Weight Offloading Strategies for Training Large DNN Models

Olivier Beaumont, Lionel Eyraud-Dubois, Alena Shilova, Xunyi Zhao

Weight Offloading Strategies for Training Large DNN Models

Olivier Beaumont*[1], Lionel Eyraud-Dubois†[1], Alena Shilova‡[1], and Xunyi Zhao§[1]

[1]Inria Bordeaux – Sud-Ouest and Université de Bordeaux, Bordeaux, France

February 18, 2022

Abstract

The limited memory of GPUs induces serious problems in the training phase of deep neural networks (DNNs). Indeed, with the recent tremendous increase in the size of DNN models, which can now routinely include hundreds of billions or even trillions of parameters, it is impossible to store these models in the memory of a GPU and several strategies have been devised to solve this problem. In this paper, we analyze in detail the strategy that consists in offloading the weights of some model layers from the GPU to the CPU when they are not used. Since the PCI bus bandwidth between the GPU and the CPU is limited, it is crucial to know which layers should be transferred (offloaded and prefetched) and when. We prove that this problem is in general NP-Complete in the strong sense and we propose a lower bound formulation in the form of an Integer Linear Program (ILP). We propose heuristics to select the layers to offload and to build the schedule of data transfers. We show that this approach allows to build near-optimal weight offloading strategies on realistic size DNNs and architectures. **Keywords:** Training of DNNs, Scheduling, Offloading Strategies

1 Introduction

Training of Deep Neural Networks (DNNs) is a computationally and memory intensive operation. From a computational point of view, there are many parallel strategies that allow to distribute the computation over several resources. These

*olivier.beaumont@inria.fr
†D.D@university.edu
‡alena.shilova@inria.fr
§xunyi.zhao@inria.fr

strategies exploit the possibility to distribute the data and treat it over several resources (data parallelism) or the DNN itself (model parallelism) or to partition the data tensors along other dimensions (tensor parallelism, filter parallelism,...).

However, contrary to the case of typical HPC applications, such as linear algebra kernels, the nature of the dependencies between the tasks prevents in general to distribute among participating resources the memory necessary to store the intermediate produced data (activations), the model parameters and the optimizer states. For example, in the data parallel approach [28, 6], the same model is replicated on all participating nodes even if the activations are actually distributed. In the model parallel approach [26, 16, 17], on the contrary, the weights are well distributed between the different resources, but to be efficient, the model parallel approach must be pipelined like in Pipedream [16, 3]. In turn, pipelined strategies require to store several models: the number of models to store, which could be large in Pipedream [16], has been reduced to two in [17] or to a constant number using a different approach in [14]. Pipedream strategies also require to significantly increase the amount of activations to store, as shown in [3].

In practice, memory needs are a major reason for the renewal of GPUs and being able to decrease memory requirements can therefore help to limit the carbon impact, since in data centers, the carbon impact associated with the production of computational resources is often more significant than the impact associated with energy consumption [9] during the exploitation phase. These memory requirements come from both the data and the models. Indeed, high dimensional data have a direct impact (usually proportional) on the size of the activations. In addition, the use of deeper models with more parameters has in general an impact on both the size of the model and the number of the activations. Obviously, high dimensional data and deeper neural networks have in general a positive impact on the accuracy, so that it is crucial in practice to design strategies limiting the memory size required on a computational node without changing the model or the data and thus the performance of the model.

Historically, minimizing the memory consumption induced by activations was the first active research axis, because in vision networks, the memory consumption related to activations was the most significant. Strategies based on re-materialization [8, 12, 10] and offloading [25, 13, 1] of activations have been proposed, as well as the combination of these two approaches [2]. Re-materialization consists in deleting some activations from memory during the forward phase and recomputing them when needed during the backward phase, which corresponds to trading memory for computation. Offloading consists in offloading activations from the (small) GPU memory to the (large) CPU memory during the forward phase, and to prefetch them back during the backward phase.

Recently, minimizing memory consumption induced by model size and optimizer states has received more attention, especially due to the explosion of model sizes. For example, in Natural Language Processing (NLP), it is not uncommon to see models with billions parameters [7, 20, 5]. A typical example of such large models is GPT-3 [5], which contains 175 billion parameters, which is about 650 GB using 32-bit floating point numbers for the weights, not counting

activations and optimizer states. As there is no GPU with such a memory size and as model parallelism increases the cost of storing activations as soon as more than a few tens of nodes are used [3], it is therefore necessary to rely on strategies to save the memory induced by model parameters.

In the literature, this problem has been addressed in several papers [21, 19, 15]. In these works, the objective is to train the largest possible model, thus to keep in the GPU memory, for each computational task related to a given layer (either forward or backward), only the data that is absolutely necessary for this task. These very aggressive optimization approaches typically lead to keeping all the weights (except those currently used) and all the states of the optimizer in the memory of the CPU, and to transfer the weights (when they are needed) from the CPU to the GPU and the gradients (as soon as they are computed) from the GPU to the CPU. This approach naturally leads to a very intensive use of the PCI bus, whose bandwidth can then become the limiting factor [15]. In the present paper, we consider the problem from an optimization point of view, focusing on the case where the training is performed on a single GPU. More precisely, we try to solve the following problem: given a set of layers defined by their sizes and processing times (forward and backward), given a PCI bus bandwidth and given a memory size, what is the strategy that minimizes the training execution time? Determining such a strategy requires solving both a selection problem (which layers should be transferred) and a scheduling problem (at what time should the transfers to and from the GPU be performed).

