain of L layers (see Figure 1). Each layer l, $1 \leq l \leq L$ is associated both to a forward operation F_l and a backward operation B_l (see Figure 1). During training, the input activation $a^{(0)}$ goes through all the forward operations to compute a prediction, the quality of which is estimated by a *loss* value $\mathcal{L}$. Then, the parameter weights of all layers have to be updated according to their effect on the loss, given by the partial derivative $\frac{\partial \mathcal{L}}{\partial a^{(l)}}$, where the updates are performed by an *optimizer*, following some predefined strategy. Overall, data dependencies are depicted in Figure 1.

We denote by $a^{(l)}$ the activation tensor output of $F_l, l \leq L$ and by $b^{(l)} = \frac{\partial \mathcal{L}}{\partial a^{(l)}}$ the back-propagated intermediate value provided as input of the backward operation B_l. In the following, we use the following notations:

- u_{F_l} denotes the duration of the forward task on the layer l;
- u_{B_l} denotes the duration of the backward task on the layer l;
- W_l denotes the memory occupation of the parameter weights for layer l;
- a_l denotes the memory occupation of the activation $a^{(l)}$ produced by F_l;
- a_l also corresponds to the memory occupation of the gradient $b^{(l)}$ produced by B_{l+1}, as each gradient has the same size as the activation with respect to which it is calculated.

The goal of model parallelism is to distribute the layers of the DNN onto P computing resources (typically GPUs denoted as processors in the rest of the paper) with limited memory M, so that each processor is in charge of a subset of the layers. The input activation thus goes through all processors to compute the loss $\mathcal{L}$, and is then back-propagated through all layers in reverse order to compute the corresponding gradients and update the weights. To avoid idle times, these computations are performed in a *pipelined* way (see Figure 2): the GPU in charge of layer l may compute several forward operations F_l before processing the first backward B_l, so that it could stay busy even while waiting for $b^{(l)}$ to be computed by the other GPUs.

Throughout this paper, we are interested in finding efficient task allocations and schedules. To help navigate the different concepts that we use, we introduce some terminology. We define the input DNN as a chain of *layers* (typically convolutional or dense layers), which are the basic operations that need to be computed and a *partitioning* $\mathcal{P}$ of this chain is a collection of *stages*, where each stage contains a contiguous set of layers. An *allocation* is an assignment of stages to the processors. An allocation is said to be *contiguous* if each processor is assigned a single stage, and by extension a partitioning is contiguous if it contains at

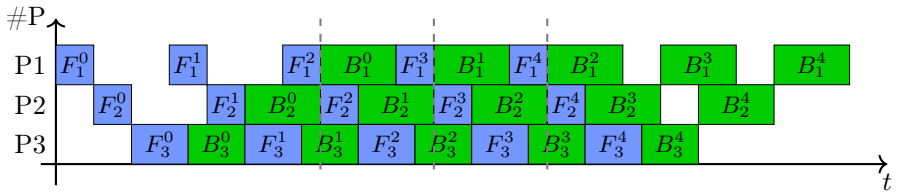


Figure 2: Pipelined schedule for 3 layers, 3 processors, and 4 iterations. The superscripts indicate iteration numbers. Period is highlighted with dashed lines.

(a) 1-periodic pattern for the schedule of Figure 2. (b) 2-periodic pattern for 3 layers where $u_F = u_B = 1$.

Figure 3: Examples of periodic patterns. The superscripts indicate shift values.

most P stages. To estimate memory requirements we also introduce *groups* of stages, where each group is a set of stages which are contiguous with respect to the ordering of the chain. Let us remark that for a fixed allocation, the results of this paper also hold true when taking communications into account. Indeed, each communication between layer l on processor p and layer $l+1$ on processor p' can be represented as an additional computation layer. It involves sending some activation $a^{(l)}$ between F_l on p and F_{l+1} on p', and a gradient $b^{(l)}$ between B_{l+1} on p' and B_l on p, for a total time of $\frac{2a_l}{\beta}$, where β is a bandwidth. Therefore, we can transform an allocation on P processors with communication costs into a partitioning on $2P-1$ resources, without communications costs.

