HAL open science

Memory Saving Strategies for Deep Neural Network Training

Alena Shilova

To cite this version:

Alena Shilova. Memory Saving Strategies for Deep Neural Network Training. Artificial Intelligence [cs.AI]. Université de Bordeaux, 2021. English. NNT : 2021BORD0335 . tel-03631459

HAL Id: tel-03631459

https://theses.hal.science/tel-03631459v1

Submitted on 5 Apr 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

SPÉCIALITÉ: INFORMATIQUE

Par **Alena SHILOVA**

Memory Saving Strategies for Deep Neural Network Training

Sous la direction de : **Olivier BEAUMONT et Alexis JOLY**
Co-encadrant : **Lionel EYRAUD-DUBOIS**

Soutenue le 7 décembre 2021

Membres du jury :

Anne Benoit	Maîtresse de Conférences	École Normale Supérieure de Lyon	Rapporteur
Ivan Oseledets	Professor	Skolkovo Institute of Science and Technology	Rapporteur
Elisabeth Brunet	Maîtresse de Conférences	Télécom Sud Paris	Examinatrice
Bruno Raffin	Directeur de Recherche	Inria	Président
Laurent Simon	Professeur	Bordeaux INP	Examinateur
Olivier Beaumont	Directeur de Recherche	Inria	Co-directeur
Alexis Joly	Directeur de Recherche	Inria	Co-directeur
Lionel Eyraud-Dubois	Chargé de Recherche	Inria	Co-encadrant

Stratégies pour économiser la mémoire lors de l'apprentissage dans les réseaux neuronaux profonds

Résumé : L'intelligence artificielle est un domaine qui a reçu beaucoup d'attention récemment. Son succès est dû aux progrès du Deep Learning, un sous-domaine qui réunit des méthodes d'apprentissage automatique basées sur les réseaux neuronaux. Ces réseaux neuronaux ont prouvé leur efficacité pour résoudre des problèmes très complexes dans différents domaines. Cependant, leur efficacité pour résoudre des problèmes dépend d'un certain nombre de facteurs : l'architecture du modèle, sa taille, comment et où l'entraînement a été effectué. La plupart des études indiquent que les modèles les plus gros permettent d'obtenir une meilleure précision, mais ils sont également plus coûteux à entraîner. Les principaux défis sont liés à la puissance de calcul et à la mémoire restreinte des machines : si le modèle est trop grand, son apprentissage peut prendre beaucoup de temps (des jours, voire des mois) ou, dans le pire des cas, il peut même ne pas tenir en mémoire. Pendant l'apprentissage, il est nécessaire de stocker à la fois les poids (paramètres du modèle), les activations (données calculées intermédiaires) et les états de l'optimiseur.

Cette situation offre plusieurs opportunités pour traiter les problèmes de mémoire, en fonction de leur origine. L'apprentissage peut être distribué sur plusieurs ressources de la plate-forme de calcul, et différentes techniques de parallélisation offrent différentes manières de distribuer la mémoire. En outre, les structures de données qui restent inactives pendant une longue période peuvent être temporairement déchargées vers un espace de stockage plus important, avec la possibilité de les récupérer ultérieurement (stratégies de déchargement). Enfin, les activations qui sont calculées à chaque itération peuvent être supprimées et recalculées plusieurs fois au cours de celle-ci (stratégies de rematérialisation). Les stratégies pour économiser la mémoire induisent généralement un surcoût en temps par rapport à l'exécution directe, par conséquent des problèmes d'optimisation doivent être considérés afin de choisir la meilleure approche pour chaque stratégie. Dans ce manuscrit, nous formulons et analysons des problèmes d'optimisation en relation avec diverses méthodes visant à réduire la consommation de mémoire pendant le processus d'apprentissage. En particulier, nous nous concentrons sur les stratégies de rematérialisation, de déchargement d'activations et de parallélisme de modèles pipelinés ; pour chacune d'entre elles, nous concevons les solutions optimales sous un ensemble d'hypothèses. Enfin, nous proposons un outil entièrement fonctionnel appelé ROTOR qui combine le déchargement d'activations et la rematérialisation et qui peut être utilisé pour l'entraînement de grands modèles avec une surcharge minimale dans PyTorch, des modèles qui autrement ne tiendraient pas dans la mémoire.

Mots-clés : Apprentissage profond, Rétropropagation, Rematérialisation, Déchargement, Parallélisme du modèle en pipeline

---

Memory Saving Strategies for Deep Neural Network Training

Abstract: Artificial Intelligence is a field that has received a lot of attention recently. Its success is due to advances in Deep Learning, a sub-field that groups together machine learning methods based on neural networks. These neural networks have proven to be effective in solving very complex problems in different domains. However, their effectiveness depends on a number of factors: the architecture of the model, its size, how and where the training is performed... Most studies indicate that the large models are more likely to achieve the smallest error, but they are also more difficult to train. The main challenges are related to insufficient computational power and limited memory of the machines: if the model is too large then it can take a long time to be trained (days or even months), or it cannot even fit in memory in the worst case. During the training, it is necessary to store the weights (model parameters), the activations (intermediate computed data) and the optimizer states.

This situation offers several opportunities to deal with memory problems, depending on their origin. Training can be distributed across multiple resources of the computing platform, and different parallelization techniques suggest different ways of dividing memory load. In addition, data structures that remain inactive for a long period of time can be temporarily offloaded to a larger storage space with the possibility of retrieving them later (offloading strategies). Furthermore, activations that are computed anew at each iteration can be deleted and recomputed several times within it (rematerialization strategies). Memory saving strategies usually induce a time overhead with respect to the direct execution, therefore optimization problems should be considered to choose the best approach for each strategy. In this manuscript, we formulate and analyze optimization problems in relation to various methods reducing memory consumption of the training process. In particular, we focus on rematerialization, activation offloading and pipelined model parallelism strategies, for each of them we design optimal solutions under a set of assumptions. Finally, we propose a fully functional tool called ROTOR that combines activation offloading and rematerialization and that can be applied to training in PyTorch, allowing to process big models that otherwise would not fit into memory.
Keywords: Deep Learning, Backpropagation, Rematerialization, Offloading, Pipelined Model Parallelism

---

Equipe-projet HiePACS
Inria Bordeaux - Sud-Ouest, 33405 Talence, FRANCE.

Acknowledgement

Writing this thesis has been a great 3-year journey that has helped me to learn a lot, grow professionally and meet a lot of wonderful people. Its successful completion would have not been possible if not for the support of a lot of people.

First of all, I would like to thank my advisors who have been very supportive all along the way: Olivier Beaumont for your very warm welcome and encouraging me to do more research and less studying, your ideas and intuition are always a great source of inspiration, while your organizational skills helped me to keep track of time; Lionel Eyraud-Dubois for listening and explaining, you helped me to see complicated things in a simple way and significantly improve my coding skills and thanks to you we can proudly offer ROTOR to the world; Alexis Joly for making me a part of Pl@ntNet, hosting me in Montpellier and your valuable expertise in Deep Learning, you gave us the model and the problem and thanks to you I learned a lot about practical aspects of Deep Learning.

Furthermore, I would like to express my gratitude to my reviewers Anne Benoit and Ivan Oseledets for careful reading of my manuscript and valuable comments that helped me to improve it afterwards. Moreover, I would like to thank as well my examiners Elisabeth Brunet, Laurent Simon and Bruno Raffin for attending my defense and asking interesting questions that led to an interesting discussion about future potential development of this thesis and thank you for granting me the title of the doctor, I will do my best to live up to it. In particular, special thanks to Bruno Raffin for leading IPL project Defi that allowed researchers from HPC, Big Data and AI to meet and collaborate, which in turn initiated this PhD thesis.

Many thanks to my collaborators and coauthors. I learned a lot from every one of you and I hope that we will meet and work together soon again.

I would like also to address myself to the people that I met in Bordeaux and in particular in Inria. All of you are so different, remarkable and cool at the same time. It was my immense pleasure to get to know you, learn from you different cultures and food, making friendship. Playing “baby-foot” and board games, “pause gouter”, having millions of tea at Inria sun-lit terrace, “apero sorties”, “apero plages” and many other happy moments I will cherish and remember all my life. I would like to thank you all for this amazing time and wish good luck to next soon-to-be doctors Esra, Martina, Yanfei, Mathieu, Romain, Marek, Xunyi, Jean-Francois. Special thanks to Martina for helping me to print the first version of my manuscript for my defense. Even more special thanks to Aurélien for going through fire and water with me.

Finally, thanks to my family and friends that stayed in Russia. Despite the distance

and world cataclysms, we stay connected, which is especially crucial in the hard times. I look forward to seeing you soon again.

Оглавление

Оглавление **vii**

Résumé long **1**
Apprentissage sur un seul noeud . . . 4
Apprentissage sur plusieurs nœuds . . . 7
Historique . . . 9

Introduction **11**
Training on a Single Device . . . 13
Training on Multiple Nodes . . . 16
Background . . . 18

Related Works **21**
Rematerialization . . . 21
Offloading . . . 24
Pipelined Model Parallelism . . . 25

I Rematerialization **29**
Introduction . . . 31

1 Multi-Chain Rematerialization **35**
1.1 Framework . . . 35
1.1.1 Introduction to Adjoint Chain . . . 35
1.1.2 Platform Model and Optimization Problem . . . 36
1.1.3 Backpropagation Graphs . . . 38
1.1.4 Multiple Adjoint Chains Computation Problem . . . 38
1.1.5 Previous Results for Single Adjoint Chain Computation Problem . . . 41
1.2 Characterization of Optimal Solutions for Multi-Adjoint Chains . . . 42
1.2.1 Motivating Example . . . 43
1.2.2 Canonical Form of Optimal Solutions . . . 45
1.2.2.1 Valid Schedule . . . 46
1.2.2.2 Canonical Solutions . . . 47
1.3 Optimal Solution for Problem MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ . . . 50

1.3.1 Minimal Memory Requirement . . . 50
1.3.2 Optimal Solution . . . 51
1.4 Simulation Results . . . 53
1.5 Conclusion . . . 55

2 Rematerialization for Heterogeneous Chains 59
2.1 Modeling and Problem Formulation . . . 59
2.1.1 Adjoint Computation Graphs for Deep Neural Networks . . . 60
2.1.2 Rematerialization Operations and Memory Usage . . . 61
2.2 Complexity . . . 64
2.3 Optimal Rematerialization Algorithm . . . 65
2.3.1 Considerations on Memory Persistence . . . 66
2.3.2 Properties of Heterogeneous Problem . . . 67
2.3.3 Dynamic Programming with Floating Materialized Activations . . . 69
2.3.4 Optimal Memory Persistent Solution . . . 73
2.4 Implementation and Validation . . . 76
2.4.1 Parameter Estimation . . . 77
2.4.2 Computation of the Optimal Sequence . . . 77
2.4.3 Comparison between Algorithm 1 and Algorithm 3 . . . 78
2.4.4 Experimental Setting . . . 78
2.4.5 Experimental Results . . . 80
2.5 Conclusion . . . 83
2.6 Additional Plots . . . 84

II Offloading 93
Introduction . . . 95
Model and Main Notations . . . 95

3 Offloading for Heterogeneous Chains 99
3.1 Preliminary Analysis . . . 99
3.1.1 Preliminary Results and Lower Bound . . . 99
3.1.2 Complexity Results . . . 100
3.2 Fractional Relaxation . . . 102
3.2.1 Structure of Optimal Solutions . . . 102
3.2.2 Greedy Algorithm . . . 103
3.3 Fractional Communications . . . 104
3.3.1 Complexity . . . 105
3.3.2 Structure of Optimal Solutions . . . 106
3.3.2.1 Forward and Backward Phases . . . 108
3.3.2.2 Idle Time between Phases . . . 110
3.3.3 Resulting Algorithm . . . 112
3.3.3.1 Dynamic Programming Algorithm . . . 112

3.3.3.2 Discretization Scheme . . . 113
3.4 Experimental Analysis . . . 114
3.4.1 Experimental Setting . . . 114
3.4.2 Representative Results . . . 116
3.4.3 Comparison to Rematerialization . . . 118
3.4.4 Complete Results – Ratios . . . 118
3.4.5 Complete Results – Througphut . . . 118
Conclusions . . . 118

4 Combination of Offloading and Rematerialization 135
4.1 Combination of Offloading and Rematerialization . . . 135
4.1.1 Complexity . . . 135
4.1.2 Additional Assumptions . . . 137
4.1.3 Rationale of the Different Operations . . . 138
4.1.4 Intuition of the Overall Scheme and State Variables . . . 138
4.2 Dynamic Programming to Compute the Optimal Sequence . . . 141
4.2.1 Forward Phase . . . 141
4.2.2 Loss: How to Concatenate Forward and Backward Phases . . . 143
4.2.3 Backward Phase . . . 144
4.2.4 Complexity . . . 146
4.3 Heuristics . . . 146
4.4 Experiments . . . 147
4.4.1 Simulation Results . . . 147
4.4.2 Experimental Results with ROTOR . . . 151
4.5 Conclusion . . . 151

III Model Parallelism 153
Introduction . . . 155
Model and Notations . . . 156

5 Pipelined Model Parallelism. Complexity 161
5.1 Complexity Results . . . 161
5.1.1 General Problem . . . 161
5.1.2 Fixed Allocation Problem . . . 162
5.2 General Periodic Schedules for Contiguous Allocations . . . 163
5.2.1 Optimal 1-periodic Schedule . . . 164
5.2.2 k-periodic Schedules . . . 166
5.3 Contiguous vs General Allocations . . . 168
5.3.1 Without Memory Constraints . . . 168
5.3.2 With Memory Constraints . . . 169
5.4 Conclusion . . . 170

6 Integer Linear Programming Approach **171**
6.1 Notations . . . 171
6.2 Communication and Computation Constraints . . . 173
6.2.1 Limit on the Number of Resources . . . 173
6.2.2 Ordering of Computational Tasks . . . 174
6.2.3 Ordering of Communication Tasks . . . 175
6.2.4 Period Length . . . 177
6.3 Memory Constraints . . . 178
6.3.1 Memory for Direct Dependencies . . . 178
6.3.2 Memory Required for Local Activations . . . 179
6.3.3 Memory Required for the Models . . . 182
6.3.4 Memory Buffer for Communications . . . 183
6.4 Final Integer Linear Program . . . 184
6.5 Experimental Results . . . 184
6.6 Conclusion . . . 187

7 MadPipe **191**
7.1 Building a Non-Contiguous Allocation . . . 191
7.1.1 Estimating Memory Usage . . . 192
7.1.2 Dynamic Programming Derivations . . . 193
7.1.3 Find the Correct Value for $\widehat{T}$ . . . 195
7.2 Scheduling with ILP . . . 195
7.3 Experimental Results . . . 196
7.3.1 Simulation Settings . . . 196
7.3.2 Comparison with ILP . . . 197
7.3.3 Simulation Results . . . 197
7.4 Conclusion . . . 200

Conclusion **205**

Литература **209**

List of Algorithms

1 Computation of an optimal schedule for REMAT(L, M_{GPU}) 73
2 OptRecFL(C, s, t, ℓ, m) – Obtain optimal sequence from the table C 74
3 Optimal persistent schedule for a chain of length L with memory M_{GPU}. . 76
4 OptRecP(C, s, t, m) – Computation of the optimal persistent sequence from table C 77
5 Dynamic Programming Algorithm for Fractional Communications 111
6 Summary of Algorithm 1F1B* for a given period T. 164
7 Optimal solution of $\text{PIPE}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$ 167
8 First phase of MadPipe: build an allocation 195

Résumé long

L'intelligence artificielle (IA) est un domaine en plein essor qui aide à résoudre de nombreux problèmes complexes comme la classification d'images, la génération de textes, la traduction... Sa naissance remonte à 1956 lors de la tenue du Workshop de Darthmouth, où son nom et les premiers objectifs du domaine ont été formulés. Depuis lors, le domaine a connu des hauts et des bas : il y a eu quelques succès dans des domaines particuliers, mais son développement a surtout été entravé par un certain nombre de problèmes, parmi lesquels la puissance limitée des ordinateurs [16], le manque de données (informations globales sur le monde) [93] et le caractère intractable [93] (il existe de nombreux problèmes qui ne peuvent être résolus de manière optimale qu'en un temps exponentiel).

La situation a changé récemment avec l'émergence d'AlexNet [56] en 2012. Il est basé sur un réseau de neurones convolutif entraîné à l'aide de l'algorithme de rétropropagation [92]. Pourtant, l'utilisation des réseaux de neurones n'était pas une nouveauté [90, 65]. L'avancée est due à l'augmentation de la profondeur du réseau neuronal, tandis que l'entraînement d'un modèle aussi grand est devenu possible grâce à l'amélioration des capacités des GPU. Ainsi, une fois entraîné sur Imagenet [25] (la grande base de données d'images), le modèle fait preuve d'une grande précision dans les tâches de classification d'images, approchant la précision humaine.

Ainsi, la nouvelle force motrice de l'IA au cours des dernières années est le développement des réseaux neuronaux profonds (DNN). Depuis la percée d'AlexNet, les DNN sont devenus plus complexes et plus profonds : leurs graphes de calcul peuvent être des graphes acycliques dirigés (DAG) qui comprennent de plus en plus d'opérations (également appelées couches). Par exemple, AlexNet ne compte que 8 couches qui forment une chaîne, alors que ResNet [45], proposé en 2015, est représenté par une chaîne avec des connexions sautées composée de 152 couches ; tous deux ont environ 60 millions de paramètres, mais il y a une différence d'environ 10% dans leur précision (ResNet surpasse AlexNet). Par ailleurs, les modèles de transformers [97, 14] qui constituent aujourd'hui l'état de l'art en matière de traitement du langage naturel (NLP) peuvent atteindre jusqu'à 175B paramètres (*e.g.* GPT-3 [14]), et sont à la fois profonds et larges.

Des modèles aussi lourds atteignent les limites des machines sur lesquelles ils sont traités. Les problèmes de mémoire peuvent apparaître aussi bien lors de l'inférence que lors de l'entraînement. L'inférence et l'apprentissage sont deux étapes bien différentes lorsqu'on travaille avec des DNN. L'inférence correspond à l'exécution du DNN afin d'obtenir des prédictions et est souvent effectuée sur des appareils embarqués comme les smartphones, qui ont une capacité de mémoire et une puissance de calcul très faibles [19]. En revanche,

l'apprentissage est le processus itératif dont le but est de mettre à jour les paramètres du modèle (également appelés poids), afin que le modèle puisse réaliser des prédictions de qualité. Ce processus est encore plus coûteux en mémoire et en calcul, il est généralement effectué sur des grappes de machines et peut prendre des heures, voire des jours, pour être accompli [67]. Par conséquent, l'exploration de modèles plus profonds et plus grands crée une demande pour de nouveaux matériels et de nouveaux algorithmes qui prennent en compte les limitations de ressources [86].

Comme la recherche en IA se développe à un rythme effréné, de plus en plus de sociétés technologiques investissent dans le développement de nouveaux types de matériel, conçus spécifiquement pour les DNN. Les CPU, GPU et TPU conviennent à l'entraînement, tandis que les FPGA et ASIC sont préférables pour l'inférence sur les dispositifs embarqués [17]. Les GPU et TPU sont devenus le principal outil de travail en raison de leur grande efficacité dans les calculs parallèles, ce qui est très pratique pour effectuer de grandes opérations matricielles (ou tensorielles). En outre, les GPU les plus récents peuvent être jusqu'à 245 fois plus rapides que les CPU modernes [17]. Cependant, par rapport aux CPU, les GPU et les cœurs TPU ne disposent pas d'une grande mémoire. Beaucoup de clusters contiennent des GPU avec 16 Go de mémoire, les plus grands centres de données peuvent avoir des GPU NVIDIA V100 Tensor Core avec 32 Go de mémoire[1]. Récemment, NVIDIA a commencé à vendre de nouveaux GPU de la série A100 dont la mémoire peut être de 40 Go et 80 Go[2]. Malgré tous les efforts déployés pour fabriquer des GPU de grande capacité, ces derniers sont susceptibles d'échouer lors de l'apprentissage d'un modèle comportant des milliards de paramètres [86], qui nécessitent au moins 1 To de mémoire uniquement pour le stockage des poids. Un tel apprentissage n'est réalisable que de manière distribuée, en utilisant plus de 1000 GPU [86]. Les cœurs TPU sont encore moins à même de prendre en charge de grands modèles, car un cœur TPU ne dispose que de 16 Go de mémoire au maximum [109]. D'autre part, même si un modèle tient sur un seul GPU avec une grande capacité de mémoire (*e.g.* 80 GB), tout le monde ne peut pas se permettre d'acheter ces GPU [101] : le coût d'un GPU V100 est d'environ 7 500 euros et le coût d'un GPU A100 40 GB est d'environ 10 000 euros [3]. En outre, il a été démontré dans [44] que pour le matériel récemment produit, une nouvelle tendance se dessine : leur impact carbone de fabrication dépasse leur impact carbone opérationnel. Par exemple, alors que les centres de données de Facebook se tournent vers les énergies renouvelables, leurs activités liées aux investissements (capex signifiant dépenses d'investissement) représentent 82 % de la production de carbone, tandis que 42 % des émissions liées aux investissements proviennent de la fabrication du matériel et des infrastructures. Cela implique que davantage d'années de service sont nécessaires pour amortir le coût de production. En d'autres termes, même si l'argent n'est pas un goulot d'étranglement, il convient de reconsidérer le remplacement de son ancien GPU par un nouveau qui dispose de plus de mémoire, afin de soutenir les idées d'IA verte [95]. Tout ce qui précède implique que les techniques d'optimisation de la mémoire logicielles sont

[1] https://www.nvidia.com/en-us/data-center/v100/
[2] https://www.nvidia.com/en-us/data-center/a100/
[3] Les prix sont basés sur eBay.

essentielles.

Les dispositifs embarqués ont également une taille de mémoire limitée, qui est nettement inférieure à celle dont dispose un GPU d'une machine parallèle. En général, les GPU modernes utilisés dans les systèmes embarqués peuvent avoir plusieurs Go de mémoire, mais ce n'est pas toujours suffisant pour effectuer l'apprentissage en Edge. Bien que l'inférence soit une routine pour les nœuds Edge, l'apprentissage n'est pas encore devenu une pratique courante. Récemment, plusieurs articles [59, 63, 105, 70, 87, 69] ont fait valoir les avantages et les intérêts potentiels de procéder à un apprentissage directement sur les appareils Edge. Parmi ceux-ci, on peut citer le renforcement de la confidentialité et de la sécurité des informations [87, 69], la réduction de la charge de la bande passante [69, 105, 70], une meilleure évolutivité [70] et une meilleure adaptation du dispositif à son contexte d'utilisation [59]. Afin de permettre l'apprentissage direct sur les dispositifs embarqués, il est nécessaire d'ajuster les modèles et les algorithmes d'apprentissage pour que la mémoire et la puissance de calcul disponibles soient utilisées de la manière la plus efficace.

Pour concevoir des stratégies efficaces en termes de mémoire pour traiter les DNN, il est important de comprendre quelles sont les sources des problèmes de mémoire. Les paramètres du modèle, également appelés poids, doivent être constamment conservés en mémoire lors de l'inférence et de l'apprentissage. Il existe plusieurs techniques permettant de compresser les poids d'un modèle formé pour l'inférence comme la factorisation low-rank, la distillation des connaissances, la quantification et l'élagage [21]. Certaines de ces stratégies ont inspiré les architectures de réseaux neuronaux économes en mémoire, comme MobileNet et ShuffleNet, qui peuvent être facilement formés même sur des systèmes à mémoire limitée.

Toutefois, d'autres facteurs influent sur la quantité de mémoire consommée lors de la phase d'apprentissage. Pour comprendre la quantité de mémoire nécessaire pour effectuer une itération d'apprentissage, il est nécessaire d'analyser les dépendances de données qui se produisent au cours de l'exécution. Une itération consiste en deux passages sur le graphe de calcul, appelés propagation vers l'avant (*forward*) et vers l'arrière (*backward*). La propagation vers l'avant est le passage direct sur le graphe (du début à la fin), elle calcule les prédictions et est suivie par l'évaluation de la fonction *perte* qui indique à quel point les prédictions sont proches des vraies valeurs cibles. La propagation vers l'arrière est le passage inverse sur le graphe (de la fin au début) pendant lequel les gradients de la perte par rapport aux poids sont calculés et utilisés pour effectuer les mises à jour des poids. La combinaison de la propagation vers l'avant et vers l'arrière entraîne des dépendances de données complexes : les entrées d'une couche i utilisées pendant la propagation vers l'avant sont nécessaires à l'opération vers l'arrière correspondant à cette couche. Cela implique que toutes les données intermédiaires générées par les opérations avant (nous appellerons par la suite ces données des *activations*) sont nécessaires pour effectuer la rétropropagation. En outre, lors de la mise à jour du modèle à l'aide des gradients, certains optimiseurs (algorithmes responsables du calcul des mises à jour des poids) nécessitent de stocker des états supplémentaires de l'optimiseur [86], dont les tailles sont proportionnelles aux poids. Globalement, en plus des poids du modèle, la machine doit pendant l'apprentissage

également stocker toutes ses activations, les gradients des poids et les états de l'optimiseur, ce qui peut conduire à une explosion de la mémoire. Dans ce travail, notre objectif est d'étudier les stratégies de réduction de la mémoire qui rendent l'apprentissage possible dans la limite de la mémoire du matériel donné. En fonction de l'endroit où l'apprentissage est effectué (soit sur une seule ressource informatique, soit de manière distribuée) les besoins en mémoire peuvent être réduits en utilisant différentes techniques, inspirées des approches du calcul haute performance (HPC), de l'ordonnancement et même de la différenciation automatique (AD).

Apprentissage sur un seul noeud

Pendant l'apprentissage, les poids du modèle et les états de l'optimiseur doivent être conservés en mémoire (soit dans la mémoire principale, soit dans la mémoire du dispositif) tout le temps. Néanmoins, il n'est pas nécessaire que les activations soient présentes en permanence. Elles sont générées pendant la propagation vers l'avant et, par conséquent, elles peuvent être écartées et recalculées plus tard en réexécutant certaines étapes vers l'avant. Cette approche est connue sous le nom de rematérialisation ou "gradient checkpointing". Une autre façon d'améliorer l'utilisation des ressources est de décharger les données d'un GPU vers un CPU et de profiter ainsi d'un espace de stockage supplémentaire (les CPU ont généralement une capacité de mémoire beaucoup plus importante que les GPU). On peut décider de décharger les activations, les poids du modèle et les états de l'optimiseur, à condition qu'ils soient rapatriés préalablement à leur utilisation. La combinaison du déchargement et de la rematérialisation est prometteuse pour obtenir les meilleures performances.

Dans ce travail, nous étudions soigneusement les problèmes liés à la rematérialisation et au déchargement, et nous nous concentrons sur la réduction de la mémoire occupée par les activations, tandis que les poids et les états de l'optimiseur sont supposés être stockés en mémoire en permanence.

Rematérialisation

La rematérialisation consiste à sélectionner seulement quelques activations qui sont sauvegardées en mémoire et utilisées pour recalculer les autres. Elle permet d'explorer un compromis entre la mémoire et les calculs. C'est un problème classique de checkpointing formulé pour les chaînes adjointes étudiées en AD. Ce problème y est résolu par programmation dynamique. Comme le graphe de calcul des chaînes adjointes peut être vu comme une version simplifiée des dépendances de données des DNNs, les solutions d'AD peuvent être adaptées aux DNNs [20]. Cependant, les approches classiques en AD considèrent des chaînes homogènes (toutes les opérations ont les mêmes coûts de calcul et de mémoire), l'application directe de leurs techniques aux DNNs conduit à des performances sous-optimales. En revanche, les DNNs sont mieux modélisés par des chaînes hétérogènes, voire des structures générales de DAG (graphe orienté sans cycle).

Les travaux récents [31, 60, 61, 52, 54] ont essayé de prendre en compte des modèles plus réalistes, bien qu'aucun résultat d'optimalité général ne soit fourni.

Néanmoins, la méthode s'est avérée utile dans la pratique. Lors de l'entraînement de réseaux neuronaux très profonds sur des données énormes, ce qui est normalement irréalisable, cette approche permet de dépasser la limite de mémoire de l'unité de calcul. Les tailles d'activation sont proportionnelles à la taille de l'entrée (résolution de l'image, longueur de la séquence de texte, ...) et à la taille du lot (nombre d'échantillons utilisés pour une mise à jour du modèle). Ainsi, la rematérialisation peut être particulièrement utile dans le cas où l'on souhaite augmenter la taille d'entrée ou la profondeur du réseau neuronal ou lorsque la formation avec une taille de lot unitaire échoue [46]. Parfois, il peut également être bénéfique d'augmenter la taille du lot, mais dans la plupart des cas, cela entraîne une baisse du débit. Par contre, il y a beaucoup de cas où une grande taille de lot mène à une convergence plus rapide et meilleure [99]. Elle peut encore être combinée avec la technique d'accumulation de gradient, où l'on augmente artificiellement la taille du lot, en exécutant un nombre n d'itérations avec un lot plus petit sans mettre à jour les poids, mais en accumulant (en faisant la somme) les gradients des poids de différentes itérations ; où toutes les n iterations, on effectue la mise à jour avec le gradient obtenu avant de recommencer à nouveau. Ainsi, la rematérialisation et l'accumulation de gradient permettent d'augmenter la profondeur du modèle, la taille de l'entrée et la taille du lot tout en maintenant un débit raisonnable [100]. Enfin, d'un point de vue écologique, la rematérialisation peut avoir un impact carbone négligeable à condition que les calculs soient effectués avec de l'énergie renouvelable [44], en comparaison avec l'achat d'un nouveau GPU avec plus de mémoire.

Notre contribution Dans cette thèse, nous analysons le problème de la recherche des stratégies optimales de rematérialisation pour les DNNs. Afin d'aborder le cas plus général des DAGs, nous considérons d'abord, dans le chapitre 1, les solutions pour les structures multi-chaînes homogènes. Les graphes multi-chaînes sont représentés par plusieurs chaînes de différentes longueurs qui sont rassemblées à la fin par la fonction de perte. Ces graphes sont similaires aux graphes des réseaux neuronaux siamois et aux réseaux d'intégration cross-modaux. Dans ce chapitre, nous étendons la programmation dynamique classique pour les chaînes adjointes afin de traiter les graphes multi-chaînes plus généraux et nous prouvons son optimalité.

Les graphes avec des coûts hétérogènes présentent un intérêt particulier. Dans le chapitre 2, nous analysons les chaînes hétérogènes, qui, bien que ne couvrant pas le cas des DAGs, correspondent à de nombreux DNNs pratiques. Nous fournissons une solution optimale ainsi qu'une heuristique moins coûteuse, qui sont toutes deux basées sur la programmation dynamique. Les expériences confirment également la meilleure performance de ces nouveaux algorithmes par rapport à l'état de l'art [20, 52]. Sur cette base, nous avons également conçu un outil ROTOR[4] compatible avec PyTorch qui réussit à réduire de manière significative la quantité de mémoire au prix d'une augmentation

[4] https://gitlab.inria.fr/hiepacs/rotor

marginale du temps d'exécution.

Déchargement (Offloading)

Le déchargement est l'autre choix populaire pour garder moins d'activations dans la mémoire du GPU. Par rapport à la rematérialisation, il n'y a pas de recalculs, donc le chemin critique est le même que dans l'exécution classique. Cependant, les communications pour décharger (respectivement précharger) les activations vers (respectivement depuis) la mémoire du CPU peuvent induire des temps morts. Par exemple, la première approche naïve [88] consistant à décharger toutes les activations et à se synchroniser après chaque opération peut entraîner un énorme délai. Afin de diminuer les temps morts, il faut choisir avec soin quelles activations décharger et quand, tout en évitant les synchronisations inutiles. Différentes heuristiques ont été proposées pour résoudre ce problème [7, 114, 64], cependant aucune d'entre elles ne propose une analyse de son optimalité. Il existe également une possibilité de décharger les poids des modèles et les états des optimiseurs [85] en plus. Dans toutes ces approches, le choix des données à transférer est déterminé par les propriétés du réseau neuronal.

Comme la rematérialisation, le déchargement peut être utilisé pour augmenter efficacement la profondeur du réseau neuronal et la taille des entrées. L'un des avantages considérables du déchargement est la possibilité de produire un surcoût nul si les transferts de données sont bien planifiés de sorte qu'ils recouvrent entièrement les opérations de calcul. Par conséquent, le déchargement permet également de traiter des lots plus importants. Cependant, les performances de cette technique dépendent fortement de la bande passante du lien de communication. Si la bande passante est faible, le déchargement perd de son attrait et ne peut guère concurrencer la rematérialisation. D'autre part, la combinaison des deux techniques devrait être plus puissante que l'une ou l'autre, car elle est suffisamment flexible pour s'adapter à tous les paramètres.

Notre contribution Dans le chapitre 3, nous introduisons formellement le problème du déchargement pour les chaînes hétérogènes. A notre connaissance, nous sommes les premiers à formuler la minimisation du surcoût comme un problème d'optimisation, dépendant du choix des activations à décharger et de l'ordonnancement des transferts de données. En général, ce problème est NP-complet au sens fort, mais nous proposons des relaxations de ce problème qui peuvent être résolues de manière optimale en temps polynomial et pseudo-polynomial et dont les solutions sont efficaces en pratique. Sur la base d'hypothèses réalistes, nos nouveaux algorithmes montrent une supériorité sur les heuristiques naïves précédemment considérées.

Ensuite, nous intégrons la rematérialisation au déchargement dans le chapitre 4. Plus précisément, nous montrons que sous un certain ensemble d'hypothèses, il est possible de trouver l'ordonnancement optimal pour la combinaison, en utilisant un nouveau programme dynamique. Nous ajoutons ce programme dynamique dans ROTOR et les nouvelles expériences démontrent que les deux approches profitent de cette union, atteignant des performances strictement meilleures dans la plupart des cas.

Apprentissage sur plusieurs nœuds

Un entraînement distribué est très courant en raison des exigences de calcul élevées des DNN modernes. Tous les tenseurs (poids, états de l'optimiseur, activations) peuvent être traités séparément et envoyés à différentes machines, atténuant ainsi la charge par processeur.

Il existe plusieurs façons de diviser et de répartir le travail sur plusieurs processeurs : par données, par tenseurs et par couches. Le choix le plus populaire est le parallélisme de données [23, 116], qui consiste à répliquer le modèle sur plusieurs ressources puis à traiter plusieurs mini-lots en parallèle, en communiquant les mises à jour uniquement à la fin de chaque itération. Cela permet d'augmenter considérablement la taille totale des lots, ce qui favorise la convergence [99]. Par conséquent, elle permet un bon passage à l'échelle, malgré la synchronisation des poids qui peut être coûteuse avec des modèles lourds. Une autre façon de répartir les données est d'utiliser la décomposition spatiale [26]. Par exemple, lors du traitement d'images avec des réseaux de neurones convolutifs (CNNs), il est possible de diviser chaque image et chaque activation en plusieurs zones, de sorte que chaque GPU traite sa propre zone, en ne communiquant aux autres GPUs que des informations de bordure appelées "halo", ce qui suffit à préserver la validité des calculs. Cette approche est pratique lorsqu'une taille de lot unitaire ne rentre pas dans la mémoire d'un GPU.

Les approches tensorielles connues sous le nom de Tensor Slicing parallélisent l'exécution du noyau des couches dans les réseaux neuronaux. Par exemple, pour les couches entièrement connectées qui effectuent une multiplication matrice-matrice (une matrice est l'entrée, l'autre est la matrice de poids), la matrice de poids est répartie sur plusieurs ressources de calcul, l'opération est effectuée en parallèle et, à la fin, les sorties sont envoyées à toutes les ressources. Pour les couches convolutionnelles qui effectuent des convolutions en utilisant différents filtres sur l'image constituée de plusieurs canaux, il est possible de paralléliser à travers différentes dimensions : hauteur, largeur, canaux et filtres [27].

Une autre méthode consiste à utiliser le parallélisme de modèle (MP) qui distribue la charge par couches. Dans ce contexte, chaque processeur est affecté à une partie du graphe (un sous-ensemble de couches) et ne conserve que les poids et les activations liés à cette partie, tandis que les calculs sont effectués en séquence. Dans sa version originale, le MP n'accélère pas l'exécution, mais il nécessite moins de mémoire par noeud. Récemment, il a été suggéré de combiner cette méthode avec le Pipelining pour obtenir une meilleure accélération dans [50, 33, 78]. Parmi elles, PipeDream [78] offre le meilleur débit et sur cette base d'autres méthodes ont été proposées [113, 43, 106, 79]. PipeDream trouve un bon équilibrage de la charge à l'aide de la programmation dynamique, montrant qu'avec un équilibrage parfait de la charge et des communications négligeables, il est possible d'éviter les temps morts pendant tout l'apprentissage. Cependant, comme dans le pipelining on injecte plusieurs mini-batchs en même temps, chaque processeur doit stocker plusieurs copies des poids et des activations pour assurer la validité de l'entraînement, ce qui annule presque les avantages offerts par la distribution.

Globalement, les différents types de parallélisme présentent des avantages et des

inconvénients [110]. Le parallélisme des données présente le meilleur passage à l'échelle puisque chaque processeur prend en charge des mini-lots de taille égale, atteignant ainsi le meilleur équilibre de charge. Cependant, ses performances sont entravées par de grands mouvements de données (une opération de réduction collective de tous les paramètres du modèle à la fin de chaque itération). Le parallélisme asynchrone des données [115] permet la meilleure utilisation des ressources en éliminant la synchronisation globale pour mettre à jour les poids à la fin de chaque itération, mais il souffre d'une moins bonne convergence à cause de la stagnation des poids [18]. Le parallélisme des données et la décomposition spatiale aident à distribuer les données (entrée et activations), néanmoins, ces deux méthodes ne peuvent pas pallier les problèmes de mémoire liés au modèle : chaque noeud fonctionne de manière indépendante, ayant sa propre copie des poids. Le découpage tensoriel permet de réduire la mémoire occupée par un modèle pour chaque processeur, mais comme chaque couche requiert une entrée complète (même si la couche elle-même est distribuée), il implique beaucoup de communications et de synchronisations après chaque couche et ne permet pas de réduire la mémoire requise pour les activations. Le parallélisme de modèle, similaire au découpage tensoriel, peut distribuer les poids du modèle, mais aussi les activations sur différentes ressources. Son évolutivité dépend du fait que les mises à jour sont effectuées de manière synchrone ou asynchrone. Le GPipe synchrone [50] sous-utilise la puissance des GPU, les laissant inactifs une bonne partie du temps. Le PipeDream [78] asynchrone peut atteindre un passage à l'échelle similaire à celui du parallélisme des données, à condition qu'il atteigne un équilibrage de charge parfait : il ne communique que les activations entre les couches placées sur des ressources différentes, qui peuvent être entièrement recouvertes par des calculs. En revanche, PipeDream comme le parallélisme de données asynchrone introduit une stagnation des poids qui affecte négativement la convergence. De plus, comme il a été mentionné précédemment, plusieurs versions de poids et d'activations doivent être conservées sur le GPU pendant un tel apprentissage, ce qui rend les besoins en mémoire comparables à ceux du parallélisme de données. Différents travaux ont également exploré diverses combinaisons de parallélismes : parallélisme de données avec parallélisme de modèles [78], parallélisme de données avec découpage tensoriel [55] et les trois types [24]. Les types de parallélisme hybrides réunissent les points forts de chaque méthode, mais trouver un bon équilibre entre les différentes approches est une tâche difficile.

Notre contribution Dans notre travail, nous nous concentrons sur le parallélisme de modèles pipeliné, car il est le plus prometteur en termes de minimisation de l'utilisation de la mémoire. Nous considérons les inconvénients de PipeDream dans le chapitre 5. PipeDream préconise des solutions contiguës qui n'allouent les couches que de manière séquentielle aux GPU (toutes les couches d'un GPU sont voisines) et utilise un ordonnancement simpliste. Nous évaluons la qualité de telles solutions et montrons qu'il y a beaucoup de marge d'amélioration. Dans le chapitre 6, nous étudions une formulation en programmation linéaire en nombres entiers (ILP) qui permet de résoudre le problème de la répartition de la charge et de la planification du pipelining. Contrairement à PipeDream, cette méthode réussit à obtenir des allocations non contiguës avec le meilleur équilibrage

de charge prenant en compte les limitations de mémoire des GPU et, en même temps, elle planifie les opérations de manière optimale.

Même si l'ILP est capable de résoudre un problème difficile, son temps d'exécution peut être extrêmement long. Ainsi, dans la pratique, nous avons besoin d'une heuristique appropriée qui peut utiliser les points forts des allocations non contiguës, et qui est en même temps facile à calculer. Nous proposons l'outil MadPipe, basé sur la programmation dynamique, qui suppose qu'il existe un processeur qui héberge des couches non voisines, tandis que les autres processeurs ne prennent que les couches connexes. Les opérations sont ordonnancées avec l'ILP du chapitre 6. Nous fournissons la description complète de cette heuristique dans le chapitre 7. Par des expériences, nous démontrons que MadPipe apporte une amélioration significative par rapport à PipeDream, ce qui confirme l'importance des allocations non contiguës.

Historique

Contexte

Cette thèse de doctorat fait partie du projet Inria IPL (plus tard DEFI) qui réunit des chercheurs dans les domaines du HPC, du Big Data et de l'IA. Ce projet permet aux personnes issues de ces différents domaines de partager leurs connaissances et leur expertise pour trouver de nouvelles idées à leurs intersections. Les collaborations nouvellement créées visent à pousser plus loin les progrès dans tous les domaines susmentionnés.

En particulier, l'un des objectifs initiaux de ce travail était de lancer une collaboration entre les équipes de HiePACS (HPC) et de Zenith (Big Data et IA) afin d'entraîner Pl@ntNet, un projet de science citoyenne pour l'identification automatique des plantes à partir de photographies, basé sur l'apprentissage automatique [4]. Au cours de plusieurs visites mutuelles entre les membres des équipes de recherche HiePACS et Zenith, plusieurs problèmes ont été identifiés en matière d'apprentissage. Cependant, la mémoire a été identifiée comme le principal goulot d'étranglement empêchant Pl@ntNet de passer à des modèles plus grands et de considérer des images plus grandes et un ensemble plus riche d'espèces. Par conséquent, nous avons décidé de nous attaquer à ce problème particulier.

Afin de résoudre les problèmes liés à la mémoire, la structure du graphe d'entraînement des réseaux de neurones et le flux de travail doivent être soigneusement analysés et modélisés. Pl@ntNet s'appuie sur PyTorch pour effectuer la formation, donc son mode de fonctionnement doit également être pris en compte. Suite à nos discussions, une collection de méthodes pour économiser la mémoire a été proposée dans ce travail. Finalement, Pl@ntNet a réussi à résoudre son problème initial de mémoire en considérant des clusters avec des GPUs qui ont une plus grande capacité de mémoire. Malgré cela, nos avancées permettent de faire face à de nouveaux défis : l'entraînement de réseaux neuronaux plus grands (*e.g.* dans le contexte de Pl@ntNet, c'est nécessaire pour une distillation efficace des connaissances), l'entraînement avec des données d'entrée plus grandes (*e.g.*

des images à plus haute résolution) et l'utilisation d'une taille de lot plus grande pour une convergence plus rapide. Nos méthodes sont utiles au-delà des applications de Pl@ntNet : afin d'entraîner de grands modèles de langage tels que GPT-3, la rematérialisation, l'offloading et le parallélisme des modèles doivent être combinés afin de réduire les besoins en mémoire d'apprentissage. Enfin, des techniques de gestion efficace de la mémoire permettent de mieux exploiter le matériel disponible, de rendre les ressources de calculs plus productivess, tout en prolongeant leur durée d'utilisation et en réduisant ainsi l'impact carbone.

Contributions pratiques et théoriques

- Le chapitre 2 et la partie II contiennent des contributions à la fois théoriques et pratiques : des solutions optimales qui prennent en compte les spécificités des réseaux neuronaux séquentiels et des frameworks d'apprentissage sont proposées et sont ensuite intégrées dans l'outil ROTOR [5] qui est entièrement fonctionnel, facile à utiliser et compatible avec PyTorch. Ces chapitres comprennent les résultats des expériences réelles menées avec ROTOR.
- Le chapitre 1 et la partie III sont plus théoriques par nature : ils fournissent une nouvelle modélisation originale qui fournit des indications précieuses sur les problèmes sous-jacents ainsi que des algorithmes optimaux, mais ils n'ont pas encore été mis en œuvre dans un framework d'apprentissage. Les résultats présentés dans ces chapitres reposent sur des simulations.

Publications

Certaines des contributions présentées dans ce travail ont été publiées : pour le chapitre 1 dans Philosophical Transactions of the Royal Society [10], pour le chapitre 3 dans Proceedings of European Conference on Parallel Processing (Euro-Par) 2020 [8] et pour le chapitre 5 dans European Conference on Parallel Processing 2021 [9]. Les résultats du chapitre 4 sont acceptés pour publication dans la conférence Neural Information Processing Systems (NeurIPS) 2021.

En dehors de cela, il existe d'autres travaux réalisés pendant ce doctorat, mais qui ne sont pas inclus dans le manuscrit par souci de concision et de cohérence. Parmi eux, il y a un travail conjoint en collaboration avec l'Imperial College London et l'Argonne National Laboratory qui considère les avantages et les défis de l'apprentissage sur l'Edge publié dans les Proceedings du Workshop on Parallel AI and Systems for the Edge 2019 [59]. Un autre travail a été effectué en collaboration avec l'Université du Colorado et l'Université Northeastern pour estimer la limite inférieure du temps d'exécution pour l'ordonnancement de la factorisation de Cholesky en tuiles, et publié dans Proceedings of European Conference on Parallel Processing 2020 [11].

[5] https://gitlab.inria.fr/hiepacs/rotor

Introduction

Artificial Intelligence (AI) is an uprising field that helps to solve numerous complex problems like image classification, text generation, translation... Its birth dates from 1956 when the Darthmouth Workshop took place and where its name and the first objectives of the field were formulated. Since then the field encountered many ups and downs: there were some successes in particular domains but mostly the further development was hindered by a number of problems among which are the limited computer power [16], lack of data (global information about the world) [93] and intractability [93] (there are a lot of problems that can only be solved optimally in exponential time).

The situation has changed recently with the emergence of AlexNet [56] in 2012. It is based on a convolutional neural network trained using backpropagation algorithm [92]. Though, using neural networks was not a novelty [90, 65]. The advance was due to the increased depth of the neural network, while training such a big model became possible thanks to improved GPU capability. Thus, when trained on Imagenet [25] (the large dataset of images), the model exhibits high precision on image classification tasks, approaching human-based precision.

Thus, the new driving force of AI in the recent years is the development of Deep Neural Networks (DNNs). Since the breakthrough of AlexNet, DNNs have become more complex and deeper: their computational graphs can be general Directed Acyclic Graphs (DAGs) that comprise more and more operations (also called layers). For example, AlexNet has only 8 layers arranged in a chain, while ResNet [45] proposed in 2015 is represented by a chain with skip-connections consisting of 152 layers; both have about 60M parameters, but there is about a 10% difference in their accuracy (ResNet outperfoms AlexNet). Furthermore, the transformer models [97, 14] that are now the state-of-the-art in Natural Language Processing (NLP) may reach up to 175B parameters (*e.g.* GPT-3 [14]), being deep and wide at the same time.

Such heavy models reach the limits of machines on which they are processed. The memory problems may appear during both *inference* and *training*. Inference and Training are two distinguished stages when working with DNNs. On the one hand, inference is the execution of DNNs in order to obtain predictions and is often performed on embedded devices like smartphones, which have very low memory capacity and computational power [19]. On the other hand, training is the iterative process whose goal is to update the model parameters (also called weights), so that the model can perform qualitative predictions. This process is even more memory and computationally expensive, it is usually performed on clusters of machines and may take hours or sometimes days to

complete [67]. Therefore, exploring further deeper and larger models creates the demand for new hardware and new algorithms that take into account resource limitations [86].

As research in AI grows at a tremendous pace, more and more technology companies invest in developing new types of hardware, designed specifically for the DNNs. CPUs, GPUs and TPUs are suitable for training, while FPGA and ASIC are preferable for inference on embedded devices [17]. GPUs and TPUs become now the main workhorse due to its high efficiency in parallel computations, which is very handy when performing large matrix (or tensor) operations. Furthermore, the most up-to-date GPU can be up to 245 times faster than the modern CPU [17]. However, in comparison to CPUs, GPUs and TPU cores do not have big memory. A lot of clusters contain GPUs with 16 GB of memory, the biggest data centres may have GPUs NVIDIA V100 Tensor Core with 32 GB of memory[6]. Recently, NVIDIA has started to sell new GPUs from A100 series whose memory can be 40 GB and 80 GB[7]. Despite all the efforts to make GPUs with large capacity, they still may fail when training a model with trillion parameters [86], which require at least 1 TB of memory just for storing the weights. Such training is only feasible in a distributed manner, using over 1000 GPUs [86]. TPU cores are even less prone to support large models, as one TPU core has at most 16 GB of memory [109]. Even if a model fits onto a single GPU with high memory capacity (*e.g.* 80 GB), not everybody can afford to buy those GPUs [101]: the cost of one GPU V100 is about 7 500 euros and the cost of GPU A100 40 GB is around 10 000 euros[8]. Moreover, it has been shown in [44] that for the recently produced hardware there is a new tendency: their manufacturing carbon impact exceeds their operational carbon impact. For example, as Facebook's data centers turn to renewable energy, their capex-related activities (capex standing for capital expense) account for 82% of carbon output, while 42% of capex emissions comes from hardware and infrastructure manufacturing. This implies that more years of service are required to amortize the production cost. In other words, even if money is not a bottleneck, one should reconsider replacing his/her old GPU by the new one that has more memory, in order to support green AI ideas [95]. All mentioned above imply that memory optimization techniques based on software are more essential.

The embedded devices also have a limited memory size, which is significantly smaller than the one that a GPU from a computer cluster has. Typically, the modern GPUs used in embedded systems may have several GB of memory, still it may not be enough to perform training on the Edge. Even though, the inference is a routine for Edge nodes, but training has not become a common practice yet. Recently, several papers [59, 63, 105, 70, 87, 69] have advocated the potential benefits and interests of doing training directly on the devices. Among them are the increased privacy and information security [87, 69], reduced load of the bandwidth [69, 105, 70], better scalability [70] and better adaptation of the device to its context of usage [59]. In order to enable direct learning on the embedded devices, it is required to adjust the models and training algorithms so that the available memory and computational power are used in the most efficient way.

[6] https://www.nvidia.com/en-us/data-center/v100/

[7] https://www.nvidia.com/en-us/data-center/a100/

[8] Prices are based on eBay

If we want to design memory efficient strategies to process DNNs, then it is important to understand what are the sources of memory problems. Model parameters also called *weights* should be kept constantly in the memory when performing both inference and training. There are several techniques that are able to compress weights of a trained model for inference: low-rank factorization, knowledge distillation, quantization and pruning [21]. Some of these strategies have inspired the memory efficient architectures of neural networks such as MobileNet, ShuffleNet that can be easily trained even on memory constrained systems.

However, other factors also affect the level of memory consumption during the training. To understand how much memory is needed to perform one iteration of the training, it is necessary to analyze the data dependencies that occur in the runtime. One iteration consists of two passes over the computational graph, called *forward* and *backward propagations*. Forward propagation is the direct pass over the graph (from the beginning to the end), it computes the predictions and is followed by the evaluation of the *loss* function that shows how close the predictions are to the true target values. Backward propagation is the reverse pass over the graph (from the end to the beginning) during which the gradients of the loss with respect to the weights are calculated and used to perform weight updates. The combination of forward and backward propagation results in complex data dependencies: the inputs of some layer i used during the forward pass are needed by the backward operation corresponding to this layer. This implies that all intermediate data generated by the forward operations (further we refer to these data as *activations*) is needed to complete backpropagation. Besides, when performing updates of the model using gradients, depending on the optimizer (the algorithm responsible for computing updates of the weights), one might need to store additionally optimizer states [86], whose sizes are proportional to weights. Overall, apart from the model weights, the machine during training should also store all its activations, gradients of weights and optimizer states, which may lead to memory explosion. In this work, our target is to study the memory saving strategies making training feasible under the memory limit of the given hardware. Depending on where the training is performed (either on a single computational resource or in the distributed way) memory needs can be reduced using different techniques, inspired by the approaches from High Performance Computing (HPC), Scheduling and even Automatic Differentiation (AD) domains.

Training on a Single Device

During the training, model weights and optimizer states should be kept in memory (either in main memory or in device memory) the entire time. Nevertheless, activations do not have to be there constantly. They are generated during the forward propagation and, therefore, they could be discarded and recomputed later by rerunning some forward steps again. This approach is known as Rematerialization or Gradient Checkpointing. Other way to improve resource utilization is to offload the data from a GPU to a CPU and like this profit from an additional storage space (CPUs have usually much larger memory capacity

than GPUs). One can decide to offload the activations, model weights and optimizer states, on condition that they are prefetched back once they are needed. The combination of both Offloading and Rematerialization is promising to obtain the best performance.

In this work, we study carefully the problems related to Rematerialization and Offloading, and we focus on reducing the memory occupied by activations, while weights and optimizer states are assumed to be stored in memory all the time.

Rematerialization

Rematerialization consists in selecting only a few activations that are saved into memory and used for recomputing the others. It helps to explore a tradeoff between memory and computations. It is a classical problem of Checkpointing formulated for adjoint chains studied in AD. This problem is solved there with a dynamic programming. As the computational graph of adjoint chains can be seen as a simplified version of DNN data dependencies, AD solutions can be adapted for DNNs [20]. However, classical approaches in AD consider homogeneous chains (all operations have the same computational and memory costs), the direct application of their techniques to DNNs results in sub-optimal performance. In contrast, the DNNs are better approximated with heterogeneous chains and even general DAG structures. The recent works [31, 60, 61, 52, 54] tried to take into account more realistic models, though no general optimality results are provided.

Nevertheless, the method proved to be useful in practice. When training very deep neural networks on huge data, which normally is unfeasible, this approach helps to surpass memory limit of the computational unit. The activation sizes are proportional to the input size (*e.g.* resolution of the image, length of the text sequence, ...) and batch-size (number of samples that are used for one update of the model). Thus, Rematerialization can be especially helpful in case one wants to increase the input size or the depth of the neural network or when the training with batch-size of one fails [46]. Sometimes, it can be beneficial to increase batch-size too, but in most cases it leads to the worse throughput. Whereas, there are a lot of cases when a large batch-size leads to a faster and better convergence [99]. It still can be combined with gradient accumulation technique, where one artificially increases batch-size, by running a number n of iterations with a smaller batch without updating the weights, but accumulating (by summing) the gradients of the weights from different iterations and after each n-th iteration performing the update with the obtained gradient and then restarting again. Like this, Rematerialization and Gradient Accumulation together help to increase depth of the model, input size and batch size while sustaining the reasonable throughput [100]. Finally, from ecological standpoint, Rematerialization may have a negligible carbon impact provided that the computations are done with renewable energy [44], in comparison with buying a new GPU with more memory.

Our Contribution In this thesis, we analyse the problem of finding the optimal rematerialization strategies for DNNs. In order to approach more general DAG case, we first consider the solutions for homogeneous multi-chain structures in Chapter 1.

Multi-chain graphs are represented by several chains of different lengths that are gathered at the end by the loss function. These graphs are similar to the graphs of Siamese Neural Networks and Cross Modal embedding networks. In this chapter, we extend classical dynamic programming for adjoint chains to deal with the more general multi-chain graphs and we prove its optimality.

Of particular interest are the graphs with heterogeneous costs. In Chapter 2, we analyze heterogeneous chains, which despite not covering the DAG case, correspond to many practical DNNs. We provide an optimal solution and also a cheaper heuristic, which are both based on dynamic programming. The experiments also confirm the better performance of these new algorithms with respect to the state-of-the-art [20, 52]. Based on it, we also designed a tool ROTOR[9] compatible with PyTorch that successfully reduce a significant amount of memory at the price of a marginal rise in the running time.

Offloading

Offloading is the other popular choice to keep fewer activations in the device memory. In comparison with Rematerialization, there are no recomputations, thus the critical path is the same as in the classical execution. However, the communications to offload (resp. prefetch) activations to (resp. from) CPU memory can induce some idle time. For example, the first naive approach [88] to offload all activations and synchronize after each operation may suffer from a huge delay. In order to diminish the idle times, one should carefully choose which activations to offload and when, while avoiding excessive synchronizations. Different heuristics were proposed to tackle this problem [7, 114, 64], however none of them provided an analysis of their optimality. There is also a possibility to offload model weights and optimizer states [85] additionally. In these approaches, the choice of which data to transfer is determined by the properties of the neural network.

Like Rematerialization, Offloading can be used to efficiently increase the depth of the neural network and input size. One considerable advantage of Offloading is the potential to produce zero overhead if data transfers are well scheduled so that they overlap entirely the compute operations. Therefore, Offloading also helps to process larger batches. However, the performance of this technique strongly depends on the bandwidth of the communication link. If bandwidth is small, than Offloading loses its attractiveness as it hardly competes with Rematerialization. Alternatively, the combination of both techniques is expected to be more powerful than either of them, being flexible enough to adjust to any settings.

Our Contribution In Chapter 3, we formally introduce the problem of Offloading for heterogeneous chains. To the best of our knowledge, we are the first to formulate overhead minimization as an optimization problem, depending on the choice of activations to be offloaded and the schedule of data transfers. In general, this problem is NP-complete in the strong sense, but we propose relaxations of this problem that can be solved optimally

[9] https://gitlab.inria.fr/hiepacs/rotor

in polynomial and pseudo-polynomial times and whose solutions are efficient in practice. Based on realistic assumptions, our new algorithms show superiority over the previously considered naive heuristics.

After, we integrate Rematerialization with Offloading in Chapter 4. More specifically, we show that under a certain set of assumptions, it is possible to find the optimal schedule for the combination, using a new dynamic program. We add this dynamic program into ROTOR and the new experiments demonstrate that both approaches profit from this union, achieving strictly better performance in most cases.

Training on Multiple Nodes

Distributed training is very common due to high computational requirements of the state-of-the-art DNNs. All tensors (weights, optimizer states, activations) may be treated separately and sent to different machines, mitigating the load per processor.

There are various ways to split and distribute the work onto several processors: data-wise, tensor-wise and layer-wise. The most popular choice is Data Parallelism [23, 116], which consists in replicating the model on multiple resources and then processing several mini-batches in parallel, communicating updates only at the end of each iteration. It helps to substantially expand the total batch-size, which helps the convergence [99]. Therefore, it achieves good scalability, despite the weight synchronization that can be costly with heavy models. Another way to spread the data is to use Spatial Decomposition [26]. For example, when processing images with Convolutional Neural Networks (CNNs), it is possible to divide each image and each activation into several areas and each GPU processes its own area, while communicating only border information called halo to other GPUs, which is enough to preserve the validity of calculations. This approach is practical when an input with batch size of one does not fit into the memory of a GPU.

Tensor-wise approaches known as Tensor Slicing parallelize the kernel execution of layers in neural networks. For example, for fully-connected layers that perform matrix-matrix multiplication (one matrix is input, the other one is weight matrix), the weight matrix is distributed across several computing resources, the operation is performed in parallel and at the end the outputs are broadcast to all resources. For convolutional layers that perform convolutions using different filters on the image consisting from several channels it is possible to parallelize across different dimensions: height, width, channels and filters [27].

Another way is to use Model Parallelism (MP) that distributes the load in a layer-wise manner. In this context, each processor is assigned to a part of the graph (a subset of layers) and keeps only weights and activations related to this part, while calculations are performed in sequence. In its original form, MP does not speed up the execution, but it requires less memory per worker. Recently, it has been suggested combining this method with Pipelining to achieve some acceleration in [50, 33, 78]. Among them PipeDream [78] offers the best throughput and based on it other methods have been proposed [113, 43,

106, 79]. PipeDream finds a good load balancing with the help of dynamic programming, showing that with the perfect load balancing and negligible communications it is possible to avoid idle times throughout the entire training. Still, as in pipelining one injects several mini-batches at the same time, each processor needs to store several copies of the weights and activations to ensure the validity of the training, which almost cancels the benefits of distributing them in the first place.

Overall, different types of parallelism have both pros and cons [110]. Data parallelism has the best scalability as each processor takes care of mini-batches of equal size, reaching the best load balance. However, its performance is hindered by large data movements (a collective reduction operation for all model parameters at the end of each iteration). Asynchronous Data Parallelism [115] allows the best resource utilization when cancelling the global synchronization to update the weights at the end of each iteration, but it suffers from worse convergence because of weight staleness [18]. Data Parallelism and Spatial Decomposition help to distribute data (input and activations), nevertheless, both cannot mitigate memory issues related to model: each worker operates independently, having its own copy of weights. Tensor Slicing helps to reduce the memory occupied by a model per processor, however, as every layer requires an entire input (even if the layer itself is distributed), it involves a lot of communications and synchronizations after each layer and it does not help to reduce memory required for the activations. Model Parallelism, similarly to Tensor Slicing, can distribute the model weights, but also activations across different resources. Its scalability depends if the updates are done synchronously or asynchronously. Synchronous GPipe [50] underuse the GPU power, leaving them idle a considerable amount of time. Asynchronous PipeDream [78] may achieve the similar scalability as Data Parallelism provided that it reaches the perfect load balancing: it communicates only activations between layers placed on different resources, which can be entirely overlapped with computations. In contrast, PipeDream as Asynchronous Data Parallelism introduces the weight staleness that affects negatively the convergence. In addition, as it is mentioned earlier, several versions of weights and activations should be kept on the device during such training, making memory needs comparable to those of Data Parallelism. Different works explored also various combinations of parallelisms: Data Parallelism with Model Parallelism [78], Data Parallelism with Tensor Slicing [55] and all three types [24]. Hybrid types of parallelism unite the strong points of each method, however finding a good balance between different approaches is a difficult task.

Our Contribution In our work, we concentrate on Pipelined Model Parallelism, as it is the most promising in terms of minimizing memory usage. We address the downsides of PipeDream in Chapter 5. PipeDream suggests the contiguous solutions that only allocate layers sequentially to GPUs (all layers on one GPU are neighboring) and uses a very simplistic schedule. We estimate the quality of such solutions and show that there is plenty of room for improvement. In Chapter 6, we design the Integer Linear Programming (ILP) that can solve the problem of finding a good load balancing and schedule for the pipelining. Unlike PipeDream, this method succeeds in obtaining the non-contiguous allocations with the best load balancing that takes into account memory limitations of GPUs and, at the

same time, it schedules the operations optimally.

Even though the ILP is able to solve a difficult problem, its completion time can be extremely large. Thus, in practice, we need a proper heuristic that can use the strong points of non-contiguous allocations, and at the same time is easy to compute. We propose a tool MadPipe based on dynamic programming, which assumes that there is one processor that accommodate non-neighboring layers, whereas other processors take only connected layers. The operations are scheduled with the ILP from Chapter 6. We provide the full description of this heuristic in Chapter 7. Through experiments, we demonstrate that MadPipe brings the significant improvement over PipeDream, which confirms the importance of non-contiguous allocations.

Background

Context

This PhD thesis is a part of Inria project IPL (later DEFI) that unites researchers from the fields of HPC, Big Data and AI. This project allows people from these different domains to share their knowledge and expertise to find new ideas at the intersection. The newly created collaborations aim at pushing further the progress in all aforementioned fields.

Particularly, one initial goal of this work was to launch a collaboration between HiePACS (HPC) and Zenith (Big Data and AI) teams in order to train Pl@ntNet that is a citizen science project for automatic plant identification through photographs based on machine learning [4]. During several mutual visits between HiePACS and Zenith research members, several problems were established with respect to training. However, memory was identified as the main bottleneck preventing Pl@ntNet from moving to bigger models and considering larger images and a richer set of species. Therefore, we have decided to tackle this particular problem.

In order to solve issues related to memory, the structure of neural networks training graph and workflow should be carefully analyzed and modelled. Pl@ntNet relies on PyTorch to perform training, thus its way of functioning has to be taken into account as well. As a result of our discussions, a collection of memory saving methods are proposed in this work. Eventually, Pl@ntNet has managed to solve its original memory problem by considering clusters with GPUs that have larger memory capacity. Nevertheless, our advances help to deal with new challenges: training larger neural networks (*e.g.* in the context of Pl@ntNet it is needed for efficient knowledge distillation), training with larger input data (*e.g.* images with higher resolution) and using larger batch-size for a faster convergence. Our methods are useful beyond Pl@ntNet applications: in order to train large language models such as GPT-3, Rematerialization, Offloading and Model Parallelism are combined together in an effort to cut training memory requirements. Finally, efficient memory techniques help to better exploit the available hardware, make computations more productive, while extending their utilization period and thus reducing the carbon impact.

Practical and Theoretical Contribution

- Chapter 2 and Part II contain both theoretical and practical contributions: optimal solutions that encompass specificities of both sequential neural networks and learning frameworks are proposed and further integrated in ROTOR tool [10], which is fully-functional, easy-to-use and compatible with PyTorch. These chapters include the results of the actual experiments with ROTOR.
- Chapter 1 and Part III are more theoretical in nature: they provide new original modelling with valuable insights in underlying problems and optimal algorithms, but they have not been implemented yet in a learning framework. Results presented in these chapters rely on simulations.

Publications

Some of the contributions presented in this work have been published: Chapter 1 in Philosophical Transactions of the Royal Society [10], Chapter 3 in Proceedings of European Conference on Parallel Processing 2020 [8] and Chapter 5 in European Conference on Parallel Processing 2021 [9]. The results of Chapter 4 are accepted for publication in Neural Information Processing Systems Conference 2021.

Apart from this, there are other works realized during this PhD, but not included into the manuscript for the sake of conciseness and consistency. Among them there is a joint work in collaboration with Imperial College London and Argonne National Laboratory that considers the advantages and challenges of training on the Edge published in Proceedings of Workshop on Parallel AI and Systems for the Edge 2019 [59]. The other work is done in collaboration with University of Colorado and Northeastern University that estimates lower bound on the execution time for schedules of the tiled Cholevsky factorization published in Proceedings of European Conference on Parallel Processing 2020 [11].

[10] https://gitlab.inria.fr/hiepacs/rotor

Related Works

Rematerialization

Checkpointing for Automatic Differentiation

Adjoints computation is a numerical method for computing the gradient of a function, which may be complex. This method is at the core of many scientific applications, from climate and ocean modeling [3] to oil refinery [15]. In addition, the structure of the underlying dependence graph is also at the basis of the backpropagation step of machine learning [57], and thus the models considered in this manuscript are based on it.

Storage has been one of the key issue with the computation of adjoints: it is required to keep all the intermediate data to compute the final gradient, but it is possible to recompute them. Therefore, the computation of adjoints has always been a trade-off between recomputations and memory requirements [38].

When one type of limited memory is available, authors of [41] showed the optimality of a binomial approach that was later implemented under the name Revolve [39]. In the latter paper, closed form formulas providing the exact position of saved data have even been proposed for homogeneous chains (each operation has the same duration and memory cost). When computation times are heterogeneous, but data sizes are identical, an optimal checkpointing strategy can be obtained with Dynamic Programming [40]. The problem of adjoint computations has received an increasing attention in the recent years with the introduction of a second level of storage of infinite capacity but with access (write and read) costs [102, 6, 5, 94, 84]. Indeed, with the increase in the problem size, the memory was not sufficient anymore to solve the problems in a reasonnable time. Hence solutions have started considering the usage of disks to store some of the intermediary data. Several work have considered this problem. In [102], a first heuristic was presented that applies the schedule provided by Revolve, where the checkpoints that stay idle the longest are stored on disk (level 2 storage). Some implementations (for example [84]) are based on a two level checkpointing strategy: the first pass (forward mode) of the adjoint graph checkpoints periodicaly to disk (level 2), then the second pass (reverse mode) reads those disk checkpoints one after the other and uses Revolve with only memory (level 1) checkpoints. The main parameter (period used for the forward checkpointing) can be chosen by the user. The algorithm designed in [6] and denoted by Disk-Revolve is able to solve this problem optimally. In a subsequent work, authors of [5] showed that the

optimal solution returned by Disk-Revolve is weakly periodic, meaning that the number of forward computations performed between two consecutive checkpoints into the second level of storage is always the same except for a bounded number of them. More recently, they extended this result for a hierarchical memory architecture with an arbitrary number of storage levels [47].

In Chapter 1, we extend the classical results of [39] to meet the requirements of more complex adjoint computations emerging in DNN graphs of Siamese Neural Networks [13, 28, 75] and Cross-Modal-Embeddings [74, 77]. They represent a class of graphs that has a shape of multiple chains joint together at the end by the last operation, which in case of Deep Learning is a loss function. Our work shows that dynamic programming is still applicable for the new type of graph, but its complexity grows exponentially with a number of chains.

Rematerialization for DNNs

When a DNN represents a single chain of layers, the computation of the gradients in the training phase is similar to Automatic Differentiation (AD). The checkpointing strategies used in AD to reduce memory consumption are known in AI as Rematerialization or gradient checkpointing strategies. Rematerialization is the method that relies on recomputations to reduce the memory footprint of a given fixed model or architecture, while obtaining the exact same output of the training phase.

Recomputations can be applied to individual layers or modules of the neural network that are known to be especially heavy. For example, the authors of [83] show, for a popular neural network like DenseNet, that using shared memory storages and recomputing concatenation and batch normalization operations during backpropagation help to go from quadratic memory cost to linear memory cost for storing feature maps. Along the same idea, reimplementations of some commonly used layers like batch normalization have been proposed [91]. In the latter case, memory usage is reduced by rewriting the gradient calculation for this layer so that it does not depend on certain activation values (so that it is no longer necessary to store them).

A generic divide-and-conquer approach based on compiler techniques is able to perform automatic differentiation for arbitrary programs [98].

The use of rematerialization strategies inspired by AD has recently been advocated for DNNs in several papers [42, 20, 57, 60, 31, 52]. A direct adaptation of the results on homogeneous chains was proposed for the case of Recurrent Neural Networks (RNNs) in [42], but cannot be extended to other DNNs. Apart from this, for practical usage, an implementation of rematerialization exists in PyTorch [1], based on a simple periodic and single-pass rematerialization strategy that exploits the ideas presented in [20]. In this strategy, the chain is divided in equal-length segments, and only the input of each segment is materialized during the forward phase. This strategy provides non-optimal solutions in terms of throughput and memory usage, because it does not benefit from the fact that more memory is available when computing the backward phase of the first segment (since values materialized for later segments have already been used). This implementation was

nevertheless used to process significantly larger models [36].

Some researches attempted to adapt rematerialization strategies to Arbitrary Computation Graphs (ACG). On the one hand, a polynomial algorithm is provided in [31] that finds the rematerialization strategy for the forward propagation that minimizes memory used to execute ACG, under the assumption that activation deletion is not allowed during the backward phase (activations can be recomputed only once), which is a very strong and restrictive assumption in practice, especially in the case of deep networks. On the other hand, in the AD literature, the process is fully recursive, allowing the full memory usage throughout the entire training, since the released memory can be used later. In what follows, we refer to such solutions as single-pass rematerialization strategies.

A similar problem is considered in [60], where activation deletion during backward propagation are possible, though similarly the framework is restrictive on several points that are crucial in terms of practical performance and applicability. First, the study is limited to unit costs for data. More importantly, the approach described in [60] is based on the computation of a tree-width decomposition of the graph and only derives the minimum computational cost associated with the minimum memory footprint. The minimum memory footprint then depends on the quality of the decomposition, which is an NP-complete problem for which constant approximation algorithms exist. In practice, the problem to be solved is rather to minimize the computational cost while meeting a given memory constraint. Indeed, limiting the search to the smallest possible memory size obviously leads to a significant additional computational cost.

Another closely related approach is Checkmate [52] in which an Integer Linear Program is proposed to solve the rematerialization problem. This program can handle arbitrary graphs by assuming a fixed ordering of the execution, and can provide a solution of minimum runtime given a memory limit. However, solving this ILP is very computationally expensive and does not converge in a reasonable time as soon as the network exceeds a few dozen layers. Its rational approximation, however, can be easily found, but may push memory usage above memory limits

At last, other approaches finely control the tradeoff between memory and computation. In [61], the authors also consider a general ACG framework. Their work can be seen as a generalization of [20] algorithm to ACGs. More specifically, their goal is to decompose the ACG into groups of nodes and during the forward phase, only the boundaries between groups are materialized. Then, during the backward phase, to perform the gradient computations of a group, it is required to recompute all the activations of the group using its input saved boundary, and then the backward phase is performed without additional recomputing operations. On the one hand, the advantage of this approach is that it is tractable for ACGs using dynamic programming. On the other hand, as in [20] and [31], the search is restricted to single-pass rematerialization strategies. As we show in Chapter 2 in the case of chains, the fact of not using activation deletion during the backward phase induces significant memory waste and additional computational costs.

In [54], the authors proposed Dynamic Tensor Rematerialization that dynamically choose which activations should be discarded and then recomputed at runtime. Still,

it is based on a heuristic approach that encourages to discard tensors that have large memory and staleness costs and that can be easily recomputed while allowing cheap recomputations of other tensors as well. This heuristic showed good results, though optimal static rematerialization methods remain more reliable, taking into account that execution times and memory costs of layers normally do not change much over iterations.

To the best of our knowledge, Chapter 2 is the first attempt to precisely model heterogeneity and more importantly the ability, offered in DNN frameworks, to combine two types of activation savings, by either storing only the layer inputs (as done in AD literature), or by recording the complete history of operations that produced the outputs (as available in autograd tools). For this model, we propose a static algorithm with an optimality proof, based on dynamic programming. This algorithm manages to find the best schedule in polynomial time.

Offloading

Offloading is a potentially complementary approach to Rematerialization that 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 [88, 7]. In [88], 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 [7] 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 main memory are relatively straightforward. Proposed strategies consist in trying to offload either all activations 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.