The paper is organized as follows. After a review of related works limited to weight offloading strategies (a more complete survey on memory saving techniques can be found in Chapter 2 of [23]) in Section 2, we present the model in Section 3 and complexity results in Section 4. Algorithms to solve the selection problem and the scheduling problems are presented in Section 5. To evaluate the quality of the proposed strategies, a lower bound based on Integer Linear Programming (ILP) is established in Section 6 and we rely on a set of simulations for transformer-based models in Section 7. Concluding remarks and perspectives are given in Section 8.

2 Related works

The methods either targeting the offloading of activations [22, 25, 1, 2], or of any tensor [13, 27, 11]. can be adjusted (though heuristically) to deal with weights in the similar way as activations. The latter group can be directly applied to weights. In [13, 11], the tensors that will not be needed for the longest time are preferred candidates for offloading. Similarly to our approach, AutoSwap [27] relies on priority scores for choosing the best candidates for offloading, though it is limited to the case of only one memory peak, which is enough for activation offloading, but not for weigths. The schedule of transfers is performed in a similar way for all above methods: offloading is performed as soon as possible and prefetching as late as possible, trying to avoid the idle time before the operation requiring the prefetched data. Despite their universality,

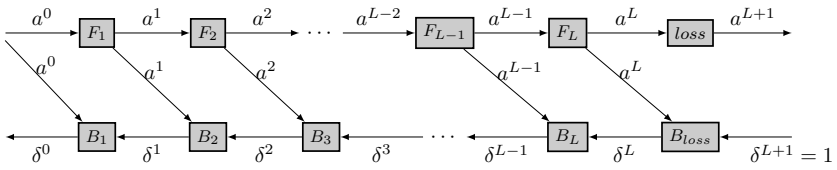

Figure 1: Data dependencies induced by the training phase of Sequential DNNs

these methods have a few disadvantages: in particular, they do not fully consider benefits of weight offloading during the backward phase contrary to the present paper.

Several recent works focus on offloading strategies of DNN weights during the training phase to limit memory consumption. The first strategy is proposed in [19] which presents the L2L (Layer-to-Layer) algorithm that we use as a basis for comparison in the Section 7. For a graph typically consisting of a sequence of encoders, a synchronous parameter-server, stored on the CPU, allows to keep a copy of the model weights and to keep on the GPU only the encoder that is being used. A strategy allowing to adapt the size of the mini-batch is then proposed, which allows to use the biggest possible mini-batch given this systematic strategy of layer offloading. Closed-form formulas allowing to estimate the training time (for all identical layers) are proposed. This strategy was then refined in ZeRO-offload [21].

Recently, in [15], the authors observed that approaches such as L2L [19] or Zero-Offload [21] generate, despite the optimizations proposed in [21], a very important data traffic on the PCI bus between the CPU and GPU. As its bandwidth is limited, it can become the bottleneck and significantly slow down the whole training process. For this purpose, it is proposed to offload only some layers, the layers from de first half being arbitrarily chosen in [15]. Our contribution in the present paper is based on the same idea, with a formalization of the optimization problem which consists in determining which layers to offload and when to do it. For this, we rely on the simplified model of [19] and we leave for future work the consideration of the optimizer states performed in [21].

3 Model

In this paper, we focus on the case of a DNN that can be written as a sequence of layers (in practice, each layer may consist of arbitrarily complex computations). DNNs are trained using forward-backward propagation that is used to compute gradients of the loss (error of the model) with respect to weights, which are later used by optimization algorithms such as Stochastic Gradient Descent. One elementary iteration of this algorithm is described in Figure 1: it consists in a traversal by a mini-batch (a set of training samples, *e.g.* images, text sequences) of the different layers of the DNN, to compute the loss and then to backpropagate

the gradients to update the network weights. We do not go into the details of the computations that can be found in [4], but it is important for us to study the dependencies and memory requirements at each step. In practice, the *loss* related operation takes very little time and does not incur additional memory costs. For simplicity, in the rest of the paper we neglect this operation and consider that B_L comes right after F_L.

Memory usage of operations The standard practice is to compute all gradients during the forward-backward propagation, and then update all the weights in an *optimization* step. This means that the weight w_i of a given layer is used in three occasions during a training step: forward of layer i (F_i), backward of layer i (B_i), and for the update of w_i in the optimization step. We propose to perform the update of w_i during backpropagation, as soon as its gradient is computed, which obviously saves memory.

As a result, each operation F_i takes as input a^{i-1} and produces a^i, which will be used by F_{i+1} and B_{i+1}. Each operation B_i takes as input a^{i-1} and δ^i and produces δ^{i-1}, while updating weights w_i of layer i from the following sequence of operations that are considered atomic:

- gradient δw_i is computed from a^{i-1} and δ^i; once this operation is performed a^{i-1} can be deleted from memory, but w_i must remain.
- δ^{i-1} is computed from δ^i and w_i; once this operation is performed δ^i can be removed from the memory, but w_i and δw_i must remain in memory.
- new w_i is computed from old w_i and δw_i; then δw_i can be removed from memory (and w_i can be offloaded), but δ^{i-1} must remain in memory to be passed to B_{i-1}.

In this paper, we assume that activations a^i are kept in memory from F_i until B_{i+1}, so that when computing B_i, all activations $|a^j|$, $0 \leq j \leq i-1$ must reside in memory. We further assume, as in the literature [4], that $|\delta^i| = |a^i|$ and $|w_i| = |\delta w_i|$.

