MadPipe: Memory Aware Dynamic Programming Algorithm for Pipelined Model Parallelism

Olivier Beaumont, Lionel Eyraud-Dubois, Alena Shilova

MadPipe: Memory Aware Dynamic Programming Algorithm for Pipelined Model Parallelism

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

November 26, 2020

Abstract

The training phase in Deep Neural Networks (DNNs) is very computationally intensive and is nowadays often performed on parallel computing platforms, ranging from a few GPUs to several thousand GPUs. The strategy of choice for the parallelization of training is the so-called data parallel approach, based of the parallel training of the different inputs (typically images) and a the aggregation of network weights with collective communications (AllReduce). The scalability of this approach is limited both by the memory available on each node and the networking capacities for collective operations. Recently, a parallel model approach, in which the network weights are distributed and images are trained in a pipeline/stream manner over the computational nodes has been proposed (Pipedream, Gpipe). In this paper, we formalize in detail the optimization problem associated with the placement of DNN layers onto computation resources when using pipelined model parallelism, and we derive a dynamic programming based heuristic, MadPipe, that allows to significantly improve the performance of the parallel model approach compared to the literature.

1 Introduction

Training Deep Neural Network (DNNs) is very time-consuming and requires the efficient use of parallelism in order to be performed in reasonable time. Indeed, supervised training requires performing numerous forward and backward pass calculations, each on a subset of input images called mini-batches (for simplicity, the presentation of this paper assumes that the input data consists of images, but it could be anything). From a scheduling point of view, task dependency graphs differ significantly from the classical HPC operations of linear algebra or scientific workflows. Indeed, each mini-batch processing induces long-range dependencies, since the backpropagation of gradients creates dependencies between each forward task and its associated backward task. In addition, when performing training in parallel, the changes in weights induced by backpropagation must themselves be propagated between the different copies of the neural network processed in parallel, which results in collective communications and even barriers in the synchronous versions, whose convergence is on the other hand generally much better than that of the asynchronous versions.

Although training is still typically performed on small clusters of GPU machines, the use of large HPC infrastructures is becoming popular [7, 15], especially because they offer high-bandwidth

and low-latency networks [8, 10]. Current approaches to parallel training of DNNs include hyper-parameter tuning and data parallelism. Hyper-parameter tuning is a simple and effective way to achieve weak scaling. This strategy reaches high scalability at the beginning of the training phase, but it does not provide any parallelism once the network hyper-parameters have been selected.

Another approach to parallel DNN training is to rely on data parallelism. Using data parallelism [17, 13], the model and all weights are replicated on all participating nodes and different mini-batches are trained in parallel on the different nodes. In this framework, all participating nodes execute forward and backward steps in parallel, and therefore all compute a gradient for all the weights in the network. Synchronization between the nodes takes place at the end of the backward stage, and all partial gradients are gathered and aggregated into one through collective communication such as `Allreduce` and then broadcast to all participating nodes through a `Broadcast` type operation. The above approach is very effective as long as high performance collective communications are implemented and as long as the network is able to support them effectively. It can even be combined with compression to limit the size of messages and thus the size of communications. 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 [11]. In general, the raw performance of data parallelism is therefore limited by the ability of the network to perform efficient collective communications, and the accuracy of the overall training process (for the same raw performance) is limited due to the resulting use of huge mini-batches. Therefore, even though data parallelism remains the state of the art technique for parallel training, several other techniques have been proposed in combination with data parallelism in order to address above issues.

In order to distribute the memory requirements associated with DNN weights storage, pipelined model parallelism is a very promising approach that has recently been advocated in many papers [5, 9, 11, 16], and which can be combined with data parallelism [4]. The general idea is to distribute different layers of a network over different resources. The practical use of the pipelined model parallelism is nevertheless a delicate issue and the analysis of the induced memory needs is the main contribution of the present paper. Recently introduced PipeDream [11] and other tools derived from it like [12, 16] rather solve the problem of finding good load balancing for different system architectures. Indeed, if tasks are well distributed between the resources where data is communicated via a communication link suited to its size, then it helps to achieve a good scalability with nearly linear speed-up. However, despite the fact that each resource is responsible only for the part of the DNN, and thus only storing a fraction $\sim \frac{1}{P}$ of model weights, it does not necessarily induce a smaller memory consumption. Indeed, it is necessary to guarantee that the training process is valid: the backward propagation should use the same weights and activations as the ones used in the forward propagation. In other words, if b samples are pipelined in the beginning of training, then each processor is required to keep its own versions of weights and activations for each sample, which induces in total a replication factor $\sim \frac{b}{P}$ of weights and activations per processor. When considering larger DNNs that are trained on large images, it becomes crucial to estimate memory as precisely as possible. Some efforts were done in this direction, as can be seen with the extension of PipeDream to PipeDream-2BW [12], but the memory measurements remain approximative and are mostly relevant for homogeneous neural networks used in natural langage processing. The question of gradient staleness in the context of pipeline model parallelism has been considered in [2].

In this paper we consider the problem of finding a good load balancing and a schedule that are able to perform the training on P given processors with memory capacity M and connected

via the network with bandwidth β. We propose an algorithm, called MadPipe (for Memory Aware Dynamic programming for PIPElining) based on the combination of dynamic programming and linear programming, with two main contributions: (i) a more precise estimation of the memory requirements, which results in allocations that can be more efficiently scheduled, and (ii) the possibility to use non-contiguous allocations of layers of the DNN to processors, which provides a better load-balancing. The running time of MadPipe is acceptable in practice, and we show that it significantly improves the resulting training performance with respect to state of the art implementation of model parallelism.

In terms of positioning, this paper focuses on the optimization of model parallelism, i.e. the question of the optimization of the throughput that can be achieved by partitioning the DNN on several computation nodes. Our first objective is to better understand the difficulty of the associated optimization problems and to put into discussion the hypotheses that are classically performed in this framework, in order to eventually propose heuristics that are more efficient than those in the literature. Secondly, in terms of performance, we will see that this approach alone makes it possible to obtain a very strong parallel efficiency, even in the presence of poor network performances, but its scalability is limited, of the order of a few tens of nodes at most. Model parallelism alone is therefore a solution of choice for platforms consisting of a few GPUs, even if they are connected by relatively low bandwidth links. In the case of larger computation platforms, the work carried out in this paper on the optimization of model parallelism can be combined with data parallelism. Indeed, the data parallel approach induces high bandwidth usage (because of the collective communications of the network weights), which limits its scalability. In this context, the optimization of model parallelism can be used to create G groups of resources, where all resources within a group are associated to the same layers of the network (G being of order ten) and data parallelism can be used within each of these groups to achieve better scalability. From the point of view of network resource consumption, we therefore essentially end up performing G collective communications roughly G times smaller on sub-platforms that are themselves roughly G times smaller, which is very efficient. The optimization of the model parallelism alone is therefore useful in all contexts in which the scalability of data parallelism alone is limited by communication resources, whether at small or large scale.

