Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training

Olivier Beaumont[1,2(✉)], Lionel Eyraud-Dubois[1,2], and Alena Shilova[1,2]

[1] Inria Bordeaux – Sud-Ouest, Bordeaux, France
{olivier.beaumont,lionel.eyraud-dubois,alena.shilova}@inria.fr
[2] University of Bordeaux, Bordeaux, France

Abstract. Training Deep Neural Networks is known to be an expensive operation, both in terms of computational cost and memory load. Indeed, during training, all intermediate layer outputs (called activations) computed during the forward phase must be stored until the corresponding gradient has been computed in the backward phase. These memory requirements sometimes prevent to consider larger batch sizes and deeper networks, so that they can limit both convergence speed and accuracy. Recent works have proposed to offload some of the computed forward activations from the device memory to the main memory and requires to determine which activations should be offloaded and when these transfers should take place. We prove that this problem is NP-complete in the strong sense, and propose two heuristics based on relaxations of the problem. We then conduct a thorough experimental evaluation of standard deep neural networks.

Keywords: Memory management · Deep Neural Network · Dynamic programming · Scheduling

1 Introduction

Training for Deep Learning Networks (DNNs) has become a major compute intensive application [9,10], typically performed on GPU clusters. The training phase involves two traversals of the graph representing the DNN, one in direct order which is called forward propagation and one in reverse order called backward propagation. This incurs high memory usage: the tensors computed during the forward phase, called forward activations, must be kept in memory until the associated backward operation is performed, since they are required to compute the gradients and to update the weights. Therefore, memory issues become crucial when performing training in DNNs, and the memory limitation of current hardware often prevents data scientists from considering larger models, larger image sizes or larger batch sizes [15,18].

For instance, when using ResNet101 with relatively small images of size 224×224 and a batch size of 32, the resulting size during training is around 5GB. For applications which require to detect small objects in the images [4], the image

M. Malawski and K. Rzadca (Eds.): Euro-Par 2020, LNCS 12247, pp. 151–166, 2020.
https://doi.org/10.1007/978-3-030-57675-2_10

resolution must be increased, and the memory required for storing activations increases quadratically with the image resolution. The situation is even worse when moving to 3D object recognition [6] or video based DNNs such as 3D-Resnet [8] or CDC [19].

Many approaches have been proposed in the literature in order to circumvent this memory issue. In this paper, we focus on an *offloading* approach (also called *memory swapping*), which consists in reducing memory usage on the GPU (device memory) by transferring some activations to the CPU (main memory), which is expected to be at least one order of magnitude larger. The corresponding algorithmic question is to determine which activations should be offloaded and when, and also when offloaded activations should be brought back (prefetched) from the main memory to the device memory. This approach has been recently considered in [1,13,14,16,20,21], where the authors advocate the general idea and propose several static heuristics to decide which activations should be offloaded. In this paper, we provide a deeper analysis of this problem. More specifically, we prove that the general problem, even for sequential models, is strongly NP-complete where only fully integral data transfers are possible and we analyze two relaxations of the problem for which we can derive optimal algorithms. These algorithms can then be used as heuristics for the general problem.

The rest of the paper is organized as follows. In Sect. 2, we discuss previous works regarding offloading, as well as other techniques to reduce memory usage during the training phase. In Sect. 3, we present the model and notations used throughout the paper, and assess the complexity of the problem. In Sect. 4, we propose a first relaxation where activations can be partially or completely offloaded into the main memory, and derive an optimal strategy. In Sect. 5, we consider the case where partial offloading is not possible, but where communications can be interrupted, and we present a dynamic programming algorithm to find the optimal schedule. In Sect. 6, we provide experimental results and we assess the efficiency of our heuristics against the previous approach [1,13,21], before presenting conclusions and perspectives in Sect. 7.

2 Related Work

In order to reduce the memory usage of storing the forward activations on a processing device, we can identify two kinds of approaches: checkpointing or offloading.

Checkpointing techniques consist selecting only a few activations that are kept in memory, and then to dynamically recompute the others at runtime. This allows to explore a tradeoff between memory usage and computational cost. The use of checkpointing strategies has recently been advocated for DNN in several papers [5,7,11], where it is referred as gradient checkpointing or rematerialization.

Offloading is a potentially complementary approach first proposed in [16]. In [16], the authors propose a simple and effective mechanism of memory virtualization, that nevertheless introduces unnecessary idle time by enforcing some synchronization between data transfers and computations of later forward activations. This approach has been later improved in [1]. Nevertheless, in both papers, the algorithmic strategies to decide which activations to offload into the main memory are relatively straightforward. Proposed strategies consist in trying to offload either all activations or only those that correspond to convolutional layers (either all convolutions or every second one). Indeed, convolutional layers are known to induce a large computational time with respect to their input size, which make them good candidates to overlap offloading and processing.

Several follow-up works offer improvements over this first attempt. In order to reduce the overhead incurred by the communications, some authors [17] recommend to add compression to decrease the communication time, while others [12] design a memory-centric architecture to help with data transfers. In [13,14], the authors implement memory virtualization by manipulating the computational graphs and inserting special operations called *swap in* and *swap out* that send the activations in and out of the device memory. Such an approach can be applied to any arbitrary Computation Graph that represent neural network training graphs. The authors of [13] improve the candidate selection and prefetching mechanisms by introducing thresholds to filter out different possibilities. Moreover, some works try to combine offloading with other memory optimizing techniques. Memory swapping and memory pooling are implemented together in [21], where candidates for swapping are found by assigning priority scores to all activations. Finally, gradient checkpointing is combined with the simple offloading approach from [16] in [20].

As a complement to these practical approaches, in this paper we perform the first theoretical analysis of the underlying optimization problem and present both a complexity proof and optimal solutions to two of its relaxations.