Several follow-up works offer improvements over this first attempt. In order to reduce the overhead incurred by the communications, some authors [89] recommend to add compression to decrease the communication time, while others [62] design a memory-centric architecture to help with data transfers. Memory Virtualization was further considered in [76, 64, 49, 114]. In [76, 64, 49], 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 ACG that represent neural network training graphs. The authors of [64] 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 [114], where candidates for swapping are found by assigning priority scores to all activations.

As a complement to these practical approaches, in this work we perform the first theoretical analysis of the underlying optimization problem: which data to offload and

how to schedule transfers. We present both a complexity proof and optimal solutions to two of its relaxations in Chapter 3.

Combination of Rematerialization and Offloading

The works, combining both approaches, are relatively recent, though the idea comes naturally from the fact that they serve the same purpose, while they make use of different resources. The speed-centric rematerialization from [20] enhanced with memory-centric rematerialization (discards activations of every segment all the time) was combined with the simple offloading approach from [88] in [108]. Then, the authors in [86] also used rematerialization of [20] with a possibility of further offloading saved checkpoints to the CPU if rematerialization only is not enough to perform training under memory constraint.

Another approach that combines recomputations and data offloading from GPU memory to CPU memory was proposed in [85]. This approach is especially useful in the case where the size of the activations is small compared to the size of the model, which is the case in some NLP models. In this case, the network weights are offloaded to the CPU memory, that serves as a parameter-server host.

The similar idea was considered in Automatic Differentiation as well and it is known as Disk Checkpointing [6], which is discussed in more details above (see Checkpointing for Automatic Differentiation). However, this model is not entirely appropriate for DNN, as it is restricted to homogeneous chains, where the communication cost is constant without possible overlap with computations.

Our goal is to find simultaneously optimal Rematerialization and Offloading strategies that could take into account their joint impact on the makespan for DNNs represented by heterogeneous complex chains. In Chapter 4, we analyse the corresponding problem. We focus on the case where the model weights stay in GPU and the activations should be moved to the CPU memory. In that case, we demonstrate how these two dynamic programs from Chapters 2 and 3 can be merged together to provide the optimal combination that outperforms basic heuristics.

Pipelined Model Parallelism

When using Model Parallelism [24], the different layers of a network are spread over different resources, so that the storage of DNN weights and activations is shared between the resources. In Model Parallelism, only activations should be communicated and transfers take place just between layers assigned to different processors, which adds up to a low total amount of data movements with respect to other types of parallelisms. Despite that, the scalability of the method is poor because of chain connections in DNN computational graph that force a sequential execution of all the tasks.

The execution within Model Parallelism can be accelerated if several mini-batches are pipelined, and thus several training iterations are active at the same time, helping to keep computing resources busy most part of the time. The practical use of Pipelined Model

Parallelism is nevertheless a delicate issue and the analysis of the induced memory needs is complex. In [50], it is proposed to split the training batch into several mini-batches, which are then pipelined through the layers of the network (and the different computing resources). Once the forward and backward phases have been computed on all these mini-batches, the weights are then updated. This approach is fairly simple to implement but has the disadvantage of leaving the computational resources largely idle (*e.g.* after the first resource has executed its forward operations on the pipelined mini-batches, it has to wait until the corresponding backward operations become available to complete the iteration). The PipeDream approach proposed in [78] improves this training process, by only forcing that the forward and backward tasks use the same model weights for a given mini-batch. Such a weakened constraint on the training process allows PipeDream to achieve a much better utilization of the processing resources, but the asynchronous updates affect badly the overall convergence of the training.

Despite its advantages, PipeDream has a number of issues: (i) degraded convergence because of weight staleness that is non-uniform with respect to different stages, (ii) poor memory management because of redundant weight and activation copies produced by non-optimal schedule, (iii) inferior load balancing being restricted to contiguous allocations, (iv) not suitable for heterogeneous GPUs.

The poor convergence of asynchronous methods has been addressed in several papers. It is caused by weight staleness when the delayed gradients are used to perform an update step. Some works [43, 18] propose to predict weights during forward and backward propagation using the momentum of the gradient. Performing the updates less regularly [43, 79] (in contrast in PipeDream they are done after each backward) helps limiting weight staleness as well. Alternatively, PipeMare [111] proposes to reschedule learning rate depending on the pipeline stage and adapt the model weights for backward so that they are defined by the most recent version of weights and the accumulated weighted difference between the model weights from successive iterations and the stage number. The last method achieves the same convergence rate as GPipe, while having the same resource utilization as PipeDream without storing multiple copies of the weights.

Another important issue related to PipeDream is the need to keep many copies of the model parameters, which can potentially cancel the benefit of using Model Parallelism. To address this issue, the same methods that help with weight staleness can be used: in [79] the updates are done so that it is possible to keep only two versions of the weights; in [18] two versions of the weights are needed too, but also one gradient and momentum should be stored. The inefficient memory utilization by PipeDream has been also observed in [51]. Unlike other works, they offer another version of pipelining different from GPipe and PipeDream. Its principle of work can be described in the following way: once all forward steps on one mini-batch are processed by all GPUs and the first backward of the last stage is done, the same GPU can proceed to the first stage of the next mini-batch by performing its forward and then the remaining forwards of the new mini-batch are executed on the other processors in the reverse order just after the backwards of the preceding mini-batch. This allows GPUs to use memory immediately after it is released during backward steps. In general, it uses memory more efficiently, though memory itself

is not considered as a constraint.

Contiguous allocations can be also a bottleneck that hinders a throughput. The authors of [29] offer a method suitable for finetuning large models. They obtain non-contiguous allocations, by coarsely building stages that have a high ratio of computation time with respect to communication time. These stages can be further allocated to any device, allowing more than one stage per processor. To find non-contiguous allocation for ACGs, the authors of [106] propose two Integer Linear Programs (ILPs) (one minimizes latency, the other one maximizes throughput for a steady state situation) and a dynamic program. The obtained solutions are optimal for inference and can be adapted for training, though those methods do not take into account pipelining nature of model parallelism and scheduling, which have a significant influence on the peak memory usage.

Some researchers have worked on extending the results of PipeDream to heterogeneous computing clusters and heterogeneous communication links [113, 81, 73]. Finding the optimal load balance and schedule for heterogeneous settings is a difficult task, thus all of them rely on some simplifications and heuristics. To solve issues in the case of high communication costs and heterogeneous networking, the authors of [113] proposed an updated dynamic programming strategy that assumes no overlap between computations and communications. The HetPipe proposal [81] considers a different way of combining Data and Model Parallelism, in which nodes may contain different GPUs. The idea of HetPipe is to heuristically split the GPUs into virtual workers that may contain heterogeneous GPUs and use Data Parallelism between virtual workers. Model Parallelism is used inside the virtual workers, based on a simplified ILP that assumes no overlap between computation and communication. Pipelined Model Parallelism in [73] is done with a help of Deep Reinforcement Learning.

Other extensions of PipeDream explore different ways of combining Model Parallelism with other types of parallelism [81, 30, 66, 68]. In the DAPPLE framework [30], Model Parallelism is implemented alongside Data Parallelism. There, the focus is on the case of several nodes, each equipped with several GPUs. DAPPLE extends PipeDream by allowing more possibilities to map a stage of the DNN to GPUs located in several nodes. The assignment problem is solved without taking memory constraints into account. Furthermore, [66] does Hybrid Parallelism, using Tensor Slicing, Data and Model Parallelism, finding job allocation with dynamic programming. However, this method does not take into account the memory constraints.

Transformers offer a new dimension for pipelined parallelism. In [68], the pipelining is not performed through micro-batching. Instead, they pipeline the tokens in the input sequence. Such approach manages to significantly accelerate the training of GPT-3. Their solution is based on dynamic programming without memory considerations.

Our work carefully investigates the limitations of PipeDream in Chapter 5. To the best of our knowledge, we are the first to notice and estimate the effect of the chosen schedule on the peak memory usage of the pipelining. We also evaluate to which extent non-contiguous allocations can be advantageous with respect to contiguous ones. Therefore, we propose an ILP in Chapter 6 that finds simultaneously the optimal load balance based on non-contiguous allocations and the optimal schedule, taking into account

all sources of memory consumption. In addition to the ILP, we also offer a heuristic MadPipe described in Chapter 7. It is based on dynamic programming that also combines non-contiguous allocations with scheduling considerations to find the best load balancing. Our methods can be combined with [43, 18, 79, 111] to improve the training convergence. Despite targeting only Model Parallelism, our solutions can be easily adapted for Hybrid Parallelism, using some simple heuristics. As a future work, our approach should be extended to heterogeneous systems as well.

Часть I

Rematerialization

Introduction

Training Deep Neural Network (DNN) is a memory-intensive operation. Indeed, the training algorithms of most DNNs require to store both the model weights and the forward activations in order to perform backpropagation. In practice, training is performed automatically and transparently to the user through autograd tools for backpropagation, such as `tf.GradientTape` in TensorFlow or `torch.autograd.backward` in PyTorch. Unfortunately, the memory limitation of current hardware often prevents data scientists from considering larger models, larger image sizes or larger batch sizes [91, 83].

To alleviate the memory limits, we propose in this part to use a technique known as *Checkpointing* in Automatic Differentiation or *Rematerialization* in Deep Learning. It proposes solutions for the problem of scheduling a graph with a shared bounded memory. This problem with homogeneous data is analogous to the Register Allocation problem or Pebble Game. Given a bounded number of unit-size registers (or memory slots), can we execute the graph while respecting the constraint that to execute a task, all its inputs need to be in a register? Authors of [96] showed that this problem is NP-complete for general task graphs. Further study showed that the problem is solvable in polynomial graph for tree-shaped graphs [72], or recently Series-Parallel graphs [53].

In this part, we are interested in what we denote by backpropagation graphs: given a directed acyclic graph with a single sink node, we construct a dual identical graph where the edges are reversed, and where each node of the initial graph is connected to its dual node. The source node of the dual graph and the sink node of the original graph are merged into a single node called *loss* (see Figure 1.1 for the case of a chain of nodes). These types of graphs have been widely studied in the context of Automatic Differentiation (AD) [37]. For a given batch size and a given neural network model and even on a single device without relying on model parallelism strategies, it enables to save memory at the price of activation recomputations. In the context of AD, networks can be seen as (long) homogeneous (i.e., all stages are identical) chains and 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 techniques consist in determining in advance which forward checkpoints should be kept into memory and which one should be recomputed from stored checkpoints when performing the backward phase. Many studies have been performed to determine optimal checkpointing strategies for AD in different contexts, depending on the presence of a single or multi level memory [6]. In the case of homogeneous chains, closed form formulas providing the exact position of checkpoints have even been proposed [39], although the general algorithmic ingredient is to derive optimal checkpointing strategies using dynamic programming [39].

The use of checkpointing strategies has been recently advocated for DNN in several papers [42, 20] and a simple periodic checkpointing strategy, non optimal but still efficient implementation is provided in PyTorch [82, 1] for the restricted case of homogeneous chains, whereas DNN models are in general more complicated. While optimal scheduling and checkpointing are still open in the general case, there are some solutions that in some way benefit from the findings in AD checkpointing strategies [20] [42]. However, they

are designed to deal with only sequential models, thus making it inappropriate for more sophisticated cases.

In Chapter 1, we present the first attempt to find optimal checkpointing strategies adapted to more general networks. We show how techniques developed in Automatic Differentiation can be extended to DNN networks. More specifically, we concentrate on the particular DNN consisting of several independent chains whose results are gathered through the computation of the loss function. This case corresponds to the case of Siamese and Cross Modal Networks. We show that this specific case, provided that all computational and memory costs are homogeneous, is still solvable using dynamic programming, but at the price of increased complexity and, hence, higher computational cost. In Section 1.1, we present our general model and the notations that will be used throughout the chapter and the part, and we present a few basic results of the Automatic Differentiation literature. Then, properties of optimal solutions are proposed in Section 1.2 and are later used in Section 1.3 to find the optimal checkpointing strategy through dynamic programming in the case of multiple chains. At last, we present our implementation and simulation results in Section 1.4, before providing concluding remarks in Section 1.5.

In Chapter 2, we focus rather on more practical use of Checkpointing (or Rematerialization). We carefully model the operations that are available in DNN frameworks and analyze the specificity of computational graphs built by learning frameworks. We show that autograd tools offer more general operations and thus more optimization opportunities than those used in Automatic Differentiation. We assume that a DNN is given as a linear sequence of modules, where internal modules can be arbitrarily complex. In practice, this assumption does not hinder the class of models that can be considered, and we propose implementations of classical networks (ResNet, Inception, VGG, DenseNet) under this model. Moreover, we prove that models with heterogeneous activation sizes (in addition to the heterogeneous computation times that was previously considered in the literature) no longer satisfy the memory persistence property, contrary to what is typically assumed in the literature on rematerialization strategies for DNNs. We derive both an optimal algorithm that does not assume memory persistence and a relaxation to obtain the optimal memory persistent solution. This relaxed solution may not be optimal in the worst case, but in practice it shows the same performance as the optimal one.

Another contribution of Chapter 2 is a complete and easy-to-use implementation of the algorithm we propose in the PyTorch framework, called ROTOR [48]. This tool automatically measures the characteristics (memory consumption, computation time) of each layer of the DNN, and then computes the forward and the backward phases while enforcing a memory limit, at the cost of a minimal amount of recomputations.

We start with model description and notations in Section 2.1. Then we show in Section 2.2 that in case of heterogeneous chains the problem is NP-complete in the weak sense. After we introduce two dynamic programming algorithms that solve the rematerialization problem in pseudo-polynomial time in Section 2.3. Our PyTorch implementation of the rematerialization algorithm is described in Section 2.4. We

show through an extensive experimental evaluation in Section 2.4.5 that the additional operations provided by autograd frameworks indeed enable to significantly increase the throughput (the average number of processed images per second) when performing training. Finally, we give a conclusion in Section 2.5.

Глава 1

Multi-Chain Rematerialization

1.1 Framework

1.1.1 Introduction to Adjoint Chain

Figure 1.1 depicts the typical task graph that is considered in Automatic Differentiation. The top line shows the calculation of some complex function, which can be composed of several elementary operations, denoted by F_ℓ for $1 \leq \ell \leq L$. We denote the result of this complex function $F_L(F_{L-1}(\dots F_2(F_1(a_0))))$ as Loss. In order to compute the derivative $\frac{\partial \text{Loss}(a_0)}{\partial a_0}$, the simplest way is to use the chain rule, *i.e.* $\frac{\partial \text{Loss}(a_0)}{\partial a_0} = \frac{\partial \text{Loss}(a_L)}{\partial a_L} \frac{\partial F_L(a_{L-1})}{\partial a_{L-1}} \cdot \dots \frac{\partial F_1(a_0)}{\partial a_0}$, which can be computed iteratively starting from $B_L(a_{L-1}) = \frac{\partial \text{Loss}(a_{L-1})}{\partial a_{L-1}} = \frac{\partial \text{Loss}(a_L)}{\partial a_L} \frac{\partial F_L(a_{L-1})}{a_{L-1}} = \delta_{L-1}$ and finishing by finding $B_1(a_0) = \frac{\partial \text{Loss}(a_1)}{\partial a_1} \cdot \frac{\partial F_1(a_0)}{\partial a_0} = B_2(a_1) \cdot \frac{\partial F_1(a_0)}{\partial a_0} = \delta_1 \cdot \frac{\partial F_1(a_0)}{\partial a_0} = \delta_0$. This iterative process is depicted in the bottom line of Figure 1.1. Besides, evaluation of $\frac{\partial F_\ell(a_{\ell-1})}{\partial a_{\ell-1}}$ for some arbitrary ℓ generally requires $a_{\ell-1}$ as an argument (*e.g.* if $f(x) = x^2$ then $\frac{\partial f(x)}{\partial x} = 2x$ and it depends on x), which creates these transversal data dependencies between the output of $F_{\ell-1}$ and the input of B_ℓ. These long term data dependences may cause some memory issues. For instance, a_1 has to be kept in memory until the end of the whole process, since it will be used to compute δ_1. Overall, the final *adjoint chain* is assumed to be homogeneous, *i.e.* the time cost of any F_ℓ is $u_F \in \mathbb{R}^+$, of any B_ℓ is $u_B \in \mathbb{R}^+$, of the Loss is $u_L \in \mathbb{R}^+$ and all activations (intermediate data a_ℓ for $0 \leq \ell \leq L$) and gradients (δ_ℓ for $0 \leq \ell \leq L$) have unitary storage cost. For example, this is the case if for any $\ell \geq 0 : a_\ell \in \mathbb{R}^+$ (*i.e.* when working with scalars), then each data occupy the same amount of memory, while elementary operations such as F_ℓ or B_ℓ have a uniform duration.

Figure 1.1 corresponds as well to the training phase of a simple sequential DNN. The general framework for the training phase is the following: we pass a global input a_0 to the chain of length L, which is propagated until Loss using F computations that are called *forward* steps. F operations typically represent a sequence of linear functions (matrix or tensor operations) followed by a non linear function (typically ReLU); such operations

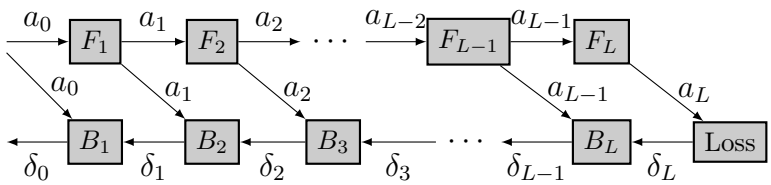


Рис. 1.1: The data dependencies in the Adjoint Computation graph.

are called *layers* in the vocabulary of deep learning. Layer ℓ may be parametrized by some matrix or tensor called *weight* W_ℓ. The choice of these weights affect Loss, which measures the error of the neural network. To minimize this error, it is required to tune the weights, by iteratively applying to them an update that depends on the gradients $\frac{\partial \text{Loss}}{\partial W_\ell}$. For that, after reaching the loss function, backpropagation is triggered, whose purpose is to compute these gradients using again the chain rule like described in the previous paragraph, since $\frac{\partial \text{Loss}}{\partial W_\ell} = \frac{\partial \text{Loss}}{\partial a_\ell} \cdot \frac{\partial F_\ell(a_{\ell-1}, W_\ell)}{\partial w_\ell} = B_{\ell+1}(a_\ell) \cdot \frac{\partial F_\ell(a_{\ell-1}, W_\ell)}{\partial W_\ell} = \delta_\ell \cdot \frac{\partial F_\ell(a_{\ell-1}, W_\ell)}{\partial W_\ell}$. Therefore, during this backward propagation phase, it is necessary to compute δ_ℓ for $1 \leq \ell \leq L$ with $B_{\ell+1}$ for weight updates and thus the same data dependencies hold as in Automatic Differentiation procedure. Unlike the classical problems of Automatic Differentiation, in general, the computational graphs for DNNs can be non-homogeneous and can have a more complex structure than the one shown in Figure 1.1. However, for simplicity, in this chapter we consider a subset of neural networks that satisfy homogeneity property: *i.e.* all forward steps have the same execution cost $u_F \in \mathbb{R}^+$, all the backward steps have the same execution cost $u_B \in \mathbb{R}^+$, the cost of Loss is $u_L \in \mathbb{R}^+$, while all input/output data have the same (unit) size; yet these neural networks should correspond to a more general graph structure than the simple adjoint chain. We consider the case of heterogeneous activation sizes and heterogeneous computing costs in Chapter 2.

1.1.2 Platform Model and Optimization Problem

In this chapter, we consider a platform consisting of a single compute element with a finite memory $\mathcal{M}$ of size m. We use this memory to store the data (either input or output) of operations. The parameters of parametrized operations are assumed to be stored in advance outside $\mathcal{M}$. Further in the text, we refer to the stored data as checkpointed data or *checkpoints* (meaning that we can redo a part of computations from them). Note that we do not consider the memory required by the actual execution of the job. Indeed, we assume in this chapter that there is some other level of memory that is always reserved for this type of operation (a special buffer).

To execute a job on this platform, at the beginning of the execution, all its inputs need to be stored in memory.

A job is represented as a Directed Acyclic Graph (DAG) $\mathcal{G} = (V, E)$, where each node of $v \in V$ represents a compute operation (with a given execution time), and each edge of $(v_1, v_2) \in E$ represents a data dependency where an output of v_1 is an input of v_2.

In accordance to the scheduling literature, we use the term *makespan* to denote the total execution time. Then given a graph $\mathcal{G}$, the problems under consideration are (i) can we execute it with a memory of size m (*pebble game problem*) and (ii) if we can, what is the minimal execution time to execute $\mathcal{G}$ (*makespan problem*) with a memory of size m.

The core of the *makespan problem* is the following: while there can be enough memory to execute the graph, the memory may not suffice to execute it in one go. Hence, we need to choose which data to store and which nodes should be recomputed. For instance, after having computed a_2, we can remove a_1 from the memory. Then, during the backward propagation phase, at the time to compute δ_1, a_1 will be recomputed from a_0 and then used immediately to compute δ_1. We can rely on checkpointing techniques known from Automatic Differentiation to solve this problem and find a good schedule. In this context, an optimal checkpointing strategy is a strategy that, given a chain and a memory size (expressed in terms of free memory slots to hold activations a_ℓ), computes which activations should be kept into memory and when, in order to minimize the number of recomputations while fitting into the memory constraint.

Definition 1 (Adjoint Computation [39, 102])**.** An adjoint computation (AC) with L time steps (or layer for DNNs) can be described by the following set of equations:

$$\begin{aligned} F_\ell(a_{\ell-1}) &= a_\ell && \text{for } 1 \leq \ell \leq L \\ B_\ell(a_{\ell-1}, \delta_\ell) &= \delta_{\ell-1} && \text{for } 1 \leq \ell \leq L \\ \textsc{Loss}(a_L) &= \delta_L \end{aligned}$$

The dependencies between these operations are represented by the graph $\mathcal{G} = (V, E)$ depicted in Figure 1.1.

Intuitively, the core of the AC problem is the following: after the execution of a forward step F_ℓ, its input is replaced by the corresponding output in the memory, then when it is required to perform the backward step B_ℓ, if the value $a_{\ell-1}$ is in the memory, then it can be completed immediately, otherwise this value should be recomputed from the closest available checkpointed value. Since F replaces its input value by the corresponding output value, in order to checkpoint a value, it is necessary to duplicate it into the memory before applying F.

Finally, our problem can be written like the following:

Problem 1 ($\textsc{Single}(L, m)$)**.** Under a memory limit m we want to minimize the makespan of the AC problem of computing δ_0 from input a_0, corresponding to the adjoint chain of length L from Figure 1.1, where the costs of every forward step, backward step and loss are u_F, u_B and u_B respectively. It is summarized in the following table:

		Initial state:	Final state:
AC chain:	size L		
Steps:	u_F, u_B, u_L		
Memory:	m	$\mathcal{M} = \{a_0\}$	$\mathcal{M} = \{\delta_0\}$

1.1.3 Backpropagation Graphs

The task graph described in the previous section belongs to a more general class of graphs: the backpropagation graphs. In general, backpropagation is a special sort of transformation of a graph:

Definition 2 (Backpropagation transformation (BP-transform))**.** Given a DAG $\mathcal{G}$ with a single sink node, the *BP-transform* of $\mathcal{G}$ is defined by the following procedure:

1. Build the dual graph $\tilde{\mathcal{G}}$ defined as the same graph where all edges are inversed.
2. For a given node in $\mathcal{G}$, connect its input edges to its dual node in $\tilde{\mathcal{G}}$.
3. Finally, merge the sink node of $\mathcal{G}$ and the source node of $\tilde{\mathcal{G}}$ as a single node Loss.

Note that the nodes of the initial graph are denoted as *forward steps*, while the nodes of the dual graph are denoted as *backward steps*.

Note that the graph in Figure 1.1 is the BP-transform of a linear chain.

The key property of the backpropagation graph that justifies this study is:

Property 1 (Properties of the BP-graph)**.** *Given a DAG $\mathcal{G}$ with n nodes and a single sink node, without recomputation of nodes, the minimal memory usage to go through the backpropagation graph of $\mathcal{G}$ is $O(n)$.*

This property originates from the fact that all input edges of graph $\mathcal{G}$ are also connected to dual nodes of $\tilde{\mathcal{G}}$. Thus cutting the BP-transform graph after a sink node of $\mathcal{G}$ should split it into $\mathcal{G}$ and $\tilde{\mathcal{G}}$ with a cut-set (the set of edges connecting the separated parts) of size $O(n)$, which represents the highest memory demand of the graph.

Indeed, to scale these types of computations, we are interested by the question of the overhead in computation when one uses much less memory space.

Another interesting property of backpropagation graphs is that, backward steps of adjoint chains are not recomputed in optimal solutions. It is due to the fact that there is only one outcoming edge from each backward operation to the next one, thus there is no point in computing them more than once to complete the execution.

1.1.4 Multiple Adjoint Chains Computation Problem

Computational graphs of neural networks may vary a lot. Multi-Layer Perceptron has a very sequential structure, therefore its BP-transform yields the graph structure of the adjoint chain (depicted in Figure 1.1). However, more advanced neural networks may contain fork-join parts, skip-connections and even grid structures in their graphs. In order to generalize the results given for the adjoint chain, we consider in this chapter transformation of *join graphs* or also called *multi-adjoint chains* (Figure 1.2). These join graphs can be encountered, for example, in the fork-join elements of Inception [104] or Inception-ResNet [103] networks. Furthermore, join graphs are used by several deep learning models such as Siamese Neural Network [13, 28, 75] or Cross-modal embeddings [74, 77].

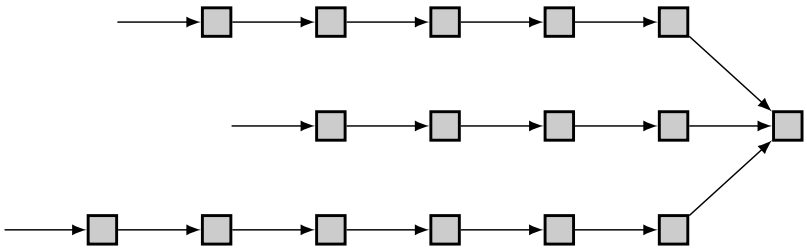


Рис. 1.2: A join graph with 3 branches of respective length 5, 4 and 6.

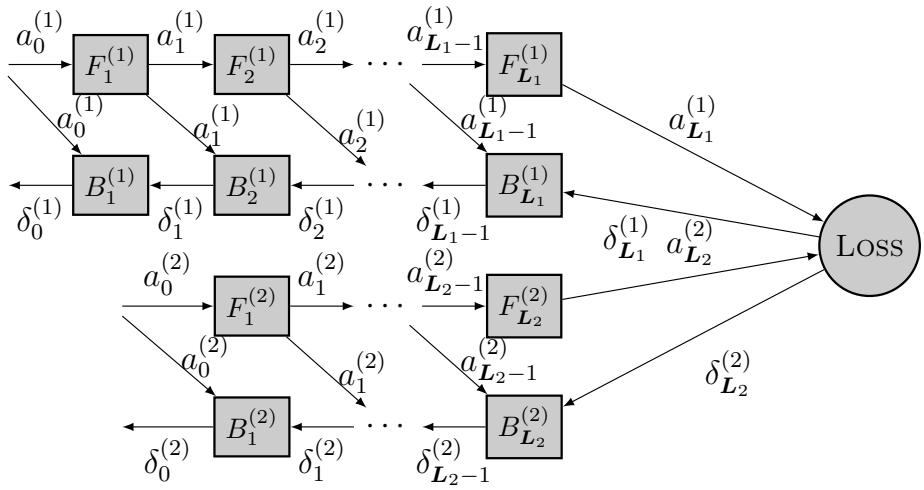


Рис. 1.3: Data dependencies in the BP-transform of a K-Join problem with two chains.

Definition 3 (BP-transform of a K-Join)**.** Given a join graph $\mathcal{G}_K$ with K branches of lengths $\boldsymbol{L} = (\boldsymbol{L}_1, \ldots, \boldsymbol{L}_K)$, its BP-transform can be described by the following set of equations:

$$
\begin{aligned}
F_i^{(j)}(a_{i-1}^{(j)}) &= a_i^{(j)} && \text{for } 1 \leq j \leq K \text{ and } 1 \leq i \leq \boldsymbol{L}_j, \\
B_i^{(j)}(a_{i-1}^{(j)}, \delta_i^{(j)}) &= \delta_{i-1}^{(j)} && \text{for } 1 \leq j \leq K \text{ and } 1 \leq i \leq \boldsymbol{L}_j, \\
\textsc{Loss}(a_{\boldsymbol{L}_1}^{(1)}, \ldots, a_{\boldsymbol{L}_K}^{(K)}) &= (\delta_{\boldsymbol{L}_1}^{(1)}, \ldots, \delta_{\boldsymbol{L}_K}^{(K)}).
\end{aligned}
$$

The dependencies between these operations are represented by the graph $\mathcal{G} = (V, E)$ depicted in Figure 1.3, in the case of $K = 2$ chains.

In the BP-transform of a K-Join model, the last forward value of each chain is required to compute Loss and to start the backward propagation phase. Before the computation of the loss (resp. after the computation of the loss), all forward (resp. backward) operations on the different chains can be performed independently, except that all chains share the same set of storage slots.

We can now present the optimization problem under consideration:

Problem 2 (MULTI-$\delta(\boldsymbol{L}, m)$)**.** Under a memory limit m we want to minimize the makespan of computing the BP-transform of a K-Join for some $K \in \mathbb{N}$ of computing $\delta_0^{(1)}, \ldots, \delta_0^{(K)}$ from inputs $a_0^{(1)}, \ldots, a_0^{(K)}$, corresponding to the join-graph with branch lengths $\boldsymbol{L} = (\boldsymbol{L}_1, \ldots, \boldsymbol{L}_K)$, where the costs of every forward step, backward step and loss are u_F, u_B and u_B respectively. It is summarized in the following table:

		Initial state	Final state
Join size:	$\boldsymbol{L} = (\boldsymbol{L}_1, \ldots, \boldsymbol{L}_K)$		
Steps:	u_F, u_B, u_L		
Memory:	m	$\mathcal{M} = \{a_0^{(1)}, \ldots, a_0^{(K)}\}$	$\mathcal{M} = \{\delta_0^{(1)}, \ldots, \delta_0^{(K)}\}$

In practice, once some branch j reaches its last backward $B_1^{(j)}$, *i.e.* the corresponding backward propagation and weight updates are completed, $\delta_0^{(j)}$ is not required anymore, therefore it can be discarded to have more memory for unfinished branches. Hence, we also consider another variant of the above problem.

Problem 3 (MULTI-$\emptyset(\boldsymbol{L}, m)$)**.** We want to solve Problem 2, where, after the backpropagation of some branch i, it is allowed to discard $\delta_0^{(i)}$ to have more available memory slots for other branches.

		Initial state	Final state
Join size:	$\boldsymbol{L} = (\boldsymbol{L}_1, \ldots, \boldsymbol{L}_K)$		
Steps:	u_F, u_B, u_L		
Memory:	m	$\mathcal{M} = \{a_0^{(1)}, \ldots, a_0^{(K)}\}$	$\mathcal{M} = \emptyset$

Let us note that in the case of only one branch all optimal solutions of MULTI-$\delta(\boldsymbol{L}, m)$ should be optimal for MULTI-$\emptyset(\boldsymbol{L}, m)$ as well.

In order to solve MULTI-$\delta(\boldsymbol{L}, m)$ and MULTI-$\emptyset(\boldsymbol{L}, m)$, we introduce the auxiliary problem MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ that takes as additional input a binary vector $\boldsymbol{b}$ such that, $\forall i \in \{1, \ldots, K\}$:

$$\boldsymbol{b}_i = \begin{cases} 1 & \text{if the result of the last backward step for the chain } i\text{, i.e. } \delta_i^{(0)}\text{, should be kept,} \\ 0 & \text{otherwise.} \end{cases}$$

Problem 4 (MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$)**.** We want to solve Problem 2, where for all branches i such that $\boldsymbol{b}[i] = 0$, it is allowed to discard $\delta_0^{(i)}$ to have more available memory slots for other branches.

		Initial state	Final state
Join size:	$\boldsymbol{L} = (\boldsymbol{L}_1, \ldots, \boldsymbol{L}_K)$		
Steps:	u_F, u_B, u_L		
Memory:	m	$\mathcal{M} = \{a_0^{(1)}, \ldots, a_0^{(K)}\}$	$\mathcal{M} = \{\delta_0^{(i)},\ \forall i \ s.t.\ \boldsymbol{b}_i = 1\}$

Let us note that MULTI-$\delta(\boldsymbol{L}, m)$ and MULTI-$\emptyset(\boldsymbol{L}, m)$ are special cases of MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ when $\boldsymbol{b} = \vec{1}$ and $\boldsymbol{b} = \vec{0}$ respectively.

1.1.5 Previous Results for Single Adjoint Chain Computation Problem

In this work, we use results from the literature for a slightly different version of SINGLE(L, m). We define ADJCHAIN(ℓ, m) the problem consisting of minimizing the makespan of the adjoint chain of size ℓ depicted in in Figure 1.4 with m memory slots, where a_0 and $\delta_{\ell+1}$ are initially stored into the memory, and where at the end of the computation δ_0 should be stored in memory. In contrast with Figure 1.1, there is no more LOSS computation at the "turn" of the graph, but instead $B_{\ell+1}$, which like other backward operations takes as an input one activation computed during forward propagation a_ℓ and one gradient $\delta_{\ell+1}$.

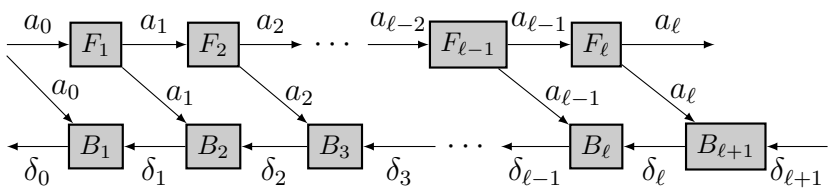


Рис. 1.4: Data dependencies in the AC graph of size ℓ in ADJCHAIN(ℓ, m).

Problem 5 (ADJCHAIN(ℓ, m))**.** We want to minimize under the memory limit m the makespan of the AC problem of computing δ_0 from input a_0 and $\delta_{\ell+1}$, corresponding to the adjoint chain of length ℓ from Figure 1.4, where the costs of every forward step and backward step are u_F, and u_B respectively. It is summarized in the following table:

		Initial state:	Final state:
AC chain:	size ℓ		
Steps:	u_F, u_B		
Memory:	m	$\mathcal{M} = \{a_0, \delta_{\ell+1}\}$	$\mathcal{M} = \{\delta_0\}$

This problem can be solved optimally with the dynamic program described below.

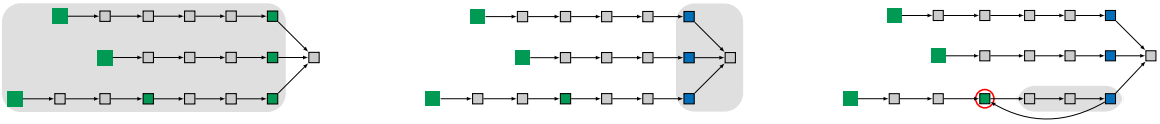
Definition 4 ($\text{Opt}_0(\ell, m)$)**.** Given $\ell \in \mathbb{N}$, and $m \in \mathbb{N}$, $\text{Opt}_0(\ell, m)$ denotes the execution time of an optimal solution to ADJCHAIN(ℓ, m).

Theorem 1 ([6])**.** *$Opt_0(\ell, m)$ can be computed with the following dynamic program*

$$\forall m \geq 2, \quad Opt_0(0, m) = u_b \tag{1.1}$$

$$\forall \ell > 0, \quad Opt_0(\ell, 3) = \frac{\ell(\ell+1)}{2} u_f + (\ell+1) u_b \tag{1.2}$$

$$\forall \ell > 0, m > 3, \quad Opt_0(\ell, m) = \min_{1 \leq i < \ell} \{i u_f + Opt_0(\ell - i, m-1) + Opt_0(i-1, m)\} \tag{1.3}$$



(a) The forward phase (b) The turn (c) The backward phase

Рис. 1.5: The three main phases of the algorithm. Green blocks correspond to the storage of "forward" data, blue blocks to storage of "backward" data.

The minimal amount of memory slots required to reverse an AC graph of size larger than 1 is equal to 3, in order to keep (i) the initial value a_0, (ii) the current forward value and (iii) the current backward value.

This problem has been heavily studied in the community of automatic differentiation. Grimm et al. [41] showed the optimality of a binomial approach, which was later implemented by Griewank and Walther [39] under the name Revolve. Revolve [39] takes a length ℓ and a number of checkpoints m (memory limit) and returns an optimal solution sequence to AdjChain(ℓ, m).

The dynamic program $\mathrm{Opt}_0(\ell, m)$ can be as well used to solve optimally Single(L, m). Indeed, AdjChain(ℓ, m) solves optimally the part of Single(L, m) that is obtained by removing the last forward and Loss computation. The optimal makespan of Single(L, m) is therefore $\mathrm{Opt}_0(L-1, m) + u_F + u_L$.

1.2 Characterization of Optimal Solutions for Multi-Adjoint Chains

In this section, we present the core idea to compute an optimal solution for BP-transform of a K-Join. In particular, we define canonical solutions (Section 1.2.2), and show that there always exists an optimal solution that is canonical. Then, in Section 1.3, we show how to compute a canonical optimal solution with a dynamic program.

Let us first notice that an algorithm for Problems 2, 3 and 4 can be decomposed into three different phases (see Figure 1.5):

- **The Forward phase**: we traverse all branches to write all inputs of the loss function in memory. During this phase one cannot compute any backward operations, but some of the input data can be stored.
- **Loss**: at the beginning of this phase, all input data of Loss are stored in memory, and are replaced by all output data at the end in the same memory locations. Indeed, the input data of the Loss will never be used anymore.
- **The Backward phase**: we read some input data that were stored earlier to backpropagate a subset of the graph.

1.2.1 Motivating Example

Let us start by presenting a toy example and its associated optimal solution in order to introduce the main ingredients of our algorithm and the main lemmas and theorems that prove its correctness. We will consider a network with 3 branches of respective lengths 4, 10 and 12 ($\boldsymbol{L} = (4, 10, 12)$). An instance of Problem 3 is also characterized by the size of the available memory m, expressed in terms of number of forward or backward values that it can host. The other parameters that describe the problem are u_F, u_B, u_L, that are all set to 1 in the toy example.

Note that the above defined instance of Problem 3 can be transformed into an instance of Multi-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$, if we set $\boldsymbol{b} = (0, 0, 0)$. Finding the schedule that achieves minimum makespan for 3 branches of lengths $\boldsymbol{L} = (4, 10, 12)$ and unitary computational costs while using at most $m = 9$ memory slots therefore corresponds to solving the problem with $\boldsymbol{L} = (4, 10, 12), m = 9, u_F = 1, u_B = 1, u_L = 1, \boldsymbol{b} = (0, 0, 0)$.

In order to solve this instance, we rely on dynamic programming and express the solution of the problem as the minimum between the solutions for its "smaller" sub-problems. The key ingredient to do this is Theorem 3 and more specifically Eq. (1.4). Eq. (1.4) says that the optimal solution can be obtained by considering all possible positions of the first checkpoint (denoted as i) taken during the forward phase (before the computation of Loss) on the branch that will finish last its backward propagation (denoted as k). To establish this result, we rely on the characterization of a class of optimal schedules that is defined in Definition 8.

In our specific toy example, the optimal (i, k) pair is $(9, 3)$ so that we checkpoint $a_9^{(3)}$. The solution for the toy example is depicted on Figure 1.6. In this picture, time is depicted on the $y-$axis and the position on the chain is depicted on the $x-$axis. The chains of respective initial lengths 4, 10 and 12 are depicted using respectively colors blue, green and red. The first 9 time units are used to perform the first 9 forward computations on the red branch, and the red branch will end up the computation with operation Revolve$(8, 9)$.

We are now left with two problems.

1. The first problem is an instance of Multi-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ with parameters ($\boldsymbol{L} = (4, 10, 3), m = 8, u_F = 1, u_B = 1, u_L = 1, \boldsymbol{b} = (0, 0, 1)$). Since one memory slot is dedicated to store $a_9^{(3)}$, the number of available memory slots is now $m = 9 - 1 = 8$. At last, $\boldsymbol{b}$ also changed since it is now (0,0,1). This change expresses the fact that the initial backward value on the last branch should now be kept in memory. Indeed, since the last chain corresponds to the truncated chain, this value is needed in order to continue the backward propagation on the red branch. This observation is the key point to explain the introduction of Multi-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$.
2. The second problem is an instance of the classical single chain problem as previously described in the Automatic Differentiation literature and consists in processing the first 9 elements of the red chain (9 last backward operations, restarting from the beginning), with all available memory slots since it will be processed last after all other branches. Its optimal time can be found with Revolve$(8, 9)$.

In Eq. (1.4), we can recognize the different terms corresponding to the first $i = 9$

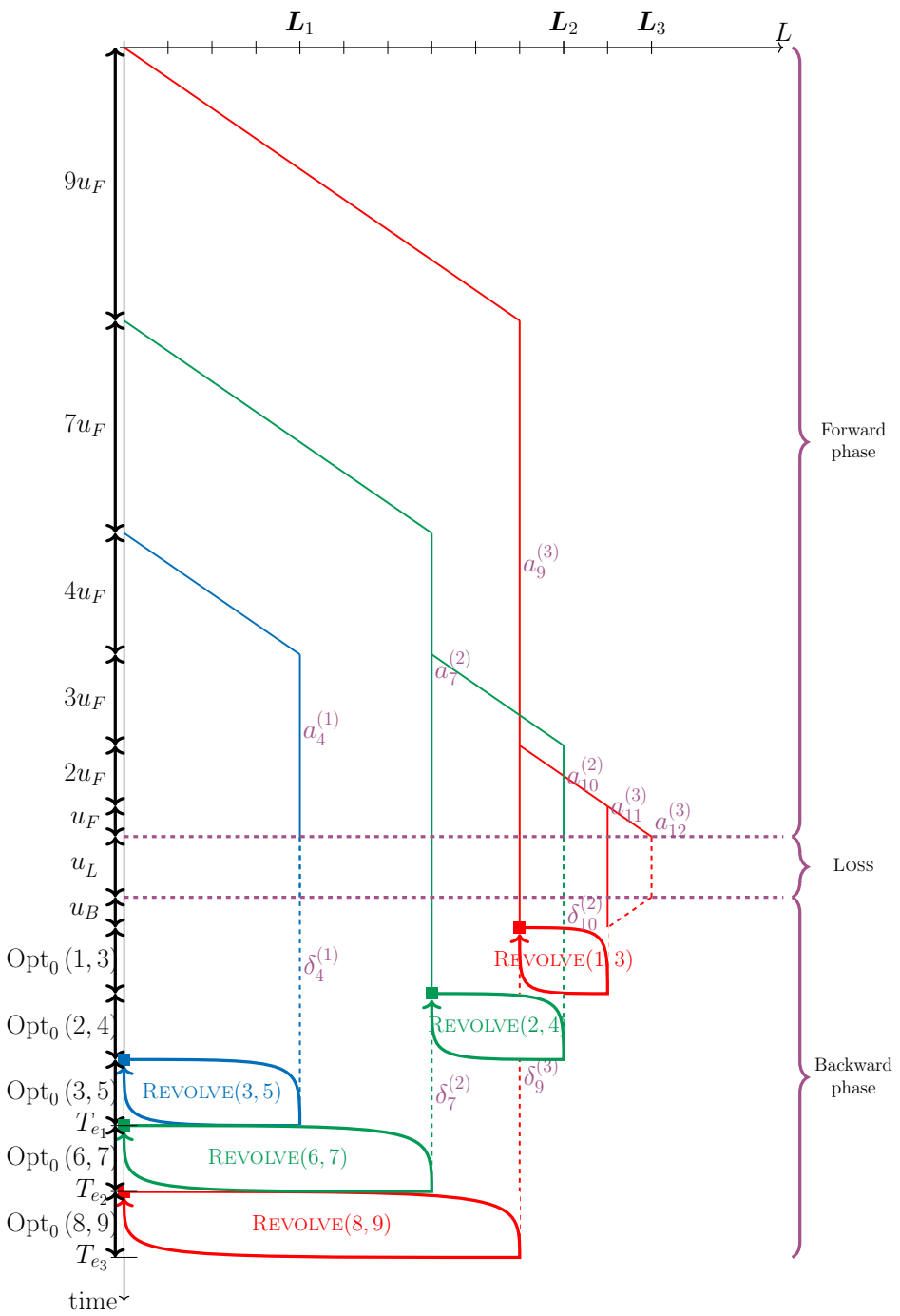


Рис. 1.6: Toy example, an optimal schedule for Multi-$\boldsymbol{b}(\boldsymbol{L} = (4, 10, 12), m = 9, \boldsymbol{b} = (0, 0, 0))$

forward steps, to the first problem MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ with parameters $(\boldsymbol{L} = (4, 10, 3), m = 8, u_F = 1, u_B = 1, u_L = 1, \boldsymbol{b} = (0, 0, 1))$ (with $k = 3$) and to the second problem $\mathrm{Opt}_0\left(i-1, m - \sum_{j \neq k} \boldsymbol{b}_j\right)$ (with all $\boldsymbol{b}_j$ being 0 except on the checkpointed branch $k = 3$).

To solve MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ with parameters $(\boldsymbol{L} = (4, 10, 3), m = 8, u_F = 1, u_B = 1, u_L = 1, \boldsymbol{b} = (0, 0, 1))$, we again rely on Eq. (1.4), and we decide to checkpoint the 7th forward value on the green branch. This again leaves us with two problems, MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ with parameters $(\boldsymbol{L} = (4, 3, 3), m = 7, u_F = 1, u_B = 1, u_L = 1, \boldsymbol{b} = (0, 1, 1))$ and $\mathrm{Opt}_0\,(6, 7 = 8 - 1)$. For $\mathrm{Opt}_0\,(\ell, m)$, the number of available checkpoints is 7 and not 8, because we need to keep both the initial forward and one backward value for the red chain, as expressed by the fact that its associated b value is 1.

To solve MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ with parameters $(\boldsymbol{L} = (4, 3, 3), m = 7, u_F = 1, u_B = 1, u_L = 1, \boldsymbol{b} = (0, 1, 1))$, we rely on a different strategy, based on the observation that 7 is the minimal number of memory slots to process 3 chains. Indeed, immediately after LOSS, 3 slots will be occupied by the initial values of the different chains and 3 slots will be occupied by the backward values produced by LOSS. Another slot is necessary to process any forward value on any chain, that can be seen as a buffer for forward computations. In general, Lemma 2 provides the general formula to find the minimum number of memory slots necessary to process multiple chains defined by $\boldsymbol{L}$ and $\boldsymbol{b}$.

In the case where the number of memory slots is minimal, we have less freedom (in particular we cannot place any additional checkpoints). We can observe that we cannot add checkpoints before LOSS, so that the question becomes how to schedule during the backward phase a set of single chains with different values of $\boldsymbol{L}$ and $\boldsymbol{b}$. During the backward phase, all single chains are independent, but they still share the memory slots. Once it is processed, the initial value of the chain, that was stored in memory can always be discarded. In addition, depending on the value of $\boldsymbol{b}$, the last backward value has to be kept into memory or not. In the case of the toy example, processing the blue chain, where no intermediate forward value has been stored, releases 2 memory slots while the red and green chains (both of length 3), release only one memory slot. On the toy example, the blue-green-red ordering to finish the forward phase is optimal.

Overall, solving this toy example requires to use all the main results presented in this chapter, and whose proofs can be found in the next sections. More specifically, Section 1.2.2 presents the characterization of the optimal solution that is later used in Theorem 3 proved in Section 1.3.2 to build the general dynamic programming formulation.

1.2.2 Canonical Form of Optimal Solutions

In this section, we show that there exist optimal solutions to MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ that follow a canonical form. In Section 1.3, we later use this caracterization to compute an optimal solution. We start by giving a formal description of a valid schedule, before showing our result.

	Operation	Input	Output	Cost
$F_i^{(j)}$	Executes one forward step for $i \in 1, \dots, K$ and $j \in \{1, \dots, \boldsymbol{L}_i\}$	$\{a_{i-1}^{(j)}\}$	$\{a_i^{(j)}\}$	u_F
$B_i^{(j)}$	Executes one backward step for $i \in \{1, \dots, K\}$ and $j \in \{1, \dots, \boldsymbol{L}_i\}$	$\{\delta_i^{(j)}, a_{i-1}^{(j)}\}$	$\{\delta_{i-1}^{(j)}\}$	u_B
Loss	Computation of the loss function using the last forward value of each chain	$\{a_{\boldsymbol{L}_1}^{(1)}, \dots, a_{\boldsymbol{L}_K}^{(K)}\}$	$\{\delta_{\boldsymbol{L}_1}^{(1)}, \dots, \delta_{\boldsymbol{L}_K}^{(K)}\}$	u_L
$S_i^{(j)}$	Replicates $a_i^{(j)}$ into the memory as a checkpoint for $i \in 1, \dots, K$ and $j \in 1, \dots, \boldsymbol{L}_i$	$\{a_i^{(j)}\}$	$\{a_i^{(j)}, a_i^{(j)}\}$	0
$D_i^{(j)}$	Discard $a_i^{(j)}$ from memory	$\{a_i^{(j)}\}$	$\varnothing$	0
$\bar{D}^{(j)}$	Discard $\delta_0^{(j)}$ from memory, so after that there are no element of j chain in the memory	$\{\delta_0^{(j)}\}$	$\varnothing$	0

Таблица 1.1: Operations performed by a schedule

1.2.2.1 Valid Schedule

Throughout the text, we represent a schedule by a sequence of operations, where the set of operations ($F_i^{(j)}$, $B_i^{(j)}$, Loss, $S_i^{(j)}$, $D_i^{(j)}$, $\bar{D}^{(j)}$) is defined in Table 1.1. Table 1.1 defines both the set of operations and the transformation of the memory made by each operation. Initially, we assume that all $a_0^{(j)}$ values are stored into memory slots.

Definition 5 (Valid Schedule)**.** A schedule $\mathcal{S}$ is valid if for every operation from the schedule its input is stored in the memory during its execution and this execution does not violate the memory limit m. Moreover, each backward operation $B_i^{(j)}$ for any $1 \leq i \leq \boldsymbol{L}_j$ and $j \in \{1, \dots, K\}$ should be present in $\mathcal{S}$ at least once.

Definition 6 (Makespan of a Schedule)**.** We define the makespan of a valid schedule $\mathcal{S}$ as

$$k_F u_F + k_B u_B + k_T u_L,$$

where k_F denotes the number of $F_i^{(j)}$ operations in $\mathcal{S}$, k_B denotes the number of $B_i^{(j)}$ operations in $\mathcal{S}$ and k_T denotes the number of Loss in $\mathcal{S}$ (we prove in the remainder of this section that in any optimal solution, $k_B = \sum_{i=1}^{K} \boldsymbol{L}_i$ and $k_T = 1$).

Therefore, our model takes into account computational costs and limited memory capacity, but it assumes that memory accesses, both for reading and writing are cheap and can be neglected.

Finally, we introduce the specific makespan of optimal schedules:

Definition 7 ($\mathrm{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$)**.** Given $\boldsymbol{L} \in \mathbb{N}^K$, $m \in \mathbb{N}$, and $\boldsymbol{b} \in \{0,1\}^K$, let $\mathrm{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$ denote the makespan of an optimal solution to MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$

1.2.2.2 Canonical Solutions

Notation 1. Let us consider an arbitrary branch with index $k \in \{1, \dots, K\}$, which has in total n_k number of checkpoints before LOSS whose indices are $i_1, \dots, i_{n_k}$ written in increasing order. The positions of these checkpoints delimit $n_k - 1$ segments $s_1, \dots, s_{n_k-1}$ corresponding to different parts of the execution.

- We denote as $FS_{s_j}^{(k)}$ for $j \in \{1, \dots, n_k - 1\}$ *the $j-th$ Forward Segment of branch k*, which represents the sequence of operations that take place between the checkpointed values $a_{i_j}^{(k)}$ and $a_{i_{j+1}}^{(k)}$:

$$FS_{s_j}^{(k)} = \text{Store } a_{i_j}^{(k)}; \text{ Forward Operations } F_{i_j+1}^{(k)}, F_{i_j+2}^{(k)}, \dots, F_{i_{j+1}-1}^{(k)}, F_{i_{j+1}}^{(k)}$$

- We denote as $BS_{s_j}^{(k)}$ for $j \in \{1, \dots, n_k - 1\}$ *the $j-th$ Backward Segment of branch k*, which represents the sequence of operations performed on branch k after LOSS and $BS_{s_{j+1}}^{(k)}$ between the first time $a_{i_j}^{(k)}$ is reused and the computation of $B_{i_j}^{(k)}$.

Properties 1. *We search the solutions that satisfy the next set of properties.*

C.1 *For any branch $k \in \{1, \dots, K\}$ the first checkpointed value is $a_0^{(k)}$, i.e. $a_{i_1}^{(k)} = a_0^{(k)}$. Indeed, as it is impossible to retrieve $a_0^{(k)}$ once discarded, thus it should be kept in memory since the start of the execution until backward $B_1^{(k)}$.*

C.2 *Having fixed the positions of checkpoints in the forward phase, any permutation of the forward segments, s.t. for all k, j: $FS_{s_j}^{(k)}$ is executed before $FS_{s_{j+1}}^{(k)}$ yields a valid forward phase and does not affect the final makespan.*

C.3 *(Memory Persistence) For all k and j, $a_{i_j}^{(k)}$, which is the input of $FS_{s_j}^{(k)}$, is not discarded from memory until the end of $BS_{s_j}^{(k)}$.*

C.4 *(Non-overlap of backward segments) No backward segment overlaps with the last backward segment, i.e. if the backward segment $BS_{s_1}^{(k)}$ for some k is the last one then all operations from $BS_{s_1}^{(k')}$ for any $k' \neq k$ (the last backward segment of another branch) should finish before $BS_{s_1}^{(k)}$.*

Lemma 1. *There always exists an optimal solution to* MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ *that satisfies Properties 1.*

Доказательство. We prove that Properties 1 do not prevent from finding the optimal solution, using basic transformations. We show that it is always possible to transform an optimal solution into another optimal solution with stronger structural properties.

Property **C.1** is obvious. Property **C.2** is also straightforward. The condition that for all k, j: $FS_{s_j}^{(k)}$ is executed before $FS_{s_{j+1}}^{(k)}$ guarantees that such permutations preserve validity, without changing the makespan or affecting feasibility of all operations scheduled after LOSS.

We can show that all optimal solutions should comply with with Property **C.3**. Let us consider an arbitrary non-persistent solution (violating Property **C.3**), where for some branch k and position i_j, its checkpointed value $a_{i_j}^{(k)}$ is discarded before the execution of $B_{i_j+1}^{(k)}$ (the end of $BS_{s_j}^{(k)}$). This implies that at some moment during the backward phase $a_{i_j}^{(k)}$ is consumed by $F_{i_j+1}^{(k)}$ just to obtain $a_{i_j+1}^{(k)}$, replacing $a_{i_j}^{(k)}$ in the memory. However, storing $a_{i_j+1}^{(k)}$ instead of $a_{i_j}^{(k)}$ during the forward phase would have reduced the total number of recomputations during the backward phase, while not increasing the memory usage (storing any activation costs the same). Thus, the obtained solution is still valid and is better than the initial one, implying that non-persistent solutions are not optimal.

Finally, we can prove by contradiction that Property **C.4** is non-restrictive. Given an optimal solution, assume that for some k the backward segment $BS_{s_1}^{(k)}$ finishes last and overlaps with other segments. Let $BS_{s_1}^{(k')}$ for some $k' \neq k$ (the last backward segment of some other branch) be the backward segment that finishes one before the last one and thus overlaps $BS_{s_1}^{(k)}$. Note the case when $k' = k$ is not possible as the backward segments of the same branches cannot overlap.

If we consider the almost identical solution where we move all operations from $BS_{s_1}^{(k)}$ after the execution of $B_1^{(k')}$, in the same order. Then one can verify that:

- No more segment overlaps with $BS_{s_1}^{(k)}$;
- The schedule after the transformation is still valid:
  - Dependencies for backward segments are not violated, as after Loss all branches are processed independently and the order of operations is preserved;
  - Memory available for $BS_{s_1}^{(k)}$ can only increase, as the memory reserved for other branches is released;
  - Memory available for other branches should not decrease: before and after this transformation $a_0^{(k)}$ is in the memory during the previous segments, while after it is the only activation from $BS_{s_1}^{(k)}$ still kept in memory.

In the end, this shows that we can transform any solution into a new solution with identical forward phase and Loss, but with strictly fewer backward segment overlapping.

□

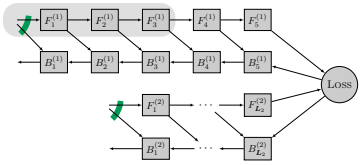
Now we define *Canonical Solutions*, that have a structure that satisfy Properties 1.

Notation 2. Given $\boldsymbol{L} = (\boldsymbol{L}_1, \ldots, \boldsymbol{L}_K)$, we denote by $\boldsymbol{L}_{[i \leftarrow x]}$ the vector $\boldsymbol{L}$ where the i-th element is replaced by the value x:

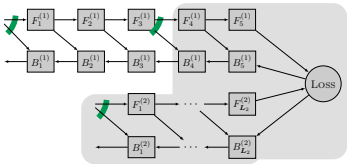
$$\boldsymbol{L}_{[i \leftarrow x]} = (\boldsymbol{L}_1, \ldots, \boldsymbol{L}_{i-1}, x, \boldsymbol{L}_{i+1}, \ldots, \boldsymbol{L}_K)$$

Definition 8 (Canonical Solutions)**.** We say that a solution $\mathcal{S}$ to Multi-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ is in a canonical form if there exists j, k, $\mathcal{S}'$ a canonical solution to Multi-$\boldsymbol{b}(\boldsymbol{L}_{[k \leftarrow \boldsymbol{L}_k - j]}, m - 1, \boldsymbol{b}_{[k \leftarrow 1]})$, and $\tilde{\mathcal{S}}$ a solution to AdjChain$(j-1, m - \sum_{j \neq k} \boldsymbol{b}_j)$, such as $\mathcal{S}$ has the following form:

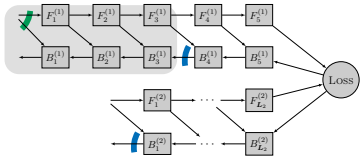
1. Replicate the input $a_0^{(k)}$ in the memory;
2. From the input $a_0^{(k)}$, j forward steps are performed on the branch k;



(a) The solution starts from a state where all inputs are stored; then it performs some forward steps on one of the branches.



(b) After this, it stores the current state and solves a smaller instance of Multi-$\boldsymbol{b}$()



(c) Finally, some backward data are stored (depending on $\boldsymbol{b}$), and an instance of AdjChain(ℓ, m) is solved.

Рис. 1.7: Graphical execution of a canonical solution

3. $\mathcal{S}'$ is performed (i.e. a canonical solution for the smaller problem where all sizes of branches are the same except for branch k, which is now smaller by j forward steps with $m-1$ memory slots).
4. From the input $a_0^{(k)}$ written on memory, $\tilde{\mathcal{S}}$ is performed (the j subsequent steps on branch k are *backpropagated* with all available checkpoints).
5. If $\boldsymbol{b}_k = 0$, $\delta_0^{(k)}$ is discarded.

The example provided in Figure 1.6 is in canonical form.

Theorem 2. *There exists an optimal solution under canonical form.*

Доказательство. The result can be shown by induction. The initialization when $\boldsymbol{L} = \vec{0}$ is trivial.

Let us assume, that for $\boldsymbol{L} > \vec{0}$, we are given an optimal solution $\mathcal{S}$ to Multi-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ that satisfy Properties 1. Suppose that $k' \in \{1, \dots, K\}$ is the branch whose backward $B_1^{(k')}$ is the last in the execution. In addition, we know all values of this branch checkpointed before Loss, given by the set of indices $\{i_1, i_2, \dots, i_{n_{k'}}\}$ in increasing order. According to **C.4**, sequence $\mathcal{S}$ is a solution where the last backward segment, $BS_{s_1}^{(k')}$ for some $k' \in \{1, \dots, K\}$, does not overlap with any other backward segment. Furthermore, according to **C.2**, we can choose that its forward phase starts with $FS_{s_1}^{(k')}$.

Finally, it suffices to observe that:

- At the beginning of $BS_{s_1}^{(k')}$, the last backward data that has been computed so far on chain k is exactly $\delta_{i_2}^{(k')}$;
- During $BS_{s_1}^{(k')}$, there is no operations from other branches or operations on layers after i_2 (because of the no overlap property during $BS_{s_1}^{(k')}$).
- At the beginning of $BS_{s_1}^{(k')}$ exactly $m - \sum_{j \neq k'} \boldsymbol{b}_j$ memory slots are available.

Hence $BS_{s_1}^{(k')}$ is exactly a schedule that solves AdjChain$((i_2 - 1, m - \sum_{j \neq k'} \boldsymbol{b}_j)$

We now have an optimal solution that has the following shape:

1. Replicate the input $a_0^{(k')}$ (the first operation of $FS_{s_1}^{(k')}$)
2. From the input $a_0^{(k')}$ written on memory, i_2 forward steps are performed on the branch k';

3. A schedule $\mathcal{S}'$ is performed.
4. From the input $a_{i_1}^{(k')} = a_0^{(k')}$ written in memory, we perform $BS_{s_1}^{(k')}$ a solution to ADJCHAIN$((i_2 - 1, m - \sum_{j \neq k'} \boldsymbol{b}_j)$.
5. If $\boldsymbol{b}_{k'} = 0$, $\delta_0^{(k')}$ is discarded.

Note that by construction, $\mathcal{S}'$ is an optimal solution to MULTI-$\boldsymbol{b}(\boldsymbol{L}_{[k' \leftarrow \boldsymbol{L}_{k'} - i_2]}, m - 1, \boldsymbol{b}_{[k' \leftarrow 1]})$, and by induction hypothesis, we can transform it into a canonical optimal solution to the same problem. Setting $k = k'$ and $j = i_2$ completes the proof. □

1.3 Optimal Solution for Problem MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$

1.3.1 Minimal Memory Requirement

In this section, we first establish in Lemma 2 the formula to compute the minimal number of memory slots in order to complete multiple chains with parameters $\boldsymbol{L}$ and $\boldsymbol{b}$. This result is later used to solve MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ in Theorem 3, since it is used to initialize the dynamic program.

Lemma 2 (Minimal required memory)**.** *The minimal amount of memory slots to solve* MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$*, is:*

$$m_{min}(\boldsymbol{L}) = \begin{cases} \tilde{K} & \text{if } (\exists i : \boldsymbol{L}_i = 1) \text{ or } (\exists i : \boldsymbol{L}_i = 0 \text{ and } \boldsymbol{b}_i = 0) \text{ or } \boldsymbol{L} = \vec{0}, \\ \tilde{K} + 1 & \text{otherwise.} \end{cases}$$

where $\tilde{K} = K + \sum_{i=1}^{K} \mathbb{I}[\boldsymbol{L}_i \neq 0]$.

Доказательство. The case where $\boldsymbol{L} = \vec{0}$ is trivial because we simply need to perform LOSS, which has a memory peak of K: its K inputs are transformed into K outputs.

For the general case, we decompose the memory peak as a function of the phase:

Forward phase: During the forward phase, all initial inputs of all branches need to be stored ($a_0^{(i)}$ values in Figure 1.3) because they are needed for the computation of $B_1^{(i)}$. In addition, at the end of the forward phase, all inputs of LOSS ($a_{\boldsymbol{L}_i}^{(i)}$ values) are also stored. Hence, the forward phase needs at least:

$$\tilde{K} = K + \sum_{i=1}^{K} \mathbb{I}[\boldsymbol{L}_i \neq 0]$$

as we use only one slot when $a_0^{(i)} = a_{\boldsymbol{L}_i}^{(i)}$.

It can be verified that at any time in the forward phase we do not need more storage space than this: during the execution of $F_j^{(i)}$ we can assume that its input uses the storage slot of $a_{\boldsymbol{L}_i}^{(i)}$ (unused at first) and replaces it by its output. Hence $\tilde{K}$ is enough for the forward phase.

Loss function: The computation of LOSS does not increase the storage needed from what was used before, indeed it simply replaces each of its input values (already stored by

hypothesis) by the corresponding output value. Finally at the end of Loss, we release the checkpoints from branches that are finished, *i.e.* those where $\boldsymbol{L}_i = 0$ and $\boldsymbol{b}_i = 0$. Hence, at the end we have $\hat{K} = \tilde{K} - \sum_{i=1}^{K} \mathbb{I}[\boldsymbol{L}_i = 0 \ \& \ \boldsymbol{b}_i = 0]$ available memory slots.

Backward phase: Finally, consider the solution that executes each branch one after the other during the backward phase. In this case, we need to perform K independent Revolve procedures on branches with lengths $\boldsymbol{L}_i - 1 \geq 0$ for any i. The peak is when we compute the first backward data: one uses $\hat{K} - 2$ checkpoints for all branches but the current one, while the current one requires at least 3 memory slots (resp. 2 memory slots if $\boldsymbol{L}_i - 1 = 0$) in addition to that (see Theorem 1). Hence, if $\hat{K} < \tilde{K}$ or $\exists\ i$, $\boldsymbol{L}_i = 1$, the peak is met during the forward phase with $\tilde{K}$ storage slots.

In the general case ($\hat{K} = \tilde{K}$ and for all i, $\boldsymbol{L}_i > 1$), it is not possible to free memory, and as no backward steps can be performed because of lack of inputs, the minimal required memory is $\tilde{K} + 1$. Hence this shows the result. □

1.3.2 Optimal Solution

Notation 3. We denote by $\mathbf{e}_i$ the vector whose every element is null except the i-th one that is equal to 1. Note that, with this notation, the size of $\mathbf{e}_i$ is ambiguous but it will be the same as the other vectors ($\boldsymbol{L}$ and $\boldsymbol{b}$) in the rest of this chapter.

We are now able to establish the following theorem on the execution time of an optimal solution of Multi-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ (and hence Multi-$\emptyset(\boldsymbol{L}, m)$):

Theorem 3 (Optimal execution time, $\mathrm{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$)**.** *Given a join DAG $\mathcal{G}_K$ with K branches of lengths $\boldsymbol{L}$, given m memory slots and a vector $\boldsymbol{b}$. The execution time $Opt_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$ of an optimal solution to* Multi-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ *is given by*

$$\begin{aligned}
Opt_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b}) &= \infty && \text{if } m < m_{\min}(\boldsymbol{L}),\\
Opt_{\text{b-multi}}\left(\vec{0}, m, \boldsymbol{b}\right) &= u_L && \text{if } m \geq K,\\
Opt_{\text{b-multi}}(\boldsymbol{e}_i, m, \boldsymbol{b}) &= u_F + u_L + u_B && \forall i,\ \text{if } m \geq K+1,
\end{aligned}$$

For other cases:

$$Opt_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b}) = \min_{\substack{1 \leq k \leq K \\ 1 \leq i \leq \boldsymbol{L}_k}} \Bigg[i u_f + Opt_{\text{b-multi}}\left(\boldsymbol{L}_{[k \leftarrow \boldsymbol{L}_k - i]}, m-1, \boldsymbol{b}_{[k \leftarrow 1]}\right) + Opt_0\left(i-1, m - \sum_{j \neq k} \boldsymbol{b}_j\right)\Bigg] \quad (1.4)$$

The complexity to compute $Opt_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$ is $O\left(mK(2L)^{K+1}\right)$ where $L = \max_{i=\{1,\ldots,K\}} \boldsymbol{L}_i$.

In practice, K is finite and small, *e.g.* the neural networks that we consider have $K = 2$ or 3.

Доказательство. We have shown in the previous section that the optimal canonical solution is an optimal solution. Hence here it suffices to show:

- That there exists a solution with execution time $\text{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$;
- That $\text{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$ gives the best execution time among any canonical solution for identical parameters.

We remind the recursive shape of a canonical solution $\mathcal{S}$ for MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ provided that some branch $k \in \{1, \dots, K\}$ is the first branch to be processed and i forward steps for some $i \in \{1, \dots, \boldsymbol{L}_k\}$ are performed until the next data checkpointing (data replication):

1. Replicate the input data (done with a zero execution cost);
2. Execute i forward steps on branch k;
3. Execute a canonical solution $\mathcal{S}'$ for MULTI-$\boldsymbol{b}(\boldsymbol{L}_{[k \leftarrow \boldsymbol{L}_k - i]}, m-1, \boldsymbol{b}_{[k \leftarrow 1]})$
4. Find a sequence $\mathcal{S}''$ for ADJCHAIN$(i-1, \tilde{m})$ on the last steps of branch k using all available checkpoints $\tilde{m}$, execute $\mathcal{S}''$.

If we denote by $T(\mathcal{S})$ (resp. $T(\mathcal{S}')$ and $T(\mathcal{S}'')$) the execution time of $\mathcal{S}$ (resp. $\mathcal{S}'$ and $\mathcal{S}''$), then we have:

$$T(\mathcal{S}) = iu_f + T(\mathcal{S}') + T(\mathcal{S}'')$$

Existence of a solution of time $\text{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$: Based on this, one can see that we can indeed reconstruct recursively a solution based on Eq. (1.4). The initialization when $\boldsymbol{L} = \vec{0}$ or $\boldsymbol{L} = \mathbf{e}_i$ is trivial.

For the fixed values of i and k, Eq. (1.4) transforms into the next expression:

$$\text{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b}) = \left[iu_f + \text{Opt}_{\text{b-multi}}\left(\boldsymbol{L}_{[k \leftarrow \boldsymbol{L}_k - i]}, m-1, \boldsymbol{b}_{[k \leftarrow 1]}\right) + \text{Opt}_0\left(i-1, m - \sum_{j \neq k} \boldsymbol{b}_j\right) \right].$$

By induction hypothesis, $\text{Opt}_{\text{b-multi}}\left(\boldsymbol{L}_{[k \leftarrow \boldsymbol{L}_k - i]}, m-1, \boldsymbol{b}_{[k \leftarrow 1]}\right)$ is the execution time of a solution $\mathcal{S}'$ to MULTI-$\boldsymbol{b}(\boldsymbol{L}_{[k \leftarrow \boldsymbol{L}_k - i]}, m-1, \boldsymbol{b}_{[k \leftarrow 1]})$, $\text{Opt}_0\left(i-1, m - \sum_{j \neq k} \boldsymbol{b}_j\right)$ is the execution time of REVOLVE for the last i steps on branch k, and $\text{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$ corresponds to the execution time of the canonical schedule:

- Perform i steps on branch k;
- Perform $\mathcal{S}'$;
- Use REVOLVE to backpropagate optimally the last $i-1$ steps of branch k.

Minimality of $\text{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$ amongst optimal solutions Similarly we show this result by induction. We only consider valid solutions, hence where the memory requirement is met.

The initialization is performed either for $\boldsymbol{L} = \vec{0}$ or $\boldsymbol{L} = \mathbf{e}_i$. For both those cases there is only one possible canonical solution and its execution time is exactly $\text{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$. Assume now that for all $\boldsymbol{b}$, and $\boldsymbol{L} < \boldsymbol{L}'$, $m \geq m_{min}(\boldsymbol{L})$ any canonical solution $\mathcal{S}$ to MULTI-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$ is such that $T(\mathcal{S}) \geq \text{Opt}_{\text{b-multi}}(\boldsymbol{L}, m, \boldsymbol{b})$.

For any given $\boldsymbol{b}'$, $m' \geq m_{min}(\boldsymbol{L}')$, we study a canonical solution $\mathcal{S}'$ to MULTI-$\boldsymbol{b}(\boldsymbol{L}', m', \boldsymbol{b}')$. Then by definition there exists k, i', $\tilde{\mathcal{S}}$ a solution to

MULTI-$\boldsymbol{b}(\boldsymbol{L}_{\left[k \leftarrow \boldsymbol{L}'_k - i'\right]}, m' - 1, \boldsymbol{b}_{[k \leftarrow 1]})$ and $\hat{\mathcal{S}}$ a solution to ADJCHAIN$(i' - 1, m' - \sum_{j \neq k} \boldsymbol{b}_j)$ and

$$
\begin{aligned}
T(\mathcal{S}') &= i' u_F + T(\tilde{\mathcal{S}}) + T(\hat{\mathcal{S}}) \\
&\geq i' u_F + \mathrm{Opt}_{\text{b-multi}}\left(\boldsymbol{L}_{\left[k \leftarrow \boldsymbol{L}'_k - i'\right]}, m' - 1, \boldsymbol{b}_{[k \leftarrow 1]}\right) + \mathrm{Opt}_0\left(i' - 1, m' - \sum_{j \neq k} \boldsymbol{b}_j\right) \\
&\geq \min_{\substack{1 \leq k \leq K \\ 1 \leq i' \leq \boldsymbol{L}_k}} \left[i' u_F + \mathrm{Opt}_{\text{b-multi}}\left(\boldsymbol{L}_{\left[k \leftarrow \boldsymbol{L}'_k - i'\right]}, m' - 1, \boldsymbol{b}_{[k \leftarrow 1]}\right) + \mathrm{Opt}_0\left(i' - 1, m' - \sum_{j \neq k} \boldsymbol{b}_j\right) \right] \\
&= \mathrm{Opt}_{\text{b-multi}}\left(\boldsymbol{L}', m', \boldsymbol{b}'\right),
\end{aligned}
$$

which proves the result.