The rest of the document is organized as follows. The related work is presented in Section 2. We introduce the notations in Section 3. In Section 4.1, we show how to generate an optimal schedule for contiguous allocations, that takes memory and communication constraints into account. Then, we propose MadPipe in Section 4.2: a new allocation scheme, more general than the one proposed in PipeDream and that considers memory and communication constraints. We show how to build an associated schedule in Section 4.3. All these approaches are compared in practice on realistic networks and platform models in Section 5. Finally, conclusions and perspectives of this work are proposed in Section 6.

2 Related Works

When using model parallelism [5, 9, 11, 16], the different layers of a network are spread over different resources, so that the replication and the synchronization of the network weights can be avoided, which saves both memory and communication resources. On the other hand, activations produced by a layer on one node needs to be transmitted to another node if the next layer of the network is placed on another resource.

The practical use of the model parallelism is nevertheless a delicate issue and the analysis of the induced memory needs is complex. In [9], it is proposed to split the training batch into several

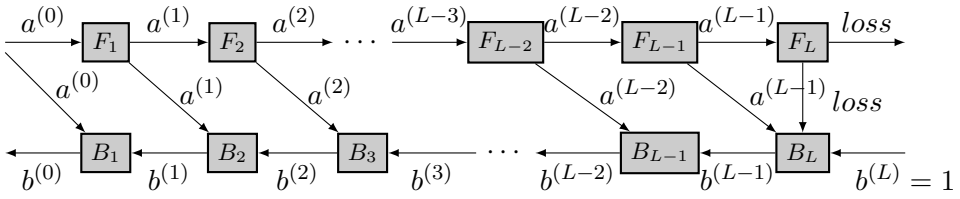


Figure 1: Graph for the Backward Propagation

mini-batches, which are then pipelined through the layers of the network (and the different computing resources). Once the forward and backward phases have been computed on all these mini-batches, the weights are then updated. This approach is fairly simple to implement but has the disadvantage of leaving the computational resources largely idle. The PipeDream approach proposed in [11] improves this training process, by only forcing that the forward and backward tasks use the same model weights for a given mini-batch. Such a weakened constraint on the training process allows PipeDream to achieve a much better utilization of the processing resources.

Following the contributions of [9] and [11], many authors have explored the use of model parallelism. To solve issues in the case of high communication costs, the authors of [16] have proposed an updated strategy which assumes no overlap between computation and communication. We show in this paper how to correctly take communications into account in the PipeDream model, which directly solves this issue. Another important issue related to PipeDream is the need to keep many copies of the network parameters (also called weights), which can potentially cancel the benefit of using model parallelism. To address this issue, different approaches have been proposed. Within the framework of homogeneous networks, a strategy has been proposed in [12] to keep only two models in memory. In this paper, we rely on the same strategy. Another strategy to keep fewer models in memory is to allow more asynchronous updates. This strategy has been explored in particular in [2], which proposes an intermediate approach to avoid gradient staleness. However, this requires two versions of the model weights and two versions of the gradients, and therefore does not improve memory usage.

Other extensions of PipeDream have recently been proposed to explore different ways of combining data and model parallelism. In the DAPPLE framework [6], the focus is on the case of several nodes, each equipped with several GPUs. DAPPLE extends PipeDream by allowing more possibilities to map a stage of the DNN to GPUs located in several nodes. The assignment problem is solved without taking memory constraints into account. The HetPipe proposal [14] considers a different way of combining data and model parallelism, in which nodes may contain different GPUs. The idea of HetPipe is to heuristically split the GPUs into virtual workers which may contain heterogeneous GPUs and use data parallelism between virtual workers. Model parallelism is used inside the virtual workers, based on a simplified ILP which assumes no overlap between computation and communication. Both these works are orthogonal to our research: we focus on a better understanding of the memory constraints of the model parallelism approach. Our techniques can later be applied in the DAPPLE or HetPipe frameworks to improve the corresponding scheduling phases. Finally, an Integer Linear Programming formulation for the problem considered here was proposed in [1]. It allows to find more general non-contiguous allocations of layers of the network, and an associated schedule which precisely respects the memory constraints. However, this formulation is not adapted to large neural networks and results in very large computation times. In this paper, we propose a

heuristic that produces efficient schedules even for large neural networks.

3 Model and Notations

In this paper, like in many papers from the related literature [3, 11, 12, 2, 16], we consider linear (or linearized) DNNs, in which each forward operation depends only on the result of the previous operation. We thus consider a chain of L layers (see Figure 1) numbered from 1 to L. Each layer l is associated both to a forward operation F_l and a backward operation B_l (see 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.

During the training phase, we assume that the set of input data (typically images) is split into mini-batches of size $\mathcal{B}$. In this context, we use the following notations:

- u_{F_l} denotes the duration of the forward task on the layer l with a mini-batch of size $\mathcal{B}$;
- u_{B_l} denotes the duration of the backward task on the layer l with a mini-batch of size $\mathcal{B}$;
- W_l denotes the size of the parameter weights for layer l;
- a_l denotes the size (in bytes) of the tensor $a^{(l)}$ produced by F_l when applied to a mini-batch of size $\mathcal{B}$;
- a_l also corresponds to the size (in bytes) of the tensor $b^{(l)}$ produced by B_{l+1}, as each gradient has the same size as the activations with respect to which it is calculated.

Our goal is to assign the layers of the DNN to P computing resources (typically GPUs), with limited memory M, assuming (as in PipeDream) that all pairs of GPUs are connected through a direct link of capacity β. Assigning layers l and $l+1$ on different GPUs induces a communication of $a^{(l)}$ in the forward phase and $b^{(l)}$ in the backward phase, and thus requires to use the corresponding communication link. We also need to schedule the forward and backward operations of each layer on its GPU, in a *pipelined* way: 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. The memory of each GPU is used to store all data required to perform the training operation:

- Parameter weights. As discussed in [12], it is sufficient to keep two versions of each parameter, plus one gradient where the updates are accumulated during training. Thus, assigning a layer l to a GPU induces a memory load of $3W_l$ for the weights.
- Activations computed in the forward phase. With pipelined executions, several forward activations need to be stored in memory at the same time, and thus it is important to take these memory costs into account when assigning layers to GPUs. The precise count of these activations is addressed in Section 4.1.
- Communication buffers. When layers l and $l+1$ are assigned to different GPUs, a communication must take place, and we assume that for a more convenient implementation, some memory is reserved as a buffer to store both $a^{(l)}$ and $b^{(l)}$ while they are being sent or received. This requires a storage of size $2a_l$ on both GPUs.