3 Model and Complexity

3.1 Computation Model

We consider the training phase of sequential DNNs, as depicted on Fig. 1. This training phase consists of two types of computations: forward propagations $(F_i)_{1\leq i\leq L}$ and backward propagations $(B_i)_{1\leq i\leq L}$. The forward step F_i requires x_i as input, and computes x_{i+1}. The backward step B_i requires x_{i+1}, x_i and y_{i+1} as inputs, and computes y_i. The objective of the elementary training phase

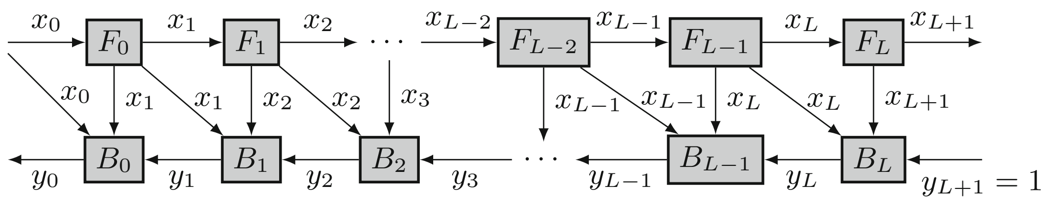


Fig. 1. Data dependencies induced the training phase of Sequential Deep Neural Networks.

is to perform the whole computation and to obtain y_0 in the smallest possible time. This computation is performed on a processing device (typically a GPU or TPU) with limited memory M_{GPU}. We denote u_{F_i} the time to process F_i, and u_{B_i} the time to process B_i. As mentioned before, the training phase is very memory intensive: since they will be needed for the backward phase, all x_i values must be stored during the forward phase, and they can only be freed once their corresponding B_i operation has been performed.

We will use the following memory model. Each data (x_i and y_i) has a given memory usage, denoted respectively by $|x_i|$ and $|y_i|$. To perform an operation (either F_i or B_i), it is necessary to have all inputs stored into memory, to reserve the memory space to store the output, and to reserve space for the temporary memory usage of the operation, denoted with ex_i^F for F_i and ex_i^B for B_i. For example, running the F_i operation requires to have at least $|x_i| + |x_{i+1}| + \text{ex}_i^F$ memory available.

In order to decrease memory usage, we assume that it is possible to *offload* some of the forward data to another memory storage (typically the main memory of the machine). The size of this memory is assumed to be large enough to store all the results and thus is not a constraint; but the speed of data transfers is limited by bandwidth β. The offloaded data can then be *prefetched* during the backward phase, so that it is available when needed to perform the corresponding backward operation. Such memory hierarchy has been considered by [1,16] as well, *i.e.* there are one GPU with limited memory and one CPU with large enough memory to store all activations of some arbitrary neural network and both are connected with the network with the bandwidth β, which we assume is fully used for any communications. More complicated cases such as multiple GPU and one CPU are out of scope of this paper and they will be left for the future work. Additionally, we assume that transfers and computations could be overlapped while only one transfer at a time is possible. Let us point out that generally x_i needs to be stored in memory in its entirety throughout the transfer: during the offloading, the memory is only released after the complete transfer, and during the prefetching, the memory is reserved as soon as the transfer begins.

We can state the decision problem associated to offloading.

Problem 1 (Offloading). Consider a training phase with L operations, with processing times u_{F_i} and u_{B_i}, data sizes $|x_i|$ and $|y_i|$, temporary memory usage ex_i^F and ex_i^B, where $0 \leq i \leq L$. Is it possible to perform this computation on a processing device with memory M_{GPU} and bandwidth β between processing device and main memory, with an execution time at most T?

3.2 Preliminary Results and Lower Bound

Proposition 1. *For fixed decisions of which data to offload, and in which order transfers should be performed, the best schedule is obtained with a* no-wait *policy, where each action (computation and data transfers) is performed as early as possible, as soon as data is available and there is enough memory.*

Given the activation sizes and the temporary memory usage, it is easy to compute the total amount of used memory (both on the computing device and in the additional memory) during the execution of each operation. We denote by m_{F_i} or m_{B_i} the amount of data required to be stored on both devices to perform F_i or B_i respectively. Let us additionally denote as M_{peak} the maximum of these values: $m_{F_i} = \mathrm{ex}_i^F + \sum_{j \le i+1} |x_j|$, $m_{B_i} = \mathrm{ex}_i^B + |y_i| + |y_{i+1}| + \sum_{j \le i+1} |x_j|$, *i.e.* $M_{peak} = \max\left(\max_{0 \le i \le L} m_{B_i}, \max_{0 \le i \le L} m_{F_i}\right)$.

Since any valid schedule must process the operation which achieves the memory peak while using at most M_{GPU} memory on the computing device, the following result holds.

Proposition 2. *The amount of data offloaded by any valid schedule is at least* $M_{peak} - M_{GPU}$.

Since any valid schedule must perform all computations, and must transfer at least this amount of data twice (for offloading and prefetching), the following lower bound on the optimal makespan holds true.

Proposition 3. *The value* $LB = \max(\sum_i u_{F_i} + u_{B_i}, 2\frac{M_{peak} - M_{GPU}}{\beta})$ *is a lower bound on the optimal makespan.*

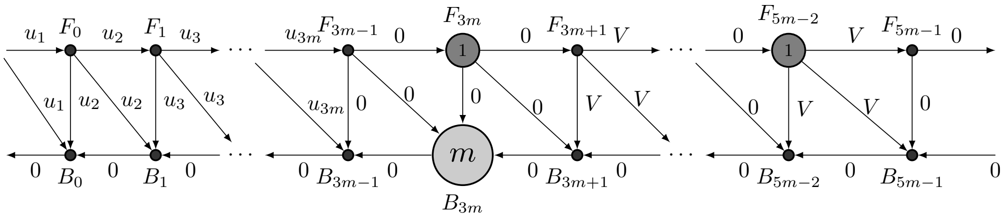
3.3 Complexity Results

Theorem 1. *Problem 1 is strongly NP-complete.*

Proof. Problem 1 clearly belongs to NP: given the start time of all forward and backward operations, and the set of offloaded data with the corresponding start time of transfers, checking that the schedule satisfies all constraints can be done in linear time.