During B_i, the memory peak is reached when δ^{i-1} is computed. We denote M_{F_i} and M_{B_i} the memory occupation during F_i and B_i (without counting the weights w_j related to other layers $j \neq i$, that can potentially be offloaded and no longer reside in memory). These values are given by:

$$M_{F_i} = |w_i| + \sum_{k \leq i} |a^k| \qquad \text{and} \quad M_{B_i} = 2|w_i| + \sum_{k \leq i} |a^k|. \tag{1}$$

Weight offloading We focus on the possibility to offload the weights (w_i) from the GPU memory to the CPU memory. We assume that there is a full-duplex communication link between the GPU and the CPU with bandwidth β, which means that we can at the same time offload a layer from the GPU and prefetch another one from the CPU. In any case, the communication buffers on

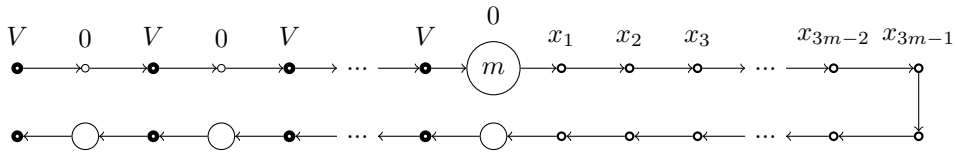

Figure 2: Instance used in the NP-completeness proof of Theorem 1. Node size represents operation times: ∘ for $u_i = 0$, ◯ for $u_i = 1$; the size of the weights is represented by node thickness and written above.

the GPU are reserved from the beginning of a prefetch operation and until the end of an offload operation.

Given that w_i is only needed for F_i and B_i, we can save memory by (a) offloading w_i after F_i and prefetch it before B_i; or (b) offloading w_i after B_i and prefetch it before F_i. In the following, we denote by *offloading* a weight w_i the action of sending w_i to the memory of the CPU, and deleting it right after. By contrast, the *uploading* operation consists in sending w_i to the CPU without deleting it on the GPU afterwards.

The problem comes however with an additional complication: only B_i modifies the value of w_i. This means that it can be beneficial to upload w_i in advance before the F_i operation (for example to benefit from an under-utilized communication link), and to delete it only after the F_i operation. It also means that between two B_i operations (on two successive batches), if w_i is offloaded after B_i, it is useless to offload it again after F_i, since that would mean rewriting the same value of w_i in the CPU memory. Therefore, it is possible that one offload operation corresponds to two prefetch operations, leading to what we call *discounted* communications. To the best of our knowledge, this *discount* possibility is not considered in other related works, and we show in Section 7 that it actually significantly improves the performance of the training phase.

We are interested in solving the following problem:

Problem 1 ($OFF(L, M_{\text{GPU}}, \beta)$)**.** *Consider a training phase with L-layers network. The operation time of forward and backward phase of the i-th layer is denoted as u_{F_i} and u_{B_i}. Given a GPU with memory M_{GPU} and bidirectional bandwidth β between the GPU and main memory, is there a schedule of uploading, delete and prefetching operations so that the training phase can be performed with an execution time at most T?*

4 Complexity Results

Theorem 1. *$OFF(L, M_{GPU}, \beta)$ is NP-complete in the strong sense.*

Proof. $OFF(L, M_{\text{GPU}}, \beta)$ clearly belongs to NP: given a schedule of all operations and communications, one can check in polynomial time that the execution is valid and fits within the execution time T.

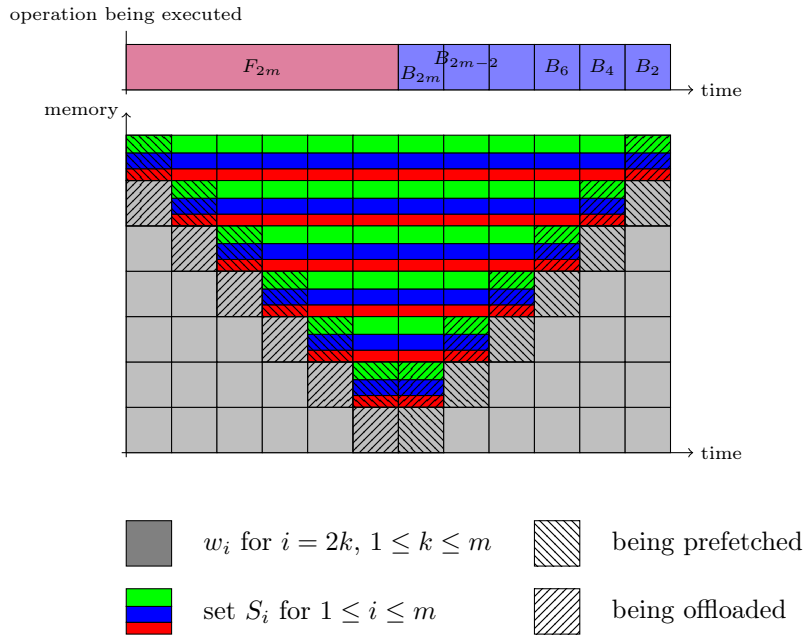

Figure 3: Instance and schedule used in the NP-completeness proof of Theorem 1.

We prove that this problem is strongly NP-hard and therefore NP-complete by a reduction from the 3-partition problem: given a set of integers $\{x_0, x_2, \ldots, x_{3m-1}\}$ such that $\sum_i x_i = mV$, decide whether it is possible to partition it into m parts $\{S_1, \ldots, S_m\}$ so that for any $j \leq m, |S_j| = 3$ and $\sum_{i \in S_i} x_i = V$. This problem is known to be NP-complete in the strong sense. Given an instance of 3-partition, we build the following instance of Problem 1:

- $L = 5m, \quad \beta = V, \quad M_{\text{GPU}} = mV + V, \quad T = 2m$
- $u_{F_i} = 0$ and $|a_i| = 0$ for all i, except $u_{F_{2m}} = m$
- $u_{B_i} = 0$ and $|w_i| = V$ for $i = 2k - 1, k \in \{1, \ldots, m\}$
- $u_{B_i} = 1$ and $|w_i| = 0$ for $i = 2k, k \in \{1, \ldots, m\}$
- $u_{B_i} = 0$ and $|w_i| = x_{i-2m}$ for $2m + 1 \leq i \leq 5m$

We claim that this instance can be scheduled in time $T = 2m$ if and only if the 3-partition instance is positive.

