Optimizing Memory Usage when Training Deep Neural Networks

Xunyi Zhao

▶ **To cite this version:**

Xunyi Zhao. Optimizing Memory Usage when Training Deep Neural Networks. Computer Science [cs]. Université de Bordeaux, 2024. English. NNT : 2024BORD0411 . tel-04890912

HAL Id: tel-04890912

https://theses.hal.science/tel-04890912v1

Submitted on 16 Jan 2025

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.

LaBRI

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Xunyi ZHAO**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Optimisation de la consommation mémoire pendant l'apprentissage

Optimizing Memory Usage when Training Deep Neural Networks

Sous la direction de : Olivier BEAUMONT

Date de soutenance : 2024/12/10

Devant la commission d'examen composée de :

Kim Thang NGUYEN	Professeur des universités, University of Grenoble	Rapporteur
Anne BENOIT	Maîtresse de conférences, ENS Lyon	Rapporteure
Denis BARTHOU	Directeur de recherche, Huawei Paris Research Center	Président
Alena SHILOVA	Docteure, Inria Lille	Examinatrice
Aurélien FROGER	Chargé de recherche, Université de Bordeaux	Examinateur
Olivier BEAUMONT	Directeur de recherche, Inria Bordeaux	Directeur de thèse

Membres invités :

Lionel EYRAUD-DUBOIS	Chargé de Recherches, Inria Bordeaux	invité
Julia GUSAK	Maîtresse de conférences, Inria Bordeaux	invité

– 2024 –

Titre Optimisation de la consommation mémoire pendant l'apprentissage

Résumé L'intelligence artificielle (IA) a connu une croissance remarquable ces dernières années, prouvant son utilité dans un large éventail de domaines, notamment la reconnaissance d'images, le traitement du langage naturel et les systèmes autonomes. Ce succès est largement dû à l'accès à des jeux de données toujours plus vastes et au développement de réseaux neuronaux profonds (Deep Neural Networks, DNNs) de taille et de complexité toujours plus importantes, permettant aux systèmes d'IA d'atteindre des niveaux de performance sans précédent. Cependant, la taille croissante des réseaux soulève des défis importants, en particulier lorsqu'il s'agit d'entraîner les modèles gigantesques sur des ressources de calcul dont la mémoire est limitée. La gestion efficace de la mémoire pendant l'apprentissage est devenue une question cruciale pour garantir que les systèmes à ressources limitées sont en mesure de réaliser l'apprentissage de gros modèles d'IA.

Il existe plusieurs stratégies pour gérer les contraintes de mémoire lors de l'entraînement des réseaux neuronaux. L'empreinte mémoire peut être distribuée sur plusieurs ressources ou compressée à l'aide d'algorithmes spécialisés qui minimisent la perte d'information. Cette thèse se concentre sur les techniques de réduction de la mémoire sans perte, principalement à des scénarios avec une seule ressource de calcul. Les approches clés comprennent d'une part la réduction de l'empreinte mémoire des activations intermédiaires en en supprimant certaines et en les recalculant lorsque nécessaire et d'autre part la réduction de la mémoire utilisée pour les paramètres en les transférant vers et depuis une mémoire RAM de plus grande capacité. Nous avons développé des algorithmes d'optimisation qui intègrent ces techniques, réduisant efficacement le pic de mémoire tout en maintenant des temps d'itération d'entraînement efficaces. Nos solutions sont mises en œuvre dans un environnement open-source, testé et compatible avec les principales bibliothèques d'IA telles que PyTorch, HuggingFace et DeepSpeed.

Mots-clés apprentissage, mémoire, IA

Title Optimizing memory usage when training Deep Neural Networks

Abstract Artificial Intelligence (AI) has seen remarkable growth in recent years, proving its utility across a wide range of fields including image recognition, nature language processing and autonomous systems. This success is largely driven by access to increasingly large datasets and the development of Deep Neural Networks (DNNs) with greater complexity and size, allowing AI systems to achieve unprecedented levels of performance. However, the growing scale of tasks presents significant challenges, particularly when it comes to training these massive models on devices with limited memory capacity. Efficiently managing memory during training has become a critical focus to ensure that even resource-constrained systems can handle complex AI tasks.

There are several strategies to address memory constraints in neural network training. Memory footprints can be distributed across multiple devices or compressed using specialized algorithms that minimize information loss. This thesis focuses on lossless memory-saving techniques, primarily applied to single-device scenarios. The key approaches include reducing the memory cost of intermediate activations by discarding and recomputing them when required, and managing parameter memory costs by swapping them to larger capacity RAM. We have developed optimization algorithms that integrate these techniques, effectively lowering peak memory usage while maintaining efficient training iteration times. Our solutions are implemented in an open-source framework, tested and compatible with leading AI libraries such as PyTorch, HuggingFace, and DeepSpeed.

Keywords training, memory, AI

Laboratoire d'accueil Topal team, Inria Bordeaux - Sud-Ouest, 33405 Talence, FRANCE.

Acknowledgement

Working on my thesis has been an incredible journey, one that exceeded my expectations and brought immense personal and professional growth. Along the way, I have not only expanded my knowledge but also enjoyed life to the fullest.

I feel incredibly fortunate to have had two exceptional advisors guiding me. From the very first day, Olivier Beaumont has been kind, supportive, and insightful. He offered valuable guidance in shaping the direction of my research while granting me the freedom to explore topics that piqued my interest. Lionel Eyraud-Dubois, on the other hand, has been instrumental in sharpening my skills in programming, mathematics, and the broader art of conducting research. I will always cherish our daily-base discussions, where new ideas evolved into something meaningful. Their mentorship has been pivotal in my development, allowing me to confidently approach all aspects of independent research by the end of my PhD.

I would like to express my heartfelt gratitude to Théotime Le Hellard, whose contributions enriched my thesis in countless ways. What started as an eight-week internship turned into an extraordinary collaboration. Together, we formed a seamless team, often describing our weekly achievements as "satisfying." Even after his internship, Théotime continued to work with us remotely and successfully created rk-GB —a long-standing team goal that had seemed elusive. His ingenuity and dedication not only inspired me to be more ambitious but also significantly elevated the quality of my work. My PhD experience would not have been as fulfilling and transformative without his partnership.

Beyond academia, I have enjoyed the life I built during these three years. I had the privilege of exploring over 20 countries and forming lasting friendships along the way. Bordeaux, with its relaxed and welcoming atmosphere, has been the perfect backdrop to this chapter of my life. Unlike the often-heard tales of PhD-related stress, I felt surrounded by a supportive and easygoing community.

To my friends and colleagues—Angela, Bhavesh, Bilal, Christos, Maryam, Alena, Esra, Martina, Yanfei, Mathieu, Romain, Marek, Jean-François, Gilles, Alexandre, Antoine, Brieuc, Adrien, Hugo, Hayfa, and Alicia—thank you for your companionship and encouragement. Finally, I owe a special thanks to my family and friends in China, whose unwavering support has been a constant source of strength.

Contents

Summaries 1
- Background 1
- Contribution 2

Résumé long 5
- Contexte 5
- Contribution 6

1 Context 9
- 1.1 Backgrounds 9
  - 1.1.1 Artificial Intelligence (AI) 9
  - 1.1.2 Tasks and Accelerators 9
  - 1.1.3 Memory requirement 11
- 1.2 Related Works 12
  - 1.2.1 Training Simplification 13
  - 1.2.2 Parallelism 14
  - 1.2.3 Re-materialization 16
  - 1.2.4 Offloading 17
  - 1.2.5 Software 18
- 1.3 Operations in PyTorch Modules 19
  - 1.3.1 Dependencies between Operations 20
  - 1.3.2 Execution of Operations 21

2 Rockmate 25
- 2.1 Introduction 25
  - 2.1.1 Checkmate 25
  - 2.1.2 Rotor 27
- 2.2 Algorithm 30
  - 2.2.1 Sketch of the Algorithm 30
- 2.3 rk-GB, Graph Builder 32
- 2.4 rk-Checkmate 34
  - 2.4.1 Optimization Problem 35
  - 2.4.2 Feasibility Constraints 37

2.4.3 Memory Constraints . . . 38
2.4.4 Schedule Construction . . . 40
2.5 RK-ROTOR . . . 40
2.5.1 Formulation Notations . . . 42
2.5.2 Algorithms . . . 42
2.5.3 DP Formulation . . . 43
2.6 Performance analysis . . . 45
2.7 Experiments . . . 46
2.7.1 Budgets Selection . . . 46
2.7.2 Performance vs. Solving Time . . . 47
2.7.3 Performance on Different Networks . . . 48
2.8 Conclusion . . . 51

3 HIREMATE **52**
3.1 Introduction . . . 52
3.2 HIREMATE . . . 54
3.2.1 Partition Framework . . . 55
3.2.2 Solving Framework . . . 57
3.3 H-Partition Algorithm . . . 60
3.3.1 Sketch of Partitioning Algorithm . . . 60
3.3.2 Convexity . . . 62
3.3.3 Candidate Groups . . . 62
3.3.4 Update of Candidates . . . 63
3.3.5 Identification of Identical `H-Clusters` . . . 64
3.3.6 Complexity Analysis of H-Partition . . . 64
3.4 H-ILP Hierarchical Formulation . . . 64
3.4.1 Context . . . 64
3.4.1.1 Compute options and phantom nodes . . . 65
3.4.1.2 Note about input dependencies . . . 66
3.4.2 Formulation . . . 66
3.4.2.1 Variables . . . 67
3.4.2.2 Objective . . . 67
3.4.3 Validity Constraints of H-ILP . . . 67
3.4.3.1 Boundary constraints . . . 68
3.4.3.2 Data dependencies . . . 68
3.4.3.3 Options-related valid constraints . . . 69
3.4.3.4 Alive status of values . . . 69
3.4.3.5 Constraints related to liveness . . . 69
3.4.4 Memory Usage Constraints . . . 70
3.4.4.1 Correction Terms . . . 71
3.4.5 Option Selection . . . 72
3.4.6 Complexity Analysis of H-ILP . . . 73
3.5 Experimental Evaluation . . . 73

3.5.1 Visualization of Partition . . . 74
3.5.2 Efficiency . . . 78
3.5.3 Performance . . . 79
3.5.4 Ablation Study . . . 79
3.6 Discussion and Conclusion . . . 82