Throughout this paper, we are interested in finding efficient task allocations (load balancing) and schedules. To help navigate the different concepts that we use throughout the paper, we introduce some terminology. We consider the input DNN as a chain of *layers*, which are the basic operations that need to be computed. A *partitioning* 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 GPUs. An allocation is said to be *contiguous* if each GPU is assigned a single stage, and by extension a partitioning is contiguous if it contains at most P stages. To estimate memory requirements we will also introduce *groups* of stages, where each group is a set of stages which are contiguous with respect to the ordering of the chain. A *schedule* of a given allocation specifies the timings of all compute operations and all communications.

We actually focus on periodic schedules, which are simpler to analyze and to implement. An Integer Linear Programming formulation has been proposed [1] to solve both the allocation and the scheduling problems at the same time, but the resulting algorithm does not scale to larger DNNs. This paper presents MadPipe, an algorithm which first solves the load balancing problem and then computes an efficient schedule based on this allocation.

We consider periodic patterns $\mathcal{S}$ of period T and we specify for each operation (forward, backward, and communication): the GPU in charge of it, a starting time t, and an index shift h. This pattern is to be repeated indefinitely: in the k-th period, this operation starts at time $kT + t$ and processes the batch number $k - h$. By convention the shift of F_1 is always 0, so that if in some period F_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 2 shows an example of a valid pattern. It is possible to imagine larger patterns that may contain each operation several times, for example patterns of period $2T$ in which each operation would be performed twice. This might allow to consider more general schedules, but is beyond the scope of this work.

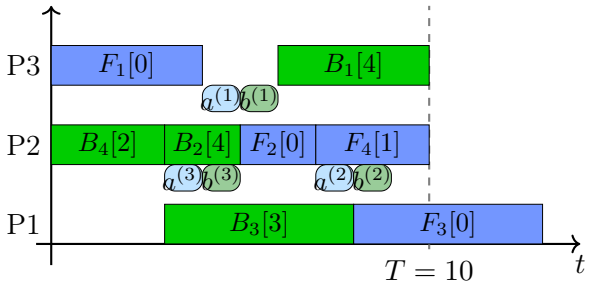


Figure 2: Example of valid pattern. The index shifts are indicated in brackets in computation tasks, communication tasks have rounded corners.

4 Heuristics

This section presents the main algorithmic contributions of this paper. We start by an optimal algorithm for computing a pattern given a contiguous allocation of layers to GPUs. We then present the MadPipe algorithm, that consists of two parts: a dynamic programming formulation to obtain a non-contiguous allocation, and an adaptation of the ILP formulation of [1] to compute a pattern adapted to this allocation.

4.1 Contiguous case (1F1B)

Previous model parallelism approaches like PipeDream [11] focus on contiguous allocations, where each GPU is assigned a consecutive set of layers of the DNN. The operations are then scheduled with a simple eager strategy: fix a pipeline depth (the number of mini-batches inserted in the pipeline), and start each operation as soon as its input data is available. This eager strategy does not provide any guarantee on the efficiency of the resulting schedule. In addition, with such an eager approach, it is very difficult to predict the memory usage that the schedule of a particular allocation will require, and so PipeDream uses a very rough estimate of the memory consumption when optimizing the allocations.

In this section, we present an algorithm that computes the optimal periodic pattern given a contiguous allocation: given a feasible period T, our algorithm provides a pattern of period T which uses the smallest amount of memory on all GPUs. This study also provides valuable insight on the required memory usage, which will be used in the MadPipe algorithm to more precisely take memory requirements into account.

Assume that we are given a fixed contiguous partitioning $\mathcal{P}$ of a linear DNN of length L on P computing resources, to be scheduled with a periodic pattern of period T. For ease of presentation, we assume that all communication times are 0; we show at the end a very simple transformation that allows to consider the general case. The partitioning $\mathcal{P}$ specifies exactly P *stages*, one for each computing resource, where each stage s_i is a contiguous set of layers. Given a stage s, we denote by F_s the forward operation which computes in sequence the forward operations of all layers of s, and similarly B_s for the backward operations. We also denote by $U_F(s)$ the total duration of the forward tasks of stage s, $U_B(s)$ the duration of backward tasks, and $U(s) = U_F(s) + U_B(s)$ the total computational load of stage s.

We propose the 1F1B* Algorithm to compute a pattern for $\mathcal{P}$. This algorithm works in two phases.

First, groups of stages are built such that in each group g, $\sum_{s\in g} U(s) \leq T$. This is done iteratively: start from the last stage s_P, add stages $s_{P-1}, s_{P-2}, \dots$ as long as the condition is fulfilled, then start a new group with the last stage that was not added. This yields 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.

Then, each group $s_i, s_{i+1}, \dots, s_j$ is scheduled in sequence: first forward operations $F_{s_i}, F_{s_{i+1}}, \dots, F_{s_j}$, then backward operations $B_{s_j}, B_{s_{j-1}}, \dots, B_{s_i}$, each performed in sequence, without idle time, as shown on Figure 3. All forward operations have the same index shift value h, and backward operations have $h' = h + g - 1$, where g is the group index. All these group schedules are then connected: to connect group $g = (s_i, \dots, s_j)$ and group $g - 1 = (s_{j+1}, \dots, s_k)$, 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 increased by 1.

This algorithm produces a valid pattern, *i.e.* a schedule which fulfills all dependencies when repeated. Indeed, by construction all dependencies within groups are satisfied, as well as the dependencies from F_{s_l} to $F_{s_{l+1}}$ for any l. Consider now a group g whose first layer is layer l. If the forward operation F_{s_l} on some mini-batch i starts at time t, then the condition $\sum_{s\in g} U(s) \leq T$ ensures that the backward operation B_{s_l} on mini-batch $i - g + 1$ ends before time $t + T$. Time $t + T$ is the starting time of F_{s_l} on mini-batch $i + 1$, and by construction it is also the starting time of $B_{s_{l-1}}$ on mini-batch $i - g + 1$, which shows that the data dependency between groups is fulfilled.