Let us first assume that there exists a solution to the 3-partition instance, i.e. sets $(S_j)_{1 \leq j \leq m}$ such that $\sum_{i \in S_j} x_i = V$, and let us build a feasible schedule, as described on Figure 3.

Before F_1, the GPU memory contains all w_i for $i \leq 2m$, thus the available memory is $M_{\text{GPU}} - \sum_{i=1}^{2m} |w_i| = V$. The forward operations for the first $2m$

layers take no time. With V free memory at the beginning of F_{2m}, prefetching w_{i+2m} for $i \in S_1$ and offloading w_1 can start simultaneously. After 1 time unit, both transfers are over since $|w_1| = \sum_{i \in S_1} |w_{i+2m}| = V$. Prefetching w_{i+2m} for $i \in S_2$ and offloading w_3 can thus start immediately. By the end of F_{2m}, all w_i for $i > 2m$ are prefetched into memory and all w_{2k-1} for $k \leq m$ are offloaded.

With w_i for $i > 2m$ in the memory and V free memory for the gradient, F_{2m+1} to F_{5m} and B_{5m} to B_{2m+1} can also be performed in no time. After the end of B_{2m+1}, the free memory is given by $M_{\text{GPU}} - \sum_{i=2m+1}^{5m} w_i = V$. During B_{2m}, w_{2m-1} is prefetched, and w_{i+2m} for $i \in S_1$ is offloaded, which allows B_{2m-1} to start without delay. This pattern is repeated until B_1. After B_1, all w_i for $i \leq 2m$ are in the GPU memory, and deleting δw_1 provides V free memory. A new cycle can start from there. In this schedule, all the offloading and prefetching are overlapped with the computations, which yields an execution time for one cycle of $u_{F_{2m}} + \sum_{k=1}^{m} u_{B_{2k}} = 2m$.

Let us now assume that there exists a valid schedule of duration $T = 2m$, i.e. without any idle time on the processing device. Since the operation time between F_{2m+1} and B_{2m+1} is 0, all w_i for $i > 2m$ have to be saved in memory before starting F_{2m+1} to avoid idle time. The memory usage is then $\sum_{i=2m+1}^{5m} |w_i| = mV$. If any w_{2k-1} for $k \in \{1, \dots, m\}$ is also saved in memory, the usage would be $mV + V$ and there will be no space for the gradient computed by backwards. Therefore, all w_{2k-1} for $k \in \{1, \dots, m\}$ must be offloaded before F_{2m+1}.

To start B_{2m-1} immediately after B_{2m}, w_{2m-1} has to be prefetched, and there should be at least V free memory for the gradient δw_{2m-1}. Given the state at the end of B_{2m+1} described above, the schedule needs to prefetch w_{2m-1} within time $u_{B_{2m}}$, and free enough memory to make room for δw_{2m-1}. Denote W the amount of memory offloaded during B_{2m}. The memory constraint to start B_{2m-1} without delay implies that $W \geq V$, and the bandwidth limitation implies that $W \leq V$. Together, we have $W = V$, which means that there exists a set S_1 such that $\sum_{i \in S_1} |w_{i+2m}| = V, i > 2m$. By repeating the same reasoning, we can find m subsets S_j such that $\sum_{i \in S_j} |w_{i+2m}| = V$, which gives a solution to the 3-partition problem and completes the proof. □

This proof actually shows a stronger result: even if the set of weights to offload is given, finding an optimal schedule is still NP-complete. Indeed, in the above instance, the set of weights to offload provides no information about how to solve the 3-partition instance.

5 Greedy algorithm

We now propose a greedy algorithm to solve $OFF(L, M_{\text{GPU}}, \beta)$. The algorithm works in two phases: a *selection* phase to find which weights will be offloaded, and a *schedule* phase, which computes how to interleave the prefetching and offloading operations with F and B operations during the training cycle of one batch.

Selection phase As mentioned in section 3, there are two options for offloading the weight of a layer k: (a) between F_k and B_k; or (b) between B_k and F_k. We use (k, d) to represent the offloading choices, where $d = 1$ represents choice (a) and $d = 0$ represents choice (b). A solution of the selection algorithm consists of the set of offloading choices (k, d) with $k \in \{1, \ldots, L\}$ and $d \in \{0, 1\}$. For an offloading choice (k, d), we say that it *covers* all the operations that are performed while w_k is no longer in memory, without taking into account the time taken by the communications: the choice $(k, 1)$ covers $F_{k+1}, \ldots, F_L, B_L, \ldots, B_{k+1}$, while $(k, 0)$ covers $B_{k-1}, \ldots B_1, F_1, \ldots F_{k-1}$).

Our algorithm for the selection phase is based on a scoring function which estimates the *benefit* associated to an offloading choice. The algorithm maintains a value E_o for each operation o (where $o \in \{F_1, \ldots, F_L, B_L, \ldots, B_1\}$), equal to the *excess* memory usage during operation o. Initially, the algorithm assumes that no weight is offloaded, *i.e.* $E_o = M_o + \sum_i |w_i| - M_{\text{GPU}}$, where M_o is defined by eq. (1). An offloading choice (k, d) reduces the excess memory usage of operation o by $\min(E_o, |w_k|)$, and its benefit is defined as the total memory saved over all operations o covered by this choice. The *cost* of an offloading choice w_k is simply the associated communication cost, equal to twice the size $|w_k|$ since it needs to be communicated back and forth. The *profit* of an offloading choice is then defined as the ratio between its benefit and its cost.