4 OFFMATE **83**
4.1 Introduction . . . 83
4.2 Motivation . . . 84
4.2.1 Memory Requirement . . . 84
4.2.2 Memory Efficient Solutions . . . 85
4.2.3 Complexity . . . 87
4.3 Method . . . 89
4.3.1 Our Approach . . . 89
4.3.2 High Complexity . . . 90
4.3.3 Framework . . . 91
4.3.4 Assumptions . . . 92
4.4 Preparing the *blocks* . . . 94
4.5 O-ILP . . . 94
4.5.1 Optimization Problem . . . 94
4.5.2 Linear Programming Formulation . . . 95
4.5.3 Adaptation: Batch Size Selection . . . 97
4.6 Post-processing . . . 98
4.7 Additional Improvements . . . 100
4.7.1 CPU Management . . . 100
4.7.2 Grouping of CPU Memory Allocations . . . 101
4.8 Experiments . . . 101
4.8.1 Fine-tuning Tasks . . . 102
4.8.2 Execution Tracing . . . 103
4.8.3 Batch Size . . . 103
4.8.4 Ablation Study . . . 104
4.9 Conclusion . . . 106

Conclusion **108**

List of Algorithms

1 Compute optimal persistent schedule for a chain of length L with memory M. 29
2 OptRec(C, s, t, m) – Obtain optimal persistent sequence from the table C . 29
3 Rockmate 32
4 Construction of schedule based on the rk-Checkmate solution 41
5 rk-Rotor for L *blocks* with memory m. 44
6 rk-Rotor-Build(Opt, s, t, m) – Computation of the schedule 44
7 H-Partition algorithm 61
8 Greedy grouping for parameter w 99

Summaries

Background

Deep neural networks (DNNs) have been found useful in various tasks, including image recognition, natural language processing, and autonomous systems. Thanks to the development of computer hardware, more and more complicated architectures of DNNs have been designed and trained with large amounts of data. There are different ways to pattern in the data set learned by a neural network model, of which the most widely used is forward and backward propagation. The procedure of forward and backward propagation is as follows: (1) A batch of data samples is passed through the neural networks that make predictions (called forward pass). (2) A predefined loss function is applied to the predictions to measure the quality of the neural network training. (3) A sequence of computations (called the backward pass) according to the chain rule is used to obtain the gradient of each parameter in the network with respect to the loss. (4) The parameters are updated based on the gradients according to various optimization algorithms. Although recent works have demonstrated that the pure forward method can also train neural networks effectively, the forward-backward propagation method is still the most widely applied method for training a neural network. Due to the high computational demands of forward and backward propagation, the training process is often deployed on Graphics Processing Units (GPUs). In a standard training task of AI models, all data related to the training task is stored in VRAM and swapped between processing cores for computation.

In recent years, DNNs of unprecedented size have been trained and even open-sourced, such as GPT [44], Bloom [60], Llama [56], and Mixtral [27]. A major challenge in training these networks is the dramatic increase in their memory requirements. A typical GPU can have a VRAM size between 2GB and 160GB, but a typical large model training can require 100-1000 GB of memory. Therefore, multiple GPUs are often used for large model training to distribute the memory footprint across multiple devices. When fewer computing resources are available, networks trained on general datasets can often be efficiently fine-tuned for specific tasks. The computational requirements of fine-tuning are typically much lower than training a network from scratch for general tasks, while the memory requirements remain at a similar level. Two primary solutions are widely used in the AI community to address memory constraints in neural network training are parallelism and model compression. Parallelism distributes the memory footprint across multiple devices, significantly reducing the memory load on each device.

However, this approach requires the availability of multiple devices, which may not always be practical. Alternatively, the memory footprint can be compressed by using less precise representations for computations. Techniques such as quantization or matrix decomposition, as pioneered by LoRA [20], compress large matrices without sacrificing too much information. However, these methods inevitably lead to some loss of precision, which can affect the training results, even if the accuracy loss is minimal.

Given these limitations, our work explores an alternative approach to maximize applicability without compromising model performance. Our objective is to recompile AI models in a way that alleviates memory constraints without altering the final output or relying on multiple devices. We focus on two key techniques:

- Re-materialization: This technique reduces memory consumption by discarding selected activations during the forward pass and recomputing them as needed during the backward pass. This effectively reduces the number of tensors stored in memory.
- Offloading: Using CPU memory, which is generally larger and less expensive than GPU memory, we offload selected data to reduce GPU memory contention.
- In addition, other strategies, such as optimizing computation directly on the CPU, are being considered to further improve memory efficiency.

Contribution

In this thesis, we present works to efficiently reduce the memory requirement in deep neural network training. Our approach targets the static model, which is optimized by forward-backward propagation. We also demonstrate the performance of our work on a single GPU, although it can be extended to data parallelism across different devices. Our work can be used in the following way: given a PyTorch Module and a sample input, an efficient algorithm is applied to recompile the module so that a new module can produce the exact same result while keeping memory usage under a given threshold at the price of additional training time. Under different training scenarios, our algorithms aim to minimize the time cost of a training iteration. Our work is built on top of PyTorch, and we have made the framework compatible with some of the most popular libraries such as HuggingFace, DeepSpeed, and PEFT. Our goal is to provide a seamless experience for researchers and engineers who want to train deep neural networks with limited resources.

Rockmate Our first work focuses on providing automated re-materialization solutions tailored to available memory resources. Given a PyTorch `nn.Module` and a sample input, Rockmate automatically recompiles it into a new `nn.Module` that produces exactly the same results while keeping the peak memory usage within a given limit. To minimize the recomputation time required for re-materialization, we combine the algorithms proposed in Checkmate [25] and Rotor [3]. The Rockmate algorithm can thus be efficiently applied to most popular model architectures, such as GPTs [44]. Rockmate works by decomposing the model into a sequence of *blocks*, where each *block* is solved by an adapted version of Checkmate to provide multiple options for re-materialization. These

options are further combined in an adapted version of Rotor, which finally generates the re-materialization schedule of the whole model. In our experiments, we test models with large training batches running on NVIDIA GPUs to measure real-world training iteration time. We show that Rockmate can significantly reduce the memory peak while adding negligible training time in real-world tasks. The work of Rockmate is presented in Chapter 2. For certain types of models such as GPTs, Rockmate can efficiently provide re-materialization solutions that reduce memory requirements without adding significant iteration time. It works well for models that can be represented in a sequence of *blocks* where each *block* is not too large, making GPTs the perfect targets. However, Rockmate has limitations when applied to models with more complex or differently structured architectures. For a wider range of architectures that do not fit this pattern, we present our next work.

Hiremate While Rockmate works efficiently on GPTs, it may not work on other architectures that cannot be represented as a sequence of small *blocks*. In these cases, generating multiple options for large *blocks* in Rockmate can be excessively time-consuming. We propose our second work, Hiremate, to address this limitation. For a model with arbitrarily complex architectures, a recursive partitioning algorithm is applied to decompose its graph representation into a hierarchical structure of graphs: each graph contains only a limited number of nodes, where each node can represent a subgraph with a large set of operations. We further develop a new ILP solver, H-Ilp, which generates a re-materialization schedule given the graph and memory limits. Unlike Checkmate, H-Ilp accepts that each node can be executed in multiple ways corresponding to the solutions of the subgraph it represents. As the subgraphs are being solved recursively, their re-materialization plans are recursively combined in a bottom-up approach, eventually producing a solution for the entire model. Hiremate includes not only H-Ilp, but also other solvers, including Checkmate and Rotor, to provide solutions when the target graph has a desirable structure. We again demonstrate the solution efficiency and re-materialization performance of H-Ilp with experiments. This work is presented in chapter 3. We consider Hiremate to be the ultimate solution to the re-materialization problem, since it includes various algorithms and has no restrictions on model architectures. However, while re-materialization solutions reduce the memory footprint of activations, model parameters can still become a bottleneck. This issue is beyond the capabilities of Hiremate and is addressed in our next work.

Offmate To further reduce the memory footprint of the various components in large model training, we propose Offmate, a solution specifically designed to reduce parameter-related memory costs. Several approaches, including offloading and CPU optimization, are integrated into Offmate, building upon re-materialization strategies introduced in previous work. Since Offmate extends the range of possible operations during scheduling, the optimization algorithm suffers from high complexity. To address this, we introduce a number of assumptions to simplify the problem while preserving the most impactful options in the solution. As a final product of our thesis, Offmate allows

the fine-tuning of a 7 billion parameter Llama [56] model on a 12GB machine, achieving a 10× memory reduction at only a 20% overhead in training time. This makes it possible for individual researchers and AI enthusiasts using consumer GPUs to fine-tune large models without having access to high-end hardware. The performance and effectiveness of OFFMATE is detailed in Chapter 4.

All source code is available as open source in the git repository `https://github.com/topal-team/rockmate`.

Résumé long

Contexte

Les réseaux neuronaux profonds (DNN en Anglais) se sont révélés utiles dans diverses tâches, notamment la reconnaissance d'images, le traitement du langage naturel et les systèmes autonomes. Grâce au développement des équipements informatiques, des architectures de plus en plus complexes de réseaux neuronaux profonds ont été conçues et entraînées avec de grandes quantités de données. Il existe différentes façons de former des motifs dans l'ensemble de données utilisé pour l'apprentissage d'un modèle de réseau neuronal, dont la plus répandue est la propagation avant et arrière. La procédure de propagation vers l'avant et vers l'arrière est la suivante : (1) Un lot d'échantillons de données passe par les réseaux neuronaux qui font des prédictions (appelé « forward pass », ou "passe avant"). (2) Une fonction de perte ("loss function") prédéfinie est appliquée aux prédictions pour mesurer la qualité de l'apprentissage du réseau neuronal. (3) Une séquence de calculs (appelée "backward pass" ou "passe arrière") selon la règle de la chaîne (chain rule) est utilisée pour obtenir le gradient de chaque paramètre du réseau par rapport à la perte. (4) Les paramètres sont mis à jour sur la base des gradients selon divers algorithmes d'optimisation. Bien que des travaux récents aient démontré qu'une méthode s'appuyant uniquement sur la passe avant peut également permettre d'entraîner des réseaux neuronaux de manière efficace, la méthode de propagation avant-arrière reste la méthode la plus largement appliquée pour l'entraînement d'un réseau neuronal. En raison des exigences de calcul élevées liées à la propagation avant et arrière, le processus de formation est souvent déployé sur des unités de traitement graphique (GPU). Dans une tâche d'apprentissage standard de modèles d'IA, toutes les données liées à la tâche de formation sont stockées dans la VRAM et échangées entre les cœurs de traitement pour le calcul.