We prove that this problem is strongly NP-hard and therefore strongly NP-complete by a reduction from the 3-partition problem: given a set of integers $\{u_0, u_2, \ldots, u_{3m-1}\}$ such that $\sum_i u_i = mV$, is it possible to partition it into m parts $\{S_1, \ldots, S_m\}$ so that for any $j \le m$, $|S_j| = 3$ and $\sum_{i \in S_j} u_i = V$. This problem is known to be NP-complete in the strong sense. Given an instance of 3-partition, we consider the following instance of Problem 1, depicted on the Figure below:

– $L = 5m$, $\beta = V$, $M_{\text{GPU}} = mV$, $T = 2m$;
– $u_{F_i} = 0$ and $|x_i| = u_i$ for $1 \le i < 3m$;
– $u_{F_i} = 1$ and $|x_i| = 0$ for $i = 3m + 2k, 0 \le k < m$;
– $u_{F_i} = 0$ and $|x_i| = V$ for $i = 3m + 2k + 1, 0 \le k < m$;
– $u_{B_i} = 0$ and $|y_i| = 0$ for all i, except $u_{B_{3m}} = m$.



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 $(S_j)_{1 \le j \le m}$ such that $\sum_{i \in S_j} u_i = V$. We can build a schedule which starts F_{3m+2k} at time k for $0 \le k < m$, and executes B_{3m} from time m to time $2m$. At time 0, before the execution of F_{3m}, the memory usage is exactly $mV = \sum_i u_i$. During the execution of F_{3m+2k}, activations x_i for $i \in S_k$ are transferred. Since $\beta = V$, this takes time exactly 1. The memory used at the end of F_{3m+2k} is thus $(m-1)V$, which allows to immediately start $F_{3m+2k+1}$. At the end of the forward phase, the memory is filled with m activations of size V. At the beginning of B_{3m}, the memory is empty: all activations of size u_i can be prefetched during the execution of B_{3m}, allowing to finish the backward phase. This schedule induces no idle time, and finishes in time exactly $T = 2m$.

Let us now assume that there exists a valid schedule of duration $T = 2m$, *i.e.* without any idle time on the processing device. For $j < m$, let us define the set S_j as the indices of the activations whose transfers are included in the execution of F_{3m+2j}. Since F_{3m+1} starts immediately after the end of F_{3m}, and since memory is only released once the transfer has been completed, the amount of data sent during F_{3m} is at least V. Since $\beta = V$ and $u_{F_{3m}} = 1$, the amount of data is exactly V, thus $\sum_{i \in S_0} u_i = V$. The same argument applies for all $j < m$, which shows that the sets S_j are a valid solution for the 3-partition instance, and completes the proof. □

From the proof of Theorem 1 follows that even when we know which activations should be offloaded, it is difficult to decide the order in which the transfers should be done. Indeed, it is clear in the instances used in the proof that the first $3m$ activations need to be offloaded, but finding the optimal ordering is hard. Because of this negative complexity result, we study two different relaxations of Problem 1 in the next sections, by relaxing the constraints stating that activations must be sent in entirety before the corresponding memory can be released. In such scenarios, all activations can be sent as soon as they are computed, *i.e.* in increasing order of their indices. This allows to compute optimal solutions in reasonable time, and the resulting algorithms can then be used as heuristic solutions for Problem 1.

4 Fractional Relaxation

In a first relaxation, let us consider that it is possible to perform partial offloading: any communication can be stopped at any time, and the data that has been transferred up to that time can be released from memory, even if the rest of the activation is still present on the computing device. With this model, it is possible to compute an optimal solution with a greedy algorithm. Let us first prove results about the structure of optimal solutions, and then use that structure to design an optimal greedy algorithm.

Structure of Optimal Solutions. In this section, let us analyze special *eager* schedules. A schedule is said *eager* if it offloads the first k activations $x_0, x_1, \ldots, x_k$ (where the last one can be partially offloaded). A schedule is said *ordered* if the data is offloaded in order of increasing indices, and prefetched in order of decreasing indices.

Lemma 1. *Any valid solution $\mathcal{S}$ can be transformed into a eager and ordered solution $\mathcal{S}'$ with the same makespan.*

Proof. Let us denote by M_{off} the amount of activation data offloaded by the schedule $\mathcal{S}$, and let us consider in $\mathcal{S}$ the time intervals $\mathcal{I}_{\text{off}}$ spent offloading data, and the time intervals $\mathcal{I}_{\text{fetch}}$ spent prefetching data. Let us consider the schedule $\mathcal{S}'$ in which all operations and data transfers are performed at the same instants as in $\mathcal{S}$, only changing which data is transferred. The first intervals of $\mathcal{I}_{\text{off}}$ are used to transfer x_0 (since it is possible to stop any communication at any time, using several intervals to transfer x_0 is not a problem), the next ones are used to offload x_1, and so on, until the amount M_{off} is reached, and similarly for the prefetched data, in reverse order. Clearly $\mathcal{S}'$ is eager and ordered.

Since the x_i values become available in the forward phase by order of increasing indices, and are consumed in the backward phase by order of decreasing indices, it is clear that transfers in $\mathcal{S}'$ are valid: an activation is offloaded only after having been produced, and in the backward phase an activation is prefetched before being used. Furthermore, since transfers occur at the same instants and at the same speed as in $\mathcal{S}$, the memory usage of $\mathcal{S}'$ is exactly the same as the memory usage of $\mathcal{S}$ at any instant. The modified $\mathcal{S}'$ schedule is thus valid. $\square$

Greedy Algorithm. According to this result, we consider only eager and ordered schedules. It is thus sufficient to find the amount of offloaded data which results in the smallest makespan. The next result shows that it is best to offload the least possible amount of data. The complete proof of this result can be found in the companion paper [3].

Lemma 2. *Let $\mathcal{S}$ and $\mathcal{S}'$ be no-wait, ordered and eager schedules which offload a quantity of data Q and Q' respectively, with $Q < Q'$. Then the makespan of $\mathcal{S}$ is not larger than the makespan of $\mathcal{S}'$.*

With Lemma 1 and 2, since $M_{peak} - M_{\text{GPU}}$ is a lower bound on the amount of data that any schedule has to offload, we can characterize an optimal schedule for this relaxed problem.