□

Remark. Note that for clarity we have used in this proof any vector $\boldsymbol{b}$ (which adds a factor of $O(2^K)$ to the complexity). To reduce the complexity, we can permute in the dynamic program the branch that has been chosen so that $\boldsymbol{b}$ is always sorted. Hence, we simply need to solve $O(K)$ different dynamic programs. The complexity would then be $O(mK^2(L)^{K+1})$ with $L = \max_i \boldsymbol{L}_i$. However, solving MULTI-$\delta(\boldsymbol{L}, m)$ does not need $\boldsymbol{b}$, as by default the output gradients of adjoint computation are not discarded, thus the problem is even simpler, requiring only $O(mKL^{K+1})$ number of steps.

In addition, this proof also provides a way to compute the optimal solution based on the computation of its execution time.

1.4 Simulation Results

In this section, we depict the results of simulations on three graph structures inspired by neural networks such as Cross-Modal embeddings (CM), Siamese networks with triplet loss and Recurrent Neural Networks (RNNs). From these neural networks we take only data dependencies, while we assume that all computational costs ($u_F = u_B = u_L = 1$) and storage costs are homogeneous, even if the actual neural networks can be heterogeneous. In this context, a multi chain network is completely defined by the lengths of the different branches and the size of the memory expressed in terms of storage slots.

We propose two observations:

1. The trade-off Makespan vs Memory usage;
2. For a fixed number of storage slots, an observation on the growth of the number of recomputations needed as a function of the number of forward operations.

In order to compare the different types of networks in a normalized way, we consider several models with analogous sizes and computational costs. For a length L, we consider three graph structures:

- Cross-Modal (CM) embedding networks with two chains of sizes $(L, 5L)$. Those CM process two different types of data (*e.g.* images and texts) and one of the chains is much longer than the other because more layers are needed to extract useful patterns (e.g. images that contain more information are usually processed with deep neural networks while text can be efficiently encoded into vectors of smaller dimensions with shallow neural networks instead). Thus, as soon as the memory m is larger than $M_{CM}(L) = 6L + 2$, then the makespan is minimal and equal to $\text{Span}^*_{CM}(L) = 12L + 1$, which corresponds to the situation where all activations are stored during the forward phase of the training.
- Siamese Neural Networks (SNN) with 3 chains of lengths $(2L, 2L, 2L)$. Here also, L can be large as these neural networks are usually applied to images, thus deep neural networks are also possible as they are better in pattern retrieval. Analogously to the case of CMs, as soon as $m \geq M_{SNN}(L) = 6L + 3$, all forward activations can be stored and therefore the makespan is minimal and equal to $\text{Span}^*_{SNN}(L) = 12L + 1$.
- Finally, we also consider the case of a single chain of length $6L$ associated with Recurrent Neural Networks (RNN). Analogously to the case of CMs of SNNs, as soon as $m \geq M_{RNN}(L) = 6L+1$, all forward activations can be stored and therefore the makespan is minimal and equal to $\text{Span}^*_{RNN}(L) = 12L + 1$. This case also serves as a lower bound on the makespan that can be reached by the previous models.

In the following, we denote by $\text{Span}^*(L) = 12L + 1$ the minimal makespan for all models.

Trade-off Memory – Makespan

The first question that we consider is, given a fixed number of forward operations (described by L for all scenarios), how does the makespan evolves as a function of the number of memory slots available.

We run our experiments for $L = 5$, 10 and 15. The plots depicting the evolution of makespan with the amount of memory are gathered in Figure 1.8. Specifically, the x-axis represents the fraction of memory slots available with respect to the minimal number M_{Opt} (which differs slightly depending on the model) to achieve minimal makespan, $\text{Span}^*(L)$. The y-axis represents the ratio between the achieved makespan and $\text{Span}^*(L)$. Thus, point (x, y) on the CM plot means that for a CM network of length $(5L, L)$, with $x \times M_{CM}$ memory slots, the makespan is $y \times \text{Span}^*(L)$.

The plots related to CM (resp. SNN) start from memory size 5 (resp. 7), which is exactly the value of $m_{\min}$ for two (resp. three) chains, as proved in Lemma 2. We can notice that the makespan first significantly decreases with the first additional memory slots. In addition, it seems that once it reaches a threshold $k \cdot \text{Span}^*$ with $k \simeq 1.5$, the makespan ratio linearly decreases to Span^* with the number of additional memory slots.

Hence, this shows that this checkpointing strategy is very efficient in decreasing the memory needs while only slightly increasing the makespan. For instance, consider the point where $m/M_{\text{Opt}} = 0.5$. This corresponds to halving the memory needs with respect to what is needed to achieve minimal makespan. In all cases, halving memory needs only induces an increase of approximatively 25% on the makespan. In addition, we can also observe that even with a very small memory (say $m_{\min} + 2$), the makespan is less than doubled compared to Span^*.

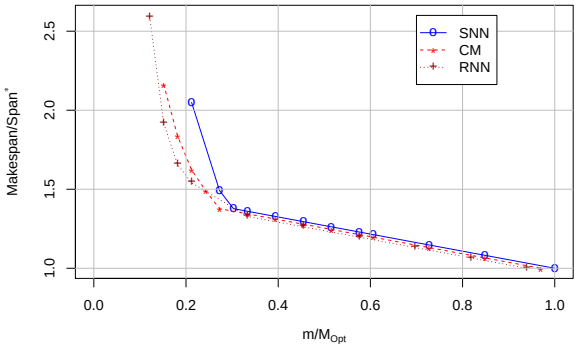
Makespan evolution for fixed memory

In Figure 1.9, we depict the dual situation, where the number of memory slots is fixed on each plot (either 7, 9, 11 or 13) and the ratio of the achieved makespan over Span^* is depicted. Several observations can be made. The first one is that the gap to the lower-bound (RNN) is rather small and decreases with the number of available checkpoints. This gap increases slowly with the size of the model. The exception is when the number of available checkpoints is exactly the minimum number of memory checkpoints ($m = 7$, SNN). This exception is not surprising given the observations of the previous paragraph and the important improvements in performance when the number of available checkpoint is slightly greater than $m_{\min}$. In addition, it is interesting to observe that for SNN and CNN the ratio follows a pattern similar to that of RNN: each curve contains different thresholds, and between those thresholds there is a performance shaped as $\alpha - \beta/L$. Indeed, for the case of RNN those performance have been proven via a closed-form formula [39]. This can motivate the search for a similar closed-form formula for the problem of chains, and with this an algorithm whose complexity is polynomial in the number of branches. Finally, another observation is that the overall growth for a fixed number of checkpoints is quite slow even with few checkpoints, hence encouraging the use of these strategies. Indeed, we can observe that for a relatively small number of checkpoints such as $m = 11$, the ratio to the optimal makespan consistently remains below 2.

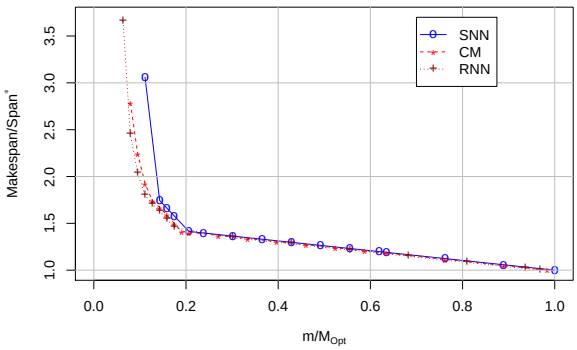
1.5 Conclusion

Being able to perform learning in deep neural networks is a crucial operation, which requires important memory needs in the feed-forward model, because of the storage of activations. These memory constraints often limit the size and therefore the accuracy of used models. The Automatic Differentiation community has implemented checkpointing strategies, which make it possible to significantly reduce memory requirements at the cost of a moderate increase in computing time. Backpropagation schemes are very similar in the cases of Automatic Differentiation and Deep Learning. Nevertheless, the diversity of task graphs (ResNet, Inception and their combinations) is much more important in the context of DNNs. It is therefore crucial to design checkpointing strategies for backpropagation on a much broader class of graphs than the homogeneous chains on which the literature on Automatic Differentiation has focused.

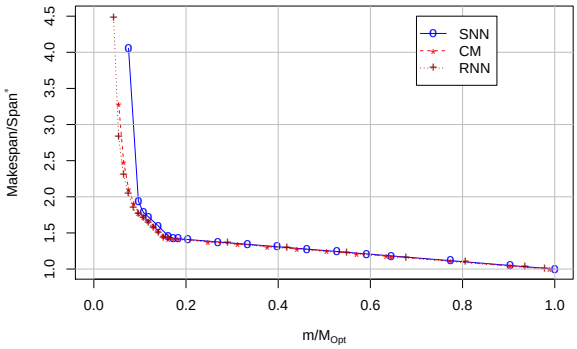
The goal of the chapter is to extend the graph class, by considering multiple parallel chains that meet when calculating the loss function. This class of graphs corresponds to more realistic neural networks such as Cross Modal networks and Siamese Neural Networks. In the case of multi-chain, we were able to build a dynamic program that allows us to compute the optimal strategy given a constraint on the size of the memory, *i.e.* a strategy that fulfills the memory constraint while minimizing the number of recalculations.



(a) $L = 5$

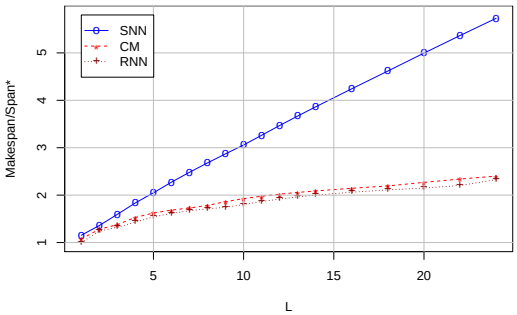


(b) $L = 10$

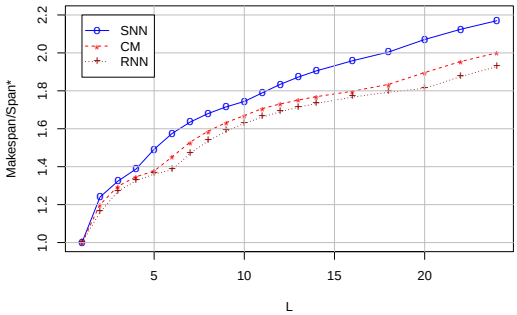


(c) $L = 15$

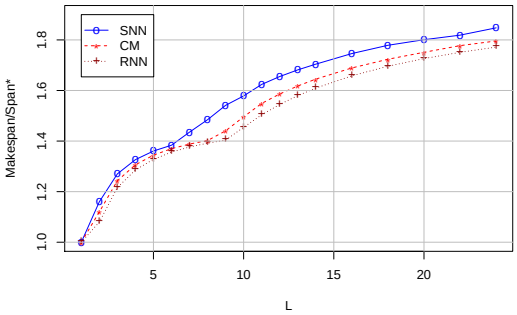
Рис. 1.8: Makespan evolution with respect to different amount of total memory.



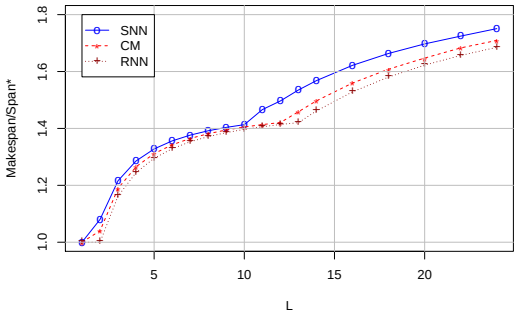
(a) $m = 7$



(b) $m = 9$



(c) $m = 11$



(d) $m = 13$

Рис. 1.9: Makespan evolution with respect to different l for fixed memory size c.

Глава 2

Rematerialization for Heterogeneous Chains

2.1 Modeling and Problem Formulation

In the previous chapter, we introduced various problems with the objective to find optimal checkpointing strategies for different types of adjoint computations. Within the context of Automatic Differentiation, the studies mostly focused on simple homogeneous chains [37, 39]. As it was observed in [20, 42], such checkpointing techniques can be applied as well to decrease memory consumption when training neural networks. Checkpointing in the learning community is usually denoted as *Rematerialization* or *Gradient Checkpointing*, which is done to avoid confusion since checkpointing in Deep Learning is similar to the notion of checkpointing in HPC, where one saves the current state of the computation to be able to restore it later.

Alternatively, neural networks are in general characterized with more complex computational graphs and heterogeneous costs of operations. Problems Multi-$\delta(\boldsymbol{L}, m)$, Multi-$\emptyset(\boldsymbol{L}, m)$ and Multi-$\boldsymbol{b}(\boldsymbol{L}, m, \boldsymbol{b})$, discussed in the previous chapter, attempted to go towards more general DAGs by targeting K-Join chains, which represent data dependencies of specific neural networks such as SNN or CM. However, these problems still considered homogeneous costs. Going to heterogeneous costs may significantly increase the complexity of the algorithm, but it may be necessary for finding optimal schedules in the context of the DNNs.

In this chapter, we analyze the impact of having heterogeneous costs in the model, while considering simple adjoint chain structure. Our contribution generalizes the previously known results for homogeneous adjoint chains. In addition, we propose a new model that provides a better description of operations and data dependencies imposed by learning frameworks like PyTorch [82] or TensorFlow [2].

2.1.1 Adjoint Computation Graphs for Deep Neural Networks

We begin with a presentation of the computation model used throughout the chapter to describe different rematerialization strategies that can be used during an iteration of the backpropagation algorithm. Let us consider a chain of L stages (*i.e.* layers or blocks of layers), numbered from 1 to L. Each stage ℓ is associated both to a forward operation F_ℓ and a backward operation B_ℓ (see Figure 2.1a). We denote by a_ℓ the activation tensor output of F_ℓ and by $\delta_\ell = \frac{\partial \text{Loss}}{\partial a_\ell}$ the backpropagated intermediate value provided as input of the backward operation B_ℓ. In the previous chapter, the switch between forward propagation and backward propagation happened during Loss computation. For notational convenience in this chapter we decompose Loss into F_{L+1} (computing $\text{Loss}(a_L)$) and B_{L+1} (computing $\frac{\partial \text{Loss}}{\partial a_L}$).

To provide a better intuition, let us consider Multi-Layer Perceptron. It consists of several Fully Connected (FC) layers, each FC layer is followed with some non-linear activation function σ. In addition, FC layer ℓ is parametrized with weights W_ℓ (matrix kernel) and b_ℓ (bias) and it operates in accordance with the following forward and backpropagation equations:

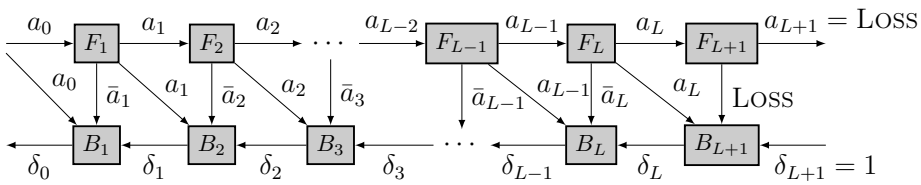
$$
\begin{aligned}
F_\ell &: a_\ell = \sigma(z_\ell) = \sigma(W_\ell a_{\ell-1} + b_\ell) \\
B_\ell &: \delta_{\ell-1} = (W_\ell)^T(\delta_\ell \odot \sigma'(z_\ell)) \\
&\frac{\partial \text{Loss}}{\partial W_\ell} = a_{\ell-1}(\delta_\ell \odot \sigma'(z_\ell)) \\
&\frac{\partial \text{Loss}}{\partial b_\ell} = \delta_\ell \odot \sigma'(z_\ell)
\end{aligned}
$$

where z_ℓ is the pre-activation vector (*i.e.* $a_\ell = \sigma(z_\ell)$). For complex blocks of layers (*e.g.* Inception modules or residual blocks), F_ℓ and B_ℓ are more complex functions that can be expressed as

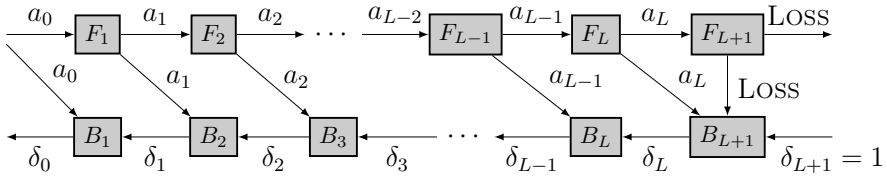
$$
\begin{aligned}
F_\ell &: a_\ell = f_\ell(\theta_\ell, a_{\ell-1}) \\
B_\ell &: \delta_{\ell-1} = \bar{f}_\ell(\theta_\ell, \delta_\ell, \bar{a}_\ell, a_{\ell-1}) \\
&\frac{\partial \text{Loss}}{\partial \theta_\ell} = \bar{g}_\ell(\delta_\ell, \bar{a}_\ell, a_{\ell-1}),
\end{aligned}
$$

where θ_ℓ is the whole set of parameters of the block and $\bar{a}_\ell$ is the set of all intermediate activation values that are required to perform the backward of the block, including a_ℓ but excluding $a_{\ell-1}$ (in the simple case of the FC layer we have $\bar{a}_\ell = \{a_\ell, z_\ell\}$). In classical implementations of the backpropagation algorithm in learning frameworks, all activation values are stored in memory during the forward step F_ℓ until the backward step B_ℓ is completed. In PyTorch, for example, the whole computational graph leading to a_ℓ is stored (this allows the autograd mechanism to build the graph for gradient computation).

The model described in Figure 2.1a corresponds well to what autograd frameworks such as PyTorch and TensorFlow provide. There are additional down paths in Figure 2.1a that are not present in the classical problems of Automatic Differentiation literature (see



(a) Graph for a general sequential deep neural network.



(b) Graph for an automatic differentiation application.

Рис. 2.1: Graphs of a general sequential Deep Neural Network and an Automatic Differentiation application.

Figure 2.1b). Hence, adjoint computations for DNNs are obtained with another type of BP-transform.

Definition 9 (Backpropagation transformation for complex nodes (Complex BP-transform))**.** Given a DAG $\mathcal{G}$ with a single sink node. The *Complex BP-transform* of $\mathcal{G}$ is defined as follows:

1. Build the dual graph $\tilde{\mathcal{G}}$ defined as the same graph where all edges are inversed.
2. For a given node in $\mathcal{G}$, connect its input edges to its dual node in $\tilde{\mathcal{G}}$.
3. Add output edges between the nodes of $\mathcal{G}$ and their corresponding dual nodes in $\tilde{\mathcal{G}}$

Let us notice that Complex BP-transform is able to describe any kind of a neural network that can be approximated with a sequential chain, the nodes of which may be arbitrarily complex (a node of such a chain could be a DAG). In this case, if F_ℓ corresponds to a complex operation that consists of several elementary operations, then $\bar{a}_\ell$ contains the output of F_ℓ and all the hidden activations produced by the internal elementary operations of layer ℓ.

2.1.2 Rematerialization Operations and Memory Usage

The basic principle of Rematerialization is to trade memory for computing time by saving only certain activations in memory and then "rematerializing" the others from the stored ones when the backward steps require them. For example, to implement this strategy in PyTorch, it is required to use three different types of forward operations.

- $F_\ell^\varnothing$ computes F_ℓ without saving any data in memory. It is equivalent to calling a forward operation under `no_grad()` in PyTorch:

```
with torch.no_grad():
    x = F[i](x)
```

- F_ℓ^{ck} computes F_ℓ while *saving the input* $a_{\ell-1}$ of the block of layers ℓ. Similarly to the previous case, it can be implemented in PyTorch by calling a forward under `no_grad()`, while saving in a separate tensor the input:

```
with torch.no_grad():
    y = F[i](x)
```

- F_ℓ^{all} computes F_ℓ while saving all the intermediate data $\bar{a}_\ell$ required by the backward step (*i.e. saving all*). This involves executing a forward operation under `enable_grad()` context, which tells to PyTorch to *record* this operation, keeping track of each elemental computational step and all intermediate outputs:

```
with torch.enable_grad():
    y = F[i](x)
```

Let us notice that the cheapest operation in terms of memory is $F_\ell^{\varnothing}$, but it does not save anything, thus no backward or recomputation can be performed from $a_{\ell-1}$. F_ℓ^{ck} is more costly in terms of memory and it allows forward recomputations from $a_{\ell-1}$, but not B_ℓ. F_ℓ^{all} is the most memory expensive operation, however it is the only one that generates the proper output necessary for performing B_ℓ. As F_ℓ^{all} is consuming a lot of memory, it may be more efficient to compute F_ℓ^{ck} first and then compute F_ℓ^{all} from $a_{\ell-1}$ later in the sequence of instructions. The summary of all operations is provided in Table 2.1.

In the following, we assume that the memory needed to store each data item is known (Section 2.4 describes how this information can be automatically measured before starting the actual training of the model). For convenience, we use a_ℓ to denote both a tensor produced by F_ℓ and its respective data size, the same applies also to $\bar{a}_\ell$ and δ_ℓ. In practice, the activation size of layer ℓ is equal to its corresponding gradient size, *i.e.* $\delta_\ell = a_\ell$. Since each stage of the chain can be arbitrarily complex, temporary data may induce a memory peak higher than the size of the sum of its input and output data. Therefore, we also introduce a memory overhead for each operation: we assume that the memory required to compute an operation is the sum of its input and output data plus a memory overhead.

Note that we focus here on reducing the memory used by activations. We assume that the memory required to store model weights, model states (optimizer states) and its gradients of the weights has already been allocated and removed from the initial available memory. This gives the final available memory M_{GPU} that we consider further as a memory limit in all our optimization problems.

We consider that, at the beginning, the memory contains a_0, *i.e.* the input data. The goal is to perform all backward steps until obtaining δ_0. We look for solutions in the form of sequences of operations. The processing of a sequence consists in executing all the operations one after the other, replacing the input of each operation by its output in the memory. The sequence is said *valid* if

- for any operation, its input is present in the memory when processed
- at any moment the memory limit is not violated.

	Operation	Input	Output	Time	Memory overhead
F_ℓ^{all}	Forward and save all	$\{a_{\ell-1}\}$ $\{\bar{a}_{\ell-1}\}$	$\{a_{\ell-1},\ \bar{a}_\ell\}$ $\{\bar{a}_{\ell-1},\ \bar{a}_\ell\}$	u_{F_ℓ}	o_{F_ℓ}
F_ℓ^{ck}	Forward and materialize input	$\{a_{\ell-1}\}$ $\{\bar{a}_{\ell-1}\}$	$\{a_{\ell-1},\ a_\ell\}$ $\{\bar{a}_{\ell-1},\ a_\ell\}$	u_{F_ℓ}	o_{F_ℓ}
$F_\ell^{\varnothing}$	Forward without saving	$\{a_{\ell-1}\}$	$\{a_\ell\}$	u_{F_ℓ}	o_{F_ℓ}
B_ℓ	Backward step	$\{\delta_\ell,\ \bar{a}_\ell,\ a_{\ell-1}\}$ $\{\delta_\ell,\ \bar{a}_\ell,\ \bar{a}_{\ell-1}\}$	$\{\delta_{\ell-1}\}$ $\{\delta_{\ell-1},\ \bar{a}_{\ell-1}\}$	u_{B_ℓ}	o_{B_ℓ}

Таблица 2.1: Generic operations available in DL frameworks.

For example, for $L = 4$ and large enough memory limit (*e.g.* if memory is enough to run forward and backward phases without recomputations), these sequences of operations are valid:

$$F_1^{ck}, F_2^{\varnothing}, F_3^{ck}, F_4^{all}, F_5^{all}, B_5, B_4, F_3^{all}, B_3, F_1^{all}, F_2^{all}, B_2, B_1 \tag{2.1}$$

$$F_1^{ck}, F_2^{\varnothing}, F_3^{ck}, F_4^{ck}, F_5^{all}, B_5, F_4^{all}, B_4, F_3^{all}, B_3, F_1^{ck}, F_2^{all}, B_2, F_1^{all}, B_1 \tag{2.2}$$

$$F_1^{ck}, F_2^{\varnothing}, F_3^{all}, F_4^{ck}, F_5^{all}, B_5, F_4^{all}, B_4, B_3, F_1^{all}, F_2^{all}, B_2, B_1 \tag{2.3}$$

The maximum memory usage of a valid sequence is defined as the peak memory reached at some moment composed of stored data plus the overhead induced by the operation in progress. The computation time of a sequence is the sum of the durations of its operations. The optimization problem is thus, given a memory limit M_{GPU}, to find a valid sequence whose memory usage does not exceed M_{GPU} and whose computation time is minimal.

Problem 6 (Remat(L, M_{GPU}))**.** We want to minimize under a given memory limit M_{GPU} the makespan of the adjoint computation corresponding to the Complex BP-transform of a chain with L layers, *i.e.* minimize time of computing δ_0 from input a_0 given a computational graph in Figure 2.1a and operations in Table 2.1.

Previous approaches The most popular way of doing Rematerialization is based on [20] and it is implemented in PyTorch [1]. It considers only single-pass rematerialization sequences: such sequences may recompute each operation only once. Typically, they divide the computational graph of a DNN into $n \in \mathbb{N}$ sequential segments and during the first forward pass only the inputs of the segments are saved so that to use them during the backward propagation to recompute the activations of this segment and execute all its backward steps at once. For example, Sequence (2.1) represents an example of such a strategy, where the neural network of length $L = 4$ is divided into 3 segments: the first segment includes layers 1 and 2, the second segment comprises just layer 3 and the last segment consists of layer 4 and the loss. In [20], the case of homogeneous chains was considered and they showed that for this case the optimal approach is to consider $n = \sqrt{L}$ segments of equal size.

In contrast, the methods used in AD can perform as much recomputations as needed, exploiting available memory in a more efficient way. This approach is more flexible than single-pass rematerialization, but the solution is more difficult to find. The optimal schedules for homogeneous chains can be found by solving the dynamic program described in Theorem 1 (see Chapter 1 Section 1.1.5). This dynamic programming has been extended to heterogeneous computing costs [40] as well. However, the solutions from AD cannot be directly used for DNN training as they correspond to different types of data dependencies (see Figure 2.1). Moreover, F^{all} operation type is not used in AD modelling, therefore the sequences generated in the context of AD should be modified to include them. To obtain the valid sequences for DNN, one need to add F_ℓ^{all} before B_ℓ for all ℓ, which does not increase the memory consumption, but the recomputation time grows significantly. This transformation yields the sequences like Sequence 2.2.

Sequence 2.3 is different from the others. It includes F^{all} type of forward operations in the middle of the forward propagation that is not directly followed by corresponding backward operations. This type of sequence is the most flexible one and suited for training with learning frameworks such as PyTorch. The previous approaches were not designed to produce this kind of sequences, but we show in Section 2.3 how to obtain them.

2.2 Complexity

Theorem 4. *The associated decision problem of* Remat(L, M_{GPU})*, i.e. if there exists a solution to* Remat(L, M_{GPU}) *whose makespan is not more than T, is NP-complete in the weak sense.*

Доказательство. It is obvious that Remat(L, M_{GPU}) belongs to NP: for a fixed sequence it is easy to simulate its execution (in linear time) and to check if it executes the chain within time T under the memory limit M_{GPU}.

Let us show that Remat(L, M_{GPU}) is NP-hard by reduction from 2-Partition problem [32]. We formulate the 2-Partition problem in the following way: for a set of values $S = \{x_i \in \mathbb{N} | 1 \leq i \leq n\}$, such that $\sum_{i=1}^{n} x_i = 2V$, does there exist two disjoint subsets S_1 and S_2: $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$, such that $\sum_{i \in S_1} x_i = \sum_{i \in S_2} x_i = V$?

From an instance of 2-Partition, we build a neural network whose forward phase is shown in Figure 2.2. The memory occupied by weights is considered to be negligible. Overall, the associated instance of Remat(L, M_{GPU}) can be summarized as follows:

- $L + 1 = 2n + 2$, $M_{\text{GPU}} = 3V$, $U_B = \sum_{i=1}^{L+1} u_{B_i}$, $T = 4V + U_B$;
- $u_{F_{2k-1}} = x_k$ and $a_{2k-1} = x_k$ for $1 \leq k \leq n$;
- $u_{F_{2k}} = 0$ and $a_{2k} = 0$ for $1 \leq k \leq n + 1$;
- $u_{F_{2n+1}} = V$ and $a_{2n+1} = V$;
- Input $a_0 = 0$;
- u_{B_i} for any $i \leq L$ can be any positive value (this proof is valid for any backward durations);
- $\bar{a}_i = \delta_i = a_i$ for all $0 \leq i \leq L + 1$.

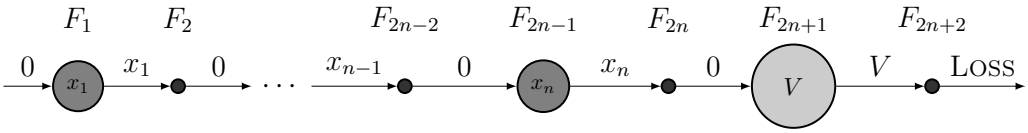


Рис. 2.2: A instance of Remat(L, M_{GPU}), where the values inside the nodes indicate the durations of the task, and the values above the arrows show the data sizes of the output tensors.

We first show that this neural network has a solution with makespan at most $T = 4V + U_B$ if there exists a solution to the 2-Partition problem. Indeed, given a solution (S_1, S_2) to the 2-Partition instance, if we store only the activations with indices a_{2k-1} with $k \in S_1$, then there is enough memory to process B_{2n+1}, which requires at least $\delta_{2n+1} + \bar{a}_{2n+1} + a_{2n} + \delta_{2n} = 2V$ amount of memory. During the backward phase it is necessary to recompute F_{2k-1} for all $k \in S_2$, thus the total duration of the recomputations is $\sum_{k \in S_2} u_{F_{2k-1}} = \sum_{k \in S_2} x_k = V$. As the forward phase lasts $\sum_{k=1}^{n} u_{F_{2k-1}} + V = 3V$, it results in total execution time $T = 4V + U_B$.

However, if there is a schedule with a makespan at most $4V + U_B$ for this problem, then let us show that there exists a solution to the 2-Partition problem. For any schedule, the forward execution time is exactly $3V$, the backward execution is U_B, leaving only time V for recomputations. Let us denote S_2 the subset of activations of non-zero cost such that a_{2k-1} for $k \in S_2$ are discarded during the forward phase and should be recomputed during the backward phase (only the activations with odd indices $2k-1$ for some $1 \leq k \leq n$ can be discarded). Since the recomputation time is at most V, then $\sum_{k \in S_2} u_{F_{2k-1}} = \sum_{k \in S_2} x_k \leq V$.

We now demonstrate that $\sum_{k \in S_2} x_k \geq V$. Indeed, it is not possible to store all activations, otherwise the occupied memory during B_{2n+1} will be $\sum_{i=0}^{2n+1} a_i + \delta_{2n} + \delta_{2n+1} = 4V > 3V = M_{\text{GPU}}$. The minimal memory requirement for B_{2n+1} is at least $2V$ therefore one should discard not less than V amount of data to complete the execution. This implies $\sum_{k \in S_2} a_{2k-1} = \sum_{k \in S_2} x_k \geq V$. Combining both inequalities gives $\sum_{k \in S_2} x_k = V$. Finally, by setting $S_1 = S \setminus S_2$, we obtain a solution to the 2-Partition problem, which completes the proof. □

2.3 Optimal Rematerialization Algorithm

In this section, we carefully analyze Remat(L, M_{GPU}) and the structure of its optimal solutions. Optimality proofs for the homogeneous case (as in AD literature) start by showing that all optimal schedules satisfy a particular property called memory persistence. We introduce this property in Section 2.3.1 and we prove by providing a counter example that there exist some instances of Remat(L, M_{GPU}) that do not have optimal memory persistent solutions.

This remark applies in particular to the model described in Figure 2.1a, but also to the AD model depicted in Figure 2.1b and used in [42], where memory persistence is implicitly used when deriving the dynamic programming solution. Nevertheless, we define in Section 2.3.2 a weaker version of memory persistence called *floating memory persistence* and we prove that this property holds true in the heterogeneous case (there always exists an optimal schedule for REMAT(L, M_{GPU}) that satisfy this property). We demonstrate in Section 2.3.3 that *floating memory persistence* can be used to derive a dynamic programming algorithm providing the optimal solution for REMAT(L, M_{GPU}), which can be applied to both models of Figure 2.1. We also describe in Section 2.3.4 how to compute the optimal solution when assuming that memory persistence holds. The resulting algorithm, although not theoretically optimal, is of crucial practical interest. Indeed, in Section 2.4, we compare the practical performance of algorithms based on the *memory persistence* and *floating memory persistence* assumptions. We demonstrate that for a large class of DNN graphs, there is no difference between the quality of the results obtained under the two assumptions. This shows that in practice, although we prove that the *memory persistence* hypothesis is wrong in general, it can nevertheless be used to generate efficient rematerialization sequences, as we propose in ROTOR (see Section 2.4).

2.3.1 Considerations on Memory Persistence

Properties 2. *The following property is common for adjoint computation problems.*

R.1 *(Memory Persistence) For any ℓ if a_ℓ is saved in memory (by $F^{ck}_{\ell+1}$ or $F^{all}_{\ell+1}$), then it is not discarded until $B_{\ell+1}$.*

If a schedule satisfies Property **R.1** then the schedule is called *memory persistent* or just *persistent* for short. Adjoint computation problems are often restricted to persistent schedules, because it can be proven [107] (there checkpoint persistence) that for homogeneous chains, there always exists an optimal schedule that is persistent. The similar result holds for Multi-Adjoint Chains introduced in Chapter 1. In a persistent schedule, between the moment when some activation a_ℓ is saved into memory and the moment when it is used for $B_{\ell+1}$, there is no operation on the previous layers $\ell' \leq \ell$. It is an important property for adjoint computation problems that facilitates the search of optimal schedules as it allows us to apply checkpointing techniques on the chain in the piecewise manner. A key observation for homogeneous activation sizes is that all optimal schedules are memory persistent: if an activation a_ℓ is stored, but deleted before being used for $B_{\ell+1}$, then it will be actually more efficient to store $a_{\ell+1}$ instead since it avoids to recompute $F_{\ell+1}$, with the exact same memory cost.

However, when activation sizes are heterogeneous, *i.e.* $\exists \ell, \ell' \in \{0, \dots, L+1\}$, *s.t.* $a_\ell \neq a_{\ell'}$, this property no longer holds. We show an example on Figure 2.3 with a chain of length is $L = n + 2$ for any $n \in \mathbb{N}$, where all backward computational costs u_{B_ℓ} are 0, as well as most of the forward computational costs, except $u_{F_1} = k \in \mathbb{N}$ (we further set $k = n - 2$) and $u_{F_2} = 2$. In addition, $a_0 = a_{L+1} = 0$, $a_1 = 1$ and $a_L = 4$, whereas for all other $\ell \neq 0, 1, L, L+1$ we have $a_\ell = 3$. Moreover, for any ℓ, $a_\ell = \bar{a}_\ell = \delta_\ell$. The memory limit is $M_{\text{GPU}} = 15$.

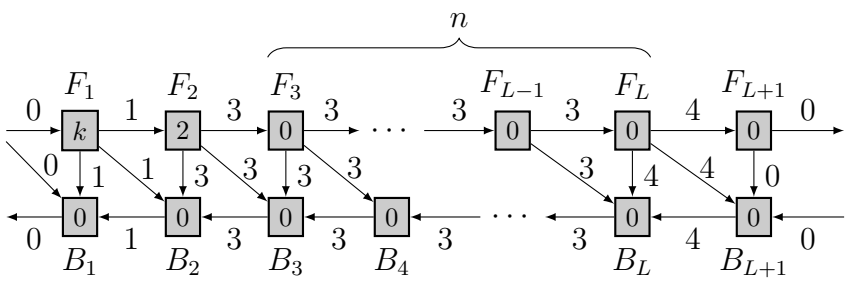


Рис. 2.3: Counter example where no memory persistent solution is optimal. Values on the edges represent the size of the activations, values inside nodes represent the computing time of the layers. The memory limit is $M_{\text{GPU}} = 15$.

Since computing B_L requires a memory of $a_{L-1} + \bar{a}_L + \delta_L + \delta_{L-1} = 3 + 4 + 4 + 3 = 14$, it is not possible to store $a_2 = 3$ during the forward phase, as our memory budget is only $M_{\text{GPU}} = 15$. We can thus identify two valid memory persistent schedules, which are candidates for optimality: either storing both a_0 and a_1 during the forward phase or storing only a_0. In the first case, a_2 is never stored before B_3, and thus F_2 is processed n times (to perform any other backward step we require memory of size $3 \times 4 = 12$, therefore storing a_1 and a_2 at the same time is still not possible). This results in a makespan $T_1 = k + 2n = 3n - 2$. In the second case, the forward phase is performed with only a_0 saved in memory during the forward propagation. Then, the computation starts from the beginning, and this time it is possible to store a_2, which allows us to compute all F_i with zero cost without recomputing F_2. At the end, it is necessary to recompute F_1, which results in a makespan $T_2 = 2(k + 2) + k = 3k + 4 = 3n - 2$.

It is also possible to build the following valid but non-persistent schedule: a_1 is saved during the forward phase, and kept in memory until the second time that F_2 is computed. Indeed, at that time, F_L has already been computed, and it is thus possible to store a_2 instead of a_1 (but not both at the same time since computing F_{L-1} requires a memory of 12). At the end it is necessary to recompute F_1, and this results in a makespan $T_0 = k + 2 \times 2 + k = 2k + 4 = 2n$.

Thus, the makespan of the non-persistent schedule is lower (by a factor $\sim \frac{3}{2}$) than the makespan of any memory persistent schedule. Nevertheless, such a counter example is not really likely to happen in practice and we show it experimentally in Section 2.4: for all classical DNNs we considered, the optimal persistent and the optimal non-persistent schedules are exactly the same, even if we do not provide a characterization of the class of graphs on which this property holds.

2.3.2 Properties of Heterogeneous Problem

To find the optimal schedule for $\textsc{Remat}(\text{L}, M_{\text{GPU}})$, we introduce a slightly weaker version of memory persistence. Further in the text, specifically for the proofs, we use the notion

of the Backward Phase given below.

Definition 10 (Backward Phase BP)**.** For a given $\ell \in \{1, \ldots, L+1\}$, we denote as phase BP_ℓ the subsequence of operations of schedule $\mathcal{S}$ that takes place between two backward operations $B_{\ell+1}$ and B_ℓ. Backward Phase BP_{L+1} corresponds to all operations in $\mathcal{S}$ before B_{L+1}, being the first backward after Loss (thus to the entire forward propagation).

Let us notice that such phases are ordered by decreasing indices, *i.e.* for any $1 \leq \ell \leq L+1$, BP_ℓ precedes $BP_{\ell-1}$. Thus, phase BP_1 is the last phase in the execution.

Properties 3. *We propose a more general version of memory persistence:*

R.1' *(Floating Memory Persistence) For any ℓ, if activation $a_{\ell-1}$ is stored with F_ℓ^{ck} or F_ℓ^{all} then until $a_{\ell-1}$ is discarded with B_ℓ or $F_\ell^{\varnothing}$, no recomputations F_i are possible for any i s.t. $i < \ell$.*

Lemma 3. *There always exists an optimal solution for* Remat(L, M_{GPU}) *that satisfies Property* ***R.1'***

Доказательство. Consider an optimal solution $\mathcal{S}$ that does not satisfy Property **R.1'**. Thus there exists $\ell, \ell' \in \{1, \ldots, L+1\}$, $\ell' \geq \ell$, *s.t.* $a_{\ell-1}$ is saved in memory and is deleted during $BP_{\ell'}$ and, simultaneously, there exists $h \geq \ell'$ so that during BP_h a sequence $F_i, \ldots, F_j$ is executed for $i, j < \ell$ *i.e.* we perform operations on the preceding layers, while $a_{\ell-1}$ still remains in the memory.

Obviously, $F_i, \ldots, F_j$ is performed to store a new activation a_j for some $j : i < j < \ell-1$. Otherwise, computing activations that come after layer ℓ would be faster starting from $a_{\ell-1}$. Let us show that $\mathcal{S}$ can be transformed into a schedule $\mathcal{S}'$ with the same makespan, but where the sequence of operations from F_i till F_j takes place in the next phase BP_{h-1}.

Before executing F_i the memory of GPU should contain at least $\{a_{i-1}, a_{\ell-1}\} \subset \mathcal{M}$. After execution of $F_i, \ldots, F_j$, several cases are possible.

1. We store both a_{i-1} and a_j ($\{a_{i-1}, a_j, a_{\ell-1}\} \subset \mathcal{M}$). If $F_i, \ldots, F_j$ are moved to the next phase BP_{h-1}, then we obtain a schedule $\mathcal{S}'$ where all phases except BP_h and BP_{h-1} are equivalent to those of $\mathcal{S}$, while the total memory consumption of $\mathcal{S}'$ is smaller ($a_j \notin \mathcal{M}$ during BP_h). Therefore, this schedule does not exceed the memory limit (thus valid), and preserves the makespan.
2. We store a_j, but discard a_i ($\{a_j, a_{\ell-1}\} \subset \mathcal{M}$).
   (a) If $a_j \geq a_{i-1}$, then, similarly, $F_i, \ldots, F_j$ can be moved to the next phase BP_{h-1}, yielding $\mathcal{S}'$ with the same makespan and lower memory consumption, thus $\mathcal{S}'$ is valid.
   (b) If $a_j < a_{i-1}$, then $\mathcal{S}$ cannot be optimal. Indeed, if a_j is stored instead of a_{i-1} in the first place (which should be feasible since $a_j < a_{i-1}$), then computing $F_i, \ldots, F_j$ in BP_h would have been unnecessary and removing these operations from $\mathcal{S}$ diminishes the makespan, meaning that $\mathcal{S}$ is not optimal.

Finally, we have shown that the optimal schedule $\mathcal{S}$ where recomputation takes place at phase BP_h can be transformed into a schedule $\mathcal{S}'$ preserving the optimality and where

the recomputation takes place at phase BP_{h-1}. The same transformation can be applied to $\mathcal{S}'$ repeatedly until $F_i, \ldots, F_j$ is moved to phase $BP_{\ell'-1}$.

□

In what follows, we denote as canonical solutions the schedules that satisfy floating memory persistence and using Lemma 3, we restrict the search to optimal canonical solutions.

Corollary 1. *In any optimal canonical solution, at any instant, only the rightmost (with the largest index) stored activation can be replaced by a new saved activation in memory, and this new activation has a larger index and a larger memory size.*

Definition 11 (Floating Materialized Activation)**.** We call the rightmost activation described in Corollary 1 as Floating Materialized Activation

Lemma 4 (Memory persistence for $\bar{a}$)**.** *There exists an optimal solution, where for any activation ℓ, no saved activation $\bar{a}_\ell$ is deleted before the corresponding backward step B_ℓ.*

Доказательство. We assume that there exists an optimal solution where for some $\ell \in \{1, \ldots, L+1\}$, $\bar{a}_\ell$ is saved and then deleted before the execution of B_ℓ. Then, let us consider the solution where a_ℓ is saved instead of $\bar{a}_\ell$. It is possible to perform all recomputations using a_ℓ and since $a_\ell \leq \bar{a}_\ell$ ($\bar{a}_\ell$ includes a_ℓ), it also helps to reduce memory consumption, thus producing a solution that is both valid and optimal in terms of makespan. This achieves the proof of the lemma. □

2.3.3 Dynamic Programming with Floating Materialized Activations

In this Section, we prove that it is possible to compute the optimal rematerialization sequence to solve Remat(L, M_{GPU}), based on the results proved in Section 2.3.2. Theorem 4 showed that Remat(L, M_{GPU}) is NP-complete in the weak sense, implying that it is necessary to discretize memory sizes to find a solution in polynomial time. After discretization, the memory limit M_{GPU} shows the total number of memory slots available on the GPU, while activation and gradient sizes ($\bar{a}_\ell, a_\ell$ and δ_ℓ) and memory overheads of operations (o_{F_ℓ} and o_{B_ℓ}) are expressed in the number of memory slots they occupy. Then, we rely on the dynamic programming algorithm to obtain the best schedule.

For a chain of length L, let us denote by $C_{\text{FL}}(s, t, \ell, m)$ the optimal execution time to perform backward steps from B_t until B_ℓ of the chain, starting from F_s, where

- a_{s-1} and δ_t are the only stored values before the execution,
- $\delta_{\ell-1}$ is the only tensor still kept in memory after the execution,
- m is the maximal available memory, without taking into account the memory occupied by a_{s-1}.

In order to proceed, one should know the minimal memory requirements for each operation. Given the chain that starts at s and finishes at t (its first backward is B_t), taking into account the data dependencies from Table 2.1 and the fact that the global

input of the chain, which is either a_{s-1} or $\bar{a}_{s-1}$, is already stored in the memory (and counted separately in all equations) and the current gradient value should be stored at any time, then

- $\mathcal{M}^{s,t,\ell}_{F^{\varnothing}_s} = \delta_t + a_s + o_{F_s}$;
- $\forall h \neq s : \mathcal{M}^{s,t,\ell}_{F^{\varnothing}_h} = \delta_t + a_{h-1} + a_h + o_{F_h}$;
- $\mathcal{M}^{s,t,\ell}_{F^{all}_s} = \delta_t + \bar{a}_s + o_{F_s}$;
- $\forall h \neq s : \mathcal{M}^{s,t,\ell}_{F^{all}_h} = \delta_t + a_{h-1} + \bar{a}_h + o_{F_h}$;
- $\mathcal{M}^{s,t,\ell}_{B_s} = \bar{a}_s + \delta_s + \delta_{s-1} + o_{B_s}$;
- $\forall h \neq s, h \geq \ell : \mathcal{M}^{s,t,\ell}_{B_h} = a_{h-1} + \bar{a}_h + \delta_h + \delta_{h-1} + o_{B_h}$.

As only $\delta_{\ell-1}$ stays in memory at the end, the floating materialized activation a_{s-1} should be fully exploited and deleted by the end of the execution. Let us introduce the following notations

$$m^{s,t,\ell}_{\varnothing} = \max_{s \leq h \leq t} \mathcal{M}^{s,t,\ell}_{F^{\varnothing}_h},$$
$$m^{s,t,\ell}_{all} = \max\left\{\mathcal{M}^{s,t,\ell}_{F^{all}_s}, \mathcal{M}^{s,t,\ell}_{B_s}\right\}.$$

Thus, $m^{s,t,\ell}_{\varnothing}$ for $1 \leq s \leq \ell < t \leq L+1$ denotes the memory required to compute all $F^{\varnothing}$ steps from s to t, and $m^{s,t,\ell}_{all}$ for $1 \leq s \leq \ell \leq L+1$ denotes the memory required to compute F^{all}_s and B_s.

Theorem 5. *$C_{FL}(s,t,\ell,m)$, the optimal makespan of a sequence that respects floating memory persistence to process the chain from layer s with backward steps from B_t until B_ℓ with available memory m, is given by:*

$$C_{FL}(s,s,s,m) = \begin{cases} u_{F_s} + u_{B_s} & m \geq m^{s,s,s}_{all} \\ \infty & m < m^{s,s,s}_{all} \end{cases}, \tag{2.4}$$

$$C_{FL}(s,t,\ell,m) = \min\left(C_1(s,t,\ell,m), \quad C_2(s,t,\ell,m)\right), \tag{2.5}$$

$$C_1(s,t,\ell,m) = \begin{cases} C^{F^{all}}_{FL}(s,t,m) & m \geq m^{s,t,\ell}_{all} \text{ and } s = \ell \\ \infty & m < m^{s,t,\ell}_{all} \text{ or } s \neq \ell \end{cases},$$

$$C_2(s,t,\ell,m) = \begin{cases} \min\limits_{\substack{s',r,t' \\ s \leq r \leq s' < t' \\ a_{s-1} \leq a_{r-1} \\ \ell < t' \leq t}} C^{F^{ck}}_{FL}(s,r,s',t,t',\ell,m) & m \geq m^{s,t,\ell}_{\varnothing} \text{ and } t \neq \ell \\ \infty & m < m^{s,t,\ell}_{\varnothing} \text{ or } t = \ell \end{cases}, \tag{2.6}$$

where

$$C_{FL}^{F^{ck}}(s,r,s',t,t',\ell,m) = \sum_{k=s}^{s'} u_{F_k} + C_{FL}(s'+1,t,t',m-a_{s'}-a_{r-1}+a_{s-1}) + C_{FL}(r,t'-1,\ell,m-a_{r-1}+a_{s-1}), \tag{2.7}$$

$$C_{FL}^{F^{all}}(s,t,m) = u_{F_s} + C_{FL}(s+1,t,s+1,m-\bar{a}_s) + u_{B_s}. \tag{2.8}$$

We can interpret these values as follows: $C_{\mathrm{FL}}^{F^{all}}(s,t,m)$ denotes the makespan for the chain from s to t, which starts by processing F_s^{all}. $C_{\mathrm{FL}}^{F^{ck}}(s,r,s',t,t',\ell,m)$ denotes the makespan for the chain from s with backward steps from B_t to B_ℓ, where

(i) forward operations from s to s' are processed with $F^{\varnothing}$ except F_r (*i.e.* $F_s^{\varnothing},\dots F_{r-1}^{\varnothing}F_r^{ck}F_{r+1}^{\varnothing},\dots F_{s'}^{\varnothing}$), during which the materialized activation moves from the position s to r, *i.e.* a_{s-1} is deleted and replaced by a_{r-1};

(ii) after $F_{s'}^{\varnothing}$ activation $a_{s'}$ is stored, thus creating another floating materialized activation;

(iii) backward phases $BP_t,\dots,BP_{t'}$ are performed with a floating materialized activation initialized by $a_{s'}$, while a_{r-1} is stored in the memory and not moving;

(iv) backward phases $BP_{t'-1},\dots,BP_\ell$ are performed with a floating materialized activation a_{r-1}, which is the rightmost saved activation after $BP_{t'}$ (no a_i for $i \geq r$ is stored, including $a_{s'}$).

Доказательство. We prove that by induction. Let us assume that $C_{\mathrm{FL}}(s',t',\ell',m)$ provides the optimal makespan when executing the chain starting at position s' and ending at t' to perform the backward steps from $B_{t'}$ till $B_{\ell'}$ for any s',t',ℓ', *s.t.* $s' \geq s$, $t' \leq t$ and $\ell' \geq s'$, except for the case $\{s'=s,t'=t,\ell'=\ell,m\}$.

Let us show that $C_{\mathrm{FL}}(s,t,\ell,m)$ provides the optimal solution for the corresponding problem. Having only a_{s-1} and δ_ℓ stored in the memory with a_{s-1} not accounted in m, there are three different possible operations to start the execution: F_s^{all}, F_s^{ck} or $F_s^{\varnothing}$.

Case 1: F_s^{all} is the first operation

In this case, both the input, which is either a_{s-1} or $\bar{a}_{s-1}$, and $\bar{a}_s$ will be saved. As memory persistence holds for $\bar{a}_s$ (see Lemma 4), then ℓ should be equal to s, otherwise according to the definition of $C_{\mathrm{FL}}(s,t,\ell,m)$, starting from F_s^{all} makes the schedule invalid. Moreover, due to floating memory persistence, no other value a_k or $\bar{a}_k$ for $0 \leq k \leq s-1$ is needed until B_{s+1}, thus computing δ_s can be done with makespan $C_{\mathrm{FL}}(s+1,t,s+1,m-\bar{a}_s)$, where the decrease in memory corresponds to the memory needed to store $\bar{a}_s$. After the completion of this chain, it is possible to perform the last backward step B_s as both $\bar{a}_s$ and a_{s-1} (or $\bar{a}_{s-1}$) are in memory. At last, $m_{all}^{s,t,\ell} \leq m$ states that processing the chain from s to t starting with F_s^{all} requires enough memory to perform the first forward and enough memory to perform the last backward B_s. Provided that the memory limit is not violated, we obtain the equation for $C_1(s,t,\ell,m)$.

Case 2: F_s^{ck} is the first operation

If the first operation is F_s^{ck}, then let us denote by $a_{s'}$ the first stored activation after a_{s-1}

(since some F^{all} operation needs to be performed before the first backward, $a_{s'}$ necessarily exists). Due to Lemma 3 and Corollary 1, while a floating materialized activation that first appears at position s' is present in memory, there is no need to consider any a_k or $\bar{a}_k$ for $k < s'$. We denote $t' : \ell < t' \leq t$ such that $a_{s'}$ is discarded not later than in the backward phase $BP_{t'}$.Thus, computing $\delta_{t'-1}$ from the input $a_{s'}$ can be done in time $C_{\text{FL}}(s'+1, t, t', m - a_{s'})$, where $s' < t'$ (floating materialized activations can only move to the right). Since $a_{s'}$ is to be stored in memory, then its memory usage should be counted outside the limit, thus $m - a_{s'}$ is the maximal available memory for this execution. Once $B_{t'}$ is processed, it remains to execute another chain that starts at position s and ends at position $t' - 1$, where the new currently stored gradient is $\delta_{t'-1}$, whereas activation $a_{s'}$ is not needed anymore and is finally removed. Putting all this together, we get the following makespan

$$\sum_{k=s}^{s'} u_{F_k} + C_{\text{FL}}(s'+1, t, t', m - a_{s'}) + C_{\text{FL}}(s, t'-1, \ell, m),$$

which is equivalent to Eq. (2.7) when $r = s$.

Case 3: $F_s^{\varnothing}$ is the first operation

Finally, if the first operation is $F_s^{\varnothing}$ then the materialized activation a_{s-1} (not $\bar{a}_{s-1}$ as $F_s^{\varnothing}$ does not take it as an input) will move to the next position a_{r-1} for some r and according to Corollary 1, it must satisfy $a_{r-1} > a_{s-1}$. Then, after the materialized value has moved to the next position, we use a similar reasoning as in Case 2, where the memory usage is fixed by subtracting from the memory the difference $a_{r-1} - a_{s-1}$, which leads to the Eq. (2.7).

In the end, $m_{\varnothing}^{s,t,\ell} \leq m$ checks the memory validity as it states that processing the chain from s to t requires at least enough memory to execute all the forward steps without saving any activation. Thus, combining Cases 2 and 3 provided that memory constraint is fulfilled brings us to $C_2(s, t, \ell, m)$.

Initialization

In order to backpropagate one layer, F_s^{all} must be executed to be able to process B_s afterwards. This requires a memory size of $m_{all}^{s,s,s}$. As stated in the definition of $C_{\text{FL}}(s, t, \ell, m)$, we assume that the input size (either a_{s-1} or $\bar{a}_{s-1}$) is not accounted in the memory limit m, and $m_{all}^{s,s,s}$ represents the highest value of the peak memory usage between forward and backward operations corresponding to layer s.

As Eq. (2.4) gives an optimal initialization, then, by induction, finding the minimal makespan among Cases 1,2 and 3 (which corresponds to computing Eq. (2.5)) yields the optimal solution.

□

Theorem 5 establishes the correctness of Algorithm 1 and Algorithm 2 to compute an optimal sequence for REMAT(L, M_{GPU}), whose makespan is $C_{\text{FL}}(1, L+1, 1, M_{\text{GPU}} - a_0)$ (the input of the chain is counted outside).

Let us notice that the output of such dynamic program is a table of size $O(M_{\text{GPU}} L^3)$ and to compute any element of this table takes $O(L^3)$. Thus, the worst case complexity of

this method to obtain $C_{\text{FL}}(1, L+1, 1, M_{\text{GPU}})$ is $O(M_{\text{GPU}} L^6)$. Therefore, for deep networks (large L values), this method can take significant time (see Figure 2.4 Section 2.4.3). Even if it has to be performed only once for the whole training process, we introduce in Section 2.3.4 an algorithm with much smaller complexity that returns the optimal solution among memory persistent valid schedules. As already noticed, even if we prove that such schedules can be in theory significantly sub-optimal in Section 2.3.1, in all the experiments that we performed in Section 2.4.5, we did not observe in practice any difference in resulting sequences, what makes the algorithm presented in Section 2.3.4 a good candidate to be used in practice.

2.3.4 Optimal Memory Persistent Solution

Let us now relax the problem, and instead of looking for general solutions we want to find the best solution among memory persistent sequences, satisfying Property **R.1**. Similarly to the general case, the problem can be solved with dynamic programming. This extends the result of [39] to the case of heterogeneous memory costs, and at the same time it is also adapted for the neural network data dependencies shown in Figure 2.1a. Similarly to the general case, we apply discretization with respect to memory sizes so that data sizes (a_ℓ, $\bar{a}_\ell$, and δ_ℓ) and operation overheads (o_{F_ℓ} and o_{B_ℓ}) are expressed in terms of memory slots and M_{GPU} shows the total number of memory slots on the GPU.

Let us prove that it is possible to compute the optimal memory persistent rematerialization sequence in the fully heterogeneous case corresponding to REMAT(L, M_{GPU}). For a chain of length L, let us denote by $C_{\text{MP}}(s, t, m)$ the optimal execution time to perform backward steps from B_t till B_s of the chain starting from position s, where

- input tensors a_{s-1} and δ_t are the only stored values before the processing of the chain,
- the maximal available memory is at most m, without taking into account the memory occupied by a_{s-1}.

Algorithm 1 Computation of an optimal schedule for REMAT(L, M_{GPU})

```
Initialize table C of size (L+1) × (L+1) × (L+1) × M_GPU
for 1 ≤ s ≤ L+1 and 1 ≤ m ≤ M_GPU do
    Initialize C[s, s, s, m] with Eq. (2.4)
for m = 1, ..., M_GPU do
    for d = 1, ..., L do
        for s = 1, ..., L+1−d do
            t = s + d
            for ℓ = s, ..., t do
                Compute C[s, t, ℓ, m] with Eq. (2.5)
return OptRecFL(C, 1, L+1, 1, M_GPU − a_0)                ▷ Alg. 2
```

Algorithm 2 OptRecFL(C, s, t, ℓ, m) – Obtain optimal sequence from the table C

if $C[s, t, \ell, m] = \infty$ **then**
  return Infeasible
else if $s = t = \ell$ **then**
  return (F_s^{all}, B_s)
else if $C[s, t, \ell, m] = C_{\text{FL}}^{F^{ck}}(s, s, s', t, t', \ell, m)$ **then**
  $\mathcal{S} \leftarrow (F_s^{ck}, F_{s+1}^{\varnothing}, \ldots, F_{s'}^{\varnothing})$
  $\mathcal{S} \leftarrow (\mathcal{S}, \text{OptRecFL}(C, s' + 1, t, t', m - a_{s'}))$
  return $(\mathcal{S}, \text{OptRecFL}(C, s, t' - 1, \ell, m))$
else if $C[s, t, \ell, m] = C_{\text{FL}}^{F^{ck}}(s, r, s', t, t', \ell, m)$ **then**
  $\mathcal{S} \leftarrow (F_s^{\varnothing}, F_{s+1}^{\varnothing}, \ldots, F_{r-1}^{\varnothing}, F_r^{ck}, F_{r+1}^{\varnothing}, \ldots F_{s'}^{\varnothing})$
  return $(\mathcal{S}, \text{OptRecFL}(C, s'+1, t, t', m - a_{s'} - a_{r-1} + a_{s-1}), \text{OptRecFL}(C, r, t'-1, \ell, m))$
else
  return $(F_s^{all}, \text{OptRecFL}(C, s + 1, t, s + 1, m - \bar{a}_s), B_s)$

The memory limits are modified in the following way:

$$m_{\varnothing}^{s,t} = \max_{s \le h \le t} \mathcal{M}_{F_h^{\varnothing}}^{s,t},$$
$$m_{all}^{s,t} = \max\left\{\mathcal{M}_{F_s^{all}}^{s,t}, \mathcal{M}_{B_s}^{s,t}\right\}.$$

$m_{\varnothing}^{s,t}$ for $1 \le s < t \le L + 1$ denotes the memory peak to compute all $F^{\varnothing}$ steps from s to t, and $m_{all}^{s,t}$ for $1 \le s \le t \le L + 1$ denotes the memory peak to run F_s^{all} and B_s.

Theorem 6. *$C_{MP}(s, t, m)$, the optimal time for any valid persistent sequence (following Property **R.1**) to process the chain from layer s to layer $t \ge s$ with available memory m, is given by*

$$C_{MP}(s, s, m) = \begin{cases} u_{F_s} + u_{B_s} & m \ge m_{all}^{s,s} \\ \infty & m < m_{all}^{s,s} \end{cases}, \tag{2.9}$$

$$C_{MP}(s, t, m) = \min\left(C_1(s, t, m), \quad C_2(s, t, m)\right), \tag{2.10}$$

$$C_1(s, t, m) = \begin{cases} C_{MP}^{F^{all}}(s, t, m) & m \ge m_{all}^{s,t} \\ \infty & m < m_{all}^{s,t} \end{cases},$$

$$C_2(s, t, m) = \begin{cases} \min\limits_{s'=s,\ldots,t-1} C_{MP}^{F^{ck}}(s, s', t, m) & m \ge m_{\varnothing}^{s,t} \\ \infty & m < m_{\varnothing}^{s,t} \end{cases},$$

where

$$C_{MP}^{F^{all}}(s,t,m) = u_{F_s} + C_{MP}(s+1,t,m-\bar{a}_s) + u_{B_s},$$
$$C_{MP}^{F^{ck}}(s,s',t,m) = \sum_{k=s}^{s'} u_{F_k} + C_{MP}(s'+1,t,m-a_{s'}) + C_{MP}(s,s',m).$$

We can interpret these values as follows: $C_{\mathrm{MP}}^{F^{all}}(s,t,m)$ is the computing time for the chain from s to t if F_s^{all} is the first operation. $C_{\mathrm{MP}}^{F^{ck}}(s,s',t,m)$ denotes the computing time for the chain from s to t if forward operations from s to $s'-1$ are processed with $F^{\varnothing}$, whereas a_{s-1} is stored in memory by F_s^{ck}.

The proof is based on the same arguments as provided in Theorem 5 with the difference that once stored, an activation is not deleted until the corresponding backward step, so that this saved value naturally divides the chain into two subchains, that can be processed one after the other. Therefore, $C_{\mathrm{MP}}^{F^{ck}}(s,s',t,m)$ is now defined with only four arguments: s that determines the start of the chain, t for the end of the chain, s' for the separation point between two subchains and m, the memory limit.

Доказательство. Analogously to the proof of Theorem 5, we prove it by induction. We assume that $C_{\mathrm{MP}}(s',t',m)$ gives an optimal makespan for chains from s' to t', having m available memory for any m, s' and t' such that $t'-s' < t-s$. Let us now show that $C_{\mathrm{MP}}(s,t,m)$ gives an optimal makespan for a chain from s and t as well. Since we are looking for a persistent schedule, and the input tensor a_{s-1} is in memory, the optimal sequence has only two possible ways to start: either with F_s^{ck} to store a_{s-1} and compute a_s, or with F_s^{all} to compute $\bar{a}_s$ and save the input a_{s-1} in the memory at the same time (see Table 2.1).

Case 1: F_s^{all} is the first operation

If the first operation is F_s^{all} then by definition the value $\bar{a}_s$ should be saved together with a_{s-1}. As memory persistence holds and no other value a_i or $\bar{a}_i$ for $0 \le i \le s-1$ is needed until B_{s+1}, computing δ_s can be done in time $C_{\mathrm{MP}}(s+1,t,m-\bar{a}_s)$, where the decrease in memory corresponds to the memory needed to store $\bar{a}_s$. After the completion of this chain, it is possible to perform the last backward step B_s as both $\bar{a}_s$ and a_{s-1} are in GPU memory. Provided that the memory constraint is fulfilled, we get the equation for $C_{\mathrm{MP}}^{F^{all}}(s,t,m)$. In this case, the memory constraint ensures that when executing the chain from s to t there is enough memory to perform F_s^{all} and B_s, while the rest of the sequence is valid if $C_{\mathrm{MP}}(s+1,t,m-\bar{a}_s)$ gives a finite value.

Case 2: F_s^{ck} is the first operation

If the first operation is F_s^{ck}, then we can denote $a_{s'}$ the first activation stored in memory after a_{s-1} (since some F^{all} operation has to be performed before the first backward, $a_{s'}$ necessarily exists). Due to memory persistence, while $a_{s'}$ is present in memory there is no need to consider any a_i or $\bar{a}_i$ for $s \le i < s'$, so computing $\delta_{s'}$ from the input $a_{s'}$ can be done optimally in time $C_{\mathrm{MP}}(s',t,m-a_{s'})$. Indeed, we assume that $a_{s'}$ is materialized in memory, but we count its memory usage outside the limit $m-a_{s'}$. Once this chain is

processed, the remaining part of the chain represents another chain that starts at position s and finishes at s', where the new currently stored gradient is $\delta_{s'}$ and $a_{s'}$ is not needed anymore and thus is finally removed. Bringing everything together yields the equation for $C_{\text{MP}}^{F^{ck}}(s, s', t, m)$. Choosing s' so that $C_{\text{MP}}^{F^{ck}}(s, s', t, m)$ is minimal guarantees the smallest possible makespan, yielding $C_2(s, t, m)$, when memory constraint is not violated. Here the memory constraint ensures that enough memory is available to execute the chain from s to t by performing all the forward steps without saving any activation.

Initialization

Finally, Eq. (2.9) is a valid initialization of the dynamic program. Indeed, in order to backpropagate one layer, F_s^{all} must be performed to be able to execute B_s afterwards. This requires a memory of size $m_{all}^{s,s}$, which represents the memory peak between forward and backward operations corresponding to layer s.

As Eq. (2.9) provides an optimal initialization, then, by induction, finding the minimal makespan among Cases 1 and 2, *i.e.* computing Eq. (2.5), provides the optimal solution. □

This theorem proves that Algorithm 3 and Algorithm 4 compute an optimal persistent sequence, for all input parameters, *i.e.* find an optimal solution for REMAT(L, M_{GPU}) under Property **R.1**. Indeed, the computing time of the returned sequence is exactly $C_{\text{MP}}(1, L+1, M_{\text{GPU}})$. In contrast with Algorithm 1, Algorithm 3 requires at most $O(M_{\text{GPU}} L^3)$ operations (instead of $O(M_{\text{GPU}} L^6)$).

Algorithm 3 Optimal persistent schedule for a chain of length L with memory M_{GPU}.

```
Initialize table C of size (L+1) × (L+1) × M_GPU
for 1 ≤ s ≤ L+1 and 1 ≤ m ≤ M_GPU do
    Initialize C[s, s, m] with Eq. (2.9)
for m = 1, ..., M_GPU do
    for d = 1, ..., L do
        for s = 1, ..., L+1−d do
            Compute C[s, s+d, m] with Equation (2.10)
return OptRecP(C, 1, L+1, M_GPU − a_0)                ▷ Alg. 4
```

2.4 Implementation and Validation

We demonstrate the applicability of our approach by presenting ROTOR [48], a tool that allows the above algorithms to be used with any Pytorch DNN based on the `nn.Sequential` container. ROTOR is used in a very similar fashion to the existing `checkpoint_sequential` tool already available in PyTorch [1], but offers a much more optimized materialization scheme. Our tool works in three phases: parameter estimation, optimal sequence computation and sequence processing. It is expected that the first two

Algorithm 4 OptRecP(C, s, t, m) – Computation of the optimal persistent sequence from table C

if $C[s, t, m] = \infty$ **then**
  return Infeasible
else if $s = t$ **then**
  return (F_s^{all}, B_s)
else if $C[s, t, m] = C_{\text{MP}}^{F^{ck}}(s, s', t, m)$ **then**
  return $(F_s^{ck}, F_{s+1}^{\varnothing}, \ldots, F_{s'}^{\varnothing}, \text{OptRecP}(C, s'+1, t, m - a_{s'}), \text{OptRecP}(C, s, s', m))$
else
  return $(F_s^{all}, \text{OptRecP}(C, s+1, t, m - \bar{a}_s), B_s)$

phases are performed only once, before the start of the training, while the sequence is used throughout the whole training process.

2.4.1 Parameter Estimation

In the parameter estimation phase, the goal is to measure the quantities (time and memory sizes) associated to the input DNN, in order to instantiate the parameters of the model described in Figure 2.1a used as input values for both Algorithm 1 and Algorithm 3. These quantities are the memory sizes a_ℓ, $\bar{a}_\ell$, and δ_ℓ, the memory overheads o_{F_ℓ}, o_{B_ℓ}, and the execution time of each operation in the sequence u_{F_ℓ}, u_{B_ℓ} for all layers $\ell : 1 \leq \ell \leq L + 1$.

Parameter estimation is done in the following way: given a chain and a fixed sample input data a_0, forward and backward operations of each stage are processed one after the other. From $a_{\ell-1}$ the forward operation F_ℓ^{all} is processed to obtain $\bar{a}_\ell$, and the backward operation with an arbitrary tensor for δ_ℓ (generated randomly with the same shape and size as a tensor a_ℓ). The execution time of each operation is measured, and the memory management interface of PyTorch is used to obtain the memory usage of $\bar{a}_\ell$ and the peak memory usage of both forward and backward operations. The memory size of a_ℓ is found by extracting the tensor a_ℓ from $\bar{a}_\ell$ and taking the number of elements in a_ℓ and multiplying it by the size of one element.

This parameter estimation assumes that the computations performed by the neural network do not depend on the input data (a very similar assumption is made for the `jit.trace()` function of PyTorch), so that the measurement on a sample input $\tilde{x}$ is representative of the actual execution on the training data x. Adapting the approach presented in this paper to a data-dependent network would require both to be able to correctly predict the execution times for each given input and to recompute the optimal sequence for each new input, and is thus out of the scope of our work.

2.4.2 Computation of the Optimal Sequence

Once all measurements have been performed, for any given memory limit M_{GPU}, the optimal floating persistent and persistent sequences can be computed using respectively

Algorithm 1 and Algorithm 3, and stored for the processing phase. As mentioned previously, in order to limit the computational cost of this phase, all measured memory sizes are *discretized*. We divide the memory of size M_{GPU} (expressed in bytes) into S memory slots ($S = 500$ is a reasonable value that provides a good trade-off between computing time and solution quality; we used this value for all experiments in this chapter), each of size $\frac{M_{\text{GPU}}}{S}$, and all memory sizes are expressed as an integer number of slots, rounded up if necessary. Thus, M_{GPU} is now limited and expressed in number of slots and is equal to $S = 500$. The complexity of the resulting algorithm is thus independent of the actual memory limit, at the cost of at most $1 + \frac{1}{S}$ overestimation of memory sizes.

2.4.3 Comparison between Algorithm 1 and Algorithm 3

We provide in ROTOR a C implementation of both dynamic programming algorithms (Algorithm 1 and Algorithm 3). The running time of Algorithm 3 on most of the networks in our experiments is below 1 second. The longest execution time was obtained with ResNet-1001 [46], which results in a chain of length 339, and an execution time below 20 seconds. Since this computation is performed only once for the whole training phase, such an execution time is completely acceptable. A comparison with the computing time of Algorithm 1 is shown in Figure 2.4.

For each considered neural network and for each set of parameters, we compute the optimal sequences produced by Algorithm 1 and Algorithm 3. Although we show in Section 2.3.1 that the ratio between the makespan produced by both algorithms can be as high as $\frac{3}{2}$, in practice the computed solutions for all test cases turned out to be exactly the same. Since the running time of Algorithm 1 is significantly higher than the one of Algorithm 3 as depicted in Figure 2.4, a reasonable strategy is to use Algorithm 3 to compute the optimal solution.

2.4.4 Experimental Setting

All experiments presented in this chapter have been performed with Python 3.5.9 and PyTorch 1.3.0. The computing node contains 40 Intel Xeon Gold 6148 cores at 2.4GHz, with a Nvidia Tesla V100-PCIE GPU card with 15.75GB of memory. We experiment with three different kinds on networks, whose implementation is available in the `torchvision` package of PyTorch: ResNet, DenseNet, and Inception v3. All three types of networks have been slightly adapted to be able to use ROTOR [48], by using a `nn.Sequential` module where applicable. We use all available depths for ResNet: 18, 34, 50, 101, 152, which are available in `torchvision`, and we also use versions with depth 200 and 1001 proposed in previous work [46]. Similarly, for DenseNet, we use depths 121, 161, 169 and 201.

We use three different image sizes: small images of shape 224×224 (which is the default and minimal image size for all models of `torchvision`), medium images of shape 500×500, and large images of shape 1000×1000. For each model and image size, we consider different batch sizes that are powers of 2, starting from the smallest batch size

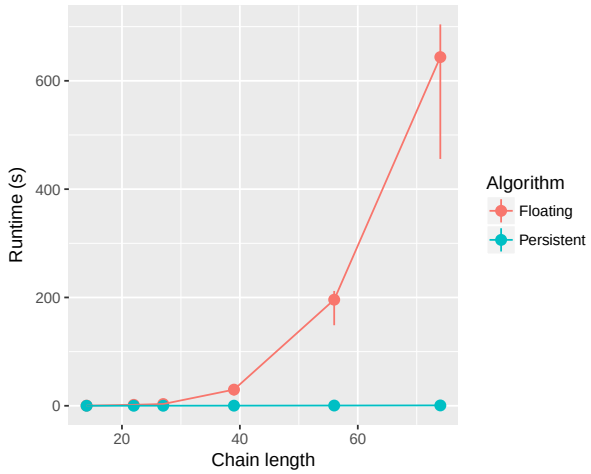


Рис. 2.4: Running times of Algorithm 1 and Algorithm 3 for different network sizes.

that ensures a reasonable throughput[1].

We compare five strategies to perform a training iteration on those models.

- The **PyTorch** strategy consists in the standard way of computing the forward and backward operations, where all intermediate activations are materialized.
- The **sequential** strategy relies on the `checkpoint_sequential` tool of PyTorch [1]. This strategy splits the chain into a given number of segments s and, during the forward phase only, stores activations at the beginning of each segment. Each forward computation is thus performed twice, except those of the last segment. We use 10 different number of segments, from 2 (always included) to $2\sqrt{L}$, where L is the length of the chain[2]. The same strategy is used in [36], but the number of segments has to be hand-tuned.
- The **revolve** strategy uses the optimal algorithm adapted to heterogeneous chains of the Automatic Differentiation model [40], and converts it to a valid solution by saving only activations a to memory, and performing a F^{all} step before each backward step to enforce validity. This is the same strategy as advocated in Appendix C of [42].
- The **optimal** strategy corresponds to ROTOR [48] and uses Algorithm 3 (or equivalently Algorithm 1 since they provide the same results for all experiments) for 10 different memory limits, equally spaced between 0 and the memory usage of the **PyTorch** strategy.
- The **checkmate** strategy is a reimplementation in our PyTorch framework of the linear programming approach presented in [52]. Since we are interested in significantly large networks, we use the approximate version, based on two-phase

[1]With small batch sizes, we observe that doubling the batch size effectively doubles the throughput, which shows that the GPU is not used efficiently in the former case.

