

# Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training

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

October 21, 2019

## 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 memory of the GPU to the memory of the CPU. This requires to determine which activations should be offloaded and when these transfers from and to the memory of the GPU should take place. We prove that this problem is NP-hard in the strong sense, and we propose two heuristics based on relaxations of the problem. We perform extensive experimental evaluation on standard Deep Neural Networks. We compare the performance of our heuristics against previous approaches from the literature, showing that they achieve much better performance in a wide variety of situations.

## 1 Introduction

Training for Deep Learning Networks (DNNs) has become a major compute intensive application [1, 2, 3], typically performed on GPU clusters [4, 5]. The training phase typically 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. Even worse, 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 needed to transfer 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 [6, 7].

Many approaches have been proposed in the literature in order to circumvent this memory issue. In this paper, we focus on an *offloading* approach, which consists in reducing memory usage on the GPU by transferring some activations to the memory of the CPU, 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 memory of CPU memory to the memory of the GPU. This approach has been recently considered in [8, 9], where the authors advocate the general idea and propose several static and dynamic heuristics to decide which activations should be offloaded. In this paper, our goal is to improve the algorithmic solutions and to study the complexity of the offloading problem. More specifically, we prove that the fully integral problem is strongly NP-complete, 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 integral case.

The rest of the paper is organized as follows. In Section 2, we discuss the related works dealing with GPU memory offloading and with memory savings during the training phase in general. In Section 3, we present the model and notations used throughout the paper, and assess the complexity of the problem. In Section 4, we propose a first relaxation where activations can be partially or completely offloaded into the memory of the CPU, and derive an optimal strategy. In Section 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 Section 6, we provide experimental results and we assess the efficiency of our heuristics against the previous approach [9], before presenting conclusions and perspectives in Section 7.

## 2 Related Work

Several approaches have been considered to limit memory usage when training Deep Neural Networks. Data parallelism [10, 11, 4] is the most used technique to perform training in parallel. When using data parallelism, the model and the weights are replicated onto all participating nodes, but different mini-batches are trained independently in parallel on the different nodes and then gradients are exchanged between nodes. This allows to efficiently train large datasets, but it still requires all network parameters and all activations to be stored on each node during the training phase. Model parallelism [12] is another approach where the network model is split into several non-overlapping parts that are distributed over the different resources. This requires to identify a well-balanced partition of the computational network, while minimizing the communications of forward and backward activations between nodes that are induced by the partitioning.

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

### 2.1 Checkpointing techniques

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. This problem has been first widely studied in the context of Automatic Differentiation (AD) [13]. In the context of AD, computational networks can be seen as long chains, in which the forward activation corresponding to the  $i$ -th stage of the chain has to be kept into memory until the  $i$ -th backward stage. Checkpointing algorithms consist in determining in advance which forward checkpoints should be kept into memory and which ones should be recomputed from stored checkpoints when performing the backward phase.

Adjoints computation is at the core of many scientific applications, from climate and ocean modeling [14] to oil refinery [15]. Many studies have been performed to determine optimal checkpointing strategies for Automatic Differentiation in different contexts. In the case of homogeneous chains, closed form formulas providing the exact position of checkpoints have been proposed [16], and in most cases the general algorithmic ingredient is to derive optimal checkpointing strategies using Dynamic Programming [16].

The use of checkpointing strategies has recently been advocated for DNN in several papers [17, 18, 19]. An implementation of a simple periodic strategy is provided in PyTorch [20, 21], but it is only optimal for the restricted case of homogeneous chains, whereas DNN models are in general more complicated.

### 2.2 Memory Offloading

Offloading is a potentially complementary approach which consists in offloading some of the forward activations from the memory of the GPU to the memory of the CPU, which is expected to be much larger [8, 9]. In [8], the authors propose a simple and effective mechanism, 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 [9] by the design of techniques to deal with memory fragmentation. Nevertheless, in both papers, the algorithmic strategies to decide which activations to offload into the memory of the CPU are relatively straightforward. Proposed strategies consist in trying to offload either all activations

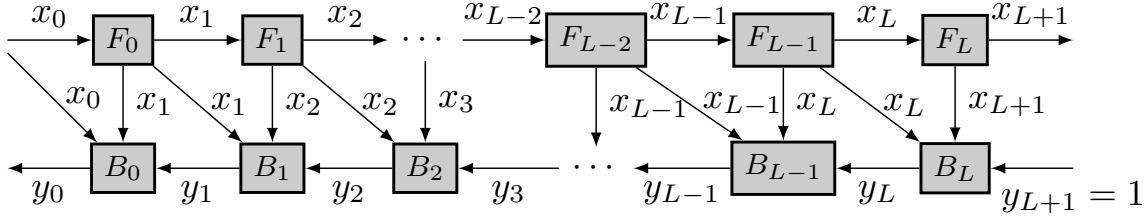


Figure 1: Data dependencies induced the training phase of Sequential Deep Neural Networks.

or only those that correspond to convolutional layers. 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.

### 3 Model and Complexity

#### 3.1 Computation Model

We consider the training phase of DNNs, as depicted on Figure 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 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  $M_{\text{GPU}}$ . 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. In order to decrease memory usage, we assume that it is possible to *offload* some of the 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  $B$ . The offloaded data can then be *prefetched* during the backward phase, so that it is available when needed to perform the corresponding backward operation.

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  $\text{ex}_i^F$  for  $F_i$  and  $\text{ex}_i^B$  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.

When transferring a value  $x_i$  (it makes no sense to transfer  $y_i$  since it is used only once immediately after being produced), we make the following assumptions:

- (A) Only one transfer can occur at a time;
- (B) Transferring  $x_i$  takes time  $B|x_i|$ ;
- (C) Transfers and computations can be completely overlapped, and do not interfere;
- (D)  $x_i$  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  $\text{ex}_i^F$  and  $\text{ex}_i^B$ , where  $0 \leq i \leq L$ . Is it possible to perform this computation on a processing device with memory  $M_{\text{GPU}}$  and bandwidth  $B$ , 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  $M_{\text{peak}}$  the maximum of these values.

$$\begin{aligned} m_{F_i} &= \text{ex}_i^F + \sum_{j \leq i+1} |x_j| \\ m_{B_i} &= \text{ex}_i^B + |y_i| + |y_{i+1}| + \sum_{j \leq i+1} |x_j| \\ M_{\text{peak}} &= \max \left( \max_{0 \leq i \leq L} m_{B_i}, \max_{0 \leq i \leq L} m_{F_i} \right) \end{aligned}$$

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

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

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  $LB = \max(\sum_i u_{F_i} + u_{B_i}, 2 \frac{M_{\text{peak}} - M_{\text{GPU}}}{B})$  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 by a reduction from the 3-partition problem: given a set of integers  $\{u_0, u_2, \dots, u_{3m-1}\}$  such that  $\sum_i u_i = mV$ , is it possible to partition it into  $m$  parts  $\{S_1, \dots, S_m\}$  so that for any  $j \leq m$ ,  $|S_j| = 3$  and  $\sum_{i \in S_j} u_i = V$ . This problem is known to be NP-hard in the strong sense.

Given an instance of 3-partition, we consider the following instance of Problem 1, depicted on Figure 2:

- $L = 5m$ ,  $B = V$ ,  $M_{\text{GPU}} = mV$ ,  $T = 2m$ ;
- $u_{F_i} = 0$  and  $|x_i| = u_i$  for  $1 \leq i < 3m$ ;
- $u_{F_i} = 1$  and  $|x_i| = 0$  for  $i = 3m + 2k$ ,  $k \in [0, n]$ ;
- $u_{F_i} = 0$  and  $|x_i| = V$  for  $i = 3m + 2k + 1$ ,  $k \in [0, n]$ ;
- $u_{B_i} = 0$  and  $|y_i| = 0$  for all  $i$ , except  $u_{B_{3m}} = m$ .

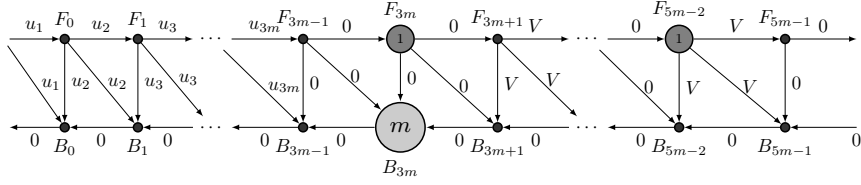


Figure 2: The instance of Problem 1 used in the reduction from 3-partition. Activation sizes are indicated on the edges and non-zero execution times of operations are written inside the corresponding nodes.

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  $\sum_{i \in S_j} u_i = V$ . We can build a schedule which starts  $F_{3m+2k}$  at time  $k$  for  $0 \leq k < m$ , 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  $B = V$ , 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  $B = V$  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.  $\square$

Theorem 1 shows that even when we know which activations should be offloaded, it is difficult to decide the order in which the transfers should be done. 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. 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.

### 4.1 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  $x_0, x_1, \dots, x_k$  (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  $M_{\text{off}}$  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  $M_{\text{off}}$  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.  $\square$

## 4.2 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.

**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  $\mathcal{S}'$ .*

*Proof.* Let us consider the schedule  $\mathcal{S}''$  obtained from  $\mathcal{S}'$  by removing all transfers (offload and prefetch) corresponding to data between  $Q$  and  $Q'$  in the eager order. Since  $\mathcal{S}'$  is valid, all data dependencies are satisfied in  $\mathcal{S}''$ . Let us now prove that  $\mathcal{S}''$  also fulfills the memory constraint.

Consider any time instant  $t$  in schedule  $\mathcal{S}'$ , and let us denote by  $m'_{\text{CPU}}(t)$  and  $m'_{\text{GPU}}(t)$  the amount of data stored on the CPU and GPU by  $\mathcal{S}'$ . If  $m'_{\text{CPU}}(t) \leq Q$ , then the data stored on GPU in  $\mathcal{S}'$  and  $\mathcal{S}''$  are the same, so  $\mathcal{S}''$  is valid at instant  $t$ . If  $m'_{\text{CPU}}(t) > Q$ , then the amount of data stored on the GPU in schedule  $\mathcal{S}''$  is  $m''_{\text{GPU}}(t) = m'_{\text{GPU}}(t) + m'_{\text{CPU}}(t) - Q$ .