There is one special case for the offloading cost: if one choice (k, d) has already been selected for one layer, then using to the *discount* opportunity described in section 3, selecting the other choice $(k, 1 - d)$ only induces one communication. In that case, the cost is halved, and the profit is thus twice as large.

At the beginning, we assume that all the weights are kept in memory for every operation. As long as the maximum memory cost exceeds the available memory of GPU, the most profitable offloading choice is selected. The updated excess memory costs E_o will change the benefits of each choice for the next run. The resulting algorithm is described in Algorithm 1.

Scheduling phase Once the offloading choices have been selected, another greedy algorithm is used to build an actual schedule. Our algorithm relies on simple ordering rules: prefetching operations are performed in the order in which the data is required: if both w_i and w_{i+1} are offloaded after the forward phase, w_{i+1} will be prefetched before w_i since it will be needed in the backward phase earlier. Offloading operations are performed in First-In-First-Out (FIFO) order, which means that data produced earlier is offloaded first. There is an exception to this: F_k does not change w_k, and thus does not produce data to be offloaded. So if $(k, 1)$ is selected in Algorithm 1, it is possible to schedule the corresponding communication *before* the forward operation. However, this is not required and we use this possibility on a best-effort basis.

Given these rules, our scheduling algorithm described in Algorithm 2 performs each operation as soon as possible. In general, an operation needs to wait for three events: (1) its input data is available (2) its resource is available (either

Algorithm 1 Greedy Selection Algorithm

Input: Cost of each operation, memory usage of each layer
Output: selection of which layer weights to offload

```
for all operations o ∈ {F_1, ..., loss, B_loss, ..., B_1} do
    E_o = Σ_i |w_i| + M_o − M_GPU
while max_o(E_o) > 0 do
    for all choices (k, d) not yet selected do
        S ← set of operations covered by (k, d)
        if (k, 1 − d) is chosen then
            profit(k, d) = · Σ_{o∈S} min(E_o, |w_k|)/|w_k|
        else
            profit(k, d) = 1/2 Σ_{o∈S} min(E_o, |w_k|)/|w_k|
    Select the choice (k, d) = arg max_{k,d} profit(k, d)
    S ← set of operations covered by (k, d)
    Recompute E_o for all o ∈ S: E_o ← max{E_o − |w_k|, 0}
```

the GPU for a computing operation, or the corresponding communication link) (3) enough memory is available on the GPU (not applicable for an offloading operation). We say that an operation is *feasible* at time t if constraint (3) is satisfied. Algorithm 1 ensures that with our ordering rules all operations eventually become feasible. In Algorithm 2, Pre_k and Off_k denote prefetch and offload operations of w_k, $T_{ava}(\cdot)$ denotes the available time of a communication link and $T_{end}(\cdot)$ denotes the ending time of an operation in Algorithm 2.

The scheduling process performs the operations in the order $B_L, \ldots, B_1$, followed by $F_1, \ldots, F_L$. With this order, it is easier to perform the early, best-effort scheduling of the offloading operations associated with a forward operation, as mentioned above: while the communications related to the backward operations are scheduled, we can keep a waiting list WL of best-effort offload opportunities, and schedule them whenever the communication link is available. However, the scheduling algorithm needs to know which data have already been offloaded or prefetched at the start of B_l. With our ordering rules, we can try all possibilities by selecting a cutoff point in the sequence of communication operations that needs to be performed, assuming that all communications before the cutoff have been performed. Algorithm 2 returns the solution with the smallest execution time among all possibilities.

The time complexity of this greedy algorithm is $O(L^3)$. In practice, it takes less than 1 minute to find a solution for a DNN with up to 128 layers.

6 Lower bound

In this section, we present an original Mixed Integer Linear Programming formulation for the $OFF(L, M_{\text{GPU}}, \beta)$ problem. Given the difficulty of the problem, we do not propose an exact formulation, but one that provides a valid lower

Algorithm 2 Greedy Scheduling Algorithm

for all cutting points in the sequence of communications **do**
  Assume communications before the cutting point are performed before B_L
  for k in $[L, \dots, 1]$ **do**
    if $(k, 1)$ has been selected **then** ▷ B_k needs Prefetch
      Schedule Pre_k at time max($T_{ava}(Pre)$, earliest time when Pre_k is feasible)
    Schedule B_k at time max($T_{end}(Pre_k)$, $T_{end}(B_{k+1})$)
    if $(k, 0)$ has been selected **then** ▷ B_k needs Offload
      while $T_{end}(B_k) - T_{ava}(Off) > \min_{k' \in WL} |w_{k'}|$ **do**
        Offload $\min |w_{k'}|$ from WL, without deleting it from GPU
      Schedule Off_k at time max($T_{end}(B_k)$, $T_{ava}(Off)$)
    else
      if $(k, 1)$ has been selected **then** ▷ F_k needs Offload
        Append k to WL ▷ WL is initially empty
  for k in $[1, \dots, L]$ **do**
    if $(k, 0)$ has been selected **then** ▷ F_k needs Prefetch
      Schedule Pre_k at time max($T_{ava}(Pre)$, earliest time when Pre_k is feasible)
    Schedule F_k at time max($T_{end}(Pre_k)$, $T_{end}(F_{k-1})$)
    if $(k, 1)$ has been selected **then** ▷ F_k needs Offload
      if w_k is not yet in CPU memory **then**
        Schedule Off_k at time max($T_{end}(F_k)$, $T_{ava}(Off)$)
      else
        Delete w_k from GPU memory at time max($T_{end}(F_k)$, $T_{ava}(Off)$)
  Finish communications before the cutting point
return the solution with lowest execution time

bound on the makespan: the result of this MILP is not a schedule, because it satisfies only a subset of the constraints of $OFF(L, M_{\text{GPU}}, \beta)$. This lower bound will be used in Section 7 to evaluate the quality of the schedules produced by the algorithms of Section 5 and the algorithms of the literature.