[2]Note that $\sqrt{L}$ is the optimal number of segments for this strategy when the chain is homogeneous.

rounding of the best fractional solution. This approach does not differentiate between a and $\bar{a}$ values; so for validity we have included a conversion step that uses F^{all} operations when necessary. The linear programs are solved with CPLEX version 12.10, with a one hour time limit. Due to the high computational cost, this strategy is only shown in a selection of plots.

For each model, image size and batch size, we perform enough iterations to ensure that the **PyTorch** strategy lasts at least 500ms, and we measure the actual memory peak and duration over 5 runs. The obtained measurements are very stable, so all plots in the next section present the median duration over the 5 runs for each experiment (on average, the difference between the highest and lowest measured throughputs are within 1% of the median). For each run, the memory peak consumption and the throughput of the experiments have been carefully measured, using the same mechanism as the one used to perform the measurement phase.

2.4.5 Experimental Results

For the sake of conciseness, we analyze carefully only a representative selection of the results; the behavior on other experiments is very similar and can be found in Section 2.6. All plots have the same structure: for a given set of parameters (network, depth, image size and batch size), we plot for each strategy the achieved throughput (in terms of images per second) against the peak memory usage. The square red dot represents the performance obtained by the standard **PyTorch** strategy, and its absence from the graph means that a memory overflow error was encountered when attempting to execute it. Purple crosses represent the results obtained with the **sequential** strategy for different number of segments. The blue line with triangles shows the result obtained with our **optimal** strategy. The green line with circles shows the result obtained with the **revolve** algorithm. When available, the orange line with crossed squares shows the performance achieved with our **checkmate** implementation[3]. We draw lines to emphasize the fact that these strategies can be given any memory limit as input, whereas the result of **sequential** is inherently tied to a discrete number of segments. We provide a representative selection of results in Figures 2.5 to 2.7, and the complete results can be found in Figures 2.8 to 2.15.

Figure 2.5 depicts the results for the ResNet neural network with depth of 101, with image size 1000×1000 and batch size 1, 2, 4, and 8. For a batch size of 1, the **PyTorch** strategy has a memory peak consumption of 2.83 GiB, which is enough to fit on this GPU. However, when the batch size is 8, the **PyTorch** strategy fails to compute the backpropagation due to memory limitations. The **sequential** strategy offers a discrete alternative by dividing the chain into a given number of segments (in this case from 2 to 11). For every batch size, the best throughput is reached when the number of segments is equal to 2. For instance, when the batch size is 8, the throughput of the **sequential**

[3]The structure of the Inception network makes it more challenging to adapt to the checkmate algorithm. Since the performance of **checkmate** on the other networks is not satisfactory, we did not perform the experiments on the Inception network with **checkmate**

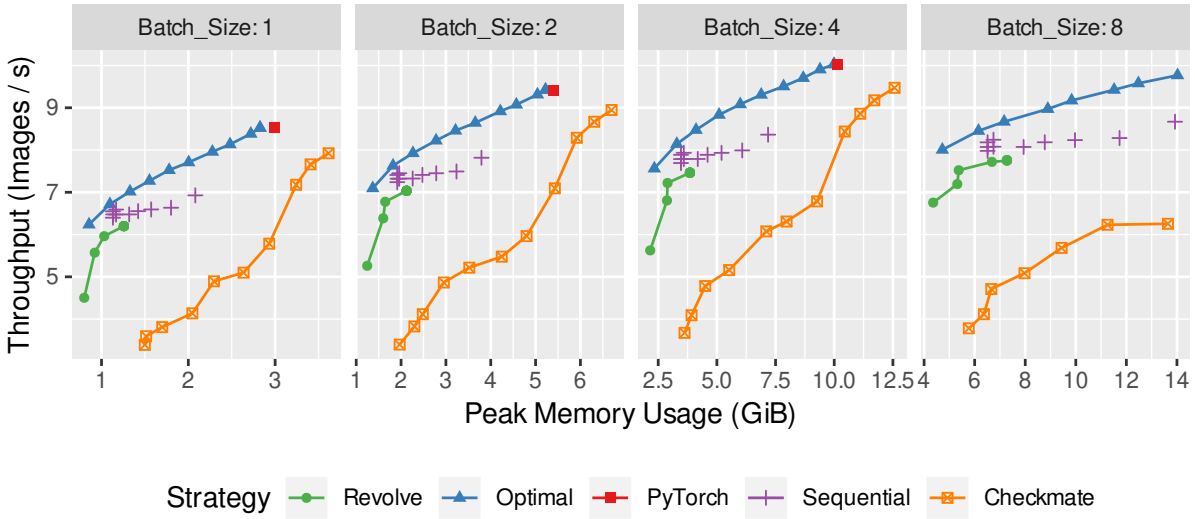


Рис. 2.5: Experimental results for the ResNet network with depth 101 and image size 1000.

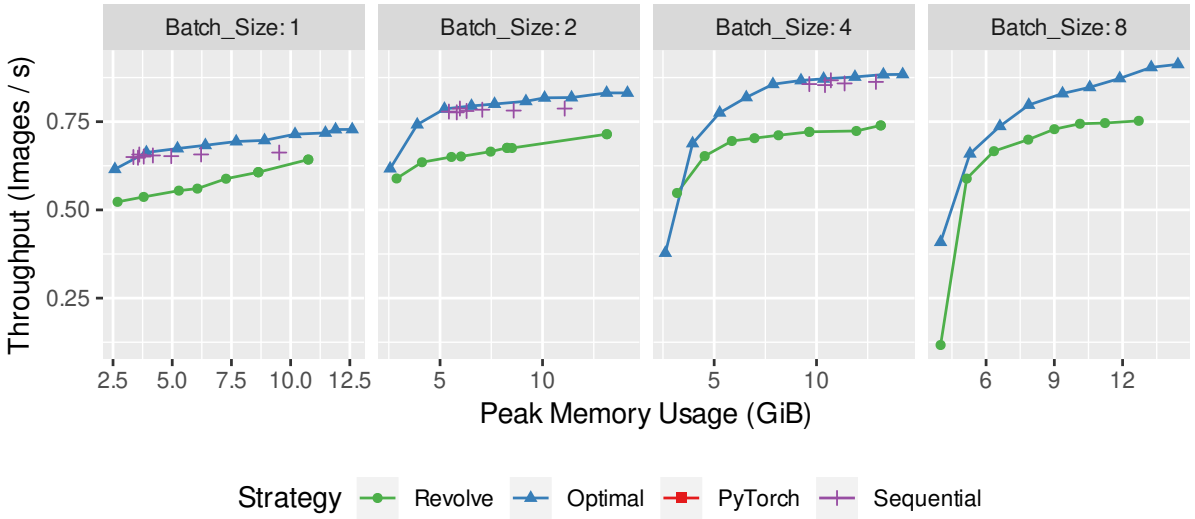


Рис. 2.6: Experimental results for the ResNet network with depth 1001 and image size 224.

strategy with 2 segments is on average 8.67 images/s with a memory peak consumption of 13.91 GiB. The **optimal** strategy offers a continuous alternative by implementing the best rematerialization strategy for any given memory bound. We can see that for a given memory peak, the **optimal** strategy outperforms the **sequential** strategy by up to 15%. For instance, when the batch size is 8, the maximum throughput achieved by the **optimal** strategy is 9.77 images/s. The previous **revolve** algorithm provides a continuous approach as well. However, it requires to compute each forward operation at least twice (once in the forward phase, once before the backward operation), which incurs a much lower throughput than both other solutions. Furthermore, since this algorithm does not consider saving the larger $\bar{a}$ values, it is unable to make use of larger memory sizes. Finally, the **checkmate** strategy has significant difficulty to optimize these deep networks because of two issues. The first issue is that the resulting linear programs are very large and cannot be solved exactly in reasonable time; we thus use the fractional relaxation proposed in [52], which does not correctly estimate the memory usage: the memory used by the solutions of **checkmate** is consistently higher than the limit provided to the linear program. The second issue is that in **checkmate**, authors do not consider the difference between a and $\bar{a}$ values: given a solution produced by the linear program, producing a correct execution sequence for the PyTorch framework requires to convert the solution using F^{all} operations, which in some cases means inserting additional recomputations, lowering the obtained throughput.

Figure 2.6 displays the same results for the ResNet with depth of 1001 and image size of 224×224. This setup requires much more memory and the **PyTorch** strategy fails even when the batch size is 1. The **sequential** strategy requires at least 6 segments for batch size 1, 10 segments for batch size 2, and 18 segments for batch size 4, and cannot perform the backpropagation when the batch size is 8. Not only does the **optimal** strategy outperform the **sequential** strategy when it does not fail but it offers a stable solution to train the neural network even with a larger batch size, which allows us to increase the achieved throughput thanks to a better GPU efficiency (0.91 for **optimal** whereas the highest throughput achieved by **sequential** is 0.86). It is interesting to note that based on the parameters estimated by ROTOR, running the setting with batch size 8 with the **PyTorch** strategy would require 225 GiB of memory, and achieve a throughput of 1.18 images/s. Additional results in Figure 2.15 also show that ROTOR is able to run this large network even with medium and large image sizes.

All these conclusions hold for every tested neural network and parameters. Figure 2.7 displays some of them and shows that the behavior of the ROTOR strategy is stable on various network sizes and image sizes. To summarize, we also compute the ratio between the highest throughput obtained by **sequential** and the throughput achieved by ROTOR with the corresponding memory usage. On average over all tested sets of parameters, **optimal** achieves 12.8% higher throughput.

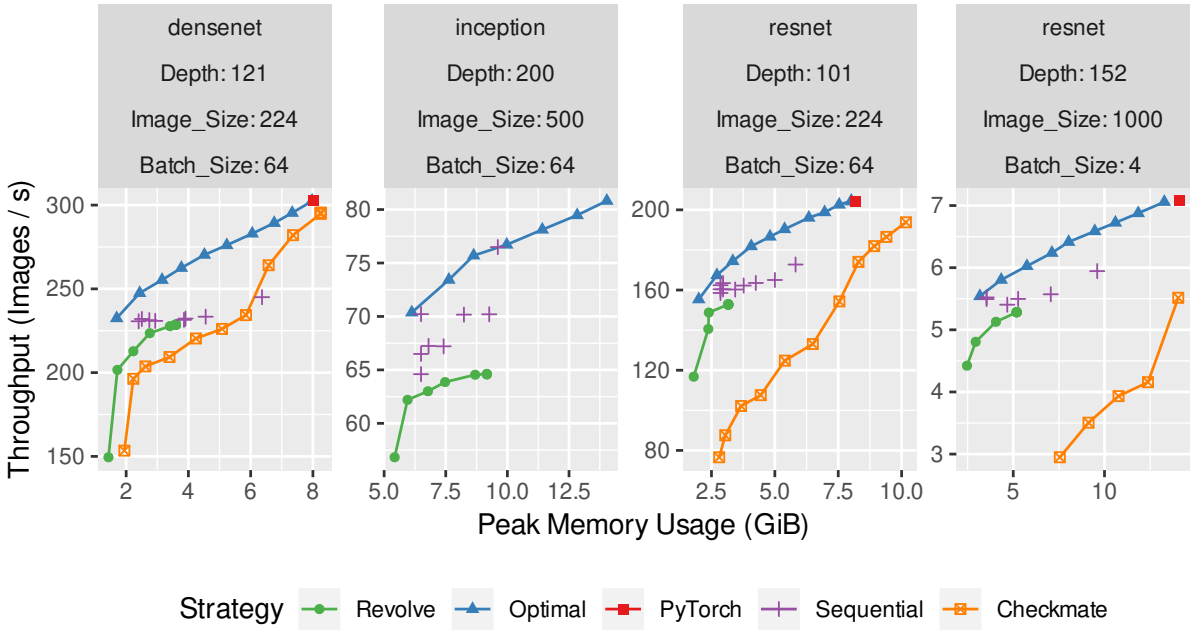


Рис. 2.7: Experimental results for several situations.

2.5 Conclusion

This chapter describes a new rematerialization strategy that leverages operations available in DNN frameworks with the capabilities of autograd functions. We carefully model backpropagation and we propose a dynamic program that computes the optimal persistent schedule for any sequentialized network. We also present ROTOR [48], which can apply this dynamic program for any sequential PyTorch module. On the theoretical side, we prove that the memory persistence property used by Automatic Differentiation community to derive dynamic programming optimal solutions no longer holds in presence of heterogeneous activation sizes. We propose a weaker version of memory persistence, called floating memory persistence, and we prove that it can be used to find optimal rematerialization strategies in the fully heterogeneous case. On the practical side, using in-depth experiments, we compare achieved results with ROTOR against (i) a periodic checkpointing strategy available in PyTorch, (ii) an optimal persistent strategy adapted from the Automatic Differentiation literature to a fully heterogeneous setting, but which does not use all the capabilities available in DNN frameworks and (iii) a rematerialization strategy based on Linear Programming (Checkmate). We show that ROTOR consistently outperforms these rematerialization strategies, for a large class of networks, image sizes and batch sizes. Our fully automatic tool ROTOR increases throughput by an average of 13% compared to its best competitor, with better flexibility since it offers the ability to specify an arbitrary memory limit. ROTOR therefore allows us to use larger models, larger

batches or larger images while automatically adapting to the memory of the training device.

2.6 Additional Plots

In this section, we provide the results (Figures 2.8-2.15) for all tested neural networks for different batch-sizes and image sizes. They consistently demonstrate the advantage of ROTOR over other rematerialization methods.

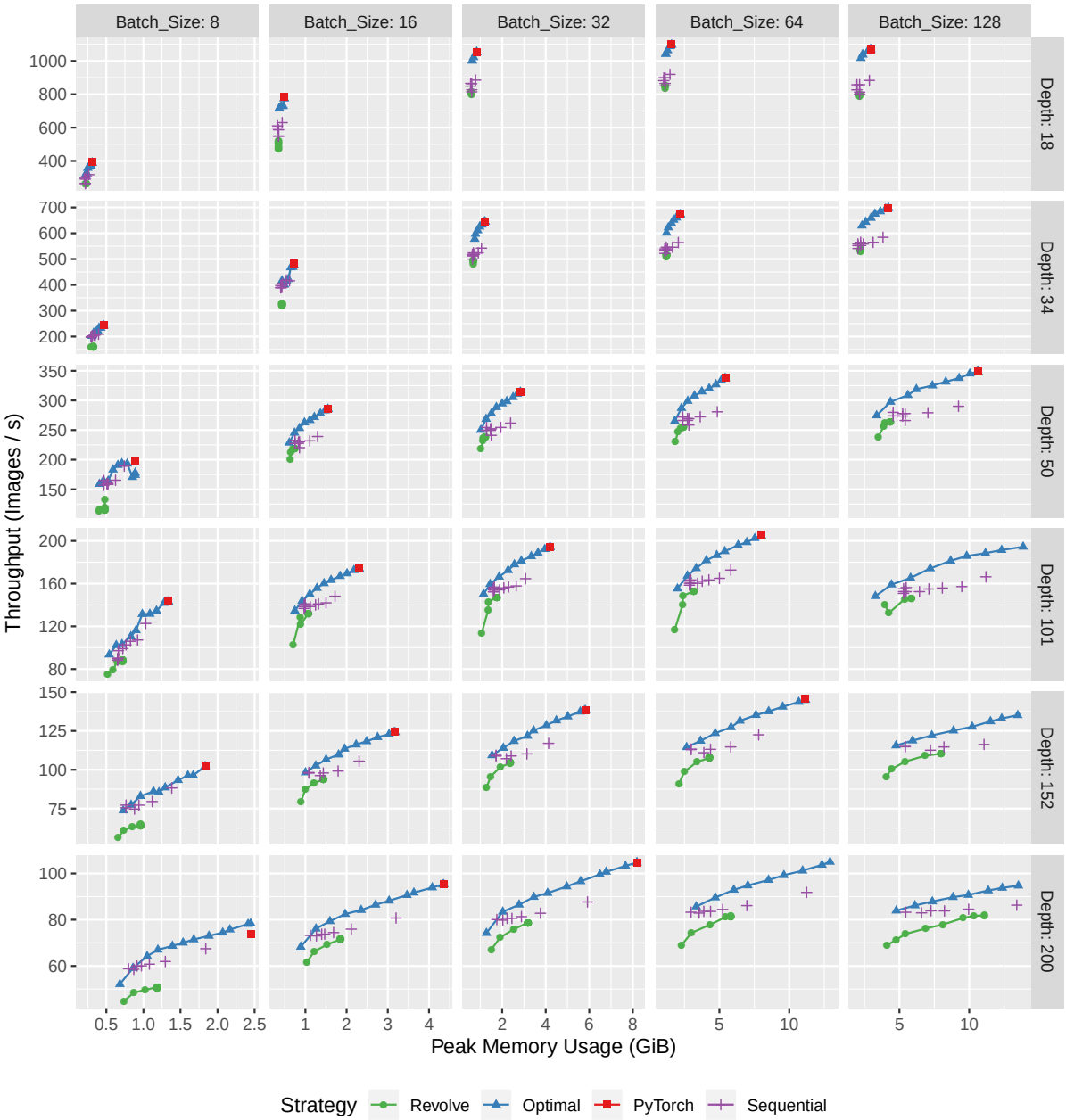


Рис. 2.8: Results for ResNet with image size 224, for different depths and batch sizes.

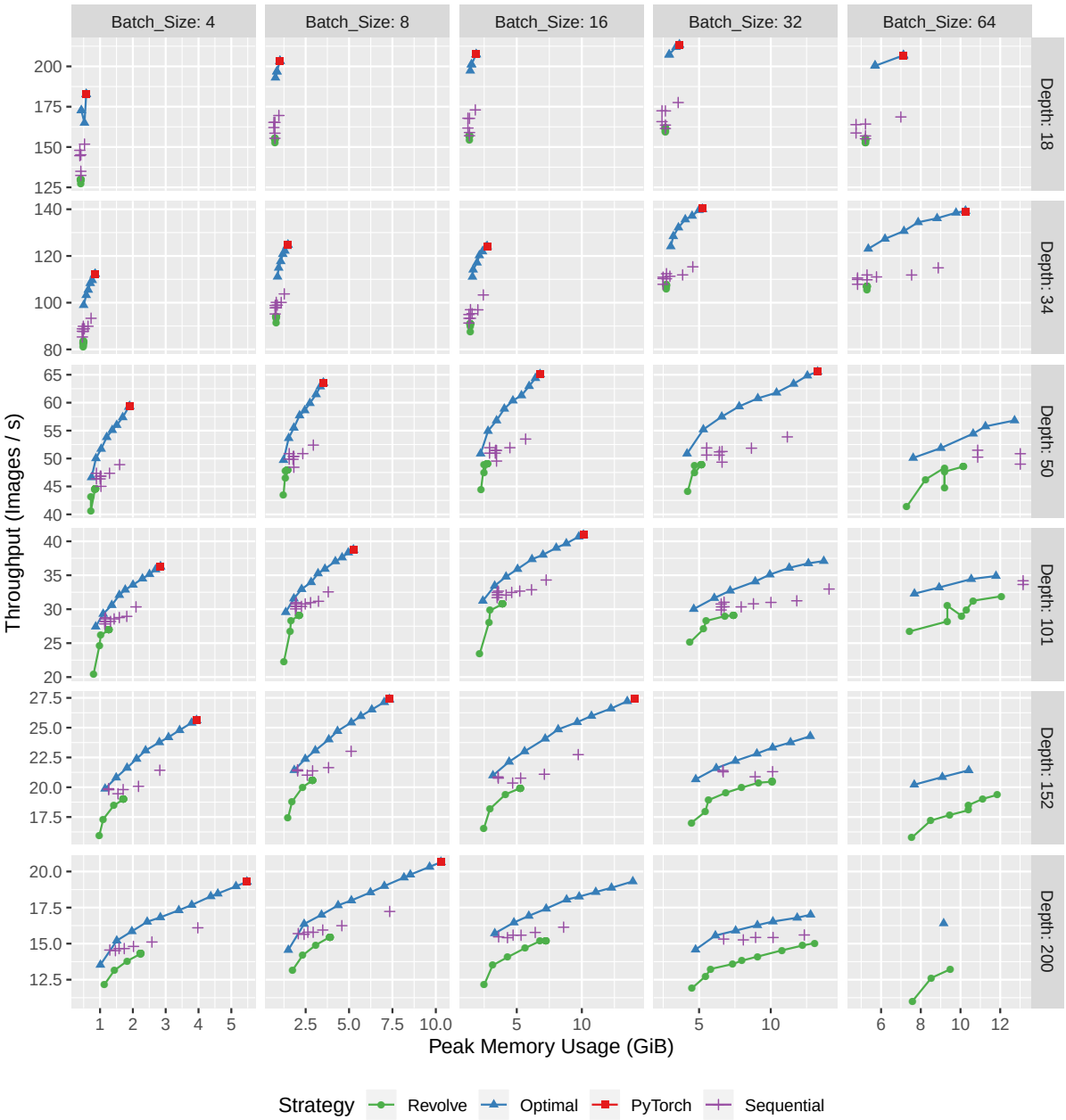


Рис. 2.9: Results for ResNet with image size 500, for different depths and batch sizes.

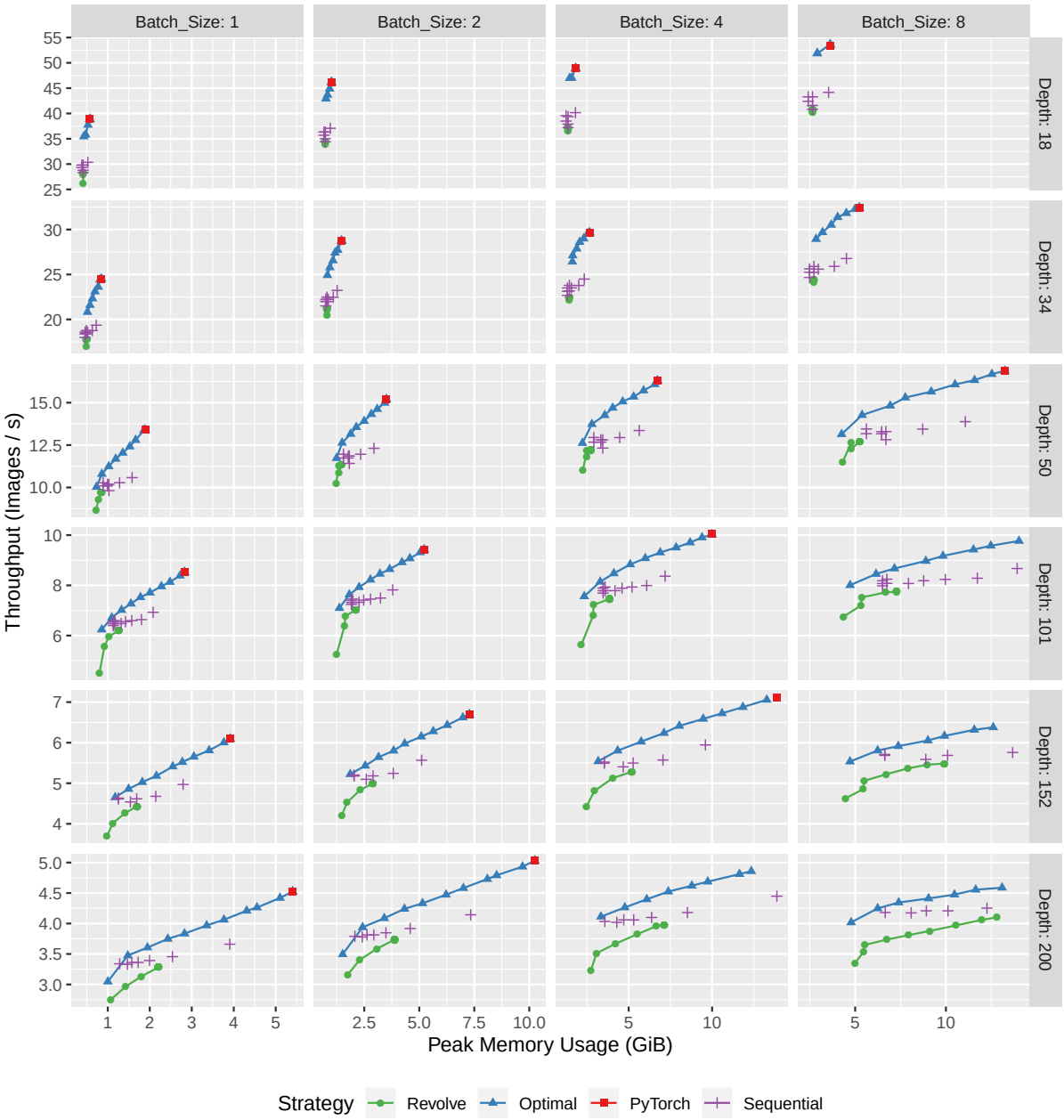


Рис. 2.10: Results for ResNet with image size 1000, for different depths and batch sizes.

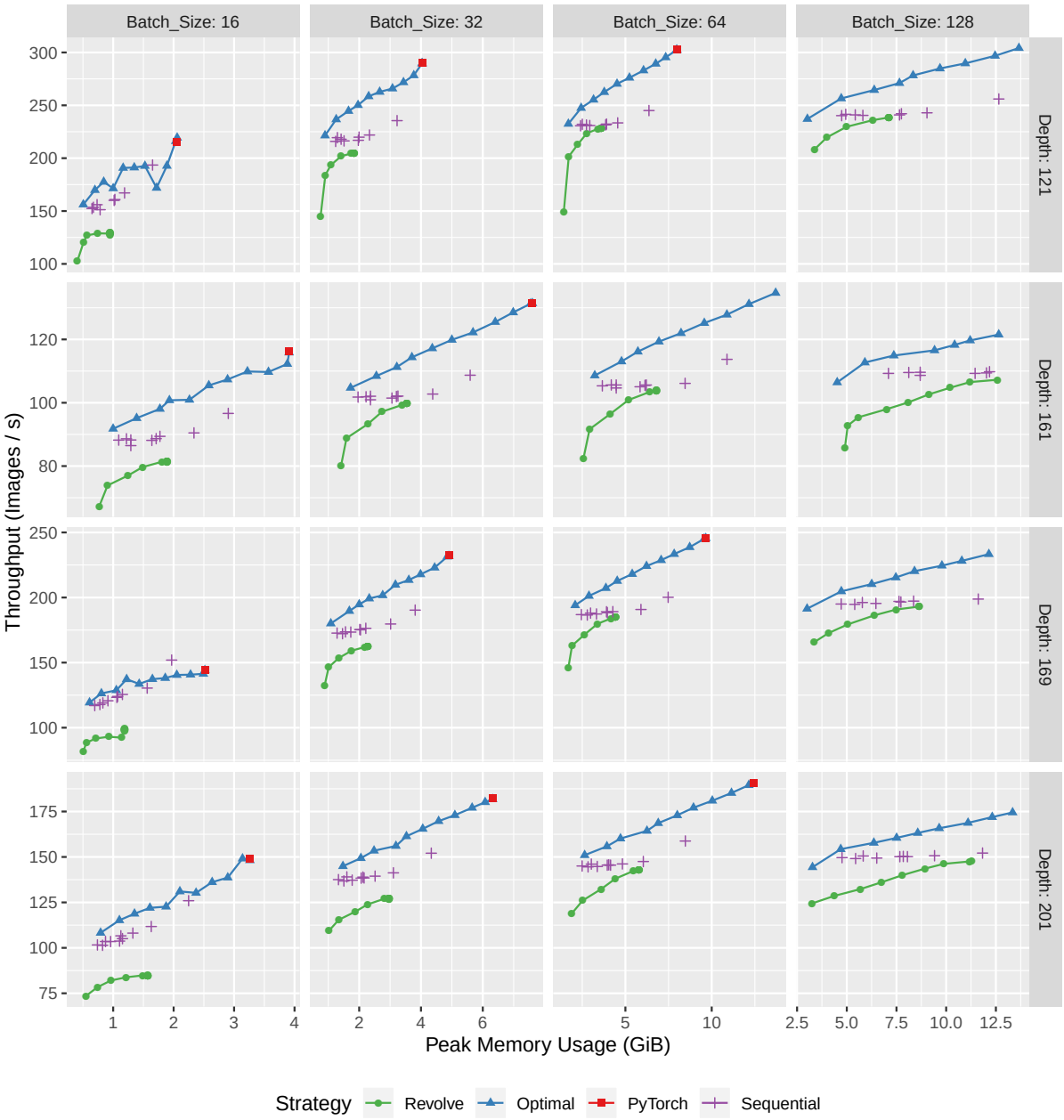


Рис. 2.11: Results for DenseNet with image size 224, for different depths and batch sizes.

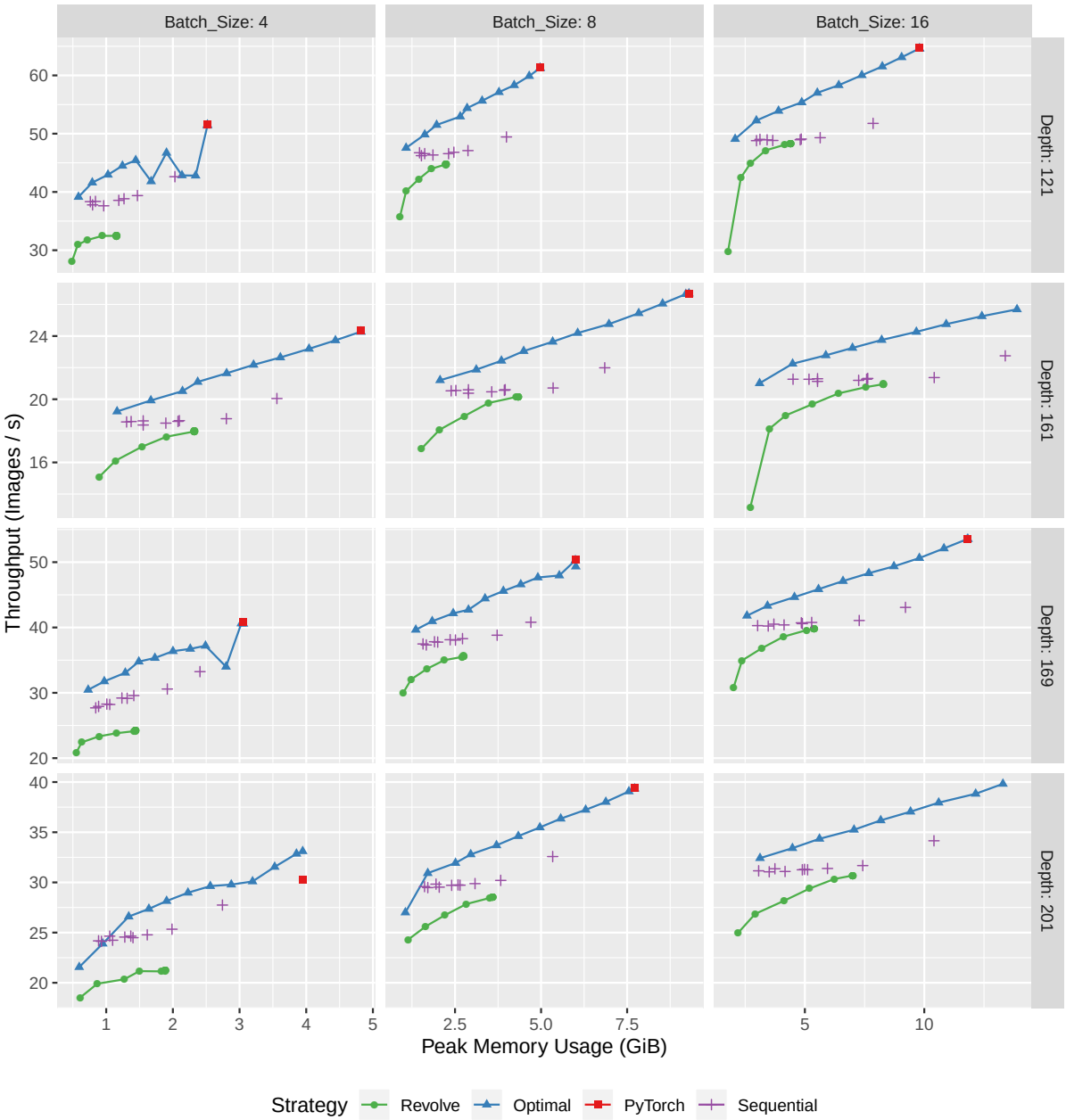


Рис. 2.12: Results for DenseNet with image size 500, for different depths and batch sizes.

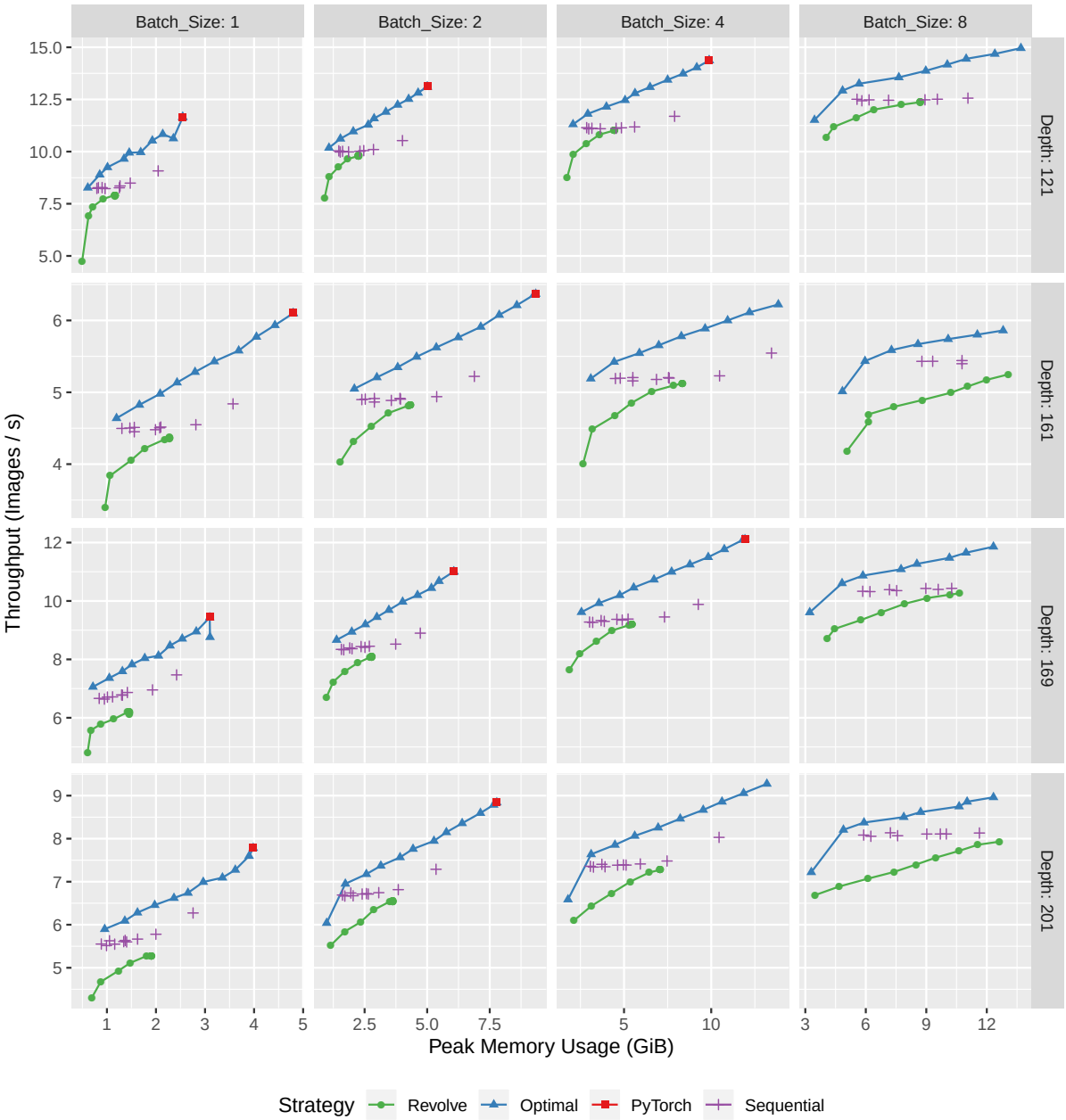


Рис. 2.13: Results for DenseNet with image size 1000, for different depths and batch sizes.

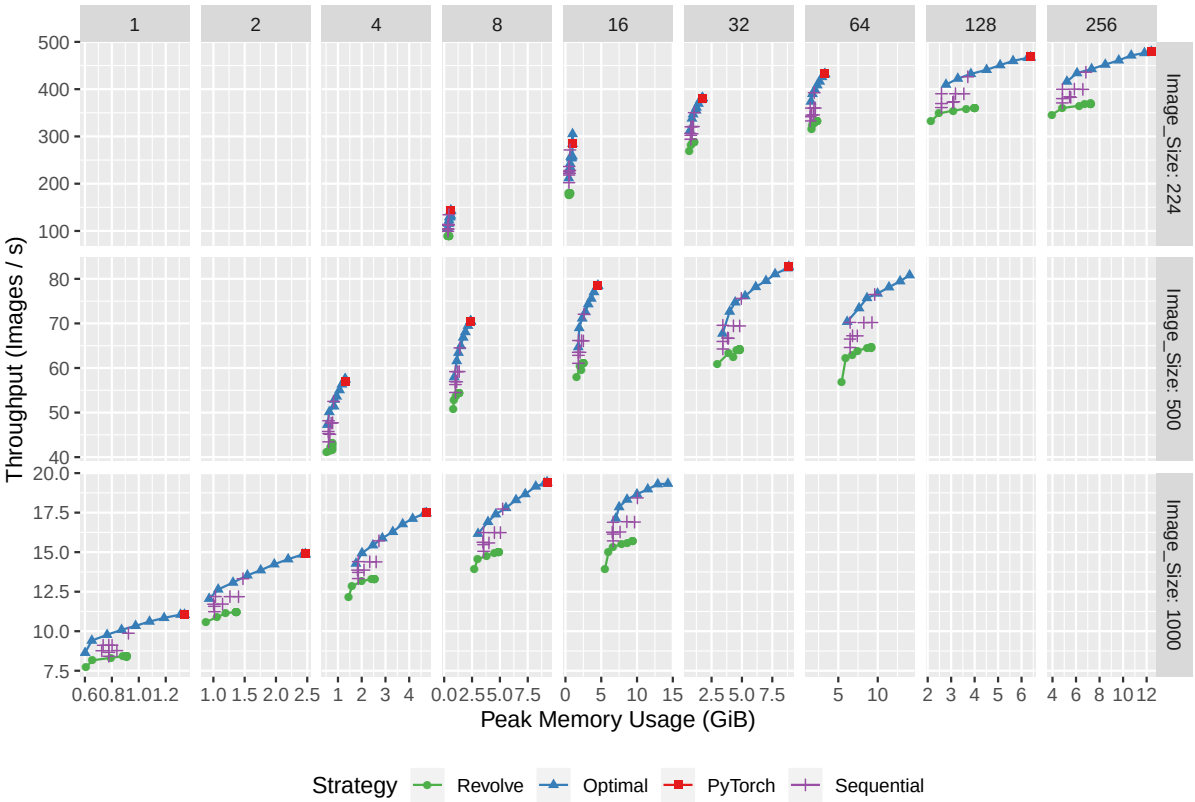


Рис. 2.14: Results for Inception v3 for different image sizes and batch sizes.

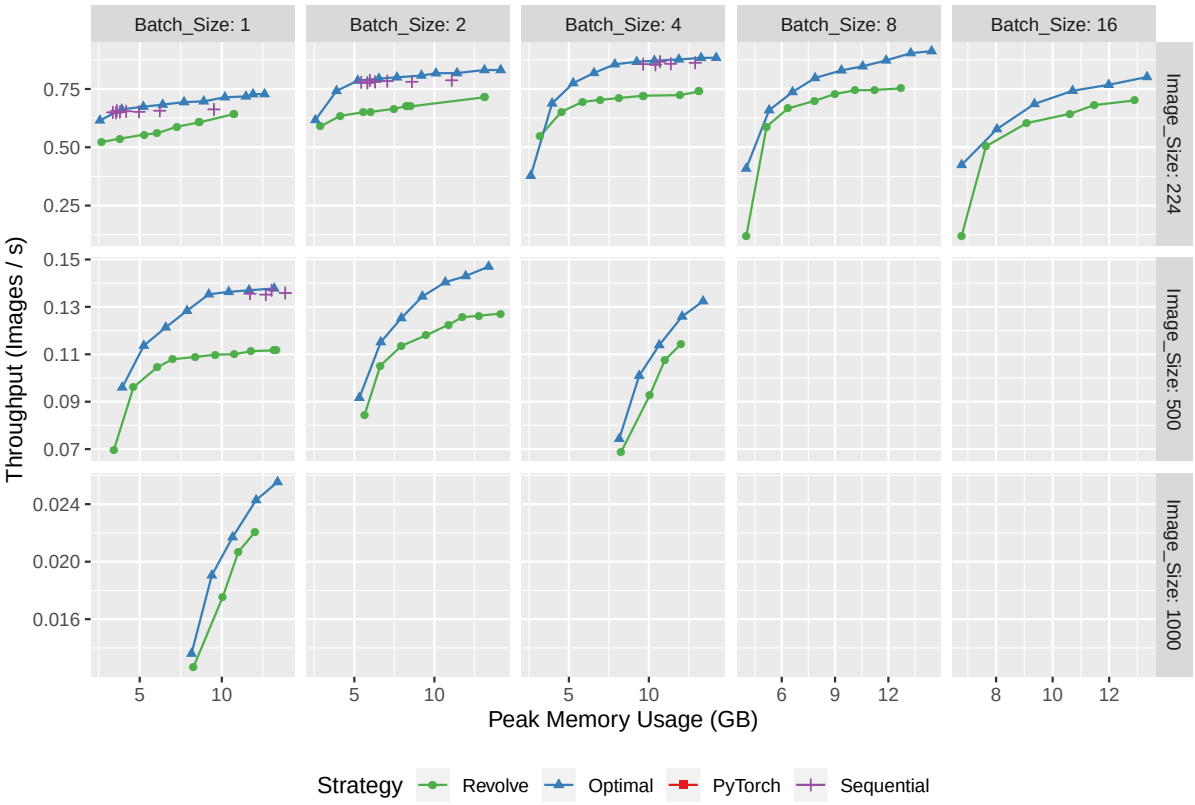


Рис. 2.15: Results for ResNet 1001, for different image sizes and batch sizes.

Часть II

Offloading

Introduction

In this part, we focus on *Offloading*, which is also known as *Memory Virtualization* or *Memory Swapping*. It 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 sent (offloaded) to the main memory 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 [88, 7], where the authors advocate the general idea and propose several static and dynamic heuristics to decide which activations should be offloaded.

More formally, the objective of Offloading is to produce a sequence of operations satisfying a given memory constraint and consisting of elementary forward, backward, offload and prefetch operations. Transfers on the PCI bus between the CPU and the GPU might slow down the final execution, when introducing idle times, which can be especially large when bandwidth is small. This problem will be solved in Chapter 3.

As discussed in Chapter 2, Rematerialization computes a sequence of valid operations consisting of elementary forward, backward and delete operations and that has a minimum execution time among all sequences that satisfy a given memory constraint. This technique works well if computations are cheap, however, it creates a non-zero overhead. As Offloading and Rematerialization both help to reduce memory usage by modifying the original schedule, by inserting additional operations, combining both approaches in the same framework can improve the final performance by mutually compensating their disadvantages. This approach will be considered in Chapter 4.

Model and Main Notations

Similarly to the previous part, we consider the training phase of sequential DNNs, as depicted on Figure 2.16. This training phase consists of two types of computations: forward operations $(F_\ell)_{1\leq\ell\leq L}$ and backward operations $(B_\ell)_{1\leq\ell\leq L}$. The forward step F_ℓ requires $a_{\ell-1}$ (or $\bar{a}_{\ell-1}$) as input, and computes a_ℓ or $\bar{a}_\ell$. The backward step B_ℓ requires $\bar{a}_\ell$, $a_{\ell-1}$ (or $\bar{a}_{\ell-1}$) and δ_ℓ as inputs, and computes $\delta_{\ell-1}$ (see Table 2.2). The distinction between a_ℓ and $\bar{a}_\ell$ is crucial in practice, since it allows us to consider computation graphs that have a sequential structure, but are not purely sequential. In this setting, it is indeed possible for F_ℓ to represent a complex operation (any Direct Acyclic Graph of layers), and thus $\bar{a}_\ell$ contains all the intermediate activations produced by F_ℓ including a_ℓ that are necessary to compute B_ℓ, whereas a_ℓ only contains the output activation of F_ℓ, that will be used by $F_{\ell+1}$. In general, a_ℓ can therefore be much smaller than $\bar{a}_\ell$. Finally, for convenience let us set that data sizes $\bar{a}_0 = a_0$ (normally the input of the chain is not produced by some other network and therefore this data does not contain something else than the input tensor) and also the gradient size $\delta_{L+1} = 0$ (the initial gradient of the backward propagation is just a scalar equal to 1, see Figure 2.16).

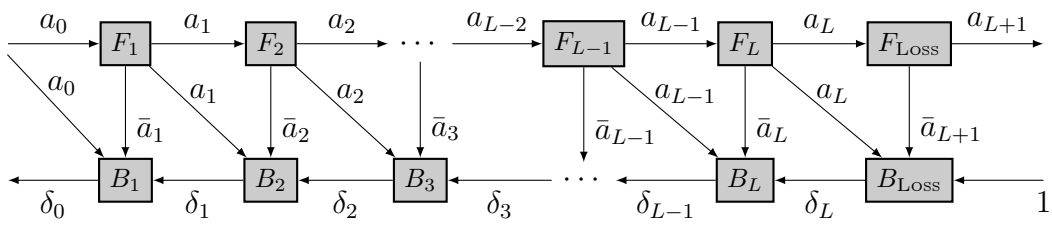


Рис. 2.16: Data dependencies induced the training phase of Sequential Deep Neural Networks.

Before the execution starts, only the input sample a_0 is present in the memory. The objective of the elementary training phase is to perform the whole computation and to obtain δ_0 from a_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_ℓ} the time to process F_ℓ, and u_{B_ℓ} the time to process B_ℓ, which are definite for a fixed input size (*i.e.* fixed batch size and image size). As mentioned before, the training phase is very memory intensive: since activations $\bar{a}_\ell$ are needed for the backward phase, all $\bar{a}_\ell$ values should be stored during the forward phase, and they can only be freed once their corresponding B_ℓ operation has been performed.

	Operation	Input	Output	Time	Memory overhead
F_ℓ^{all}	Forward and save all	$\{a_{\ell-1}\}$ $\{\bar{a}_{\ell-1}\}$	$\{a_{\ell-1},\ \bar{a}_\ell\}$ $\{\bar{a}_{\ell-1},\ \bar{a}_\ell\}$	u_{F_ℓ}	o_{F_ℓ}
F_ℓ^{ck}	Forward and materialize input	$\{a_{\ell-1}\}$ $\{\bar{a}_{\ell-1}\}$	$\{a_{\ell-1},\ a_\ell\}$ $\{\bar{a}_{\ell-1},\ a_\ell\}$	u_{F_ℓ}	o_{F_ℓ}
$F_\ell^{\varnothing}$	Forward without saving	$\{a_{\ell-1}\}$	$\{a_\ell\}$	u_{F_ℓ}	o_{F_ℓ}
B_ℓ	Backward step	$\{\delta_\ell,\ \bar{a}_\ell,\ a_{\ell-1}\}$ $\{\delta_\ell,\ \bar{a}_\ell,\ \bar{a}_{\ell-1}\}$	$\{\delta_{\ell-1}\}$ $\{\delta_{\ell-1},\ \bar{a}_{\ell-1}\}$	u_{B_ℓ}	o_{B_ℓ}

Таблица 2.2: Generic operations available in DL frameworks.

We further associate the notations used for activations a_ℓ, $\bar{a}_\ell$ and gradients δ_ℓ with their respective memory sizes, *i.e.* we use $\bar{a}_\ell$ to denote both a tensor representing an activation of layer ℓ and also its data size. To perform an operation (either F_ℓ or B_ℓ), it is necessary to have all inputs stored into memory, to reserve the memory space to store the output, and to reserve additional space for the temporary memory usage of the operation, denoted with o_{F_ℓ} for F_ℓ and o_{B_ℓ} for B_ℓ.

One way to decrease memory usage is to use Rematerialization, which has been discussed in the previous part. The alternative way is to use Offloading. 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 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. A similar 2-level memory hierarchy has been considered by [88, 7] as well. More complex architectures (*e.g.* k-level memory hierarchy for an arbitrary k or multiple GPUs and one CPU) are out of scope of this work and they will be left for future work.

In order to express this mechanism formally, we introduce two additional operations O_ℓ and P_ℓ that correspond respectively to offloading to and prefetching from CPU memory. These operations O_ℓ and P_ℓ can be applied both to $a_{\ell-1}$ and $\bar{a}_{\ell-1}$, depending on which type of activation has been produced. Their implementation is feasible, by using CUDA streams, which asynchronously offload and prefetch activations from and to the CPU memory, while independent computations are performed.

In this part, we consider two problems. We start with a simpler problem of minimizing the execution time with the limited memory M_{GPU} using Offloading to deal with the memory limit, but without recomputations. When no recomputations are allowed, only F_ℓ^{all} are possible during forward propagation, so that after loss computation the backward operations can have all the necessary input. In this case, all forward operations generate only activations of type $\bar{a}_\ell$, therefore only these activations can be sent to the CPU (and $\bar{a}_0 = a_0$, which is a chain input). In addition, we make the following assumptions:

Assumptions 1. *Common Offloading assumptions are the following:*

Assumptions on communications:

O.1 *Only one transfer can occur at a time (indeed, transferring more than one object at a time would not help to release memory faster);*

O.2 *Transferring x units of data takes x/β amount of time, i.e. bandwidth is fully used;*

O.3 *Transfers and computations can be completely overlapped except for synchronizations that take place either when the GPU needs to wait for memory releases to resume computations or when some backward B_ℓ waits for $\bar{a}_{\ell-1}$ to be prefetched);*

O.4 *Transfers do not affect computation costs u_{F_ℓ} and u_{B_ℓ} for all layers $1 \leq \ell \leq L+1$.*

Basic properties:

O.5 *δ_ℓ are not transferred: without recomputations after being produced with $B_{\ell+1}$ they are immediately consumed with B_ℓ, thus there is no interest in sending them from the GPU;*

Simplifying assumptions:

O.6 *Weights are not transferred. Though it creates an additional opportunity to reduce memory peaks, we concentrate in our work primarily on activations. Adding this option into an optimization problem can be done in future works.*

Implementation limitation:

O.7 *$\bar{a}_\ell$ needs to be stored in memory in its entirety throughout the transfer: during offloading, the memory is only released after the transfer completes, and during prefetching, the memory is reserved as soon as the transfer begins.*

We can now state the decision problem associated to Offloading, considered in Chapter 3.

Problem 7 (Offloading $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$)**.** Consider a training phase with L operations, corresponding to a chain as depicted on Figure 2.16 with processing times and memory usage described in Table 2.2. Using F_ℓ^{all}, B_ℓ, O_ℓ and P_ℓ and under Assumptions 1, 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 ?

The second problem that we are going to consider in Chapter 4 is to solve the minimization problem under memory limit M_{GPU} using both recomputations and offloading techniques. Consequently, our goal is to find an optimal sequence that can consist of F_ℓ^{all}, B_ℓ, F_ℓ^{ck}, $F_\ell^{\varnothing}$, O_ℓ and P_ℓ operations to compute δ_0 from a_0.

Problem 8 (Combination of Offloading and Rematerialization $\textsc{OffRemat}_{int}(L, M_{\text{GPU}}, \beta)$)**.** Consider a training phase with L operations, corresponding to a chain, depicted on Figure 2.16 with processing times and memory usage described in Table 2.2. Using all operations from Table 2.2 together with O_ℓ and P_ℓ and under Assumptions 1, 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 ?

Problems 7 and 8 are stated as decision problems. Both problems can be reformulated as optimization problems, where it is required to find a schedule with a minimal T satisfying constraints. Further, $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$ and $\textsc{OffRemat}_{int}(L, M_{\text{GPU}}, \beta)$ can refer to both decision and optimization problems.

Глава 3

Offloading for Heterogeneous Chains

In this chapter, we analyse $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$, which consists in finding a way to process a chain from Figure 2.16 with high memory demand using Offloading. First, we introduce the general properties of optimal solutions for this problem in Section 3.1.1. We prove that this problem is NP-complete in the strong sense in Section 3.1.2. We further present different relaxations of this problem and provide the optimal solutions for each of them (see Sections 3.2 and 3.3). At the same time, being able to solve the relaxed problems allows us to construct efficient heuristics for $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$, which is demonstrated experimentally in Section 3.4.

3.1 Preliminary Analysis

3.1.1 Preliminary Results and Lower Bound

To begin with, we establish several properties of the optimal offloading schedules. The first one is *no-wait* policy, or performing offloading as early as possible (or as late as possible for prefetching). In this context, for example, as early as possible means that the offloading should start immediately after the data is produced and the communication link is available, while as late as possible means that the prefetching should be scheduled in the way that its completion coincides with the start of the backward operation that uses the prefetched data. This strategy sustains low memory consumption in the middle of the execution, when the memory peak usually takes place, consequently prevents from unnecessary idle times caused by late memory releases (or early memory allocations).

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 computation and data offloading is performed as early as possible, and each data prefetching as late as possible.*

Доказательство. If activation a_ℓ is chosen for offloading, then postponing its offloading start can result in the higher memory usage on the GPU within the introduced transfer delay. This in turn can limit the number of feasible solutions because of higher memory

charge, while it does not help to reduce the total execution time, as idle times occur only when the memory releases are performed after the end of transfers. The symmetrical situation is with prefetching. Putting prefetching earlier inflicts the higher memory consumption, because prefetching makes GPU memory occupation increase. Therefore, there are less feasible solutions, whereas the total execution time does not decrease (making sure that prefetching should end before the start of the corresponding backward is enough to guarantee that the backward task is executed as early as possible). □

In the context of $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$, we apply only F_ℓ^{all}, saving all the previous activations. Given the information from Table 2.2, it is easy to compute the total amount of used memory during the execution of each operation. We denote by $\mathcal{M}_{F_\ell}$ or $\mathcal{M}_{B_\ell}$ the minimal amount of data required to be stored on the GPU to perform F_ℓ^{all} or B_ℓ respectively, which is composed of its input size (except for $\bar{a}_{\ell-1}$ that we consider separately), its output size, its temporary memory usage and additionally a currently stored gradient. Let us additionally denote as M_{peak} the memory peak that is achieved when executing $F_1^{all} \dots F_L^{all} F_{L+1}^{all} B_{L+1} \dots B_1$ if nothing is offloaded:

$$\mathcal{M}_{F_\ell} = o_{F_\ell} + \bar{a}_\ell + \delta_{L+1} = o_{F_\ell} + \bar{a}_\ell, \tag{3.1}$$

$$\mathcal{M}_{B_\ell} = o_{B_\ell} + \bar{a}_\ell + \delta_\ell + \delta_{\ell-1}, \tag{3.2}$$

$$M_{peak} = \max_{1 \le \ell \le L+1} \left\{ \max\left(\mathcal{M}_{B_\ell}, \mathcal{M}_{F_\ell}\right) + \sum_{k<\ell} \bar{a}_k \right\}. \tag{3.3}$$

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

Доказательство. The proof is trivial, since any valid schedule must process the operation that achieves the memory peak while using at most M_{GPU} memory on the computing device. □

Finally, we give the result on the general lower bound for $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$.

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

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

3.1.2 Complexity Results

Theorem 7. $\textsc{Off}_{int}(L, M_{GPU}, \beta)$ *is strongly NP-complete.*

Доказательство. $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$ clearly belongs to NP: given the start time of all forward and backward operations, and the set of offloaded data with the corresponding

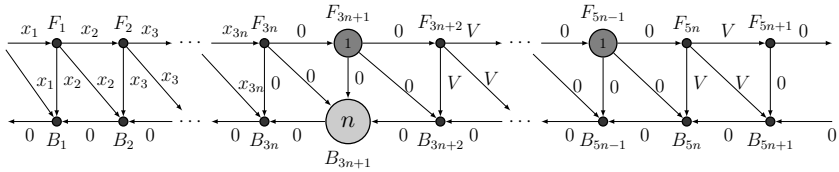


Рис. 3.1: The instance of $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$ used in the reduction from 3-Partition. Activation sizes are indicated on the edges and non-zero execution times of operations are written inside the corresponding nodes.

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 $\{x_1, x_2, \ldots, x_{3n}\}$ such that $\sum_i x_i = nV$ and $V/4 < x_i < V/2, \forall i$, is it possible to partition it into n parts $\{S_1, \ldots, S_n\}$ so that for any $j \leq n$, $|S_j| = 3$ and $\sum_{i \in S_j} x_i = V$. This problem is known to be NP-complete in the strong sense.

Given an instance of 3-Partition, we consider the following instance of $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$, depicted on Figure 3.1:

- $L + 1 = 5n + 1$, $\beta = V$, $M_{\text{GPU}} = nV$, $T = 2n$;
- $u_{F_i} = 0$ and $\bar{a}_{i-1} = x_i$ for $1 \leq i \leq 3n$;
- $\bar{a}_{3n} = 0$;
- $u_{F_i} = 1$ and $\bar{a}_i = 0$ for $i = 3n + 2k - 1, 1 \leq k \leq n$;
- $u_{F_i} = 0$ and $\bar{a}_i = V$ for $i = 3n + 2k, 1 \leq k \leq n$;
- $u_{F_{5n+1}} = 0$ and $\bar{a}_{5n+1} = 0$;
- $u_{B_i} = 0$ and $\delta_i = 0$ for all i, except $u_{B_{3n+1}} = n$.

We claim that this instance can be scheduled in time $T = 2n$ if and only if the 3-Partition instance has a solution.

Let us first assume that there exists a solution to the 3-Partition instance, *i.e.* sets $(S_j)_{1 \leq j \leq n}$ such that $\sum_{i \in S_j} x_i = V$. We can build a schedule that starts $F_{3n+2k-1}$ at time $k - 1$ for $1 \leq k \leq n$, and executes B_{3n+1} from time n to time $2n$. At time 0, before the execution of F_{3n+1}, the memory usage is exactly $nV = \sum_i x_i$. During the execution of $F_{3n+2k-1}$, 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 $F_{3n+2k-1}$ is thus $(n-1)V$, which allows F_{3n+2k} to start without delay. At the end of the forward phase, the memory is filled with n activations of size V. At the beginning of B_{3n+1}, the memory is empty: all activations of size x_i can be prefetched during the execution of B_{3n}, allowing the backward phase to finish without idle time. This schedule induces no idle time, and finishes in time exactly $T = 2n$.

Let us now assume that there exists a valid schedule of duration $T = 2n$, *i.e.* without any idle time on the processing device. For $j \leq n$, let us define the set S_j as the indices of the activations whose transfers are included in the execution of $F_{3n+2j-1}$. Since F_{3n+2} starts immediately after the end of F_{3n+1}, and since memory is only released once the

transfer is completed, the amount of data sent during F_{3n+1} is at least V. Since $\beta = V$ and $u_{F_{3n+1}} = 1$, the amount of data is exactly V, thus $\sum_{i \in S_1} x_i = V$. The same argument applies for all $j \leq n$, 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 7, it 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 $3n$ activations need to be offloaded, but finding the optimal ordering is hard. Because of this negative complexity result, we study two different relaxations of $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$ in the next sections, by relaxing the constraints stating that activations should be sent in entirety before the corresponding memory can be released. In such scenarios, all activations should be sent as soon as they are computed, *i.e.* in increasing order of their indices. This allows us to compute optimal solutions in reasonable time, and the resulting algorithms can then be used as heuristic solutions for $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$.

3.2 Fractional Relaxation

In a first relaxation, let us assume 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.

Assumptions 2. *For the following problem we consider Assumptions 1(**O.1**-**O.6**). Moreover we relax Assumption **O.7** as follows:*

O.7' *$\bar{a}_\ell$ can be offloaded partially to the CPU, and partial discards on the GPU are allowed and can be performed as soon as the memory needs to be freed.*

Problem 9 (Fractional Relaxation $\textsc{Off}_{frac}(L, M_{\text{GPU}}, \beta)$)**.** Consider a training phase with L operations, corresponding to a chain, depicted on Figure 2.16 with processing times and memory usage described in Table 2.2. Using F_ℓ^{all}, B_ℓ, O_ℓ and P_ℓ and under Assumptions 2, 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 ?

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.

3.2.1 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 $\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_k$ (where the last one can be partially offloaded) for some value $k \geq 0$.

- A schedule is said *ordered* if the data is offloaded in order of increasing indices, and prefetched in order of decreasing indices.

Lemma 5. *Under Assumptions 2 any valid solution $\mathcal{S}$ can be transformed into an eager and ordered solution $\mathcal{S}'$ with the same makespan.*

Доказательство. 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 $\bar{a}_0$ (since it is possible to stop any communication at any time, using several intervals to transfer $\bar{a}_0$ is not a problem), the next ones are used to offload $\bar{a}_1$, 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 $\bar{a}_\ell$ 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, which completes the proof. □

3.2.2 Greedy Algorithm

According to this result, we restrict the search to eager and ordered schedules. It is thus sufficient to find the amount of offloaded data that results in the smallest makespan. The next result shows that it is best to offload the least possible amount of data.

Lemma 6. *Let $\mathcal{S}$ and $\mathcal{S}'$ be no-wait, ordered and eager schedules that offload a quantity of data Q and Q' respectively, with $Q < Q'$. Then, under Assumptions 2 the makespan of $\mathcal{S}$ is not larger than the makespan of $\mathcal{S}'$.*

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

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

The schedule $\mathcal{S}''$ is thus a valid, eager and ordered schedule that offloads a quantity of data Q ($\mathcal{S}''$ is not necessarily no-wait). 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 $\mathcal{S}'$.

□

With these two lemmas, 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 8. *For a given instance, the no-wait, eager, ordered schedule that offloads a quantity $M_{peak} - M_{GPU}$ of data is optimal for $\textsc{Off}_{frac}(L, M_{GPU}, \beta)$.*

By rounding up the number of offloaded activations, this result provides a heuristic for the original integral $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$, that we call Greedy. The Greedy heuristic returns the no-wait, eager, ordered schedule that offloads (entirely) the first k activations, where k is the smallest index such that $\sum_{\ell \leq k} \bar{a}_\ell \geq M_{peak} - M_{\text{GPU}}$.

However, it may happen that this Greedy schedule offloads too much data because of the rounding procedure, which is reflected on its performance shown in Section 3.4. In the next section, we thus analyze a more sophisticated relaxation in order to obtain a more precise algorithm.

3.3 Fractional Communications

Let us now consider another relaxation of $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$, in which an activation must be either entirely offloaded or not offloaded at all. However, it is still allowed to do partial discards: at any time the already offloaded part can be deleted from GPU memory and memory for prefetched data can be allocated in several steps.

Assumptions 3. *For the following problem we consider Assumptions 1(**O.1**-**O.6**) and we relax Assumption **O.7** as follows:*

O.7" *$\bar{a}_\ell$ can only be offloaded in its entirety to the CPU, but partial discards on the GPU are allowed and can be performed as soon as the memory needs to be freed.*

Problem 10 (Fractional Communications $\textsc{Off}_{comm}(L, M_{\text{GPU}}, \beta)$)**.** Consider a training phase with L operations, corresponding to a chain, depicted on Figure 2.16 with processing times and memory usage described in Table 2.2. Using F_ℓ^{all}, B_ℓ, O_ℓ and P_ℓ and under Assumptions 3, 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 ?

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.

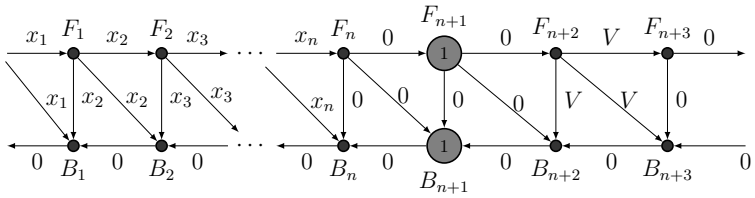


Рис. 3.2: The instance of $\textsc{Off}_{comm}(L, M_{\text{GPU}}, \beta)$ that corresponds to the 2-Partition problem with values x_i.

3.3.1 Complexity

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 7. *Under Assumptions 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 5: transforming $\mathcal{S}$ using the correct order provides a valid schedule. The result is weaker, because an eager schedule that offloads the same data might not be valid for $\textsc{Off}_{comm}(L, M_{\text{GPU}}, \beta)$ (the last activation might not be fully offloaded).

Theorem 9. $\textsc{Off}_{comm}(L, M_{GPU}, \beta)$ *is NP-complete in the weak sense.*

Доказательство. $\textsc{Off}_{comm}(L, M_{\text{GPU}}, \beta)$ clearly belongs to NP, in the same way as $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$.

Let us prove that it is NP-hard by reduction for the 2-Partition problem, which can be stated as: given n positive integers $x_1, x_2, \ldots, x_n$ such that $\sum_i x_i = 2V$, is is possible to partition them in two subsets S_1 and S_2 such that $\sum_{i \in S_1} x_i = \sum_{i \in S_2} x_i = V$?

Given an instance of 2-Partition, let us consider an instance $\mathcal{I}$ of $\textsc{Off}_{comm}(L, M_{\text{GPU}}, \beta)$, depicted on Figure 3.2:

- $L + 1 = n + 3$, $\beta = V$, $M_{\text{GPU}} = 2V$, $T = 2$;
- $\delta_i = 0$ for all i;
- $u_{F_i} = u_{B_i} = 0$ and $\bar{a}_{i-1} = x_i$ for $1 \leq i \leq n$;
- $u_{F_{n+1}} = u_{B_{n+1}} = 1$ and $\bar{a}_n = 0$;
- $u_{F_{n+2}} = u_{B_{n+2}} = 0$ and $\bar{a}_{n+1} = 0$;
- $u_{F_{n+3}} = u_{B_{n+3}} = 0$ and $\bar{a}_{n+2} = V$.

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 $\sum_{i \in S_1} x_i = V$. It is thus possible to offload all the corresponding activations x_i during operation F_{n+1} (since $\beta = V$), and then to prefetch them during operation B_{n+1}.

This allows us to perform $F_{n+2}, F_{n+3}, B_{n+3}, B_{n+2}$ immediately after F_{n+1}, since only a quantity V of data from the first activations is stored. Once B_n has been performed, all other B_ℓ can be performed as well, which results in a schedule of length 2.

Conversely, 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 $Q = \sum_{i \in S_1} x_i$. Since $M_{peak} = 3V$, it is clear that $Q \geq V$. Since $\beta = V$, 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. □

3.3.2 Structure of Optimal Solutions

According to Lemma 7, our objective is now to find the best ordered schedule. In this section, we derive a dynamic program that explores all possible ordered and no-wait schedules and computes makespans for each covered case.

The principal idea of our dynamic program is to move across layers in increasing order and, for each layer ℓ, it needs to decide whether its input should be offloaded or not. This decision has several impacts:

- it may impose some idle time during the forward phase and the backward phase when overlapping the communications with computations;
- it can also contribute to an idle time between the phases, when loss is computed;
- it affects the feasibility of processing the next operations.

Consequently, different choices of offloaded activations may lead to different makespans. The goal of the dynamic program is to detect the schedule with the minimal makespan out of all available schedules.

To take into account the effects caused by the previous decisions, we need to define a set of variables that describes the state of the system at any instant. On the one hand, this set of variables should be as small as possible, since it has a direct influence on the size of the data structure and on the computing time to solve the dynamic program. On the other hand, these variables must be chosen wisely and they must contain enough information to make decisions for subsequent layers and be updated according to these decisions. To evaluate correctly memory constraints and to compute idle times, it is important to know how memory may vary between layers i and $i+1$ during communications: from some minimal memory occupation till maximal memory occupation. Both can be described for the forward and the backward phases using only three state variables: in addition of the index i of the currently considered layer we use A_i, Δ_{F_i} and Δ_{B_i}.

- A_i denotes the total GPU memory occupied by the saved values among $\bar{a}_0, \ldots, \bar{a}_{i-2}$ that are not transferred to the CPU;
- Δ_{F_i} denotes the amount of data from $\bar{a}_0, \ldots, \bar{a}_{i-2}$ that the schedule still needs to offload after F_{i-1}, if there is no data ready for offloading then it takes a negative value that indicates how long the communication link was idle since the last offload;
- Δ_{B_i} denotes the amount of data $\bar{a}_0, \ldots, \bar{a}_{i-2}$ that the schedule should prefetch before

starting B_{i-1}, if there is no data ready for prefetching then it takes a negative value that indicates how long the communication link stays idle until the next prefetch.

The negative values of Δ_{F_i} and Δ_{B_i} are necessary for computing the idle time that take place at loss calculation when connecting offloading and prefetching. For example, knowing how much time the communication link is idle after loss can help with overlapping of the remaining offloading with some backward steps, thus avoiding an unnecessary synchronization of offloading with the end of forward propagation. We provide more detailed analysis of this idle time and how it depends on the values of $\Delta_{F_{L+2}}$ and $\Delta_{B_{L+2}}$ in Section 3.3.2.2.

With these state variables, we can compute M_{F_i}, the memory on the GPU after executing F_{i-1}, and M_{B_i}, the memory on the GPU before executing B_{i-1} (excluding δ_{i-1}) that are given by

$$M_{F_i} = A_i + \Delta^+_{F_i} + \bar{a}_{i-1}, \tag{3.4}$$

$$M_{B_i} = A_i + \Delta^+_{B_i} + \bar{a}_{i-1}, \tag{3.5}$$

where we use the convention:

$$x^+ = \max\{x, 0\}.$$

These memory values represent the maximal memory occupation between layers i and $i+1$ for the forward and backward phases respectively, and are used for computing idle times when overlapping communications with computations, according to Lemma 8. However, to estimate the feasibility of the scheduled operations, we also need to know the memory after everything has been offloaded, which is either $A_i + \bar{a}_{i-1}$ (when $\bar{a}_{i-1}$ stays on the GPU) or A_i (when $\bar{a}_{i-1}$ is sent to the CPU), which we use to obtain the maximal memory available on the GPU.

The updates of the state variables follow simple rules:

- if no new data is offloaded (resp. prefetched), then Δ_{F_i} (resp. Δ_{B_i}) is constantly decreasing from index i to index $j, j > i$ at speed β;
- if new data has to be offloaded (resp. prefetched), then the data size is added into $\Delta^+_{F_i}$ (resp. $\Delta^+_{B_i}$), *i.e.* if Δ_{F_i} (resp. Δ_{B_i}) is negative before the update, then it should be first reset to zero and then updated by the new data size;
- A_i is updated if new data is saved on the GPU (without being later offloaded to the CPU).

The next lemma describes the general scheme for optimally overlapping the communications with any sequence of operations. Lemma 8 is valid for both prefetching and offloading, as their behavior is symmetrical. Offloading takes place at the beginning of the sequence, making available memory increase at a speed of β from its initial value m_{min}. On the contrary, prefetching is done at the end of the sequence, making available memory decrease at the speed β until reaching m_{min}. Performing offloading as soon as possible and prefetching as late as possible allows us to have more available memory for the middle of the execution. Note that m_{min} could be equivalently replaced with $M_{\text{GPU}} - M_{\max}$, where $M_{\max}$ denotes the maximal memory occupation (either M_{F_i} or M_{B_i}). We apply this lemma to compute the idle time when overlapping communications

with just one operation at a time and also the idle time when overlapping a group of operations with communications. The last case is especially helpful when evaluating the idle time at the junction of offloading and prefetching and when considering more complex case of combined Offloading and Rematerialization.

Lemma 8. *Let us consider a fixed sequence of operations, for which the available memory increases (resp. decreases) during execution because of data offloading (resp. prefetching), with its minimum at m_{min}. Let us denote by $\mathcal{M}_o^{\mathcal{S}}$ the memory required to process operation $o \in \mathcal{S}$, and d_o the distance between o and the beginning (resp. end) of sequence $\mathcal{S}$, i.e. the cumulative duration of operations taking place before (resp. after) operation o. Then, the execution of $\mathcal{S}$ needs to be delayed by some idle time*

$$\epsilon = \max\left(\frac{\max_{o\in\mathcal{S}}(\mathcal{M}_o^{\mathcal{S}} - \beta d_o) - m_{min}}{\beta}, 0\right).$$

Доказательство. In this proof, we concentrate on offloading, since the case of prefetching is done analogously. The idle time is not zero when it is not possible to overlap entirely communications and computations because of the memory limit. During offloading, the available memory (excluding the memory needed for operations of $\mathcal{S}$) is increasing starting from m_{min}, and its current value can be found with $m(d) = m_{min} + \beta d$, where d shows the time distance to the beginning of the sequence. If these values are higher than the memory required for every operation, then no idle time takes place. However, if the memory is not large enough, then it is necessary to add more waiting time at the beginning of the execution to increase the memory at the moment when more memory is needed. Such increase is minimal when it is equal to $\max_{o\in\mathcal{S}}(\mathcal{M}_o - m(d_o))/\beta$, with $o^* = \arg\max_{o\in\mathcal{S}}(\mathcal{M}_o - m(d_o))$ corresponding to the point when the memory needs exceed the most the available memory. □

3.3.2.1 Forward and Backward Phases

Let us consider any ordered, no-wait schedule $\mathcal{S}$ that processes the chain from layer i and memory state described with some arbitrary A_i, Δ_{F_i} and Δ_{B_i}. Using Lemma 8, we find the idle times that occur during F_i and B_i executions:

$$\epsilon_{F_i} = \max\left(\frac{\mathcal{M}_{F_i} - M_{\text{GPU}} + M_{F_i}}{\beta}, 0\right) \tag{3.6}$$

and

$$\epsilon_{B_i} = \max\left(\frac{\mathcal{M}_{B_i} - M_{\text{GPU}} + M_{B_i}}{\beta}, 0\right). \tag{3.7}$$

The values for these equations can be obtained from the state variables and general information on the chain.