Theorem 2. *For a given instance, the no-wait, eager, ordered schedule which offloads a quantity $M_{peak} - M_{GPU}$ of data is optimal.*

By rounding up the number of offloaded activations, this result provides a heuristic for the original integral problem, that we call Greedy. The Greedy heuristic returns the no-wait, eager, ordered schedule which offloads (entirely) the first k activations, where k is the smallest index such that $\sum_{i \le k} |x_i| \ge M_{peak} - M_{\text{GPU}}$.

However, it may happen that this Greedy schedule offloads too much data because of the rounding procedure. In the next section, we thus analyze a more sophisticated relaxation in order to obtain a more precise algorithm.

5 Fractional Communications

Let now consider another formulation of Problem 1, in which an activation must be either entirely offloaded or not offloaded at all. However, it is still allowed to stop a communication at any time and resume it later. In this section, we first prove that this problem is NP-complete in the weak sense, and then propose a pseudo-polynomial optimal algorithm based on Dynamic Programming.

5.1 Complexity

Problem 2 (Offloading with interruptions). Consider a training phase with L operations, with processing times u_{F_i} and u_{B_i}, data sizes $|x_i|$ and $|y_i|$, temporary memory usage ex_i^F and ex_i^B, where $0 \le i \le L$. Is it possible to perform this computation on a processing device with memory M_{GPU} and bandwidth β between the processing device and the main memory, with an execution time at most T, if communications can be interrupted and partial?

Let us first note that Proposition 1 also holds for this problem (it is always better to schedule with a no-wait policy). We can also state a result similar to the one of the fully fractional case.

Lemma 3. *Any valid solution $\mathcal{S}$ can be transformed into an ordered solution $\mathcal{S}'$ with the same makespan.*

The proof is the same as the one of Lemma 1: transforming $\mathcal{S}$ using the correct order provides a valid schedule. The result is weaker, because an eager schedule which offloads the same data might not be valid for Problem 2 (the last activation might not be fully offloaded).

The next theorem shows that Problem 2 is less difficult than Problem 1. Its proof is omitted here and can be found in the companion report [3].

Theorem 3. *Problem 2 is NP-complete in the weak sense.*

5.2 Structure of Optimal Solutions

According to Lemma 3, our objective is now to find the best ordered schedule. In this section, we derive properties of all ordered and no-wait schedules, which will allow to obtain a dynamic programming algorithm in the next section.

Forward and Backward Phases. Let us consider any ordered, no-wait schedule $\mathcal{S}$. Let M_{F_i} denote the device memory occupied at the end of F_{i-1} (it should contain all data not offloaded at this instant, plus x_i which is the output of F_{i-1}). Let Δ_{F_i} denote the amount of data from $x_0, \ldots, x_{i-1}$ that $\mathcal{S}$ offloads after the end of F_{i-1}. If this amount is zero, let us denote by Av_F the time between the end of the last offload and the end of F_{i-1}, and let $\Delta_{F_i} = -Av_F \cdot \beta$. Moreover, let us set $\Delta^+_{F_i} = \max\{0, \Delta_{F_i}\}$. We aim to characterize the delay ϵ^F_i between the end of F_{i-1} and the start of F_i.

Let us first remark that since $\mathcal{S}$ is a valid schedule, there is enough memory to process F_i at some point, which means that $\Delta^+_{F_i}$ needs to be large enough, $M_{F_i} - \Delta^+_{F_i} + \mathrm{ex}^F_i + |x_{i+1}| \leq M_{\mathrm{GPU}}$

If $M_{F_i} + \mathrm{ex}^F_i + |x_{i+1}| \leq M_{\mathrm{GPU}}$, then F_i can start immediately after the end of F_{i-1}, and since $\mathcal{S}$ is no-wait, then $\epsilon^F_i = 0$. Otherwise, processing F_i can start as soon as enough memory has been released by offloading data at rate β. This yields $\epsilon^F_i = \frac{M_{F_i} + \mathrm{ex}^F_i + |x_{i+1}| - M_{\mathrm{GPU}}}{\beta}$. In summary,

$$\epsilon^F_i = \max\left(0, \frac{M_{F_i} + \mathrm{ex}^F_i + |x_{i+1}| - M_{\mathrm{GPU}}}{\beta}\right) \tag{1}$$

Let us now derive recursive equations to obtain $M_{F_{i+1}}$ and $\Delta_{F_{i+1}}$ from M_{F_i} and Δ_{F_i}. These equations depend on whether x_i is offloaded in $\mathcal{S}$.

If x_i is offloaded, then the amount of data ready to be offloaded at the end of F_{i-1} is $\Delta^+_{F_i} + |x_i|$. Until the end of F_i, the amount of data that can be offloaded is at most $(\epsilon^F_i + u_{F_i})\beta$. Hence we obtain

$$\Delta_{F_{i+1}} = \Delta^+_{F_i} + |x_i| - (\epsilon^F_i + u_{F_i})\beta \tag{2}$$

$$M_{F_{i+1}} = M_{F_i} + |x_{i+1}| - \min\left(\Delta^+_{F_i} + |x_i|, (\epsilon^F_i + u_{F_i})\beta\right). \tag{3}$$

If x_i is not offloaded, we can write similar equations, except that $|x_i|$ is not added to the amount of data to be offloaded. This yields

$$\Delta_{F_{i+1}} = \Delta_{F_i} - (\epsilon^F_i + u_{F_i})\beta \tag{4}$$

$$M_{F_{i+1}} = M_{F_i} + |x_{i+1}| - \min\left(\Delta^+_{F_i}, (\epsilon^F_i + u_{F_i})\beta\right) \tag{5}$$

Let us now derive similar results about the backward phase. We first modify $\mathcal{S}$ to process all backward operations and perform all prefetching operations as *late* as possible without changing the makespan of the schedule. We then define M_{B_i} as the device memory occupied right before processing B_{i-1} (thus it does not take into account the output of B_{i-1}, which is y_{i-1}). Let us also define Δ_{B_i} as the amount of data from $x_L, x_{L-1}, \ldots, x_i$) that $\mathcal{S}$ prefetches before starting B_{i-1}, and if this amount is zero, then $\Delta_{B_i} = -Av_B \cdot \beta$, where Av_B is the time between the start of B_{i-1} and the start of the first prefetch operation. Finally, let ϵ^B_i denote the delay between the end of B_i and the start of B_{i-1}.