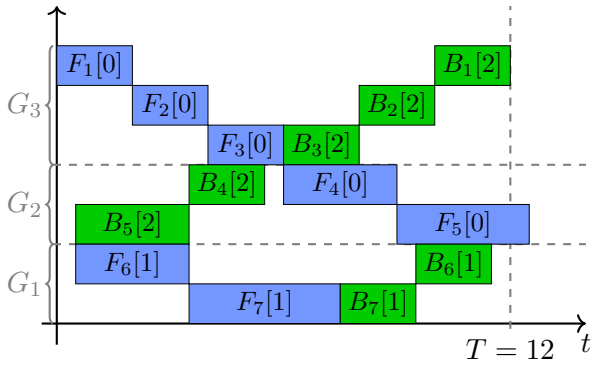


Figure 3: Example of a 1F1B* schedule with 3 groups.

In terms of memory usage, all stages of group g (for $1 \leq g \leq G$) need to store exactly g input activations: if F_{s_l} is processed on mini-batch i, the next backward operation B_{s_l} is executed on mini-batch $i+g-1$, and thus after F_{s_l} the corresponding GPU needs to store the activations for mini-batches i to $i+g-1$.

Proposition 1. *Given a contiguous partitioning $\mathcal{P}$, consider any periodic pattern $\mathcal{S}$ of period T. For any i, $\mathcal{S}$ keeps at least as many active batches for stage s_i than the 1F1B* defined above.*

Proof. From the valid schedule obtained by repeating pattern $\mathcal{S}$, for any stage $s_i \in \mathcal{P}$ we introduce the function

$$\hat{m}_{s_i}(t) = \#F_{s_i}(t' < t) - \#B_{s_i}(t' < t).$$

This function measures how many more F_{s_i} operations have taken place by time t than B_{s_i} operations. Consequently, it also denotes the number of active batches of stage s_i that are stored at time t in memory. We also use $m_{s_i} = \max_t \hat{m}_{s_i}(t)$ to denote the maximal amount of active batches related to stage s_i.

We first notice that for all t and i, $\hat{m}_{s_{i+1}}(t) \leq \hat{m}_{s_i}(t)$. Indeed, since $\mathcal{S}$ is a valid pattern,

$$\begin{aligned}\#F_{s_{i+1}}(t' < t) &\leq \#F_{s_i}(t' < t)\\ \#B_{s_{i+1}}(t' < t) &\geq \#B_{s_i}(t' < t),\end{aligned}$$

so that $m_{s_{i+1}} \leq m_{s_i}$.

Assume that for some j, $m_{s_j} = m_{s_{j+1}}$. In this case, there exists a time τ such that $\hat{m}_{s_j}(\tau) = \hat{m}_{s_{j+1}}(\tau)$, which is only possible if $\#F_{s_{j+1}}(t' < \tau) = \#F_{s_j}(t' < \tau)$ and $\#B_{s_{j+1}}(t' < \tau) = \#B_{s_j}(t' < \tau)$. Then, between the end of F_{s_j} and the end of $B_{s_{j+1}}$, no operation can take place for stage 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 the number of active batches above m_{s_j}.

For all j, if we denote by δ_j the delay between F_{s_j} and the next B_{s_j} in the periodic schedule, the previous considerations imply (see Figure 4): if $m_{s_j} = m_{s_{j+1}}$, then $\delta_j \geq \delta_{j+1} + U(s_{j+1})$. Recursively, if $m_{s_j} = m_{s_{j+1}} = \cdots = m_{s_{j+p}}$ for some j and p, then $\delta_j \geq \sum_{k=j+1}^{j+p} U(s_k)$.

Since the period T is the time between two executions of F_{s_j}, it is clear that $T \geq U(s_j) + \delta_j$, which yields

$$\text{if } m_{s_j} = \cdots = m_{s_{j+p}}, \text{ then } T \geq \sum_{k=j}^{j+p} U(s_k).$$

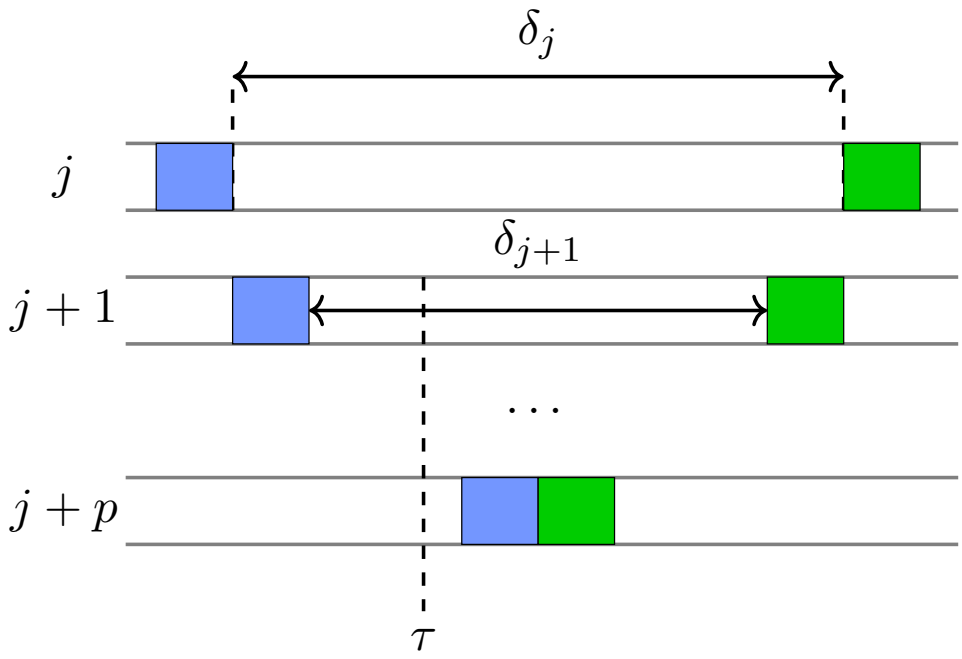


Figure 4: Idle time imposed by keeping the same amount of active batches. At the moment of memory peak the tasks from different processors form "V"-shape.