Ces dernières années, des réseaux DNN d'une taille sans précédent ont été appris et même mis en libre accès, comme GPT [44], Bloom [60], Llama [56], et Mixtral [27]. L'un des principaux défis de l'apprentissage pour ces réseaux est l'augmentation considérable de leurs besoins en mémoire. Un GPU typique peut avoir une taille de VRAM comprise entre 2 et 160 Go, mais la formation d'un grand modèle typique peut nécessiter 100 à 1000 Go de mémoire. C'est pourquoi plusieurs GPU sont souvent utilisés pour l'apprentissage de grands modèles afin de répartir l'empreinte mémoire sur plusieurs périphériques. Lorsque moins de ressources informatiques sont disponibles, les réseaux formés sur des ensembles

généraux de données peuvent souvent être affinés de manière efficace pour des tâches spécifiques. Les exigences de calcul de cet affinage sont généralement beaucoup plus faibles que la formation d'un réseau à partir de zéro pour des tâches générales, tandis que les exigences de mémoire restent à un niveau similaire. Deux solutions principales sont largement utilisées dans la communauté de l'IA pour répondre aux contraintes de mémoire dans la formation des réseaux neuronaux : le parallélisme et la compression des modèles. Le parallélisme répartit l'empreinte mémoire sur plusieurs périphériques, ce qui réduit considérablement la charge de mémoire sur chaque périphérique. Toutefois, cette approche nécessite de disposer de plusieurs périphériques, ce qui n'est pas toujours possible. Il est également possible de réduire l'empreinte mémoire en utilisant des représentations moins précises pour les données et les calculs. Des techniques telles que la quantification ou la décomposition matricielle, mises au point par LoRA [20], permettent de comprimer de grandes matrices sans sacrifier trop d'informations. Toutefois, ces méthodes entraînent inévitablement une perte de précision qui peut affecter les résultats de l'apprentissage, même si la perte de précision est minime.

Compte tenu de ces limitations, notre travail explore une approche alternative pour maximiser l'applicabilité sans compromettre la performance du modèle. Notre objectif est de recompiler les modèles d'IA d'une manière qui atténue les contraintes de mémoire sans altérer le résultat final ou dépendre de plusieurs périphériques. Nous nous concentrons sur deux techniques clés :

- Re-matérialisation : Cette technique réduit la consommation de mémoire en éliminant certaines activations au cours de la passe avant et en les recalculant selon les besoins au cours de la passe arrière. Cela permet de réduire efficacement le nombre de tenseurs stockés en mémoire.
- Délestage : En utilisant la mémoire du CPU, qui est généralement plus grande et moins chère que la mémoire du GPU, nous déchargeons les données sélectionnées pour réduire la contention de la mémoire du GPU.
- D'autres stratégies, telles que l'optimisation directe des calculs sur le CPU, sont également envisagées pour améliorer encore l'efficacité de la mémoire.

Contribution

Dans cette thèse, nous présentons des travaux visant à réduire efficacement les besoins en mémoire dans le cadre de de l'apprentissage des réseaux neuronaux profonds. Notre approche cible un modèle statique, qui est optimisé par la propagation avant-arrière. Nous démontrons également la performance de notre travail sur un seul GPU, bien qu'il puisse être étendu au parallélisme de données sur différents périphériques. Notre travail peut être utilisé de la manière suivante : étant donné un module PyTorch et un échantillon d'entrée, un algorithme efficace est appliqué pour recompiler le module de sorte qu'un nouveau module puisse produire exactement le même résultat tout en maintenant l'utilisation de la mémoire en dessous d'un seuil donné au prix d'un temps d'apprentissage accru. Dans différents scénarios d'apprentissage, nos algorithmes cherchent à minimiser le coût

en temps d'une itération d'apprentissage. Notre travail s'appuie sur PyTorch, et nous avons rendu le cadre compatible avec certaines des bibliothèques les plus populaires telles que HuggingFace, DeepSpeed et PEFT. Notre objectif est de fournir une expérience transparente aux chercheurs et aux ingénieurs qui souhaitent entraîner des réseaux neuronaux profonds avec des ressources limitées.

ROCKMATE Notre premier travail se concentre sur la mise à disposition de solutions de rematérialisation automatisées et adaptées aux ressources mémoire disponibles. Étant donné un PyTorch `nn.Module` et un échantillon d'entrée, ROCKMATE le recompile automatiquement dans un nouveau `nn.Module` qui produit exactement les mêmes résultats tout en maintenant le pic d'utilisation de la mémoire dans une limite donnée. Pour minimiser le temps de recalcul nécessaire à la rematérialisation, nous combinons les algorithmes proposés dans CHECKMATE [25] et ROTOR [3]. L'algorithme ROCKMATE peut donc être appliqué efficacement à la plupart des architectures de modèles populaires, telles que GPTs [44].

ROCKMATE fonctionne en décomposant le modèle en une séquence de *blocks*, où chaque *block* est géré par une version adaptée de CHECKMATE afin de fournir de multiples options de rematérialisation. Ces options sont ensuite combinées dans une version adaptée de ROTOR, qui génère finalement l'ordonnnacement des opératios de rematérialisation de l'ensemble du modèle. Dans nos expériences, nous testons les modèles avec de grands lots de formation fonctionnant sur des GPU NVIDIA afin de mesurer le temps d'itération de l'apprentissage dans le monde réel. Nous montrons que ROCKMATE peut réduire de manière significative le pic de mémoire tout en ajoutant un temps d'apprentissage négligeable dans les tâches réelles. Le travail de ROCKMATE est présenté dans le chapitre 2. Pour certains types de modèles tels que les GPT, ROCKMATE peut fournir efficacement des solutions de re-matérialisation qui réduisent les besoins en mémoire sans ajouter un temps d'itération significatif. Il fonctionne bien pour les modèles qui peuvent être représentés sous la forme d'une séquence de *blocks* où chaque *block* n'est pas trop grand, ce qui fait des modèles comme GPT des cibles parfaites. Cependant, ROCKMATE a des limites lorsqu'il est appliqué à des modèles ayant des architectures plus complexes ou structurées différemment. Nous présentons ci-après nos prochains travaux portant sur un plus large éventail d'architectures qui ne correspondent pas à ce modèle.

HIREMATE Si ROCKMATE fonctionne efficacement sur les GPT, il peut ne pas fonctionner sur des architectures qui ne peuvent pas être représentées sous la forme d'une séquence de petits *blocks*. Dans de tels cas, la génération de multiples options pour les grands *blocks* dans ROCKMATE peut en effet prendre beaucoup de temps. Nous proposons notre deuxième contribution, HIREMATE, pour répondre à cette limitation. Pour un modèle avec une architecture arbitrairement complexe, un algorithme de partitionnement récursif est appliqué pour décomposer sa représentation sous forme de graphe en une structure hiérarchique de graphes : chaque graphe ne contient qu'un nombre limité de nœuds, où chaque nœud peut représenter un sous-graphe avec un large ensemble d'opérations. Nous développons également un nouveau solveur ILP, H-ILP, qui génère

un programme de re-matérialisation en fonction du graphe et des limites de mémoire. Contrairement à CHECKMATE, H-ILP accepte que chaque nœud puisse être exécuté de plusieurs manières correspondant aux solutions du sous-graphe qu'il représente. Comme les sous-graphes sont résolus récursivement, leurs plans de rematérialisation sont combinés récursivement dans une approche ascendante, produisant finalement une solution pour l'ensemble du modèle. HIREMATE inclut non seulement H-ILP, mais aussi d'autres solveurs, y compris CHECKMATE et ROTOR, pour fournir des solutions lorsque le graphe cible a la structure souhaitée. Nous démontrons à nouveau l'efficacité de la solution et les performances de re-matérialisation de H-ILP à l'aide d'expériences. Ce travail est présenté dans le chapitre 3. Nous considérons HIREMATE comme la solution ultime au problème de la re-matérialisation, puisqu'il inclut divers algorithmes et n'a pas de restrictions sur les architectures de modèles. Cependant, si les solutions de rematérialisation réduisent l'empreinte mémoire des activations, les paramètres du modèle peuvent toujours devenir un obstacle. Cette question dépasse les capacités de HIREMATE et est abordée dans les travaux qui suivent.

OFFMATE Pour réduire davantage l'empreinte mémoire des différents composants lors de l'apprentissage de grands modèles, nous proposons OFFMATE, une solution spécialement conçue pour réduire les coûts mémoire liés aux paramètres. Plusieurs approches, y compris le délestage et l'optimisation sur l'unité centrale (CPU), sont intégrées dans OFFMATE, en s'appuyant sur les stratégies de re-matérialisation introduites dans des travaux antérieurs. Étant donné qu'OFFMATE étend la gamme des opérations possibles lors de l'ordonnancement, l'algorithme d'optimisation souffre d'une grande complexité. Pour y remédier, nous introduisons un certain nombre d'hypothèses pour simplifier le problème tout en préservant les options les plus importantes dans la solution. En tant que produit final de notre thèse, OFFMATE permet le réglage fin d'un modèle Llama [56] de 7 milliards de paramètres sur une machine de 12 Go, permettant ainsi une réduction de la mémoire d'un facteur 10 pour seulement un surcoût de 20% dans le temps d'apprentissage. Cela permet aux chercheurs individuels et aux passionnés d'IA utilisant des GPU grand public d'affiner de grands modèles sans avoir accès à du matériel haut de gamme. Les performances et l'efficacité de OFFMATE sont détaillées dans le chapitre 4.

L'ensemble du code source est disponible en open source dans le dépôt git `https://github.com/topal-team/rockmate`.

Chapter 1

Context

1.1 Backgrounds

1.1.1 Artificial Intelligence (AI)

The formal inception of Artificial intelligence (AI) began in the mid-20th century. In 1956, the Dartmouth Conference, organized by John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon, is widely considered the birth of AI as an academic discipline. However, the limitations of these systems and the difficulty of capturing real-world complexity led to periods of reduced funding and interest, known as AI winters. The field experienced a resurgence in the 1980s and 1990s with advancements in machine learning, driven by increased computational power and the advent of neural networks. In 2012, AlexNet [31] has made breakthrough in the ImageNet [12] classification task, thanks to the development of Deep neural networks (DNNs). Since then, DNNs become the driving force of AI and succeed in multiple fields including image recognition, translation and game playing.