With the same reasoning as above, we obtain

$$\epsilon_i^B = \max\left(0, \frac{M_{B_i} + \mathrm{ex}_i^F + |x_{i+1}| + |y_{i+1}| - M_{\mathrm{GPU}}}{\beta}\right) \tag{6}$$

$$\Delta_{B_{i+1}} = |x_i| + \max\left(0, \Delta_{B_i} - (\epsilon_i^B + u_{B_i})\beta\right) \qquad \text{if } x_i \text{ is offloaded} \tag{7}$$

$$\Delta_{B_{i+1}} = \Delta_{B_i} - (\epsilon_i^B + u_{B_i})\beta \qquad \text{otherwise} \tag{8}$$

Computing $M_{B_{i+1}}$ is not necessary, as one can notice that for all i, $M_{B_i} - \Delta^+_{B_i} = |y_i| + |x_i| + \sum_{j<i, j \text{ not offloaded}} |x_j|$, and $M_{F_i} - \Delta^+_{F_i} = |x_i| + \sum_{j<i, j \text{ not offloaded}} |x_j|$. Thus, $M_{B_i} - \Delta^+_{B_i} = |y_i| + M_{F_i} - \Delta^+_{F_i}$, which allows to compute M_{B_i} once all three other values are known.

Idle Time Between Phases. The connection between forward phase and backward phase is defined through Lemma 4 that shows how to compute the idle time between them. The proof of this result is provided in [3].

Lemma 4. *The idle time between phases are given in Eq. (9):*

$$\epsilon_G = \max \begin{cases} 0, \\ \frac{\Delta_{F_{L+1}} + \Delta_{B_{L+1}}}{\beta}, \\ \max_{\{j \le L \mid \sum_{i=j+1}^{L} \beta u_{B_i} < -\Delta_{B_{L+1}}\}} \frac{R_j^B + M_{F_{L+1}} - M_{GPU}}{\beta} - \sum_{i=j+1}^{L} u_{B_i}, \\ \max_{\{j \le L \mid \sum_{i=j+1}^{L} \beta u_{F_i} < -\Delta_{F_{L+1}}\}} \frac{R_j^F + M_{B_{L+1}} - M_{GPU}}{\beta} - \sum_{i=j+1}^{L} u_{F_i}, \end{cases} \tag{9}$$

where $R_j^B = ex_j^B + |y_j| + |y_{j-1}| - \sum_{i>j+1} |x_i|$ *and* $R_j^F = ex_j^F - \sum_{i>j+1} |x_i|$.

5.3 Resulting Algorithm

To formalize the dynamic programming algorithm, let us define $\textsc{Idle}(i, m, d_F, d_B)$ as the smallest possible sum of idle times between (i) the start of the schedule and the end of F_{i-1} and (ii) the start of B_{i-1} and the end of the schedule, for all schedules $\mathcal{S}$ such that $M_{F_i} = m$, $\Delta_{F_i} = d_F$, $\Delta_{B_i} = d_B$.

Any schedule starts with a memory occupation of $|x_0|$, and no idle time, so we can define $\textsc{Idle}(0, |x_0|, 0, 0) = 0$, and $\textsc{Idle}(0, m, d_F, d_B) = \infty$ for all other values of m, d_F, d_B. In order to compute $\textsc{Idle}(i, m, d_F, d_B)$ for all i and all relevant values of m, d_F, d_B, we use hash tables $\textsc{Idle}_i$ indexed with (m, d_F, d_B), with the understanding that if (m, d_F, d_B) is not stored in $\textsc{Idle}_i$, then $\textsc{Idle}(i, m, d_F, d_B) = \infty$. This leads to Algorithm 1, where $\textsc{Idle}_i$ values are used to update $\textsc{Idle}_{i+1}$ values, with two possible cases, either with a schedule that offloads x_i, or with a schedule that does not.

Algorithm 1. Dynamic Programming Algorithm for Fractional Communications

```
$\textsc{Idle}_i \leftarrow \text{HashTable}()$ for $0 \leq i \leq L$
$\textsc{Idle}_0(|x_0|, 0, 0) = 0$
for $i \in \{0, \ldots, L\}$ do
    for $M_{F_i}, \Delta_{F_i}, \Delta_{B_i} \in \textsc{Idle}_i$ do
        $M_{B_i} \leftarrow |y_i| + \Delta^+_{B_i} + M_{F_i} - \Delta^+_{F_i}$
        if $|x_{i+1}| + \max(M_{F_i} + \text{ex}^F_i - \Delta_{F_i}, M_{B_i} + \text{ex}^B_i + |y_{i+1}| - \Delta_{B_i}) \leq M_{\text{GPU}}$ then
            Compute $\epsilon^F_i, \epsilon^B_i$ from equations (1) and (6)
            Compute $M_F, \Delta_F, \Delta_B$ if $x_i$ is offloaded (equations (2), (3) and (7))
            $\textsc{Idle}_{i+1}(M_F, \Delta_F, \Delta_B) \leftarrow \min\left(\textsc{Idle}_{i+1}(M_F, \Delta_F, \Delta_B), \textsc{Idle}_i(M_{F_i}, \Delta_{F_i}, \Delta_{B_i}) + \epsilon^F_i + \epsilon^B_i\right)$
            Compute $M'_F, \Delta'_F, \Delta'_B$ if $x_i$ is not offloaded (equations (4), (5) and (8))
            $\textsc{Idle}_{i+1}(M'_F, \Delta'_F, \Delta'_B) \leftarrow \min\left(\textsc{Idle}_{i+1}(M'_F, \Delta'_F, \Delta'_B), \textsc{Idle}_i(M_{F_i}, \Delta_{F_i}, \Delta_{B_i}) + \epsilon^F_i + \epsilon^B_i\right)$
for $M_F, \Delta_F, \Delta_B \in \textsc{Idle}_{L+1}$ do
    Compute $\epsilon_G$ according to equation (9)
    $\textsc{TotalIdle}(M_F, \Delta_F, \Delta_B) \leftarrow \textsc{Idle}_{L+1}(M_F, \Delta_F, \Delta_B) + \epsilon_G$
Get $M^*_F, \Delta^*_F, \Delta^*_B$ which minimizes $\textsc{TotalIdle}(M_F, \Delta_F, \Delta_B)$
Backtrack in $\textsc{Idle}_{L+1}, \ldots, \textsc{Idle}_0$ to obtain optimal offload decisions
```

Once $\textsc{Idle}_{L+1}$ is computed, TotalIdle can be found by adding the corresponding idle time ϵ_G between the forward and backward phases. Then, the smallest value in TotalIdle is the smallest possible idle time for any ordered, no-wait schedule. Finally, we can identify which offload decisions have led to this idle time, and then obtain the description of the corresponding schedule.