Let us first remark that for $\mathcal{S}$ to be a valid schedule, there should be enough memory to process F_i and B_i at least when data Δ_{F_i} and Δ_{B_i} reside entirely in the CPU. Thus,

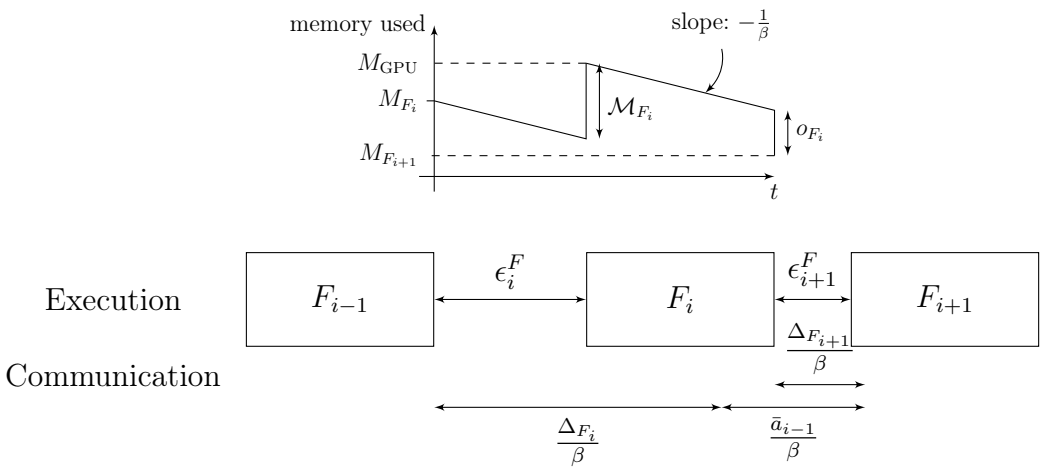


Рис. 3.3: Notations used in the Forward phase, assuming $\bar{a}_{i-1}$ is offloaded.

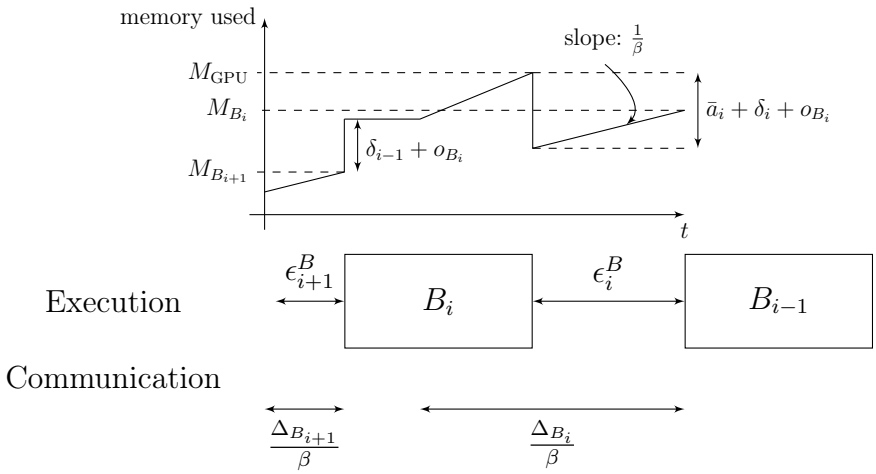


Рис. 3.4: Notations used in the Backward phase. Case when prefetching Δ_{B_i} cannot start because of memory requirement for B_i.

all valid schedules should satisfy the next constraints:

$$\mathcal{M}_{F_i} \leq M_{\text{GPU}} - A_i - \bar{a}_{i-1}, \tag{3.8}$$
$$\mathcal{M}_{B_i} \leq M_{\text{GPU}} - A_i - \bar{a}_{i-1}. \tag{3.9}$$

Let us now derive recursive equations to obtain A_{i+1}, $\Delta_{F_{i+1}}$ and $\Delta_{B_{i+1}}$. These equations depend on whether $\bar{a}_{i-1}$ is offloaded in $\mathcal{S}$ or not.

If $\bar{a}_{i-1}$ is offloaded, then the amount of data ready to be offloaded after F_{i-1} is $\Delta^+_{F_i} + \bar{a}_{i-1}$, whereas the amount of data that should be prefetched before B_{i-1} is at least $\bar{a}_{i-1}$. Until the end of F_i, the amount of data that can be offloaded is at most $(\epsilon_{F_i} + u_{F_i})\beta$, while within the execution of B_i the amount of data that can be prefetched is $(\epsilon_{B_i} + u_{B_i})\beta$ Hence, we obtain

$$\Delta_{F_{i+1}} = \Delta^+_{F_i} + \bar{a}_{i-1} - (\epsilon_{F_i} + u_{F_i})\beta, \tag{3.10}$$
$$\Delta_{B_{i+1}} = \max\left(\Delta_{B_i} - (\epsilon_{B_i} + u_{B_i})\beta, 0\right) + \bar{a}_{i-1}, \tag{3.11}$$
$$A_{i+1} = A_i. \tag{3.12}$$

If $\bar{a}_{i-1}$ is not offloaded, we can write similar equations, except that $\bar{a}_{i-1}$ is not added to the amount of data to be offloaded or prefetched, thus $\Delta_{F_{i+1}}$ and $\Delta_{B_{i+1}}$ will simply decrease. This yields

$$\Delta_{F_{i+1}} = \Delta_{F_i} - (\epsilon_{F_i} + u_{F_i})\beta, \tag{3.13}$$
$$\Delta_{B_{i+1}} = \Delta_{B_i} - (\epsilon_{B_i} + u_{B_i})\beta, \tag{3.14}$$
$$A_{i+1} = A_i + \bar{a}_{i-1}. \tag{3.15}$$

3.3.2.2 Idle Time between Phases

With these derivations, we have accounted for all idle times in schedule $\mathcal{S}$, except for a possible idle time ϵ_G between the end of F_{L+1} and the start of B_{L+1}. Several situations are possible. If $\mathcal{S}$ performs no offloading after F_{L+1}, and no prefetching before B_{L+1} (*i.e.* $\Delta_{F_{L+2}}$ and $\Delta_{B_{L+2}} \leq 0$), then no idle time occurs and $\epsilon_G = 0$. If $\mathcal{S}$ performs both kind of transfers (*i.e.* both $\Delta_{F_{L+2}}$ and $\Delta_{B_{L+2}}$ are positive), then the idle time corresponds to transferring the total data, $\epsilon_G = \frac{\Delta_{F_{L+2}} + \Delta_{B_{L+2}}}{\beta}$.

Otherwise, if for example $\Delta_{F_{L+2}} > 0$ and $\Delta_{B_{L+2}} \leq 0$, the schedule $\mathcal{S}$ can continue 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 $Av_B = -\frac{\Delta_{B_{L+2}}}{\beta}$ the time between the start of B_{L+1} and the start of the first prefetch operation, and let us set $U^B_j = \sum_{i=j+1}^{L+1} u_{B_i}$. According to 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 $U^B_j < Av_B$. Moreover, starting B_j requires to have at least

$\mathcal{R}_{B_j} = \delta_{j-1} + \delta_j + o_{B_j} - \sum_{i \geq j+1} \bar{a}_i$ available memory, where we account for the removal of $\bar{a}_i$. Thus, applying Lemma 8, we get

$$\epsilon_G \geq \max \left(\frac{\max\limits_{\substack{j \leq L+1 \\ U_j^B < Av_B}} \{\mathcal{R}_{B_j} - U_j^B \beta\} - M_{\text{GPU}} + M_{F_{L+2}}}{\beta}, 0 \right).$$

Another lower bound on ϵ_G is given by the fact that the offloading must finish before the prefetching starts, $\epsilon_G \geq \frac{\Delta_{F_{L+2}}}{\beta} - Av_B$. These are the only constraints on ϵ_G, hence we have

$$\epsilon_G = \max \left(\frac{\Delta_{F_{L+2}}}{\beta} - Av_B, \max\limits_{\substack{j \leq L+1 \\ U_j^B < Av_B}} \frac{\mathcal{R}_{B_j} - M_{\text{GPU}} + M_{F_{L+2}}}{\beta} - U_j^B, 0 \right). \tag{3.16}$$

Finally, 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 $Av_F = -\frac{\Delta_{F_{L+2}}}{\beta}$ the time between the end of the last offload and the end of F_{L+1}, $U_j^F = \sum_{i=j+1}^{L+1} u_{F_i}$, and $\mathcal{R}_{F_j} = o_{F_j} - \sum_{i \geq j+1} \bar{a}_i$.

Algorithm 5 Dynamic Programming Algorithm for Fractional Communications

$O_i \leftarrow$ HashTable() for $1 \leq i \leq L+1$
$O_1(0,0,0) = 0$
for $i \in \{1, \ldots, L+1\}$ **do**
  for $A_i, \Delta_{F_i}, \Delta_{B_i} \in O_i$ **do**
    Compute M_{B_i} and M_{F_i} with Eq. (3.4) and (3.5)
    if Constraints (3.8) and (3.9) are satisfied **then**
      Compute $\epsilon_{F_i}, \epsilon_{B_i}$ from equations (3.6) and (3.7)
      Compute A, Δ_F, Δ_B if $\bar{a}_{i-1}$ is offloaded (equations (3.10), (3.11) and(3.12))
      $O_{i+1}(A, \Delta_F, \Delta_B) \leftarrow \min\left(O_{i+1}(A, \Delta_F, \Delta_B), O_i(A_i, \Delta_{F_i}, \Delta_{B_i}) + u_{F_i} + \epsilon_{F_i} + u_{B_i}\epsilon_{B_i}\right)$
      Compute A', Δ'_F, Δ'_B if $\bar{a}_{i-1}$ is not offloaded (equations (3.13), (3.14) and (3.15))
      $O_{i+1}(A', \Delta'_F, \Delta'_B) \leftarrow \min\left(O_{i+1}(A', \Delta'_F, \Delta'_B), O_i(A_i, \Delta_{F_i}, \Delta_{B_i}) + u_{F_i} + \epsilon_{F_i} + u_{B_i} + \epsilon_{B_i}\right)$
for $A, \Delta_F, \Delta_B \in O_{L+2}$ **do**
  Compute ϵ_G according to equations (3.17) and (3.16)
  $O_{tot}(A, \Delta_F, \Delta_B) \leftarrow O_{L+2}(A, \Delta_F, \Delta_B) + \epsilon_G$
Get $A^*, \Delta_F^*, \Delta_B^*$ that minimizes $O_{tot}(A, \Delta_F, \Delta_B)$
Backtrack in $O_{L+2}, \ldots, O_1$ to obtain optimal offload decisions

With considerations similar as above, we obtain

$$\epsilon_G = \max\left(\frac{\Delta_{B_{L+2}}}{\beta} - Av_F, \max_{\substack{j\leq L+1\\ U_j^F<Av_F}} \frac{\mathcal{R}_{F_j} - M_{\text{GPU}} + M_{B_{L+2}}}{\beta} - U_j^F, 0\right). \tag{3.17}$$

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 $A_i, \Delta_{F_i}, \Delta_{B_i}$. The exact decisions of which data from $\bar{a}_0, \ldots, \bar{a}_{i-2}$ has actually been offloaded is not required. This allows us to design dynamic programming algorithm to identify the offloading decisions that induce the smallest idle time.

3.3.3 Resulting Algorithm

3.3.3.1 Dynamic Programming Algorithm

To formalize the dynamic programming algorithm, let us define $O(i, A, \Delta_F, \Delta_B)$ as the smallest possible execution time 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 $A_i = A$, $\Delta_{F_i} = \Delta_F$, $\Delta_{B_i} = \Delta_B$.

Any schedule starts with nothing stored on the GPU without any scheduled offloads or prefetches and no idle time takes place before F_1 and after B_1, so we can define $O(1, 0, 0, 0) = 0$, and $O(1, A, \Delta_F, \Delta_B) = \infty$ for all other values of A, Δ_F, Δ_B. In order to compute $O(i, A, \Delta_F, \Delta_B)$ for all i and all relevant values of A, Δ_F, Δ_B, we use hash tables O_i indexed with (A, Δ_F, Δ_B), with the understanding that if (A, Δ_F, Δ_B) is not stored in O_i, then $O(i, A, \Delta_F, \Delta_B) = \infty$. This leads to Algorithm 5, where O_i values are used to update O_{i+1} values, with two possible cases, either with a schedule that offloads $\bar{a}_{i-1}$, or with a schedule that does not.

Once O_{L+2} is computed, O_{tot} can be found by adding the corresponding idle time ϵ_G between the forward and backward phases. Then, the smallest value in O_{tot} is the smallest possible execution time for any ordered, no-wait schedule. Finally, we can identify which offload decisions led to this solution, 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: A is between 0 and M_{GPU}, Δ_F and Δ_B may vary from $-\sum_i(u_{F_i})\beta$ for Δ_F and $-\sum_i(u_{B_i})\beta$ for Δ_B to M_{GPU}. The number of possible values is thus $O(M_{\text{GPU}}(M_{\text{GPU}} + \sum_i u_{F_i}\beta)(M_{\text{GPU}} + \sum_i u_{B_i}\beta))$, and the complexity of Algorithm 5 is $O(LM_{\text{GPU}}(M_{\text{GPU}} + \sum_i u_{F_i}\beta)(M_{\text{GPU}} + \sum_i u_{B_i}\beta))$, which is indeed pseudo-polynomial.

Theorem 10. *Under Assumptions 3, the problem of finding the minimal processing time for the chain from Figure 2.16, using operations F_ℓ^{all}, B_ℓ, O_ℓ and P_ℓ, under memory limit M_{GPU} and bandwidth β, can be solved optimally with a dynamic programming algorithm whose complexity is $O(LM_{GPU}(M_{GPU} + \sum_i u_{F_i}\beta)(M_{GPU} + \sum_i u_{B_i}\beta))$. If A, Δ_F and Δ_B are discretized using N values, then we can find an approximate solution after $O(LN^3)$ number of operations.*

Доказательство. Algorithm 5 explores all possible states, generated by different offloading choices made for each layer of the neural network. For each explored state it computes the minimal execution time associated state, therefore backtracking the solution with a global minimal makespan returns an optimal solution. □

There is a simpler way to calculate idle time between phases, by setting $\epsilon_G = \frac{\Delta_F^+ + \Delta_B^+}{\beta}$. There is no guarantee that it provides an optimal value, as it corresponds to the situation when we synchronize the end of the forward phase with the end of offloading. Despite being sub-optimal, this estimation is cheaper to calculate, as it does not require negative values of Δ_{F_i} and Δ_{B_i}. Thus in the dynamic programming where this simple estimation is applied, Δ_{F_i} and Δ_{B_i}, having smaller range of possible values, can be discretized with a better precision, which potentially can improve the final result of the algorithm.

This optimal algorithm for the fractional communications model can be turned into a heuristic for the original $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$, which we call DynProg. DynProg computes the optimal set of activations for the fractional communications model with Algorithm 5, and outputs the no-wait, ordered schedule that offloads exactly these activations.

3.3.3.2 Discretization Scheme

To keep its computing time reasonable, we also include in DynProg a discretization scheme by fixing a number N of memory slots, where each slot has size $\frac{M}{N}$. All memory sizes are then expressed as an integer number of slots, by rounding up if necessary, setting for example $\widetilde{o_{F_i}} = \left\lceil o_{F_i} \cdot \frac{N}{M} \right\rceil$. The values o_{F_i}, o_{B_i}, and δ_i are rounded up in this way.

It is also necessary to discretize the values of $u_{F_i}\beta$ and $u_{B_i}\beta$, but these values need to be rounded *down* to ensure the feasibility of the produced solution in the original problem. However, since these values are always used as partial sums $\sum_{j=1}^{i} u_{F_j}\beta$, we can perform a more precise rounding procedure: the partial sums are discretized by setting $\widetilde{\sigma_j^F} = \left\lfloor \frac{N}{M} \cdot \sum_{j=1}^{i} u_{F_j}\beta \right\rfloor$, and then the individual values can be recovered by $\widetilde{u_{F_i}\beta} = \widetilde{\sigma_{i+1}^F} - \widetilde{\sigma_i^F}$. The same procedure is applied to obtain $\widetilde{u_{B_i}\beta}$.

Finally, we take special care of the discretization of the $\bar{a}_i$ values. It is indeed crucial to obtain values as close as possible to the original values: rounding up directly the values in the same way as we did for the δ_i values ensures that the resulting solution is always feasible, but may require to offload significantly more data to ensure that all the non-offloaded activations fit into memory. However, rounding down some of the values may result in an unfeasible selection of non-offloaded data. We thus adopt an *iterative* scheme: we start with an optimistic rounding of the values, and if the resulting solution is not feasible, we choose among all discretized values that were rounded down the value $\widetilde{\bar{a}_i}$ that is the closest to its continuous version $\bar{a}_i$. Since optimal solutions of the dynamic program often choose to offload the first activations, the optimistic rounding is obtained by rounding up the partial sums $\sum_{j=0}^{i} \bar{a}_j$, in a similar way as for $u_{F_j}\beta$ values.

Further, we set $N = 500$, which allows us to obtain an approximate solution of a good precision. This helps to compute the dynamic programming over a reasonable amount of time, which for all practical cases is not higher than 1 minute. Since it is done only once whereas the neural network training may take days, this time is acceptable in practice.

3.4 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.

3.4.1 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, o_F, o_B, and the sizes of $\bar{a}_i$ and δ_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 β 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 224×224 (which is the default and minimal image size for all models of `torchvision`), 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 (Section 3.2), DynProg (based on Algorithm 5, see Section 3.3.3), AutoSwap, TFLMS and Vdnn, where the last three approaches are based on the state-of-the-art methods used in the previous works. AutoSwap [114] is a score-based heuristic that uses a weighted average of 4 priority scores to decide which activations should be offloaded in priority, where the best weight combination is obtained with Bayesian Optimization. TFLMS [64] 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 is parameterized with the number of tensors to be offloaded and how many layers in advance the data should be prefetched. We present TFLMS results on activation offloading obtained with the hyperparameters that achieve the lowest makespan among all possible values for a given memory size.

Our Vdnn is a heuristic based on ideas from VDNN++ [7]. 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 input of every

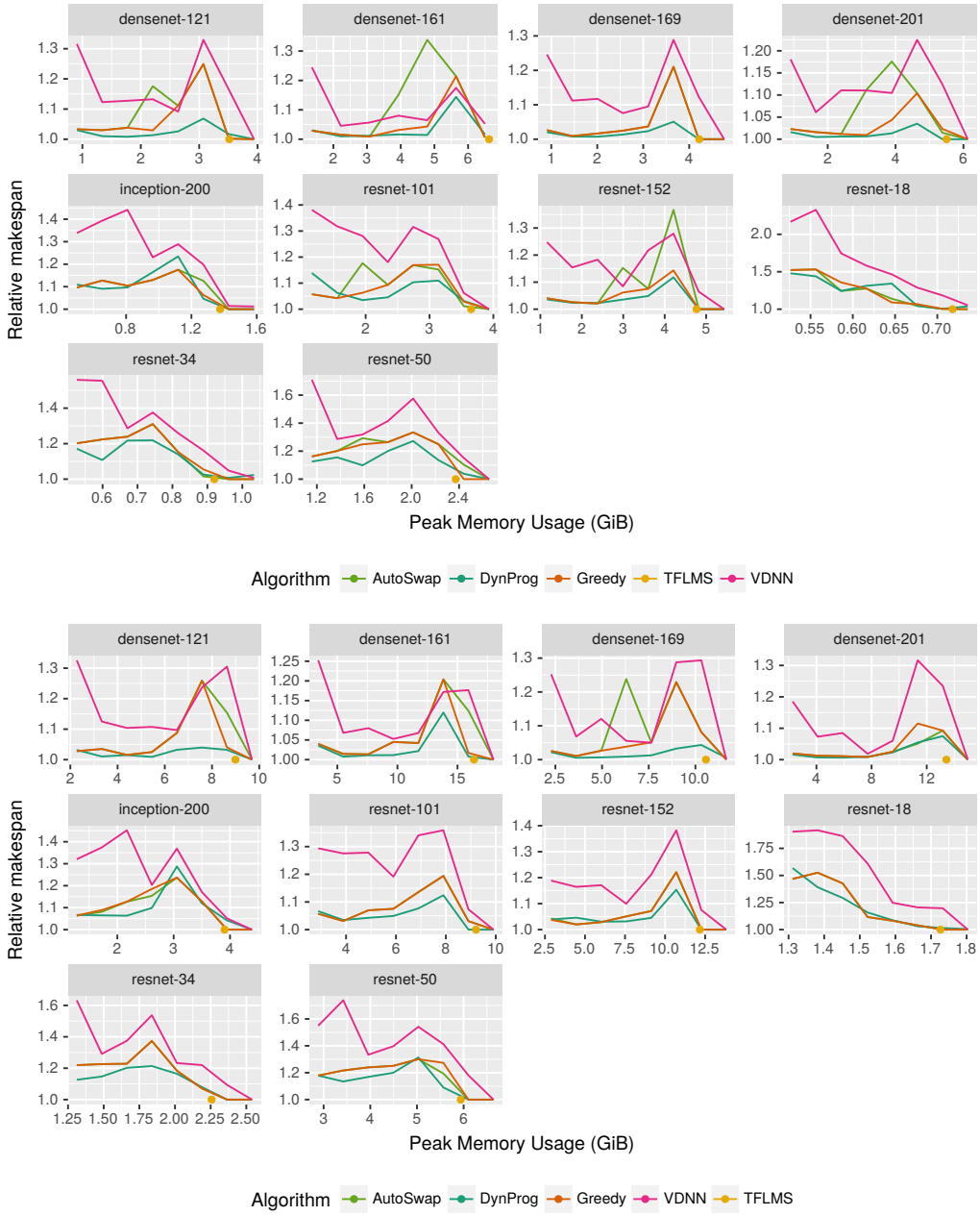


Рис. 3.5: Relative makespan (with respect to the lower bound) obtained by different algorithms for different memory limits. Experimental results are provided for image size 224 and batch size 32 (top), image size 500 and batch size 16 (bottom).

convolution, or of 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 $\frac{u_{F_i}}{\bar{a}_{i-1}}$ for all operations, and then for all possible thresholds, we compute the no-wait, ordered schedule that offloads all the activations whose ratio is above the threshold, and the one that offloads half of them. VDNN outputs the best schedule out of all these choices.

Note that these experiments cover real scenarios obtained from measurements performed on the plaform described above; however the results presented here are obtained by simulation, in which we compute the schedules for the different heuristics, and estimate their expected duration (and thus the corresponding throughput) based on the measurements. In particular, this allows us to consider cases where the memory limit is higher than the available memory on our GPU.

3.4.2 Representative Results

A representative selection of obtained results is depicted in Figure 3.5, where different types of network of different length are considered with a given image and batch sizes. 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 is necessary 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 obtained 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 made available through offloading, and thus it 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 $M_{peak} - M_{\text{GPU}}$). Such cases are significantly 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.

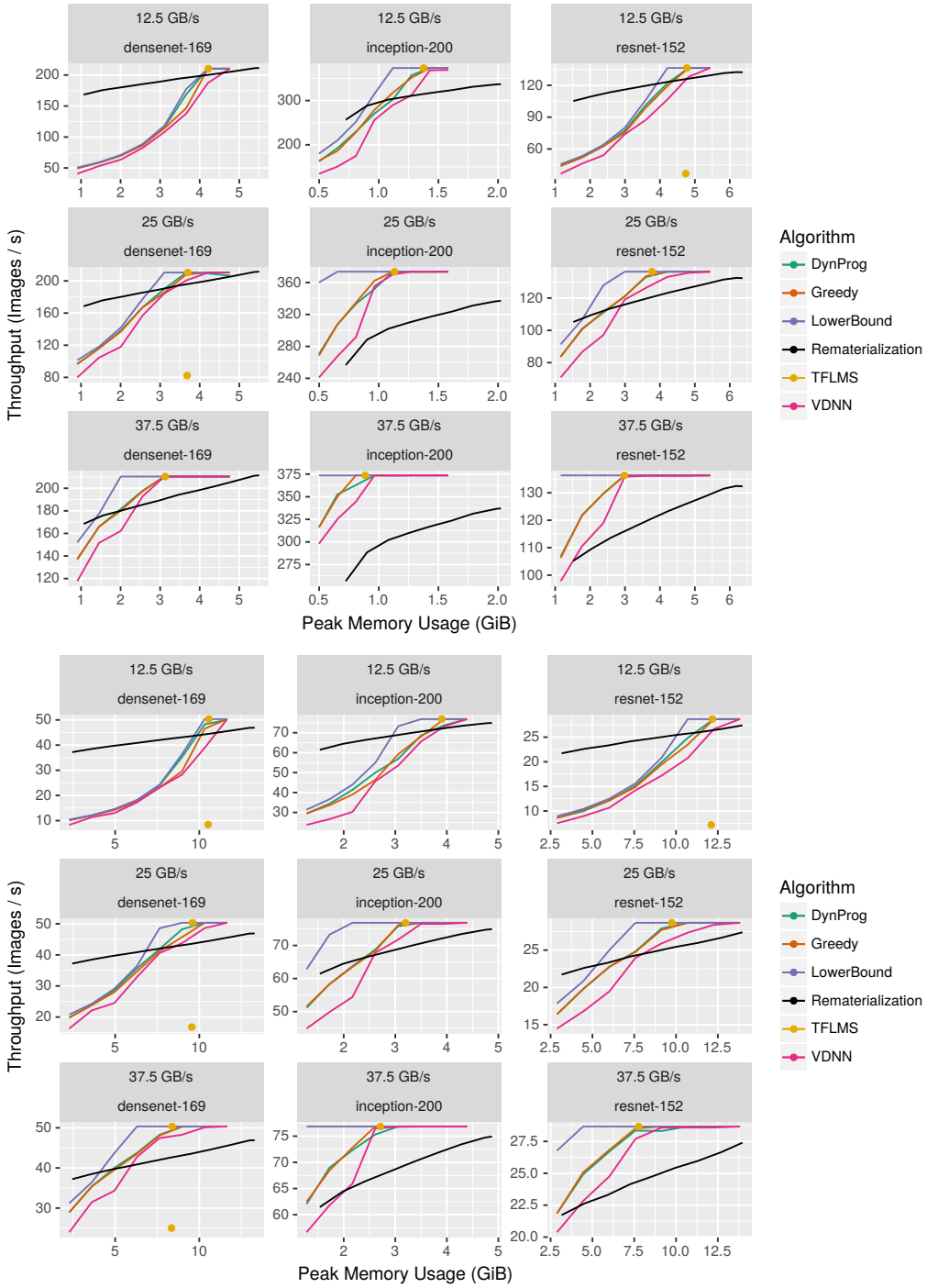


Рис. 3.6: Comparison to Rematerialization for various bandwidth values, with image size 224 and batch size 32 (top), with image size 500 and batch size 16 (bottom).

3.4.3 Comparison to Rematerialization

An alternative to Offloading is Rematerialization that we considered in Chapter 2, in which memory savings are achieved by discarding activations and recomputing them later. In Figure 3.6, we compare the throughput (in terms of processed images per second) obtained by the offloading algorithms and by an optimal rematerialization strategy from Chapter 2. We observe that for the bandwidth measured on our hardware, 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, achieving optimal throughput over a wide range of memory limits.

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 when training DNNs.

3.4.4 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 3.7-3.13. They consistently demonstrate the advantage of using DynProg over other offloading methods.

3.4.5 Complete Results – Througphut

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: $T = \frac{\text{batch size}}{\text{makespan}}$. The results are shown on Figures 3.14-3.20. These plots also include the performance of the rematerialization strategy. Since this strategy was tested in an actual environment, the rematerialization results are unavailable for all cases where memory usage is larger than what is available on the GPU. They confirm that Rematerialization is preferable to Offloading in case of low memory, while in case when the memory limit is close to M_{peak} Offloading may demonstrate a better performance.

Conclusions

In this chapter, we have addressed the problem of applying offloading techniques for memory saving on the GPU during the training phase of Deep Neural Networks. Previous works [88, 7, 76, 64, 114, 108] advocated to offload some of the data onto the main memory, and to prefetch them back when needed. We have proposed 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 have proved that this problem is NP-Complete in the strong sense, and proposed two heuristics based on relaxations of the problem. The Greedy heuristic always offloads the first activations

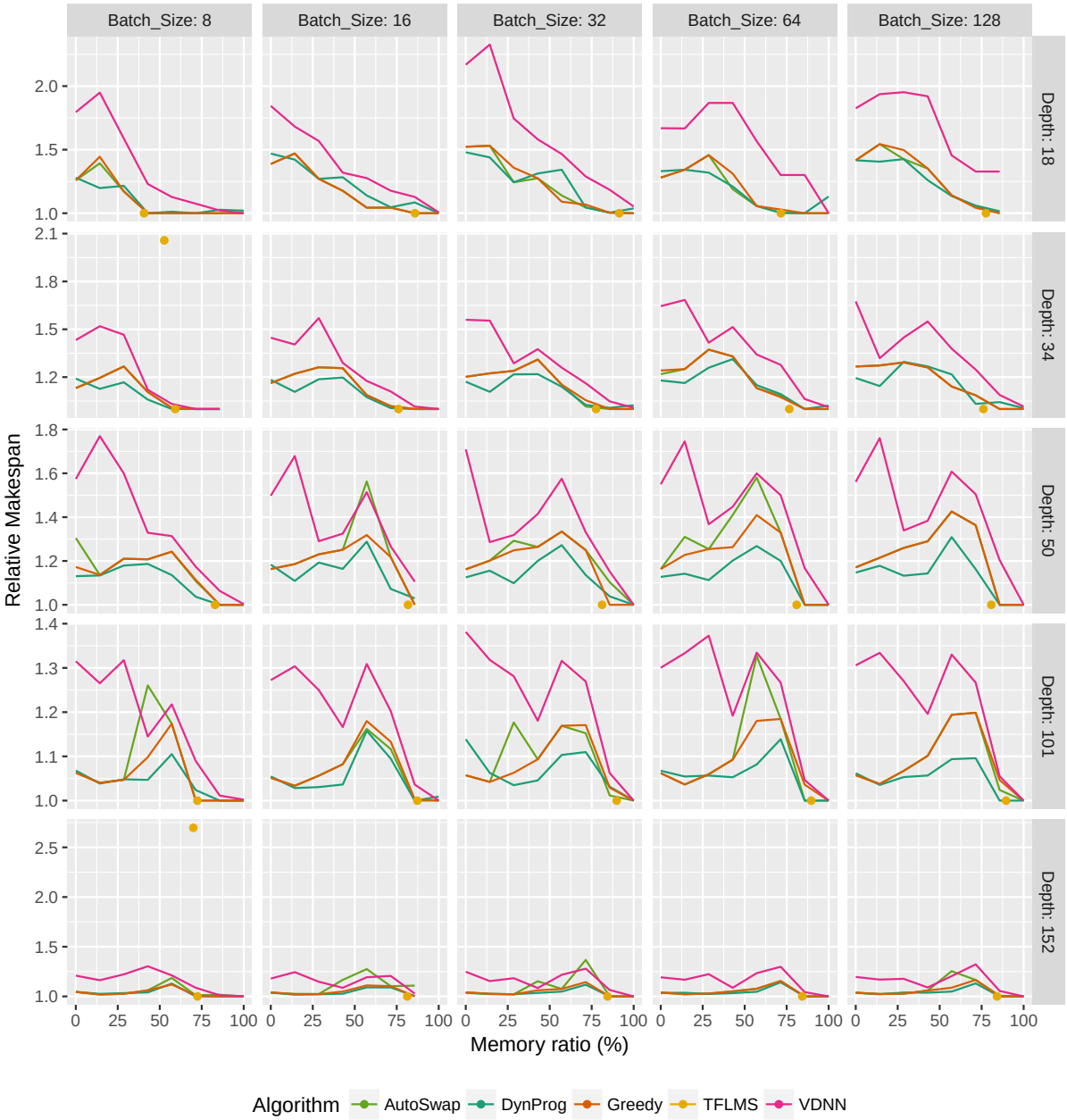


Рис. 3.7: Relative Makespan results for ResNet with small images (224x224).

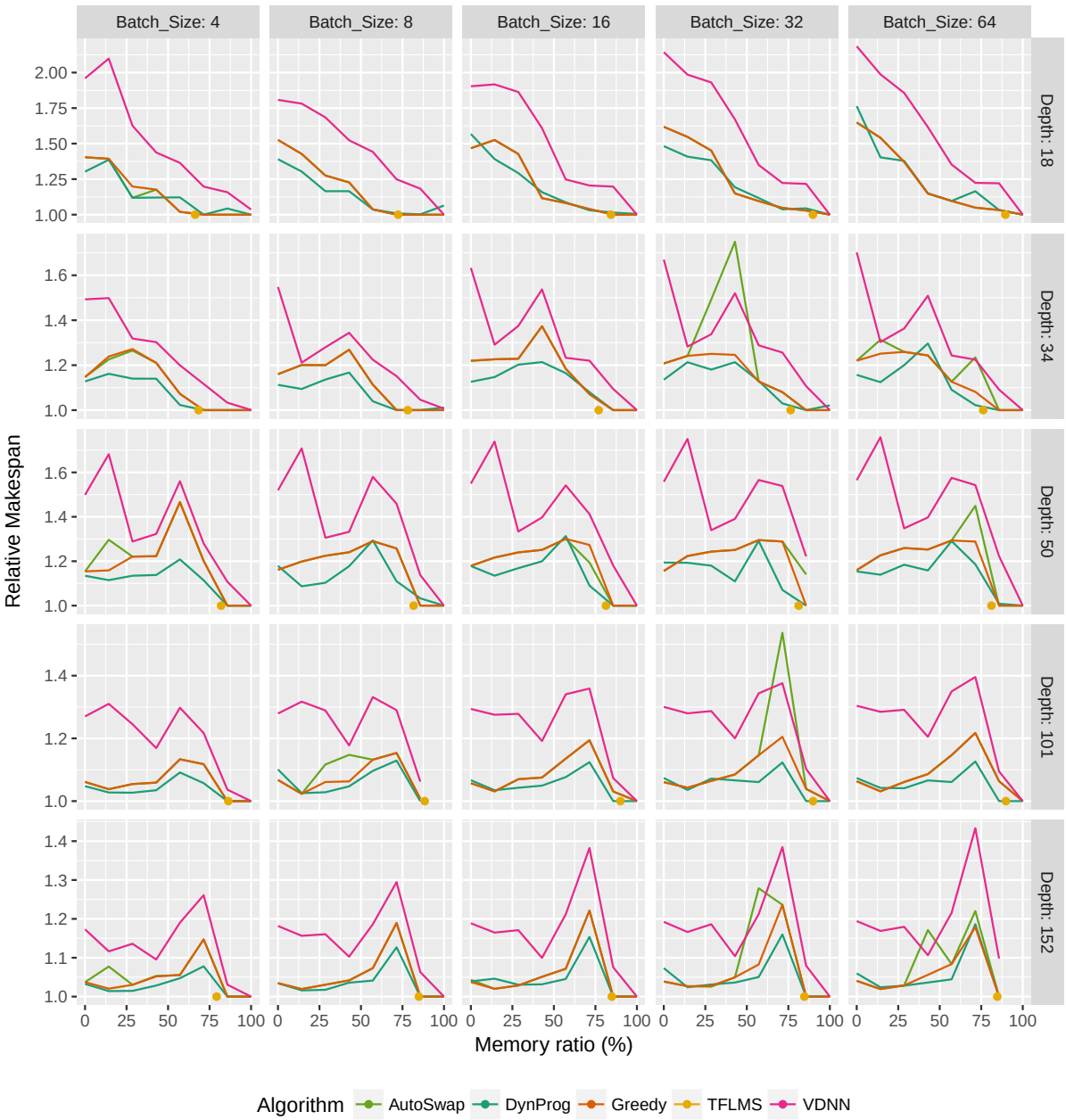


Рис. 3.8: Relative Makespan results for ResNet with medium images (500x500).

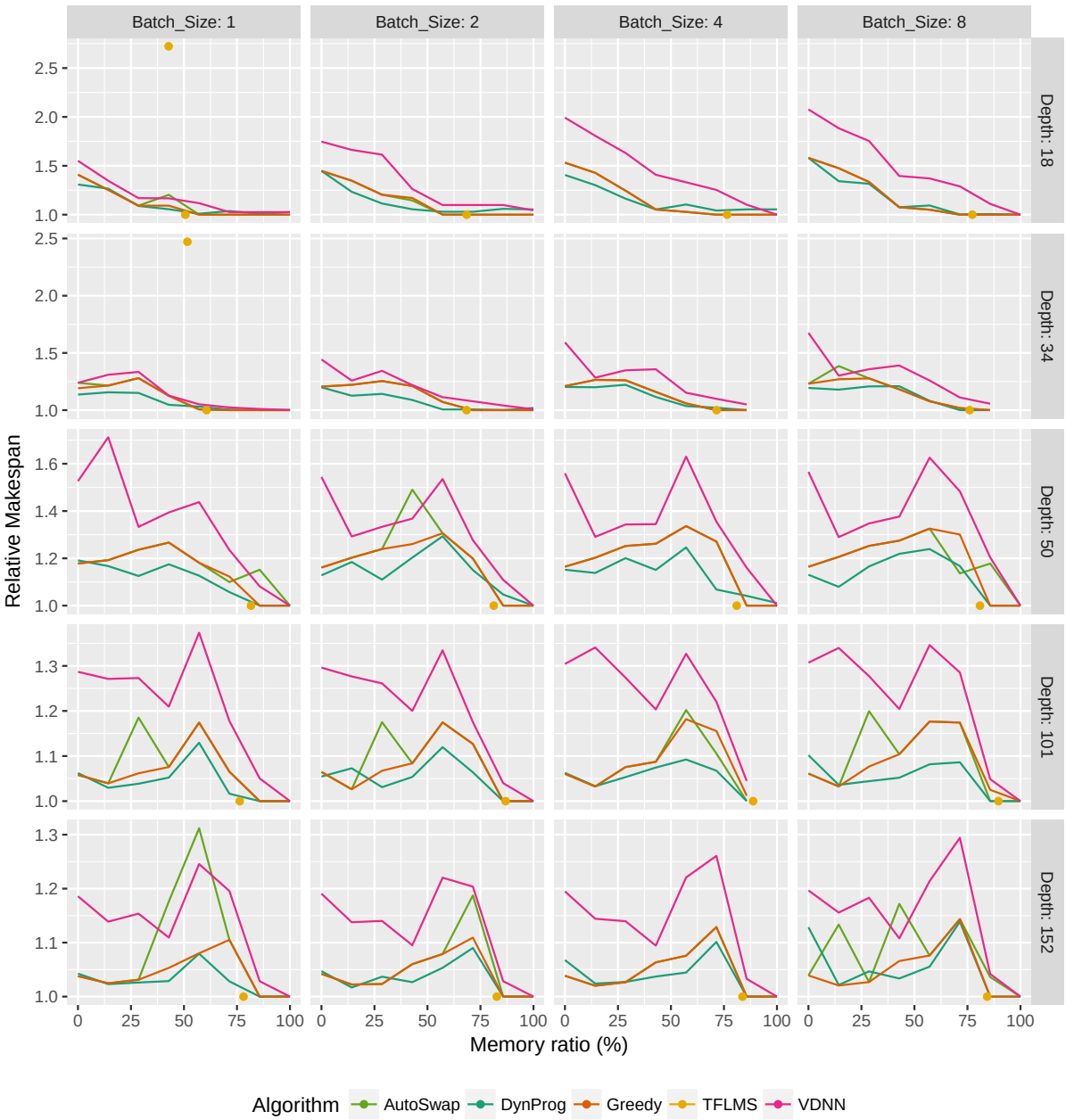


Рис. 3.9: Relative Makespan results for ResNet with large images (1000x1000).

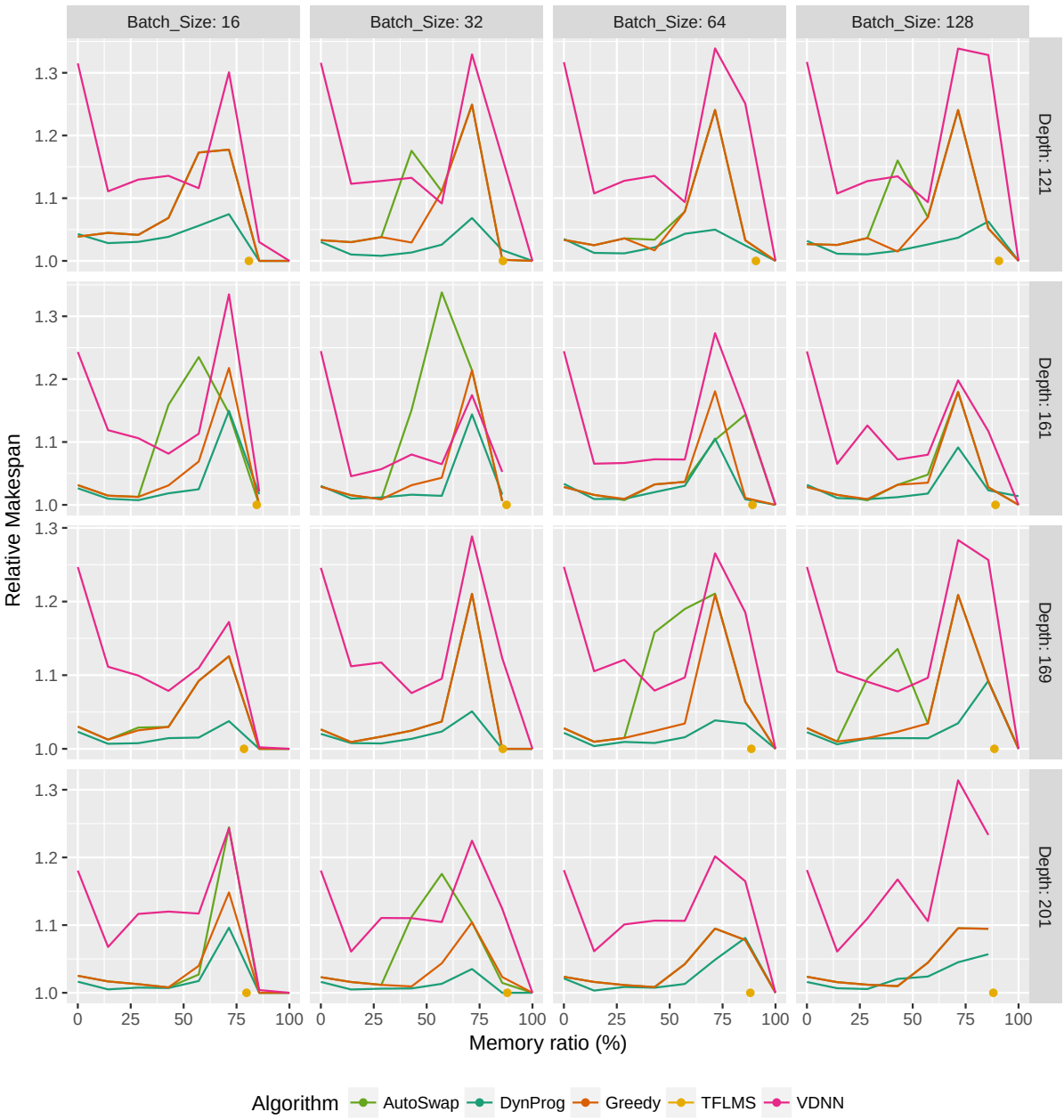


Рис. 3.10: Relative Makespan results for DenseNet with small images (224x224).

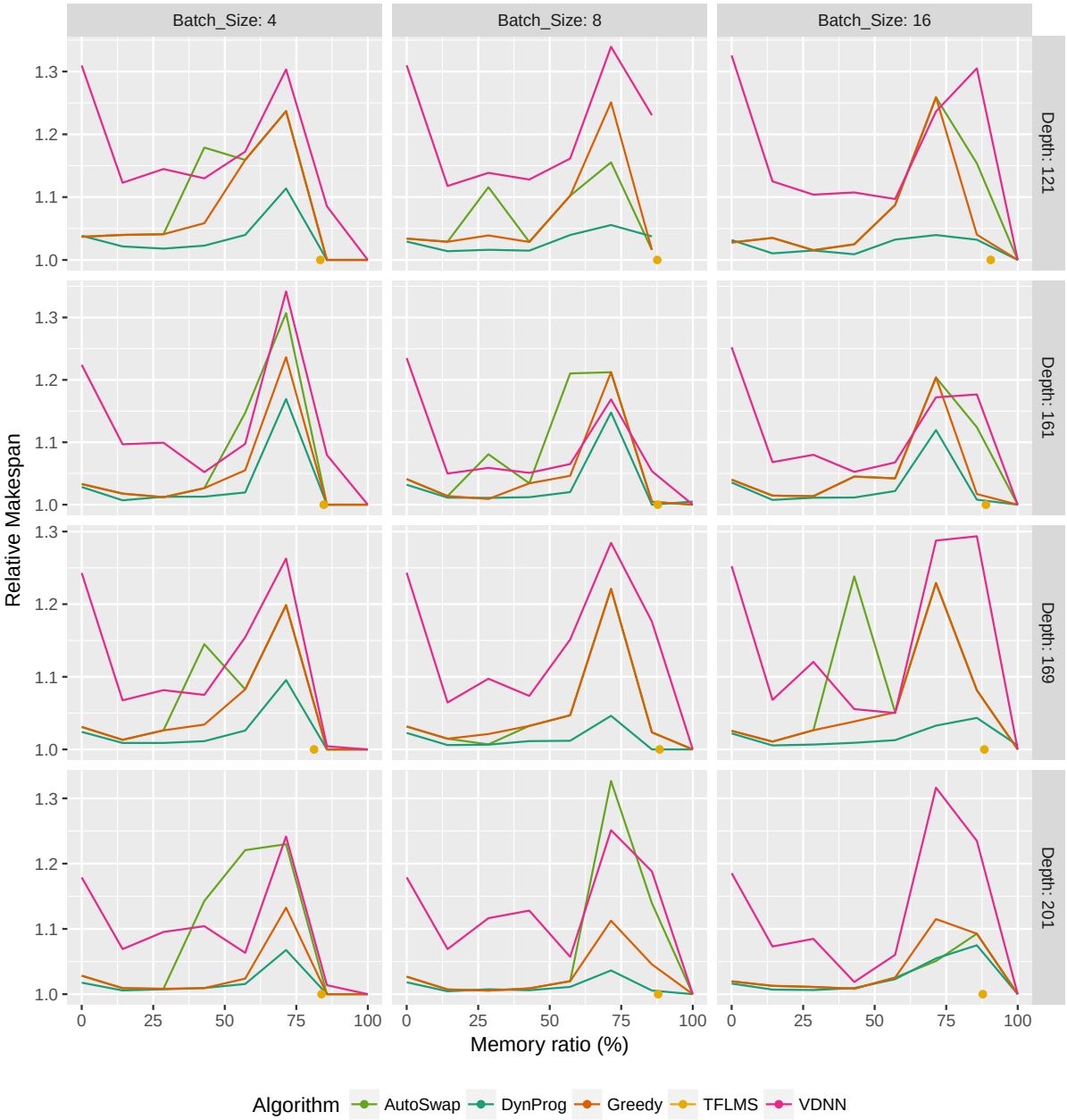


Рис. 3.11: Relative Makespan results for DenseNet with medium images (500x500).

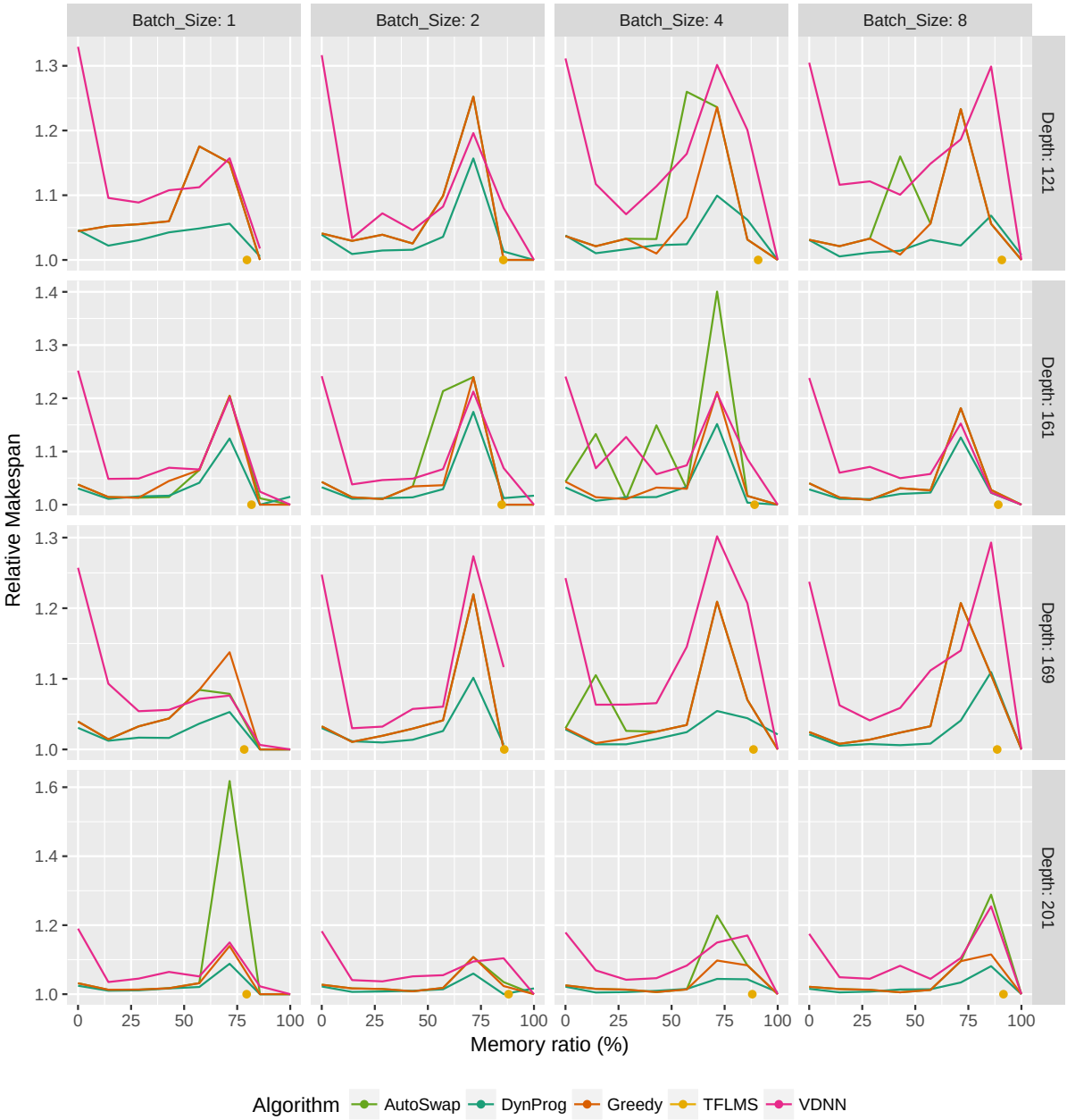


Рис. 3.12: Relative Makespan results for DenseNet with large images (1000x1000).

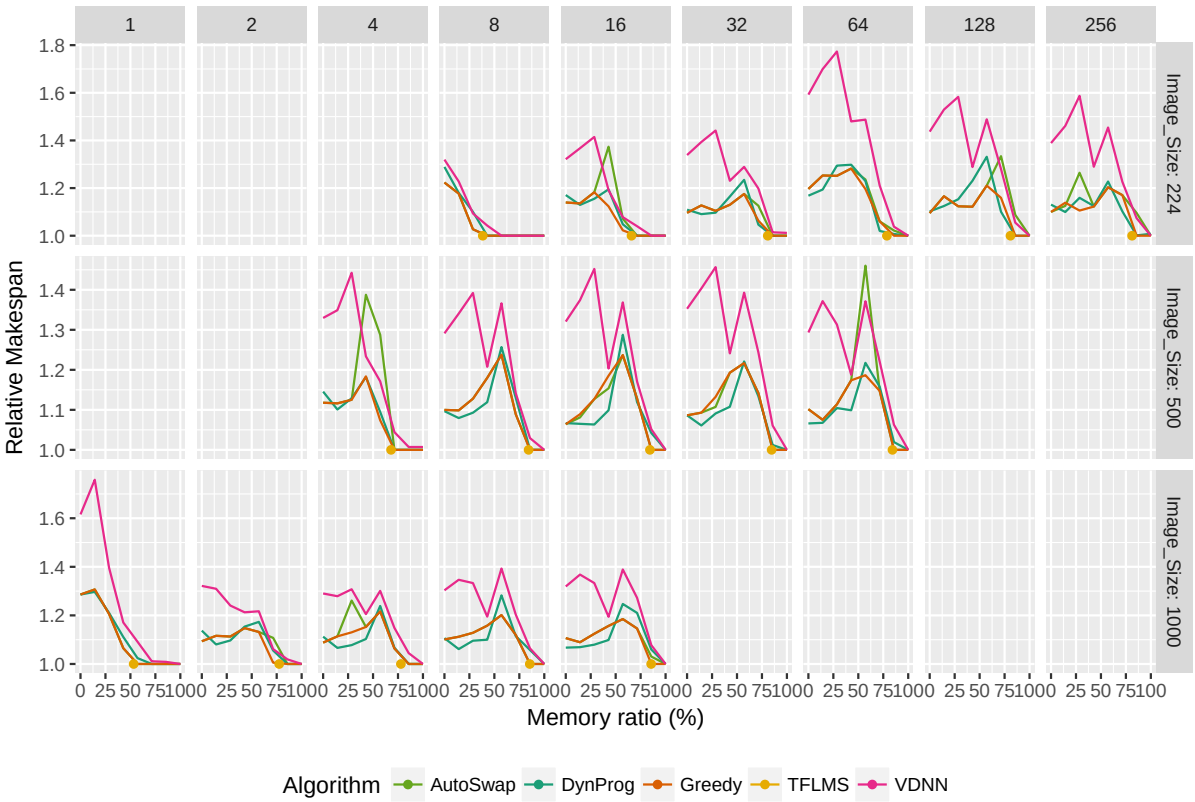


Рис. 3.13: Relative Makespan results for Inception.

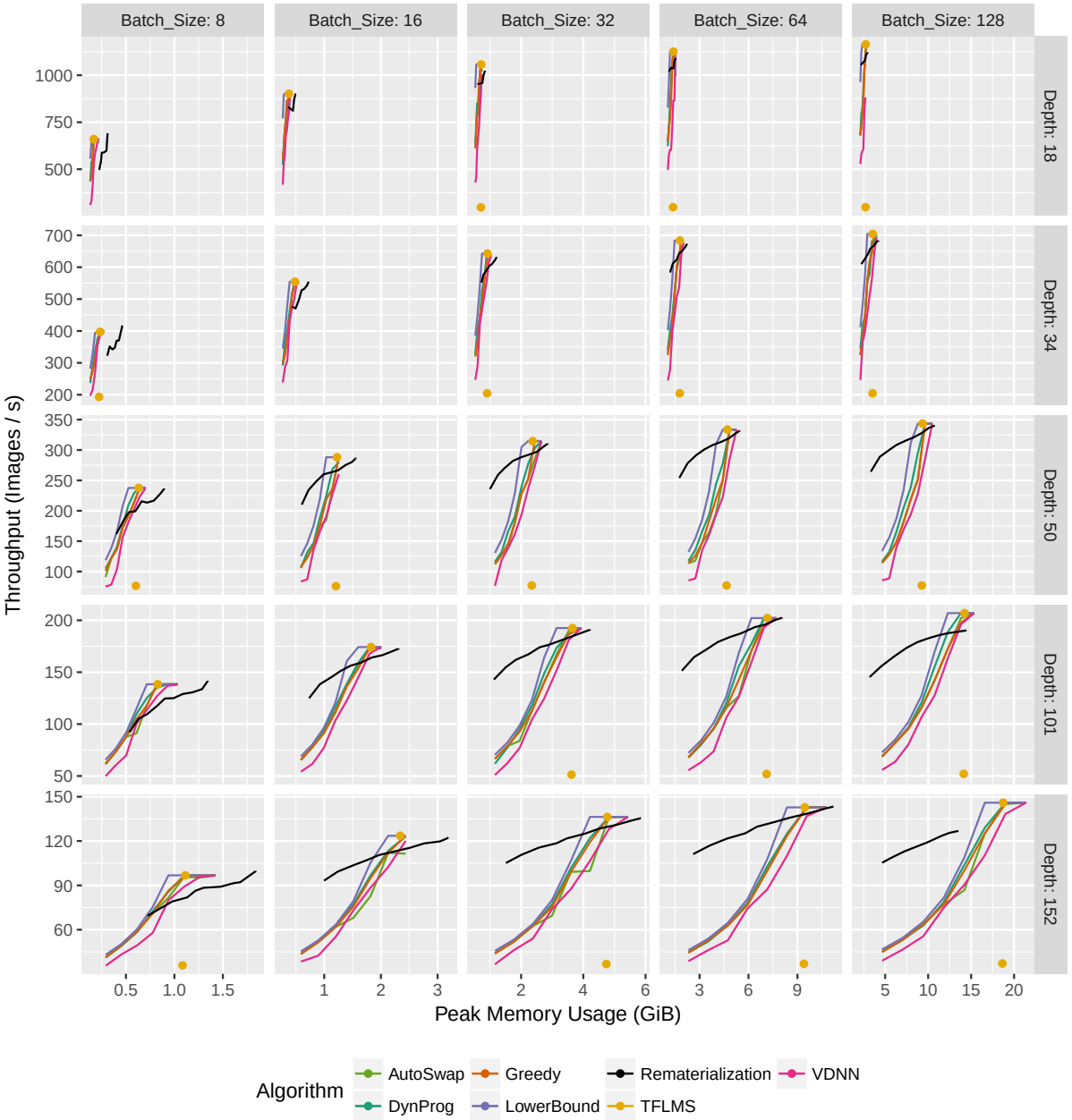


Рис. 3.14: Throughput results for ResNet with small images (224x224).

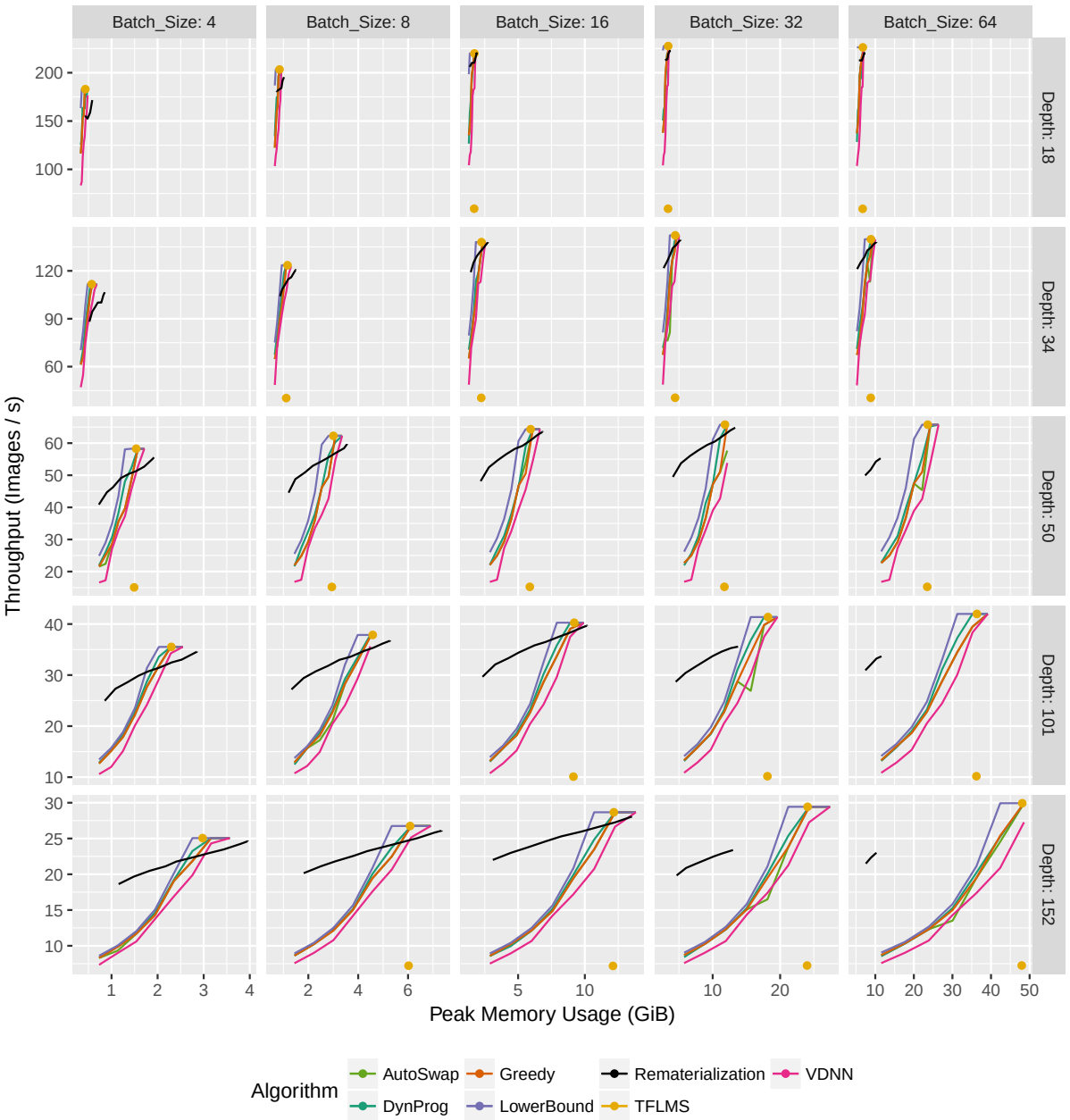


Рис. 3.15: Throughput results for ResNet with medium images (500x500).

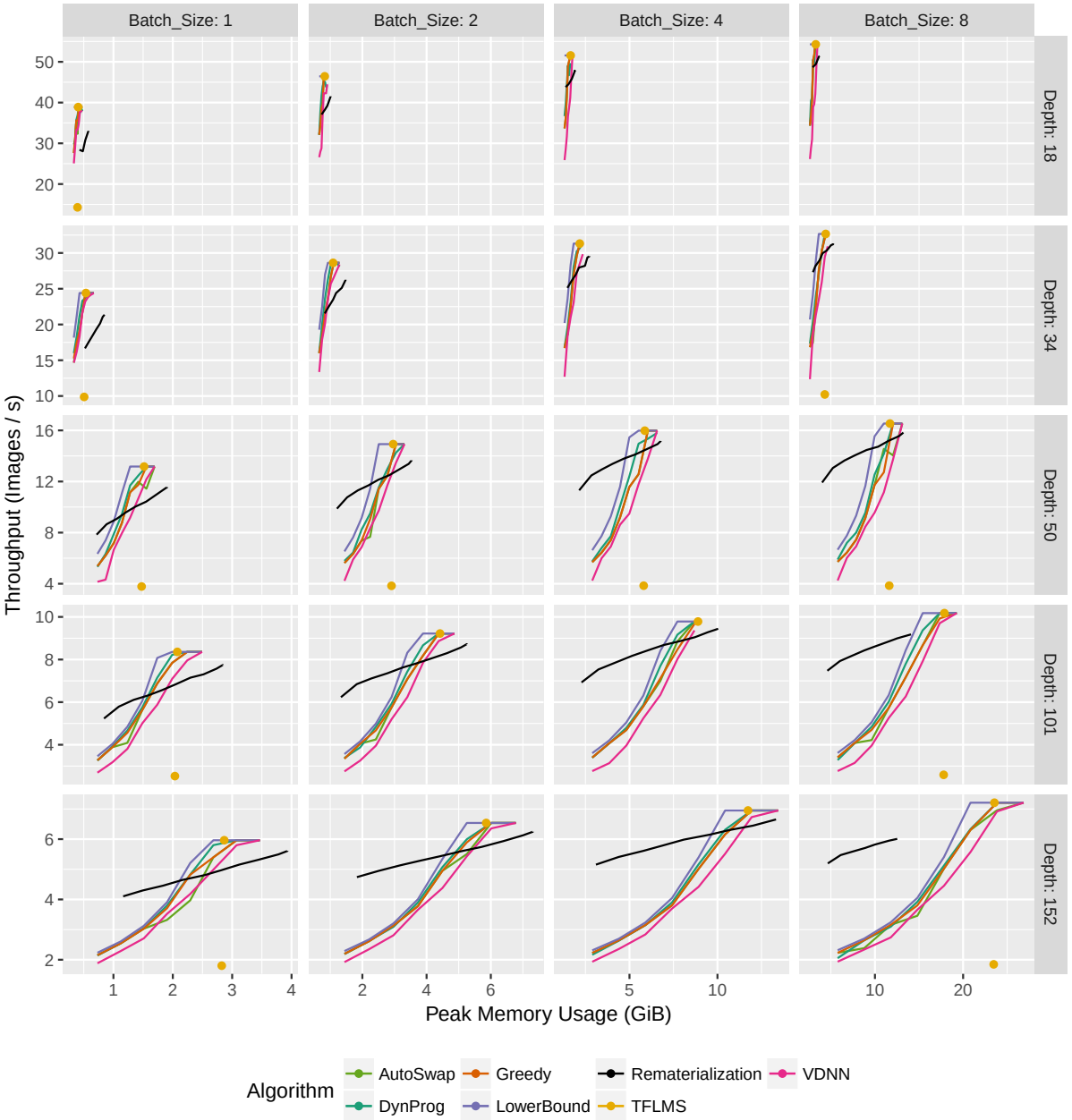


Рис. 3.16: Throughput results for ResNet with large images (1000x1000).

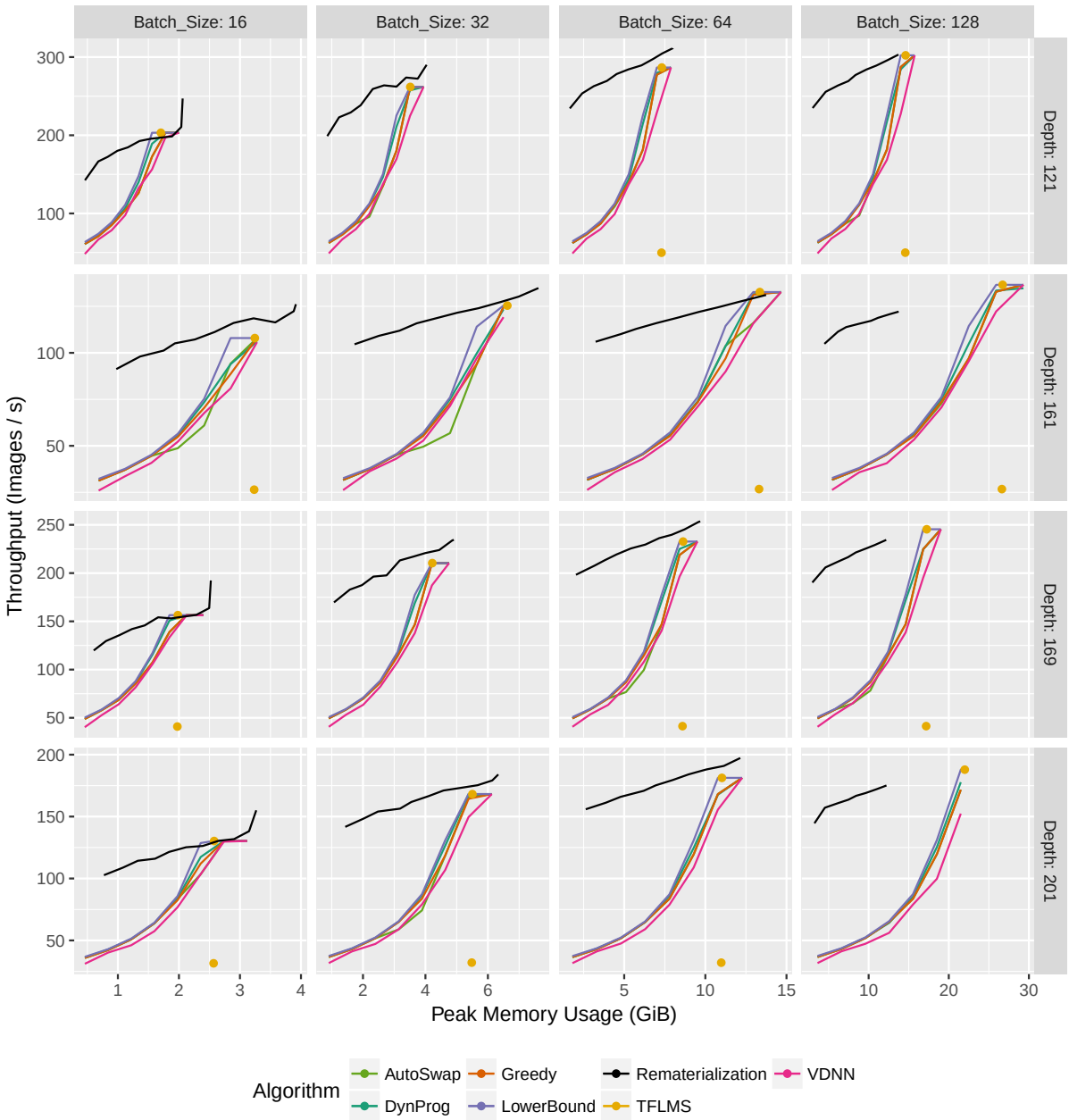


Рис. 3.17: Throughput results for DenseNet with small images (224x224).

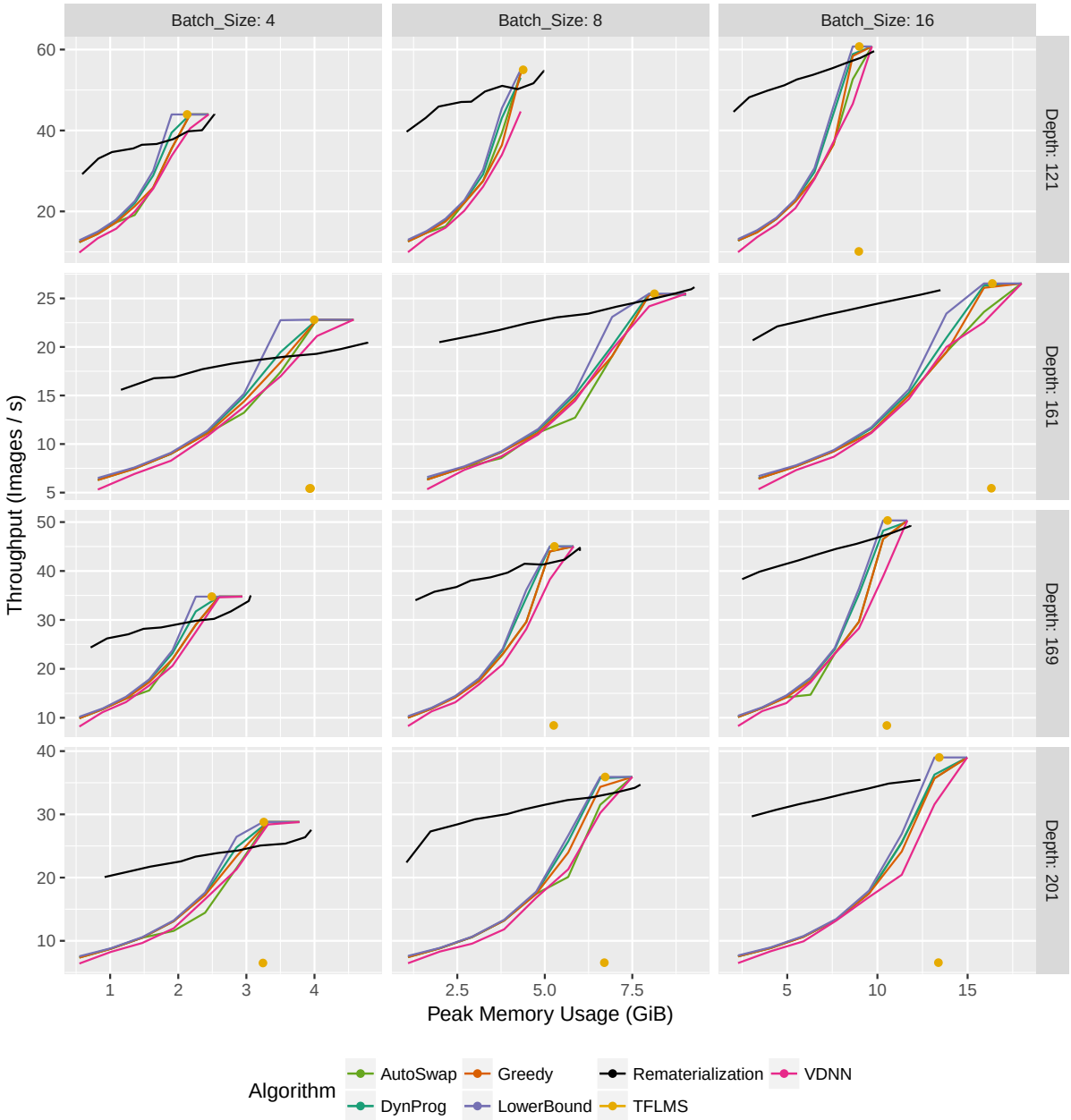


Рис. 3.18: Throughput results for DenseNet with medium images (500x500).