A *schedule* $\mathcal{S}$ of a given allocation specifies the timings of all compute operations. In order to keep the description of schedules compact, we actually focus on periodic schedules. A schedule is periodic if it consists in the repetition of a pattern, and more precisely k-periodic if the pattern contains each computation layer exactly k times. We consider k-periodic patterns of period T, which specify for each operation (forward and backward): the processor in charge of it, a starting time t, and an index shift h. This pattern is to be repeated indefinitely: in the j-th period, this operation starts at time $jT+t$ and processes the batch number $jk+h$. By convention the shift of the first B_1 operation of the pattern is always 0, so that if in some period this B_1 processes batch index i, an operation with shift h processes batch index $i+h$. A pattern is valid if the schedule obtained in this way is valid, *i.e.* fulfills the dependencies of Figure 1. Figure 3a shows an example of the 1-periodic pattern associated with the schedule of Figure 2, and Figure 3b shows a 2-periodic pattern.

3.2 Memory constraints

In addition to enforcing data dependencies, we need to ensure that the schedules fit into the memory capacity M of the processors. As already noted, during the training phase, there are two main sources of memory usage: parameter weights, and forward activations. As discussed in [16], it is sufficient to keep two versions of the parameter weights. Moreover, as discussed in [18], a certain number of additional copies of the model, for gradients and optimizer states, are required. Their number only depends on the choice of the optimizer and not on the allocation or on the schedule. Overall, we denote with W_l the overall memory load induced by assigning layer l to a processor. On the other hand, with pipelined executions, several forward activations of a given layer l need to be stored in memory at the same time, and this depends on the particular schedule. For instance, in the case of Figure 2, F_1^2, F_1^3 and F_1^4 simultaneously reside in memory before B_1^2 releases F_1^2.

For a schedule $\mathcal{S}$, we define the *number of concurrent activations* (NCA) of layer l as the maximum number of activations $a^{(l)}$ that are stored at any point in time. For a general schedule, this can be expressed as $\text{NCA}_l = \max_t \#F_l(t' < t) - \#B_l(t' < t)$, where $\#F_l(t' < t)$ counts the number of F_l operations performed until time t. For a k-periodic schedule $\mathcal{S}$, NCA_l can be computed from the values of the shifts: for any F_l whose shift is h, if the preceding B_l has shift h', then the number of concurrent activations just after this forward operation is $h - h'$. The value of NCA_l for $\mathcal{S}$ is thus the maximum value of $h - h'$ over all forward operations F_l. As an example, in the pattern of Figure 3a, $\text{NCA}_1 = 3$ and $\text{NCA}_2 = 2$ (here it is necessary to duplicate the pattern to find the preceding B_2), while for the pattern of Figure 3b, $\text{NCA}_1 = \text{NCA}_2 = 2$. Given a schedule $\mathcal{S}$, if a processor p processes a set of layers L_p, its memory usage is given by $M^{\mathcal{S}}(p) = \sum_{l \in L_p} W_l + \text{NCA}_l a_l$.

Formal optimization problem We can now formally define the scheduling problem for model parallelism. We are given P processors with memory M and L layers with forward and backward computation times u_{F_l} and u_{B_l}, parameter occupation W_l and activation sizes a_l. A solution is represented by an allocation and a corresponding valid k-periodic schedule $\mathcal{S}$ with a period T, so that for all processors p, $M^{\mathcal{S}}(p) \leq M$, and our objective is to find a solution which minimizes the normalized period T/k.

4 Complexity Results

In this section, we analyze the complexity of this problem. We first show that even without memory constraints, finding an optimal allocation is a hard problem. Then we consider the problem of finding a pattern for a fixed allocation, and show that this problem is also NP-difficult.

In both cases, we use a reduction from the 3-partition problem [9]: 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.

4.1 General Problem

Proving the complexity of the general problem does not require to take memory constraints into account, and only relies on the basic underlying allocation problem.

Theorem 1. The decision problem of determining if there exists an allocation and a periodic pattern whose normalized period is at most T is strongly NP-Complete.

Proof. Given an instance of 3-partition, we consider the following instance of our problem:

- $L = 3m$, $P = m$;
- $\forall l,\ a_l = W_l = 0$, so that memory constraints are not a concern;
- $\forall l \leq L,\ u_{F_l} = u_l$ and $u_{B_l} = 0$ (actually we can use any choice of values such that $u_{F_l} + u_{B_l} = u_l$).

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

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 processor. There always exists a shift assignment such that the schedule is valid. Since there is no memory issues (all sizes are set to 0), we obtain a valid 1-periodic schedule.