By contradiction, assume now that for some stage s_i, the schedule $\mathcal{S}$ uses fewer active batches than the 1F1B* schedule, *i.e.* $m_{s_i} < g_i$, where g_i is the group number of stage s_i. Consider the largest index i for which $m_{s_i} < g_i$. Denote by $s_{i+1}, \ldots, s_{i+p}$ the group of stage s_{i+1}, so that $m_{s_i} = m_{s_{i+1}} = \cdots = m_{s_{i+p}} = g_{i+1} < g_i$. 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 $T < \sum_{k=i}^{i+p} U(s_k)$.

□

For a fixed partitioning, all other memory requirements (parameter weights and communication buffers) are constant and do not depend on the schedule. Thus, the 1F1B* schedule induces the smallest possible memory usage on all processors, among all valid periodic patterns for this partitioning.

This result also holds true when taking communications into account: in a contiguous partitioning, we can consider each communication as if it was a computation layer: the communication between stage s_i on processor p and s_{i+1} on processor p' 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 $C(l) = \frac{2a_l}{\beta}$, with the same dependencies as for a normal computation layer. Therefore, we can transform a partitioning on P resources with communication costs into a partitioning on $2P-1$ resources, without communications costs, and apply the 1F1B* algorithm on this transformed partitioning.

4.2 Building a non-contiguous allocation

In this section, we present the first part of the MadPipe algorithm: a dynamic programming algorithm to build a non-contiguous allocation. However, we do not consider general allocations, because solving the corresponding problem (as is done with an Integer Linear Program in [1]) is too costly and can only be done for small or medium size networks. Instead, we focus on a specific case: we look for allocations in which all processors are allocated only one stage (like in a contiguous allocation, we call these processors *normal*), except for one *special* processor which may receive any number of stages. As shown in Section 5, this is enough to significantly improve the load balancing.

A non-contiguous allocation is a partitioning of the layers into stages, but it may contain more than P stages: some processors (in our case, the special processor) can be assigned several stages. Such an allocation also specifies an assignment of stages to processors. We define the period of an allocation as the period that it can achieve if memory constraints were ignored; it can be computed as the total load of the most loaded resource (either a GPU or a communication link).

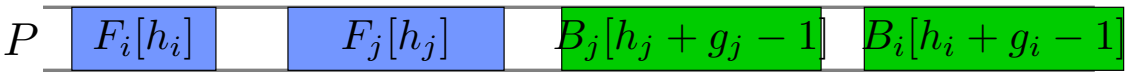
In the following, we denote $U(k,l) = \sum_{i=k}^{l} u_{F_i} + u_{B_i}$ the total computational cost of layers k to l, and $C(k) = \frac{2a_k}{\beta}$ the communication time associated with layer k. The above discussion about 1F1B* are used to accurately estimating the memory usage for all normal processors, based on the total computation time of layers further down the chain. The 1F1B* algorithm requires a target period, so our dynamic programming method uses a target $\hat{T}$ as input, and computes the best possible period for an allocation in which memory needs are computed assuming a period $\widehat{T}$.

4.2.1 Estimating memory usage

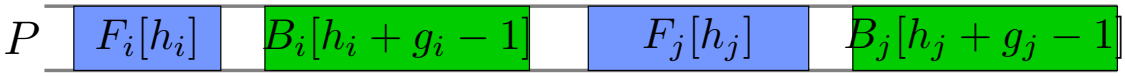
Assume that layers k to l are assigned to a normal processor, while requiring to keep g activations in memory. The memory usage on this processor is $\mathcal{M}(k,l,g) = \sum_{i=k}^{l}(3W_i + g \cdot a_{i-1}) + 2 \cdot (a_{k-1} + a_l)$, where $3W_i$ represents the memory usage of the model parameters, $g \cdot a_{i-1}$ corresponds to the activations, and $2 \cdot (a_{k-1} + a_l)$ accounts for the communication buffers (if $k = 1$ or $l = L$, the corresponding term should be removed since no communication takes place). To compute the value g, assume that we are given a lower bound V on the delay between the execution of F_l on some batch and the execution of B_l on the *same* batch. Then these layers can be scheduled with g activations in memory if and only if $V + U(k,l) \leq g \cdot \widehat{T}$. Hence, we can compute the number of activations to be kept in memory for layers k to l as $g(k,l,V) = \left\lceil \frac{V+U(k,l)}{\widehat{T}} \right\rceil$, and the memory usage is given by $\mathcal{M}(k,l,g(k,l,V))$.

For the special processor however, estimating the memory usage is more difficult. We can use the same $g(k,l,V)$ formula to compute the number of activations to be kept in memory for each stage assigned to the special processor. However, as can be seen on Figure 5, for a given allocation on this processor, the memory peak depends on how the different stages are scheduled. Specifically, assume that several stages are assigned to the special processor, where stage s_i is part of group g_i. If all forward operations are performed in sequence followed by all backward operations, then after the end of the last forward operation, stage s_i stores g_i activations, for a total memory peak of $\sum_i g_i a_i$. On the other hand, if the backward operation of each stage is performed just after its forward operation, then after some $F_{s_{i_0}}$ we still have g_{i_0} activations stored for stage s_{i_0}, but only $g_i - 1$ activations for the other stages. Thus the overall memory peak is $\max_i g_i a_i + \sum_{j \neq i}(g_j - 1)a_j$. Many more ways of interleaving the operations can take place, and determining which ones are compatible with the schedule of the other processors is difficult. In any case, at least $g_i - 1$ activations for each stage s_i need to be stored at all times. For this reason, in this first part of MadPipe, we under-estimate the

memory requirement of the special processor by considering that it requires $\mathcal{M}(k, l, g-1)$, and rely on a modified version of the ILP given in [1] to compute a feasible schedule in the second part of MadPipe (see Section 4.3).



(a) Worst case. Memory needed to store the activations: $g_i a_i + g_j a_j$ (reached between F_j and B_j).



(b) Best case. Memory needed to store the activations: $\max\{g_i a_i + (g_j-1)a_j, (g_i-1)a_i + g_j a_j\}$ (reached either after F_i or after F_j).

Figure 5: Two schedules with different memory peaks with two layers assigned on the special processor: layer i with shift h_i and group g_i, layer j with shift h_j and group g_j.

4.2.2 Dynamic Programming derivations

To specify our dynamic programming algorithm, we fix a target value $\widehat{T}$, and we define $T(l, p, t_P, m_P, V)$ as the smallest period of an allocation of the first l layers on p normal processors which fulfills the above memory constraints, where (i) the delay between the end of F_l and the start of the corresponding B_l on the same batch is at least V, and (ii) assuming that the special processor has already been assigned layers that induce a computational time t_P and memory usage m_P.