Since  $\mathcal{S}$  is valid,  $Q \geq M_{\text{peak}} - M_{\text{GPU}}$ . Furthermore, by definition of  $M_{\text{peak}}$ ,  $m'_{\text{GPU}}(t) + m'_{\text{CPU}}(t) \leq M_{\text{peak}}$ . Thus,

$$\begin{aligned} m''_{\text{GPU}}(t) &\leq m'_{\text{GPU}}(t) + m'_{\text{CPU}}(t) + M_{\text{GPU}} - M_{\text{peak}} \\ &\leq M_{\text{GPU}} \end{aligned}$$

The schedule  $\mathcal{S}''$  is thus a valid, eager and ordered schedule which offloads a quantity of data  $Q$ . The schedule  $\mathcal{S}$  offloads the same data in the same order; since  $\mathcal{S}$  is no-wait, by Proposition 1 the makespan of  $\mathcal{S}$  is not larger than the makespan of  $\mathcal{S}''$ , which is equal to the makespan of  $\mathcal{S}'$ .  $\square$

With these two lemmas, since  $M_{\text{peak}} - M_{\text{GPU}}$  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  $M_{\text{peak}} - M_{\text{GPU}}$  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  $\sum_{i \leq k} |x_i| \geq M_{\text{peak}} - M_{\text{GPU}}$ .

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.

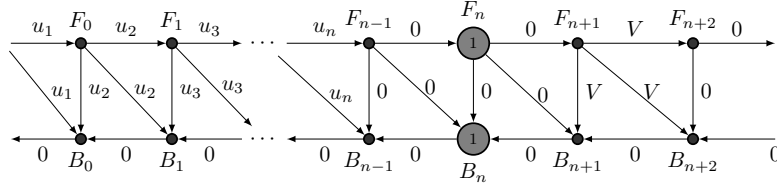


Figure 3: The instance of Problem 2 which corresponds to the 2-partition problem with values  $u_i$ .

## 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  $\text{ex}_i^F$  and  $\text{ex}_i^B$ , where  $0 \leq i \leq L$ . Is it possible to perform this computation on a processing device with memory  $M_{\text{GPU}}$  and bandwidth  $B$ , with an execution time at most  $T$ , if communications can be interrupted?

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).

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

*Proof.* Problem 2 clearly belongs to NP, in the same way as Problem 1.

Let us prove that it is NP-hard by reduction for the 2-Partition problem, which can be stated as: given  $n$  positive integers  $u_1, u_2, \dots, u_n$  such that  $\sum_i u_i = 2V$ , is it possible to partition them in two subsets  $S_1$  and  $S_2$  such that  $\sum_{i \in S_1} u_i = \sum_{i \in S_2} u_i = V$ ?

Given an instance of 2-Partition, let us consider an instance  $\mathcal{I}$  of Problem 2, depicted on Figure 3, with  $L = n + 3$ ,  $B = V$  and  $M_{\text{GPU}} = 2V$ . All values  $y_i$  have size 0, and all operations  $F_i$  and  $B_i$  take time 0, except for  $F_n$  and  $B_n$  whose duration is 1. The input size of the first  $n$  operations are given by values  $u_i$ , and the input size of  $F_{n+2}$  is  $|x_{n+2}| = V$ . Note that  $\mathcal{I}$  can be computed in polynomial time. We claim that  $\mathcal{I}$  admits a valid schedule of length  $T = 2$  if and only if the instance of 2-Partition has a solution.

If the 2-partition problem has a solution, then there exist subsets  $S_1$  and  $S_2 = S \setminus S_1$  such that  $\sum_{i \in S_1} u_i = V$ . It is thus possible to offload all the corresponding activations  $x_i$  during operation  $F_n$  (since  $B = V$ ), and then to prefetch them during operation  $B_n$ . This allows to perform  $F_{n+1}, F_{n+2}, B_{n+2}, B_{n+1}$  immediately after  $F_n$ , since only a quantity  $V$  of data from the first activations is stored. Once  $B_n$  has been performed, all other  $B_i$  can be performed as well, which results in a schedule of length 2.

On the other hand, let us assume that there exists a schedule of length 2. Let us denote as  $S_1$  the set of indices corresponding to activations that are offloaded in that schedule (remember that each activation is either offloaded completely or not at all), and let  $Q = \sum_{i \in S_1} u_i$ . Since  $M_{\text{peak}} = 3V$ , it is clear that  $Q \geq V$ . Since  $B = V$ , the makespan of the schedule is at least  $\frac{2Q}{V}$  (this is the time it takes to offload and prefetch  $S_1$ ), thus  $Q \leq V$ . This implies that  $S_1$  is a solution of the 2-Partition instance, which completes the proof.  $\square$

## 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.

### 5.2.1 Forward and Backward phases

Let us consider any ordered, no-wait schedule  $\mathcal{S}$ . Let  $M_{F_i}$  denote the GPU 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  $\Delta_{F_i}$  denote the amount of data from  $x_0, \dots, x_{i-1}$  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  $\Delta_{F_i} = -Av_B \cdot B$ . Moreover, let us set  $\Delta_{F_i}^+ = \max\{0, \Delta_{F_i}\}$ . We aim to characterize the delay  $\epsilon_i^F$  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,  $M_{F_i} - \Delta_{F_i} + \text{ex}_i^F + |x_{i+1}| \leq M_{\text{GPU}}$

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

$$\epsilon_i^F = \max\left(0, \frac{M_{F_i} + \text{ex}_i^F + |x_{i+1}| - M_{\text{GPU}}}{B}\right) \quad (1)$$

Let us now derive recursive equations to obtain  $M_{F_{i+1}}$  and  $\Delta_{F_{i+1}}$  from  $M_{F_i}$  and  $\Delta_{F_i}$ . 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  $(\epsilon_i^F + u_{F_i})B$ . Hence we obtain

$$\Delta_{F_{i+1}} = \Delta_{F_i}^+ + |x_i| - (\epsilon_i^F + u_{F_i})B \quad (2)$$

$$M_{F_{i+1}} = M_{F_i} + |x_{i+1}| - \min(\Delta_{F_i}^+ + |x_i|, (\epsilon_i^F + u_{F_i})B). \quad (3)$$

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

$$\Delta_{F_{i+1}} = \Delta_{F_i} - (\epsilon_i^F + u_{F_i})B \quad (4)$$

$$M_{F_{i+1}} = M_{F_i} + |x_{i+1}| - \min(\Delta_{F_i}^+, (\epsilon_i^F + u_{F_i})B) \quad (5)$$

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 GPU 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  $\Delta_{B_i}$  as the amount of data from  $x_L, x_{L-1}, \dots, x_i$  that  $\mathcal{S}$  prefetches before starting  $B_{i-1}$ , and if this amount is zero, then  $\Delta_{B_i} = -Av_B \cdot B$ , where  $Av_B$  is the time between the start of  $B_{i-1}$  and the start of the first prefetch operation. Finally, let  $\epsilon_i^B$  denote the delay between the end of  $B_i$  and the start of  $B_{i-1}$ .

With the same reasoning as above, we obtain

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

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

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



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.

### 5.2.2 Idle time between phases

With these derivations, we have accounted for all idle times in schedule  $\mathcal{S}$ , except for a possible idle time  $\epsilon_G$  between the end of  $F_L$  and the start of  $B_L$ . Several situations are possible. If  $\mathcal{S}$  performs no offloading after  $F_L$ , and no prefetching before  $B_L$  (*i.e.*  $\Delta_{F_{L+1}}$  and  $\Delta_{B_{L+1}} \leq 0$ ), then no idle time occurs and  $\epsilon_G = 0$ . If  $\mathcal{S}$  performs both kind of transfers (*i.e.* both  $\Delta_{F_{L+1}}$  and  $\Delta_{B_{L+1}}$  are positive), then the idle time corresponds to transferring the total data,  $\epsilon_G = \frac{\Delta_{F_{L+1}} + \Delta_{B_{L+1}}}{B}$ .

Otherwise, if for example  $\Delta_{F_{L+1}} > 0$  and  $\Delta_{B_{L+1}} \leq 0$ , the schedule  $\mathcal{S}$  can perform offloading during the first backward operations. This is possible only if enough memory is available to perform the operations, so it may result in some idle time if it is necessary to wait until enough data has been offloaded. Let us denote by  $Av_B = -\frac{\Delta_{B_{L+1}}}{B}$  the time between the start of  $B_L$  and the start of the first prefetch operation, and let us set  $U_j^B = \sum_{i=j+1}^L u_{B_i}$ . According to the above derivations, since no prefetching occurs, all the operations performed during this time have no idle time between them. These operations are thus all the  $B_j$  such that  $U_j^B < Av_B$ .

We can compute the available memory at any time  $t$  after the end of  $F_L$ , where we pretend that no  $x_i$  is removed from memory until  $Av_B$ :  $M(t) = (M_{\text{GPU}} - M_{F_{L+1}}) + tB$ . At the start of  $B_j$ , the memory available is  $M(\epsilon_G + \sum_{i=j+1}^L u_{B_i})$ . Starting  $B_j$  requires to have at least  $R_j^B = \text{ex}_j^B + |y_j| + |y_{j-1}| - \sum_{i>j+1} |x_i|$  available memory, where we account for the removal of  $x_i$ s. This provides an upper bound on  $\epsilon_G$ :  $\epsilon_G \geq \frac{R_j^B + M_{F_{L+1}} - M_{\text{GPU}}}{B} - \sum_{i=j+1}^L u_{B_i}$

This is true for all the  $B_j$  operations that occur during  $Av_B$ . Another upper bound on  $\epsilon_G$  is given by the fact that the offloading must finish before the prefetching starts,  $\epsilon_G \geq \frac{\Delta_{F_{L+1}}}{B} - Av_B$ . These are the only constraints on  $\epsilon_G$ , hence we have

$$\epsilon_G = \max \left( 0, \frac{\Delta_{F_{L+1}}}{B} - Av_B, \max_{\substack{j \leq L \\ U_j^B < Av_B}} \frac{R_j^B + M_{F_{L+1}} - M_{\text{GPU}}}{B} - U_j^B \right). \quad (9)$$

On the other hand, if  $\Delta_{B_{L+1}} > 0$  and  $\Delta_{F_{L+1}} \leq 0$ , schedule  $\mathcal{S}$  can perform prefetching during the last forward operations. Let us denote by  $Av_F = -\frac{\Delta_{F_{L+1}}}{B}$  the time between the end of the last offload and the end of  $F_L$ ,  $U_j^F = \sum_{i=j+1}^L u_{F_i}$ , and  $R_j^F = \text{ex}_j^F - \sum_{i>j+1} |x_i|$ .

With considerations similar as above, we obtain

$$\epsilon_G = \max \left( 0, \frac{\Delta_{B_{L+1}}}{B} - Av_F, \max_{\substack{j \leq L \\ U_j^F < Av_F}} \frac{R_j^F + M_{B_{L+1}} - M_{\text{GPU}}}{B} - U_j^F \right) \quad (10)$$

The results obtained in this section show that in order to compute how future offloading decisions affect the idle time of a schedule, one only needs to know the values of  $M_{F_i}, \Delta_{F_i}, M_{B_i}, \Delta_{B_i}$ . The exact decisions of which data from  $x_0, \dots, x_{i-1}$  has actually been offloaded is not required. This allows to design a Dynamic Programming algorithm to identify the offloading decisions that induce the smallest idle time.