The most recent prominence moment for AI came with the release of Chat-GPT [44], showing the dominating capabilities of the Large Language Models (LLMs). LLMs have demonstrated good performance in various language-related domains, including question answering, code generation, and natural language translation. Inspired by the transformer structure [58], LLMs with different architectures such as GPT [44], Bloom [60], Llama [56] and Mistral [27] have been trained and demonstrate excellent performance on general tasks. Based on these large pre-trained models, which require a significant amount of resources to train, many more specialized models have been proposed and contribute to a variety of fields such as law [22], medicine [61], and finance [35].

1.1.2 Tasks and Accelerators

The artificial intelligence (AI) models are trained to learn the pattern of collected data so that insightful predictions can be made to practical tasks. The life cycle of AI models can be split into two phases: training and inference. Training an AI model involves feeding

it large amounts of data and allowing it to learn patterns and relationships within that data through algorithms and iterative optimization. This process adjusts the trainable parameters to minimize errors in its predictions. Inference, on the other hand, applies the trained model to generate predictions on real tasks. During inference, the model uses the knowledge it gained during training to analyze and interpret the new data, producing results quickly and efficiently without further learning. In this thesis, we focus on the phase of training AI models.

The typical training procedure of an AI model involves a cycle of forward and backward passes through the network. It begins with the forward pass, where input data is fed into the model, passing through its layers in a predefined order. Each layer performs specific computations, transforming the data until the final layer produces the outputs. The output of each layer is called *intermediate activations*, serving as the input of the next layer. A loss function is defined on the predicted outputs, often based on comparing the predicted results to the true targets, (if known for the training samples). It is assumed that lowering the loss makes the model more accurate, thus trainable parameters in the model should be modified accordingly. In the backward pass, the gradient of the loss with respect to intermediate activations and model parameters are calculated by the chain rule. The former is released during the computation of backpropagation while the latter is the target of forward and backward propagation thus stored in memory. At the end of backpropagation, the gradient of the loss function with respect to each model parameter is generated, indicating how changes to each parameter contribute to reducing the loss. An optimization algorithm, like Stochastic Gradient Descent (SGD), uses these gradients to update the trainable parameters of the model in a direction that reduces the loss. This process is repeated over many iterations, allowing the model to progressively learn and improve its predictive accuracy.

Due to the high demand on computations in forward and backward passes, the training process is often deployed on Graphics Processing Units (GPUs). GPUs are crucial in AI model training due to their ability to perform parallel processing efficiently. Unlike CPUs, which are designed for general-purpose computing, GPUs excel at performing highly parallel computations simultaneously. This makes them ideal for the large-scale matrix multiplications that are fundamental in training deep learning models. By leveraging thousands of efficient cores, GPUs can process vast amounts of data in parallel, significantly speeding up the training process. The power of GPUs also rely on the optimized memory access patterns. A Video Random-Access Memory (VRAM) is designed to be physically close to GPUs (often integrated onto the graphic cards), usually serving as a buffer for GPUs to lower the latency. The latest VRAM technology is called high-bandwidth memory (HBM), which can be found on most dedicated GPUs since 2010s. A typical GPU may have VRAM size between 2GB and 160GB. In a standard training task of AI models, all the data related to the training task is stored in VRAM and swapped between processing cores for computations. This makes the size of VRAM a common bottleneck for training large AI models. Modern LLMs often takes hundreds to thousands of GPUs to train in parallel. For example, the 175-billion-parameter GPT-3 requires 1024 NVIDIA A100 GPUs to train in 34 days by estimation [42], which potentially

cost millions of dollars.

Since training LLMs from scratch is increasingly expensive, many studies have been performed based on pre-trained models. Fine-tuning is a process in AI model development where a pre-trained model is further trained on a more specific dataset to be adapted to a particular task. Initially, the model is trained on a large, generic dataset, capturing a wide range of features and patterns. During fine-tuning, this pre-trained model undergoes additional training on a smaller, task-specific dataset. This process allows the model to leverage the general knowledge it acquired initially while specializing in the nuances of the new data. Fine-tuning is especially useful when the task-specific dataset is limited in size, as it benefits from the robustness and generalization capabilities of the pre-trained model. This approach not only speeds up training but also often results in better performance compared to training a model from scratch, making fine-tuning a powerful technique in transfer learning and a cornerstone in the development of specialized AI applications.

However, a common challenge for fine-tuning LLMs is the memory bottleneck of the training process. While the number of desired computational operations significantly reduces in fine-tuning, the requirement of fitting the training-related data on GPU VRAM tends to stay consistent. In particular, most billion-parameter models can hardly be stored on consumer graphics cards, which typically have between 8GB and 24GB of video RAM (VRAM). Depending on the choice of hyperparameters and optimization settings, the training process may require substantially more memory than the VRAM available on a single GPU. It is thus difficult to participate in LLM fine-tuning without access to large-scale computational resources.

1.1.3 Memory requirement

In this section, we analyze the memory requirement in training AI models. The memory footprint when training a large model consists of several parts:

- intermediate activations with size M_{act}. M_{act} depends on the input batch size and which can be rebuilt using re-materialization to reduce the associated peak memory usage;
- model parameters with size M_{param};
- parameter gradients with size M_{p_grad}. M_{p_grad} is equal to the size of all trainable parameters;
- optimizer states with size M_{opt_st}. M_{opt_st} depends on the optimizer chosen for the task.

Once the model is selected, the size of M_{param} depends only on the data type of the parameters. Both M_{p_grad} and M_{opt_st} are proportional to the number of trainable parameters. If methods like weight-freezing or Parameter Efficient Fine-Tuning (PEFT) are applied, M_{p_grad} can be significantly smaller than M_{param}. M_{opt_st} also depends on the optimizer. Specifically, Adam optimizer [28] and its derived optimizers family store momentum and variance for each parameter gradient, which makes $M_{opt_st} = 2 \times M_{p_grad}$ if the same data type is used.

Depending on the training setup, parameters may be updated after every iteration or

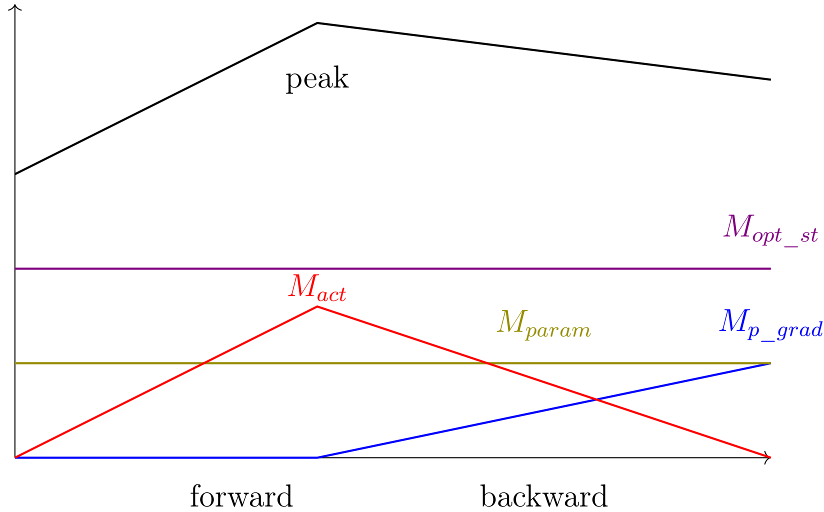


Figure 1.1: Memory requirement of different parts. Black line shows the summation of all part.

every several iterations (usually called gradient accumulation); the latter requires to keep all the parameter gradients in VRAM which makes the memory peak $M_{act} + M_{param} + M_{p_grad} + M_{opt_st}$. On the other hand, it is common to update parameters and release their gradients after every iteration. In this case, parameter gradients size increases during backward phase while intermediate activation size decreases. The peak memory is then smaller than $M_{act} + M_{param} + M_{p_grad} + M_{opt_st}$.

For example, the relative sizes of each part when training a full-precision Llama2-7B model with batch size 4 and sequence length 512 are $M_{param} = M_{p_grad} = 25$GB and $M_{act} = 41$GB. If the model is optimized with the Adam optimizer, $M_{opt_st} = 50$GB. If gradient accumulation is not used and updates are performed during the backward phase, parameter gradients can be removed from memory at each iteration, so that M_{p_grad} and M_{act} are not always required simultaneously. A simplified demonstration of memory requirements of different parts is shown in Figure 1.1. The peak memory usage for the corresponding training task is expected to be 125GB $< M_{act}+M_{param}+M_{p_grad}+M_{opt_st} =$ 141GB, which is still not feasible on any consumer GPU.

1.2 Related Works

In this section, we explore a range of solutions for reducing memory requirements in AI model training, from simulated algorithms to software deployment. We cover the existing approaches within the following categories:

1. Training simplification, reducing memory requirements by changing the training task (for example, using compression or lower precision);
2. Parallelism, distributing the data over multiple devices;
3. Re-materialization, deleting and recomputing intermediate activations to reduce the

memory peak;
4. Offloading, swapping memory between GPU VRAM and CPU RAM;
5. Software that uses one or more of the approaches above to handle practical training tasks.

1.2.1 Training Simplification

Modifying the training task in specific ways can effectively reduce memory usage and often speed up training. However, this comes at the cost of potentially impacting training precision. It is well-known that deep learning models are often overly complex for their tasks [64], and certain disturbance or simplification of the model have been found to improve the generalization capability [54]. Therefore, it is challenging to assess how different simplification methods will impact training performance.

Lower precision The basic example of simplifying a training task is using half precision to represent numerical values. While single precision (`float32`) is typically the default data type for model and activations, half precision (`float16` or `bfloat16`) can be employed instead. For GPUs that support these data types, training or fine-tuning in half precision can significantly accelerate the training process. Mixed-precision [40] is a method that uses half precision for the forward and backward passes while retraining single precision for the optimizer states to update model parameters accurately. This approach reduces the size of intermediate activations by half and speedup training remarkably. However, mixed-precision increases the memory usage of model parameters since both half precision and single precision copies of model parameters are saved. Overall, it may **not** reduce the total memory requirement.