Let us now assume that there exists a pattern of normalized period T. Then, since one layer cannot be split between two processors, then each processor is allocated to different layers for a total duration at most T. Since the overall load is mT, the load on each GPU must be exactly T. □

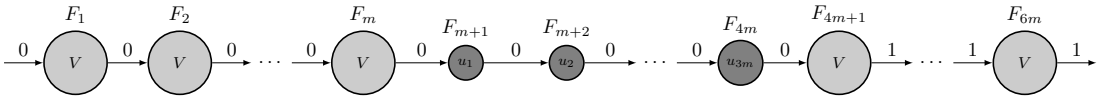


Figure 4: Network Instance for the Proof of Theorem 2

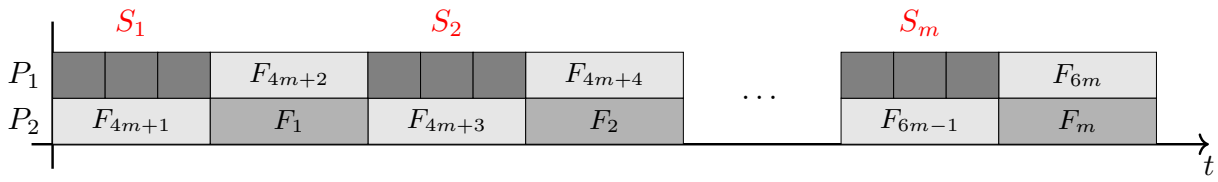


Figure 5: Valid Pattern for the Proof of Theorem 2

4.2 Fixed Allocation Problem

In what follows, we prove that even when the allocation is given, *i.e.* if we know which layer is assigned to which processor, the problem remains strongly NP-Complete, so that both the allocation and the scheduling problems are hard. In this case, given a memory limit M and a task allocation on P processors, the goal is to schedule tasks in periodical manner so that the resulting period is minimal. We prove below in Theorem 2 that even this simpler problem remains NP-hard in the strong sense, what shows that both scheduling and resource allocation are difficult.

Theorem 2. The allocation of layers being fixed, the decision problem of determining if there exists a periodic schedule of normalized period at most T is NP-Complete in the strong sense.

Proof. Given an instance of 3-partition, we consider the following instance of our problem, where the network is depicted in Figure 4 and the processing resources are defined as follows:

- $L = 6m$, $P = 2$, $M = m$, the target period is $T = 2mV$;
- $a_l = 0$ for $l \leq 4m$, and $a_l = 1$ for $4m + 1 \leq l \leq 6m$, while $W_l = 0$ for all l;
- $u_{F_l} = V$ for $l \leq m$ or $l \geq 4m + 1$, and $u_{F_l} = u_{l-m}$ for $m + 1 \leq l \leq 4m$;
- $u_{B_l} = 0$ for all l;
- P_1 is assigned to all layers l for $m + 1 \leq l \leq 4m$, and to even layers $4m + 2, 4m + 4, \ldots, 6m$;
- P_2 is assigned to all layers l for $l \leq m$, and to odd layers $4m + 1, 4m + 3, \ldots, 6m - 1$.

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 5. Since $W_l = 0$ for all l, the memory costs come from storing the activations. Moreover, all operations $F_l, l \geq 4m + 1$ can use the same shift value. Since activation sizes a_l are zero for $l \leq 4m$, the shift values of the other forward operations have no effect on the memory usage and can thus be chosen in a way that makes the pattern valid. Therefore, each layer $l \geq 4m + 1$ has $\text{NCA}_l = 1$, so the memory usage on each processor is exactly m. This shows that there exists a valid pattern of throughput T where all constraints are satisfied.

Let us now assume that there exists a valid schedule $\mathcal{S}$ of period T. For simplicity, we assume that $\mathcal{S}$ is 1-periodic; however all the arguments can be generalized to a k-periodic schedule. We first prove that operation $F_{4m+1}, \ldots, F_{6m}$ are scheduled as depicted in Figure 5. Since NCA values are at least 1 and the memory capacity is m, the pattern must satisfy $\text{NCA}_l = 1$ for these layers. Denote by h the shift of F_{6m}; it is easy to see that it is best for all B_l operations (whose durations are negligible) to be performed just after F_{6m} with the same shift h. Hence, the only way to obtain $\text{NCA}_l = 1$ for $l \geq 4m + 1$ is to process F_j just

before F_{j+1} with the same shift value h. Since $\mathcal{S}$ has period $T = 2mV$, there can be no idle time between these operations. Therefore, operations $F_{4m+1}, \ldots, F_{6m}$ are scheduled as depicted in Figure 5.