## 5.3 Dynamic Programming Algorithm

To formalize the dynamic programming algorithm, let us define  $\text{IDLE}(i, m, d_F, d_B)$  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  $M_{F_i} = m, \Delta_{F_i} = d_F, \Delta_{B_i} = d_B$ .

---

**Algorithm 1** Dynamic Programming Algorithm for Fractional Communications

---

```

IDLEi ← HashTable() for 0 ≤ i ≤ L
IDLE0(|x0|, 0, 0) = 0
for i = 0, . . . , L do
  for MFi, ΔFi, ΔBi ∈ IDLEi do
    MBi ← |yi| + ΔBi+ + MFi - ΔFi+
    if MFi + exiF - ΔFi + |xi+1| ≤ MGPU and MBi + exiB + |xi+1| + |yi+1| - ΔBi ≤ MGPU then
      Compute εiF, εiB from equations (1) and (6)
      Compute MF, ΔF, ΔB if xi is offloaded (equations (3), (2) and (7))
      IDLEi+1(MF, ΔF, ΔB) ← min (IDLEi+1(MF, ΔF, ΔB), IDLEi(MFi, ΔFi, ΔBi) + εiF + εiB)
      Compute M'F, Δ'F, Δ'B if xi is not offloaded (equations (5), (4) and (8))
      IDLEi+1(M'F, Δ'F, Δ'B) ← min (IDLEi+1(M'F, Δ'F, Δ'B), IDLEi(MFi, ΔFi, ΔBi) + εiF + εiB)
  for MF, ΔF, ΔB ∈ IDLEL+1 do
    Compute εG according to equations (10) and (9)
    IDLEL+1(MF, ΔF, ΔB) ← IDLEL+1(MF, ΔF, ΔB) + εG
  Get MF*, ΔF*, ΔB* which minimizes IDLEL+1(MF, ΔF, ΔB)
  Backtrack in IDLEL, . . . , IDLE0 to obtain optimal offload decisions
```

---

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

Once  $\text{IDLE}_{L+1}$  is computed, it can be updated by computing the corresponding idle time  $\epsilon_G$  between the forward and backward phases. Then, the smallest value in  $\text{IDLE}_{L+1}$  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 in the following way:  $M_F$  and  $M_B$  are between 0 and  $M_{\text{GPU}}$ ,  $\Delta_F$  is between  $-\sum_i (u_{F_i} + u_{B_i})B$  and  $\sum_i x[i]$ , and  $\Delta_B$  is fixed once the three other values are specified. The number of possible values is thus  $O(M_{\text{GPU}}^2(\sum_i x[i] + u_{F_i}B + u_{B_i}B))$ , and the complexity of Algorithm 1 is  $O(LM_{\text{GPU}}^2(\sum_i x[i] + u_{F_i}B + u_{B_i}B))$ , which is indeed pseudo-polynomial.

## 6 Experimental Analysis

This section presents experimental results obtained on three different kinds of networks, whose implementation is available in the `torchvision` package of PyTorch: ResNet, DenseNet, and Inception v3.

### 6.1 Experimental Setting

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  $u_F, u_B, \text{ex}^F, \text{ex}^B$ , and the sizes of  $x_i$  and  $y_i$  by performing measures on sample data. These measurements were performed on a node equipped with a Nvidia Tesla V100-PCIE GPU card with 15.75GB of memory. We also measured the bandwidth  $B$  to transfer data using PyTorch from the GPU to the RAM, and obtained around 12.2GB/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  $224 \times 224$  (which is the default and minimal image size for all models of `torchvision`), medium images of shape  $500 \times 500$ , and large images of shape  $1000 \times 1000$ . During the training phase, for higher efficiency, it is classical to process images in *batches*, where several

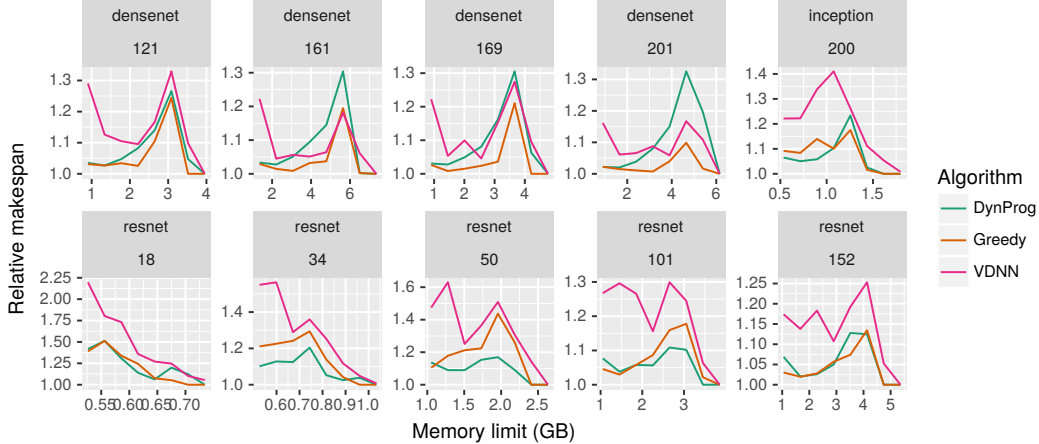


Figure 4: Experimental results for image size 224 and batch size 32.

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 three different algorithms: GREEDY (Section 4), DYNPROG (Algorithm 1), and VDNN.

To keep the computing time of DYNPROG reasonable, all memory values are discretized: we fix a number  $S$  of memory slots (500 is a reasonable value in practice), where each slot has size  $\frac{M}{S}$ , and all memory sizes are expressed as an integer number of slots, rounded up if necessary. Algorithm 1 outputs the optimal offloading decisions for the fractional communication relaxation, and DYNPROG returns the no-wait and ordered schedule with the same offload decisions.

VDNN is a heuristic based on ideas from VDNN++ [9]. The original VDNN++ offloads only the input of convolutions, because "*CONV layers have a much longer computation latency, being more likely to effectively hide the latency of offload/prefetch*", and explores two possibilities: either offload the output of every convolution, or every other convolution. However, in our setting the layers are not annotated to know which ones corresponds to convolutions. Hence, in VDNN, we first compute the ratio  $\frac{u_{F_i}}{|x_i|}$  for all operations, and then for all possible thresholds, we compute the no-wait, ordered schedule which offloads all the activations whose ratio is above the threshold, and the one which offloads half of them. VDNN outputs the best schedule out of all these choices.

## 6.2 Representative Results

A representative selection of achieved results is depicted in Figure 4. 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 as given by Proposition 2, and the plots show the ratio of the makespan obtained by each algorithm to the lower bound. We observe that both GREEDY and DYNPROG outperform the VDNN heuristic in most cases, especially in low memory scenarios. However, DYNPROG obtains relatively poor performance on the largest DenseNet networks. This comes from the discretization errors, that accumulate over long chains ( $L$  is about 100 in these cases). Still, the ratio of DYNPROG remains below 1.3, which is much more stable than the overall results of VDNN.

We next present the complete set of results, with two different presentations: we first show relative makespan like in the previous plot, which helps in comparing the performance of heuristics. Then we show the throughput (number of images processed by second) obtained for each case, which is the metric of interest for training DNNs.

### 6.3 Complete Results – Ratios

To allow for a better view of the results, the memory limits on the next plots are scaled: 0% corresponds to the amount of memory required to process the chain, and 100% corresponds to the memory peak  $M_{peak}$ . The results are shown on Figures 5-11.

### 6.4 Experimental results – Throughput

On the next plots, we show raw results, without normalizing with the lower bound. The  $y$  axis shows the throughput obtained, which is the number of images processed by seconds, and it computed with  $T = \frac{\text{batch size}}{\text{makespan}}$ . The results are shown on Figures 12-18.

## 7 Conclusions

In this paper, we address the problem of memory usage during the training phase of Deep Neural Networks. Previous works [8, 9] have advocated to offload some of the data onto the memory of the CPU, 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-hard in the strong sense, and we propose two heuristics based on simpler 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. It allows to obtain better solutions in favorable cases, but achieves worse performance with deeper neural networks, because of the discretization scheme. In any case, both algorithms provide significant improvements over the previous approach.

This work opens several promising research directions. A validation with real experiments 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 seems to be complementary of the checkpointing approach: some activations can be transferred to the CPU while others can be recomputed. Solving the corresponding algorithmic problem might be challenging, but would certainly yield significant improvement for training large and deep models.

## References

- [1] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.
- [2] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
- [3] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.
- [4] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, large minibatch sgd: Training imagenet in 1 hour,” 2017.
- [5] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, “Imagenet training in minutes,” 2017.
- [6] S. Rota Bulò, L. Porzi, and P. Kotschieder, “In-place activated batchnorm for memory-optimized training of dnns,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5639–5647.