Consider any such allocation of the first l layers. The last layer l is part of a stage $k, \dots, l$, with $k \leq l$. From k, l and V we can compute a lower bound V' on the time between the end of F_{k-1} and the start of B_{k-1}, by mimicking the group-making process of the 1F1B* procedure. Denote by $g^0 = \left\lceil \frac{V}{\widehat{T}} \right\rceil$ the group number of the previously considered stage. Layers k to l can use the same group if $\left\lceil \frac{V+U(k,l)}{\widehat{T}} \right\rceil = g^0$, in which case the delay between the end of the communication of $a^{(k-1)}$ and the start of the communication of $b^{(k-1)}$ is $V + U(k, l)$. Otherwise, it is necessary to start a new group, which implies that this delay is $g^0 \cdot \widehat{T} + U(k, l)$. The same reasoning applies to the group number of the communication phase for $a^{(l)}$ and $b^{(l)}$. By introducing the notation

$$x \oplus y = \begin{cases} x + y & \text{if } \left\lceil \frac{x}{\widehat{T}} \right\rceil = \left\lceil \frac{x+y}{\widehat{T}} \right\rceil \\ \widehat{T} \cdot \left\lceil \frac{x}{\widehat{T}} \right\rceil + y & \text{otherwise,} \end{cases}$$

we obtain $V' = (V \oplus U(k, l)) \oplus C(k)$.

The resulting stage made of layers $k, \dots, l$ can then be assigned either to a normal processor, or to the special one. Assigning it to a normal processor is only feasible if $\mathcal{M}(k, l, g(k, l, V)) \leq M$, and it means one less processor is available to allocate all layers from 0 to $k-1$. Hence this yields a period

$$T_N(k) = \max(U(k, l), C(k-1), T(k-1, p-1, t_P, m_P, V')).$$

On the other hand, assigning this stage to the special processor induces a load $t'_P = t_P + U(k, l)$, and a lower bound on the memory usage of $m'_P = m_P + \mathcal{M}(k, l, g(k, l, V) - 1)$. This is only feasible

if $m'_P \leq M$, and yields a period of

$$T_S(k) = \max(t'_P, C(k-1), T(k-1, p, t'_P, m'_P, V')).$$

Putting it all together, the value of $T(l, p, t_P, m_P, V)$ can be determined as the best possible choice:

$$T(l, p, t_P, m_P, V) = \min\left(\min_{k \leq l} T_N(k), \min_{k \leq l} T_S(k)\right),$$

where for each case we only consider the feasible values of k as defined above. This allows to recursively compute all values of $T(\cdot)$. Indeed, we can easily compute the values of T corresponding to $l = 0$ or $p = 0$: if there are no more layers to allocate, then $T(0, p, t_P, m_P, V) = t_P$; if no normal processor is available, then all layers must be assigned to the special processor, which is feasible if $m_P + \mathcal{M}(1, l, g(1, l, V) - 1) \leq M$ and yields $T(l, 0, t_P, m_P, V) = U(1, l) + t_P$.

We thus obtain the following allocation algorithm, called MadPipe-DP : recursively compute all possible values for $T(l, p, t_P, m_P, V)$ to obtain $T(L, P-1, 0, 0, 0)$, which is equal to the period of the resulting allocation. Then all the decisions along the path that leads to this result (the values k and the choice between T_N and T_S) provide a partitioning of the layers into stages, and an assignment of each stage either to a normal or to the special processor.

4.2.3 Find the correct value for $\widehat{T}$

The above MadPipe-DP has two interesting properties: first, the resulting period $T = \text{MadPipe-DP}(\widehat{T})$ is a non-increasing function of $\widehat{T}$, since a higher value of $\widehat{T}$ allows to store fewer activations and thus makes the memory constraints less restrictive. Second, for all $\widehat{T}$, scheduling the allocation produced by MadPipe-DP requires a period at least T for the load balance, and at least $\widehat{T}$ to ensure that the memory constraints are fulfilled. Hence, the value $\widehat{T}^*$ minimizes $\max(\text{MadPipe-DP}(\widehat{T}^*), \widehat{T}^*)$.

Both arguments are monotonic in opposite directions, and we can therefore use a modified binary search algorithm to find $\widehat{T}^*$ (see Algorithm 1). At each step, if $T = \text{MadPipe-DP}(\widehat{T})$, we know that $\min(T, \widehat{T})$ is a lower bound for $\widehat{T}^*$, and $\max(T, \widehat{T})$ is an upper bound. We perform this algorithm for a fixed number of iterations. In practice, $K = 10$ iterations are enough to obtain a good solution.

Algorithm 1 First phase of MadPipe: build an allocation

Require: K (number of iterations)
1: $\text{lb} \leftarrow U(1, L)/P$
2: $\text{ub} \leftarrow U(1, L) + \sum_{i=2}^{L} C(i)$
3: $\widehat{T}_1 \leftarrow \text{lb}$
4: **for** $i = 1, \ldots, K$ **do**
5: $\quad T_i \leftarrow \text{MadPipe-DP}(\widehat{T}_i)$
6: $\quad \widetilde{T}_i \leftarrow \max\{T_i, \widehat{T}_i\}$
7: $\quad \text{lb} \leftarrow \max\{\text{lb}, \min(T_i, \widehat{T}_i)\}$
8: $\quad \text{ub} \leftarrow \min\{\text{ub}, \widetilde{T}_i\}$
9: $\quad \widehat{T}_{i+1} \leftarrow (\text{lb} + \text{ub})/2$
return $\min_i \widetilde{T}_i$ and the associated allocation