The number of values kept in the hash table can be bounded in the following way: M_F is between 0 and M_{GPU}, Δ_F and Δ_B are between $-\sum_i (u_{F_i})\beta$ and M_{GPU}. The number of possible values is thus $O(M_{\text{GPU}}(M_{\text{GPU}} + u_{F_i}\beta)^2)$, and the complexity of Algorithm 1 is $O(LM_{\text{GPU}}(M_{\text{GPU}} + u_{F_i}\beta)^2)$, which is indeed pseudo-polynomial.

This optimal algorithm for the fractional communications model can be turned into heuristic DynProg for the original problem. DynProg computes the optimal set of activations for the relaxed model with Algorithm 1, and outputs the no-wait, ordered schedule which offloads exactly these activations.

Practical Considerations. The integration of offloading in Deep Learning frameworks is generally not completely trivial. A first solution is the one adopted by Vdnn[1] and consists in implementing an ad-hoc system to do the training by directly managing computation operations and data transfer operations between the main memory and the device memory. It is possible to use a solution of this type, by directly integrating our algorithms in addition to the heuristics proposed in [1]. This solution allows great flexibility and low-level management of all data movements and allocations, but it limits the possible adoption by not relying on classical Deep Learning frameworks. TFLMS [13] is directly built on top of TensorFlow. The principle consists in modifying the task graph by explicitly integrating swap tasks (between the device memory and the main memory). This approach is very interesting because it has a high level of integration with

[1] https://github.com/shriramsb/vdnn-plus-plus/.

TensorFlow, but on the other hand, it is only possible to specify the relative order of the transfer tasks with respect to the computation tasks and not to perform them at specific dates. As the scheduling of tasks is controlled by TensorFlow itself, it is therefore not possible a priori to make just-in-time communications and allow for a perfect overlap of computations and communications. The situation with PyTorch is also complex, because implementing an offloading solution requires to transfer not only easily accessible tensors, but also the complete data structure that are necessary for executing backward operations. Manipulating this requires operating on PyTorch internals and is out of the scope of this paper. Therefore, we rely on simulations to compare algorithms and scheduling, and we postpone their implementation in Deep Learning frameworks until there is an easier and more explicit support of data exchange.

As mentioned above, the dynamic program algorithm has a pseudo-polynomial complexity, and its running time can get large for deep networks. To keep the running time reasonable, we implement a rounding procedure (the details are given in the research report [3]). This allows to keep all running times below 25 s. Since this computation is performed only once for the whole training phase, such an execution time is completely acceptable.

6 Experimental Analysis

Experimental Setting. This section presents experimental results obtained on three different kinds of networks: ResNet, DenseNet, and Inception v3. We have slightly modified these networks to represent them as linear chains, by grouping each non-linear part of the graph in a virtual layer. We have obtained the values of u_F, u_B, ex^F, ex^B, and the sizes of x_i and y_i by performing measures on sample data on a node equipped with a Nvidia Tesla V100-PCIE GPU card with 15.75 GB of memory. We also measured the bandwidth β to transfer data using PyTorch from the GPU to the RAM, and obtained around 12.5 GB/s.

We use all available depths for ResNet (18, 34, 50, 101, 152) and DenseNet (121, 161, 169 and 201). We use three different image sizes: small images of shape 224×224, medium images of shape 500×500, and large images of shape 1000×1000. During the training phase, for higher efficiency, it is classical to process images in *batches*, where several images are processed independently. For each model and image size, we consider different batch sizes that are powers of 2, starting from the smallest batch size that ensures a reasonable throughput. For each case, we compute schedules with five different algorithms: Greedy (Sect. 4), DynProg (based on Algorithm 1, see Sect. 5.3), AutoSwap, TFLMS and Vdnn, where the last three approaches are based on the state-of-the-art methods used in the previous works. AutoSwap [21] is a score-based heuristic which uses a weighted average of 4 priority scores to decide which activations should be offloaded in priority. The best weight combination is obtained with Bayesian Optimization. TFLMS [13] is a heuristic designed for general graphs (not necessarily sequential) in high bandwidth settings, but it does not use any profiling information and thus cannot adapt to the available memory. TFLMS