Affine quantization A more sophisticated approach is integer quantization. The most popular implementation uses `int8`, which stores 256 values to represent the numerical values stored in `float32`. The idea of affine quantization [23] is to project the range of values in `float32` to the space in `int8`. A value x in `float32` can be written as $x = S \times (x_q - Z)$, where x_q is the quantized value in `int8`. The quantized values can be obtained by: $x_q = round(x/S+Z)$ for any given x within a certain range. Once the scale S and the zero-point Z are determined, only values within certain range can be represented in this scheme. Values outside this range are usually clipped to the closest representable value. Due to the limited space of `int8`, different values of x may be mapped to the same quantized value x_q. The information loss can affect the training results, necessitating sophisticated algorithms to tune the quantization scheme.

Parameter Efficient Fine-Tuning Parameter-Efficient Fine-Tuning (PEFT) is an advanced approach in machine learning that focuses on fine-tuning a subset of model parameters, rather than the entire model. By selectively adjusting only the most critical parameters, PEFT maintains high accuracy and generalization capabilities, making it particularly valuable in scenarios where computational efficiency and resource constraints

are paramount. PEFT has gained popularity for its ability to achieve competitive results with fewer parameters, thereby enhancing the scalability and practicality of deploying large models in real-world applications. In particular, the Low-Rank Adaptation (LoRA) family [20, 14] has shown that using a small fraction of the training parameters can achieve good performance on various tasks. We use the following example to illustrate the idea of LoRA: consider a layer $y = Wx$ where $x \in \mathbb{R}^n$ is the input vector, $y \in \mathbb{R}^n$ is the output vector, and $W \in \mathbb{R}^{n\times n}$ is the weight matrix. During the optimization of W, one copy of gradient and two copies of optimizer states (assuming the Adam optimizer is used) are stored in memory. If parameters are stored in single precision (4 bytes), the optimization stage requires $4(4n^2 + O(n))$ bytes memory in total. In LoRA, on the other hand, it is assumed that the pre-trained W is a good representation of the current layer and only requires a minor adjustment for fine-tuning. Therefore, the layer can be redefined as $y = Wx + ABx$, where $A \in \mathbb{R}^{n\times r}$ and $B \in \mathbb{R}^{r\times n}$ are the only trainable matrices. The layer can thus be written as $y = W'x$ where $W' = W + AB$. With $r \ll n$, replacing W' with AB provides less flexibility but reduces the memory requirement to $n^2 + O(n) + O(rn)$. Other PEFT methods include training only the input embedding layer [1], training hidden states [38], and training with a sparse mask over the weights [55].

1.2.2 Parallelism

Using multiple devices has become a popular approach for training large AI models. In addition to speeding up the training procedure by leveraging the combined computational power of multiple devices, it may also reduce the memory requirement on any single device. Various methods have been proposed to improve training efficiency, depending on how the training tasks are distributed across multiple devices.

Data parallelism Data parallelism is a fundamental technique in parallel training [11]. This approach involves dividing a large batch of input data into smaller chunks and sending them to different devices (usually the same type of GPUs). While each device holds the same copy of the model, the gradients generated during backpropagation are different on each device due to the variety of sample input. After the backward pass, the devices communicate the generated gradients to obtain the average gradients of loss over all the entire batch of input data. The averaged gradients will be used to optimize the parameters before the next iteration. If the input data is distributed over N devices, the activation size on each device is divided by N. However, data parallelism does not reduce the parameter-related memory usage since each device keeps a full copy of the model. Moreover, the gradients obtained on each device need to be exchanged after each iteration, causing a high communication cost on large size models.

Model parallelism When the model size is too large, it can be beneficial to split the parameters across different devices. These techniques are called model parallelism. Depending on how the parameters are divided onto different devices, there are multiple strategies for model parallelism.

It is a common choice to split the parameters layer-wise so that each device only needs to hold the parameters of a few layers of the neural networks. Unlike data parallelism, the computation of different devices in model parallelism depend on each others, thus only one device (or multiple devices corresponding to the same layers) is utilized at a time in the naive approach. The efficiency can be significantly improved if multiple devices can be utilized at the same time. Pipeline parallelism [41, 63, 17, 15, 34, 39] is the method to split the input batch of data into smaller micro-batches, allowing different devices to synchronize their computation. After the device handling the first layer has computed the first micro-batch, it will start computing the second micro-batch while the next device begins computing the second layer of the first micro-batch. By smartly synchronizes the computations, different devices will be in use most of the time. The communication cost of pipeline parallelism is only the activations passed between different devices, which are the input data of a layer. In transformer architectures, the input of each layer is at least one order of magnitude smaller than the total activations size of the layer, making pipeline parallelism a low-communication approach. Pipeline parallelism could significantly reduce the memory requirements of each device with reasonably low overhead by splitting the task smartly.

When a singe layer of the neural network is too large, it can be beneficial to split the parameters across tensor axes. Megatron-LM [53] proposed tensor parallelism to parallelize the matrix multiplication $Y = XA$ as :

$$X = \left[X_1, X_2\right], A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix},$$

so that $[Y_1, Y_2] = [(X_1 A_1), (X_2 A_2)]$. Adaptation is required for the case when non-linear operations are applied after the matrix multiplication. Communication occurs when outputs need to be used for the same computation. Unlike data parallelism, which splits the computation across the bacth-axis, tensor parallelism distributes the weight matrix A, reducing the memory requirements for parameters. Combining tensor parallelism with data parallelism allows a memory-intensive computation to be distributed across multiple devices.

The combination of pipeline parallelism and tensor parallelism is designed to achieve low latency on modern architectures of GPU clusters. The NVIDIA technology NVLink enables bandwidth between 100 GB/s to 1 TB/s for GPUs within a server, making it ideal for high communication cost parallelism (for example, tensor parallelism and data parallelism). Communication between servers in a cluster, typically realized through PCI Express with bandwidth an order of magnitude lower than NVLink's, is sufficient to handle the lower communication cost of pipeline parallelism. Alpa [67] uses integer linear programming and dynamic programming to obtain a near-optimal combined parallelism solution for a given cluster.

1.2.3 Re-materialization

In order to reduce the memory requirements of a training task, certain selected intermediate activations can be released during the training iteration and recomputes when needed. This approach is called re-materialization. Historically, re-materialization strategies have their origins in the checkpointing techniques developed in the context of automatic differentiation (AD). Because of this application context, these works have focused mainly on the case of homogeneous chains, i.e. models consisting of a sequence of identical blocks. In this context, it is possible to find optimal solutions and even closed form formulas can be derived to automatically find the activations to keep and the ones to delete. The classical gradient checkpoint strategies allow to select a subset of intermediate activations to store during the forward pass and allowing the others to be released and recomputed during the backward pass. A more general form of re-materialization could consist of recomputing backward operations to re-generate the activation gradients as well. The re-materialization algorithm explores the balance in the trade-off between the computation time and the memory efficiency. For a n-layer homogeneous chain of deep neural network, gradient checkpoint can achieve $O(\log n)$ memory consumption of activations with $O(n \log n)$ time cost [8].

Transformer models If the computational graph of a model is determined, fixed decisions can often be made easily for operations to be recomputed. Megatron-LM [30] uses *selective activation recomputation* on transformer-based models, where only certain pre-defined operations will be recomputed. It is efficient in the transformer-based models but provides no insight on general forms of models. FlashAttention [10, 9] uses a similar idea for the specific Attention module of transformers where some selected intermediate tensors are recomputed during the backward pass.

Arbitrary models For heterogeneous models, sophisticate algorithms are useful to achieve memory usage reduction with lower timer overhead. Without knowing the detailed computational cost, re-materialization decisions can be made dynamically during the training process [29]. On the other hand, most approaches for static neural networks involve inspecting the computational cost and propose a desired schedule based on the requirements. It is proved to be NP-Complete in the strong sense to find the optimal schedule on any heterogeneous models given a fixed budget for peak memory usage [7]. Near-optimal solutions are found for different situations with limitations [7, 24, 2].

On heterogeneous sequential models, Rotor [7] has been proposed to find the optimal re-materialization schedule within a reasonable time through dynamic programming (DP). In the strict sequential models, all the intermediate activations are used exactly once during forward. With some limitations, Rotor could work on models that are not strictly sequential where activations can be used more than once in the local operations. For example, ResNet [19] contains residual block where one layer relies on not only the previous layer but also the layer before. These locally non-sequential blocks do not disrupt the overall sequential-like structure of the model. For models with such so-called *skip*

connections, the connected parts, such as a residual block are considered as one *layer* in Rotor. The decisions are made on whether to compute or delete all the activations in the layer as a unit.

With the weak limitation of assuming a fixed order of computation nodes, Checkmate [24] uses integer linear programming (ILP) to obtain an optimal re-materialization solution over any computational graph. Unlike Rotor, Checkmate allows backward operations to be recomputed as well. Although the ILP model can find the optimal solution with minimal limitations, it involves $O(n^2)$ binary variables, where n is the number of nodes in the graph. For large neural networks with hundreds to thousands of nodes, Checkmate is not feasible for obtaining an optimal solution in a reasonable amount of time.

Moccasin [2] is a model based on Checkmate, replacing the ILP as a Constraint Programming (CP) formulation. The key contribution of Moccasin is to limit the number of times a node can be recomputed, significantly reducing the complexity of the optimization model. In practice, the authors found that recomputing each node at most twice is often sufficient for general computational graph. Although this approach markedly lowers the complexity compared to Checkmate's ILP, the Moccasin CP optimization can still take hours to obtain an optimal solution.

1.2.4 Offloading

The modern architecture of GPUs typically featured dedicated VRAM, which is generally fixed in size and an order of magnitude lower than the size of CPU RAM. Hence, one could exploit the larger space on CPU RAM to form a hierarchical memory accessing structure. When the activations and parameters exceed the capacity of VRAM, they can be offloaded to CPU RAM and paged back to VRAM when needed. Note that this approach relies on the bandwidth between VRAM and RAM, which is usually 10-24 GB/s bidirectional per GPU realized through PCI Express. Assuming the size of VRAM is between 10 and 100 GB and the size of communication amount is within 10 to 1000 GB, the communication time can take from seconds to tens of seconds, which is usually not negligible during a training iteration. It is thus crucial to properly synchronize the communication and computation when performing offload. If scheduled properly, communicating can be fully overlapped with computation operations, eliminating any additional time costs. Unlike re-materialization, offloading can be useful to reduce the memory requirements of not only intermediate activations but also model parameters.