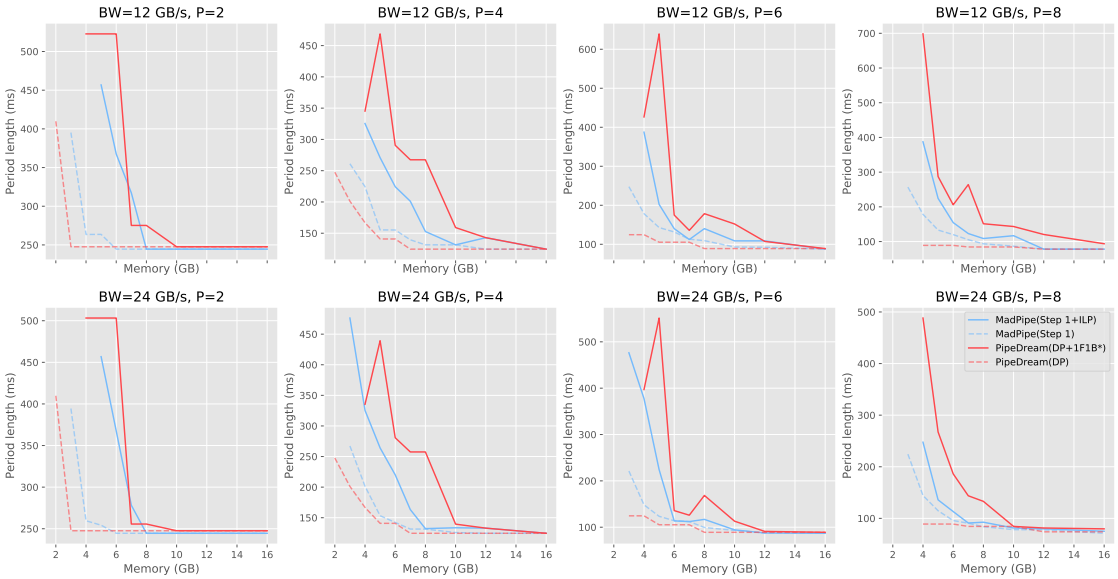


Figure 6: Comparison of periods for ResNet-50 with image size 1000 and batch size 8.

4.3 Scheduling with ILP

As discussed above, it might not be possible to schedule the allocation returned by Algorithm 1 within the expected period because of the approximation used to estimate the memory usage of the special processor. In order to obtain a valid schedule, the second step of the MadPipe algorithm uses the ILP formulation proposed in [1] to compute an efficient valid schedule which uses the same partitioning.

Assume that Algorithm 1 returns a solution where the layers are partitioned into N stages $(s_1, \ldots, s_N)$. Scheduling this partitioning is almost equivalent to scheduling a (shorter) chain network of length N, where each stage of the partition is considered as a layer of the transformed chain. Each layer of this new chain is thus more computationally expensive, with $u'_{F_i} = U(s_i)$. The equivalence is however not perfect since data dependencies are different. Indeed, if stage s_i contains layers $(j, \ldots, k)$, the output of its forward operation F_{s_i} is the activation $a^{(k)}$ produced by layer k. However, the backward operation B_{s_i} requires $(a^{(j-1)}, \ldots, a^{(k-1)})$, and not only $a^{(j-1)}$.

We thus define, for each stage s, the stored activation cost as $\bar{a}_s = \sum_{i \in s} a_{i-1}$. Our final algorithm MadPipe uses a modified version of the ILP [1], which uses a_s for the communication between stages, and $\bar{a}_s$ for the memory storage constraints. Since the number N of stages in the partition returned by the first step is much lower than the length L of the original chain, the processing time of the ILP on this modified instance is reasonable in all cases.

5 Experimental Results

5.1 Simulation Settings

In this section, we present the simulation results obtained for different state-of-the-art and widely used neural networks. The data necessary to perform the simulations were obtained by profiling the neural networks to measure the durations and memory costs of the different operations involved in

Figure 7: Geometric mean of ratios over different values of P and β, for all networks.

the training. As mentioned in Section 3, all the networks are considered as adjoint chains as depicted in Figure 1. A classic linearization approach, also used for PipeDream [11], is used to transform the computational graphs of these neural networks into chains, by greedily grouping layers as necessary.

In this evaluation, we compare two scheduling algorithms. The state-of-the-art algorithm for pipelined model parallelism is PipeDream [11], which produces a contiguous partitioning. This algorithm assumes that the number of activations for all layers is at most P (the number of ressources), whereas we know from Section 4.1 that the first layers may require to store up to $2P-1$ activations, since we also need to take communication stages into account. We thus use 1F1B* to obtain a valid schedule from the partitioning returned by PipeDream. We compare this algorithm to the MadPipe algorithm presented above.

The formulation of MadPipe-DP involves continuous variables t_P, m_P and V, which need to be discretized for the implementation. The choice of the granularity for this discretization is a tradeoff between the precision of the solution and the computational effort required to obtain the solution. In these experiments, 11 equally distributed values between 0 and M are used for representing m_P, 51 equally distributed values between 0 and $U(1,L) + \sum_{i=2}^{L} C(i)$ for V and 101 values between 0 and $U(1,L)$ for t_P. Such a discretization scheme allows to achieve good results in reasonable time. Overall, the first step of MadPipe takes several seconds for the smaller networks, and up to 15 minutes for the large networks. For the second step, the ILP is executed with a one-minute time limit, but finishes earlier with an optimal solution in most cases. Even though this is significantly slower than the dynamic program of PipeDream, the improved partitionings allow to increase the overall throughput of the training phase. Indeed, this optimization process is expected to be executed once for a given network and a given machine, whereas the training phase involves many runs with the same stages, with an expected runtime of several hours or even days.

We consider a wide variety of situations. The measurements were performed on the ResNet-50, ResNet-101, Inception, and DenseNet-121 networks, with large image sizes of 1000×1000 and batch size of 8. Such a setting with large activations makes it difficult to train these neural networks on a single GPU. The number of GPUs varies from 2 to 8, and the available memory per GPU varies from $M = 3$ GB to $M = 16$ GB. Even though GPUs have a fixed memory size (usually 16GB), exploring lower memory values allows to assess the sensitivity of the algorithms to more constrained scenarios. These scenarios can be seen as representative of cases with larger batch size or larger image sizes. The bandwidth measured on our platform is $\beta = 12$ GB/s, and we also performed experiments with $\beta = 24$ GB/s to explore the possibilities offered by better networking capabilities.