Then, the operations $F_{m+1}, \ldots, F_{4m}$ with durations u_i need to be scheduled on P_1, where there are exactly m holes of size V. Hence, the packing on these tasks into the holes creates a solution to the initial 3-partition instance, what completes the NP-Completeness proof. □

5 General Periodic Schedules for Contiguous Allocations

In this section, we analyze in more details the scheduling aspect of our problem. In the following, we thus consider that the allocation is fixed and contiguous and scheduling is done at the stage level (sets of consecutive layers), as scheduling inside stages is straightforward, which is equivalent to the special case of a network of length P to be processed on P processors. We present two results in this context: we first show how to compute, for a given period T, a 1-periodic pattern which minimizes the memory usage; then we provide examples showing the benefit of using k-periodic schedules for $k > 1$.

To simplify the presentation, we use the notations bound to stages. For example, for stage s_i we compose all forward operations and backward operations inside a stage into one forward step F_{s_i} and one backward step B_{s_i}. We also denote $U(s_i)$ as the total sum of all computational costs of some stage s_i.

5.1 Optimal 1-periodic Schedule

To reduce the number of concurrent activations, we propose the 1F1B$^*(T)$ algorithm to compute a pattern for some fixed contiguous allocation $\mathcal{P}$ and a given period T. This algorithm works in three phases, described in Algorithm 1.

Algorithm 1 Summary of Algorithm 1F1B* for a given period T.

Build G groups greedily such that $\sum_{s \in g} U(s) \leq T$, starting from s_P
Schedule operations within group g as an Equal Shift Pattern
Connect the groups with no idle time between the forward operations

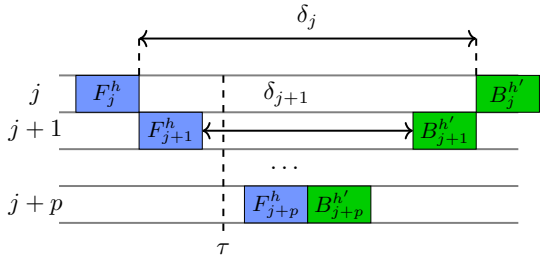
Phase 1: Groups are built such that each group g satisfies the condition $\sum_{s \in g} U(s) \leq T$. This is done iteratively: start from the last stage s_P, add stages $s_{P-1}, s_{P-2}, \ldots$ as long as the condition is fulfilled, then start a new group with the last stage that was not added. This leads to G groups; for simplicity, groups are numbered in the order of their creation, so that group 1 contains s_P and group G contains s_1.
Phase 2: Operations inside a given group g are scheduled with an Equal Shift Pattern where backward operations have a fixed shift h, and forward operations have shift $h + g - 1$.

Definition 1 (Equal Shift Pattern)**.** An Equal Shift Pattern (V-shape) is a part of a schedule in which consecutive forward operations are performed one after the other on their respective processors with the same shift h, followed by the sequence of corresponding backward operations, all having the same index shift h'. On each processor, the time between the forward operation and the corresponding backward is idle (as in Figure 6a).

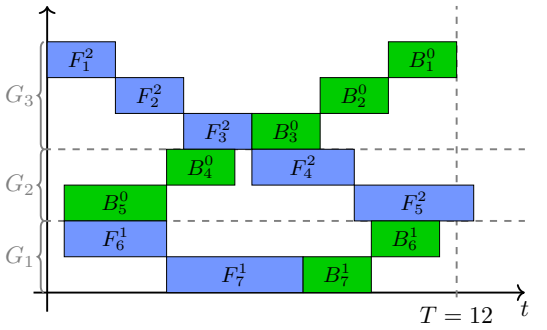
Phase 3: All these group schedules are then connected: to connect group $g = (s_i, \ldots, s_j)$ and group $g - 1 = (s_{j+1}, \ldots, s_k)$, the schedule starts $F_{s_{j+1}}$ just after F_{s_j}, with the same index shift. After this connection, if any operation starts later than T, its starting time is lowered by T and its index shift decreased by 1 (see Figure 6b).