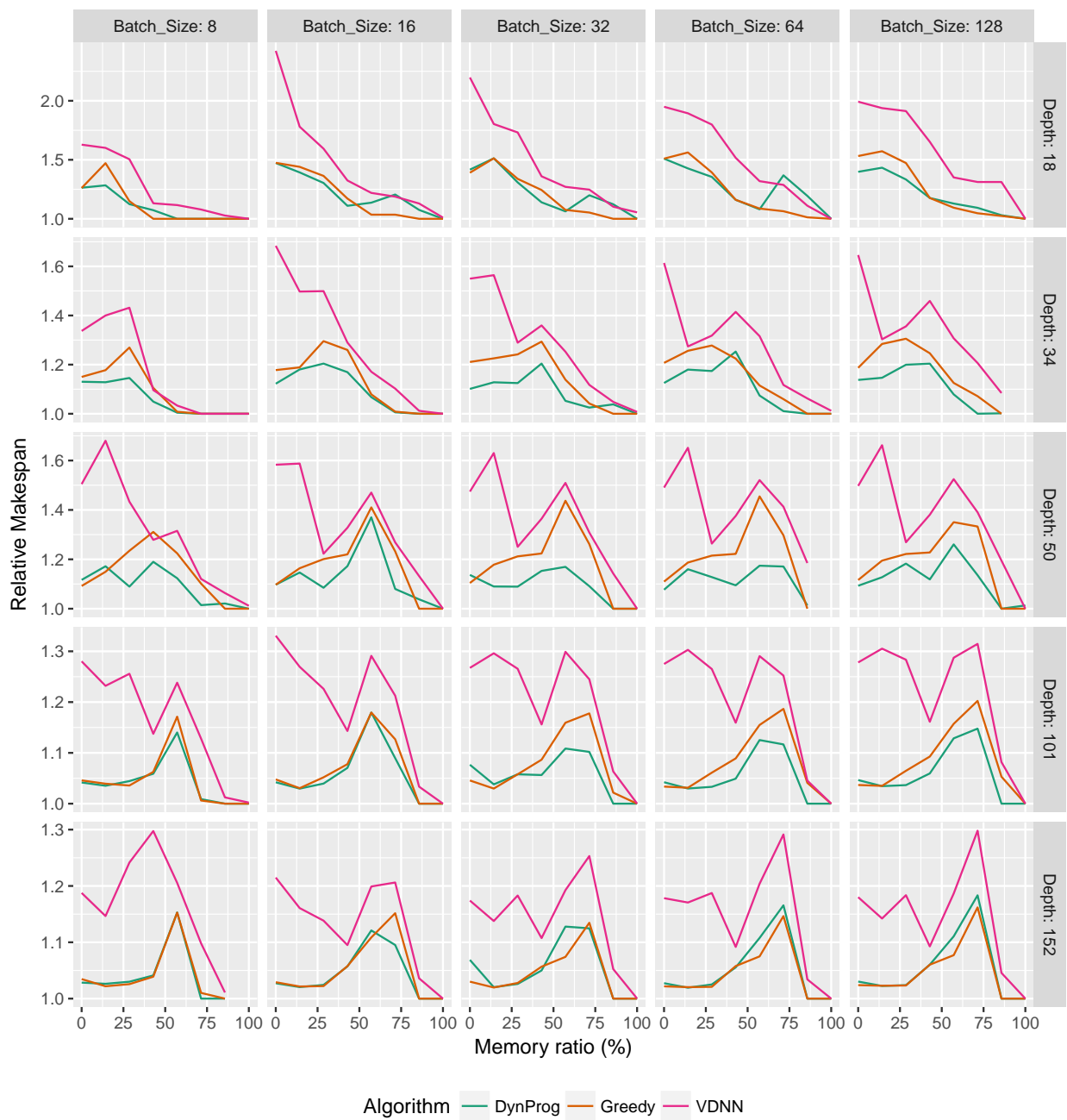


Figure 5: Relative Makespan results for ResNet with small images (224x224).

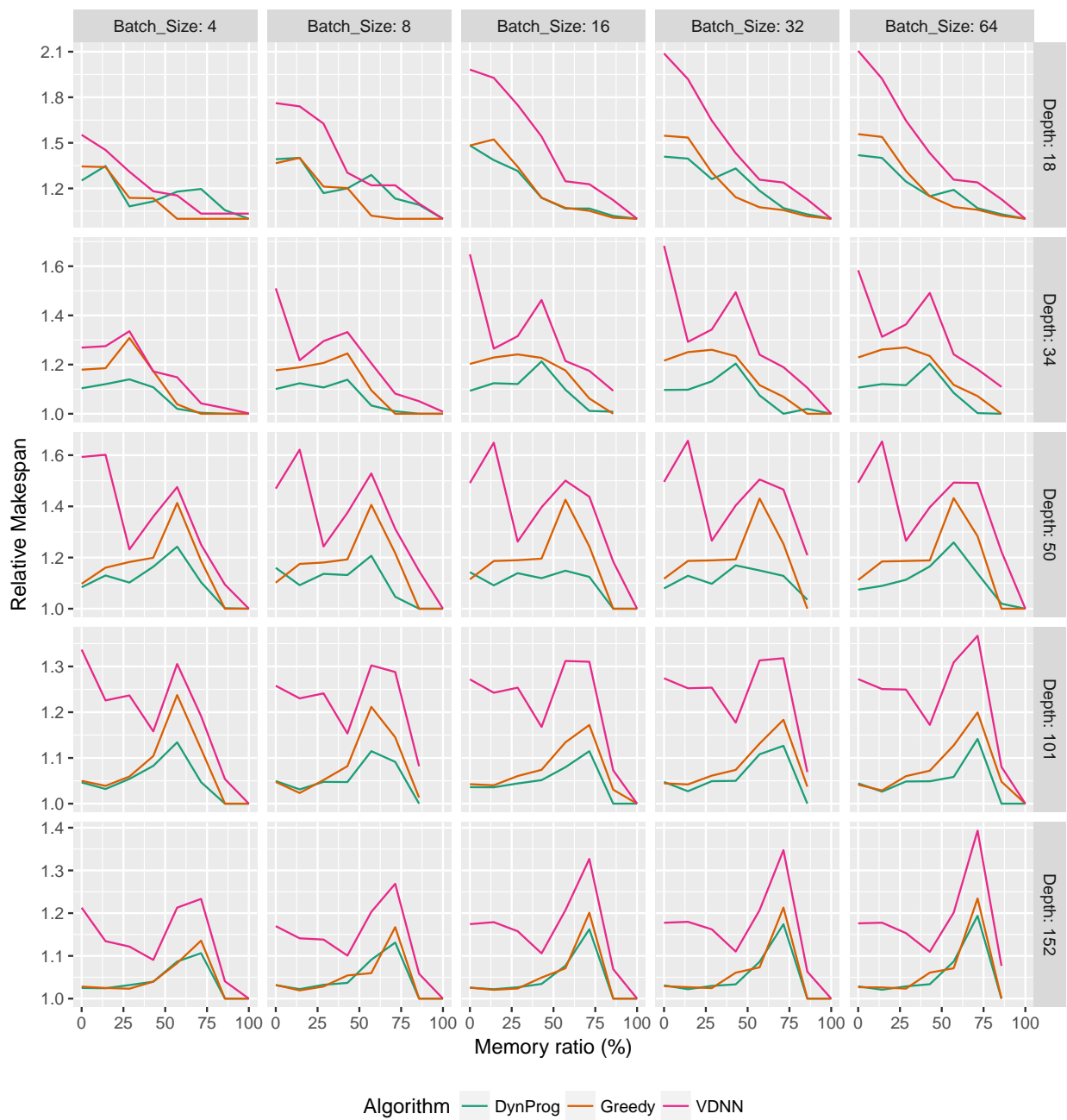


Figure 6: Relative Makespan results for ResNet with medium images (500x500).

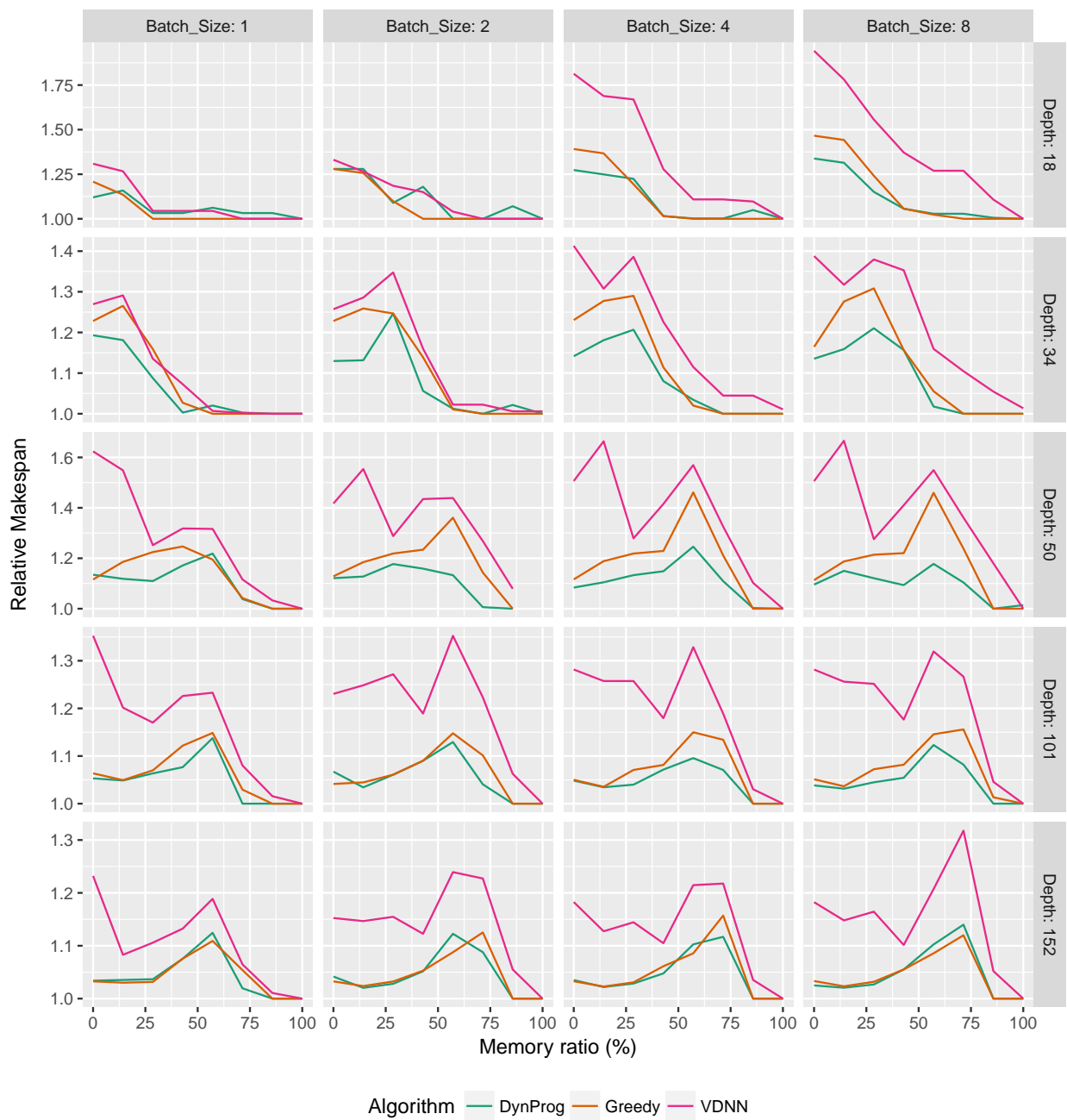


Figure 7: Relative Makespan results for ResNet with large images (1000x1000).