We number the operations from 1 to $2L$, starting from B_L, so that the L-th operation is B_1, F_i is the $(L+i)$-th operation, and B_i is the $(L-i+1)$-th operation. In the following, we use $\overleftrightarrow{\sum}_{j=a}^{b}$ to denote a sum from operation a to operation b, potentially cycling back to 1 if $b < a$: in that case, $\overleftrightarrow{\sum}_{j=a}^{b} X_j = \sum_{j=a}^{2L} X_j + \sum_{j=1}^{b} X_j$.

For each operation j, our formulation considers the interval between the start of operation j and the start of the next operation. Let us define the following continuous variables:

- for all i, $O_{i,j}$ represents the amount of data from w_i offloaded from GPU to CPU (note that this transfer does not change the GPU memory used)
- for all i, $P_{i,j}$ represents the amount of data from w_i prefetched from CPU
- for all i, $D_{i,j}$ represents the amount of data from w_i deleted from the GPU
- $idle_j$ represents the idle time after operation j

In addition, for each layer i we define three binary variables, that encode offloading choices:

- $D_i^{(1)} = 1$ if w_i is removed from memory between F_i and B_i (similar to the choice $(i, 1)$ in Algorithm 1), and 0 otherwise.
- $D_i^{(0)} = 1$ if w_i is removed from memory between B_i and F_i (similar to the choice $(i, 0)$ in Algorithm 1), and 0 otherwise.
- $O_i^* = 1$ if w_i is offloaded at some point, and 0 otherwise.

The constraints of our formulation can be classified in four groups:

Idle time. We make sure the communications are finished before going to the next operation, with a special case to ensure that the transfer of w_i does not take place during its backward:

$$\forall j, \quad u_j + idle_j \geq \sum_i O_{i,j}/\beta \qquad \text{and} \quad u_j + idle_j \geq \sum_i P_{i,j}/\beta \tag{2}$$

$$\forall i, \quad idle_{L-i+1} \geq O_{i,L-i+1}/\beta \tag{3}$$

Memory constraints. Between operations j and $j+1$, $(P_{i,j} - D_{i,j})$ indicates the change in memory usage associated with w_i. We define $m_{i,k}$ as the memory usage of w_i at the beginning of operation k. Since w_i is completely in GPU memory before B_i, we get $m_{i,(L-i+1)} = |w_i|$. We can thus deduce $m_{i,k} = |w_i| + \overleftrightarrow{\sum}_{j=L-i+1}^{k-1} P_{i,j} - D_{i,j}$. Recall that the memory peak during operation k is M_k, as defined by eq. (1). The global memory constraint is given by

$$\forall k, \sum_i m_{i,k} \leq M_{\text{GPU}} - M_k \tag{4}$$

Validity constraints. We ensure that the cycle is stable, and that w_i is completely in GPU at the start of F_i (operation $L+i$):

$$\forall i, \quad \sum_{j=0}^{2L-1} P_{i,j} - D_{i,j} = 0 \qquad \text{and} \qquad \sum_{j=L-i+1}^{L+i-1} P_{ij} - D_{ij} = 0 \tag{5}$$

Additionally, we write constraints on w_i at the beginning of every operation k to ensure that the amount of prefetched data is not higher than the deleted data, and that no data is lost when deleting from GPU memory:

$$\forall i, k, \quad \overleftrightarrow{\sum_{j=L-i+1}^{k-1}} P_{i,j} - D_{i,j} \geq 0 \quad \text{and} \quad \overleftrightarrow{\sum_{j=L-i+1}^{k-1}} O_{i,j} + P_{i,j} - D_{i,j} \geq 0 \tag{6}$$

Binary constraints. Finally, we add constraints to ensure that the weight of a given layer is either deleted/offloaded completely, or not at all.

$$\forall i, \quad \sum_{j=1}^{2L} O_{i,j} = |w_i| O_i^* \tag{7}$$

$$\forall i, \quad \overleftrightarrow{\sum_{j=L+i}^{L-i}} D_{i,j} = |w_i| D_i^{(1)} \qquad \text{and} \qquad \sum_{j=L-i+1}^{L+i-1} D_{i,j} = |w_i| D_i^{(0)} \tag{8}$$

Obviously, any solution to $OFF(L, M_{\text{GPU}}, \beta)$ can be transformed into a set of variables that fulfill these contraints. We thus obtain the following lower bound:

Theorem 2. *The optimal execution time T^* of problem $OFF(L, M_{GPU}, \beta)$ is not smaller than the optimal value of the MILP defined by:*

$$\min \sum_j u_j + idle_j$$
$$\textit{s.t.} \quad \textit{eqs. (2) to (8)}$$

Note that the opposite is not true: it is not clear that any solution to this MILP can be turned into a valid solution of $OFF(L, M_{\text{GPU}}, \beta)$. Indeed, the memory constraints (4) are only expressed at the beginning of each operation, instead of during the while interval from the beginning of an operation to the next. The solution of the MILP therefore only provides a lower bound, not the optimal value of $OFF(L, M_{\text{GPU}}, \beta)$.