It is easy to see that this algorithm produces a valid pattern. In the following, we prove that for a given period T, the 1F1B* pattern minimizes the NCA of all layers, among all 1-periodic patterns. For this purpose, we start by showing that the Equal Shift Pattern is necessary to avoid increasing the NCA between two stages[2]

[2]all layers of the same stage have the same NCA



(a) Phase 2: Equal Shift Pattern



(b) Phase 3: Example of a 1F1B* schedule with 3 groups.

Figure 6: Scheduling groups in 1F1B*

Lemma 1. Consider any schedule $\mathcal{S}$ for a contiguous partitioning $\mathcal{P}$. If successive stages verify $\text{NCA}_{s_j} = \cdots = \text{NCA}_{s_{j+p}}$, then $\mathcal{S}$ contains a Equal Shift Pattern for these stages.

Proof. Since $\mathcal{S}$ fulfills the dependencies described in Figure 1, the following holds for any stage s

$$\forall t, (\#F_s(t' < t) - \#F_{s-1}(t' < t)) \leq 0 \leq (\#B_s(t' < t) - \#B_{s-1}(t' < t)),$$

so that $\forall t, (\#F_s(t' < t) - \#B_s(t' < t)) \leq (\#F_{s-1}(t' < t) - \#B_{s-1}(t' < t))$ and $\text{NCA}_s^{\mathcal{S}} \leq \text{NCA}_{s-1}^{\mathcal{S}}$. Let us consider stage j, we have $\text{NCA}_{s_j} = \text{NCA}_{s_{j+1}}$. Therefore, there exists a time τ in $\mathcal{S}$ when the memory peak is reached for both stage s_{j+1} and stage s_j, which is only possible if $\#F_{s_{j+1}}(t' < \tau) = \#F_{s_j}(t' < \tau)$ and $\#B_{s_{j+1}}(t' < \tau) = \#B_{s_j}(t' < \tau)$. This shows that F_{s_j} and $F_{s_{j+1}}$ process the same batch (and similarly for B_{s_j} and $B_{s_{j+1}}$). Furthermore, since memory peaks always take place after forward operations, no operation can take place for stage s_j between the end of F_{s_j} and the start of B_{s_j}: the input data for B_{s_j} needs to be produced by $B_{s_{j+1}}$, and processing another forward operation F_{s_j} would increase NCA_{s_j}. Recursively, for any $k \leq p$, all forward operations $F_{s_{j+k}}$ process the same batch, and no operation can take place for stage s_{j+k} between the end of $F_{s_{j+k}}$ and the start of $B_{s_{j+k}}$, which concludes the proof. □

Theorem 3. Consider a contiguous partitioning $\mathcal{P}$ and any 1-periodic schedule $\mathcal{S}$ of period T. For any layer l, the schedule $\mathcal{S}$ uses more concurrent activations than the schedule 1F1B$^*(T)$, *i.e.* $\forall l, \text{NCA}_l^{\text{1F1B}^*} \leq \text{NCA}_l^{\mathcal{S}}$.

Proof. It is easy to see that in 1F1B*, a layer l of group g has $\text{NCA}_l^{\text{1F1B}^*} = g$. Assume that in $\mathcal{S}$, $\text{NCA}_{s_j} = \text{NCA}_{s_{j+1}} = \cdots = \text{NCA}_{s_{j+p}}$ for some j and p. By lemma 1, there is an Equal Shift Pattern for stages s_j to s_{j+p}, so if we denote by δ_j the delay between F_{s_j} and the next B_{s_j} (see Figure 6a), we have $\delta_j \geq \delta_{j+1} + U(s_{j+1})$, and recursively, $\delta_j \geq \sum_{k=j+1}^{j+p} U(s_k)$. Since the period T is the time between two executions of F_{s_j} in $\mathcal{S}$, it is clear that $T \geq U(s_j) + \delta_j$, which yields: if $\text{NCA}_{s_j} = \cdots = \text{NCA}_{s_{j+p}}$, then $T \geq \sum_{k=j}^{j+p} U(s_k)$. By contradiction, assume now that for some stage s_i, the schedule $\mathcal{S}$ uses fewer concurrent activations than the 1F1B* schedule, *i.e.* $\text{NCA}_{s_i} < g_i$, where g_i is the group number of stage s_i, and consider the largest such index i (for larger indices $j > i$, we thus have $\text{NCA}_{s_j} = g_j$). Denote by $s_{i+1}, \ldots, s_{i+p}$ the group of stage s_{i+1}, so that $\text{NCA}_{s_i} = \text{NCA}_{s_{i+1}} = \cdots = \text{NCA}_{s_{i+p}} = g_{i+1} < g_i$. By the previous result, $T \geq \sum_{k=i}^{i+p} U(s_k)$. However, according to the 1F1B* procedure, $g_i > g_{i+1}$ means that stage s_i could not fit in the group of s_{i+1}, which can only happen if $T < \sum_{k=i}^{i+p} U(s_k)$. This results in a contradiction and completes the proof. □