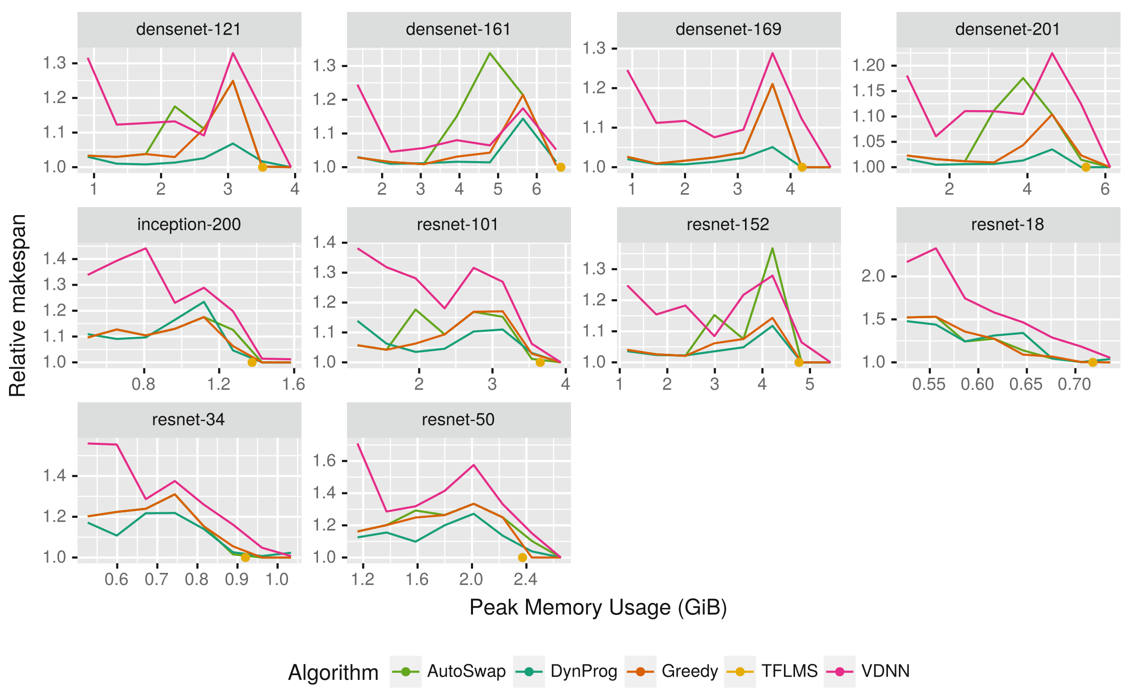


Fig. 2. Experimental results for image size 224 and batch size 32.

is parameterized with the number of tensors to be offloaded and how many layers in advance the data should be prefetched, and we present the performance achieved by the best configurations. In VDNN++ [1], the authors identify convolutional layers as having a much longer computation latency. Their approach is to offload the input of either all convolutional layers, or of half of them. In our implementation of VDNN, we identify as candidates the layers for which the ratio $\frac{u_{F_i}}{|x_i|}$ is above a given threshold. For all possible thresholds, we compute the no-wait, ordered schedule which offloads all these candidates, and the one which offloads half of them. VDNN outputs the best schedule out of all these choices.

Representative Results. A representative selection of achieved results is depicted in Fig. 2, where different types of network of different length are considered with a given image and batch size. For each network, we run all algorithms with a memory limit varying from the minimum amount of memory required to run the network, to M_{peak} which allows to process the network with no offloading. In each case, we also compute the lower bound LB (Proposition 3), and the plots show the ratio of the makespan achieved by each algorithm to the lower bound, thus points where the ratio is 1 correspond to optimal solutions. We observe that both GREEDY and DYNPROG outperform the VDNN heuristic in all cases, especially in low memory scenarios. Once correctly parameterized, TFLMS is able to obtain optimal makespan for the highest memory limit values. But it is unable to delay forward computations until enough memory is available, and thus can not adapt to low memory settings when bandwidth is scarce. AUTOSWAP often produces the same solution as the GREEDY algorithm (for a much higher computational cost), but its performance depends on the random procedure of

Bayesian Optimization and is thus very inconsistent. The DynProg approach obtains significantly better performance than Greedy. The difference is small in many cases, except for the DenseNet networks where DynProg is able to consistently obtain almost optimal solutions. The spike that can be observed on these graphs for Greedy and Vdnn correspond to the memory limit M_{GPU} for which both terms of the lower bound LB are almost equal (*i.e.*, the total execution time is very close to the time to transfer $M_{peak} - M_{\mathrm{GPU}}$). Such cases are more difficult to solve because both criteria need to be optimized carefully.

Overall, DynProg obtains much more stable performance than Vdnn and AutoSwap, and produces solutions over a much wider range than TFLMS. Furthermore, DynProg is able to consistently achieve a ratio below 1.2, which means that its throughput is at least 83% of the highest possible throughput.

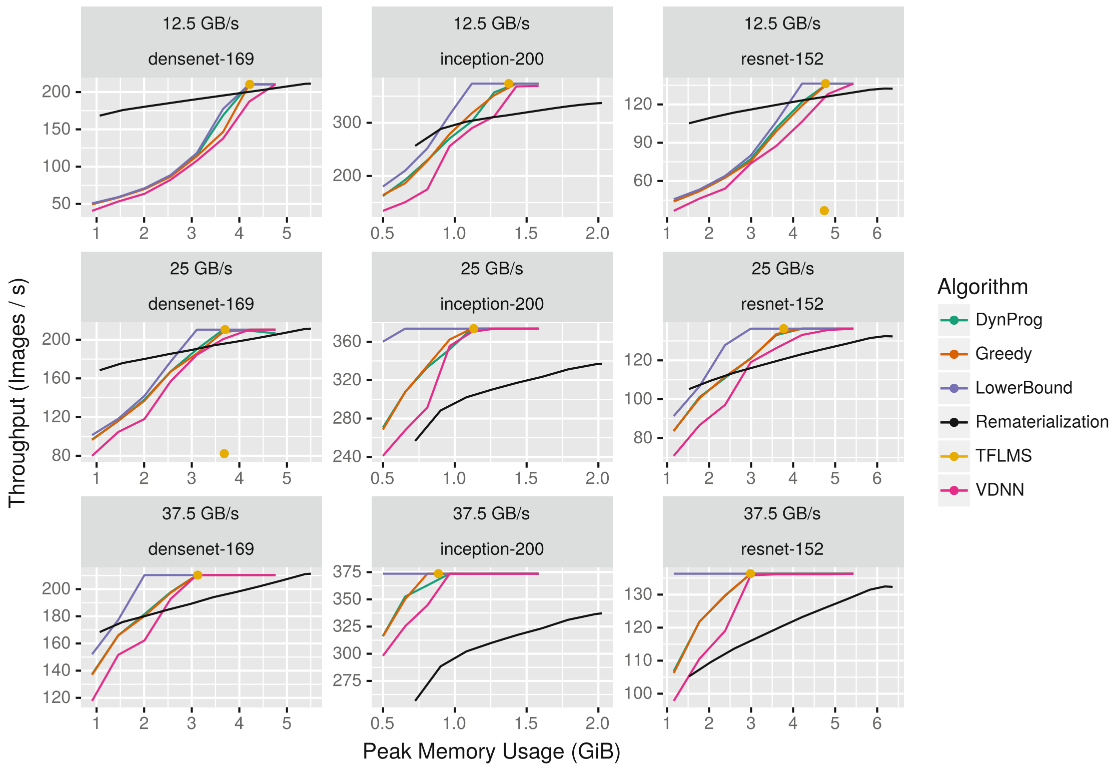


Fig. 3. Comparison to rematerialization for image size 224 and batch size 32, for various bandwidth values.