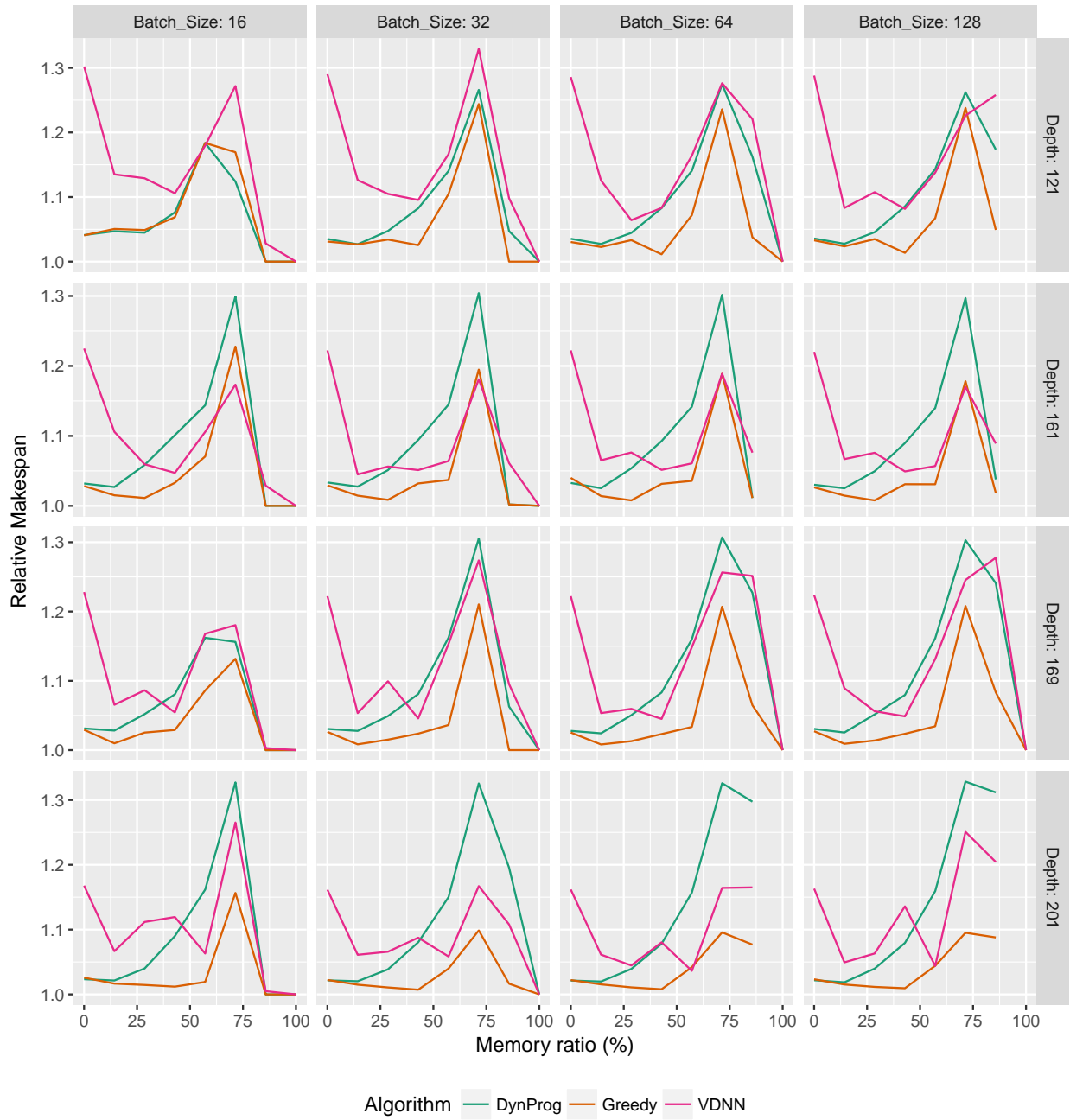


Figure 8: Relative Makespan results for DenseNet with small images (224x224).



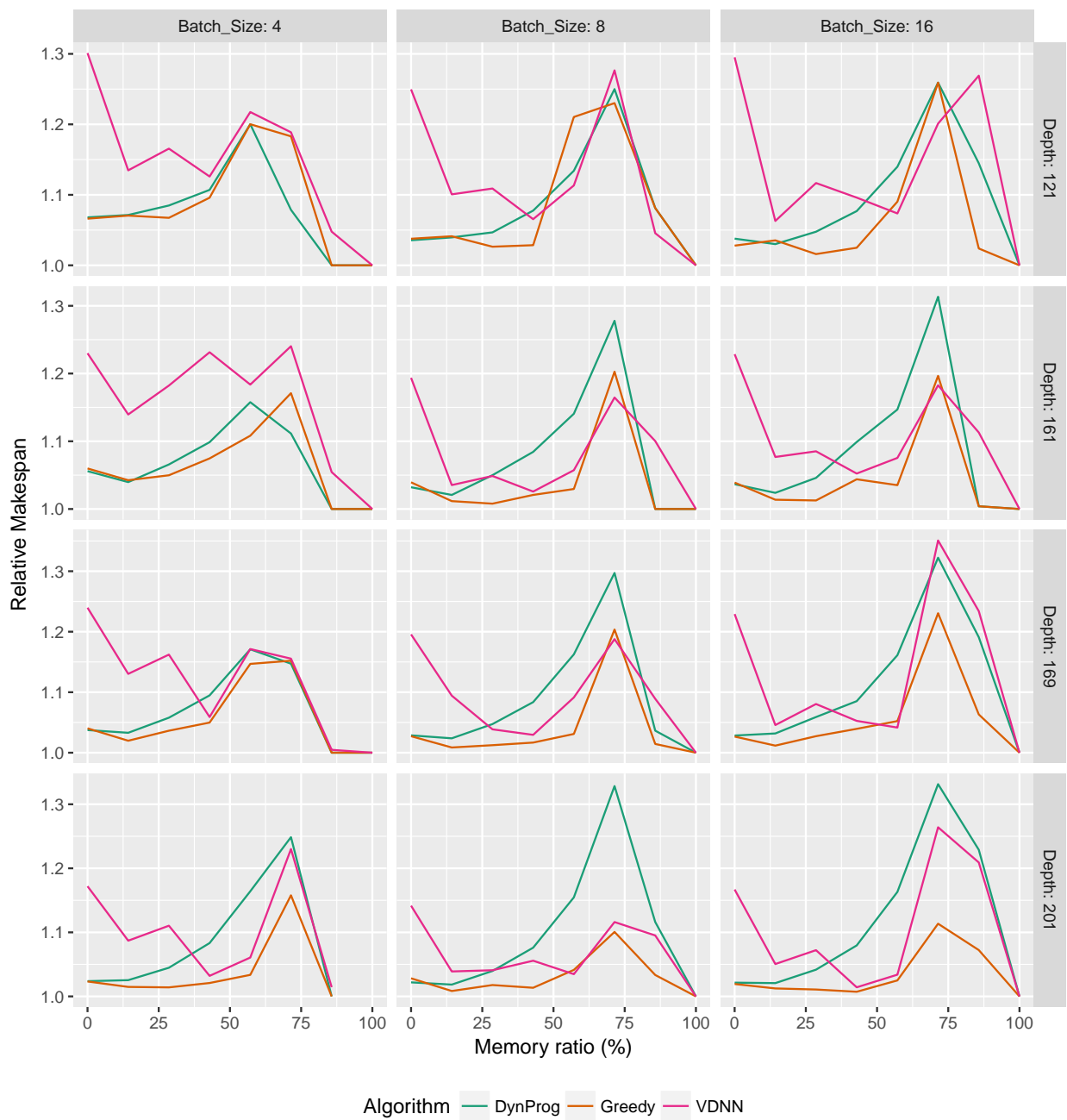


Figure 9: Relative Makespan results for DenseNet with medium images (500x500).

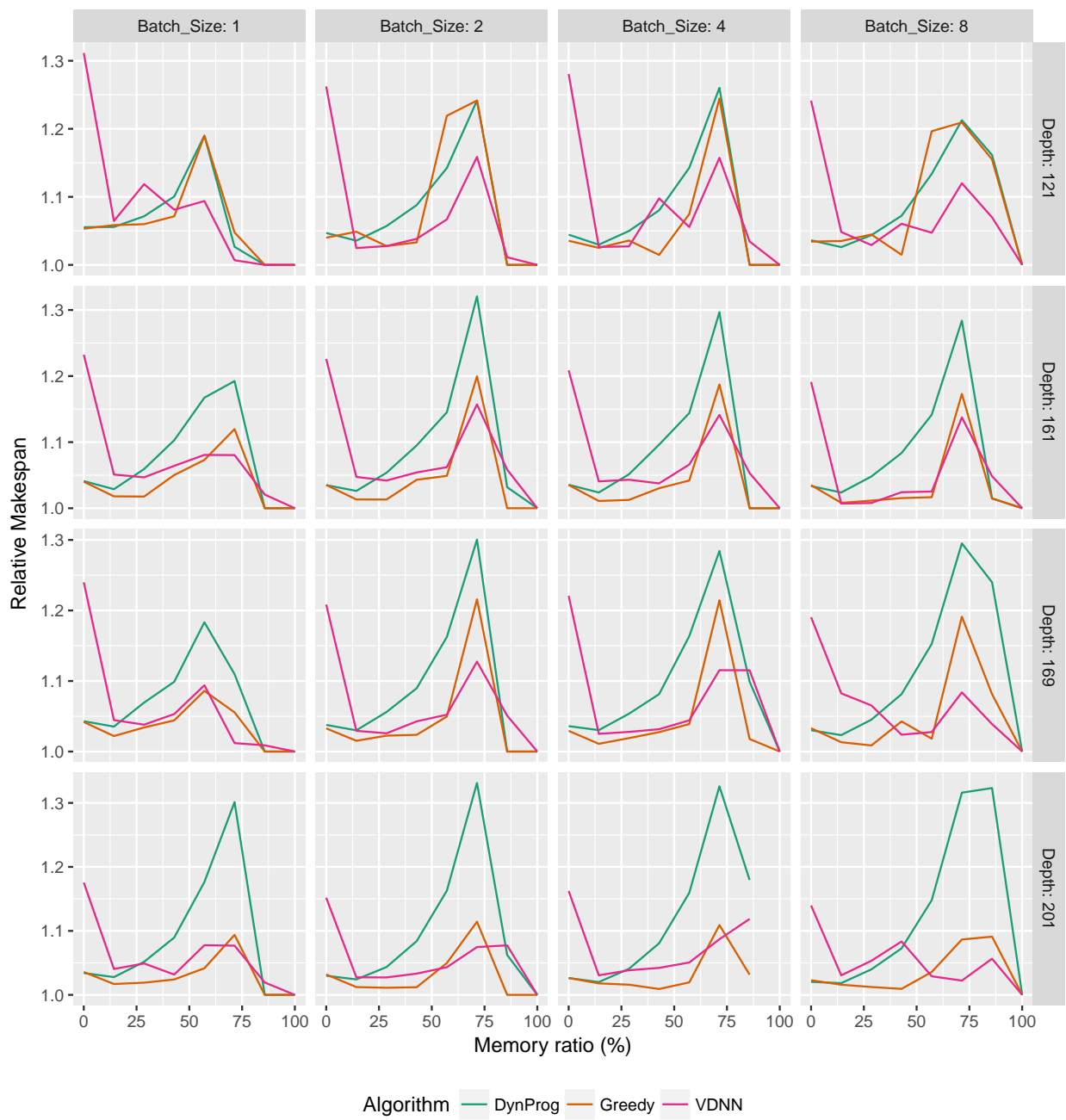


Figure 10: Relative Makespan results for DenseNet with large images (1000x1000).