For a fixed partitioning, all other memory requirements are constant and do not depend on the schedule, so that 1F1B* schedule is optimal with respect to memory usage among all valid 1-periodic patterns. Note that in case when each group consists of only one stage, 1F1B* behaves as 1F1B schedule used in [15].

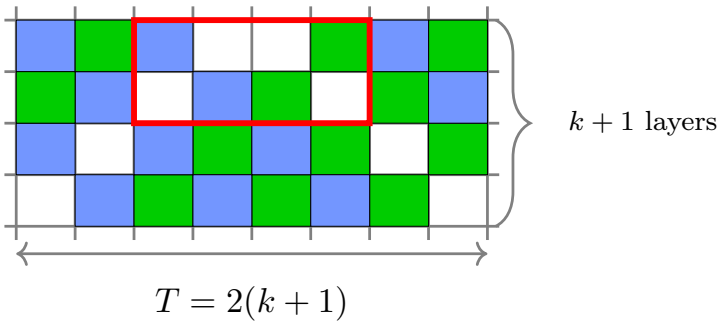


Figure 7: k-periodic pattern for an homogeneous instance. The Equal Shift Pattern is highlighted in thick red.

5.2 k-periodic Schedules

Theorem 4. $\forall k$, k-periodic schedules are sometimes necessary to reach optimal throughput, *i.e.* there are examples where no j-periodic schedule with $j < k$ is able to provide the same throughput as a k-periodic schedule.

Proof. For a given k, let us consider an instance where $P = L = k + 1$, and $M = k + 1$. All layers have the same durations[3] $u_{F_l} = u_{B_l} = 1$ and activation sizes $a_l = 1$, and different weights: $W_1 = 1$, $W_{l+1} = l$ for $l \geq 1$. For such an instance, the memory constraints imply that any valid schedule should satisfy $\text{NCA}_1 \leq k$, and $\text{NCA}_{l+1} \leq k + 1 - l$ for $1 \leq l \leq k$.

Let us consider the k-periodic schedule obtained by unrolling the standard 1-periodic pattern with no idle times, and removing all operations related to every $(k + 1)$-th batch. The resulting pattern has period $2(k + 1)$ and normalized period $2(1 + \frac{1}{k})$. It is depicted on Figure 7 for $k = 3$. The highlighted Equal Shift Pattern shows how this pattern ensures $\text{NCA}_1 = k$. On the other hand, consider any j-periodic schedule $\mathcal{S}$ which satisfies the memory constraints. Since $\text{NCA}_1^{\mathcal{S}} \leq k$, and since NCA_l values are non-increasing with l, there must exist a layer l such that $\text{NCA}_l^{\mathcal{S}} = \text{NCA}_{l+1}^{\mathcal{S}}$. From Lemma 1, there is necessarily a Equal Shift Pattern between these layers, during which layer l is idle for at least $u_F + u_B = 2$ units of time. The period $T^{\mathcal{S}}$ of $\mathcal{S}$ is thus at least $2j + 2$, leading to a normalized period at least $2(1 + \frac{1}{j})$. If $j < k$, this is always higher than the normalized period of the k-periodic pattern described above. □

This example shows the benefit of considering more general schedules than the 1-periodic patterns usually explored in the literature [15]. Furthermore, the simple k-periodic pattern used in this proof can easily be applied to many practical cases where memory capacity is limited. Indeed, for a given contiguous allocation with P stages, all such k-periodic patterns for $k < P$ explore a tradeoff between throughput and memory usage: lower values of k have higher normalized period, but lower values of NCA_l.