Comparison to Rematerialization. An alternative to offloading is rematerialization [7], in which memory savings are achieved by discarding activations and recomputing them later. In Fig. 3, we compare the throughput (in terms of processed images per second) obtained by the offloading algorithms and by an optimal rematerialization strategy [2]. We observe that for the bandwidth measured on our hardware, the rematerialization is significantly more interesting, except for the higher memory limits. However, if the bandwidth is two or three times larger, the interest of offloading becomes significant, allowing to perform at optimal throughput over a wide range of memory limits.

More results are available in the companion paper [3].

7 Conclusions

In this paper, we address the problem of memory usage during the training phase of Deep Neural Networks. Previous works [1, 13, 14, 16, 20, 21] advocated to offload some of the data onto the main memory, and to prefetch them back when needed. We propose a formal algorithmic model of the corresponding scheduling problem, where the goal is to identify which activations should be offloaded so as to minimize the total execution time. We prove that this problem is NP-Complete in the strong sense, and we propose two heuristics based on relaxations of the problem. The Greedy heuristic always offloads the first activations in the network. This very simple technique nevertheless achieves good results in our experimental evaluation. The DynProg algorithm is a more sophisticated approach which takes into account the fact that activations cannot be partially transferred which allowed to obtain mostly better solutions. In any case, both algorithms provide significant improvements over the previous approaches.

A promising research direction is the validation through real experiments, that would allow to confirm the relevance of the assumptions made in the model. Since our theoretical analysis shows that being able to offload activations partially makes the problem much easier, it could be very interesting to assess in which cases this could be technically feasible. Finally, this offloading technique is complementary of the checkpointing approach: some activations can be transferred to the main memory while others can be recomputed. Solving the mixed checkpointing and offloading corresponding algorithmic problem might be challenging, but would certainly yield a significant improvement for training large and deep models.

References

1. Shriram, S.B., 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). https://doi.org/10.1109/IPDPS.2019.00030
2. Beaumont, O., Eyraud-Dubois, L., Herrmann, J., Joly, A., Shilova, A.: Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. Research Report RR-9302, Inria Bordeaux Sud-Ouest, November 2019. https://hal.inria.fr/hal-02352969
3. Beaumont, O., Eyraud-Dubois, L., Shilova, A.: Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training, October 2019. https://hal.inria.fr/hal-02316266, working paper or preprint
4. Carranza-Rojas, J., Goeau, H., Bonnet, P., Mata-Montero, E., Joly, A.: Going deeper in the automated identification of herbarium specimens. BMC Evol. Biol. **17**(1), 181 (2017). https://doi.org/10.1186/s12862-017-1014-z
5. Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016)
6. Feng, Y., Zhang, Z., Zhao, X., Ji, R., Gao, Y.: GVCNN: group-view convolutional neural networks for 3D shape recognition. In: IEEE CVPR, pp. 264–272. IEEE (2018). https://doi.org/10.1109/CVPR.2018.00035

7. 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). https://doi.org/10.5555/3157382.3157559
8. Hara, K., Kataoka, H., Satoh, Y.: Can spatiotemporal 3D CNNs retrace the history of 2D CNNs and imagenet? In: IEEE CVPR, pp. 6546–6555. IEEE (2018). https://doi.org/10.1109/CVPR.2018.00685
9. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE CVPR. IEEE (2016). https://doi.org/10.1109/cvpr.2016.90
10. Huang, G., Liu, Z., van der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: IEEE CVPR. IEEE (2017). https://doi.org/10.1109/CVPR.2017.243
11. Kumar, R., Purohit, M., Svitkina, Z., Vee, E., Wang, J.: Efficient rematerialization for deep networks. In: Advances in Neural Information Processing Systems, pp. 15146–15155 (2019)
12. Kwon, Y., Rhu, M.: Beyond the memory wall: a case for memory-centric HPC system for deep learning. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 148–161. IEEE (2018). https://doi.org/10.1109/MICRO.2018.00021
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. Meng, C., Sun, M., Yang, J., Qiu, M., Gu, Y.: Training deeper models by GPU memory optimization on tensorflow. In: Proceedings of ML Systems Workshop in NIPS (2017)
15. Pleiss, G., Chen, D., Huang, G., Li, T., van der Maaten, L., Weinberger, K.Q.: Memory-efficient implementation of densenets. arXiv preprint arXiv:1707.06990 (2017)
16. 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). https://doi.org/10.1109/MICRO.2016.7783721
17. Rhu, M., O'Connor, M., Chatterjee, N., Pool, J., Kwon, Y., Keckler, S.W.: Compressing DMA engine: leveraging activation sparsity for training deep neural networks. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 78–91. IEEE (2018). https://doi.org/10.1109/HPCA.2018.00017
18. Rota Bulò, S., Porzi, L., Kontschieder, P.: In-place activated batchnorm for memory-optimized training of DNNs. In: IEEE CVPR, pp. 5639–5647. IEEE (2018). https://doi.org/10.1109/cvpr.2018.00591
19. Shou, Z., Chan, J., Zareian, A., Miyazawa, K., Chang, S.F.: CDC: convolutional-de-convolutional networks for precise temporal action localization in untrimmed videos. In: IEEE CVPR, pp. 5734–5743. IEEE (2017). https://doi.org/10.1109/CVPR.2017.155
20. Wang, L., et al.: SuperNeurons: dynamic GPU memory management for training deep neural networks. SIGPLAN Not. **53**(1), 41–53 (2018). https://doi.org/10.1145/3200691.3178491
21. 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)