Activation offloading Several methods use activation offloading [50, 5, 6] to reduce the memory requirement of training. Based on Rotor [7], POFO [6] successfully combines re-materialization and activation offloading on sequential models. In order to simplify the problem, POFO assumes that offloading can only happen during the forward phase and prefetching during the backward phase. In addition, POFO assumes that memory of a tensor can be released and allocated continuously before its communication is finished. With the assumptions to simplify the problem, an optimal solution can be found through

POFO in a reasonable amount of time. Based on Checkmate [24], the authors of POET [45] added activation offloading to the ILP. The objective of ILP is also modified to minimize the energy cost of the training iteration. The energy objective can be easily written as a linear function of computation cost and communication cost. Due to the high complexity of ILP, a default solving time limit of 10 minutes is used in POFO to ensure reasonable solving results on large models. Both algorithms can effectively reduce the time overhead of re-materialization by exploiting communication bandwidth.

Parameter offloading To limit memory consumption, some recent works focus on strategies for offloading model parameters during the training phase. The first strategy is proposed in [46], which introduces the L2L (Layer-to-Layer) algorithm. For a graph typically consisting of a sequence of encoders, the L2L algorithm uses a synchronous parameter server on the CPU to keep a copy of the model parameters while storing only the layer in use on the GPU. A strategy for adapting the size of the mini-batch is then proposed, in order to use the largest possible mini-batch given this systematic strategy of layer offloading. Closed-form formulas are proposed to estimate the training time (for all identical layers). This strategy was then adapted in ZeRO-Offload [49] to offload optimizer states to the CPU. It allows significant memory savings with the Adam optimizer while achieving higher throughput compared to L2L [46]. However, ZeRO-Offload [49] keeps all parameters in the GPU (in single precision), which requires significantly more memory than L2L when the model is very deep. Despite the optimizations proposed in [49], it has been recently observed in [37] that approaches such as L2L [46] or ZeRO-Offload [49] generate a very large amount of traffic on the PCI Express bus between the CPU and the GPU. Since PCI bus bandwidth is limited, this traffic may become a bottleneck and slow down the whole training process significantly. For this purpose, it is proposed to offload only a subset of layers, typically the first half of layers in [37].

Combined offloading Some methods proposed for offloading any tensor [33, 57, 21], so that they can be applied to both activations and parameters. In [33, 21], the tensors that have been unused for the longest time are preferred candidates for offloading. AutoSwap [57] relies on priority scores to select the best candidates for offloading, but it is limited to the case of a single memory peak, which is appropriate for activation offloading, but not for parameter offloading. The schedule of transfers is similar for all the above methods: offloading is performed as soon as possible, and prefetching is performed as late as possible, trying to prevent idle time before the operation requiring prefetched data. Despite their generalization, these methods have some drawbacks: in particular, they do not fully take into account the benefits of offloading parameters during the backward phase.

1.2.5 Software

In this section, we present some software tools that provide implementations for practical training/fine-tuning of neural networks with limited resources.

FlashAttention [10, 9] Tailored for transformer-based models, FlashAttention is now adapted into many frameworks to support large language models (LLMs) training. It redefines the implementation of the attention layer, significantly reducing the memory required for intermediate activations. By optimizing memory access on the GPU, FlashAttention can also effectively accelerate the training of the model. Since FlashAttention optimizes the memory access of CUDA operations, the implementation depends on the hardware architecture of GPUs.

ZeRO [47] The library ZeRO, which developed by Microsoft, supports distributed training of neural networks. ZeRO has several stages that exploit different resources for training:

- ZeRO-1 partitions the optimizer states onto different devices, and each device will handle the optimization of part of the network but running forward and backward computations of the entire network.
- ZeRO-2 also partitions the model parameter gradients onto different devices.
- Finally, ZeRO-3 partitions the model parameters onto different devices. This requires a high communication cost since each device needs to collect the model parameters during the forward and backward passes.

Further works based on ZeRO framework exploit more resources.

- ZeRO-Offload [49] adds CPU optimization to the framework so that optimizer states are created on CPU RAM directly. Model parameter gradients are offloaded to CPU RAM before the optimization, and updated parameters are loaded back to GPU VRAM in every iteration before they are used for computations.
- ZeRO-Infinity [48] further exploits the space of NVMe disk to store the model parameters. It also optimizes the bandwidth usage for data movements.

QLoRA [14] Specifically designed for fine-tuning, QLoRA was proposed to integrate quantization and low-rank adaptation. It also includes a paged-optimizer that offloads the optimizer states to CPU RAM. It is integrated into many frameworks including HuggingFace libraries, which allows the users to easily apply it on their fine-tuning tasks.

Rotor [7] Rotor is a library designed for models in the form of `torch.nn.Sequential`. It implements the algorithm of Rotor [7] and POFO [6] for sequential PyTorch models, given a fixed memory budget. It is important to note that models defined as `torch.nn.Sequential` may not be strictly sequential in terms of atomic operations. In such cases, non-sequential blocks, such as Residual blocks in ResNet, will be considered as a single layer in Rotor.

1.3 Operations in PyTorch Modules

In this section, we use an example model to introduce the computational graph for PyTorch Modules and the possible execution order of the operations in order to reduce

the memory requirements. We define a simple model with the Python code shown in Code 1.2. Note that all the parameters are named as w_i while the activations and inputs are named x_i. The following sections explain how to obtain a computational graph of operations and tensors, from which possible solutions can be proposed based on different assumptions.

```
import torch
def forward(x0):
    x0_ = x0.transpose()
    x1 = torch.matmul(x0_, w1)
    x2 = torch.matmul(x1, w2)
    x3 = x1 + x2
    x4 = torch.matmul(x3, w4)
    return x4
```

Code 1.2: Toy Example

1.3.1 Dependencies between Operations

Given a PyTorch Module, we define a graph builder called RK-GB to obtain its computational graph. RK-GB works as follows: based on `torch.export`, the model is inspected with a sample input to generate the computational graph, detailing dependencies between operations. Operations that do not allocate GPU memory (e.g. `transpose()` in the example) are merged so that each node in the graph has at least one output tensor node with GPU memory allocation. An example computational graph for the **forward** phase is shown in Figure 1.3. In the graph, circles represent tensors occupying memory, while squares represent functions generating tensors. F_1, F_2, F_3, F_4 correspond to the code in line 4, 5, 6, 7 in Code 1.2. Line 3 is not considered as a node in the graph since no memory allocation is needed from this operation. An arrow from a tensor to a function indicates that the function takes the tensor as input, and an arrow from a function to a tensor represents that the tensor is generated from the function.

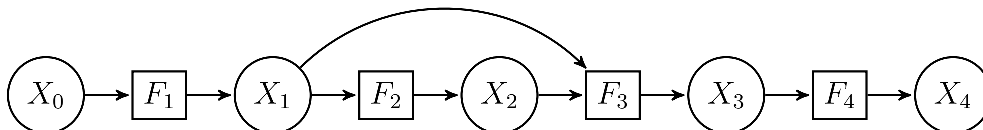


Figure 1.3: Forward graph

We separate the representations of the forward operations (noted as F_i) and their outputs (noted as X_j) in the graph. They are called `cNode` and `aNode`, short for computation node and allocation node. These representations are useful when the operations may have multiple outputs. In practice, it is hard to obtain atomic operations in PyTorch. The computational nodes obtained through `torch.export` may indeed create

multiple tensors and allocate more memory than the size of the final output tensor. Specifically, some tensors are created and saved only for the backpropagation. Unlike x_j defined explicitly in Code 1.2, those tensors are referred only in `.grad_fn` of other tensors. In the later chapters, we refer to those tensors as *phantoms* and apply special rules to them since they are created only in forward and used in backward. The choice of separating computation and allocation representations in the graph has other benefits, which are explained through the later chapters with more specific scenarios.

We present the computational graph in Figure 1.4 for the forward and backward phases. The parameters are also represented in the graph. A graph containing forward and backward dependencies of a model is called FB-Graph. Any FB-Graph is a directed bipartite graph, where any edge is between a `cNode` and an `aNode`, representing the dependency between a tensor allocation and a computation. If the direction is from `cNode` to `aNode`, it means the tensor is computed through the computation operation; otherwise, it means the tensor is needed by the computation operation.

Note that multiple functions may have the same output tensor. For example, node G_1 shows the case where one tensor is generated from two different functions: $\frac{\partial loss}{\partial X_1} = \frac{\partial loss}{\partial X_2}\frac{\partial X_2}{\partial X_1} + \frac{\partial loss}{\partial X_3}\frac{\partial X_3}{\partial X_1}$, where two functions corresponding to the two terms on the right-hand side are computed separately. In this case, memory is allocated after the execution of any function (B_2 or B_3 for G_1), but the tensor is only complete when all associated functions are executed before it can be taken as input by the further operations.

Another special node is *Loss*. As its name suggests, it represents the loss function which occurs after the forward phase. As described in Section 1.1.2, the loss value represents the quality of the output predictions of a model and the purpose of forward-backward propagation is to reduce the loss values over the training dataset. In the example provided in Code 1.2, we do not show the loss function explicitly. It could be as simple as the average over the output X_4, or an entire model with multiple operations. The key feature of *Loss* node is that is can be arbitrary complicated and we are often interested at the memory saved before it. Every FB-Graph must contain one and only one *Loss* node.

In PyTorch AutoGrad, the functions shown in Graph 1.4 are executed in the order defined by source code. Note that the order specified by the code may neither be the only nor the optimal order. The tensors generated during the execution (intermediate activations) will be deleted as soon as they are no longer needed for the further functions. For instance, X_3 in Graph 1.4 will be deleted after F_4 because no further functions depend on it. All the other tensors will be kept in memory until the backward phase since they are needed for backward. The parameters are not deleted in the default execution, and their gradients GW will be used by the optimization after the backward phase.