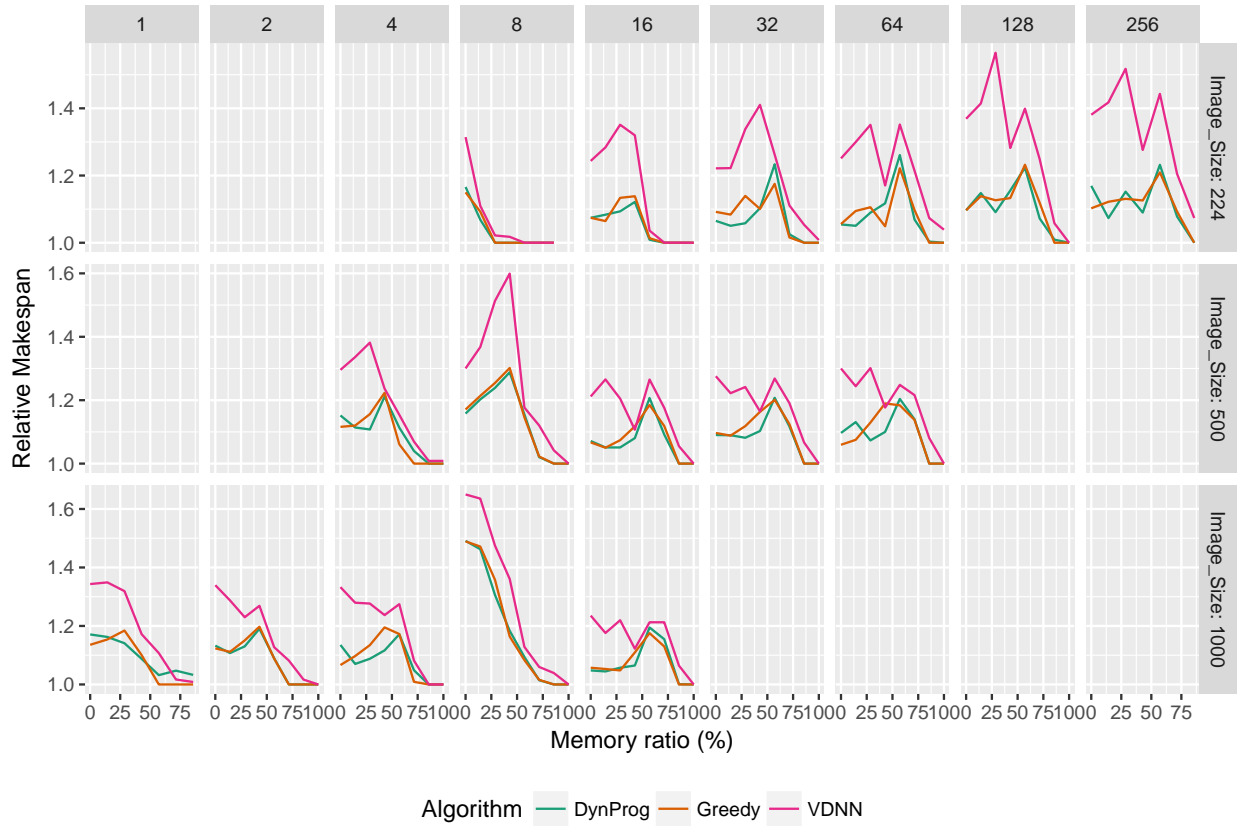


Figure 11: Relative Makespan results for Inception.

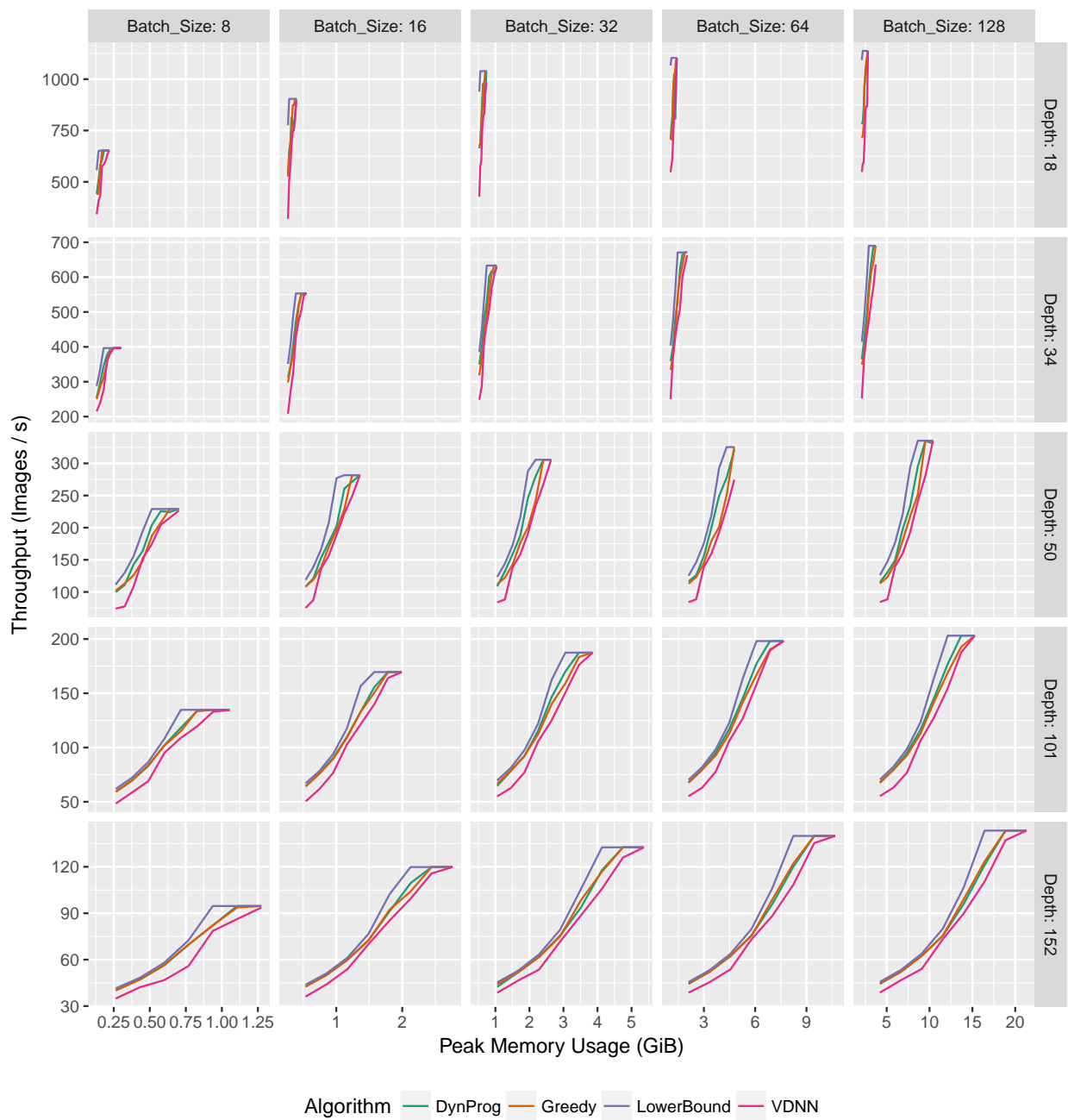


Figure 12: Throughput results for ResNet with small images (224x224).

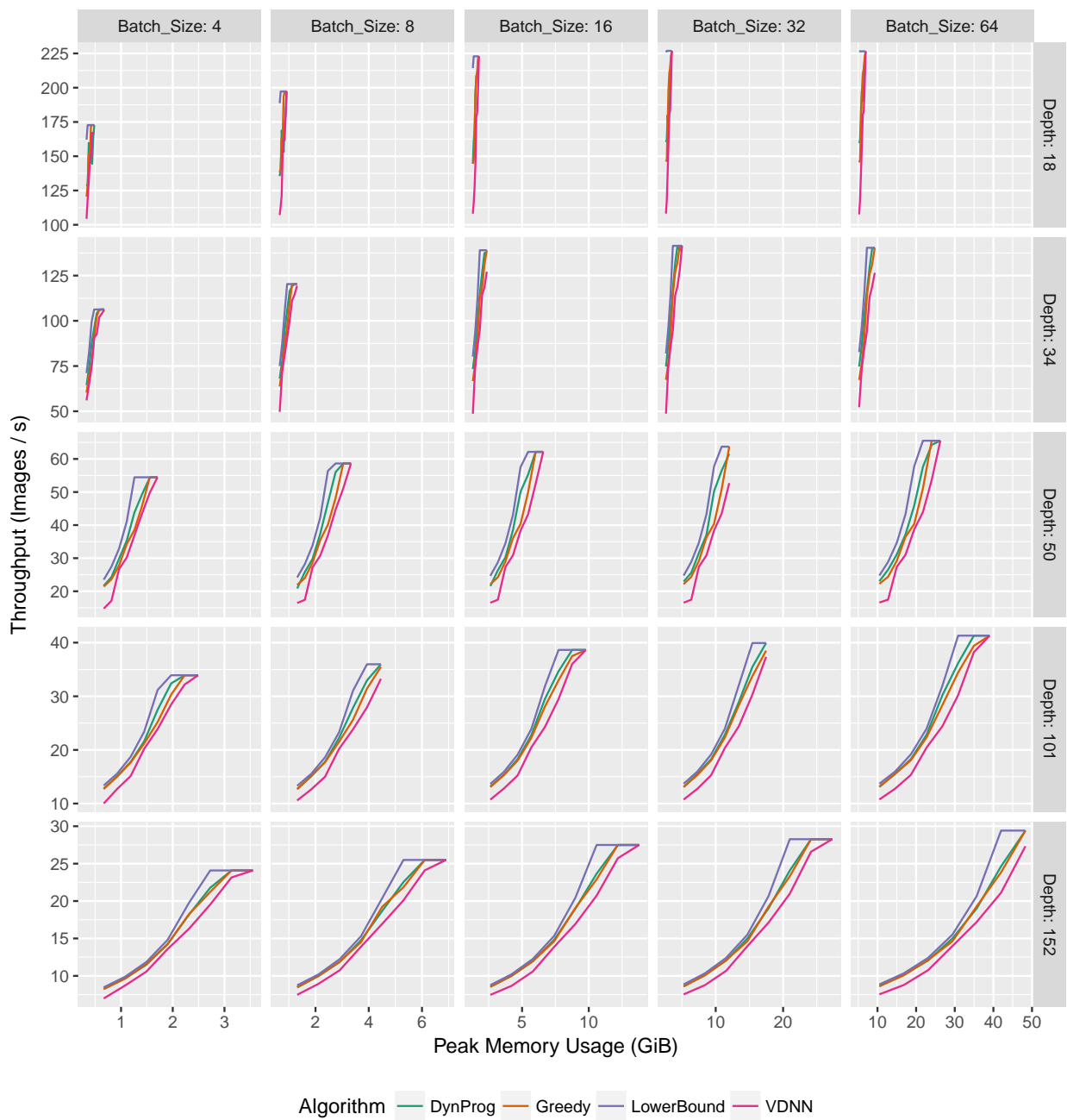


Figure 13: Throughput results for ResNet with medium images (500x500).