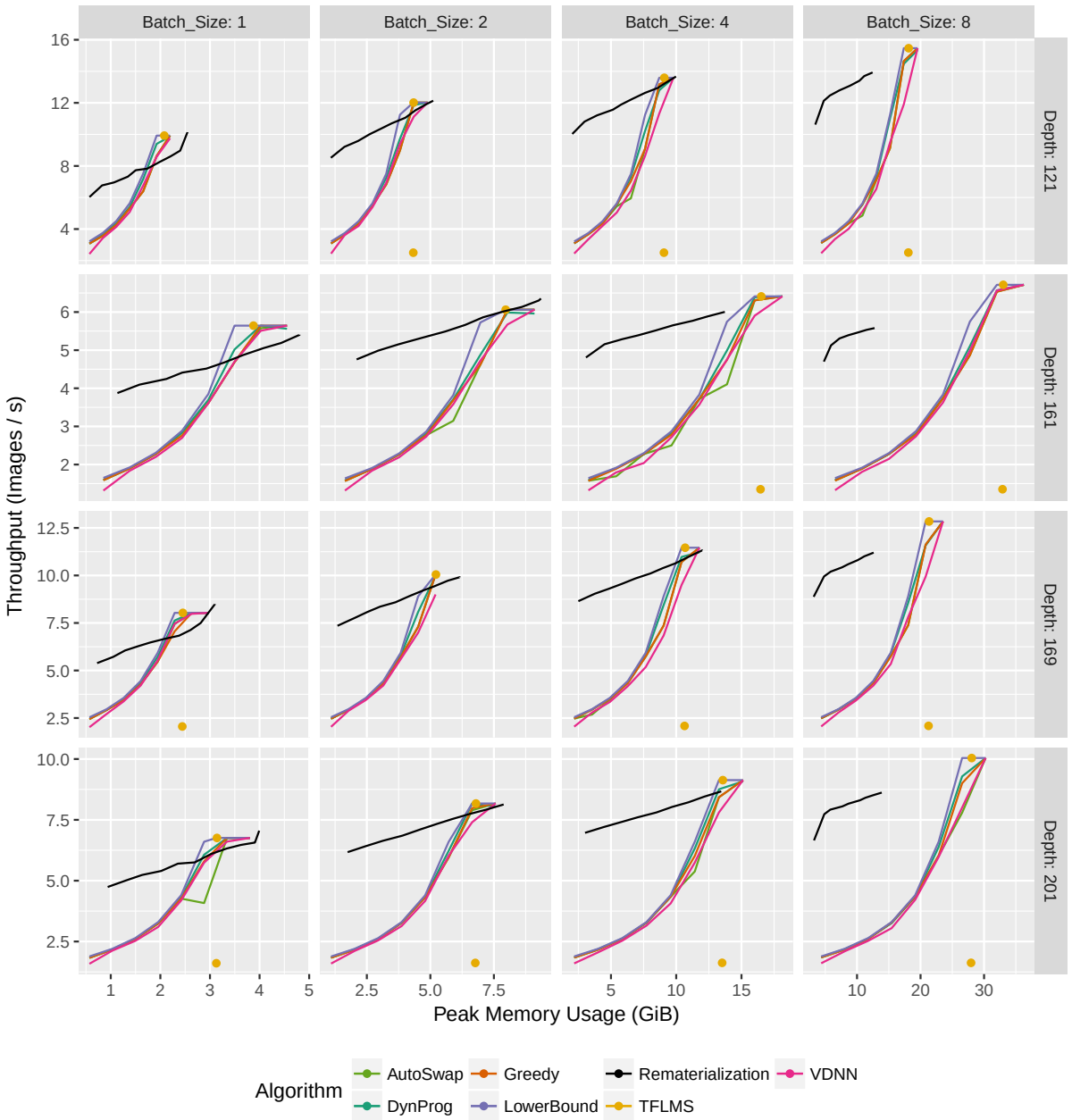


Рис. 3.19: Throughput results for DenseNet with large images (1000x1000).

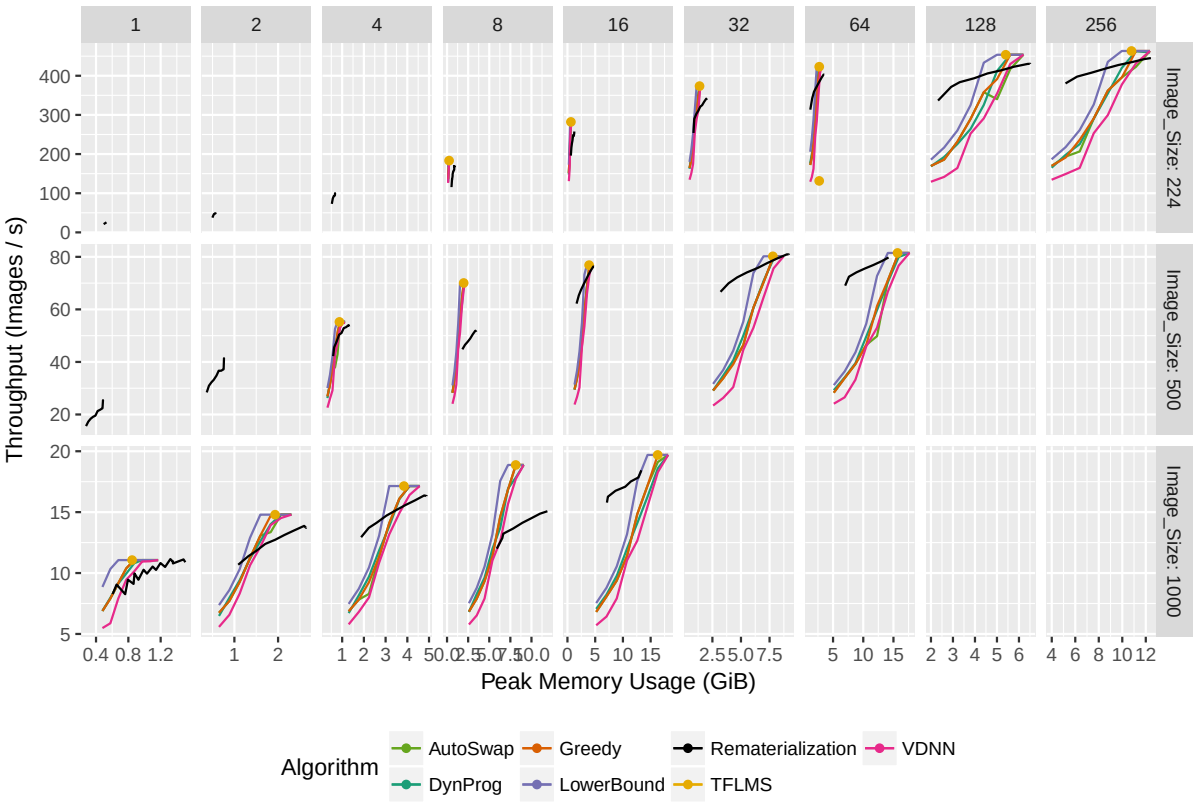


Рис. 3.20: Throughput results for Inception.

in the network. This very simple technique nevertheless achieved good results in our experimental evaluation. The DynProg algorithm is a more sophisticated approach that takes into account the fact that activations cannot be partially transferred, which leads to mostly better solutions. In any case, both algorithms provide significant improvements over the previous approaches.

When compared to Rematerialization, Offloading performs better if memory limit is high enough and close to M_{peak}. Whereas, if memory limit is low, then Offloading significantly drops in performance, becoming worse than Rematerialization. The limitations of both approaches can be mitigated if they are combined together. Therefore, in the next chapter we consider how to find an optimal combination of these methods.

Глава 4

Combination of Offloading and Rematerialization

As shown in the previous chapter offloading techniques can be very efficient when the memory peak of the execution just slightly exceeds the memory available on the device or if there is a fast communication link to the larger memory. However, as it can be observed in the simulation plots, it is less competitive when memory consumption should be significantly reduced and it gives in to rematerialization approach. As there are situations where each method performs significantly better than the other, the combination of both methods might profit from both sides and, eventually, outperform each individual method.

Hence, in this chapter we discuss $\textsc{OffRemat}_{int}(L, M_{\text{GPU}}, \beta)$. We start with the description of the dynamic program called `pofo` in Section 4.1, which combines the elements of the solutions for Rematerialization (Algorithm 3) and Offloading (Algorithm 5). Indeed, for each individual problem there exists a dynamic program providing an optimal solution under some assumptions. We also prove in Section 4.2 that the final dynamic program optimally solves $\textsc{OffRemat}_{comm}(L, M_{\text{GPU}}, \beta)$, which is a relaxed version of $\textsc{OffRemat}_{int}(L, M_{\text{GPU}}, \beta)$ with some additional assumptions. Furthermore, we supplement our contribution with a few heuristics in Section 4.3, which are cheaper than `pofo`. In Section 4.4 the results of our experiments are presented and we compare all techniques and evaluate the potential improvement of `pofo` over the pure Rematerialization or pure Offloading.

4.1 Combination of Offloading and Rematerialization

4.1.1 Complexity

Similarly to $\textsc{Off}_{int}(L, M_{\text{GPU}}, \beta)$, the requirement of offloading activations entirely and do discards on the GPU only when the transfer is completed makes the problem NP-complete in the strong sense.

Theorem 11. $\textsc{OffRemat}_{int}(L, M_{GPU}, \beta)$ *is strongly NP-complete.*

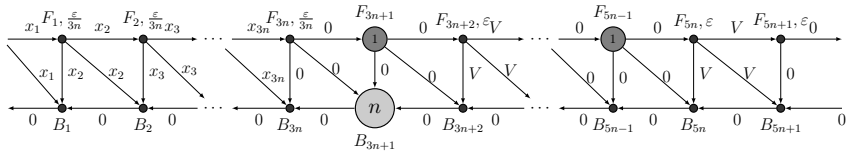


Рис. 4.1: The instance of $\textsc{OffRemat}_{int}(L, M_{\text{GPU}}, \beta)$ used in the reduction from 3-Partition. Activation sizes are indicated on the edges and non-zero execution times of operations are written inside the corresponding nodes or on the right from the operation names.

Доказательство. Analogously to the proof of Theorem 7, we show it by reducing this problem to 3-Partition problem and for that we take the example from the proof of Theorem 7 and modify it. Instead of having $u_{F_i} = 0$ for $1 \leq i \leq 3n$ and $u_{F_j} = 0$ for $j = 3n + 2k, 1 \leq k \leq n$, we set $u_{F_i} = \varepsilon/3n$ and $u_{F_j} = \varepsilon$ respectively (see Figure 4.1). The choice of ε can be arbitrary as long as the next inequality is fulfilled

$$0 < \varepsilon < \min_{\substack{1 \leq i < j < k \leq 3n \\ x_i + x_j + x_k > V}} \frac{x_i + x_j + x_k}{V} - 1.$$

Then for the chain from Figure 4.1, the decision problem should be formulated in the following way: is there a schedule that can execute the chain in time $T = 2n + 3n\frac{\varepsilon}{3n} + n\varepsilon$ with GPU memory nV and bandwidth V?

This problem belongs to NP-class: given the schedule it is easy to estimate if the chain is executed within time limit T. The value ε can be also found in $O(n^3)$ number of steps, applying straightforward linear search over all possible triplets.

On the one hand, the same reasoning as in Theorem 7 can be used to show that if there exists the solution to a 3-Partition problem, then we can build a schedule that inserts no delays in the execution of the chain, thus yielding the final makespan $2n + \sum_{i=1}^{3n} u_{F_i} + \sum_{k=1}^{n} u_{F_{3n+2k}} = 2n + 3n\frac{\varepsilon}{3n} + n\varepsilon$.

On the other hand, let us assume that there exists the schedule for the given chain that executes it without any idle time, thus without recomputing any forward step. Now let us show that it leads to the solution of 3-Partition problem. Indeed, ε is chosen carefully and depends on all x_i: it is small enough, so that for any triplet of $1 \leq i < j < k \leq 3n$ that satisfy $x_i + x_j + x_k > V$, it also holds $x_i + x_j + x_k > V(1 + \varepsilon)$. The latter condition ensures that during the first $3n + 1$ forward steps it is impossible to offload a triplet of x_i, x_j, x_k, unless their sum is less or equal than V. The same holds also for each pair of forwards F_{3n+2k} and $F_{3n+2k+1}$ for $1 \leq k < n$. As we need to offload at least nV amount of data to avoid idle times, there should exist at least n triplets of x_i, x_j and x_k such as $x_i + x_j + x_k = V$, which completes the proof.

□

4.1.2 Additional Assumptions

Because of the NP-completeness in the strong sense of $\textsc{OffRemat}_{int}(L, M_{\text{GPU}}, \beta)$, we introduce some additional assumptions in order to make the problem tractable.

Assumptions 4. *For the following problem we consider Assumptions 1(**O.1**-**O.6**) and the following assumptions.*

Simplifying Assumptions:

OR.1 *Memory persistence holds: if for some layer ℓ activation $\bar{a}_\ell$ is computed, then no operations related to layers ℓ' with $\ell' < \ell$ take place until B_ℓ. As it was shown in Chapter 2, that in general the optimal solutions are not bound to follow this assumption. On the one hand, even without this assumption the problem of pure Rematerialization can be solved using dynamic programming, but at the cost of a higher computational cost. On the other hand, our experiments showed that this restriction did not hinder the performance for all practical vision networks. Therefore, to simplify the formulas and the final complexity of the dynamic program, we impose this constraint.*

OR.2 *Offloading decisions before loss: we offload only the activations computed before the loss. This is a reasonable assumption since Offloading introduces delays and therefore should be performed as early as possible.*

OR.3 *Synchronization at the loss: all offloading operations must complete before the computation of the loss and prefetching operations (each prefetching operation being performed only once) start only after the computation of the loss. As it was discussed in Chapter 3 Section 3.3.2.2, in general, it can be beneficial to allow offloading to continue in the beginning of the backward phase and the same for prefetching during the forward phase. However, it increases the complexity of the resulting dynamic program, since state variables Δ_{F_ℓ} and Δ_{B_ℓ} should take negative values as well, increasing their range. Besides, the approach used in the previous chapter to compute ϵ_G (like in Eq. (3.16) or (3.17)) cannot be applied in a straightforward way to this model. Because of recomputations it is not anymore possible to reconstruct memory profile of other operations at the last level of the dynamic program recursion.*

OR.4 *Prefetching only once: if the value a_ℓ or $\bar{a}_\ell$ is prefetched, then it should be discarded only after the backward B_ℓ. Similarly to **OR.2**, it helps to limit the number of delays caused by data transfers.*

OR.5 *Asynchronous and continuous offloading: Activations should be offloaded/prefetched entirely, but memory allocation and release can be performed in several steps during the communication. This is the same as the clause **O.7"** of Assumptions 3. It proved to be efficient in the case of pure Offloading and helped to provide a tractable context without degrading the solution quality.*

Problem 11 (Combination of Offloading and Rematerialization $\textsc{OffRemat}_{comm}(L, M_{\text{GPU}}, \beta)$)**.** Consider a training phase with L operations, corresponding to a chain, depicted on Figure 2.16 with processing times and memory usage described in Table 2.2. Using all operations from Table 2.2 together with O_ℓ and

P_ℓ and under Assumptions 4, 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 ?

We propose a dynamic program to optimally solve $\textsc{OffRemat}_{comm}(L, M_{\text{GPU}}, \beta)$. We further denote this algorithm as `pofo`, for "Persistent with Offloading during Forward Only".

4.1.3 Rationale of the Different Operations

The `pofo` algorithm is based on a sequence of choices, that consist in deciding, for each $1 \leq \ell \leq L$:

(i) which type of operation F_ℓ we are going to use and whether we are going to compute a_ℓ or $\bar{a}_\ell$,

(ii) whether we are going to keep the input value, which is either $a_{\ell-1}$ or $\bar{a}_{\ell-1}$, in the memory of the GPU, offload it in the memory of the CPU or completely delete it from the memory,

(iii) how to compute B_ℓ.

Concerning (i), the size of $\bar{a}_\ell$ is in general larger than a_ℓ, especially in the case of grouped layers. Both can be used to compute $F_{\ell+1}$ and $B_{\ell+1}$, but if a_ℓ only is kept in memory (either CPU or GPU), then F_ℓ will have to be recomputed during the backward phase before B_ℓ. Concerning (ii), if we delete $a_{\ell-1}$ or $\bar{a}_{\ell-1}$ from memory, then it will be necessary to recompute $F_{\ell-1}$ before processing B_ℓ, but it will save memory for the whole subsequent sequence $F_{\ell+1} \dots F_L B_L \dots B_{\ell+1}$. If $a_{\ell-1}$ or $\bar{a}_{\ell-1}$ is offloaded on CPU memory (which may take some time), the memory will be available for the subsequent sequence, though $a_{\ell-1}$ or $\bar{a}_{\ell-1}$ will have to be prefetched before computing B_ℓ. The interest of Offloading obviously depends on the bandwidth β of the communication link. Concerning (iii), if the input activation ($a_{\ell-1}$ or $\bar{a}_{\ell-1}$) has been deleted from memory, we will have to recompute it, starting from the last kept activation (either a_k or $\bar{a}_k$ for some $k < \ell - 1$). Several rematerialization sequences are possible to do this operation that have potentially different durations and that offer different prefetching possibilities, depending on their memory profile, and the choice of the optimal sequence will depend on the memory state before computing B_ℓ, as detailed in Section 4.2.

4.1.4 Intuition of the Overall Scheme and State Variables

The dynamic programming solution is built in several steps. We consider separately the forward propagation within which the offloading of activations can be done. The forward propagation is followed then with the loss calculation synchronized with the end of offloading operations. After loss computation, the backward propagation interleaved with the prefetching starts.

The general principle of `pofo` is the following. During the forward phase (before the computation of the loss), we consider the layers in an increasing order (from 1 to L). At each step, we have to decide which operation to implement among F_ℓ^{all}, F_ℓ^{ck}, $F_\ell^{\varnothing}$ and in

the case where the input activation is kept, we have to decide if we add O_ℓ. Inspired by the ideas of Automatic Differentiation [39], our dynamic programming relies on the following remark: any solution can be decomposed into parts, where each part computes layers s to t ($t \geq s$), and only the input of layer s is stored in memory. Starting from a layer s, `pofo` needs to decide whether the input will be saved with F_s^{ck} or F_s^{all}. It also needs to decide the index t of the end of the corresponding part in the solution, so that $t+1$ starts the next part of the sequence, and its input will be the first activation kept in memory after s. A recursive call to the sub-chain starting from $t+1$ allows to obtain the corresponding running time. Between the two saved inputs of layers s and $t+1$, a sequence of $F_k^{\varnothing}$ takes place. As forwards F_s^{ck} and F_s^{all} correspond to two different behaviors, they generate two different cases of the dynamic program that are considered in Section 4.2.1. In addition, an offloading decision needs to be made for the input of layer s, either store it in the memory of the GPU or offload it to the CPU; this decision impacts the memory available for the rest of the chain and idle times due to communications.

In order to use dynamic programming, we need to define a set of variables that describes the state of the system at any instant. The state variables presented in the previous chapter are suitable for this problem as well, though in this case it is also important to know which type of the input the algorithm is working on:

- A_s denotes the total GPU memory occupied by the saved values among $a_0, \ldots, a_{s-2}$ and $\bar{a}_1, \ldots, \bar{a}_{s-2}$ that are not transferred to the CPU.
- Δ_{F_s} denotes the amount of data from $a_0, \ldots, a_{s-2}$ and $\bar{a}_1, \ldots, \bar{a}_{s-2}$ that the schedule still needs to offload after F_{s-1}.
- Δ_{B_s} denotes the amount of data from $a_0, a_1, \ldots, a_{s-2}$ and $\bar{a}_1, \ldots, \bar{a}_{s-2}$ that the schedule should prefetch before starting B_{s-1}.
- x is a boolean specifying whether the input of layer s is a_{s-1} (if $x = 0$) or $\bar{a}_{s-1}$ (if $x = 1$).

Let us notice that Δ_{F_s} and Δ_{B_s} stay positive in this dynamic programming, due to Assumptions 4 (in particular **OR.3**).

We denote as $\mathcal{I}_s^x = (1-x)a_{s-1} + x\bar{a}_{s-1}$ the size of the input activation for layer s. With these state variables, we can compute $M_{F_s}^x$, the memory on the GPU after executing F_{s-1}, and $M_{B_s}^x$, the memory on the GPU before executing B_{s-1} (excluding δ_{s-1}) are given by

$$M_{F_s}^x = A_s + \Delta_{F_s} + \mathcal{I}_s^x, \tag{4.1}$$

$$M_{B_s}^x = A_s + \Delta_{B_s} + \mathcal{I}_s^x. \tag{4.2}$$

These memory values represent the maximal memory occupation between layers s and t for the forward and backward phases respectively, and are used for computing idle times when overlapping communications with computations, according to Lemma 8, proven in Chapter 3 Section 3.3.2. On the other hand, to estimate the feasibility of the scheduled operations, we also need to know the memory after everything has been offloaded, which is either $A_s + \mathcal{I}_s^x$ (when $\mathcal{I}_s^x$ stays on the GPU) or A_s (when $\mathcal{I}_s^x$ is sent to the CPU), which we use to obtain the maximal memory available on the GPU.

The updates of the state variables follow simple rules described below

- if no new data is offloaded (prefetched), then Δ_{F_s} (Δ_{B_s}) is constantly decreasing from index s to index t at speed β, until reaching zero;
- if new data has to be offloaded (prefetched), then the data size is added into Δ_{F_s} (Δ_{B_s});
- A_s is updated if new data is saved on the GPU (without being later offloaded to the CPU).

Our goal is to find the sequence of operations that minimizes the overall execution time. However, both Rematerialization and Offloading can induce extra time with respect to the execution of the sequence $F_1 \dots F_L B_L \dots B_1$ which can be computed given infinite memory on the GPU:

- **During the forward phase**, Offloading helps to keep GPU memory low, but, if the transfers to the CPU are not fast enough due to limited bandwidth, some idle time might occur, waiting for enough memory to be freed by offloading. This is analyzed in details in Section 4.2.1.
- **At the interface between the forward and the backward phase**, we need to enforce that offloading and prefetching are well synchronized at the loss computation respecting Assumption **OR.3**. This can in turn introduce some idle time on the GPU, which will be analyzed in Section 4.2.2.
- **During the backward phase** (after the computation of the loss), there might be two sources of delays: recomputations of discarded activations and idle times induced by prefetching. Indeed, some prefetching operations may not be completed by the time the activation is needed, if they are delayed because of memory constraints. In fact, these two sources of extra delays are not independent, since a longer rematerialization sequence with lower memory needs might allow more overlapping of prefetching with computations and thus avoid idle time later. We propose an auxiliary dynamic program that includes the computation of intermediate idle times in each recursive call to find the best schedule under prefetching. The idle times for prefetching are found using Lemma 8. This is analyzed in Section 4.2.3.

The above mentioned ingredients can be used to build the dynamic programming algorithm `pofo`, solving optimally the problem in pseudo-polynomial time. Theorem 12 states this result, while the complete description of the dynamic program and the proof of the theorem can be found in Section 4.2. The algorithm is pseudo-polynomial in the memory limit and it relies on discretization of the memory values. In the experiments of Section 4.4, we systematically use 50 discretization steps (scaling so that $N = 50$). Compared to pure Offloading or pure Rematerialization, the quality of discretization is degraded ($N = 500$ against $N = 50$) because of more computationally expensive dynamic programming, however our experiments showed that $N = 50$ has no discretization effect (we tried more steps to confirm this) and a reasonable execution time below 4 minutes in the worst cases. Since optimization is performed only once before the training starts, we argue that this is fully acceptable.

Theorem 12. *Under Assumptions 4, the problem of finding the minimal processing time for the chain from Figure 2.16, using operations from Table 2.2 together with offloading O_ℓ and prefetching P_ℓ, under memory limit M_{GPU} discretized with N values and bandwidth*

β, can be solved optimally with a dynamic program with a complexity of $O(L^2N^3 + L^3N^2)$ up to discretization effects.

4.2 Dynamic Programming to Compute the Optimal Sequence

In this section, we detail the dynamic programming equations whose intuition has been given in Section 4.1.

In order to proceed, one should know the minimal memory requirements for each operation. Given the chain between s and t, taking into account the data dependencies from Table 2.2 and the fact that the global input of the chain $\mathcal{I}_s^x = (1-x)a_{s-1} + x\bar{a}_{s-1}$ (it is equal to a_{s-1} when $x = 0$ and $\bar{a}_{s-1}$ when $x = 1$) is already stored in the memory (and counted separately in all equations) and the current gradient value should be stored at any time, then

- $\mathcal{M}^{s,t}_{F_s^\varnothing} = \delta_t + a_s + o_{F_s}$;
- $\forall h \neq s : \mathcal{M}^{s,t}_{F_h^\varnothing} = \delta_t + a_{h-1} + a_h + o_{F_h}$;
- $\mathcal{M}^{s,t}_{F_s^{all}} = \delta_t + \bar{a}_s + o_{F_s}$;
- $\forall h \neq s : \mathcal{M}^{s,t}_{F_h^{all}} = \delta_t + a_{h-1} + \bar{a}_h + o_{F_h}$;
- $\mathcal{M}^{s,t}_{B_s} = \bar{a}_s + \delta_s + \delta_{s-1} + o_{B_s}$;
- $\forall h \neq s : \mathcal{M}^{s,t}_{B_h} = a_{h-1} + \bar{a}_h + \delta_h + \delta_{h-1} + o_{B_h}$.

4.2.1 Forward Phase

In this section, we provide the analysis of the forward phase, *i.e.* for all operations that take place before the computation of the loss. We consider the subchain that starts from layer s and assume that its input $\mathcal{I}_s^x$ is saved by F_s^{ck} or F_s^{all}. To denote the total duration of the execution from F_s to B_s, we use $\mathrm{OC}_x(s, A_s, \Delta_{F_s}, \Delta_{B_s})$. We also distinguish between $\mathrm{OC}_x^{F^{all}}$ (resp. $\mathrm{OC}_x^{F^{ck}}$) that represents the duration of the chain execution when the first operation is constrained to be F_s^{all} (resp. F_s^{ck}). By construction, the first operation cannot be $F_s^\varnothing$ since we assume that the input of the chain must not be discarded.

Case 1: F_s^{all} is the first operation

If the first operation is F_s^{all}, then in the optimal schedule the next operation cannot be $F_{s+1}^\varnothing$. Indeed, as F_s^{all} stores $\bar{a}_s$ together with the input $\mathcal{I}_s^x$, then this $\bar{a}_s$ should stay in the memory until the backward operation B_s. At the same time, as $F_{s+1}^\varnothing$ discards its input after its execution, then each time when we need to recompute a_s we need to recompute F_s. Therefore, if the next operation is $F_{s+1}^\varnothing$, the overall duration does not change if the first instance of F_s^{all} is replaced by F_s^{ck}, while F_s^{all} replaces the last F_s that computes a_s in the sequence. Simultaneously, this transformed sequence keeps less data in memory, which contradicts the optimality of F_s^{all} being followed by $F_{s+1}^\varnothing$.

Since we assume memory persistence (see **OR.1**),*i.e.* $\bar{a}_s$ stays in the memory until the backward operation B_s, the sequence after F_s^{all} and before B_s can be computed

	offload input $v = 1$	no offload $v = 2$
new A^v	A_s	$A_s + \mathcal{I}_s^x$
new Δ_F^v	$\max\{\Delta_{F_s} + \mathcal{I}_s^x - D_F, 0\}$	$\max\{\Delta_{F_s} - D_F, 0\}$
new Δ_B^v	$\max\{\Delta_{B_s} - D_B, 0\} + \mathcal{I}_s^x$	$\max\{\Delta_{B_s} - D_B, 0\}$

Таблица 4.1: Values for the new state

with a recursive call to the dynamic program. After this sequence, B_s can be directly executed, having all the necessary input already stored in device memory. This shows that $\mathrm{OC}_x^{F^{all}}(s, A_s, \Delta_{F_s}, \Delta_{B_s})$ can be computed as

$$\mathrm{OC}_x^{F^{all}}(s, A_s, \Delta_{F_s}, \Delta_{B_s}) = u_{F_s} + u_{B_s} + \min_{v=1,2} \mathrm{OC}_{x=1}(s+1, A_{s+1}^v, \Delta_{F_{s+1}}^v, \Delta_{B_{s+1}}^v) + \epsilon_F + \epsilon_B,$$

where the values ϵ_F and ϵ_B represent the idle time required for communications in the forward and backward respectively, and can be determined using Lemma 8. Taking into account that the maximal memory occupation for F_s^{all} and B_s are given by $M_{F_s}^x$ and $M_{B_s}^x$ respectively, we obtain

$$\epsilon_F = \max\left(\frac{\mathcal{M}_{F_s^{all}}^{s,L} - M_{\mathrm{GPU}} + M_{F_s}^x}{\beta}, 0\right) \text{ and } \epsilon_B = \max\left(\frac{\mathcal{M}_{B_s}^{s,L} - M_{\mathrm{GPU}} + M_{B_s}^x}{\beta}, 0\right).$$

We also need to compute the new state variables $A_{s+1}^v, \Delta_{F_{s+1}}^v, \Delta_{B_{s+1}}^v$, where the value of v indicates whether the input $\mathcal{I}_s^x$ is offloaded or not. During the forward step, the communication link is able to offload a quantity of data given by $D_F = (u_{F_s} + \epsilon_F)\beta$; similarly, the data that can be prefetched during the backward step is given by $D_B = (u_{B_s} + \epsilon_B)\beta$. These values can be used to obtain the new state variables, as described in Table 4.1.

The final sequence is valid if memory limit is not violated. More specifically, if $\mathcal{M}_{F_s^{all}}^{s,L} + \mathcal{I}_s^x > M_{\mathrm{GPU}} - A_s$ or $\mathcal{M}_{B_s}^{s,L} + \mathcal{I}_s^x > M_{\mathrm{GPU}} - A_s$ then we set $\mathrm{OC}_x^{F^{all}}(s, A_s, \Delta_{F_s}, \Delta_{B_s}) = \infty$.

Case 2: F_s^{ck} is the first operation

After an F_s^{ck} operation, any forward operation or offload operation is possible. Let us assume that the next saved activation is a_t for some $t \geq s$, which implies that after F_s^{ck} there is a sequence of $F_k^{\varnothing}$ for $s < k \leq t$. Due to memory persistence, it also implies that the sequence processing layers from $t+1$ till L can be obtained recursively, looking at $\mathrm{OC}_{x=0}(t+1, A_{t+1}^v, \Delta_{F_{t+1}}^v, \Delta_{B_{t+1}}^v)$ (once again, v denotes whether the input $\mathcal{I}_s^x$ is offloaded or not). Afterwards, processing B_k for $s < k \leq t$ requires recomputing forward operations from a_{s-1} (or $\bar{a}_{s-1}$). Since no offloading operation can be performed after the computation of the loss, this corresponds to computing a rematerialization sequence between layers s and t that should have Δ_{B_s} prefetched by the end. $C_\Delta(s, t, A_s + \mathcal{I}_s^x, \Delta_{B_s})$ is used to compute the optimal duration to process layers from s to t, having $A_s + \mathcal{I}_s^x$ as a cumulative storage of all activations at the GPU, and Δ_{B_s} that corresponds to the data that has to

be prefetched. We show in Section 4.2.3 how to compute $C_\Delta(s,t,A,\Delta)$. This yields the formula for $\mathrm{OC}_x^{F^{ck}}(s,A_s,\Delta_{F_s},\Delta_{B_s})$

$$\mathrm{OC}_x^{F^{ck}}(s,A_s,\Delta_{F_s},\Delta_{B_s}) = \min_{s\le t\le L-1}\left(\sum_{k=s}^{t} u_{F_k} + \min_{v=1,2}\mathrm{OC}_{x=0}(t+1,A^v_{t+1},\Delta^v_{F_{t+1}},\Delta^v_{B_{t+1}}) + C_\Delta(s,t,A_s+\mathcal{I}^x_s,\Delta_{B_s}) + \epsilon_F\right),$$

where ϵ_F is computed with the help of Lemma 8

$$\epsilon_F = \max\left(\frac{\max_{k=s,\dots,t}(\mathcal{M}^{s,L}_{F^\varnothing_k} - \beta\sum_{h=s}^{k-1}u_{F_h}) - M_{\mathrm{GPU}} + M^x_{F_s}}{\beta}, 0\right).$$

Similarly to the Case 1, the values of the new state variables A^v, Δ^v_F and Δ^v_B can be updated using the formulas in Table 4.1. The only difference is the possible amount of offloaded data and prefetched data. In this case $D_F = (\sum_{k=s}^{t} u_{F_k} + \epsilon_F)\beta$ and $D_B = C_\Delta(s,t,A,\Delta_{B_s})\beta$.

The final sequence is valid if memory limit is not violated. Thus, if for some $k \ge s$ we have $\mathcal{M}^{s,L}_{F^\varnothing_k} + \mathcal{I}^x_s > M_{\mathrm{GPU}} - A_s$ when $\mathcal{I}^x_s$ is not offloaded (or when $k = s$) or $\mathcal{M}^{s,L}_{F^\varnothing_k} > M_{\mathrm{GPU}} - A_s$ when $\mathcal{I}^x_s$ is offloaded, then we set $\mathrm{OC}_x^{F^{ck}}(s,A_s,\Delta_{F_s},\Delta_{B_s}) = \infty$.

Combining everything together

Therefore, $\mathrm{OC}_x(s,A_s,\Delta_{F_s},\Delta_{B_s})$ can be computed as

$$\mathrm{OC}_x(s,A_s,\Delta_{F_s},\Delta_{B_s}) = \min\begin{cases}\mathrm{OC}_x^{F^{all}}(s,A_s,\Delta_{F_s},\Delta_{B_s})\\ \mathrm{OC}_x^{F^{ck}}(s,A_s,\Delta_{F_s},\Delta_{B_s})\end{cases} \tag{4.3}$$

4.2.2 Loss: How to Concatenate Forward and Backward Phases

In Section 4.2.1, we have shown how to compute the optimal duration of the sequence, using dynamic programming with $\mathrm{OC}_x(s,A_s,\Delta_{F_s},\Delta_{B_s})$. This dynamic program finds the solution through recursive calls to smaller sub-chains, until reaching the subchain consisting of only loss computation. We further represent the loss computation with F_{L+1} and B_{L+1} operations. The loss should be computed with F^{all}_{L+1} and B_{L+1}, so that this case is similar to $\mathrm{OC}_x^{F^{all}}$ and

$$\mathrm{OC}_x(L+1,A_{L+1},\Delta_{F_{L+1}},\Delta_{B_{L+1}}) = u_{F_{L+1}} + u_{B_{L+1}} + \epsilon_F + \epsilon_B + \epsilon_G,$$

where ϵ_F and ϵ_B are found with the same expressions as the idle times for Case 1 of Forward phase (see Section 4.2.1).

As no offloading is possible during backward propagation and no prefetching is possible during forward propagation (see **OR.3**), the idle time between the phases comes from the completion of both communication tasks, *i.e.*

$$\epsilon_G = (\Delta_{F_{L+2}} + \Delta_{B_{L+2}})/\beta,$$

where $\Delta_{F_{L+2}}$ and $\Delta_{B_{L+2}}$ can be computed from $\Delta_{F_{L+1}}$ and $\Delta_{B_{L+1}}$, using formulas from Table 4.1 when the input is not offloaded, given $D_F = (u_{F_{L+1}} + \epsilon_F)\beta$ and $D_B = (u_{B_{L+1}} + \epsilon_B)\beta$.

4.2.3 Backward Phase

The situation in the case of backward is a bit more complex. We must indeed perform all the operations $B_t, \ldots, B_{s+1}$, with only $\mathcal{I}_s^x$ and δ_t into memory. This is nevertheless enough since we can execute the whole forward chain $F_{s+1}, \ldots, F_t$ from a_s and then the whole chain $B_t, \ldots, B_{s+1}$ using the values computed during the processing of the forward chain.

In general, due to memory constraints, it is not possible to perform $F_{s+1}, \ldots, F_t B_t, \ldots, B_{s+1}$ in sequence. Since we assume that offloading cannot take place in the backward phase (see **OR.2**), we rely on Rematerialization in order to save memory if needed. Our goal is to find a valid schedule, that satisfies memory constraints and whose duration is minimal. One additional difficulty comes from prefetched data. Indeed, several valid rematerialization sequences for computing the backward steps will differ both by their duration and by the amount of data that can be prefetched during the sequence. Indeed, for a given Δ_{B_s}, the sequences that enable to prefetch a lot of data are preferable since they can induce a lower idle times for backwards $B_k, k > t$.

Let $C_\Delta(s, t, A, \Delta)$ denotes the optimal duration to execute the chain between layers s and t with A denoting the cumulative size of all activations already stored in GPU, *including input* $\mathcal{I}_s^x$, and given minimal available memory $M_{\text{GPU}} - A - \Delta$. In the beginning of the execution $\mathcal{I}_s^x$ and δ_t are stored in the device memory. Moreover, $\mathcal{I}_s^x$ must stay in the memory until the end of the execution. Therefore, the first operation should be F_s^{all} or F_s^{ck}. Depending on the first operation, the optimal duration can be read in $C_\Delta^{F^{all}}(s, t, A, \Delta)$ or $C_\Delta^{F^{ck}}(s, t, A, \Delta)$ respectively.

Case 1: F_s^{all} is the first operation

Let us first notice that after F_s^{all} the activation $\bar{a}_s$ must be kept in the memory until its associated backward operation (due to memory persistence **OR.1**). We must solve the subproblem of finding the optimal duration for the chain between layers $s+1$ and t, which can be scheduled optimally with $C_\Delta(s+1, t, A + \bar{a}_s, \Delta_1)$, according to our assumption. Once B_{s+1} is performed, then B_s can be directly executed, as all necessary input are already in the device memory. Therefore, $C_\Delta^{F^{all}}(s, t, A, \Delta)$ is given by

$$C_\Delta^{F^{all}}(s, t, A, \Delta) = u_{F_s} + C_\Delta(s+1, t, A + \bar{a}_s, \Delta_1) + u_{B_s} + \epsilon_{F_s^{all}} + \epsilon_{B_s},$$

where Δ_1 is defined by

$$\Delta_1 = \max\{\Delta - (u_{B_s} + \epsilon_{B_s})\beta, 0\}.$$

We can also estimate the minimal available memory when F_s^{all} is executed, which is given by $M_{\text{GPU}} - A - \Delta_2$ where

$$\Delta_2 = \max\{\Delta_1 - C_\Delta(s+1, t, A + \bar{a}_s, \Delta_1)\beta, 0\}.$$

The idle times caused by prefetching can in turn be computed with the help of Lemma 8 as

$$\epsilon_{B_s} = \max\left(\frac{\mathcal{M}_{B_s}^{s,t} - M_{\text{GPU}} + A + \Delta}{\beta}, 0\right) \text{ and } \epsilon_{F_s^{all}} = \max\left(\frac{\mathcal{M}_{F_s^{all}}^{s,t} - M_{\text{GPU}} + A + \Delta_2}{\beta}, 0\right).$$

$\mathcal{M}_{F_s^{all}}^{s,t} > M_{\text{GPU}} - A$ or $\mathcal{M}_{B_s}^{s,t} > M_{\text{GPU}} - A$ shows that there is not enough memory to perform this sequence, and as previously the dynamic programming cost should be set to ∞.

Case 2: F_s^{ck} is the first operation

Let us suppose that F_s^{ck} is used to save $\mathcal{I}_s^x$ and consider the next value $a_{s'}$ to be kept in memory. To compute this value, a sequence of $F^\varnothing$ operations from layer $s+1$ till layer s' is performed. Due to memory persistence, after checkpointing $a_{s'}$ we can find the optimal sequence from $s'+1$ to t by reading $C_\Delta(s'+1, t, A + a_{s'}, \Delta_2)$. Once $B_{s'+1}$ is performed, the chain from s to s' can be scheduled using $C_\Delta(s, s', A, \Delta_1)$. Thus, $C_\Delta^{F^{ck}}(s, t, A, \Delta)$ is given by

$$C_\Delta^{F^{ck}}(s, t, A, \Delta) = \min_{s \le s' < t} \sum_{k=s}^{s'} u_{F_k} + C_\Delta(s'+1, t, A + a_{s'}, \Delta_2) + C_\Delta(s, s', A, \Delta_1) + \epsilon_F$$

The parameters for the recursive call of the dynamic program can be found using

$$\Delta_1 = \Delta \text{ and } \Delta_2 = \max\{\Delta_1 - C_\Delta(s, s', A, \Delta_1)\beta, 0\}.$$

Similarly to the previous case, we can estimate the minimal available memory when forwards from s to s' are performed for the first time. This minimal memory is given by $M_{\text{GPU}} - A - \Delta_3$, where

$$\Delta_3 = \max\{\Delta_2 - C_\Delta(s'+1, t, A + a_{s'}, \Delta_2)\beta, 0\}.$$

The idle time caused by prefetching can be computed with the help of Lemma 8 as

$$\epsilon_F = \max\left(\frac{\max_{k=s,\dots,s'}(\mathcal{M}_{F_k^\varnothing}^{s,t} - \beta(\sum_{h=k+1}^{s'} u_{F_h})) - M_{\text{GPU}} + A + \Delta_3}{\beta}, 0\right).$$

$\mathcal{M}_{F_k^\varnothing}^{s,t} > M_{\text{GPU}} - A$ for some $k,\ s \le k < t$ shows that there is not enough memory to perform this sequence, and as previously the dynamic programming cost should be set to ∞.

Initialization: solution for a single layer Each recursive call to the dynamic program is solving the problem for strictly smaller sub-chains. The recursion stops when reaching the sub-chain consisting of only one layer. The schedule for one layer s is straightforward: perform F_s^{all} and B_s, *i.e.*

$$C_\Delta(s, s, A, \Delta) = u_{F_s} + u_{B_s} + \epsilon$$

where, according to Lemma 8

$$\epsilon = \max\left(\frac{\mathcal{M}_{F_s^{all}}^{s,s} - \beta u_{B_s} - M_{\text{GPU}} + A + \Delta}{\beta}, \frac{\mathcal{M}_{B_s}^{s,s} - M_{\text{GPU}} + A + \Delta}{\beta}, 0\right).$$

Again, this solution will be infeasible if there is not enough memory to perform any operation, *i.e.* if $\mathcal{M}_{F_s^{all}}^{s,s} > M_{\text{GPU}} - A$ or $\mathcal{M}_{B_s}^{s,s} > M_{\text{GPU}} - A$. In this case, the dynamic programming cost should be set to ∞.

Combining everything together:

Therefore, $C_\Delta(s, t, A, \Delta)$ can be computed as

$$C_\Delta(s, t, A, \Delta) = \min \begin{cases} C_\Delta^{F^{all}}(s, t, A, \Delta) \\ C_\Delta^{F^{ck}}(s, t, A, \Delta) \end{cases} \tag{4.4}$$

4.2.4 Complexity

Finally, finding the optimal duration schedule for a chain of length L corresponds to computing $\text{OC}_{x=0}(1, 0, 0, 0)$ using the dynamic program presented above. The analysis of the complexity of the above dynamic program can be decomposed in two parts. Let N denote the number of discretized values for the memory M_{GPU}. During the backward phase, the size of the state space is $O(L^2N^2)$, and for Case 2, computing a new value requires $O(L)$ operations, which leads to $O(L^3N^2)$ operations. In the forward phase, the size of the state space is $O(LN^3)$, and again computing a new value requires $O(L)$ operations. This results in $O(L^2N^3)$ operations for the forward phase. Therefore, the overall complexity is given by $O(L^2N^3 + L^3N^2)$. In practice, we observe that for all the experiments presented in Section 4.4, discretizing the memory with $N = 50$ values is enough since considering a finer discretization does not lead to any practical improvement.

4.3 Heuristics

We have implemented two other sophisticated heuristics for comparison purposes. These heuristics are not bound with the Assumptions 4, but come without any optimality guarantee for the produced sequences.

In the first heuristic, called `opportunist`, the objective is to use the communication medium as much as possible. We compute the first layer with F^{all} and offload its input and output, ensuring that the memory will remain fully available for the rest of the

computation. The next layers are computed with $F^{\varnothing}$ until the end of communications. We then start a new communication by performing the next layer with F^{all} and offloading its input and output, and so on until the end of the sequence can be entirely performed in memory. This process thus builds *groups* of layers between two F^{all} operations. We then compute the backward phase for each group using the `rematerialization` algorithm (see Chapter 2 Section 2.3.4), and concatenate them (with the necessary prefetches) to obtain the final sequence. Note that `opportunist` is conceptually close to the implementation from Superneurons [108].

The second heuristic is called `autocapper`. It relies on an internal `capper` algorithm, that uses an increased memory limit $M' > M_{\text{GPU}}$ as an additional input. `capper` computes a pure rematerialization sequence with the limit M', finds the peak memory usage in the sequence and offloads the lowest indexed activation present in GPU memory at that time. This process is repeated until the sequence fits in the memory limit M_{GPU}. `autocapper` calls `capper` with 40 values of M', evenly spaced between the target M_{GPU} and M^{high}, the memory required without Rematerialization or Offloading. For each value M', the resulting sequence from `capper` is simulated and the best one is kept.

Note that, when `autocapper` does not perform recomputations, it behaves as the Greedy heuristic from Chapter 3 Section 3.2.

4.4 Experiments

4.4.1 Simulation Results

We measured running times and memory occupation of several networks from PyTorch `torchvision` package: ResNet, DenseNet and Inception, with a batch size of 16 and images of 500×500 pixels. We also experimented with other batch and image sizes, and obtained very similar results. Time measurements were performed on a NVidia Tesla V100 GPU with 16GB of memory. We also measured the bandwidth obtained when transferring PyTorch tensors from and to the GPU and obtained 12GB/s. The simulation results presented here were obtained using 4 cores of a 24-core Haswell Intel® Xeon® E5-2680 v3 at 2,5 GHz, with 128GB of memory, and used about one hour of total computation.

We consider 5 algorithms in total: our dynamic program `pofo`, the optimal `rematerialization`-only algorithm from Chapter 2 Section 2.3.4, the optimal `offload`-only approach (DynProg heuristic from Chapter 3 Section 3.3), and both heuristics `autocapper` and `opportunist`. We use the `sequential` time as a reference: it is the time that it would take to process the forward and backward phases with infinite memory, equal to the sum of forward and backward times of all operations. For each network, we compute the highest and lowest memory requirements (denoted respectively M^{high} and M^{low}): M^{high} is obtained with the `sequential` approach, while M^{low} is obtained by recomputing everything from the beginning at each step of the backward phase. We can thus explore the whole range of achievable memory sizes for this network, by considering values within the interval $[M^{\text{low}}, M^{\text{high}}]$: for a given ratio $\alpha \in [0, 1]$, the memory limit

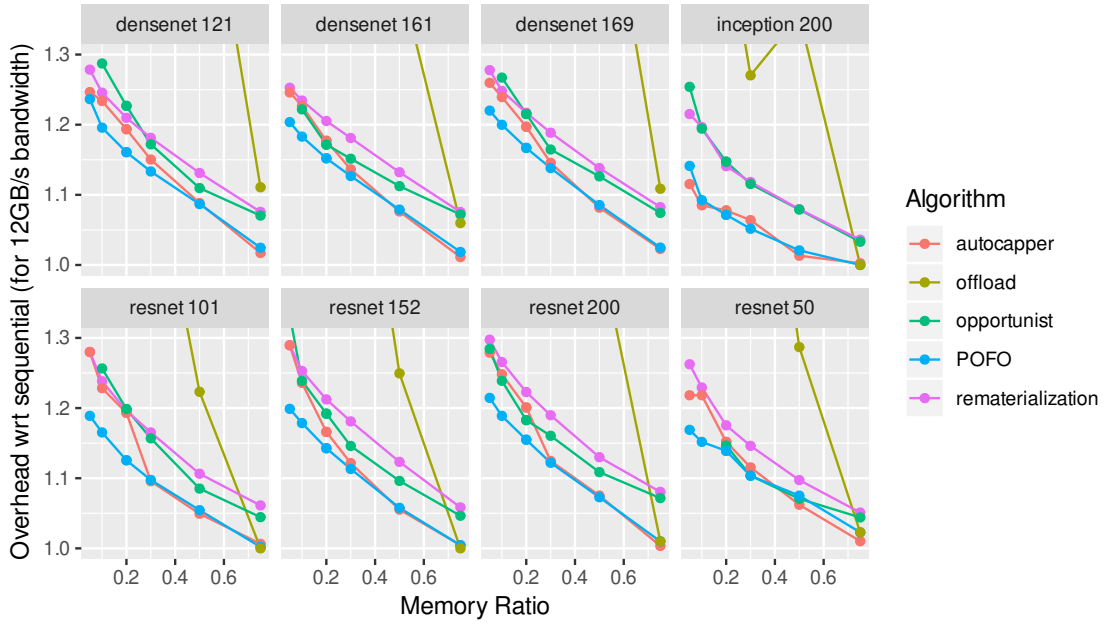


Рис. 4.2: Simulation results for fixed bandwidth $\beta = 12\text{GB/s}$ and varying memory ratio α.

M_{GPU} is set to $(1-\alpha)M^{\text{low}} + \alpha M^{\text{high}}$. For the algorithms which perform Offloading, we also vary the bandwidth value β, from 12GB/s (corresponding to a realistic scenario) to 36GB/s (corresponding to possible future improvements in communication capabilities).

The results are shown on four plots: Figure 4.2 presents the behavior of the algorithms for fixed bandwidth and varying memory constraint, Figure 4.3 explores the effects of increasing the bandwidth for a given memory limit, Figure 4.4 shows what is the portion of the total activation data that is transferred to the CPU for different memory limits, while Figure 4.5 depicts how this value changes with varying bandwidth for $\alpha = 0.2$.

We draw the following conclusions:

- When memory is large ($\alpha = 0.75$), Offloading is more effective than Rematerialization: with $\beta = 12\text{GB/s}$, there is time to offload and prefetch enough data to run the whole chain. However, for smaller memory limits, the `offload`-only policy exhibits much worse performance than `rematerialization` (unless more bandwidth is available, see Figure 4.3).
- Unless in some cases with small memory, the `opportunist` policy achieves slightly better performance than pure `rematerialization`, but is sometimes unable to produce a solution when the memory limit is too low. On the other hand, the more sophisticated `autocapper` algorithm obtains very good performance for memory ratios above 0.5.
- The `pofo` algorithm successfully combines the advantages of both Rematerialization and Offloading, and consistently outperforms both of them. When the memory limit

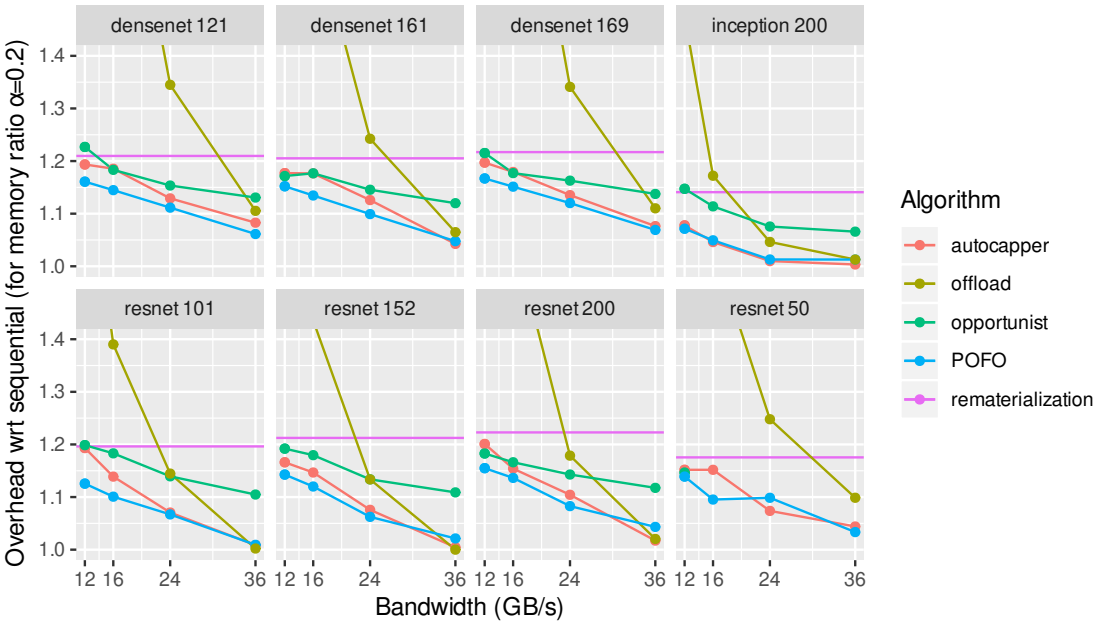


Рис. 4.3: Simulation results for fixed memory ratio $\alpha = 0.2$ and varying bandwidth.

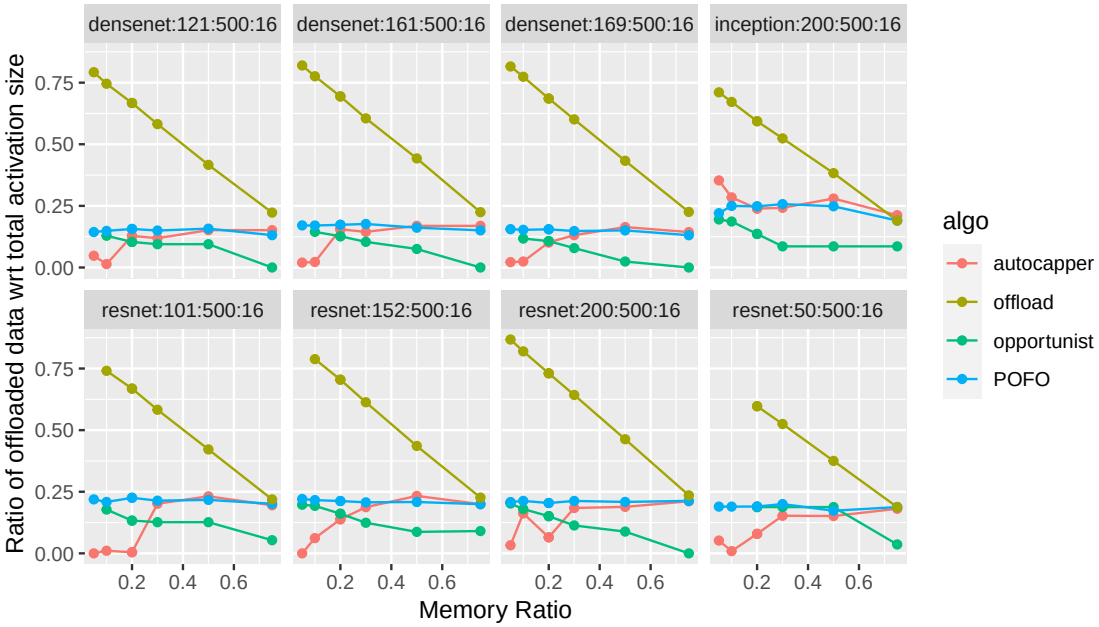


Рис. 4.4: Relative size of data transferred to the CPU *w.r.t.* the total activation size for a fixed bandwidth $\beta = 12$GB/s.

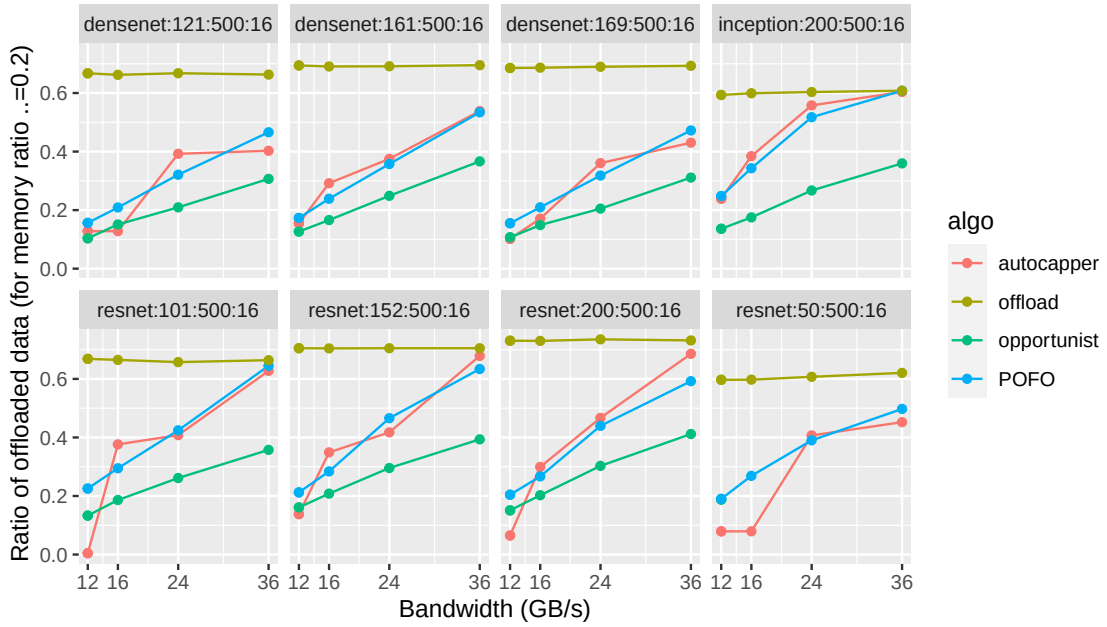


Рис. 4.5: Relative size of data transferred to the CPU *w.r.t.* the total activation size for varying bandwidth and fixed memory ratio $\alpha = 0.2$

is high enough, the optimization problem is relatively easy, and `autocapper` and `pofo` achieve similar performance. In some cases `autocapper` is marginally better than `pofo`, as `pofo` produces solutions under Assumptions 4, while partial memory releases (assumption **OR.5**) are difficult to implement in practice. For lower memory limits, the more optimized `pofo` algorithm is able to produce much better sequences.

- As can be seen on Figure 4.3, the `offload`-only approach works perfectly when the bandwidth is high. Indeed, if there is enough communication capability to offload all the data while it is produced, it is possible to avoid recomputations without inducing idle time. Compared to pure offloading techniques, our `pofo` solution allows to train models with large activations with cheaper communication links.
- From Figure 4.4, for $\beta = 12$GB/s and a given neural network, `pofo` tends to offload the same amount of data for different memory limits, while `autocapper` keeps the same rate when memory limit is high and then it drops to zero when memory limit is near M^{low}. Possible explanation for `pofo` offloading a constant amount of data might be the fact that `pofo` does offloading non-stop throughout the entire forward propagation. Since `pofo` does not offload during the backward phase and the forward phase duration is constant, `pofo` always offloads roughly the same quantity of data. This difference in the behavior of `pofo` and `autocapper` may explain the advantage of `pofo` when available memory is scarce.
- From Figure 4.5, it can be observed that the amount of offloaded data grows almost linearly with bandwidth for `pofo` and `opportunist` (larger bandwidth implies that

proportionally more communications can be overlapped with forward propagation).
- Compared to the `sequential` sequences, our `pofo` algorithm allows to divide the memory used by the activations by a factor 4 to 6, with an overhead below 20%.

4.4.2 Experimental Results with ROTOR

Due to efficiency of `pofo` and `autocapper`, they can be integrated into ROTOR to improve its performance. It is possible to offload data in PyTorch (preview version at the moment) using the function `saved_tensors_hooks(pack_hook, unpack_hook)`, which will be available in the future version PyTorch 1.10. This function allows to register a hook that can capture all tensors generated by an operation and then "pack" them (compress or offload) during the forward propagation and "unpack" them (extract or prefetch) during the backward propagation. We have implemented a preliminary version of algorithms `pofo` and `autocapper` and made them available in ROTOR [48]. We further present the current results in Figure 4.6 obtained by testing these new extensions of ROTOR on the same neural networks as in Section 4.4.1.

Overall, these preliminary experiments confirm that combining Offloading with Rematerialization allows to significantly improve over pure Rematerialization in most scenarios. In particular, `pofo` is still the best algorithm for memory optimization among the ones considered in the plot. For most cases, it either shows the smallest overhead or it behaves in the same way as other methods. However, DenseNet-169 is the exception to this trend, where `rematerialization`-only is strictly better than `pofo` or `autocapper`, but even in this case `pofo` outperforms `autocapper`. On the other hand, the case of ResNet-101 demonstrates the significant improvement of `pofo` over other strategies, suggesting that there exist cases for which the benefit is important.

This discrepancy in the simulation results and ROTOR results comes from the fact that the current implementation of `pofo` and `autocapper` in ROTOR uses more memory than it is expected. We believe that further optimization of this preliminary implementation should allow to obtain even better performance.

4.5 Conclusion

In this chapter, we have formalized the problem of the optimal combination of Rematerialization and Offloading, which are two classical strategies for coping with memory limitations on a GPU. We have shown that the optimal solution can be computed using dynamic programming, in a few seconds or a few minutes for very deep networks. From experiments, we have shown that the combination of Offloading and Rematerialization is very efficient and allows, in many cases, to transparently perform training with 4 to 6 times less memory, at the cost of a 10-20% time overhead.

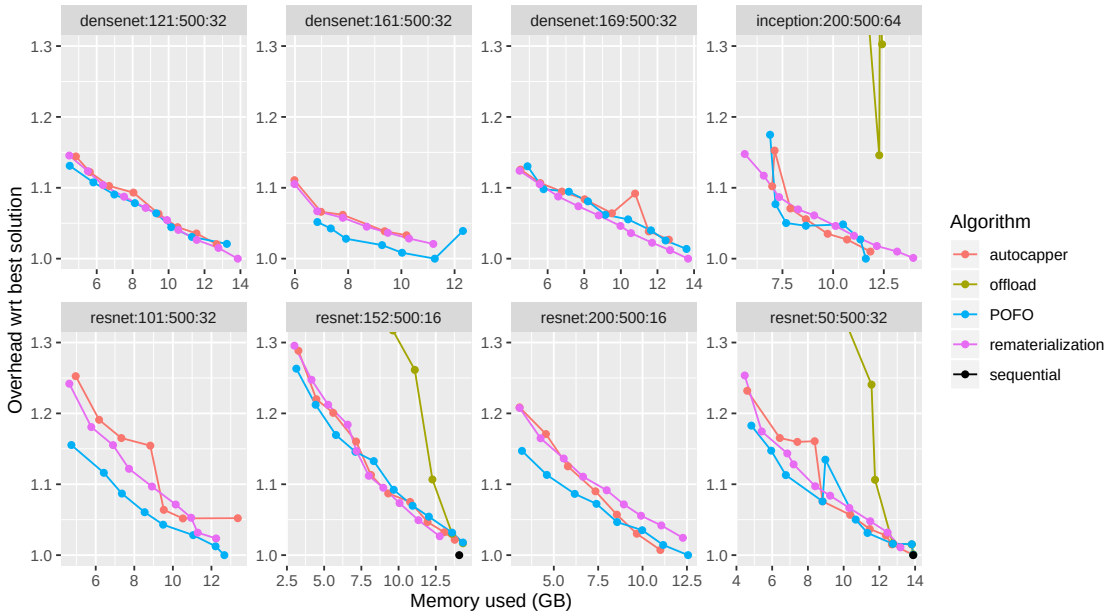


Рис. 4.6: Relative execution time of ROTOR implementations of `pofo`, `autocapper` and `rematerialization`-only methods for a fixed bandwidth $\beta = 12\text{GB/s}$. The results are obtained by performing actual runs of neural network training loops with ROTOR

Часть III

Model Parallelism

Introduction

In the previous chapters, we considered memory saving techniques for training on a single device. In practice, neural networks are often trained in parallel. Moreover, both small groups of GPU machines and large HPC infrastructures [112] are commonly used, especially when HPC machines offer high-bandwidth and low-latency networks [71, 22]. Parallelism is very efficient in distributing the load and it can be done in several different ways.

The first approach is to use parallelism at the level of the node, which makes the best use of the available multi-core by optimizing the individual compute kernels, which usually consist of tensor computations. This approach has been widely used in the context of GPUs and TPUs and has made the success of frameworks such as TensorFlow [2] or PyTorch [82].

At a larger scale, the best known approach to parallel DNN training is the so-called data parallel approach. Using Data Parallelism [116], the model weights are replicated on all participating nodes. Then, different mini-batches are trained in parallel on different nodes: all participating nodes execute forward and backward phases in parallel, and thus all of them compute gradients for all weights in the network. Synchronization between the nodes takes place at the end of the backward step, and all gradients are collected and aggregated through collective communications. The above approach is possible as long as two conditions are fulfilled: (i) the communication network infrastructure must be able to support the collective communications of the weights without inducing too much idle time and (ii) each participating node must be able to store all network (model) weights and activations corresponding to the processing of a mini-batch.

In many cases, deep and heavy models bring better prediction quality, but they may have large memory requirements, which makes the training impossible. As it was discussed above, the memory consumption during the training phase is composed of two main parts [34]: the storage of forward activations until the associated backward operation and the storage of the DNN weights. To limit the memory requirements resulting from the storage of network weights, a natural approach is to distribute the layers of the DNN over several computation resources. This approach, known as Model Parallelism, has been advocated in many papers [24, 50, 78, 113]. Each batch is processed by a sequence of processors, and only activations are communicated between processors. This approach is orthogonal to Data Parallelism and can naturally be combined with it (a batch is divided into several mini-batches that are processed at the same time, while Model Parallelism is used to execute one mini-batch). Even though, Model Parallelism can actually reduce memory requirements, it cannot accelerate computations because of the sequential nature of the training: operations of forward and backward propagation cannot be executed independently (see Figure 4.7). To obtain some speedup using this approach, it is necessary to process several mini-batches in parallel, using a pipelined approach. However, in turn, processing several mini-batches simultaneously induces extra memory requirements as it was shown in [78].

All known solutions for Pipelined Model Parallelism [78, 79] rely on a certain number

of assumptions, that make the problem tractable and allow to derive practical solutions. In particular, these solutions only consider (i) *contiguous* allocations, where each processor is assigned a contiguous set of layers from the network and (ii) simplistic periodic schedules, where all processors alternate between one forward and one backward computations.

In Chapter 5, we establish the complexity of both resource allocation and scheduling problems in Section 5.1. We also show that from a theoretical perspective, allowing more general solutions provides significant improvement in terms of throughput. Indeed, k-periodic complex schedules with $k > 1$ (Section 5.2) help to improve a throughput when memory is a constraint. Furthermore, non-contiguous allocations (Section 5.3) are more flexible and can significantly improve load balancing.

In Chapter 6, we continue to study the advantages of the non-contiguous allocations. Despite the NP-completeness of the problem, we can express the optimization problem as an Integer Linear Programming. In Section 6.4, we carefully model the set of rules for the valid allocations and schedules, taking into account all sources of memory consumption. The performance of the ILP-based solution, both in terms of solution quality and running time are analyzed in Section 6.5.

The ILP from Chapter 6 provides optimal non-contiguous allocations for 1-periodic schedules. On the other hand, its computation cost is very high and time-consuming, making its usage difficult in practice. In Chapter 7, we propose an heuristic algorithm, called MadPipe (for Memory Aware Dynamic programming for PIPElining). This heuristic is presented in Section 7.1. It is based on the combination of dynamic programming and linear programming, with two main contributions: (i) a more precise estimation of the memory requirements than in PipeDream, which results in allocations that can be more efficiently scheduled, and (ii) the possibility to use non-contiguous allocations of layers of the DNN to processors, which provides a better load-balancing. The scheduling is performed with a modified ILP given in Section 7.2. The running time of MadPipe is acceptable in practice, and it significantly improves the resulting training performance with respect to PipeDream, which is demonstrated in Section 7.3.

Model and Notations

Notations

Like in the previous chapters, we consider linear DNNs, in which each forward operation depends only on the result of the previous operation, so that the network is a chain of L layers with the computation of $F_{L+1} = F_{\text{Loss}}$ at the end. Each layer ℓ, $1 \leq \ell \leq L+1$ is associated both to a forward operation F_ℓ and a backward operation B_ℓ (see Figure 4.7). During training, the input activation a_0 goes through all forward operations to compute a prediction whose the quality is estimated by a Loss. Then, the parameter weights of all layers have to be updated according to their effect on the loss, given by the partial derivative $\frac{\partial \text{Loss}}{\partial a_\ell}$, where the updates are performed by an *optimizer*, following some predefined strategy.

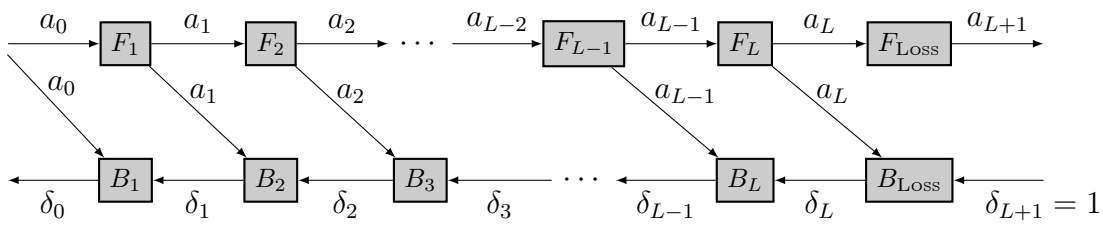


Рис. 4.7: Data dependencies induced the training phase of Linear Deep Neural Networks.

Unlike the previous chapters, we focus here only on the case of linear heterogeneous chains, whose nodes represent simple elementary functions that do not generate intermediate data. Therefore, the chain from Figure 4.7 does not have the downward paths from forward tasks to backward tasks. Only the diagonal paths still remain that connect the input of a forward task with its corresponding backward task. Thus, we consider just a simple BP-transform from Definition 2, where $\bar{a}$ tensors are not needed to perform the backward operations. This implies that the solutions provided in Chapters 5, 6 and 7 are not directly applicable to learning frameworks (see the discussion in Chapter 2), though their results can be easily extended to the more general linearized chains (Complex BP-transform from Definition 9, corresponding to Figure 2.16), but it is omitted here to ease the presentation of the main results of this part.

Further, we use the following notations:

- P indicates the number of computing resources, with memory M;
- β denotes the bandwidth between any two resources;
- u_{F_ℓ} denotes the duration of the forward task on the layer ℓ;
- u_{B_ℓ} denotes the duration of the backward task on the layer ℓ;
- W_ℓ denotes the memory occupation of the parameter weights for layer ℓ;
- a_ℓ refers to both the tensor and its memory occupation of the activation produced by F_ℓ;
- δ_ℓ refers to both the tensor and its memory occupation of the gradient produced by $B_{\ell+1}$; in practice, each gradient has the same memory size as the activation with respect to which it is calculated, *i.e.* $\delta_\ell = a_\ell$;

Most of these notations are not new and used as well in other chapters, except for W_ℓ that we now take into account when distributing the load onto processors. In this part, we assume that we have P identical processors (typically GPUs) with memory M and they are all interconnected with communication links of bandwidth β.

The goal of Model Parallelism is to distribute the layers of the DNN onto P computing resources with limited memory, so that each processor is in charge of a subset of the layers. The input activation thus goes through all processors to compute Loss, and is then backpropagated through all layers in reverse order to compute the corresponding gradients and update the weights. To avoid idle times, these computations are performed in a *pipelined* way (see Figure 4.8): the GPU in charge of layer ℓ may compute several forward operations F_ℓ before processing the first backward B_ℓ, so that it could stay busy

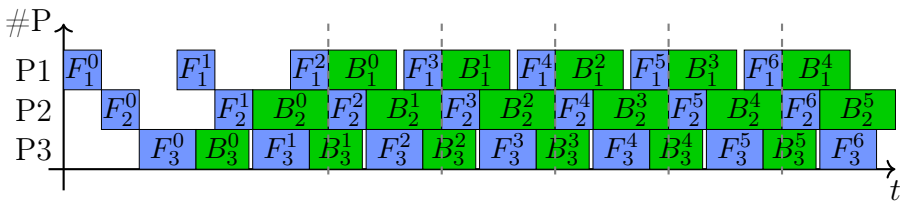


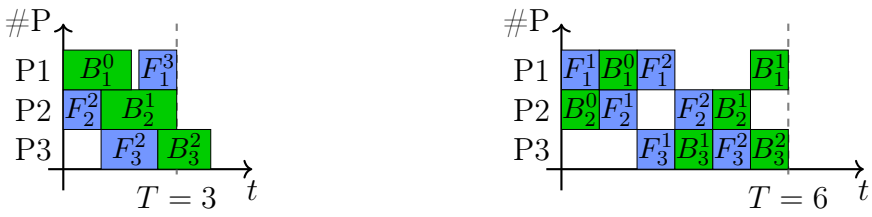
Рис. 4.8: Pipelined schedule for 3 layers and 3 processors. The superscripts indicate iteration numbers. Period is highlighted with dashed lines.

even while waiting for δ_ℓ to be computed by the other GPUs.

Throughout this chapter, we are interested in finding efficient task allocations and schedules. To help navigate the different concepts that we use, we introduce some terminology. We define the input DNN as a chain of *layers* (typically convolutional or dense layers), which are the basic operations that need to be computed, and a *partitioning* $\mathcal{P}$ of this chain is a collection of *stages*, where each stage contains a contiguous set of layers. An *allocation* $\mathcal{A}$ is an assignment of stages to the processors. An allocation is said to be *contiguous* if each processor is assigned a single stage, and by extension a partitioning is contiguous if it contains at most P stages. We use $\mathcal{A}_\mathcal{C}$ to denote the set of contiguous allocations, while the set of more general non-contiguous allocations is denoted as $\mathcal{A}_{n\mathcal{C}}$. To estimate memory requirements we also introduce *groups* of stages, where each group is a set of stages which are contiguous with respect to the ordering of the chain. A *schedule* $\mathcal{S}$ of a given allocation specifies the timings of all compute operations.