7 Simulation results

To estimate time and memory costs when training GPT-2 and Bert models, we rely on the hyper-parameters from Megatron-LM [24]. We test both DNNs with a number of layers varying from 36 to 144 and batch sizes of 16, 32 and 64. For both models, we use the hidden size of 3072 and 24 attention heads. The number of parameters ranges from 4.2B to 16.4B. For a batch size of 16, about 20MB of activations are saved in GPU memory per layer, while about 1GB to 2GB intermediate activations will be recomputed for each layer during the backward. The execution times for the simulations are experimentally obtained on a GPU V100 with 16GB memory. Since a large memory is needed to store the intermediate activations generated in each transformer block, M_{GPU} used for simulation is smaller than 16GB and depends on the batch size. IBM ILOG CPLEX [18] is used to find the solution for the ILP as described in Section 6.

7.1 Results

Table 1 provides the simulation results using the measurements from real machines. The lower bound defined in Section 6 gives no idle time in all settings. Our greedy algorithm defined in Section 5 shows about 1% of overhead with respect to the lower bound. It demonstrates a significant improvement compared to the overhead of L2L [19], which is 10% to 50% larger than the lower bound. The performance of L2L highly depends on the processing times, that are approximately linear with respect to the batch size. We also notice that offloading weights more than once does not bring a significant change to the performance of the greedy algorithm in these simulations.

7.2 Bandwidth

To study the effect of bandwidth, another set of simulations has been performed for different values of β. The results with the ILP show almost no idle time when the bandwidth is higher than 4GB/s. The comparison between blue and green plots shows that our greedy algorithm is very close to the lower bound for a given bandwidth. The L2L plot indicates that L2L always causes a reduction in throughput compared to our greedy algorithm, less significant when both algorithms need long transfer times. Also, we notice that Greedy* is always slower than Greedy, which means that the discount is useful, especially for certain bandwidth ranges. In some cases, not considering the discount opportunity

Model				Duration (ms)			
Network	Depth	Batch_size	$\sum_j u_j$	ILP	Greedy	L2L	Greedy*
GPT-2	74	64	47421	47421	47493	52707	47493
GPT-2	56	64	35553	35553	35625	39542	35625
GPT-2	38	64	23694	23694	23838	26387	23838
GPT-2	74	32	23697	23697	23769	28983	23769
GPT-2	56	32	17762	17762	17834	21751	17834
GPT-2	38	32	11840	11840	11948	14533	11948
GPT-2	74	16	11612	11612	11684	16898	11715
GPT-2	56	16	8697	8697	8769	12686	8800
GPT-2	38	16	5794	5794	5902	8487	5934
Bert	144	64	34486	34486	34499	38379	34499
Bert	96	64	22965	22965	22978	25577	22978
Bert	144	32	17443	17443	17483	21336	17483
Bert	96	32	11617	11617	11657	14230	11657
Bert	144	16	9090	9090	9183	12983	9215
Bert	96	16	6058	6058	6085	8670	6085

Table 1: The bidirectional bandwidth is 12GB/s for both prefetch and offload. Greedy* represents the Greedy algorithm while weights are not only offloaded once.

can induce up to 5 times more idle time.

8 Conclusion

In this paper, we consider GPU memory savings during training, by using the possibility to store the network layers in the CPU memory when they are not used. We propose a sophisticated model for the memory associated with the storage of layer weights, which takes into account all opportunities to save memory: by updating weights as soon as possible and by performing only one offload per forward/backward cycle. We have established the NP-completeness of the layer selection and transfer scheduling problem, and we have proposed an efficient heuristic to solve it in practice. The perspectives opened by this work are numerous. The extension to the offloading of optimizer states and activations are natural perspectives, as well as the possibility to perform weight updates on the CPU. Moreover, the proposed strategy is more efficient when the batch size is large, which suggests combining it with re-materialization approaches to increase the batch size.

References

[1] Beaumont, O., Eyraud-Dubois, L., Shilova, A.: Optimal gpu-cpu offloading strategies for deep neural network training. In: European Conference on

Parallel Processing. pp. 151–166. Springer (2020)

[2] Beaumont, O., Eyraud-Dubois, L., Shilova, A.: Efficient combination of rematerialization and offloading for training dnns. Advances in Neural Information Processing Systems **34** (2021)

[3] Beaumont, O., Eyraud-Dubois, L., Shilova, A.: Pipelined model parallelism: Complexity results and memory considerations. In: European Conference on Parallel Processing. pp. 183–198. Springer (2021)

[4] Bottou, L.: Large-scale machine learning with stochastic gradient descent. In: Proceedings of COMPSTAT'2010, pp. 177–186. Springer (2010)

[5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)

[6] Chu, C.H., Kousha, P., Awan, A.A., Khorassani, K.S., Subramoni, H., 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. pp. 1–12 (2020)

[7] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)

[8] Gruslys, A., Munos, R., Danihelka, I., Lanctot, M., Graves, A.: Memory-efficient backpropagation through time. In: Advances in Neural Information Processing Systems. pp. 4125–4133 (2016)

[9] Gupta, U., Kim, Y.G., Lee, S., Tse, J., Lee, H.H.S., Wei, G.Y., Brooks, D., Wu, C.J.: Chasing carbon: The elusive environmental footprint of computing. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). pp. 854–867. IEEE (2021)

[10] Herrmann, J., Beaumont, O., Eyraud-Dubois, L., Hermann, J., Joly, A., Shilova, A.: Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. arXiv preprint arXiv:1911.13214 (2019)

[11] Huang, C.C., Jin, G., Li, J.: Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 1341–1355 (2020)

[12] Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Keutzer, K., Stoica, I., Gonzalez, J.E.: Checkmate: Breaking the memory wall with optimal tensor rematerialization (2019)

[13] Le, T.D., Imai, H., Negishi, Y., Kawachiya, K.: Tflms: Large model support in tensorflow by graph rewriting. arXiv preprint arXiv:1807.02037 (2018)

[14] Li, S., Hoefler, T.: Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14 (2021)