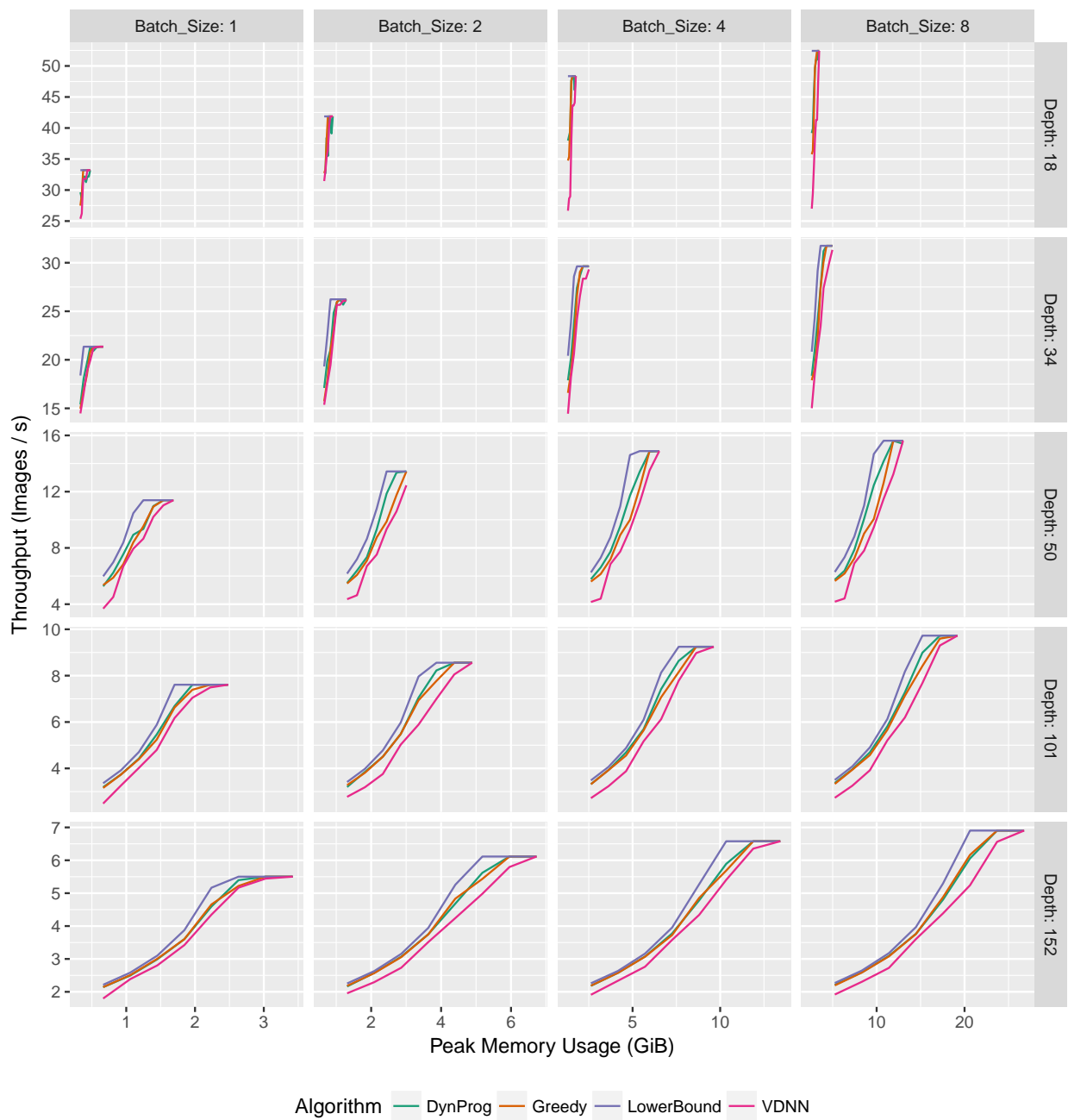


Figure 14: Throughput results for ResNet with large images (1000x1000).

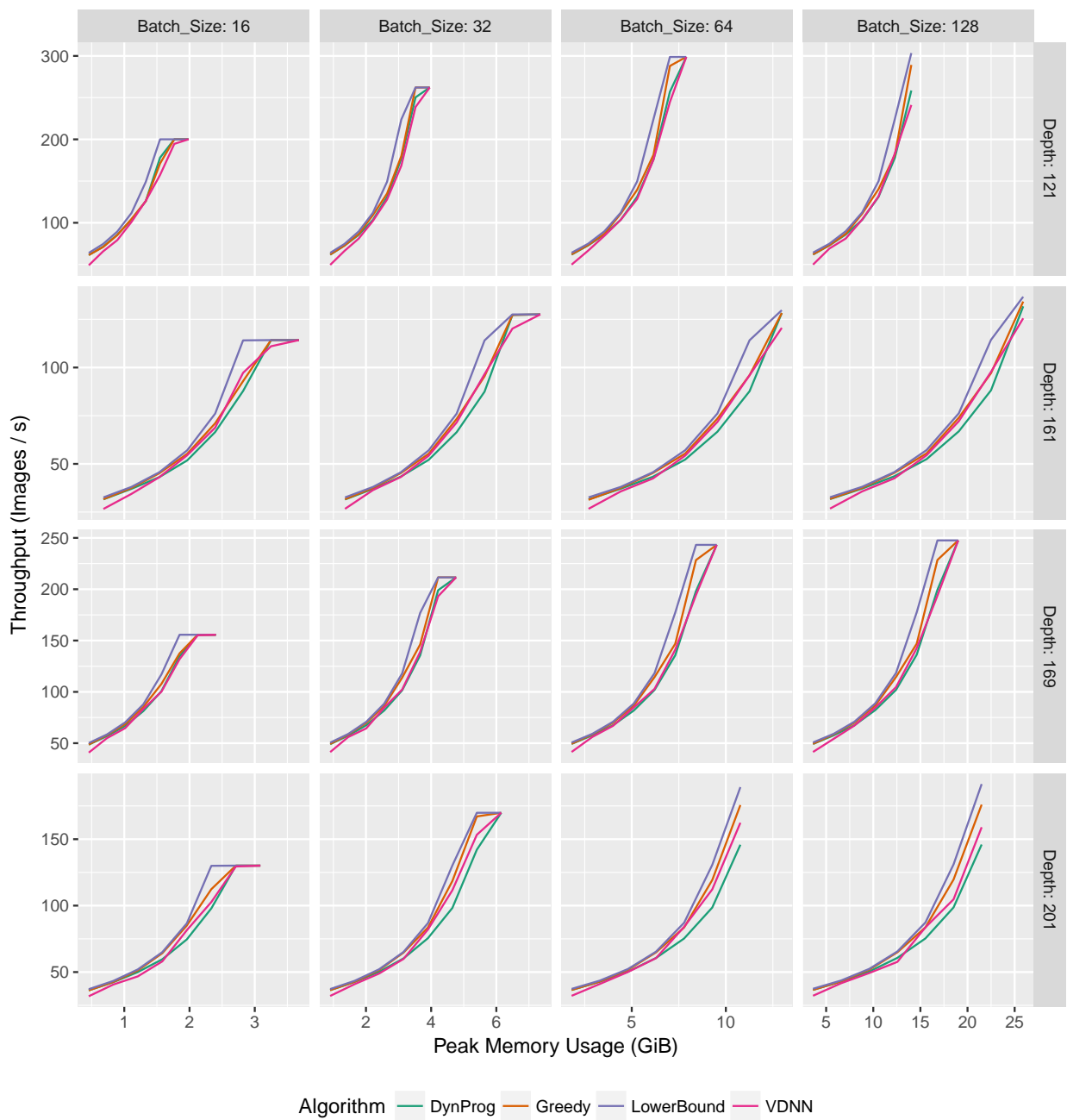


Figure 15: Throughput results for DenseNet with small images (224x224).

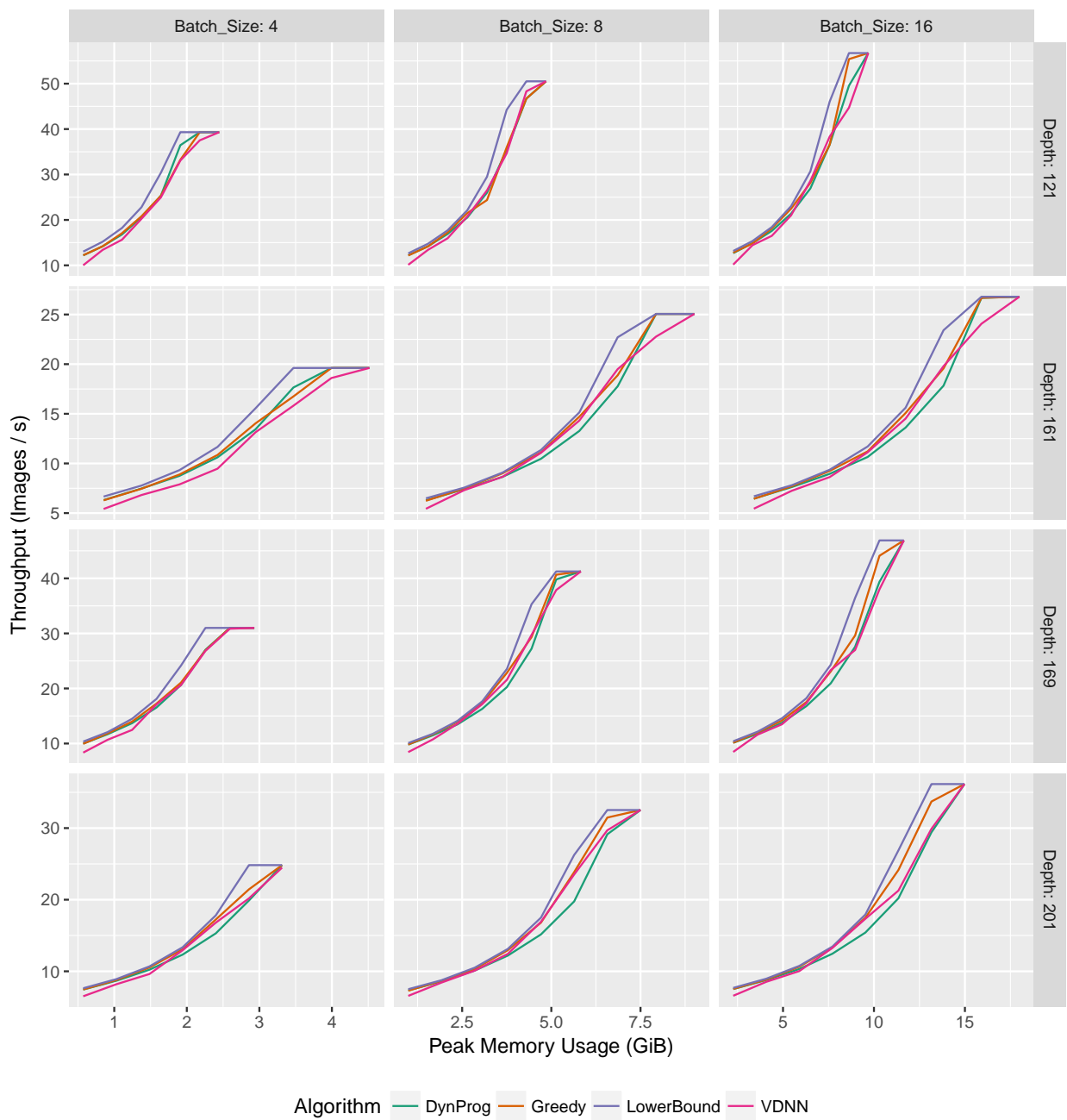


Figure 16: Throughput results for DenseNet with medium images (500x500).