When assigning layers to the resources, it is important to take into account the communication time between them. Even though, once the allocation is fixed, each communication between layer ℓ on processor p and layer $\ell+1$ on processor p' can be represented as an additional pseudo-computation layer. It involves sending some activation a_ℓ between F_ℓ on p and $F_{\ell+1}$ on p', and a gradient δ_ℓ between $B_{\ell+1}$ on p' and B_ℓ on p, for a total time of $\frac{a_\ell+\delta_\ell}{\beta}$, where β is the bandwidth of the corresponding link used in exclusive mode. Therefore, an allocation on P processors with communication costs is equivalent to an allocation on $2P-1$ resources, without communication costs. Thanks to this transformation, we can ignore the communications between the tasks, when solving a scheduling problem for a fixed allocation.

In order to keep the description of schedules compact, we actually focus on periodic schedules. A schedule is periodic if it consists in the repetition of a pattern, and more precisely k-periodic if the pattern contains each computation task exactly k times. A k-periodic pattern of period T specifies for each operation (forward and backward): the processor in charge of it, a starting time t, and an index shift h. This pattern is to be repeated indefinitely: in the j-th period, this operation starts at time $jT+t$ and processes the mini-batch number $jk+h$. By convention the shift of the first B_1 operation of the pattern is always 0, so that if in some period this B_1 processes mini-batch index i, an operation with shift h processes mini-batch index $i+h$. A pattern is valid if the schedule obtained in this way is valid, *i.e.* fulfills the dependencies of Figure 4.7. Figure 4.9a



(a) 1-periodic pattern for the schedule of Figure 4.8.

(b) 2-periodic pattern for 3 layers where $u_F = u_B = 1$.

Рис. 4.9: Examples of periodic patterns. The superscripts indicate shift values.

shows an example of the 1-periodic pattern associated with the schedule of Figure 4.8, and Figure 4.9b shows a 2-periodic pattern. We use $\mathcal{S}^k$ to denote the set of k-periodic schedules.

Memory Constraints

In addition to enforcing data dependencies, we need to ensure that the schedules fit into the memory capacity M of the processors. As already noted, during the training phase, there are two main sources of memory usage: parameter weights, and forward activations. As discussed in [79], it is sufficient to keep two versions of the parameter weights. Moreover, as discussed in [86], a certain number of additional copies of the model, for gradients and optimizer states, are required. Their number only depends on the choice of the optimizer and not on the allocation or on the schedule. Overall, we denote with W_ℓ the memory occupied by model weights of some layer ℓ and C the overall number of model copies. On the other hand, with pipelined executions, several forward activations $a_{\ell-1}$ for a given layer ℓ need to be stored in memory at the same time, and this depends on the particular schedule. For instance, in the case of Figure 4.8, F_1^2, F_1^3 and F_1^4 simultaneously reside in memory before B_1^2 releases F_1^2.

To estimate memory needs, for a schedule $\mathcal{S}$, we define the *number of concurrent activations* (NCA) of layer ℓ as the maximum number of activations $a_{\ell-1}$ that are stored at any point in time. For a general schedule, this can be expressed as $\text{NCA}_\ell = \max_t \#F_\ell(t' < t) - \#B_\ell(t' < t)$, where $\#F_\ell(t' < t)$ counts the number of F_ℓ operations performed until time t. For a k-periodic schedule $\mathcal{S}$, NCA_ℓ can be computed from the values of the shifts: for any F_ℓ whose shift is h, if the preceding B_ℓ has shift h', then the number of concurrent activations just after this forward operation is $h - h'$. The value of NCA_ℓ for $\mathcal{S}$ is thus the maximum value of $h - h'$ over all forward operations F_ℓ. As an example, in the pattern of Figure 4.9a, $\text{NCA}_1 = 3$ and $\text{NCA}_2 = 2$ (here it is necessary to duplicate the pattern to find the preceding B_2), while for the pattern of Figure 4.9b, $\text{NCA}_1 = \text{NCA}_2 = 2$. Let us use $M^{\mathcal{S}}(p)$ to denote the memory peak achieved on processor p when schedule $\mathcal{S}$ is applied. Then, if a processor p processes a set of layers L_p, its memory usage can be approximated with $M^{\mathcal{S}}(p) \simeq \sum_{\ell \in L_p} CW_\ell + \text{NCA}_\ell a_{\ell-1}$ (temporary memory usage of each operations and communicated data is not accounted).

We can now formally define the scheduling problem for model parallelism.

Problem 12 ($\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$)**.** Given P processors with memory M and bandwidth β between them and L layers with forward and backward computation times u_{F_ℓ} and u_{B_ℓ}, parameter occupation W_ℓ and activation sizes a_ℓ (see Figure 4.7), we want to find an allocation $\mathcal{A}$ and a corresponding valid k-periodic schedule $\mathcal{S}^k$ for some $k \in \mathbb{N}$ with a period T, so that for all processors p, $M^{\mathcal{S}}(p) \leq M$, and which minimizes the normalized period T/k.

Sometimes, we are interested in finding the best schedule $\mathcal{S}$ for a fixed allocation scheme $\mathcal{A}$. Then problem $\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$ becomes the following.

Problem 13 ($\textsc{Pipe}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$)**.** Given an instance of problem $\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$ and for a fixed allocation $\mathcal{A}$, we want to find a valid k-periodic schedule $\mathcal{S}$ for some $k \in \mathbb{N}$ with a period T (so that for all processors $1 \leq p \leq P$, $M^{\mathcal{S}}(p) \leq M$), which minimizes the normalized period T/k.

In the following chapters, we analyze both problems and propose solutions to them.

Глава 5

Pipelined Model Parallelism. Complexity

This chapter covers some main theoretical results concerning problems $\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$ and $\textsc{Pipe}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$. We start with introducing the complexity results. After, we analyze the limitations of the state-of-the-art pipelining frameworks such as [78]. We distinguish two main limitations: the use of 1-periodic schedules and contiguous allocations. Both can be significantly outperformed by more general approaches and we explore the potential benefits of moving towards k-periodic schedules and non-contiguous allocations.

However, we do not propose solutions in this chapter and we leave it for the next chapters. The results of this chapter are purely theoretical and are obtained with assumption of zero communication costs, though they remain true even when communications are non-negligible.

5.1 Complexity Results

In this section, we analyze the complexity of $\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$. We first show that even without memory constraints, finding an optimal allocation is NP-complete problem. Then we consider the problem of finding a pattern for a fixed allocation $\textsc{Pipe}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$, and show that this problem is NP-complete because of memory constraints.

In both cases, we use a reduction from the 3-Partition problem [32]: given a set of integers $\{x_1, x_2, \ldots, x_{3n}\}$ such that $\sum_i x_i = nV$, is it possible to partition it into n parts $\{S_1, \ldots, S_n\}$ so that for any $j \leq n$, $|S_j| = 3$ and $\sum_{i \in S_j} x_i = V$. This problem is known to be NP-complete in the strong sense.

5.1.1 General Problem

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

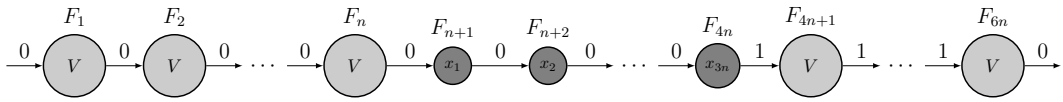


Рис. 5.1: Network Instance for the Proof of Theorem 14

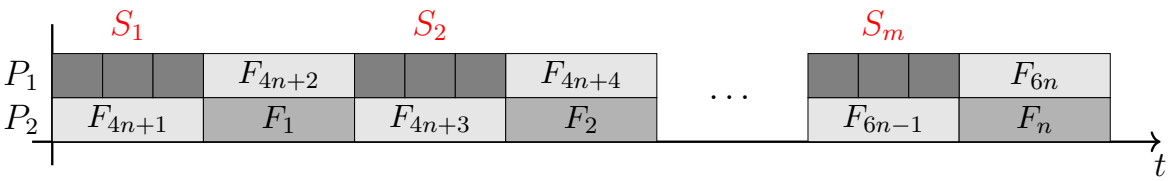


Рис. 5.2: Valid Pattern for the Proof of Theorem 14

Theorem 13. *The decision problem* $\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$ *of determining if there exists an allocation and a periodic pattern whose normalized period is at most* T *is strongly NP-Complete for any* M*.*

Доказательство. Given an instance of 3-Partition, we consider the following instance of our problem $\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$:

- $L = 3n$, $P = n$;
- $\forall \ell$, $a_\ell = W_\ell = 0$, so that memory constraints are not a concern;
- $\forall \ell \leq L$, $u_{F_\ell} = x_\ell$ and $u_{B_\ell} = 0$ (actually we can use any choice of values such that $u_{F_\ell} + u_{B_\ell} = x_\ell$).

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

Let us assume that there exists a solution to the 3-Partition instance. Then, we build a pattern where each group S_i is scheduled (in any order) on a different processor. There always exists a shift assignment such that the schedule is valid. Since there is no memory issues (all sizes are set to 0), we obtain a valid 1-periodic schedule.

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

5.1.2 Fixed Allocation Problem

In what follows, we prove in Theorem 14 that even when the allocation $\mathcal{A}$ is given, *i.e.* if we know which layer is assigned to which processor, the problem $\textsc{Pipe}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$ remains strongly NP-Complete. This demonstrates that both scheduling and resource allocation are difficult.

Theorem 14. *The allocation of layers* $\mathcal{A}$ *being fixed, the decision problem* $\textsc{Pipe}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$ *of determining if there exists a periodic schedule of normalized period at most* T *is NP-Complete in the strong sense.*

Доказательство. Given an instance of 3-Partition, we consider the following instance of our problem $\textsc{Pipe}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$, where the network is depicted in Figure 5.1 and the processing resources are defined as follows:

- $L + 1 = 6n$, $P = 2$, $M = n$, $\beta = 0$, the target period is $T = 2nV$;
- $a_{\ell-1} = 0$ for $1 \leq \ell \leq 4n$, and $a_{\ell-1} = 1$ for $4n + 1 \leq \ell \leq 6n$, while $W_\ell = 0$ for all ℓ;
- $u_{F_\ell} = V$ for $1 \leq \ell \leq n$ or $\ell \geq 4n + 1$, and $u_{F_\ell} = x_{\ell-n}$ for $n + 1 \leq \ell \leq 4n$;
- $u_{B_\ell} = 0$ for all ℓ;
- P_1 is assigned to all layers ℓ for $n + 1 \leq \ell \leq 4n$, and to even layers $4n + 2, 4n + 4, \ldots, 6n$;
- P_2 is assigned to all layers ℓ for $1 \leq \ell \leq n$, and to odd layers $4n+1, 4n+3, \ldots, 6n-1$.

Let us assume that there exists a solution to the 3-Partition instance. Then, we build a pattern where each group S_i is scheduled as depicted in Figure 5.2. Since $W_\ell = 0$ for all ℓ, the memory costs come from storing the activations. Moreover, all operations F_ℓ, $\ell \geq 4n + 1$ can use the same shift value. Since activation sizes a_ℓ are zero for $\ell \leq 4n$, the shift values of the other forward operations have no effect on the memory usage and can thus be chosen in a way that makes the pattern valid. Therefore, each layer $\ell \geq 4n+1$ has $\textsc{nca}_\ell = 1$, so the memory usage on each processor is exactly n. This shows that there exists a valid pattern of throughput T where all constraints are satisfied.

Let us now assume that there exists a valid schedule $\mathcal{S}$ of period T. For simplicity, we assume that $\mathcal{S}$ is 1-periodic; however all the arguments can be generalized to a k-periodic schedule. We first prove that operation $F_{4n+1}, \ldots, F_{6n}$ are scheduled as depicted in Figure 5.2. Since $\textsc{nca}$ values are at least 1 and the memory capacity is n, the pattern must satisfy $\textsc{nca}_\ell = 1$ for these layers. Denote by h the shift of F_{6n}; it is easy to see that it is best for all B_ℓ operations (whose durations are negligible) to be performed just after F_{6n} with the same shift h. Hence, the only way to obtain $\textsc{nca}_\ell = 1$ for $\ell \geq 4n + 1$ is to process F_ℓ just before $F_{\ell+1}$ with the same shift value h. Since $\mathcal{S}$ has period $T = 2nV$, there can be no idle time between these operations. Therefore, operations $F_{4n+1}, \ldots, F_{6n}$ are scheduled as depicted in Figure 5.2.

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

5.2 General Periodic Schedules for Contiguous Allocations

In this section, we analyze in more details the scheduling aspect of our problem. In the following, we thus consider that the allocation $\mathcal{A}_\mathcal{C}$ is fixed and contiguous and scheduling is done at the stage level (sets of consecutive layers), as scheduling inside stages is straightforward, which is equivalent to the special case of a network of length P to be processed on P processors. Hence, we focus here on problem $\textsc{Pipe}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}_\mathcal{C}, M, P)$. We present two results in this context: we first show how to compute, for a given period T, a

1-periodic pattern $\mathcal{S}^1$ which minimizes the memory usage, which gives an insight on how to solve problem $\textsc{Pipe}_{\mathcal{S}^1|\mathcal{A}}(\mathcal{A}_\mathcal{C}, M, P)$ optimally; then we provide examples showing the benefit of using k-periodic schedules $\mathcal{S}^k$ for $k > 1$.

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

5.2.1 Optimal 1-periodic Schedule

The authors of [78] propose a 1F1B schedule, which is a particular case of greedy, 1-periodic schedule $\mathcal{S}^1$ where the number of concurrent activations of the first stage is $\textsc{nca}_1 = P$. For unbalanced allocations, especially when taking communications into account, this can be too conservative. To reduce the number of concurrent activations, we propose the 1F1B$^*(T)$ algorithm to compute a pattern for some fixed contiguous allocation $\mathcal{P}$ and a given period T. This algorithm works in three phases, described in Algorithm 6.

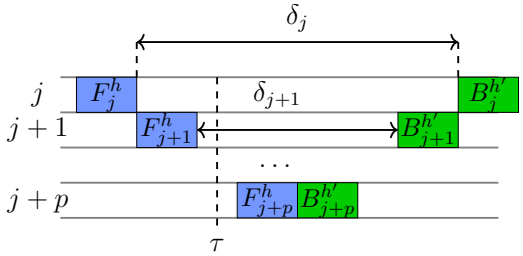
Algorithm 6 Summary of Algorithm 1F1B* for a given period T.

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

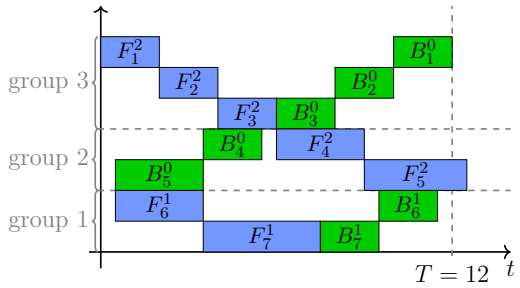
Phase 1: Groups are built such that each group g satisfies the condition $\sum_{s\in g} U(s) \leq T$. This is done iteratively: start from the last stage s_P, add stages $s_{P-1}, s_{P-2}, \ldots$ as long as the condition is fulfilled, then start a new group with the last stage that was not added. This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

Definition 12 (Equal Shift Pattern)**.** An Equal Shift Pattern (V-shape) is a part of a schedule in which consecutive forward operations are performed one after the other on their respective processors with the same shift h, followed by the sequence of corresponding backward operations, all having the same index shift h'. Between the forward operation and the corresponding backward, each processor remains idle (as in Figure 5.3a).

Phase 3: All these group schedules are then connected: to connect group $g = (s_i, \ldots, s_j)$ and group $g - 1 = (s_{j+1}, \ldots, s_k)$, the schedule starts $F_{s_{j+1}}$ just after F_{s_j}, with the same index shift. After this connection, if any operation starts later than T, its starting time is lowered by T and its index shift is decreased by 1. For example, in Figure 5.3b, group 2 and group 3 are connected in such a way that all forwards start before T, therefore they have the same index shift, but the backwards of group 2, which should have index shift 1 (according to Phase 2), are scheduled after the pattern limit T, thus, in order to fit them correctly in the pattern, their starting times are lowered by T and their index shifts



(a) Phase 2: Equal Shift Pattern



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

Рис. 5.3: Scheduling groups in 1F1B*

become 0. In contrast, when connecting group 2 and 1, forwards of group 1 cannot be scheduled just after forwards of group 2 inside the pattern, therefore, their starting times are decreased by T and their index shifts are set to 1, while backwards are scheduled according to Phase 2, without going beyond the pattern limit.

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

Lemma 9. *Consider any schedule $\mathcal{S}$ for a contiguous allocation $\mathcal{A}_\mathcal{C}$. If successive stages verify $\text{NCA}_{s_j} = \cdots = \text{NCA}_{s_{j+p}}$, then $\mathcal{S}$ contains Equal Shift Pattern for these stages.*

Доказательство. Since $\mathcal{S}$ fulfills the dependencies described in Figure 4.7, the following holds for any stage s

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

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

Theorem 15. *Consider a contiguous allocation $\mathcal{A}_\mathcal{C}$ and any 1-periodic schedule $\mathcal{S}^1$ of period T. For any layer ℓ, the schedule $\mathcal{S}^1$ does not use fewer concurrent activations than the schedule 1F1B*(T), i.e. $\forall \ell$, $\text{NCA}_\ell^{1F1B^*} \leq \text{NCA}_\ell^{\mathcal{S}^1}$.*

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

Доказательство. It is easy to see that in 1F1B*, a layer ℓ of group g has $\text{NCA}_\ell^{\text{1F1B*}} = g$. Assume that in $\mathcal{S}^1$, $\text{NCA}_{s_j} = \text{NCA}_{s_{j+1}} = \cdots = \text{NCA}_{s_{j+p}}$ for some j and p. By Lemma 9, there is an Equal Shift Pattern for stages s_j to s_{j+p}, so if we denote by δ_j the delay between F_{s_j} and the next B_{s_j} (see Figure 5.3a), we have $\delta_j \geq \delta_{j+1} + U(s_{j+1})$, and recursively, $\delta_j \geq \sum_{k=j+1}^{j+p} U(s_k)$. Since the period T is the time between two executions of F_{s_j} in $\mathcal{S}^1$, it is clear that $T \geq U(s_j) + \delta_j$, which yields: if $\text{NCA}_{s_j} = \cdots = \text{NCA}_{s_{j+p}}$, then $T \geq \sum_{k=j}^{j+p} U(s_k)$. By contradiction, assume now that for some stage s_i, the schedule $\mathcal{S}^1$ uses fewer concurrent activations than the 1F1B* schedule, *i.e.* $\text{NCA}_{s_i} < g_i$, where g_i is the group number of stage s_i, and consider the largest such index i (for larger indices $j > i$, we thus have $\text{NCA}_{s_j} = g_j$). Denote by $s_{i+1}, \ldots, s_{i+p}$ the group of stage s_{i+1}, so that $\text{NCA}_{s_i} = \text{NCA}_{s_{i+1}} = \cdots = \text{NCA}_{s_{i+p}} = g_{i+1} < g_i$. By the previous result, $T \geq \sum_{k=i}^{i+p} U(s_k)$. However, according to the 1F1B* procedure, $g_i > g_{i+1}$ means that stage s_i could not fit in the group of s_{i+1}, which can only happen if $T < \sum_{k=i}^{i+p} U(s_k)$. This results in a contradiction and completes the proof. □

For a fixed partitioning, all other memory requirements are constant and do not depend on the schedule, so that 1F1B* schedule is optimal with respect to memory usage among all valid 1-periodic patterns $\mathcal{S}^1$. Note that in case when each group consists of only one stage, 1F1B* behaves as 1F1B schedule used in [78].

As it was mentioned before, this result also holds true when taking communications into account: we can consider each communication as if it was a computation layer: the communication between stage s_i on processor p and s_{i+1} on processor p' involves sending some activation a_ℓ between F_ℓ, $\ell \in s_i$ on p and $F_{\ell+1}$ $\ell + 1 \in s_{i+1}$ on p', and a gradient δ_ℓ between $B_{\ell+1}$ on p' and B_ℓ on p, for a total time of $\frac{a_\ell + \delta_\ell}{\beta}$, with the same dependencies as for a normal computation layer. Therefore, we can transform a partitioning on P resources with communication costs into a partitioning on $2P-1$ resources, without communications costs, and apply the 1F1B* algorithm on this transformed partitioning.

Finally, Theorem 15 demonstrated that 1F1B* minimizes memory consumption for a fixed period T. Moreover, it is clear from the description of Algorithm 6 that if T increases then the number of groups G decreases, hence NCA_ℓ for any ℓ decreases as well. Therefore, we can obtain the optimal schedule with a period T^* for $\text{PIPE}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$ with the help of binary search (see Algorithm 7).

5.2.2 k-periodic Schedules

Theorem 16. *$\forall k$, k-periodic schedules are sometimes necessary to reach optimal throughput, i.e. there are examples where no j-periodic schedule with $j < k$ is able to provide the same throughput as a k-periodic schedule.*

Доказательство. For a given k, let us consider an instance where $P = L+1 = k+1$, and $M = k+1$. All layers ℓ have the same durations[2] $u_{F_\ell} = u_{B_\ell} = 1$ and activation sizes $a_\ell = 1$, and different weights: $W_1 = 1$, $W_{\ell+1} = \ell$ for $\ell \geq 1$. For such an instance, the memory

[2]Our arguments actually apply to any homogeneous case where $u_{F_\ell} + u_{B_\ell}$ is constant over all layers ℓ.

Algorithm 7 Optimal solution of $\textsc{Pipe}_{\mathcal{S}|\mathcal{A}}(\mathcal{A}, M, P)$

Require: K (number of iterations)
1: $\text{lb} \leftarrow U(1, L)/P$
2: $\text{ub} \leftarrow U(1, L)$
3: **for** $i = 1, \ldots, K$ **do**
4: $\quad T_i \leftarrow (\text{lb} + \text{ub})/2$
5: $\quad$ Find peak memory consumption $M_{peak} = \max_{p \leq P} M^{\text{1F1B}^*(T_i)}(p)$
6: $\quad$ **if** $M_{peak} > \text{M}$ **then**
7: $\quad\quad \text{lb} \leftarrow T_i$
8: $\quad$ **else**
9: $\quad\quad \text{ub} \leftarrow T_i$
return $\text{1F1B}^*(\text{ub})$

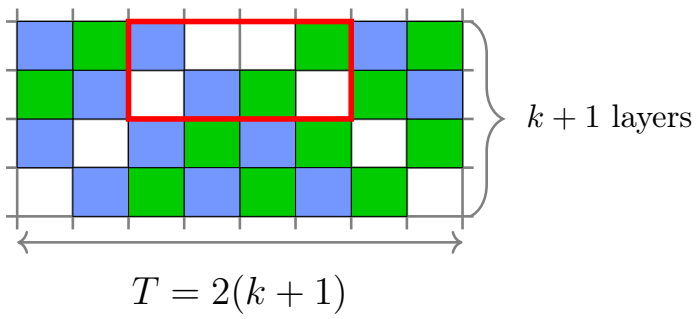


Рис. 5.4: k-periodic pattern for an homogeneous instance. The Equal Shift Pattern is highlighted in thick red.

constraints imply that any valid schedule should satisfy $\text{NCA}_1 \leq k$, and $\text{NCA}_{\ell+1} \leq k+1-\ell$ for $1 \leq \ell \leq k$.

Let us consider the k-periodic schedule $\mathcal{S}^k$ obtained by unrolling the standard 1-periodic pattern with no idle times, and removing all operations related to every $(k+1)$-th mini-batch. The resulting pattern has period $2(k+1)$ and normalized period $2(1+\frac{1}{k})$. It is depicted in Figure 5.4 for $k = 3$. The highlighted Equal Shift Pattern shows how this pattern ensures $\text{NCA}_1 = k$. On the other hand, consider any j-periodic schedule $\mathcal{S}^j$ which satisfies the memory constraints. Since $\text{NCA}_1^{\mathcal{S}^j} \leq k$, $\text{NCA}_\ell \geq 1$ for all ℓ, and since NCA_ℓ values are non-increasing with ℓ, there must exist a layer ℓ such that $\text{NCA}_\ell^{\mathcal{S}^j} = \text{NCA}_{\ell+1}^{\mathcal{S}^j}$. From Lemma 9, there should be a Equal Shift Pattern between these layers, during which layer ℓ is idle for at least $u_F + u_B = 2$ units of time. The period $T^{\mathcal{S}^j}$ of $\mathcal{S}^j$ is thus at least $2j+2$, leading to a normalized period at least $2(1+\frac{1}{j})$. If $j < k$, this is always higher than the normalized period of $\mathcal{S}^k$ described above. □

This example shows the benefit of considering more general schedules than the 1-periodic patterns usually explored in the literature [78]. Furthermore, the simple k-periodic pattern used in this proof can easily be applied to many practical cases where

memory capacity is limited. For example, if we cancel in 1F1B* each $k+1$-th iteration then we can obtain a k-periodic schedule based on 1F1B* with smaller NCA_ℓ for all ℓ from groups $g \geq k+1$. Thus, for a given contiguous allocation with P stages, all such k-periodic patterns for $k < P$ explore a tradeoff between throughput and memory usage: lower values of k have higher normalized period, but lower values of NCA_ℓ.

5.3 Contiguous vs General Allocations

Despite being widely used in practice, contiguous allocations $\mathcal{A}_\mathcal{C}$ can significantly limit the performance. In this section we compare the non-contiguous allocations $\mathcal{A}_{n\mathcal{C}}$ with the contiguous ones $\mathcal{A}_\mathcal{C}$, and we show that in general $\mathcal{A}_{n\mathcal{C}}$ can reach a throughput which can be up to two times larger than the one of $\mathcal{A}_\mathcal{C}$, when memory is not a bottleneck. While under memory constraints, the improvement in the performance can be arbitrarily high. At the same time, as the non-contiguous allocations are more flexible, they could be the only possible option to execute some large models. Unlike the previous section, any resource can now accommodate an arbitrary set of layers (that can be non-consecutive), thus we do not use the notions of stages and groups anymore. Moreover, we talk about processing costs of layers as their combined computing times of forward and backward operations.

5.3.1 Without Memory Constraints

As a simple starting example, let us consider a chain with 3 layers to be processed on 2 processors, where the processing costs of the layers are 1, 2 and 1 respectively. It is clear that the smallest period achieved by a contiguous allocation is 3: the second layer is sharing resource either with the first or the last layers. On the other hand, a non-contiguous allocation allows to run the first and last layers on one processor, and the layer of cost 2 on the other processor, resulting in a period of 2 and no idle time on any processor. The overhead of the contiguous constraint is thus $\frac{3}{2}$ in this case. The following theorem shows that the ratio is asymptotically close to 2 in the worst case.

Theorem 17. *On any chain, the period of the best contiguous allocation $\mathcal{A}_\mathcal{C}$ is at most twice the period of the best non contiguous allocation $\mathcal{A}_{n\mathcal{C}}$. Furthermore, for any $k \geq 1$, there exists a chain for which the period of $\mathcal{A}_\mathcal{C}$ is $2 - \frac{1}{k}$ times larger than $\mathcal{A}_{n\mathcal{C}}$.*

Доказательство. To prove the first result, consider any chain C, and denote by T^* the period of $\mathcal{A}_{n\mathcal{C}}$ for this chain. Clearly $T^* \geq \frac{\sum_l u_{F_\ell} + u_{B_\ell}}{P}$, and $T^* \geq \max_l(u_{F_\ell} + u_{B_\ell})$. We can build a contiguous allocation $\mathcal{A}_\mathcal{C}$ with period at most $2T^*$ with a greedy Next Fit procedure: add layers to the first processor as long as the total load is below $2T^*$, move to the next processor and repeat. Since no layer has cost more than T^*, each processor except maybe the last one has load at least T^*. This shows that this procedure ends before running out of processors.

Let us now prove the second statement, with an example inspired from [12]. For any $k \geq 1$, let us set $\epsilon = \frac{1}{2k+1}$. Let $P = 2k+1$, and let us build the chain C_k with $k+1$ parts:

the first k parts contain 4 layers with computation costs $(k, \epsilon, k-1, \epsilon)$; the last part contains one layer of cost k, $(k-2)(2k+1)+1$ layers of cost ϵ, and one layer of cost 1. Note that the total number of layers of cost ϵ is $2k+(k-2)(2k+1)+1 = (k-1)(2k+1)$.

There exists an allocation $\mathcal{A}_{n\mathcal{C}}$ with period $T^* = k$ for chain C_k: $k+1$ processors process a layer of cost k, 1 processor processes a layer of cost $k-1$ and the layer of cost 1, and $k-1$ processors process a layer of cost $k-1$ and $2k+1$ layers of cost ϵ. In this allocation, no processor has any idle time.

Chain C_k contains $2k+2$ layers with cost at least 1. On any contiguous allocation $\mathcal{A}_{\mathcal{C}}$ on $2k+1$ processors, at least one processor p processes two such layers. If it processes one layer of cost k and one of cost $k-1$, it also processes the layer of cost ϵ between them, and thus its load is at least as $2k-1+\epsilon$. If it processes the layer of cost 1 and the last layer of cost k, it also processes all layers of cost ϵ in between, for a total load at least $k+((k-2)(2k+1)+1)\,\epsilon+1 = 2k-1+\epsilon$. This shows that there is no contiguous allocation with period $2k-1$ or less, which concludes the proof. □

5.3.2 With Memory Constraints

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

Lemma 10. *Non contiguous allocations are sometimes required in order to process training under memory constraints.*

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

Theorem 18. *If there exist both a valid contiguous allocation $\mathcal{A}_{\mathcal{C}}$ and a valid non-contiguous allocation $\mathcal{A}_{n\mathcal{C}}$ given a memory constraint, then the ratio between achieved throughputs of $\mathcal{A}_{n\mathcal{C}}$ and $\mathcal{A}_{\mathcal{C}}$ can be arbitrarily large.*

Доказательство. Let us consider the chain depicted in Figure 5.5. For an arbitrarily chosen k, this chain consists of a sequence of k layers with processing cost 1 and model weight $M-1$, followed by a layer with processing cost k and model weight M and followed by k layers with processing cost $k-1$ and model weight 1. We want to execute this chain on $P = k+2$ resources with memory limit $M \geq k$.

Then, a valid solution consists in grouping, $\forall i \leq P$ layer i and layer $k+i+1$ on processor i, to dedicate processor $k+1$ to layer $k+1$ and to leave processor $k+2$ idle. The required memory $M-1+1$ can fit into the memory and the processing time on each ressource is $k+1$. This yields a non-contiguous allocation $\mathcal{A}_{n\mathcal{C}}$ with a period $T^* = k$.

If we use contiguous allocation $\mathcal{A}_{\mathcal{C}}$, the first $k+1$ layers must be on separate processors, because of the memory contraint. Then, the last k layers must be on the last remaining

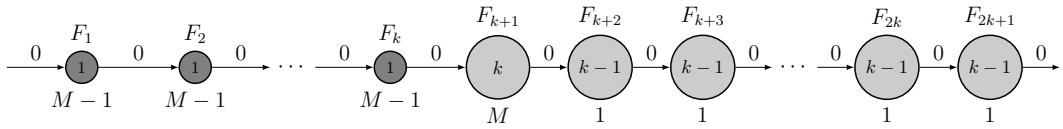


Рис. 5.5: Bad Ratio for Contiguous Allocations and Memory Constraint

processor, that should be feasible due to $M \geq k$. In such scenario, the period is at least as $k(k-1)$, which is $k-1$ times larger than the one of $\mathcal{A}_{n\mathcal{C}}$, which concludes the proof. □

5.4 Conclusion

In this chapter, we consider the possibility of applying model parallelism. It is an attractive parallelization strategy that allows in particular not to replicate all weights of a DNN on all the computation resources. In addition, this parallelism can be combined with other parallel strategies for better scalability. Following the ideas proposed in PipeDream [78] we consider the combination of Pipelining and Model Parallelism, which allows to obtain a better resource utilization.

Nevertheless, the combination of Pipelining and Model Parallelism requires to store more activations at the nodes, which in turn causes memory consumption problems. The practical solutions proposed in the literature rely on a number of hypotheses and limit the search to greedy 1-periodic schedules $\mathcal{S}^1$ and contiguous allocations $\mathcal{A}_{\mathcal{C}}$. On the contrary, we analyze in detail the complexity of the underlying scheduling and resource allocation problems, and prove that these hypotheses prevent, in the general case, to find optimal solutions, which reinforces the interest of the search for more general strategies. Therefore, Chapter 6 and Chapter 7 further analyze the case of non-contiguous allocations and propose the solutions in that context.

Глава 6

Integer Linear Programming Approach

In the previous chapter, we identified the main challenges with respect to Pipelined Model Parallelism. It was shown that in order to solve $\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$ optimally it is important to consider non-contiguous allocations and k-periodic schedules for $k > 1$. However, $\textsc{Pipe}_{\mathcal{S}\&\mathcal{A}}(L, M, P, \beta)$ is NP-complete in the strong sense, thus finding a solution to it is non trivial.

In this chapter, we propose an Integer Linear Programming (ILP) to tackle the above problem. The primary goal is to generate the solutions with non-contiguous allocations $\mathcal{A}_{n\mathcal{C}}$ as they are expected to bring the significant improvement in the memory constrained setting (see Theorem 18). However, representing general k-periodic schedules requires specifying starting times for each operation k times in the pattern, which results in a huge number of variables. Therefore, we restrict our search to 1-periodic schedules $\mathcal{S}^1$ to limit the complexity of the final algorithm. In Section 6.4, we present the main equations that constitute the ILP and we show that its solution corresponds to the optimal solution of $\textsc{Pipe}_{\mathcal{S}^1\&\mathcal{A}}(L, M, P, \beta)$.

6.1 Notations

Let us denote by P the total number of available GPUs and let us assume (as in PipeDream) that all pairs of GPUs are connected through a direct link of capacity β (the devices do not compete for bandwidth). Moreover, we assume that each GPU is equipped with an available memory of size M.

Similarly to Chapter 5, we represent each DNN as a chain so that the task graph corresponding to forward and backward propagation is depicted in Figure 4.7 (we keep the same model to describe DNN training loop and its data dependencies).

To represent operations in the ILP, we consider F_ℓ and B_ℓ as computational tasks T_ℓ, where if $\ell \leq L + 1$ then $T_\ell = F_\ell$ and if $\ell > L + 1$ then $T_\ell = B_{2L-\ell+3}$ (in total there are $2L + 2$ tasks). However, unlike the previous chapter, in order to provide solutions to $\textsc{Pipe}_{\mathcal{S}^1\&\mathcal{A}}(L, M, P, \beta)$ we need to take communications between the devices into account. Thus, we also introduce communication tasks T^c_ℓ. These communication

tasks correspond to sending forward activations or gradients from one computation resource to another, and their cost is 0 between two successive layers allocated to the same computing resource (in total there are $2L + 1$ possible communications). The sequence of tasks associated with the processing of a mini-batch is therefore given by $T_1, T_1^c, T_2 \ldots, T_{L-1}^c, T_L, T_{L+1}, T_{L+1}^c, T_{L+2}, \ldots, T_{2L+1}^c, T_{2L+2}$.

We denote by a_ℓ the activation tensor produced by T_ℓ, $\ell \leq L+1$ and by $a_{2L-\ell+2} = \delta_\ell = \frac{\partial \text{Loss}}{\partial a_\ell}$ the backpropagated intermediate gradient value provided as input of the backward operation $T_{2L-\ell+3} = B_\ell$. Moreover, we define the durations of tasks and the sizes of output data for each respective task:

- a_ℓ the size (in bytes) of the tensor a_ℓ produced by $T_\ell, \ell \leq L + 1$;
- $a_{\ell'}$ the size (in bytes) of the tensor $a_{\ell'} = \delta_{2L-\ell'+2}$ produced by $T_{\ell'}, \ell' > L + 1$, in general, we assume that tensors a_ℓ and δ_ℓ have the same size, *i.e.* $\forall \ell \leq L, a_{2L-\ell+2} = a_\ell$;
- $d_\ell = u_{F_\ell}$ the duration of the forward task on the layer $1 \leq \ell \leq L + 1$;
- $d_{2L-\ell+3} = u_{B_\ell}$ the duration of the backward task on the layer $1 \leq \ell \leq L + 1$;
- $\widetilde{d}_\ell = \frac{a_\ell}{\beta}$ the duration of communication task $1 \leq \ell \leq 2L + 1$;

Each GPU has limited memory of size M. This memory is used to store all data required to perform the training operation. More precisely, these memory requirements can have different origins:

- Model weights. Since we are considering Model Parallelism, we assume that the $L + 1$ layers of the network are split across the P GPUs. If processor k is in charge of layer ℓ, then it has to store the corresponding weight denoted as W_ℓ. As we will see, in order to update the weights and to use consistent weights during both the forward and the backward phases, processor k in practice stores several copies C of the weights. In what follows, in order not to add more weight and activation copies, we assume that the processor in charge of a layer is in charge of processing both the forward and backward tasks associated to this layer.
- Activations. Let us now concentrate on activations. As depicted in Figure 4.7, activation $a_{\ell-1}$ (the input of task T_ℓ) must be kept in memory until task $T_{2L-\ell+3}$ consumes it to produce $\delta_{\ell-1}$. Therefore, a memory of size $a_{\ell-1}$ must be reserved to store an activation between tasks T_ℓ and task $T_{2L-\ell+3}$. As it was discussed in the previous chapter, due to pipelining each processor may keep several copies of $a_{\ell-1}$ at the same time, *i.e.* $\text{NCA}_\ell \geq 1$ for all ℓ.
- Gradients. Let us now concentrate on gradients δ_ℓ. As depicted in Figure 4.7, the gradient δ_ℓ produced by task $T_{2L-\ell+2}$ is consumed by $T_{2L-\ell+3}$ to produce $\delta_{\ell-1}$. Therefore, a memory of size $a_{2L-\ell+2}(= a_\ell)$ must be reserved to store an activation between tasks $T_{2L-\ell+2}$ and $T_{2L-\ell+3}$ to produce $\delta_{\ell-1}$. Thus, gradients are kept in memory for a much shorter time than activations.
- Communication buffers. When tasks T_ℓ and $T_{\ell+1}$ are assigned to different GPUs, a communication takes place, and we assume for convenience that some memory is reserved as a buffer to store a_ℓ while it is sent or received. This buffer is permanently present throughout the entire execution and requires a storage of size a_ℓ on both GPUs.

In this chapter, we search for optimal solutions with non-contiguous allocations $\mathcal{A}_{nc}$ and 1-periodic schedules $\mathcal{S}^1$ discussed in the previous chapter. The example of possible solutions is shown on Figure 6.1.

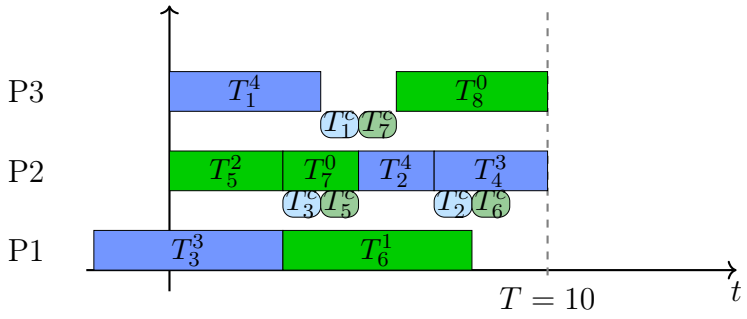


Рис. 6.1: Example of valid pattern. The index shifts are in the superscipts near task names

6.2 Communication and Computation Constraints

We present in this section an Integer Linear Program to find a valid pattern with minimum period length. We concentrate on scheduling issues on both computational and communication resources in Section 6.2, then we consider memory related issues in Section 6.3.

We first present the main variables used in this ILP (other variables are introduced later):

- T denotes the period considered in the 1-periodic schedule $\mathcal{S}^1$;
- $z_{\ell,\ell'}$ is equal to 1 if task T_ℓ and task $T_{\ell'}$ are processed on the same resource, and 0 otherwise (it is implied that $z_{\ell,\ell} = 1$ and $z_{\ell,\ell'} = z_{\ell',\ell}$);
- τ_ℓ is the starting time of task T_ℓ in the pattern ;
- $\widetilde{\tau}_\ell$ is the starting time in the considered period of the communication of the output of T_ℓ.

In several places, we use a large constant K that needs to be larger than the period, for example, we can use $K = \sum_\ell d_\ell + \sum_{\ell'} \widetilde{d}_{\ell'}$, which corresponds to the worst possible period.

6.2.1 Limit on the Number of Resources

In order to provide a limit on the number of resources used, we introduce variable f_ℓ that is equal to 1 if and only if task T_ℓ is the lowest-index task processed on its resource. To

this end, we consider the following set of constraints:

$$\forall \ell < \ell' < \ell'', \quad z_{\ell',\ell''} \geq z_{\ell,\ell'} + z_{\ell,\ell''} - 1 \tag{6.1}$$

$$\forall \ell, \quad f_\ell \geq 1 - \sum_{\ell' < \ell} z_{\ell,\ell'} \tag{6.2}$$

$$\sum_\ell f_\ell \leq P, \tag{6.3}$$

and we show that they are enough to obtain the following:

Lemma 11. *Constraints (6.1)-(6.3) ensure that at most P resources are used in the schedule.*

Доказательство. Constraint (6.1) ensures the consistency of the $z_{\ell,\ell'}$ variables: for any ℓ, ℓ', ℓ'', if $z_{\ell,\ell} = 1$ and $z_{\ell,\ell''} = 1$, then $z_{\ell',\ell''} = 1$. They can thus be used to define an equivalence relation between tasks, where each class contains tasks that are processed on the same resource. Then, Constraint (6.2) ensures that f_ℓ is exactly 1 for the task with the smallest index among all the tasks processed on a given resource, and is trivially satisfied for all other tasks. Therefore, the sum of f_ℓ provides the total number of allocated resources, and constraint (6.3) enforces that no more than P resources are used in the schedule. Reciprocally, in any valid solution that uses no more than P resources, there exists an assignment of f_ℓ variables such that $\sum_\ell f_\ell \leq P$, *i.e.* the assignment where $f_\ell = 1$ for the task with the smallest index processed on the resource and 0 for all other tasks. □

In addition, we consider only schedules where forward tasks $T_{L+2-\ell}$ for $\ell \leq L+1$ are placed on the same resource as their respective backward tasks $T_{L+\ell+1}$, so we add the following equation to the Linear Program

$$\forall \ell \leq L+1, \quad z_{L+2-\ell,L+\ell+1} = 1. \tag{6.4}$$

6.2.2 Ordering of Computational Tasks

Let us now consider tasks that are processed on the same resource. In order to enforce that two tasks processed on the same resource cannot overlap, we introduce a set of variables $w_{\ell,\ell'}$ and the following equations, valid for all $\ell \neq \ell'$:

$$\tau_\ell - \tau_{\ell'} + K(1 - z_{\ell,\ell'} + w_{\ell,\ell'}) \geq d_{\ell'}, \tag{6.5}$$

$$\tau_{\ell'} - \tau_\ell + K(2 - z_{\ell,\ell'} - w_{\ell,\ell'}) \geq d_\ell, \tag{6.6}$$

$$w_{\ell,\ell'} \leq z_{\ell,\ell'}. \tag{6.7}$$

As explained above, we use K throughout the proofs, in order to define a condition that should be valid as soon as a boolean variable x is equal to 1. The general idea is to use $(1-x)K$ in the equation as follows: if $x = 1$, then $(1-x)K = 0$ and the rest of the condition must be satisfied. Conversely, if $x = 0$ then $(1-x)K$ is significantly larger than the other terms and the condition is automatically satisfied, regardless of the value of the other variables. An example of the use of this technique can be found below in Lemma 12.

Lemma 12. *Constraints* (6.5)-(6.7) *ensure the following:*

- *If T_ℓ and $T_{\ell'}$ are assigned to the same resource, then either $T_{\ell'}$ starts after the end of T_ℓ (and $w_{\ell,\ell'} = 1$), or T_ℓ starts after the end of $T_{\ell'}$ (and $w_{\ell,\ell'} = 0$).*
- *If T_ℓ and $T_{\ell'}$ are not assigned to the same resource, then $w_{\ell,\ell'} = 0$.*

Доказательство. Let us assume that T_ℓ and $T_{\ell'}$ are assigned to the same resource. Then, by definition, $z_{\ell,\ell'} = 1$ and we obtain

$$\begin{aligned} &\tau_\ell - \tau_{\ell'} + K w_{\ell,\ell'} \geq d_{\ell'}, \\ &\tau_{\ell'} - \tau_\ell + K(1 - w_{\ell,\ell'}) \geq d_\ell \\ \text{and } &w_{\ell,\ell'} \leq 1. \end{aligned}$$

In turn, $w_{\ell,\ell'} = 1$ implies $\tau_\ell - \tau_{\ell'} + K \geq d_{\ell'}$ and $\tau_{\ell'} \geq \tau_\ell + d_\ell$. The first constraint is always true since K is large and the second constraint implies that $T_{\ell'}$ starts after the end of T_ℓ. The proof for $w_{\ell,\ell'} = 0$ is symmetric and is omitted here.

Then, let us assume that T_ℓ and $T_{\ell'}$ are not assigned to the same resource. Then, by definition, $z_{\ell,\ell'} = 0$ and we obtain

$$\begin{aligned} &\tau_\ell - \tau_{\ell'} + K(1 + w_{\ell,\ell'}) \geq d_{\ell'}, \\ &\tau_{\ell'} - \tau_\ell + K(2 - w_{\ell,\ell'}) \geq d_\ell \\ \text{and } &w_{\ell,\ell'} \leq 0. \end{aligned}$$

These three constraints are compatible since the first two are always true independently of the value of $w_{\ell,\ell'}$, by definition of K, and the last one enforces $w_{\ell,\ell'} = 0$. □

6.2.3 Ordering of Communication Tasks

If tasks T_ℓ and $T_{\ell+1}$ are not processed on the same resource, a communication should take place for the output of T_ℓ. We recall that a_ℓ denotes the data (either an activation or a gradient) computed by T_ℓ and needed by $T_{\ell+1}$. In order to schedule these communications, we define a new set of variables $\widetilde{z}_{\ell,\ell'}$, which play an analogous role to $z_{\ell,\ell'}$ for communication tasks. More precisely, we want $\widetilde{z}_{\ell,\ell'} = 1$ if the communications of a_ℓ and $a_{\ell'}$ share the same communication link, and $\widetilde{z}_{\ell,\ell'} = 0$ otherwise. We prove that this property is enforced by the following equations, for all $\ell \neq \ell'$.

Lemma 13. *The following equations define $\widetilde{z}_{\ell,\ell'}$ through $z_{\ell,\ell'}$, so that $\widetilde{z}_{\ell,\ell'} = 1$ if and only if T^c_ℓ and $T^c_{\ell'}$ occupy the same communication link and, therefore, there exist exactly two distinct processors that execute T_ℓ, $T_{\ell+1}$, $T_{\ell'}$ and $T_{\ell'+1}$, so that tasks T_ℓ and $T_{\ell+1}$ are placed on the different resources and the same for $T_{\ell'}$ and $T_{\ell'+1}$:*

$$\widetilde{z}_{\ell,\ell'} \geq z_{\ell,\ell'} + z_{\ell+1,\ell'+1} - z_{\ell,\ell+1} - 1 \tag{6.8}$$

$$\widetilde{z}_{\ell,\ell'} \geq z_{\ell,\ell'+1} + z_{\ell+1,\ell'} - z_{\ell,\ell+1} - 1, \tag{6.9}$$

$$\widetilde{z}_{\ell,\ell'} \leq 1 - z_{\ell,\ell+1} \tag{6.10}$$

$$\widetilde{z}_{\ell,\ell'} \leq 1 - z_{\ell',\ell'+1} \tag{6.11}$$

$$\widetilde{z}_{\ell,\ell'} \leq z_{\ell,\ell'} + z_{\ell+1,\ell'} \tag{6.12}$$

$$\widetilde{z}_{\ell,\ell'} \leq z_{\ell,\ell'+1} + z_{\ell+1,\ell'+1} \tag{6.13}$$

Доказательство. Constraints (6.10) and (6.11) ensure that $\widetilde{z}_{\ell,\ell'} = 0$ if any of a_ℓ or $a_{\ell'}$ does not require a communication (because the corresponding tasks are processed on the same resource). In the following, we assume that both $z_{\ell,\ell+1}$ and $z_{\ell',\ell'+1}$ are 0 (we need to communicate both a_ℓ and $a_{\ell'}$), and we denote by P_i the processor that runs T_ℓ and by P_j the processor that runs $T_{\ell+1}$. We consider several cases:

First case: a_ℓ and $a_{\ell'}$ share the same communication link.

In that case, $T_{\ell'}$ must be processed either on P_i or P_j, otherwise the communication of $a_{\ell'}$ occupies another link than (P_i, P_j). Therefore, constraint (6.12) simply becomes $\widetilde{z}_{\ell,\ell'} \leq 1$. Similarly, $T_{\ell'+1}$ must be processed either on P_i or on P_j, so that constraint (6.13) simply becomes $\widetilde{z}_{\ell,\ell'} \leq 1$.

Since $z_{\ell,\ell+1} = 0$, constraints (6.8) and (6.9) become

$$\widetilde{z}_{\ell,\ell'} \geq z_{\ell,\ell'} + z_{\ell+1,\ell'+1} - 1,$$
$$\widetilde{z}_{\ell,\ell'} \geq z_{\ell,\ell'+1} + z_{\ell+1,\ell'} - 1.$$

Since the communication of $a_{\ell'}$ use the link between P_i and P_j, $T_{\ell'}$ and $T_{\ell'+1}$ must be processed on these processors. Hence, either $T_{\ell'}$ is on P_i (and $T_{\ell'+1}$ is on P_j) or $T_{\ell'}$ is on P_j (and $T_{\ell'+1}$ is on P_i) and therefore, one of the conditions above enforces $\widetilde{z}_{\ell,\ell'} = 1$.

Second case: a_ℓ and $a_{\ell'}$ do not share the same communication link.

Let us first focus on constraint (6.8). We claim that both $z_{\ell,\ell'}$ and $z_{\ell+1,\ell'+1}$ can not be 1 at the same time. Indeed, this would imply that T_ℓ and $T_{\ell'}$ are processed on P_i, and also that $T_{\ell+1}$ and $T_{\ell'+1}$ are processed on P_j, and thus that a_ℓ and $a_{\ell'}$ share the same communication link. Since $z_{\ell,\ell+1} = 0$, the right hand side of constraint (6.8) is at most 0. Using the same analysis for constraint (6.9), we prove that the first two constraints simply become $\widetilde{z}_{\ell,\ell'} \geq 0$.

We now prove by contradiction that at least one among constraints (6.12) and (6.13) enforces $\widetilde{z}_{\ell,\ell'} = 0$. Indeed, $z_{\ell,\ell'} + z_{\ell+1,\ell'} \geq 1$ implies that $T_{\ell'}$ is processed either on P_i or P_j, and similarly $z_{\ell,\ell'+1} + z_{\ell+1,\ell'+1} \geq 1$ implies that $T_{\ell'+1}$ is processed either on P_i or P_j. Since we assume that $z_{\ell',\ell'+1} = 0$, having both $z_{\ell,\ell'} + z_{\ell+1,\ell'} \geq 1$ and $z_{\ell,\ell'+1} + z_{\ell+1,\ell'+1} \geq 1$ is in contradiction with the fact that a_ℓ and $a_{\ell'}$ do not use the same link.

Therefore, in this second case, the system of constraints enforces $\widetilde{z}_{\ell,\ell'} = 0$. □

Finally, we can ensure a correct ordering of the communications without overlap in a way similar to Lemma 12. We introduce binary variables $\widetilde{w}_{\ell,\ell'}$ together with the following equations, for all $\ell \neq \ell'$:

$$\widetilde{\tau}_\ell - \widetilde{\tau}_{\ell'} + K(1 - \widetilde{z}_{\ell,\ell'} + \widetilde{w}_{\ell,\ell'}) \geq \widetilde{d}_{\ell'} \tag{6.14}$$
$$\widetilde{\tau}_{\ell'} - \widetilde{\tau}_\ell + K(2 - \widetilde{z}_{\ell,\ell'} - \widetilde{w}_{\ell,\ell'}) \geq \widetilde{d}_\ell \tag{6.15}$$
$$\widetilde{w}_{\ell,\ell'} \leq \widetilde{z}_{\ell,\ell'} \tag{6.16}$$

Lemma 14. *Constraints (6.8)-(6.16) ensure that:*

- $\widetilde{z}_{\ell,\ell'} = 1$ *if and only if both* a_ℓ *and* $a_{\ell'}$ *need to be communicated and their communications are assigned to the same link.*

- *In that case, either the communication of $a_{\ell'}$ starts after the end of the communication of a_ℓ (and $\widetilde{w}_{\ell,\ell'} = 1$), or the communication of a_ℓ starts after the end of the communication of $a_{\ell'}$ (and $\widetilde{w}_{\ell,\ell'} = 0$).*
- *In the opposite case, $\widetilde{w}_{\ell,\ell'} = 0$.*

Доказательство. The proof is similar to the proof of Lemma 12, replacing $w_{\ell,\ell'}$ by $\widetilde{w}_{\ell,\ell'}$ and $z_{\ell,\ell'}$ by $\widetilde{z}_{\ell,\ell'}$, and is therefore omitted here. □

6.2.4 Period Length

In order to obtain a valid pattern from the variables defined so far, we use without loss of generality the following conventions: the ending times of all tasks and communications are between 0 and T, and task T_1 starts at time 0:

$$\forall \ell, \quad 0 \leq \tau_\ell + d_\ell \leq T \tag{6.17}$$

$$\forall \ell, \quad 0 \leq \widetilde{\tau}_\ell + \widetilde{d}_\ell \leq T \tag{6.18}$$

$$\tau_1 = 0 \tag{6.19}$$

We cannot specify that all *starting* times should be non-negative: as can be seen on Figure 6.1, in general the patterns on different processors are not aligned to start at the same time. So in order to ensure that each resource is occupied for a duration at most T, we include the following constraints that state that the distance between the ending time and starting time of two tasks assigned to the same resource is at most T:

$$\forall \ell \neq \ell', \quad T \geq \tau_\ell + d_\ell - \tau_{\ell'} - K(1 - z_{\ell,\ell'}) \tag{6.20}$$

$$\forall \ell \neq \ell', \quad T \geq \widetilde{\tau}_\ell + \widetilde{d}_\ell - \widetilde{\tau}_{\ell'} - K(1 - \widetilde{z}_{\ell,\ell'}) \tag{6.21}$$

Lemma 15. *Without considering memory constraints, from any valid 1-periodic pattern $\mathcal{S}^1$, we can obtain values for all variables T, τ_ℓ, $z_{\ell,\ell'}$, f_ℓ, $\widetilde{\tau}_\ell$, $\widetilde{z}_{\ell,\ell'}$, $w_{\ell,'}$ and $\widetilde{w}_{\ell,\ell'}$ that respect equations (6.1)-(6.21), and vice-versa.*

Доказательство. If all variables fulfill the constraints (6.1)-(6.21), then Lemmas 11, 12 and 14 ensure that the pattern built from the values of τ_ℓ and $z_{\ell,\ell'}$ is a valid pattern. Furthermore, constraint (6.20) ensures that for any ℓ and ℓ' such that $z_{\ell,\ell'} = 1$, T is no smaller than $\tau_\ell + d_\ell - \tau_{\ell'}$ and $\tau_{\ell'} + d_{\ell'} - \tau_\ell$, depending on which task starts first. Since a forward task is always allocated to the same resource as the respective backward task (Constraint (6.4)), all used resources process at least two tasks. The same can be said for communication tasks, which ensures that T is a valid period for the constructed pattern.

Reciprocally, let us consider any valid pattern, and assign values to all the variables according to this pattern. As discussed above, this can be done in a way that respects constraints (6.17)-(6.19) without loss of generality. The above lemmas ensure that constraints (6.1)-(6.16) are satisfied. Since T is a valid period, constraints (6.20), (6.21) and (6.18) are satisfied for any ℓ and ℓ' such that $z_{\ell,\ell'} = 1$. Finally, if $z_{\ell,\ell'} = 0$, these constraints are automatically satisfied since K is large. □

6.3 Memory Constraints

In this section, we focus on the memory usage induced by the pattern described in previous section. The memory needs have different origins:

- If two successive tasks T_ℓ and $T_{\ell+1}$ are processed on the same resource, the output of T_ℓ needs to be stored in memory until it is processed by $T_{\ell+1}$. This is addressed in Section 6.3.1.
- The main point of pipelining is that during one period, the forward task $T_{L-\ell+2}$ and its associated backward task $T_{L+\ell+1}$ for $1 \geq \ell \geq L+1$ do not operate on the same mini-batch. This implies that the processor in charge of these operations must store several activations produced by the forward task and not yet consumed by the corresponding backward task. This will be addressed in Section 6.3.2.
- Processors need to store the weights of the layers that they process. This will be addressed in Section 6.3.3.
- When $T_{\ell'-1}$ and $T_{\ell'}$ are not processed on the same resource, $a_{\ell'-1}$ is received by the resource in charge of $T_{\ell'}$ and is kept in memory until the next $T_{\ell'}$ is performed. Similarly, when $T_{\ell'}$ and $T_{\ell'+1}$ are not processed on the same resource, $a_{\ell'}$ must be sent by the resource in charge of $T_{\ell'}$ and is kept in memory until the end of associated communication. This will be addressed in Section 6.3.4.

In order to avoid symmetries in the formulation of the Integer Linear Program, we provide a formulation based on tasks and task collocations rather than on processing resources. We therefore compute, for each task T_ℓ, the amount of memory required *at the instant when T_ℓ is performed* respectively by the storage of models $\mathrm{M}_\ell^{(\mathrm{MOD})}$, by direct dependencies $\mathrm{M}_\ell^{(\mathrm{DIR})}$, by local activations $\mathrm{M}_\ell^{(\mathrm{ACT})}$ and by external activations and gradients $\mathrm{M}_\ell^{(\mathrm{EXT})}$.

6.3.1 Memory for Direct Dependencies

As depicted on Figure 6.1, the output $a_{\ell'}$ of a forward task $T_{\ell'}$ is used twice: first by the next forward task $T_{\ell'+1}$, then by the corresponding backward task $T_{2L-\ell'+2}$. In this section, we account for the memory consumption of $a_{\ell'}$ from $T_{\ell'}$ until $T_{\ell'+1}$; the memory consumption until the backward task will be accounted for in Section 6.3.2.

To evaluate $\mathrm{M}_\ell^{(\mathrm{DIR})}$, let us assume that tasks ℓ, ℓ' and $\ell'+1$ are processed on the same resource. We are interested in the following event: the output produced by $T_{\ell'}$ occupies the memory of the resource when task T_ℓ is performed. This event occurs in three possible situations (see Figure 6.2):

- $T_{\ell'}$ is processed before T_ℓ, and T_ℓ before $T_{\ell'+1}$;
- T_ℓ is processed before $T_{\ell'+1}$, and $T_{\ell'+1}$ before $T_{\ell'}$;
- $T_{\ell'+1}$ is processed before $T_{\ell'}$, and $T_{\ell'}$ before T_ℓ.

We therefore need to consider three variables: $w_{\ell',\ell}$, $w_{\ell,\ell'+1}$ and $w_{\ell'+1,\ell'}$. Obviously, all three variables cannot be equal to one within a pattern, and the list above shows that the event occurs if and only if exactly two of these variables are equal to one. We thus introduce a binary variable $o_{\ell,\ell'}$ for all ℓ and ℓ' with $\ell \neq \ell'$ and $\ell \neq \ell'+1$, with the following

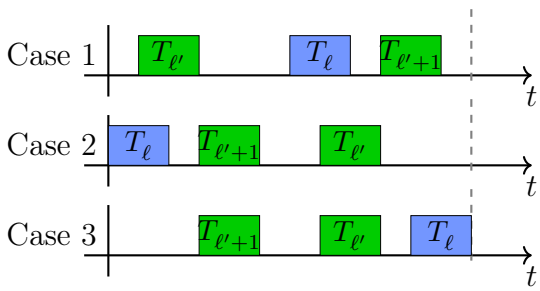


Рис. 6.2: Different Cases for direct dependencies, where T_ℓ, $T_{\ell'}$ and $T_{\ell'+1}$ are on the same processor.

constraint:

$$o_{\ell,\ell'} \geq w_{\ell',\ell} + w_{\ell,\ell'+1} + w_{\ell'+1,\ell'} - 1 \tag{6.22}$$

We obtain the following lemma:

Lemma 16. *Consider any valid pattern according to Lemma 15, and assume that variables $o_{\ell,\ell'}$ satisfy Constraint (6.22).*

If the output produced by $T_{\ell'}$ is present in memory as a direct dependency when task T_ℓ is performed, then $o_{\ell,\ell'} \geq 1$. The total amount of memory that is occupied by direct dependencies is at most $\mathrm{M}_\ell^{(\mathrm{DIR})} = \sum_{\ell'=1}^{2L+2} o_{\ell,\ell'} a_{\ell'}$.

6.3.2 Memory Required for Local Activations

Let us now consider the memory required by the storage of local activations, between the instant when they are computed by a forward task and the instant when they are consumed by the associated backward task. To achieve this, we need to analyze precisely the shifts in indices of the mini-batches that are processed during the same period.

We observe that two consecutive tasks in the task graph, either (T_ℓ, T_ℓ^c) or (T_ℓ^c,$T_{\ell+1}$), can operate on the same mini-batch during a given period if they appear in the proper order, *i.e.* T_ℓ before T_ℓ^c or T_ℓ^c before $T_{\ell+1}$. Otherwise, they must operate on different mini-batches. An example showing the path of different mini-batches for a simple case of two processors and no communication is shown on Figure 6.3.

When two successive tasks are too far apart in the pattern, it can even happen (in rare cases) that they have to process mini-batches with an index shift of two. Figure 6.4 shows such a case, which happens if and only if the difference between the end time of T_ℓ and the start time of $T_{\ell+1}$ is more than T.

To evaluate the shifts of indices in the pattern, we introduce new boolean variables associated to task ℓ: v_ℓ and v'_ℓ are used to determine the shift between T_ℓ and T_ℓ^c, where v_ℓ is 1 if the shift is at least 1, and v'_ℓ is 1 if the shift is 2. Variables $\widetilde{v}_\ell$ and $\widetilde{v}'_\ell$ have the same meaning for the shift between T_ℓ^c and $T_{\ell+1}$. For all $1 \leq \ell \leq 2L+2$, we include the

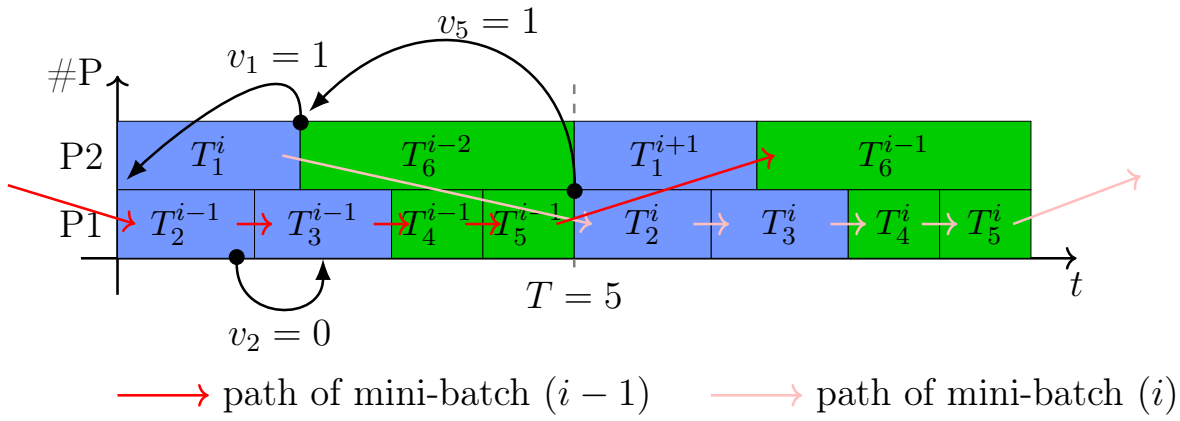


Рис. 6.3: Example schedule (without communications) with paths of different mini-batches. The superscripts indicate the batch indices. Black arrows point from the end of a task T_ℓ to the start of $T_{\ell+1}$, and show the value of the associated v_ℓ variable.

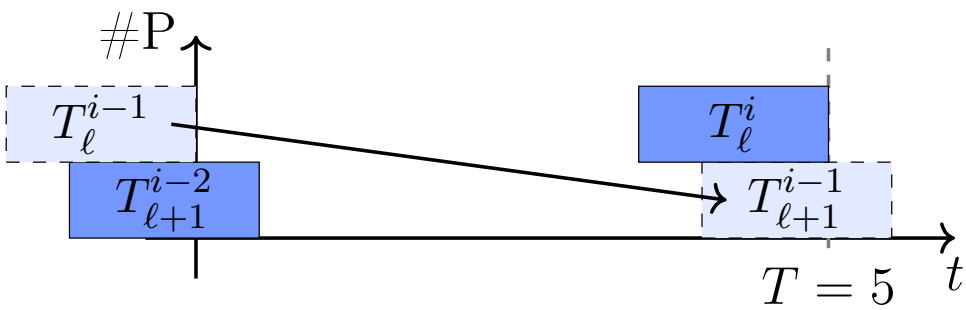


Рис. 6.4: Example schedule where if task T_ℓ processes mini-batch i during the period, $T_{\ell+1}$ needs to process mini-batch $i-2$. Tasks indicated with dashed lines belong to different periods (the previous one for T_ℓ, the next one for $T_{\ell+1}$).

following constraints:

$$\tau_\ell + d_\ell - \widetilde{\tau}_\ell + K(1 - v_\ell) \geq 0 \tag{6.23}$$
$$\widetilde{\tau}_\ell - (\tau_\ell + d_\ell) + Kv_\ell \geq 0 \tag{6.24}$$
$$\widetilde{\tau}_\ell + \widetilde{d}_\ell - \tau_{\ell+1} + K(1 - \widetilde{v}_\ell) \geq 0 \tag{6.25}$$
$$\tau_{\ell+1} - (\widetilde{\tau}_\ell + \widetilde{d}_\ell) + K\widetilde{v}_\ell \geq 0 \tag{6.26}$$
$$\tau_\ell + d_\ell - (\widetilde{\tau}_\ell + T) + 2K(1 - v'_\ell) \geq 0 \tag{6.27}$$
$$\widetilde{\tau}_\ell + T - (\tau_\ell + d_\ell) + 2Kv'_\ell \geq 0 \tag{6.28}$$
$$\widetilde{\tau}_\ell + \widetilde{d}_\ell - (\tau_{\ell+1} + T) + 2K(1 - \widetilde{v}'_\ell) \geq 0 \tag{6.29}$$
$$\tau_{\ell+1} + T - (\widetilde{\tau}_\ell + \widetilde{d}_\ell) + 2K\widetilde{v}'_\ell \geq 0 \tag{6.30}$$

Lemma 17. *Consider any valid pattern according to Lemma 15, and assume that variables v_ℓ, v'_ℓ, $\widetilde{v}_\ell$ and $\widetilde{v}'_\ell$ satisfy Constraints (6.23)-(6.30).*

If i denotes the mini-batch performed by T_{L+1}, then for any $1 \leq \ell \leq L+1$, the index of the mini-batch performed by T_ℓ is at least $i + \sum_{m=\ell}^{L}(v_m + \widetilde{v}_m + v'_m + \widetilde{v}'_m)$, and the index of the mini-batch performed by $T_{2L+3-\ell}$ is at most $i - \sum_{m=L+1}^{2L+2-\ell}(v_m + \widetilde{v}_m + v'_m + \widetilde{v}'_m)$.

Hence, the number of activations of type $a_{\ell-1}$ that needs to be stored at the beginning of the period of this processor is $\sum_{m=\ell}^{2L+2-\ell} v_m + \widetilde{v}_m + v'_m + \widetilde{v}'_m$.

Доказательство. Similarly to Lemma 12, we can show using the definition of K proposed above that constraints (6.23)-(6.26) ensure that for any ℓ, $v_\ell = 1$ if $\widetilde{\tau}_\ell < \tau_\ell + d_\ell$, and $v_\ell = 0$ otherwise; likewise, $\widetilde{v}_\ell = 1$ if $\tau_{\ell+1} < \widetilde{\tau}_\ell + \widetilde{d}_\ell$, and $\widetilde{v}_\ell = 0$ otherwise. Additionally, constraints (6.27)-(6.30) ensure that if $\widetilde{\tau}_\ell + T < \tau_\ell + d_\ell$, then $v'_\ell = 1$, and $v'_\ell = 0$ otherwise; if $\tau_{\ell+1} + T < \widetilde{\tau}_\ell + \widetilde{d}_\ell$, then $\widetilde{v}'_\ell = 1$, and $\widetilde{v}'_\ell = 0$ otherwise.

The claimed result can be proved by induction. For any $1 \leq \ell \leq L+1$, let us assume that the index of the mini-batch performed by $T_{\ell+1}$ is at least $I = i + \sum_{m=\ell+1}^{L} v_m + \widetilde{v}_m + v'_m + \widetilde{v}'_m$.

The ordering of $\widetilde{\tau}_\ell$ and $\tau_{\ell+1}$ can yield three possible cases:

- If $\widetilde{\tau}_\ell + \widetilde{d}_\ell \leq \tau_{\ell+1}$, then $\widetilde{v}_\ell = \widetilde{v}'_\ell = 0$ and both computation $T_{\ell+1}$ and communication T^c_ℓ can process the same mini-batch in the same period: T^c_ℓ can process mini-batch I.
- If $\widetilde{\tau}_\ell + \widetilde{d}_\ell > \tau_{\ell+1}$ and $\widetilde{\tau}_\ell + \widetilde{d}_\ell \leq \tau_{\ell+1} + T$, then $\widetilde{v}_\ell = 1$ and $\widetilde{v}'_\ell = 0$. In this case, similar to the one shown on Figure 6.3 with tasks T_1 and T_2, T^c_ℓ cannot process mini-batch I: if it does, the result arrives too late and task $T_{\ell+1}$ is not able to process mini-batch I. However the T^c_ℓ can process mini-batch $I+1$.
- If $\widetilde{\tau}_\ell + \widetilde{d}_\ell > \tau_{\ell+1} + T$, then $\widetilde{v}_\ell = 1$ and $\widetilde{v}'_\ell = 1$. In that case, similar to the one shown on Figure 6.4, T^c_ℓ can only process mini-batch $I+2$.

Therefore, in all cases the index of the mini-batch processed by T^c_ℓ is $I + \widetilde{v}_\ell + \widetilde{v}'_\ell$.

The same reasoning can be applied to the possible index shift between T_ℓ and T^c_ℓ, this time involving v_ℓ and v'_ℓ. We thus prove that the index of the mini-batch performed by T_ℓ during the current period is at least $I + \widetilde{v}_\ell + \widetilde{v}'_\ell + v_\ell + v'_\ell = i + \sum_{m=\ell}^{L}(v_m + \widetilde{v}_m + v'_m + \widetilde{v}'_m)$, which achieves the proof for forward tasks.

The proof for backward tasks is similar and is omitted here. □

As mentioned above, we are interested in $\mathrm{M}_\ell^{(\mathrm{ACT})}$, which is the memory consumed by the set of activations at the time when task T_ℓ is performed, on the processor that computes T_ℓ. We thus introduce integer variables $\sigma_{\ell,'\ell}$, equal to the number of activations of type $a_{\ell'-1}$ stored on the processor that computes T_ℓ, which satisfy the following constraints:

$$\forall \ell', \ell \quad \sigma_{\ell,\ell'} \leq 8(L+1) z_{\ell,\ell'} \tag{6.31}$$

$$\forall \ell', \ell \quad \sigma_{\ell,\ell'} \geq \sum_{m=\ell'}^{2L+2-\ell'} (v_m + v'_m + \widetilde{v}_m + \widetilde{v}'_m) - 8(L+1)(1 - z_{\ell,\ell'}) \tag{6.32}$$

We use $8(L+1)$ as an upper bound on $\sigma_{\ell,\ell'}$ for any ℓ, ℓ'. It roughly corresponds to the situation when all $v_m, v'_m, \widetilde{v}_m, \widetilde{v}'_m$ are equal to 1 for all $m : 1 \leq m \leq 2L+1$.

Since Lemma 17 only provides the number of activations at the beginning of the period, we also need to account for the following events that may take place between the beginning of the period and instant τ_ℓ (i) a forward task $T_{\ell'}$, $\ell' \leq L+1$ is computed, inducing an extra activation $a_{\ell'-1}$ in memory, and (ii) a backward task $T_{2L-\ell'+1}$, $\ell' \leq L$ is computed, removing an activation $a_{\ell'-1}$ from memory.

Lemma 18. *Consider any valid pattern according to Lemma 15, satisfying Constraints (6.23)-(6.32).*

The amount of memory occupied when task T_ℓ is performed by activations required by future backward tasks is $\mathrm{M}_\ell^{(\mathrm{ACT})}$:

$$\mathrm{M}_\ell^{(\mathrm{ACT})} = a_{\ell-1} + a_\ell + \sum_{\ell'=1}^{L+1} (\sigma_{\ell,\ell'} + w_{\ell',\ell} - w_{2L+3-\ell',\ell}) a_{\ell'-1}.$$

Доказательство. On the one hand, if ℓ' is on the same processor as ℓ, then $\sigma_{\ell,\ell'}$ is at least the number of replicas for a layer ℓ' derived in Lemma 17, otherwise $\sigma_{\ell,\ell'} = 0$. Indeed, if T_ℓ and $T_{\ell'}$ share the same resource then $z_{\ell',\ell} = 1$ and since the number of replicas is less than $8(L+1)$ for any layer, $\sigma_{\ell,\ell'} \geq \sum_{m=\ell'}^{2L+2-\ell'} (v_m + v'_m + \widetilde{v}_m + \widetilde{v}'_m)$. On the other hand, when T_ℓ and $T_{\ell'}$ are on different resources $z_{\ell',\ell} = 0$ and $\sigma_{\ell',\ell} \leq 0$.