6 Contiguous vs General Allocations

Despite being widely used in practice, contiguous allocations can impose significant limitations on the performance. In this section we compare the non-contiguous allocations with the contiguous ones, and we show that in general non-contiguous allocations can reach a throughput which can be up to two times greater than the one of contiguous allocations, when memory is not a bottleneck. While under memory constraints, the improvement in the performance can be arbitrarily high. At the same time, as the non-contiguous allocations are more flexible, they could be the only possible option to execute some large models. Unlike the previous section, any resource can now accommodate an arbitrary set of layers (that can be non-consecutive),

[3] Our arguments actually apply to any homogeneous case where $u_{F_l} + u_{B_l}$ is constant over all layers l.

thus we do not use the notions of stages and groups anymore. Moreover, we talk about processing costs of layers as their combined computing times of forward and backward operations.

6.1 Without memory constraints:

As a simple starting example, a chain with 3 layers should be processed on 2 processors, where the processing costs of each layer are 1, 2 and 1 respectively. It is clear that the smallest period achievable by a contiguous allocation is 3: the second layer is sharing resource either with the first or the last layers. 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 5. 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 [4]. 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. □

6.2 With memory constraints:

The situation is worse when we explicitly take memory into account. Further, for the sake of simplicity, we do not consider activation sizes but model weights only.

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

Proof. It is easy to see on the following example: the chain with 3 layers, whose weights W_l are respectively 1, 2 and 1, and it should be executed on 2 processors with memory limit M equal to 2. In such case, contiguous allocations are not possible, as they demand at least a memory of size 3. On the other hand, with non-contiguous allocations allowed, first and the third layers can be placed on one device, leaving the second layer alone on the other device, which provides a valid allocation. □

Theorem 6. 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.

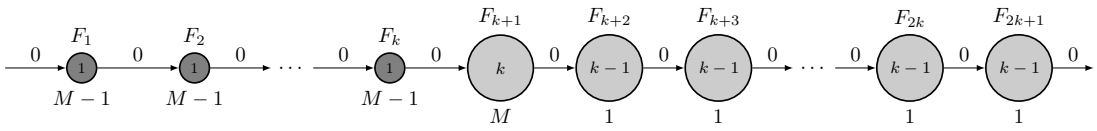


Figure 8: Bad Ratio for Contiguous Allocations and Memory Constraint

Proof. Let us consider the chain depicted in Figure 8. For an arbitrarily chosen k, this chain consists of a sequence of k layers with processing cost 1 and model weight $M-1$, followed by a layer with processing cost k and model weight M and followed by k layers with processing cost $k-1$ and model weight 1. We want to execute this chain on $P = k+2$ resources with memory limit $M \geq k$.

Then, a valid solution consists in grouping, $\forall i \leq P$ layer i and layer $k+i+1$ on processor i, to dedicate processor $k+1$ to layer $k+1$ and to leave processor $k+2$ idle. The required memory $M-1+1$ can fit into the memory and the processing time on each ressource is $k+1$. This gives us a final period $T^* = k$.

If we use contiguous allocation, the first $k+1$ layers must be on separate processors, because of the memory contraint. Then, the last k layers must be on the last remaining processor, that should be feasible due to $M \geq k$. In such scenario, the period is at least as $k(k-1)$, which is $k-1$ times larger than the one of non-contiguous allocation, which concludes the proof. □

7 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 [15] we consider the combination of pipelining and model parallelism, which allows to obtain a better resource utilization. Then, model parallelism can be enhanced 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. The practical solutions proposed in the literature rely on a number of hypotheses, and limit the search to greedy 1-periodic schedules and contiguous allocations. On the contrary, we analyze in detail the complexity of the underlying scheduling and resource allocation problems, and prove that these hypotheses prevent, in the general case, to find optimal solutions, which reinforces the interest of the search for more general strategies.

References

[1] 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.

[2] 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.

[3] Beaumont, O., Eyraud-Dubois, L., and Shilova, A. Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training. In *Proceeding of EuroPar 2020* (2020).

[4] 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.

[5] 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*.

[6] 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.

[7] 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.

[8] 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.

[9] Garey, M. R., and Johnson, D. S. *Computers and intractability*, vol. 174. freeman San Francisco, 1979.

[10] Glorot, X., and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th international conference on artificial intelligence and statistics*.

[11] 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.

[12] 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.

[13] 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).

[14] 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.

[15] 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 SOSP'19*.

[16] Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel dnn training. *arXiv preprint arXiv:2006.09503* (2020).

[17] 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.

[18] Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), SC '20, IEEE Press.

[19] You, Y., Zhang, Z., Demmel, J., Keutzer, K., and Hsieh, C.-J. Imagenet training in 24 minutes.

[20] 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*.

[21] 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.