1.3.2 Execution of Operations

The target of our algorithms is to generate a schedule of operations which satisfies two conditions:

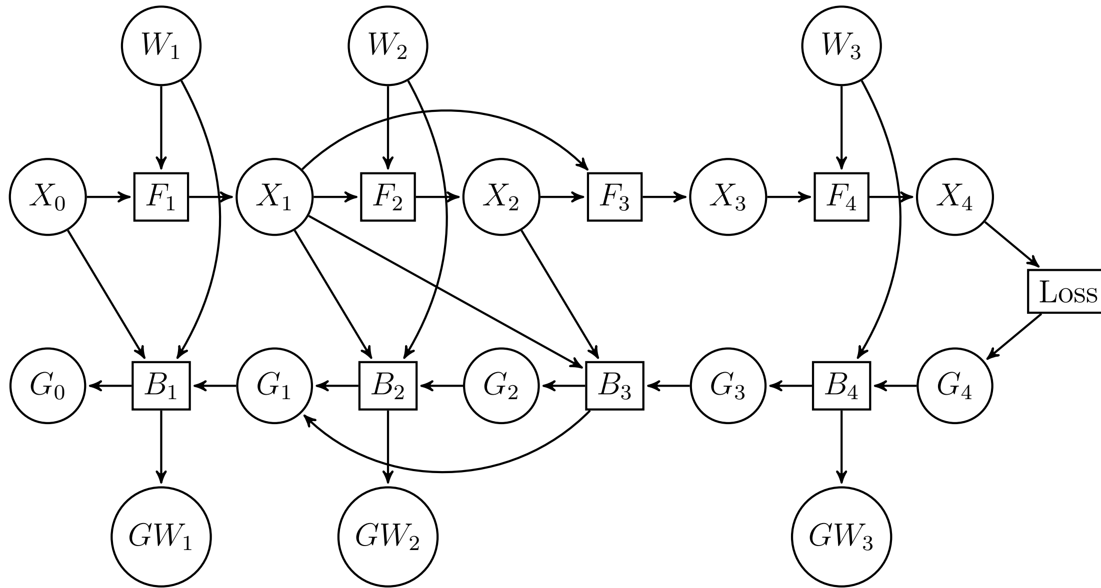


Figure 1.4: Forward and backward graph

- Operations are executed with the correct inputs present. For example, B_1 can only be executed with G_1 correctly computed, which requires **both** B_2 and B_1 executed.
- The maximum memory allocation of all the present tensors does not exceed a predefined size, which is usually the size of GPU VRAM.

Different strategies can be applied to find the schedule that satisfies the two conditions above. In the following sections, we provide two simple examples using re-materialization and offloading. To avoid complicating the problem, we do not assign sizes to each tensor, thus not quantifying the second condition.

Re-materialization As discussed in Section 1.2.3, re-materialization is the method of deleting tensors during execution and recomputing them when needed. We use the model in Figure 1.4 to provide an example of a re-materialization solution. Assume we choose to delete and recompute X_2, the forward and backward execution can be written as code 1.5. We need to define backward functions `B()` to not automatically traversing the entire AutoGrad graph. In this example, we removed the size of X_2 from the memory requirement between F_4 and B_4 at the cost of extra time to compute F_2. Note that it may not reduce the peak memory of the overall iteration, which depends on the size of different tensors.

Note that the *Loss* node is also possible to recompute in practice, but we do not consider this possibility within the scope of this thesis.

Offloading Another approach to reduce memory requirement is offloading, which involves swapping data between GPU VRAM and CPU RAM. Code 1.6 shows a possible solution for the model from Figure 1.4 using the offloading techniques. Unlike

```
import torch
def forward(x0):
    x0_ = x0.transpose()
    x1 = torch.matmul(x0_, w1)
    x2 = torch.matmul(x1, w2)
    x3 = x1 + x2
    del x2
    x4 = torch.matmul(x3, w4)
    # x3 is deleted automatically by PyTorch here
    return x4

def backward(g4):
    g3 = B4(x3, g4, w3)
    x2 = torch.matmul(x1, w2)
    g2 = B3(x2, x1, g3)
    g1 = B2(x1, g2, w2)
    g0 = B1(x0, g1, w1)
```

Code 1.5: Toy Example with Re-materialization

re-materialization, offloading may not incur additional time per iteration because offloading and prefetching operations can occur simultaneously with computation operations in different CUDA streams. However, it raises the complexity of scheduling due to the dependencies between operations. For example, computation of backward function B_2 needs to wait for the prefetching of W_1 to be finished, which starts only after memory is allocated on GPU. Consequently, it is complicated to find the optimal schedule and heuristic approaches are required in practice.

Our contribution In this thesis, our contribution is to find the optimal or near optimal solutions to PyTorch modules under various assumptions, primarily focusing on re-materialization and offloading approaches. Our final deliverable is a practical software tool designed for customized training tasks, facilitating efficient management of memory resources during model execution.

```
import torch
def forward(x0):
    x0_ = x0.transpose()
    x1 = torch.matmul(x0_, w1)
    with torch.cuda.stream(offload_stream):
        w1_cpu.copy_(w1)
    x2 = torch.matmul(x1, w2)
    del w1
    x3 = x1 + x2
    x4 = torch.matmul(x3, w4)
    # x3 is deleted automatically by PyTorch here
    return x4

def backward(g4):
    g3 = B4(x3, g4, w3)
    x2 = torch.matmul(x1, w2)
    g2 = B3(x2, x1, g3)
    with torch.cuda.stream(prefetch_stream):
        w1.copy_(w1_cpu)
    g1 = B2(x1, g2, w2)
    g0 = B1(x0, g1, w1)
```

Code 1.6: Toy Example with Offloading

Chapter 2

ROCKMATE

2.1 Introduction

As discussed in Section 1.2.3, re-materialization is a method to reduce the memory requirements of a training task by deleting a selection of intermediate activations and recomputing them when needed. Multiple approaches have been proposed to balance the tradeoff between the memory requirement reduction and the time overhead in re-materialization. One classical problem is to minimize the recomputing time while the peak memory is constrained to a given budget. In this chapter, we present our work ROCKMATE, which combines two existing algorithms ROTOR and CHECKMATE. This introduction presents the formulation of CHECKMATE and ROTOR, along with their strength and limitations. Specifically, we show that the existing works CHECKMATE and ROTOR are not ideal for solving the optimization problem on models structured as a long sequence of blocks, for example GPT [44]. ROCKMATE works exceptionally well for those neural networks by smartly combining CHECKMATE and ROTOR.

2.1.1 CHECKMATE

In this section, we introduce the Integer Linear Programming (ILP) formulation of CHECKMATE [25]. Unlike the FB-Graph defined in Section 1.3.1, CHECKMATE is based on a computational graph where a computation operation and its output tensors are wrapped into a single node. The graph is defined as follows: A computation graph $G = (V, E)$ is a directed acyclic graph with n nodes $V = \{v_1, \dots, v_t\}$, where v_i represents the i-th operation in a PyTorch model. The re-materialization solution of the model can be represented as a sequence of v_i. The sequence is unrolled into T stages and only allows a computation node to be computed once per stage. $S_{t,i} \in \{0, 1\}$ indicates whether the results of operation i is saved in memory at stage $t - 1$ until stage t. $R_{t,i} \in \{0, 1\}$ is a binary variable reflecting whether computation node c_i is computed at time stage t. The breaking down of a re-materialization schedule into the binary variable representation is showed in Figure 2.1.

If given infinite memory budget, the Linear Programming model can be formalized as:

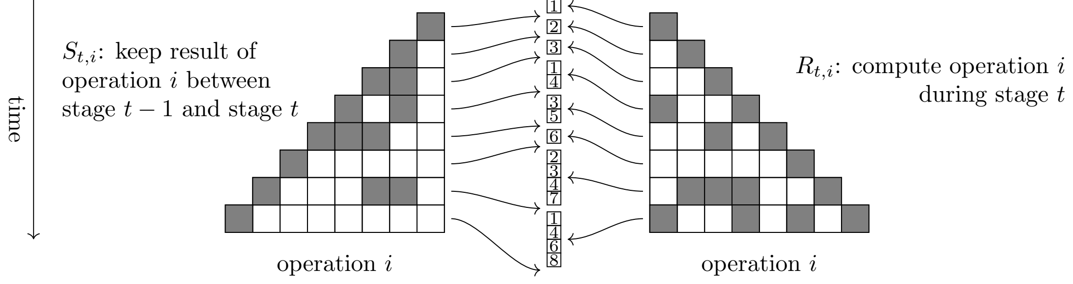


Figure 2.1: Visualization of CHECKMATE formulation where the middle columns shows a sequence of operations being (re)computed in order, and its representation in ILP variables is showed on the right. The left side shows when the tensor are saved in memory.

$$
\begin{aligned}
& \underset{R,S}{\arg\min} \sum_{t=1}^{n} \sum_{i=1}^{t} C_i R_{t,i} \\
& \text{ subject to} \\
& R_{t,j} \leq (R_{t,i} + S_{t,i}) \quad \forall t, \forall i \rightarrow j \\
& S_{t,i} \leq R_{t-1,i} + S_{t-1,i} \quad \forall t \geq 2, \forall i, \\
& \sum_i S_{1,i} = 0, \\
& \sum_t R_{t,n} \geq 1, \\
& R_{t,i}, S_{t,i} \in \{0,1\} \forall t \forall i
\end{aligned}
$$

C_i is the time cost of operation i. The constraints ensure that each operation is computed with its input in memory and the tensors can only be generated from the source computation.

To represent the memory usage of a schedule, a new variable FREE as $\text{FREE}_{t,i,k} = 1$ for $(v_i, v_k) \in E$ if v_i is deallocated in stage t after evaluating node v_k otherwise 0.

$U_{t,k}$ is defined as the bytes of memory in use after evaluating v_k. Before evaluating v_{k+1}, v_k and its dependencies (parents) may be deallocated if there are no future uses. Then, an output tensor for the result of v_{k+1} is allocated, consuming memory M_{k+1}.

$$
U_{t,k+1} = U_{t,k} - \text{ mem_freed }_t(v_k) + R_{t,k+1} M_{k+1},
$$

where $mem_freed_t(v_k)$ is the amount of memory freed by deallocating v_k and its parents at stage t. Let

$$
\text{DEPS}[k] = \{i : (v_i, v_k) \in E\}, \text{ and } \text{USERS}[i] = \{j : (v_i, v_j) \in E\}
$$

denote parents and children of a node, respectively. Then, in terms of auxiliary variable $\text{FREE}_{t,i,k}$, for $(v_i, v_k) \in E$,

$$\text{mem_freed}_t\,(v_k) = \sum_{i \in \text{DEPS}[k] \cup \{k\}} M_i * \text{FREE}_{t,i,k}\,, \text{ and}$$

$$\text{FREE}_{t,i,k} = R_{t,k} * \underbrace{(1 - S_{t+1,i})}_{\text{Not checkpoint}} \prod_{j \in \text{USERS}[i]} \underbrace{(1 - R_{t,j})}_{\text{Not dep.}}$$

where the second factor ensures that M_i bytes are freed only if v_i is not checkpointed for the next stage. The final factors ensure that $\text{FREE}_{t,i,k} = 0$ if any child of v_i is computed in the stage, since then v_i needs to be retained for later use.