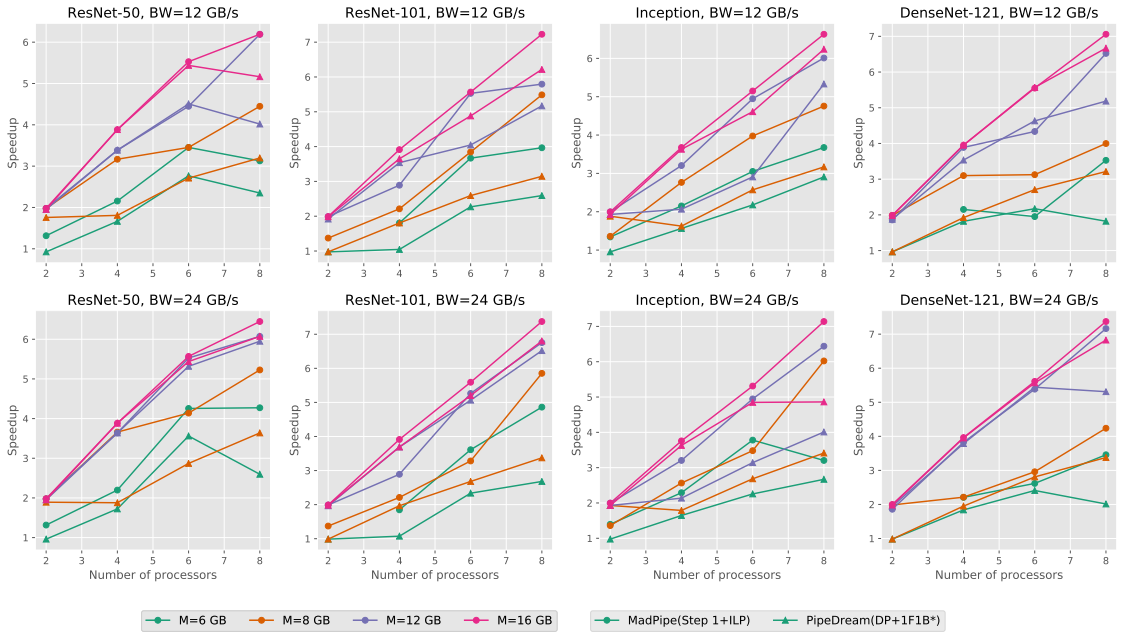


Figure 8: Speedup comparison for all networks.

5.2 Simulation Results

To save space, Figure 6 shows the simulation results only for the ResNet-50 network, but the next plots show that our conclusions apply to all networks. For both algorithms, the dashed lines represent the period of the partitionings obtained in the first phase by the respective dynamic programs, and the period of the valid schedule is depicted with a solid line. This figure presents results in terms of period duration, hence lower is better (the throughput, in terms of images processed by second, is proportional to the inverse of the period).

These plots highlight the fact that MadPipe allows to obtain significantly more efficient schedules in most cases, especially when the memory is more constrained, and for $P > 2$. The period achieved by PipeDream is routinely 20% larger than what MadPipe can provide, and in some cases the solution of PipeDream is up to two or even three times slower. The dashed lines show that the partitioning produced by PipeDream is very optimistic and expects to achieve a very small period, but then turns out infeasible, resulting in a very high overhead. On the other hand, the partitioning obtained by the first step of MadPipe has a higher period because it considers more memory constraints, but it overall results in a more efficient schedule.

We can also observe the behavior of all these algorithms when M increases. As expected, the period of the partitionings (the dashed lines) are non-increasing with M, and reach a lower bound when the memory limit is high enough to no longer be a constraint. On the other hand, the period of the schedules produced by 1F1B* (and even by MadPipe sometimes) is not monotonic: since the memory is not estimated perfectly, it may happen that with more memory available, the dynamic program finds a solution which ends up being unfeasible, and requires a higher period to be able to run. This erratic behavior is nevertheless much more visible for PipeDream.

On Figure 7, we display the results of the same simulations for all networks, normalized with respect to the MadPipe algorithm. For each case, we compute the ratio of the period obtained by a

solution to the period obtained by MadPipe. The plots show, for each value of the memory limit M and for several neural networks, the geometric mean of these ratios over all values of P and β. For the solid red line that corresponds to PipeDream (DP+1F1B*) solution, a value below 1 means that PipeDream is more efficient, and a value above 1 means that our solution is more efficient than PipeDream. This shows that the performance improvement offered by MadPipe is valid in a wide range of scenarios, especially for lower values of memory. Indeed, the overhead of PipeDream over MadPipe is consistently over 20% when the available memory is below 10GB.

Finally, we present on Figure 8 another visualisation of the same results, which aims at highlighting the scalability of the produced schedules when the number of processors increases. On this figure, the plots provide the speedup of the produced schedules compared to the sequential execution of the network, *i.e.* $U(1, L)$. We can observe that the pipelined model parallelism achieves a good scalability for settings with large memory like $M = 12$ or 16, and that MadPipe exhibits better scalability than PipeDream. When less memory is available, it is more difficult to use all the computing resources efficiently and the speedup gets worse.

Increasing the bandwidth does not dramatically improve this behavior, which shows that it is not due to communication issues. This reduced scalability when memory is tight comes from the heterogeneity between layers and from the increased memory pressure. Indeed, when the number of GPUs increases, the number of activations to be stored also increases, in particular for the first layers of the network, which generally handle the larger activations. Therefore, the memory becomes the main bottleneck and idle times have to be added to fit into the memory limit. This explains why the scalability of MadPipe is much better than what can be achieved with PipeDream.

6 Conclusion

In this paper, we consider the algorithmic optimization of model parallelism for training deep convolutional networks on large images. Model parallelism is an attractive parallelization strategy that allows in particular to avoid replicating all the weights of the network on all the computation resources. Following the ideas proposed in PipeDream [11], we propose to combine pipelining and model parallelism, which allows to achieve better resource utilization. Nevertheless, the combination of pipelining and model parallelism requires to store more activations on the nodes, which in turn induces memory issues. In this paper, we propose an optimal scheduling algorithm for the contiguous case, in which each GPU computes a contiguous set of layers, which optimizes the throughput under memory constraints. Furthermore, we design a sophisticated two-phase scheduling strategy which produces non-contiguous schedules, based on a dynamic program to group the network layers into stages. We show, using a large set of simulations on a variety of computing platforms and neural networks, that the solution we propose achieves a significant increase in throughput compared to PipeDream, especially when the memory is a strong constraint. A natural perspective of this work is to combine this approach with data parallelism to improve scalability for the tightest memory cases.

References

[1] Beaumont, O., Eyraud-Dubois, L., and Shilova, A. Pipelined Model Parallelism: Complexity Results and Memory Considerations. working paper or preprint, Oct. 2020.

[2] 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).

[3] Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[4] Das, D., Avancha, S., Mudigere, D., Vaidynathan, K., Sridharan, S., Kalamkar, D., Kaul, B., and Dubey, P. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).

[5] 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.

[6] Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., Diao, L., Liu, X., and Lin, W. Dapple: A pipelined data parallel approach for training large models, 2020.

[7] 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.

[8] 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–.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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).

[14] Park, J. H., Yun, G., Yi, C. M., Nguyen, N. T., Lee, S., Choi, J., Noh, S. H., and ri Choi, Y. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism, 2020.

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

[16] 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.

[17] 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.