[15] Lin, J., Yang, A., Bai, J., Zhou, C., Jiang, L., Jia, X., Wang, A., Zhang, J., Li, Y., Lin, W., et al.: M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining. arXiv preprint arXiv:2110.03888 (2021)

[16] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N.R., Ganger, G.R., Gibbons, P.B., Zaharia, M.: PipeDream: generalized pipeline parallelism for DNN training. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 1–15 (2019)

[17] Narayanan, D., Phanishayee, A., Shi, K., Chen, X., Zaharia, M.: Memory-efficient pipeline-parallel dnn training. In: Meila, M., Zhang, T. (eds.) Proceedings of the 38th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 139, pp. 7937–7947. PMLR (18–24 Jul 2021), https://proceedings.mlr.press/v139/narayanan21a.html

[18] Nickel, S., Steinhardt, C., Schlenker, H., Burkart, W., Reuter-Oppermann, M.: Ibm ilog cplex optimization studio. In: Angewandte Optimierung mit IBM ILOG CPLEX Optimization Studio, pp. 9–23. Springer (2020)

[19] Pudipeddi, B., Mesmakhosroshahi, M., Xi, J., Bharadwaj, S.: Training large neural networks with constant memory using a new execution algorithm. arXiv preprint arXiv:2002.05645 (2020)

[20] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research **21**(140), 1–67 (2020), http://jmlr.org/papers/v21/20-074.html

[21] Ren, J., Rajbhandari, S., Aminabadi, R.Y., Ruwase, O., Yang, S., Zhang, M., Li, D., He, Y.: {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21). pp. 551–564 (2021)

[22] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., Keckler, S.W.: vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture. p. 18. IEEE Press (2016)

[23] Shilova, A.: Memory Saving Strategies for Deep Neural Network Training. Ph.D. thesis, PhD thesis, University of Bordeaux, France (2021)

[24] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053 (2019)

[25] Shriram, S., Garg, A., Kulkarni, P.: Dynamic memory management for gpu-based training of deep neural networks. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE Press (2019)

[26] Zhan, J., 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). pp. 55–60. IEEE (2019)

[27] Zhang, J., Yeung, S.H., Shu, Y., He, B., Wang, W.: Efficient memory management for gpu-based deep learning systems. arXiv preprint arXiv:1903.06631 (2019)

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

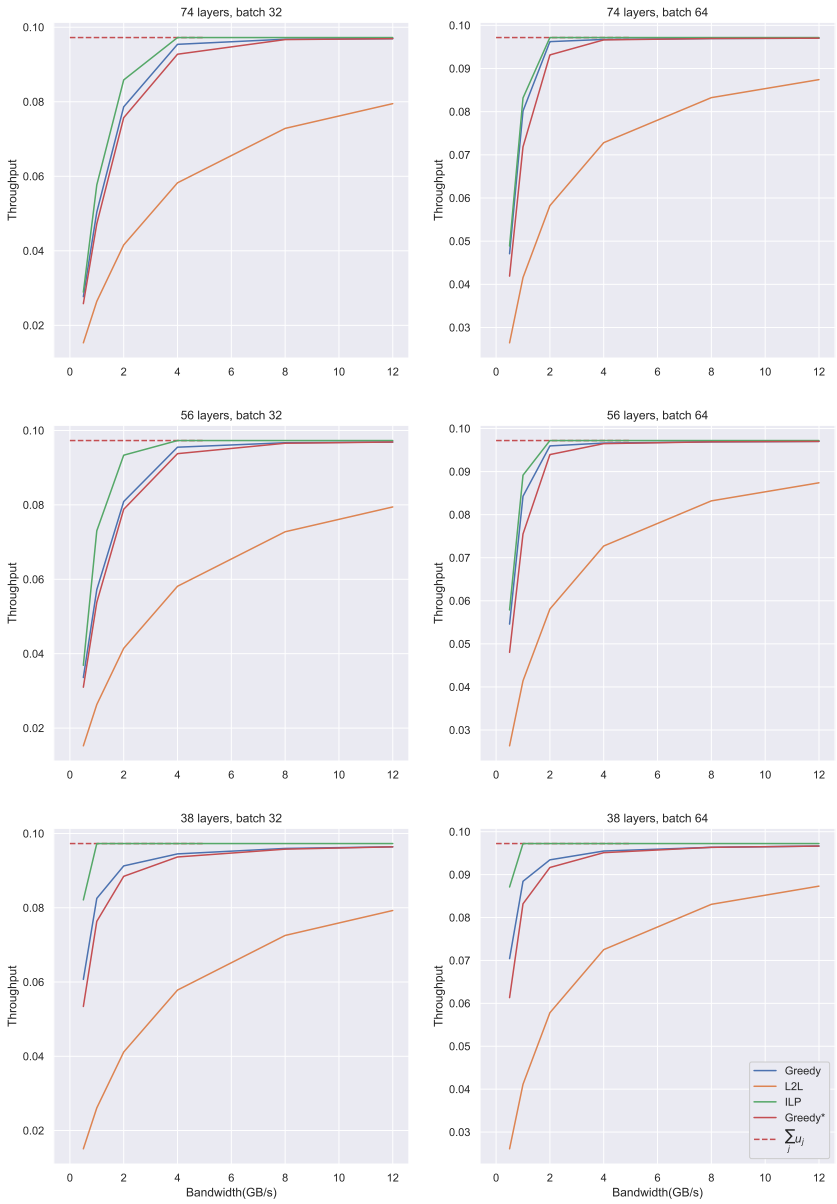

Figure 4: To study the effect of bandwidth, we use the same measurement showed in Table1, while the bandwidth varies from 0.5,1,2,4,8 and 12GB/s. The batch size is chosen as 32 or 64.