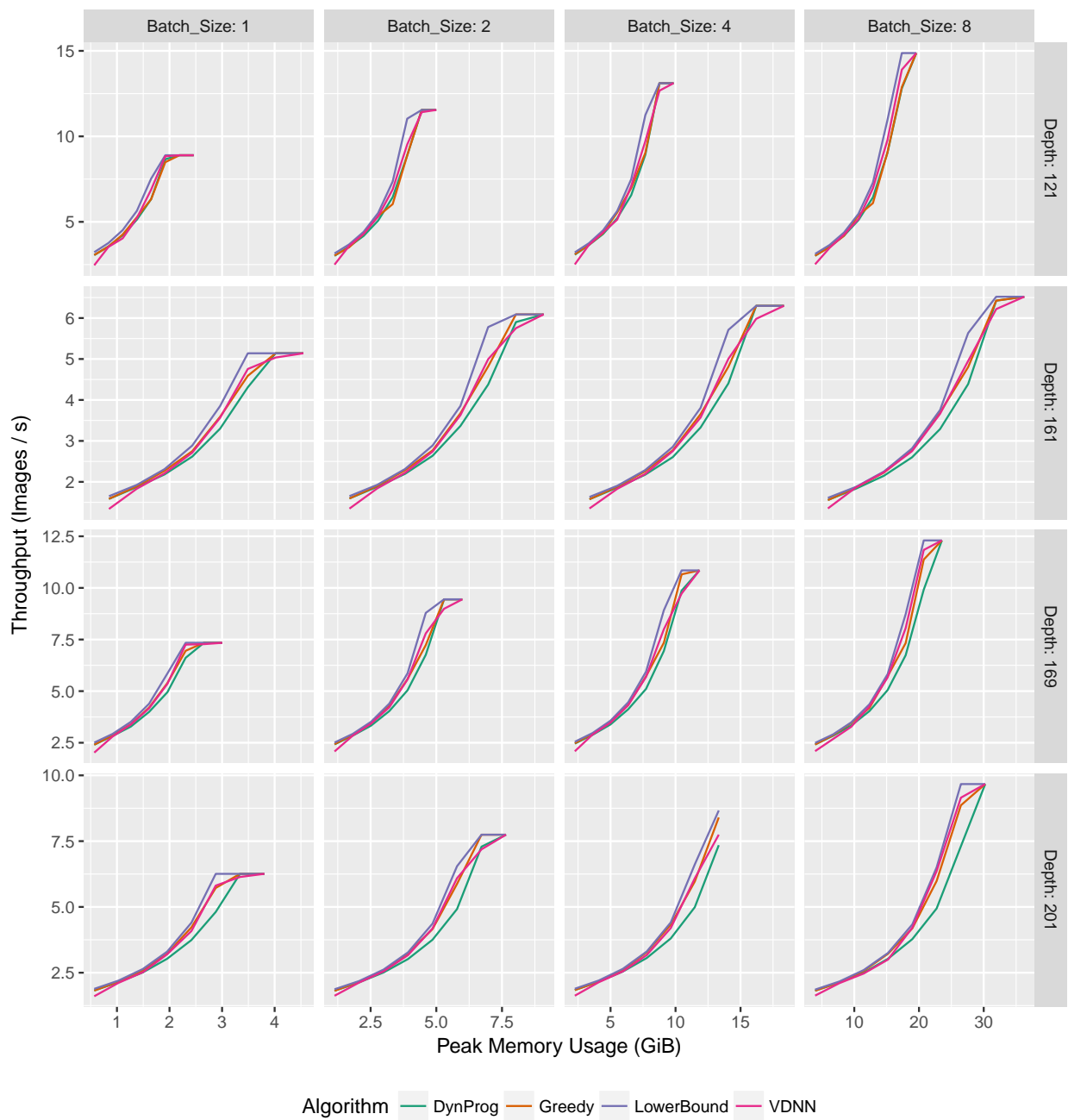


Figure 17: Throughput results for DenseNet with large images (1000x1000).

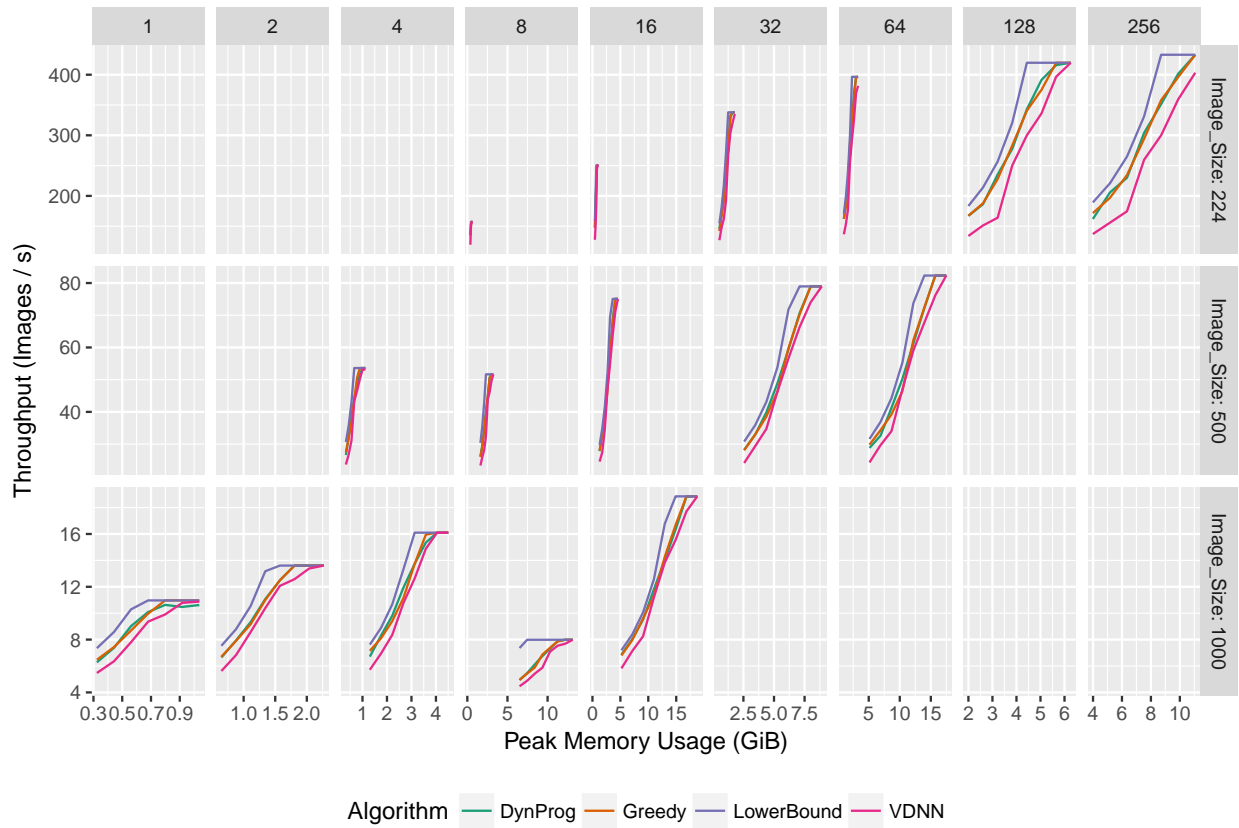


Figure 18: Throughput results for Inception.

- [7] G. Pleiss, D. Chen, G. Huang, T. Li, L. van der Maaten, and K. Q. Weinberger, “Memory-efficient implementation of densenets,” *arXiv preprint arXiv:1707.06990*, 2017.
- [8] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, “vdmn: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 18.
- [9] S. S. B, A. Garg, and P. Kulkarni, “Dynamic memory management for gpu-based training of deep neural networks,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Press, 2019.
- [10] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, “Parallelized stochastic gradient descent,” in *Advances in neural information processing systems*, 2010, pp. 2595–2603.
- [11] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang, “Gpu asynchronous stochastic gradient descent to speed up neural network training,” *arXiv preprint arXiv:1312.6186*, 2013.
- [12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, “Large scale distributed deep networks,” in *Advances in neural information processing systems*, 2012, pp. 1223–1231.
- [13] A. Griewank, “On automatic differentiation,” *Mathematical Programming: recent developments and applications*, vol. 6, no. 6, pp. 83–107, 1989.
- [14] A. Adcroft, J. Campin, S. Dutkiewicz, C. Evangelinos, D. Ferreira, G. Forget, B. Fox-Kemper, P. Heimbach, C. Hill, E. Hill *et al.*, “Mitgcm user manual,” 2008.
- [15] P. Brubaker, *Engineering Design Optimization using Calculus Level Methods*, 2016.
- [16] A. Griewank and A. Walther, “Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 1, pp. 19–45, 2000.
- [17] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, “Memory-efficient backpropagation through time,” in *Advances in Neural Information Processing Systems*, 2016, pp. 4125–4133.
- [18] T. Chen, B. Xu, C. Zhang, and C. Guestrin, “Training deep nets with sublinear memory cost,” *arXiv preprint arXiv:1604.06174*, 2016.
- [19] N. Kukreja, J. Hückelheim, and G. J. Gorman, “Backpropagation for long sequences: beyond memory constraints with constant overheads,” *arXiv preprint arXiv:1806.01117*, 2018.
- [20] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.
- [21] “Periodic checkpointing in pytorch,” 2018, <https://pytorch.org/docs/stable/checkpoint.html>.