According to Lemma 17 the value $\sum_{m=\ell'}^{2L+2-\ell'} (v_m + v'_m + \widetilde{v}_m + \widetilde{v}'_m)$ represents the number of activations $a_{\ell'-1}$ stored in the beginning of the period, but this number may vary within the period: it increases by one after each task $T_{\ell'}$ ($F_{\ell'}$) and it decreases by one after task $T_{2L+3-\ell'}$ ($B_{\ell'}$). Thus, the last term in the equation for $\mathrm{M}_\ell^{(\mathrm{ACT})}$ corresponds to all activations that have been stored before task T_ℓ and the rest represents the memory needed to perform task T_ℓ.

□

6.3.3 Memory Required for the Models

If there are more than one active mini-batch in the pipeline then it is required to store several copies of the model to compute valid gradients (in order to process one mini-batch,

forward F_ℓ and backward B_ℓ for any layer ℓ should use the same weights). In PipeDream, for instance, they suggest using as much copies of the weights as there are concurrent mini-batches in pipeline. As it was shown in [79], having just two models stored is enough to perform training. Indeed, we need one model to execute forward propagation and one model to be updated. These two models should constantly exchange their "roles"to make training feasible. For example, for the case illustrated in Figure 4.8, one may use two versions of model weights W^1 and W^2 in the following way: mini-batches 0, 1, 2 are performed with W^1 (W^1 is active and W^2 is passive), mini-batches 3, 4, 5 are performed with W^2 (now W^2 is active and W^1 is passive), mini-batches 6, 7, 8 are performed with W^1 and so on. The gradients from mini-batches 0, 1, 2 are accumulated, W^1 is updated after the backward pass on mini-batch 2 by applying the aggregated gradient to W^2; next, the gradients from mini-batches 3, 4, 5 are accumulated, W^2 is updated after the backward pass on mini-batch 5 by applying the aggregated gradient to W^1... For the general case, if $\text{NCA}_1 = n$, two weight versions switch their modes (from active to passive and vice versa) after each n-th iteration, furthermore, at the end of each n-th backward the active weight version is updated by applying its gradients to the passive weight version.

Since the computed gradients have the same shape as model weights, they require the same amount of memory. Furthermore, as it was observed also in [86], depending on the optimizer and if mixed precision is used in the training, the equivalent of C model copies should be stored in the memory. The minimal number of copies is $C = 3$ (2 slots for models in pipelining and 1 slot for gradients), though it may be higher than 10 (in [86] without pipelining $C = 16$). Thus,

$$\text{M}_\ell^{(\text{MOD})} = C \sum_{\ell,\ell' \leq L+1} z_{\ell,\ell'} W_{\ell'}. \tag{6.33}$$

Further in Section 6.5, we fix $C = 3$, which corresponds to the minimal required storage space.

6.3.4 Memory Buffer for Communications

Another type of memory usage on a processor P_i is the buffer memory to store activations or gradients: *incoming* ones, that were computed by another processor and then sent to P_i, or *outgoing* ones, that were computed on P_i and need to be sent away.

Assumptions 5. *We assume that a buffer is allocated for each incoming and outgoing data for the whole duration of the execution and they are not shared between different data.*

We introduce binary variables $b_{\ell,\ell'}$ for all ℓ and ℓ', together with the following constraints:

$$b_{\ell,\ell'} \geq z_{\ell,\ell'} - z_{\ell,\ell'+1} \tag{6.34}$$

$$b_{\ell,\ell'} \geq z_{\ell,\ell'+1} - z_{\ell,\ell'} \tag{6.35}$$

Lemma 19. *Consider any valid pattern according to Lemma 15, satisfying Constraints (6.34) and (6.35). If a buffer is required for $a_{\ell'}$ on the processor that computes T_ℓ, then $b_{\ell,\ell'} = 1$. Hence, the memory reserved for buffers at the start of task T_ℓ is at most $\mathrm{M}_\ell^{(\text{BUF})} = \sum_{\ell'=1}^{2L+2} b_{\ell,\ell'} a_{\ell'}$.*

Доказательство. Constraint (6.34) ensures that $b_{\ell,\ell'} \geq 1$ if there exists a buffer for $a_{\ell'}$, when it is sent out from processor P_i, while constraint (6.35) checks if we need a buffer for $a_{\ell'}$ for processor P_i to receive it. Thus, total memory reserved for buffers on processor P_i with T_ℓ can be found by summing $b_{\ell,\ell'} a_{\ell'}$ for all ℓ', which completes the proof. □

6.4 Final Integer Linear Program

We can now define the complete Linear Program for our problem: the objective is to minimize T, subject to Contraints (6.1)-(6.35), together with

$$\begin{aligned}
\forall \ell, \quad & \mathrm{M}_\ell^{(\text{MOD})} + \mathrm{M}_\ell^{(\text{ACT})} + \mathrm{M}_\ell^{(\text{DIR})} + \mathrm{M}_\ell^{(\text{BUF})} \leq M \\
\forall \ell, \quad & T, \tau_\ell, \widetilde{\tau}_\ell \in \mathbb{R} \\
\forall \ell, \ell' \quad & \sigma_{\ell,\ell'} \in \mathbb{Z} \\
\forall \ell, \quad & f_\ell, v_\ell, v'_\ell, \widetilde{v}_\ell, \widetilde{v}'_\ell \in \{0,1\} \\
\forall \ell, \ell', \quad & z_{\ell,\ell'}, w_{\ell,\ell'}, \widetilde{z}_{\ell,\ell'}, \widetilde{w}_{\ell,\ell'}, o_{\ell,\ell'}, b_{\ell,\ell'} \in \{0,1\}
\end{aligned} \tag{6.36}$$

Theorem 19. *The solution of the above Integer Linear Program provides an optimal solution to* $\textsc{Pipe}_{\mathcal{S}^1\&\mathcal{A}}(L, M, P, \beta)$ *under Assumptions 5*

Доказательство. Lemma 15 shows that with the help of the ILP we can find a 1-periodic schedule with a minimal period T when memory is not limited ($M \to \infty$), while Lemmas 16-19 introduce the variables that describe memory consumption on each processor at any moment. Therefore, adding Constraint (6.36) guarantees that the ILP finds solutions that satisfy memory limit. Bringing everything together leads to the result of the theorem. □

6.5 Experimental Results

In this section, we present simulation results obtained for different state-of-the-art ResNet neural networks of size 18, 34 and 50, which are widely used for a large range of tasks. In order to perform these simulations, we first perform the profiling of the neural networks to measure the durations and memory costs of different operations involved in the training. As mentioned in Section 6.1, this work only considers networks in the shape of adjoint chains as depicted in Figure 4.7. In the case of ResNet networks, a simple linearization approach is enough to transform the neural network computational graphs into chains. It was previously discussed in Chapter 2

We implemented the ILP from Section 6.4 using the CPLEX solver [80]. In all our experiments, the execution time was limited to one hour. In case there is still a gap after one hour, we keep the best current solution computed by the solver. For ResNet-18 up to ResNet-50, the solutions produced were of very good quality (see the discussion below), though the solver was unable to prove its optimality. The results obtained are therefore heuristic in nature.

This time limit is reasonable, because the computed solution can be used during the entire training phase associated with a given image and mini-batch sizes, a given computing platform and a given network. It is common for the training to last several hours/days on a parallel platform, which makes this approach acceptable.

To evaluate the quality of the solutions produced by the integer linear program, we compare the results obtained with those of PipeDream [78], which is the state of the art solution for Pipelined Model Parallelism. In practice, PipeDream takes as input the memory limit and the characteristics of the platform, computes the number of mini-batches to be inserted in the pipeline, called NOAM, and finds a partitioning of the network that is used for model parallelism. PipeDream then uses a greedy 1F1B strategy to schedule tasks. More details about PipeDream limitations can be found in Chapter 5.

Despite these limitations, PipeDream can be used to produce a large number of solutions that can be used to build valid solutions that fulfill the memory constraints. We use this approach in the following experiments. For a large number of possible memory targets and possible NOAM values, we produce the allocations computed by PipeDream, simulate the execution of the eager scheduling strategy for 500 mini-batches, and we evaluate a posteriori the actual memory consumption and the average length of the period that can be obtained. We thus obtain a set of (memory, period) pairs that correspond to feasible solutions. Our observations indicate that the solution produced by PipeDream generally consumes much more memory than the target value. Nevertheless, since the execution time of PipeDream is small, obtaining a set of good valid solutions through this "exhaustive" approach is still practical. In Figures 6.5 to 6.11, blue dots correspond to the actual memory consumption and observed period of solutions computed with this approach. The red dots correspond to the best solutions found by CPLEX for our ILP after one hour, for different values of the memory limit.

First of all, it can be observed that in case of small networks the solutions returned by our ILP are almost always better than the solutions returned by the exhaustive approach based on PipeDream, even if optimality cannot be guaranteed. It can be observed that the ILP is able to find better solutions both in cases where memory is scarce (Figures 6.6 and 6.10) and where memory is abundant (Figures 6.5 and 6.8). When memory is abundant, the ability of the ILP to use non-contiguous allocations helps to achieve better load-balancing. When memory is scarce, the precise scheduling formulation of the ILP leads to better solutions by reducing memory costs. The solutions produced by the ILP are therefore of very high quality when the size of the neural network is not too large.

There are two examples where the ILP gives worse results than PipeDream. The first example is Figure 6.7, where in the case of a memory size of 1 GB, PipeDream finds

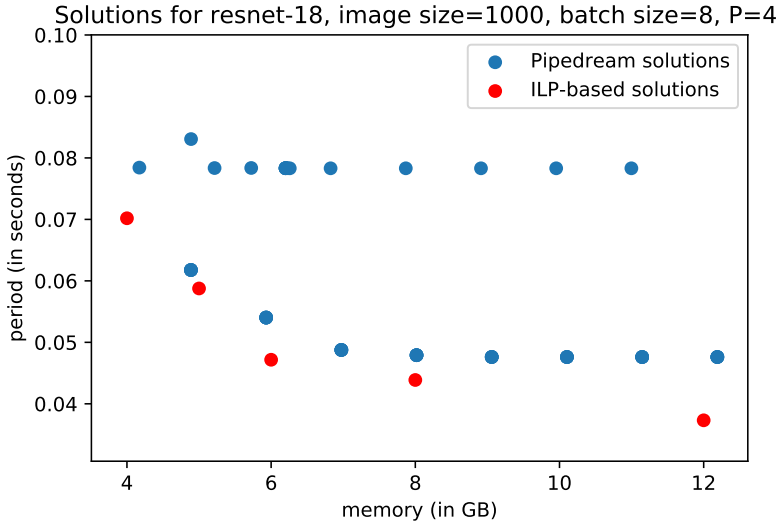


Рис. 6.5: Results with ResNet-18 and 4 processors

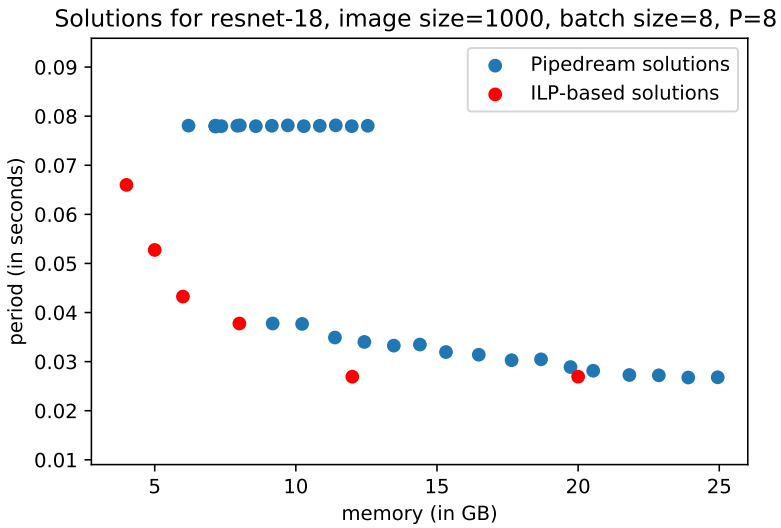


Рис. 6.6: Results with ResNet-18 and 8 processors

solutions that strictly dominate those of the ILP. The second example is Figure 6.11, where the neural network is too large and heavy and the ILP fails to find a better solution than PipeDream within the time limit. Despite that, the ILP still succeeds to find some solutions when memory is low, while PipeDream cannot train if available memory is less than 6 GB. On larger neural networks such as ResNet-101 or DenseNet-121 (of respective depths 101 and 121), one hour of execution is sometimes not enough to find integral solutions of good quality. In this case, it is necessary to consider approximate approaches that could find a good valid solution in a reasonable time. To address the large DNNs, we

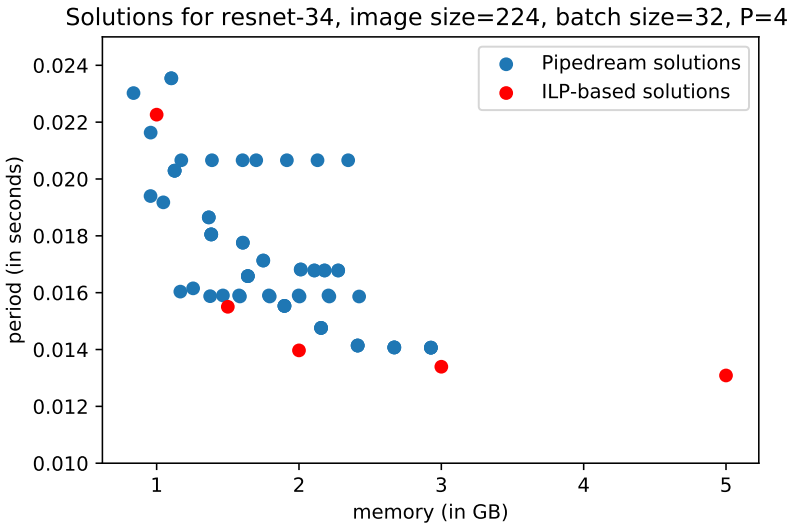


Рис. 6.7: Results with ResNet-34 and 4 processors

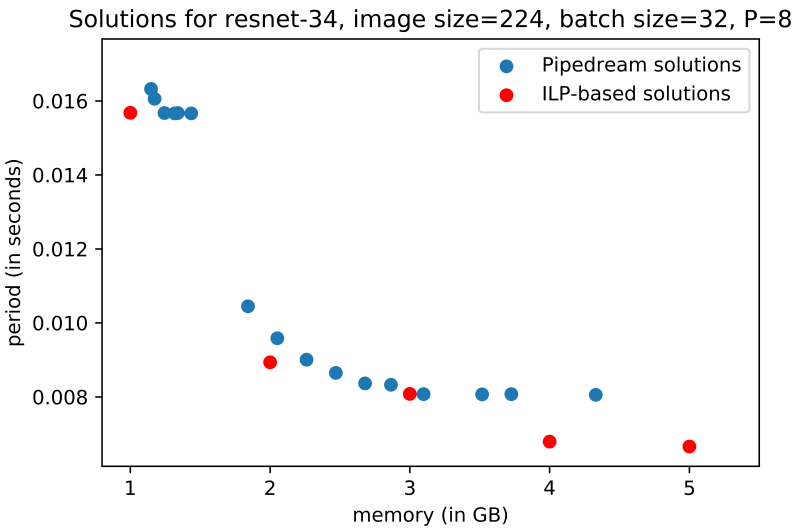


Рис. 6.8: Results with ResNet-34 and 8 processors

therefore propose MadPipe, a tool based on dynamic programming, in Chapter 7.

6.6 Conclusion

In this chapter, we consider the problem of pipelined model parallelism. Following the ideas of the previous chapter (Chapter 5), where we observed that non-contiguous allocations may significantly improve the throughput of the training, we want to find the solutions

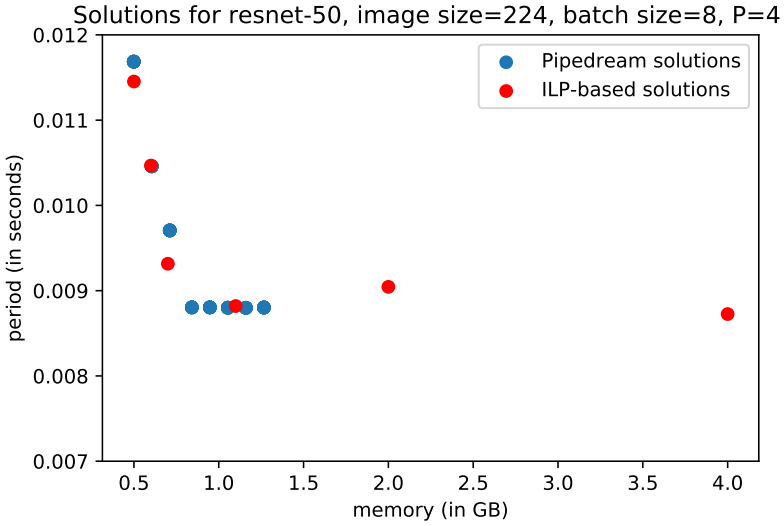


Рис. 6.9: Results with ResNet-50 and 4 processors

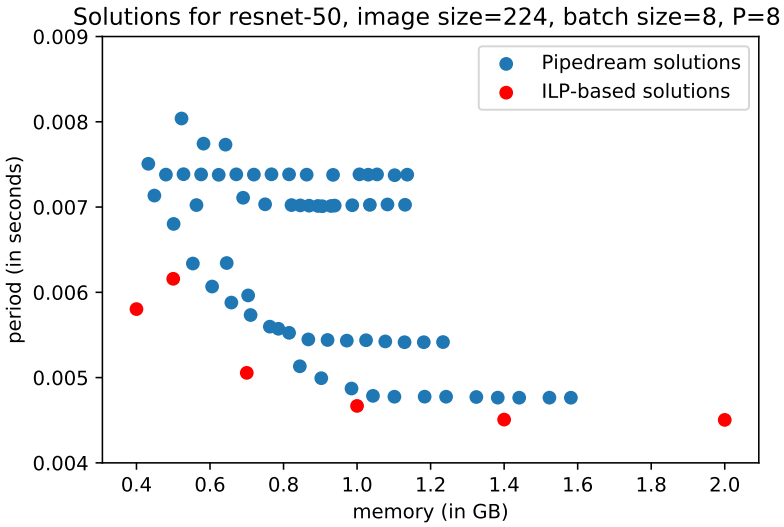


Рис. 6.10: Results with ResNet-50 and 8 processors

that exploit this idea. Nevertheless, the combination of Pipelining and Model Parallelism requires to store more activations at the nodes, which in turn causes memory problems. We propose a very fine analysis of the memory costs induced by this combination. We integrate these memory considerations together with non-contiguous allocations in the ILP. Thus, our ILP helps to find a general allocation of layers that explicitly takes actual memory costs into account, contrary to what is done in PipeDream.

Through experiments on medium size networks (ResNet-18 to ResNet-50), we show that the ILP is able to compute in reasonable time solutions that are better than

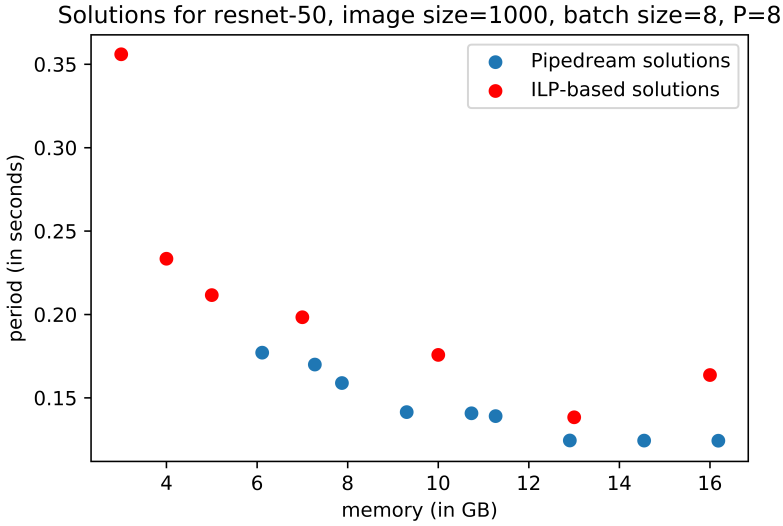


Рис. 6.11: Results with ResNet-50 and 8 processors

those computed by PipeDream, by both providing good non-contiguous allocations and good scheduling strategies. Nevertheless, the computing cost induced by the integer programming approach becomes too large for very deep neural networks, and therefore, new heuristic solutions are required in this case. We consider one of such heuristics in Chapter 7.

Глава 7

MadPipe

We showed in the previous chapter that problem $\textsc{Pipe}_{\mathcal{S}^1\&\mathcal{A}}(L, M, P, \beta)$ can be solved optimally with Integer Linear Programming. However, solving the ILP may take exponential time, therefore it may require hours of execution to achieve the optimal solutions even for small size problems. Therefore, it is important to set a time limit for the ILP execution in order to obtain the solutions in the reasonable time. Indeed, the experiments (see Section 6.5) were done only for small or medium size neural networks and some results are heuristic in nature as the optimal solutions are not always reached by the solver within the time limit.

In this chapter, we propose a heuristic MadPipe that is able to find solutions for large neural networks in polynomial time. This heuristic is based on dynamic programming for finding non-contiguous allocations under assumption that there is one processor working with several stages, while the remaining processors process only one stage. Section 7.1 presents the equations for the dynamic program, whereas Section 7.2 describes how to use the ILP to schedule MadPipe produced allocation. Finally, we demonstrate that MadPipe allocations can significantly outperform PipeDream allocations even when PipeDream solutions are scheduled with the optimal 1-periodic schedule 1F1B*.

7.1 Building a Non-Contiguous Allocation

In this section, we present the first part of the MadPipe algorithm: a dynamic programming algorithm to build a non-contiguous allocation. However, we do not consider general allocations, because solving the corresponding problem (as is done with an Integer Linear Program in Chapter 6) is proved to be NP-complete in the strong sense (Theorem 13 in Chapter 5 Section 5.1). Instead, we focus on a specific case: we look for allocations in which every processor is allocated for only one stage (like in a contiguous allocation, we call these processors *normal*), except for one *special* processor that may receive any number of stages. As shown in Section 7.3, this is enough to significantly improve load balancing.

A non-contiguous allocation $\mathcal{A}_{n\mathcal{C}}$ is a partition of the layers into more than P stages: some processors (in our case, the special processor) can be designated for several stages.

$\mathcal{A}_{nC}$ also specifies an assignment of stages to processors. We associate the period to each such allocation that can be achieve if memory constraints were ignored. Thus, it can be computed as the total load of the most loaded resource (either a GPU or a communication link).

In the following, we denote $U(i,j) = \sum_{\ell=i}^{j} u_{F_\ell} + u_{B_\ell}$ the total computational cost of a stage $s(i,j)$ that starts at the layer i and ends at the layer j, and $C(j) = \frac{a_j+\delta_j}{\beta}$ the communication time of its output (stage $s(i,j)$ communicates the output of F_j and the input of B_j with the next stage). Further, we use the discussion about 1F1B* (see Chapter 5 Section 5.2.1) to accurately estimate the memory usage for all normal processors, based on the total computation time of layers further down the chain. The 1F1B* algorithm requires a target period, so our dynamic programming method uses a target $\hat{T}$ as input, and computes the best possible period for an allocation in which memory needs are computed assuming a period $\widehat{T}$.

7.1.1 Estimating Memory Usage

Let us assume that layers i to j are assigned to a normal processor, while $\text{NCA}_\ell = g$ for any $\ell : i \leq \ell \leq j$. The memory usage on this processor is $\mathcal{M}(i,j,g) = \sum_{\ell=i}^{j}(CW_\ell + ga_{\ell-1}) + a_{i-1} + a_j + \delta_{i-1} + \delta_j$, where CW_ℓ represents the memory usage of the model parameters (we set $C = 3$ as in Chapter 6 Section 6.3), $ga_{\ell-1}$ corresponds to the activations, and $a_{i-1} + a_j + \delta_{i-1} + \delta_j$ accounts for the communication buffers (if $i = 1$ or $j = L$, the corresponding term should be removed since no communication takes place). To compute the value g, assume that we are given a lower bound V on the delay between the execution of F_j on some mini-batch and the execution of B_j on the *same* mini-batch. Then these layers can be scheduled with g activations in memory if and only if $V + U(i,j) \leq g \cdot \widehat{T}$. Hence, we can compute the number of activations to be kept in memory for layers i to j as $g(i,j,V) = \left\lceil \frac{V+U(i,j)}{\widehat{T}} \right\rceil$, and the memory usage is given by $\mathcal{M}(i,j,g(i,j,V))$. Note that this estimation of g corresponds to computing the group number in 1F1B* (see Algortithm 6 in Chapter 5 Section 5.2) and it is based on the fact that a layer ℓ of group g has $\text{NCA}_\ell^{\text{1F1B*}} = g$.

For the special processor however, estimating the memory usage is more difficult. We can use the same formula $g(i,j,V)$ to compute the number of activations to be kept in memory for each stage assigned to the special processor. However, as can be seen on Figure 7.1, for a given allocation on this processor, the memory peak depends on how the different stages are scheduled. Specifically, assume that several stages are assigned to the special processor, where stage s_k is part of group $g(s_k)$ (in this context function $g(\cdot)$ returns a group number of a stage). If all forwards of stages are performed in sequence followed by all backwards, then after the end of the last forward operation, each stage s_k on a special processor (without loss of generality, let us assume that it is processor P) stores $g(s_k)$ activations, for a total memory peak of

$$\mathcal{M}_{\text{worst}}^{(\text{ACT})} = \sum_{s_k \text{ on } P} g(s_k) \sum_{\ell \in s_k} a_{\ell-1}.$$

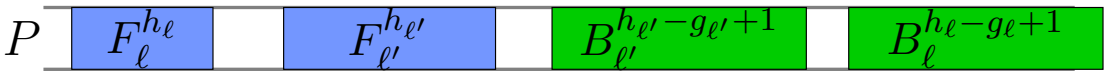
It corresponds to the case illustrated in Figure 7.1a.

Alternatively, if the backward operation of each stage is performed just after its forward operation, then after any backward the memory occupied by activations measures $\sum_{s_k \text{ on } P}(g(s_k)-1)\sum_{\ell \in s_k} a_{\ell-1}$. Executing forward $F_{s_{k'}}$ of some stage $s_{k'}$ should increase the number of activations stored for stage $s_{k'}$ up to $g(s_{k'})$. Thus, the overall memory peak can be found by taking the maximum value among all $s_{k'}$ on processor P:

$$\mathcal{M}_{\text{best}}^{(\text{ACT})} = \max_{s_{k'} \text{ on } P} \left\{ g(s_{k'}) \sum_{\ell \in s_{k'}} a_{\ell-1} + \sum_{\substack{s_k \neq s_{k'} \\ s_k \text{ on } P}} (g(s_k)-1) \sum_{\ell \in s_k} a_{\ell-1} \right\}.$$

This situation is depicted in Figure 7.1b.

Many more ways of interleaving the operations can take place, and determining which ones are compatible with the schedule of the other processors is difficult. The true value of $\mathcal{M}^{(\text{ACT})}$ lies between $\mathcal{M}_{\text{best}}^{(\text{ACT})}$ and $\mathcal{M}_{\text{worst}}^{(\text{ACT})}$. In any case, at least $g(s_k)-1$ activations for each stage s_k need to be stored at all times. For this reason, in this first part of MadPipe, we underestimate the memory requirement of the special processor by considering that it requires $\mathcal{M}(i,j,g-1)$, and rely on a modified version of the ILP given in Chapter 6 Section 6.4 to compute a feasible schedule that satisfies the memory constraints in the second part of MadPipe (see Section 7.2).



(a) Worst case. Memory needed to store the activations: $g_\ell a_{\ell-1} + g_{\ell'} a_{\ell'-1}$ (reached between $F_{\ell'}$ and $B_{\ell'}$).

(b) Best case. Memory needed to store the activations: $\max\{g_\ell a_{\ell-1} + (g_{\ell'}-1)a_{\ell'-1}, (g_\ell-1)a_{\ell-1} + g_{\ell'}a_{\ell'-1}\}$ (reached either after F_ℓ or after $F_{\ell'}$).

Рис. 7.1: Two schedules with different memory peaks with two layers assigned on the special processor: layer ℓ with index shift h_ℓ and group g_ℓ, layer ℓ' with index shift $h_{\ell'}$ and group $g_{\ell'}$.

7.1.2 Dynamic Programming Derivations

To specify our dynamic programming algorithm, we fix a target value $\widehat{T}$, and we define $T(j,p,t_P,m_P,V)$ as the smallest period of an allocation of the first j layers on p normal processors that fulfills the above memory constraints, where (i) the delay between the end of F_j and the start of the corresponding B_j on the same mini-batch is at least V, and

(ii) assuming that the special processor has already been allocated for some layers that induce a computational time t_P and memory usage m_P.

Consider any such allocation of the first j layers. The last layer j is part of some stage starting at layer i, with $i \leq j$. From i, j and V we can compute a lower bound V' on the time between the end of F_{i-1} and the start of B_{i-1}, by mimicking the group-making process of the 1F1B* procedure (Algorithm 6 of Chapter 5). Denote by $g^0 = \left\lceil \frac{V}{\widehat{T}} \right\rceil$ the group number of the previously considered stage. Layers i to j can use the same group if $\left\lceil \frac{V+U(i,j)}{\widehat{T}} \right\rceil = g^0$, in which case the delay between the end of the communication of a_{i-1} and the start of the communication of δ_{i-1} is $V + U(i,j)$. Otherwise, it is necessary to start a new group $g^0 + 1$, which implies that this delay is $g^0 \cdot \widehat{T} + U(i,j)$. The same reasoning applies to the group number of the communication of a_{i-1} and δ_{i-1}. By introducing the notation

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

we obtain $V' = (V \oplus U(i,j)) \oplus C(i-1)$.

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

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

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

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

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

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

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

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

7.1.3 Find the Correct Value for $\widehat{T}$

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

Both arguments are monotonic in opposite directions, and we can therefore use a modified binary search algorithm to find $\widehat{T}^*$ (see Algorithm 8). At each step, if $T = \text{MadPipe-DP}(\widehat{T})$, we know that $\min(T, \widehat{T})$ is a lower bound for $\widehat{T}^*$, and $\max(T, \widehat{T})$ is an upper bound. We perform this algorithm for a fixed number of iterations. In practice, $K = 10$ iterations are enough to obtain a good solution.

Algorithm 8 First phase of MadPipe: build an allocation

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

7.2 Scheduling with ILP

As discussed above, it might not be possible to schedule the allocation returned by Algorithm 8 within the expected period because of the approximation used to estimate the memory usage of the special processor. In order to obtain a valid schedule, the second step of the MadPipe algorithm makes use of the ILP formulation from Chapter 6 Section 6.4 to compute an efficient valid schedule that uses the same partitioning.

Assume that Algorithm 8 returns a solution where the layers are partitioned into N stages $(s_1, \ldots, s_N)$. Scheduling this partitioning is almost equivalent to scheduling a (shorter) chain network of length N, where each stage of the partition is considered as a layer of the transformed chain. Each layer of this new chain is thus more computationally expensive, with $u'_{F_{s_k}} = U(s_k)$. The equivalence is however not perfect since data dependencies are different. Indeed, if stage s_k contains layers $(i, \ldots, j)$, the output of

its forward operation F_{s_k} is the activation a_j produced by layer j. However, the backward operation B_{s_k} requires $(a_{i-1}, \ldots, a_{j-1})$, and not only a_{j-1}.

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

7.3 Experimental Results

7.3.1 Simulation Settings

In this section, we present the simulation results obtained for different state-of-the-art and widely used neural networks. The data necessary to perform the simulations were obtained by profiling the neural networks to measure the durations and memory costs of the different operations involved in the training. As mentioned before, all the networks are considered as adjoint chains as depicted in Figure 4.7. A classic linearization approach, also used for PipeDream [78], is used to transform the computational graphs of these neural networks into chains, by greedily grouping layers as necessary (see Chapter 2).

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

We consider a wide variety of situations. The measurements were performed on the ResNet-50, ResNet-101, Inception, and DenseNet-121 networks, with large image sizes of 1000×1000 and mini-batch size of 8. Such a setting with large activations makes it difficult to train these neural networks on a single GPU. The number of GPUs varies from 2 to 8, and the available memory per GPU varies from $M = 3$ GB to $M = 16$ GB. Even though GPUs have a fixed memory size (usually 16GB), exploring lower memory values helps to assess the sensitivity of the algorithms to more constrained scenarios. These scenarios

can be seen as representative of cases with larger batch size or larger image sizes. The bandwidth measured on our platform is $\beta = 12$ GB/s, and we also performed experiments with $\beta = 24$ GB/s to explore the possibilities offered by better networking capabilities.

7.3.2 Comparison with ILP

MadPipe is designed as a heuristic, so, obviously, it cannot find the optimal solution. However, due to discretization, it is able to find a good load balancing in a polynomial time, thus it can process large neural networks when the ILP cannot. Here, we compare the ILP presented in the previous chapter Section 6.4, PipeDream whose behavior is simulated in the same way as in Chapter 6 Section 6.4 and MadPipe executed as described in Section 7.3.1.

Figure 7.2 shows the period lengths for a simple neural network ResNet-34. It can be easily seen that MadPipe behaves as PipeDream for high memory limits and has higher period than the ILP-based solutions. However, in case of low memory limit (1 GB), MadPipe outperforms the ILP solution. Indeed, it is harder for the ILP to converge to the optimal solution when memory constraint is too tight.

Figure 7.3 demonstrates another situation. As it was previously observed in Chapter 6 Section 6.5, ResNet-50 with a large image size becomes a hard problem for the ILP to solve, showing worse performance than PipeDream (due to lack of convergence). Alternatively, MadPipe reduces the period by 1.5 times with respect to PipeDream, proving that this heuristic can be an interesting alternative. Further comparison between PipeDream and MadPipe is provided in the next section for the larger networks that the ILP fails to process.

7.3.3 Simulation Results

Now, we compare only two scheduling algorithms: PipeDream and MadPipe. PipeDream does not use the optimal 1-periodic schedule (see Section 5.2.1). In the previous section, the best period of PipeDream is found by simulating a lot of runs for different NOAM values. Instead, we can use 1F1B* schedule (Chapter 5 Section 5.2.1), which is proved to be optimal for a fixed allocation (thus we can as well take the one returned by PipeDream). This allows a fairer comparison between these two approaches and to assess the advantage of using non-contiguous allocations.

Figures 7.4-7.7 show the simulation results for the ResNet-50, ResNet-101, Inception and DenseNet-121, representing how period lengths of solutions are affected by different memory limits. For both PipeDream and MadPipe, the dashed lines represent the period of the allocations obtained in the first phase by the respective dynamic programs, and the period of the valid schedule is depicted with a solid line. This figure presents results in terms of period duration, hence lower is better (the throughput, in terms of images processed by second, is proportional to the inverse of the period).

These plots highlight the fact that MadPipe helps to obtain significantly more efficient schedules in most cases, especially when the memory is more constrained, and for $P > 2$.

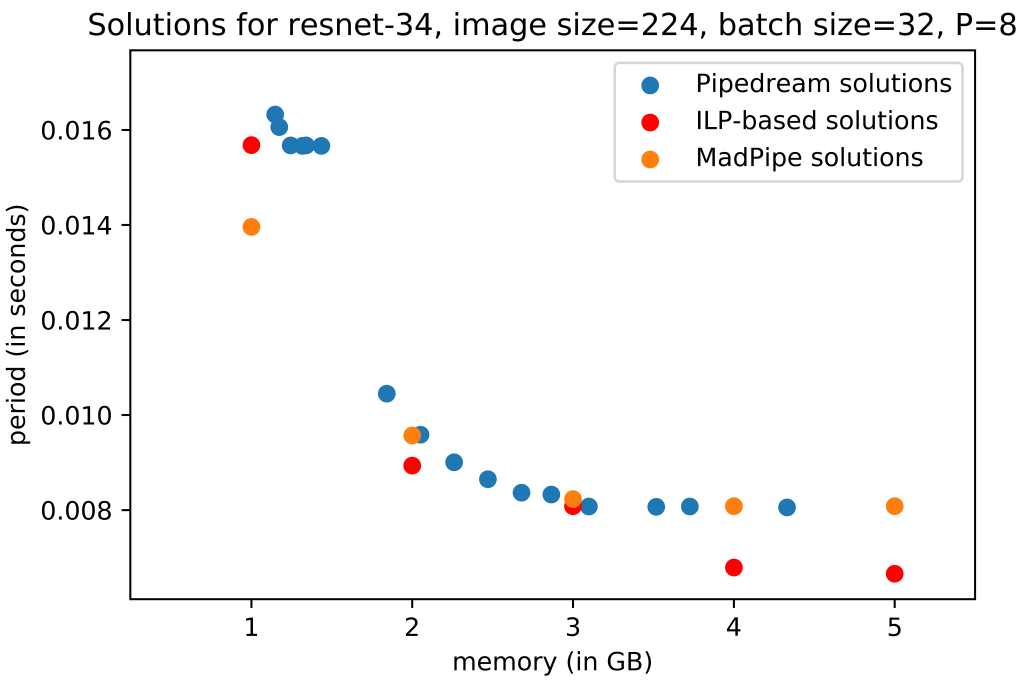


Рис. 7.2: Comparison of the ILP, MadPipe and PipeDream for ResNet-34 with image size 224, mini-batch size 32 and $P = 8$.

The period achieved by PipeDream is routinely 20% larger than what MadPipe can provide, and in some cases the solution of PipeDream is up to two or even three times slower. On the one hand, the dashed lines show that the partitioning produced by PipeDream is very optimistic and expects to achieve a very small period, but then turns out infeasible, resulting in a very high overhead. On the other hand, the partitioning obtained by the first step of MadPipe has a higher period because it considers more memory constraints, but it generally results in a more efficient solution.

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

On Figure 7.8, we display the results of the same simulations for all networks, normalized with respect to the MadPipe algorithm. For each case, we compute the ratio of the period obtained by a solution to the period obtained by MadPipe. The plots show, for each value of the memory limit M and for several neural networks, the geometric mean of these ratios over all values of P and β. For the solid red line that corresponds

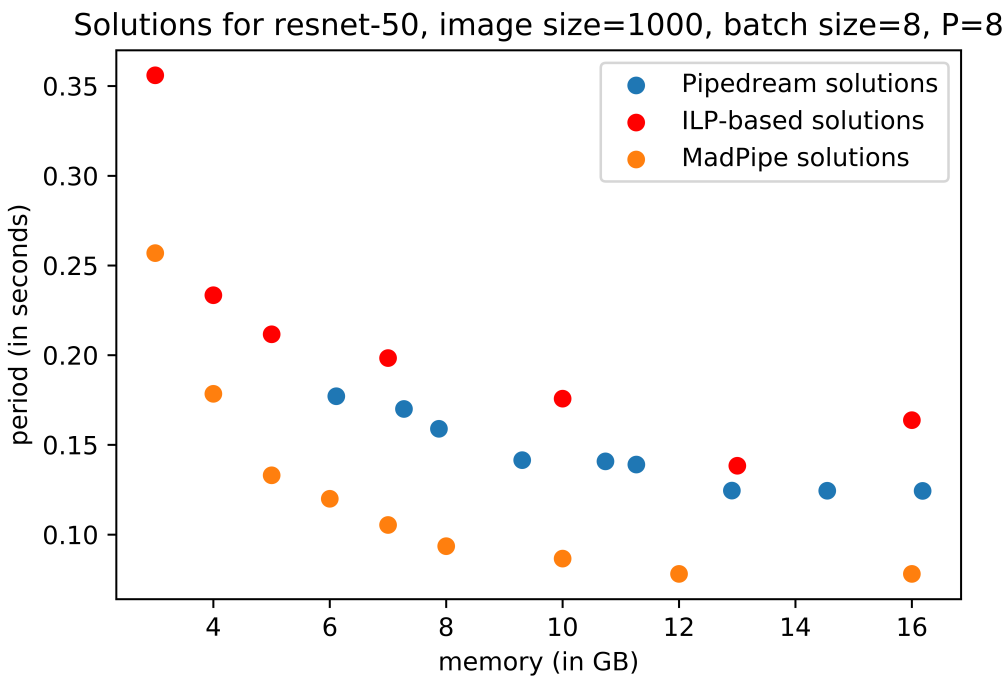


Рис. 7.3: Comparison of the ILP, MadPipe and PipeDream for ResNet-50 with image size 1000, mini-batch size 8 and $P = 8$.

to PipeDream (DP+1F1B*) solution, a value below 1 means that PipeDream is more efficient, and a value above 1 means that our solution is more efficient than PipeDream. This shows that the performance improvement offered by MadPipe is valid in a wide range of scenarios, especially for lower values of memory. Indeed, the overhead of PipeDream over MadPipe is consistently over 20% when the available memory is below 10GB.

Finally, we present on Figure 7.9 another visualisation of the same results that aims at highlighting the scalability of the produced schedules when the number of processors increases. On this figure, the plots provide the speedup of the produced schedules compared to the sequential execution of the network, *i.e.* $U(1, L)$. We can observe that the pipelined model parallelism achieves a good scalability for settings with large memory like $M = 12$ or 16, and that MadPipe exhibits better scalability than PipeDream. When less memory is available, it is more difficult to use all the computing resources efficiently and the speedup gets worse.

Increasing the bandwidth shows only a marginal improvement in speedup, which implies that communications do not significantly hinder the performance. This reduced scalability when memory is tight comes from the heterogeneity between layers and from the increased memory pressure. Indeed, when the number of GPUs increases, the number of activations to be stored also increases, in particular for the first layers of the network, which generally handle the larger activations. Therefore, the memory becomes the main bottleneck and idle times have to be added to fit into the memory limit. This explains why

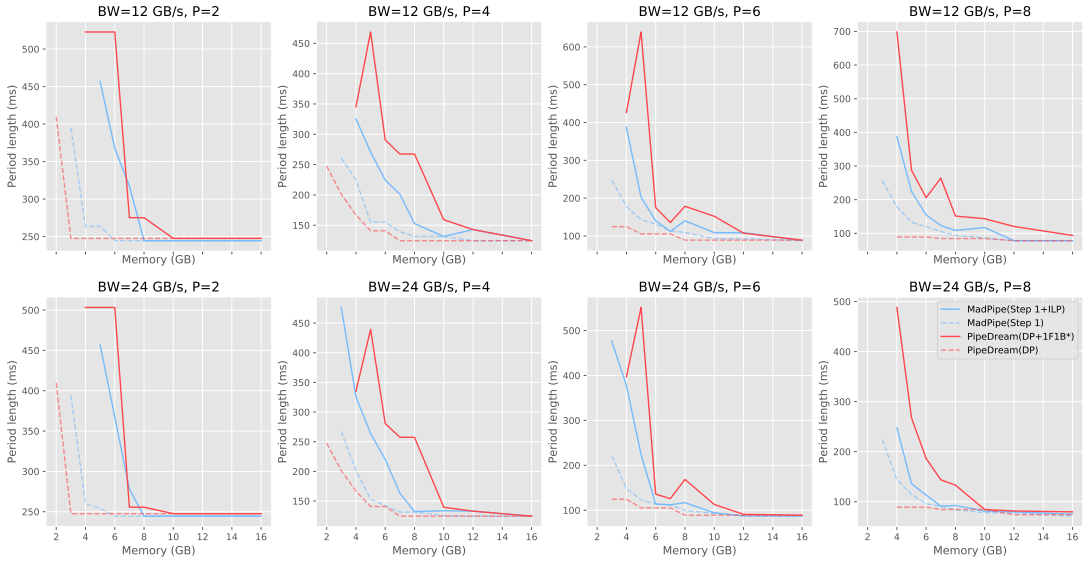


Рис. 7.4: Comparison of periods for ResNet-50 with image size 1000 and mini-batch size 8.

the scalability of MadPipe is much better than what can be achieved with PipeDream.

7.4 Conclusion

In this chapter, we propose a sophisticated two-phase scheduling strategy that produces non-contiguous schedules, based on a dynamic program to group neural network layers into stages. We show, using a large set of simulations on a variety of computing platforms and neural networks, that our solution MadPipe achieves a significant increase in throughput compared to PipeDream, especially when the memory is a strong constraint.

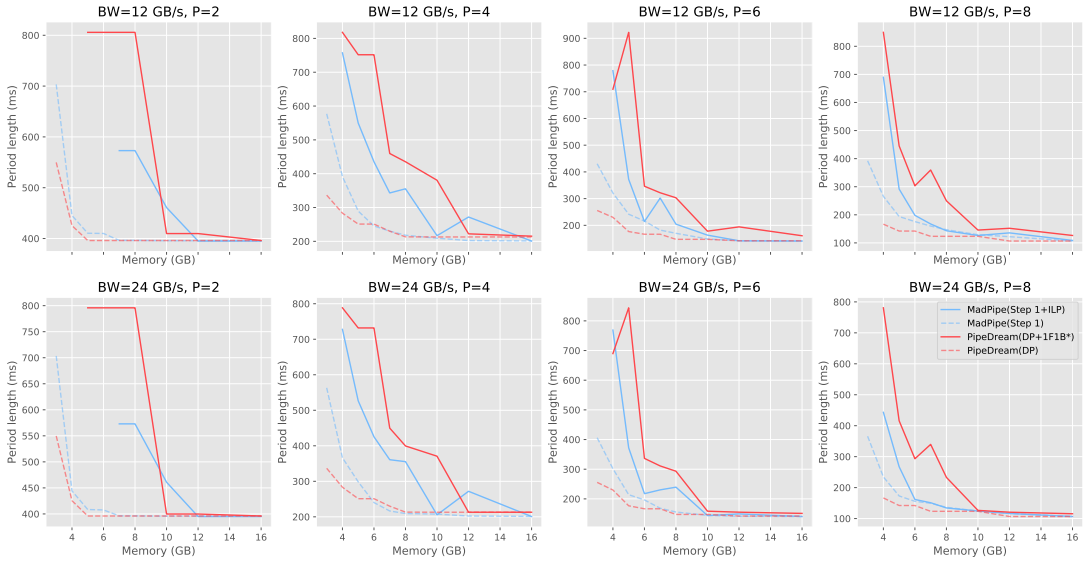


Рис. 7.5: Comparison of periods for ResNet-101 with image size 1000 and mini-batch size 8.

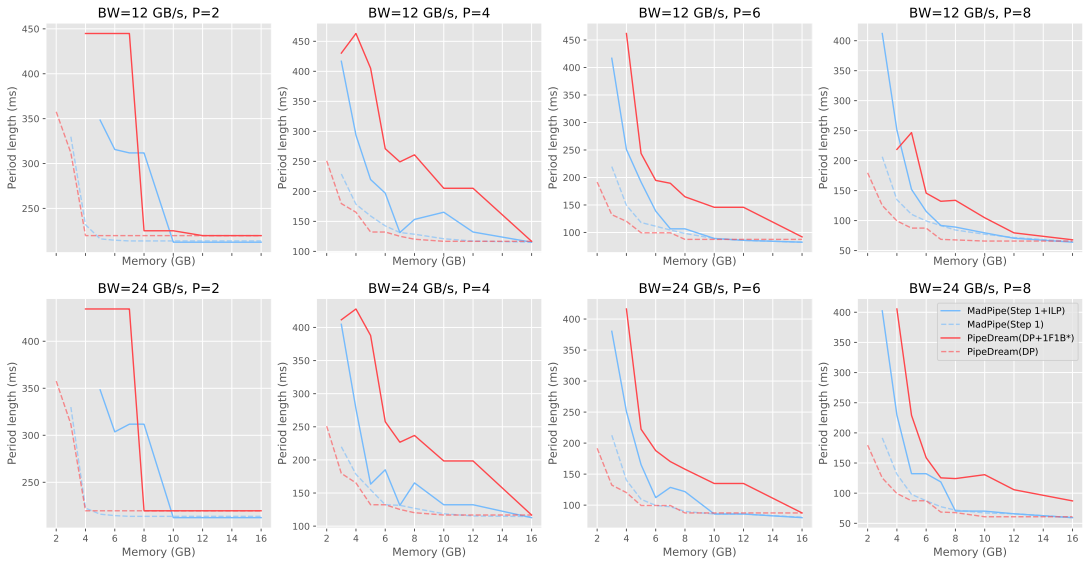


Рис. 7.6: Comparison of periods for Inception with image size 1000 and mini-batch size 8.

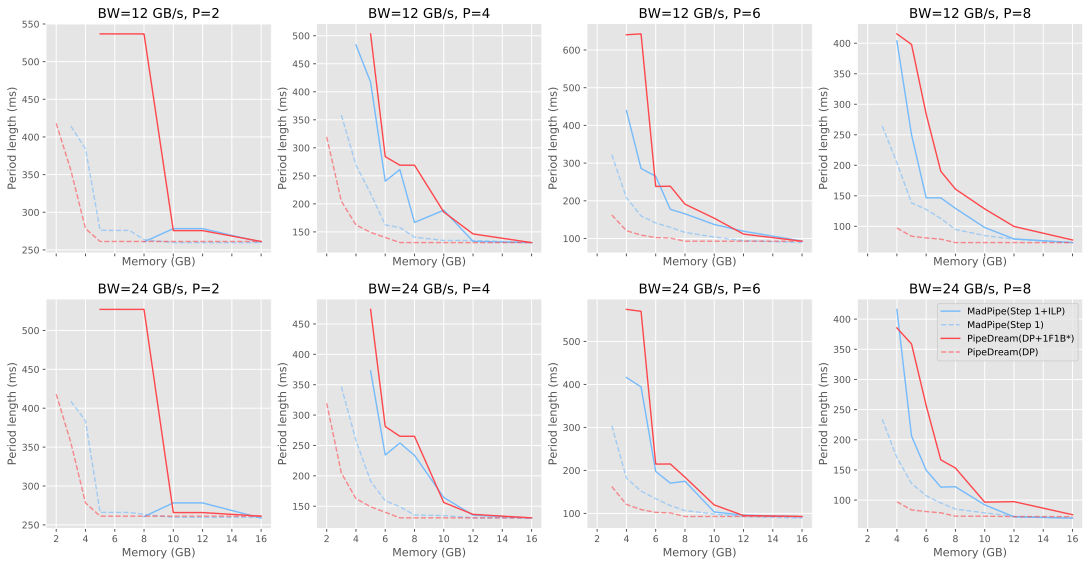


Рис. 7.7: Comparison of periods for DenseNet-121 with image size 1000 and mini-batch size 8.

Рис. 7.8: Geometric mean of ratios over different values of P and β, for all networks.

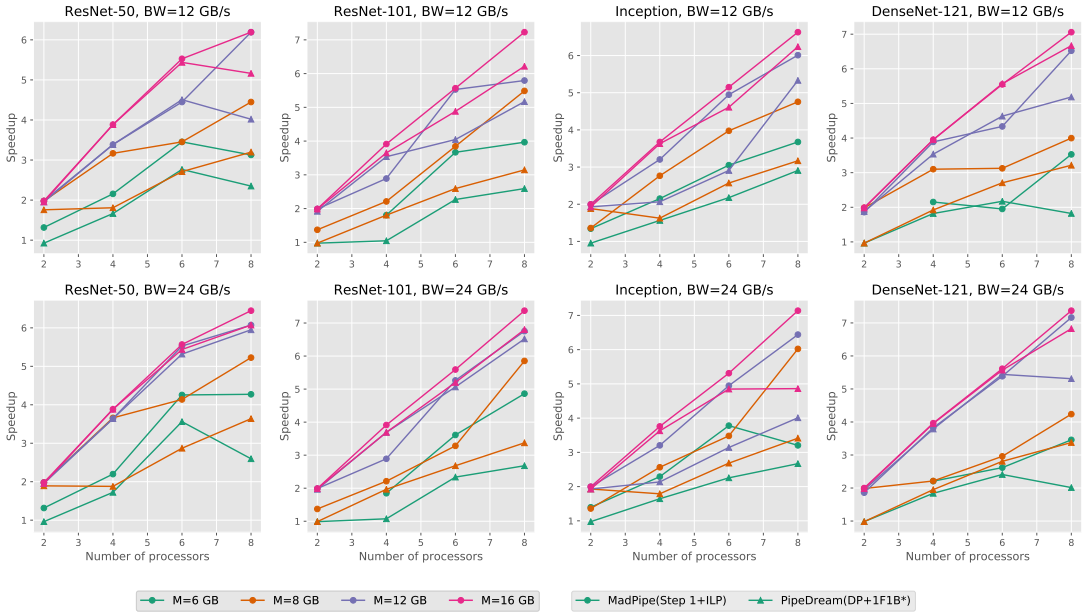


Рис. 7.9: Speedup comparison for all networks.

Conclusion

This work has addressed three different ways of reducing memory consumption during the training of neural networks: Rematerialization, Offloading and Pipelined Model Parallelism. Rematerialization and Offloading focus on lowering the memory usage of activations, while Pipelined Model Parallelism tackles all sources of memory problems.

Rematerialization as discussed in Part I is based on the checkpointing method from Automatic Differentiation community and solves the problem of executing forward-backward propagation graph while keeping only a limited number of activations in memory. It achieves memory reduction by recomputing the missing activations during the backward phase. The previous works on AD cover only simple homogeneous adjoint chain computations, yet neural networks require a more general approach adapted for DAGs and heterogeneous costs. We have offered two new extensions of classical approaches. Chapter 1 considers a multi-chain graph that appears in some neural networks, but the case is simplified by assuming homogeneous costs of operations. Chapter 2, alternatively, concentrates on heterogeneous costs for a simple chain structure. Nevertheless, the contribution of Chapter 2 is more general, since a large class of neural networks can be approximated with heterogeneous chains whose nodes can be arbitrarily complex. Both cases are expressed as optimization problems, to which optimal solutions are proposed. Furthermore, to meet the requirements of the learning frameworks such as PyTorch, the model of Chapter 2 reflects the additional dependencies imposed by autograd mechanism from PyTorch. It allows us to integrate the solutions from this chapter into a tool named ROTOR that can be directly used when training models in PyTorch. Our experimental evaluation on standard vision networks have shown that ROTOR finds significantly better solutions than its competitors in a reasonable time.

A further research direction for our rematerialization algorithms could be to extend them to general heterogeneous DAGs. Finding optimal solutions is complicated in this case. Indeed, it can be seen from the multi-chain case of Chapter 1 where the complexity of the problem grows exponentially with the number of branches. It is still possible to simplify the problem by narrowing it to a special sub-class of graphs that most neural network graphs belong to. For example, chains with skip-connections are common in vision networks, and thus finding the optimal solutions in this case could further increase the performance of ROTOR. Alternatively, proposing the best way of linearizing the graphs (representing DAGs in a sequential form) can profit from already implemented methods in ROTOR that are designed for heterogeneous chains.

Among other potential improvements is combining rematerialization approaches with

activation compression. This was done in Automatic Differentiation [58], but it should be applied carefully, as lossy compression risks degrading convergence. One interesting alternative to Rematerialization is to design neural networks in a reversible manner (*e.g.* RevNet [35]). Thus, if one has an activation of some layer ℓ, then it is possible to obtain activation of layer $\ell - 1$ by applying the inverse of its forward. This contrasts with classical rematerialization methods as the recomputations are now done in the reverse order that coincides with the direction of the backward pass, allowing us to keep only one activation all the time. If a neural network contains reversible blocks, then rematerialization algorithms can be adjusted to take them into account to perform the bi-directional recomputations making the approach even more flexible. Let us also note that the idea of trading computations with memory can be useful for other applications than DNN training and thus should be considered for other typical problems from HPC.

Offloading, which is covered in Part II, reduces memory usage in a different way. It sends part of the data to an external storage (usually a GPU transfers data to a CPU). In Chapter 3, the pure Offloading problem is considered, whereas in Chapter 4 its combination with Rematerialization is proposed. We have shown that the general problem of Offloading is NP-complete in the strong sense, though there exist some tractable relaxations, which can be solved optimally. We have proposed the optimal solutions based on dynamic programming, which have been integrated into ROTOR. New experiments have demonstrated that our solutions manage to bring together the best of both worlds and beat the previous state-of-the-art approaches.

Similarly to Rematerialization, our offloading solutions are limited to heterogeneous chains. Thus, further works to extend our results to general DAGs might be very beneficial, and it can be done following the same directions as proposed for Rematerialization above. In addition, Integer Linear Programming can be devised to take into account all underlying complex dependencies and constraints. Finally, applying offloading not only to activations, but also to weights and optimizer states can significantly increase the performance of the methods and their applicability. Thus, it should be also considered in a future work.

The final approach is Pipelined Model Parallelism studied in Part III. We have discovered a couple of limitations of the current state-of-the-art method PipeDream: contiguous allocations and 1-periodic schedules (see Chapter 5). For each limitation, we have also evaluated its potential impact on the throughput and memory usagee. As non-contiguous allocations can be arbitrarily better than contiguous allocations, finding optimal non-contiguous allocations is especially attractive. Thus, we have presented a way of solving optimally the load balancing and scheduling problem in the context of non-contiguous allocations based on an ILP in Chapter 6. In addition, we have proposed a relatively cheap heuristic called MadPipe relying on dynamic programming in Chapter 7. Both ILP and MadPipe appear to be more adaptive to tight memory limits and demonstrate twice better throughput than PipeDream allocations with an optimal 1-periodic schedule.

There are opportunities for further enhancement. Firstly, both ILP and MadPipe are not adapted for general DAGs and thus should be changed to take general data dependencies into account. Secondly, despite non-optimality of 1-periodic schedules,

they continue to dominate in the applications. Whereas, as it has been shown in this manuscript, schedules have a direct control over peak memory consumption. Therefore, designing optimal k-periodic schedules can be important in pursuit of further reduction in memory. Another promising direction is to combine Model Parallelism with Rematerialization and Offloading. It is already partially implemented in some methods like GPipe, but it is done in a straightforward way without solving the corresponding optimization problems. Furthermore, it is worth considering the optimization problems of combining different types of parallelisms such as Data Parallelism, Model Parallelism and Tensor Slicing together.

Overall, the global direction is to extend all the approaches mentioned before to deal with general DAGs and, eventually, combine everything in one unified framework. In order to do it efficiently, the optimization problems should be formulated and analyzed in order to find the best strategies. Finally, sometimes it is beneficial to consider unconventional ways of doing training as RevNet [35], for example, which stores only one activation at a time thanks to reversible blocks of neural networks. Going beyond the established methods might help to discover new ground-breaking approaches to deal with memory issues, and therefore enable data scientists, the end-users of these strategies, to train bigger, deeper and more accurate networks, without having to buy new computational resources that are expensive both in terms of money and carbon impact.

Литература

[1] Periodic checkpointing in pytorch, 2018. https://pytorch.org/docs/stable/checkpoint.html. (Pages 22, 31, 63, 76, and 79).

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association. (Pages 59 and 155).

[3] A Adcroft, JM Campin, S Dutkiewicz, C Evangelinos, D Ferreira, G Forget, B Fox-Kemper, P Heimbach, C Hill, E Hill, et al. Mitgcm user manual, 2008. (Pages 21).

[4] Antoine Affouard, Jean-Christophe Lombardo, Hervé Goëau, Pierre Bonnet, and Alexis Joly. Pl@ntNet, April 2019. (Pages 9 and 18).

[5] Guillaume Aupy and Julien Herrmann. Periodicity in optimal hierarchical checkpointing schemes for adjoint computations. *Optimization Methods and Software*, 32(3):594–624, 2017. (Pages 21).

[6] Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. Optimal multistage algorithm for adjoint computation. *SIAM Journal on Scientific Computing*, 38(3):232–255, 2016. (Pages 21, 25, 31, and 41).

[7] Shriram S B, Anshuj Garg, and Purushottam Kulkarni. Dynamic memory management for gpu-based training of deep neural networks. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Press, 2019. (Pages 6, 15, 24, 95, 97, 114, and 118).

[8] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Optimal gpu-cpu offloading strategies for deep neural network training. In *European Conference on Parallel Processing*, pages 151–166. Springer, 2020. (Pages 10 and 19).

[9] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Pipelined model parallelism: Complexity results and memory considerations. In *European Conference on Parallel Processing*, pages 183–198. Springer, 2021. (Pages 10 and 19).

[10] Olivier Beaumont, Julien Herrmann, Guillaume Pallez, and Alena Shilova. Optimal memory-aware backpropagation of deep join networks. *Philosophical Transactions of the Royal Society A*, 378(2166):20190049, 2020. (Pages 10 and 19).

[11] Olivier Beaumont, Julien Langou, Willy Quach, and Alena Shilova. A makespan lower bound for the tiled cholesky factorization based on alap schedule. In *European Conference on Parallel Processing*, pages 134–150. Springer, 2020. (Pages 10 and 19).

[12] Joan Boyar, Leah Epstein, and Asaf Levin. Tight results for next fit and worst fit with resource augmentation. *Theoretical Computer Science*, 411(26):2572 – 2580, 2010. (Pages 168).

[13] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a"siamese"time delay neural network. In *Advances in neural information processing systems*, pages 737–744, 1994. (Pages 22 and 38).

[14] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. (Pages 1 and 11).

[15] Phil Brubaker. *Engineering Design Optimization using Calculus Level Methods*. 2016. (Pages 21).

[16] Bruce G Buchanan. A (very) brief history of artificial intelligence. *Ai Magazine*, 26(4):53–53, 2005. (Pages 1 and 11).

[17] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. *IEEE Access*, 8:225134–225180, 2020. (Pages 2 and 12).

[18] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839*, 2018. (Pages 8, 17, 26, and 28).

[19] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proc. IEEE*, 107(8):1655–1674, 2019. (Pages 1 and 11).

[20] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. (Pages 4, 5, 14, 15, 22, 23, 25, 31, 59, and 63).

[21] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53(7):5113–5155, 2020. (Pages 3 and 13).

[22] Ching-Hsiang Chu, Pouya Kousha, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and Dhabaleswar K Panda. Nv-group: link-efficient reduction for distributed deep learning on modern dense gpu systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020. (Pages 155).

[23] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016. (Pages 7 and 16).

[24] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012. (Pages 8, 17, 25, and 155).

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. (Pages 1 and 11).

[26] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. Improving strong-scaling of cnn training by exploiting finer-grained parallelism. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2019. (Pages 7 and 16).

[27] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. Channel and filter parallelism for large-scale cnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 10. ACM, 2019. (Pages 7 and 16).

[28] William Du, Michael Fang, and Margaret Shen. Siamese convolutional neural networks for authorship verification. Proceedings, 2017. (Pages 22 and 38).

[29] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 381–396. USENIX Association, 2021. (Pages 27).

[30] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models, 2020. (Pages 27).

[31] Jianwei Feng and Dong Huang. Optimal gradient checkpoint search for arbitrary computation graphs, 2018. (Pages 5, 14, 22, and 23).

[32] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979. (Pages 64 and 161).

[33] Alexander L Gaunt, Matthew A Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. Ampnet: Asynchronous model-parallel training for dynamic neural networks. *arXiv preprint arXiv:1705.09786*, 2017. (Pages 7 and 16).

[34] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010. (Pages 155).

[35] Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 2211–2221, 2017. (Pages 206 and 207).

[36] Priya Goyal. Pytorch memory optimizations via gradient checkpointing, 2018. (Pages 23 and 79).

[37] Andreas Griewank. On automatic differentiation. *Mathematical Programming: Recent Developments and Applications*, 6(6):83–107, 1989. (Pages 31 and 59).

[38] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and software*, 1(1):35–54, 1992. (Pages 21).

[39] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000. (Pages 21, 22, 31, 37, 42, 55, 59, 73, and 139).

[40] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008. (Pages 21, 64, and 79).

[41] José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 95–106. SIAM, Philadelphia, PA, 1996. (Pages 21 and 42).

[42] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016. (Pages 22, 31, 59, 66, and 79).

[43] Lei Guan, Wotao Yin, Dongsheng Li, and Xicheng Lu. Xpipe: Efficient pipeline model parallelism for multi-gpu dnn training. *arXiv preprint arXiv:1911.04610*, 2019. (Pages 7, 17, 26, and 28).

[44] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE. (Pages 2, 5, 12, and 14).

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. (Pages 1 and 11).

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Computer Vision – ECCV 2016*, pages 630–645. Springer International Publishing, 2016. (Pages 5, 14, and 78).

[47] Julien Herrmann. H-revolve: a framework for adjoint computation on synchronous hierarchical platforms. *ACM Transactions on Mathematical Software (TOMS)*, 46(2):1–25, 2020. (Pages 22).

[48] HiePACS team. Rotor, 2019. (Pages 32, 76, 78, 79, 83, and 151).

[49] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020. (Pages 24).

[50] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019. (Pages 7, 8, 16, 17, 26, and 155).

[51] Arpan Jain, Ammar Ahmad Awan, Asmaa M Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G Anthony, Hari Subramoni, Dhableswar K Panda, Raghu Machiraju, and Anil Parwani. Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020. (Pages 26).

[52] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization, 2019. (Pages 5, 14, 15, 22, 23, 79, and 82).

[53] Enver Kayaaslan, Thomas Lambert, Loris Marchal, and Bora Uçar. Scheduling series-parallel task graphs to minimize peak memory. *Theoretical Computer Science*, 707:1–23, 2018. (Pages 31).

[54] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020. (Pages 5, 14, and 23).

[55] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014. (Pages 8 and 17).

[56] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012. (Pages 1 and 11).

[57] Navjot Kukreja, Jan Hückelheim, and Gerard J Gorman. Backpropagation for long sequences: beyond memory constraints with constant overheads. *arXiv preprint arXiv:1806.01117*, 2018. (Pages 21 and 22).

[58] Navjot Kukreja, Jan Hückelheim, Mathias Louboutin, Paul Hovland, and Gerard Gorman. Combining checkpointing and data compression to accelerate adjoint-based optimization problems. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 87–100, Cham, 2019. Springer International Publishing. (Pages 206).

[59] Navjot Kukreja, Alena Shilova, Olivier Beaumont, Jan Huckelheim, Nicola Ferrier, Paul Hovland, and Gerard Gorman. Training on the edge: The why and the how. In *1st Workshop on Parallel AI and Systems for the Edge, Rio de Janeiro, Brazil*, 2019. (Pages 3, 10, 12, and 19).

[60] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient rematerialization for deep networks. In *Advances in Neural Information Processing Systems*, pages 15146–15155, 2019. (Pages 5, 14, 22, and 23).

[61] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *arXiv preprint arXiv:1905.11722*, 2019. (Pages 5, 14, and 23).

[62] Youngeun Kwon and Minsoo Rhu. 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)*, pages 148–161. IEEE, 2018. (Pages 24).

[63] Hamed F Langroudi, Zachariah Carmichael, and Dhireesha Kudithipudi. Deep learning training on the edge with low-precision posits. *arXiv preprint arXiv:1907.13216*, 2019. (Pages 3 and 12).

[64] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting. *arXiv preprint arXiv:1807.02037*, 2018. (Pages 6, 15, 24, 114, and 118).

[65] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989. (Pages 1 and 11).

[66] Jiange Li, Yuchen Wang, Jinghui Zhang, Jiahui Jin, Fang Dong, and Lei Qian. Pipepar: A pipelined hybrid parallel approach for accelerating distributed dnn training. In *2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 470–475. IEEE, 2021. (Pages 27).

[67] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *arXiv preprint arXiv:2002.11794*, 2020. (Pages 2 and 12).

[68] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. *arXiv preprint arXiv:2102.07988*, 2021. (Pages 27).

[69] Wei Yang Bryan Lim, Nguyen Cong Luong, Dinh Thai Hoang, Yutao Jiao, Ying-Chang Liang, Qiang Yang, Dusit Niyato, and Chunyan Miao. Federated learning in mobile edge networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3):2031–2063, 2020. (Pages 3 and 12).

[70] Deyin Liu, Xu Chen, Zhi Zhou, and Qing Ling. Hiertrain: Fast hierarchical edge ai learning with hybrid parallelism in mobile-edge-cloud computing. *IEEE Open Journal of the Communications Society*, 1:634–645, 2020. (Pages 3 and 12).

[71] J. Liu, , W. Yu, J. Wu, D. Buntinas, , D. K. Panda, and P. Wyckoff. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*, 24(1):42–51, Jan 2004. (Pages 155).

[72] Joseph WH Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM Journal on Algebraic Discrete Methods*, 8(3):375–395, 1987. (Pages 31).

[73] Yingchi Mao, Zijian Tu, Fagang Xi, Qingyong Wang, and Shufang Xu. Tapp: Dnn training for task allocation through pipeline parallelism based on distributed deep reinforcement learning. *Applied Sciences*, 11(11):4785, 2021. (Pages 27).

[74] Javier Marin, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Salvador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba. Recipe1m: A dataset for learning cross-modal embeddings for cooking recipes and food images. *arXiv preprint arXiv:1810.06553*, 2018. (Pages 22 and 38).

[75] Jonathan Masci, Davide Migliore, Michael M Bronstein, and Jürgen Schmidhuber. Descriptor learning for omnidirectional image matching. In *Registration and Recognition in Images and Videos*, pages 49–62. Springer, 2014. (Pages 22 and 38).

[76] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Proc. of ML Systems Workshop in NIPS*, 2017. (Pages 24 and 118).

[77] M. Mueller, A. Arzt, S. Balke, M. Dorfer, and G. Widmer. Cross-modal music retrieval and applications: An overview of key methodologies. *IEEE Signal Processing Magazine*, 36(1):52–62, Jan 2019. (Pages 22 and 38).

[78] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019. (Pages 7, 8, 16, 17, 26, 155, 161, 164, 166, 167, 170, 185, and 196).

[79] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7937–7947. PMLR, 18–24 Jul 2021. (Pages 7, 17, 26, 28, 155, 159, and 183).

[80] Stefan Nickel, Claudius Steinhardt, Hans Schlenker, Wolfgang Burkart, and Melanie Reuter-Oppermann. Ibm ilog cplex optimization studio. In *Angewandte Optimierung mit IBM ILOG CPLEX Optimization Studio*, pages 9–23. Springer, 2020. (Pages 185).

[81] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism, 2020. (Pages 27).

[82] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch, 2017. (Pages 31, 59, and 155).

[83] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q Weinberger. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990*, 2017. (Pages 22 and 31).

[84] GC Pringle, DC Jones, S Goswami, SHK Narayanan, and D Goldberg. Providing the archer community with adjoint modelling tools for high-performance oceanographic and cryospheric computation. 2016. (Pages 21).

[85] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*, 2020. (Pages 6, 15, and 25).

[86] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020. (Pages 2, 3, 12, 13, 25, 159, and 183).

[87] Jinke Ren, Guanding Yu, and Guangyao Ding. Accelerating dnn training in wireless federated edge learning systems. *IEEE Journal on Selected Areas in Communications*, 39(1):219–232, 2020. (Pages 3 and 12).

[88] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 18. IEEE Press, 2016. (Pages 6, 15, 24, 25, 95, 97, and 118).

[89] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018. (Pages 24).

[90] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. (Pages 1 and 11).

[91] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. In-place activated batchnorm for memory-optimized training of dnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5639–5647, 2018. (Pages 22 and 31).

[92] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. (Pages 1 and 11).

[93] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002. (Pages 1 and 11).

[94] Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver nek5000. *Procedia Computer Science*, 80:1147–1158, 2016. (Pages 21).

[95] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green ai. *Commun. ACM*, 63(12):54–63, November 2020. (Pages 2 and 12).

[96] Ravi Sethi. Complete register allocation problems. *SIAM journal on Computing*, 4(3):226–248, 1975. (Pages 31).

[97] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. (Pages 1 and 11).

[98] Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, 2018. (Pages 22).

[99] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017. (Pages 5, 7, 14, and 16).

[100] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631*, 2019. (Pages 5 and 14).

[101] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019. (Pages 2 and 12).

[102] Philipp Stumm and Andrea Walther. Multistage approaches for optimal offline checkpointing. *SIAM Journal on Scientific Computing*, 31(3):1946–1967, 2009. (Pages 21 and 37).

[103] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017. (Pages 38).

[104] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. (Pages 38).

[105] Zeyi Tao and Qun Li. esgd: Communication efficient distributed deep learning on the edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018. (Pages 3 and 12).

[106] Jakub Tarnawski, Amar Phanishayee, Nikhil R Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators. *arXiv preprint arXiv:2006.16423*, 2020. (Pages 7, 17, and 27).

[107] Andrea Walther. *Program reversal schedules for single-and multi-processor machines*. PhD thesis, PhD thesis, Institute of Scientific Computing, Technical University Dresden, Germany, 1999. (Pages 66).

[108] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. *SIGPLAN Not.*, 53(1):41–53, February 2018. (Pages 25, 118, and 147).

[109] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019. (Pages 2 and 12).

[110] Weizheng Xu, Youtao Zhang, and Xulong Tang. *Parallelizing DNN Training on GPUs: Challenges and Opportunities*, page 174–178. Association for Computing Machinery, New York, NY, USA, 2021. (Pages 8 and 17).

[111] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3, 2021. (Pages 26 and 28).

[112] Yang You, Zhao Zhang, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Imagenet training in 24 minutes. 09 2017. (Pages 155).

[113] Jun Zhan and Jinghui Zhang. Pipe-torch: Pipeline-based distributed deep learning in a gpu cluster with heterogeneous networking. In *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*, pages 55–60. IEEE, 2019. (Pages 7, 17, 27, and 155).

[114] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *arXiv preprint arXiv:1903.06631*, 2019. (Pages 6, 15, 24, 114, and 118).

[115] Shanshan Zhang, Ce Zhang, Zhao You, Rong Zheng, and Bo Xu. Asynchronous stochastic gradient descent for dnn training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6660–6663. IEEE, 2013. (Pages 8 and 17).

[116] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010. (Pages 7, 16, and 155).