In order to write it in a linear form, the authors use the following formulation which is an equivalent:

$$\begin{aligned} \text{FREE}_{t,i,k} &\in \{0,1\} \\ 1 - \text{FREE}_{t,i,k} &\leq \text{ num_hazards }(t,i,k) \\ \kappa\,(1 - \text{FREE}_{t,i,k}) &\geq \text{ num_hazards }(t,i,k) \end{aligned}$$

where

$$\text{num_hazards }(t,i,k) = (1 - R_{t,k}) + S_{t+1,i} + \sum_{\substack{j \in \text{USERS}[i] \\ j > k}} R_{t,j}$$

and κ is the maximum value which num_hazards can assume. Once the memory representation is written in linear form, constraints $U_{t,k} \leq M_{budget}$ is enough to make sure the ILP solution respects the memory budget.

The major limitation of CHECKMATE is the high complexity. CHECKMATE ILP formulation contains $O(|V||E|)$ binary variables. In some large models, when each node represents a single PyTorch operation, $|V|$ and $|E|$ can be hundreds to thousands. The high complexity of the ILP formulation prevents CHECKMATE from being applied to large models.

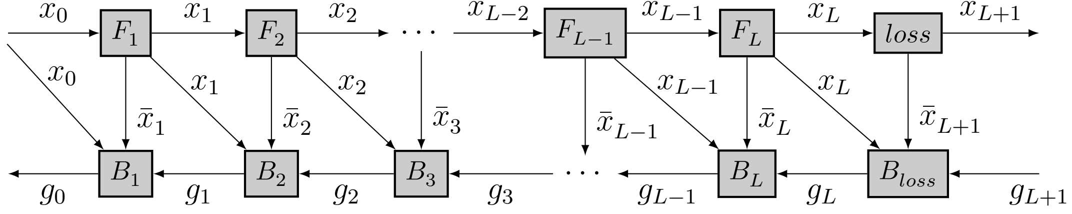
2.1.2 ROTOR

In this section, we present the Dynamic Programming (DP) formulation of ROTOR [4]. The algorithm is based on the memory persistence assumption:

Assumption 2.1.1. *Any checkpointed value is kept in memory until it is used in the backward phase.*

ROTOR works for sequential networks which can be seen as a chain of *blocks*, each having one input and one output. For a chain of length L, we denote by $C_{BP}(s,t,m)$ the optimal execution time to process the chain from stage s to stage t with peak memory at most m, assuming that the input tensors x_{s-1} and g_t are stored in memory, but the size of x_{s-1} should not be counted in the memory limit m. We also introduce $\bar{x}_i$ to represent the

tensors passed from forward of the i-th layer to its backward operations. The following notations are used in the formulation:



	Operation	Input	Output	Time	Overhead
F_ℓ^{all}	Forward(save all)	$\{x_{\ell-1}\}$ $\{\bar{x}_{\ell-1}\}$	$\{x_{\ell-1}, \bar{x}_\ell\}$ $\{\bar{x}_{\ell-1}, \bar{x}_\ell\}$	u_f^ℓ	o_f^ℓ
F_ℓ^{ck}	Forward(save input)	$\{x_{\ell-1}\}$ $\{\bar{x}_{\ell-1}\}$	$\{x_{\ell-1},\ x_\ell\}$ $\{\bar{x}_{\ell-1},\ x_\ell\}$	u_f^ℓ	o_f^ℓ
$F_\ell^\varnothing$	Forward(save nothing)	$\{x_{\ell-1}\}$	$\{x_\ell\}$	u_f^ℓ	o_f^ℓ
B_ℓ	Backward	$\{g_\ell, \bar{x}_\ell, x_{\ell-1}\}$ $\{g_\ell, \bar{x}_\ell, \bar{x}_{\ell-1}\}$	$\{g_{\ell-1}\}$ $\{g_{\ell-1}, \bar{x}_{\ell-1}\}$	u_b^ℓ	o_b^ℓ

$$m_\varnothing^{s,t} = \max\left(|g_t| + |x_s| + o_f^s, |g_t| + \max_{s+1\le j<t}\left(|x_{j-1}| + |x_j| + o_f^j\right)\right)$$

$$m_{\text{all}}^{s,t} = \max\left(|g_t| + |\bar{x}_s| + o_f^s, |g_s| + |\bar{x}_s| + o_b^s\right)$$

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

$C_{BP}(s, t, m)$, the optimal time for any valid persistent sequence to process the chain from stage s to stage $t \ge s$ with available memory m, is given by

$$C_{BP}(s, s, m) = \begin{cases} u_a^s + u_b^s & m \ge m_{\text{all}}^{s,s} \\ \infty & m < m_{\text{all}}^{s,s} \end{cases} \tag{2.1}$$

$$C_{BP}(s, t, m) = \min\left(C_1(s, t, m), C_2(s, t, m)\right) \tag{2.2}$$

$$C_1(s,t,m) = \begin{cases} \min_{s'=s+1\ldots t} \sum_{k=s}^{s'-1} u_f^k + C_{BP}\left(s', t, m - x_{s'-1}\right) & \\ \quad +C_{BP}\left(s, s'-1, m\right) & m \ge m_\varnothing^{s,t} \\ \infty & m < m_\varnothing^{s,t} \end{cases}$$

$$C_2(s,t,m) = \begin{cases} u_f^s + C_{BP}\left(s+1, t, m - \bar{x}_s\right) + u_b^s & m \ge m_{all}^{s,t} \\ \infty & m < m_{all}^{s,t} \end{cases}, \quad \text{where}$$

Authors of Rotor [4] prove that Algorithm 1 and Algorithm 2 compute an optimal sequence, for all input arguments.

Algorithm 1: Compute optimal persistent schedule for a chain of length L with memory M.

```
1  Initialize table C of size (L+1) × (L+1) × M ;
2  for 1 ≤ s ≤ L+1 and 1 ≤ m ≤ M do
3      Initialize C[s, s, m] with Equation 2.1 ;
4  for s = 1, ..., L do
5      for t = s+1, ..., L+1 do
6          for m = 1, ..., M do
7              Compute C[s, t, m] with Equation 2.2 ;
8  return OptRec(C, 1, L+1, M − x_0) ;
```

Algorithm 2: OptRec(C, s, t, m) – Obtain optimal persistent sequence from the table C

Input : C, s, t, m
Output: Optimal persistent sequence

```
1  if C[s, t, m] = ∞ then
2      return Infeasible ;
3  else if s = t then
4      return (F^s_all, B^s) ;
5  else if C[s, t, m] = C_ck(s, s', t, m) then
6      S ← (F^s_ck, F^{s'+1}_ck, ..., F^s_ck) ;
7      S ← (S, OptRec(C, s', t, m − x_{s'−1})) ;
8      return (S, OptRec(C, s, s' − 1, m)) ;
9  else
10     return (F^s_all, OptRec(C, s+1, t, m − x_s), B^s) ;
```

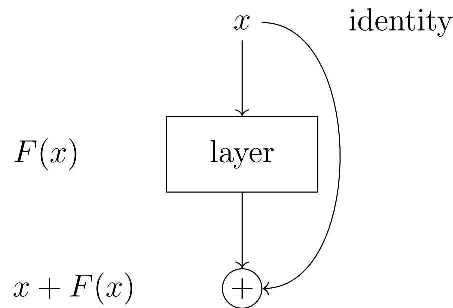


Figure 2.2: ResNet [19] contains residual blocks shown here.

For the non-sequential networks where activations can be used more than once in the local operations, ROTOR works with certain limitations. For example, ResNet [19] contains residual block (as shown in Figure 2.2) where one layer relies on not only the previous layer but also the layer before. These locally non-sequential blocks do not disrupt the overall sequential-like structure of the model. For models with such so-called *skip connections*, the connected parts, such as a residual block are considered as one *layer* in ROTOR. The decisions are made on whether to compute or delete all the activations in the layer as a unit.

Our contribution summary The main idea of this chapter is to combine the ideas of (i) CHECKMATE, which finds good solutions in the case of general graphs but is slow, and (ii) ROTOR, which finds the optimal solution only in the case of sequential networks, can find the solution quickly.

The GPT neural networks are not completely sequential, but they can be decomposed in a sequence of blocks, where each block contains several operations. It is a typical example where, in order to use ROTOR, it is necessary to aggregate all the operations of the same block together. ROTOR therefore decides at the scale of the whole block whether to keep all the data or to delete them all during the forward phase. CHECKMATE, on the other hand, sees the whole graph describing the model and can therefore decide, independently and at the level of each operation, whether to keep its data or not.

The solution we propose is called ROCKMATE; a pseudo-code is provided in Algorithm 3 and explained below. The main idea is to apply **CHECKMATE inside each block** and to apply **ROTOR on the complete sequence of blocks**. For this purpose, it is necessary to obtain the complete graph of all operations of the neural network, and to adapt both CHECKMATE and ROTOR to this new setting.

2.2 Algorithm

2.2.1 Sketch of the Algorithm

ROCKMATE works as follows: given a `torch.nn.Module`, it first generates the whole data-flow graph of forward operations and then it divides it into a sequence of blocks (RK-GB, described in Section 2.3). Based on this, it uses a refined version of CHECKMATE on each block independently to generate sub-solutions (RK-CHECKMATE, Section 2.4). Finally, it combines these sub-solutions with an adapted version of ROTOR to obtain a global solution (RK-ROTOR, Section 2.5). The sketch of ROCKMATE's algorithm is described in Algorithm 3.

The first phase is called RK-GB (for GraphBuilder). It occurs on line 2 of Algorithm 3 and is described in more details in Section 2.3. RK-GB takes as input a model expressed as a PyTorch `nn.Module` and automatically (i) extracts the Directed Acyclic Graph (DAG) of all the operations performed in the model, (ii) divides it into a sequence of blocks and (iii) detects all the blocks which have identical structures. For each unique block, the

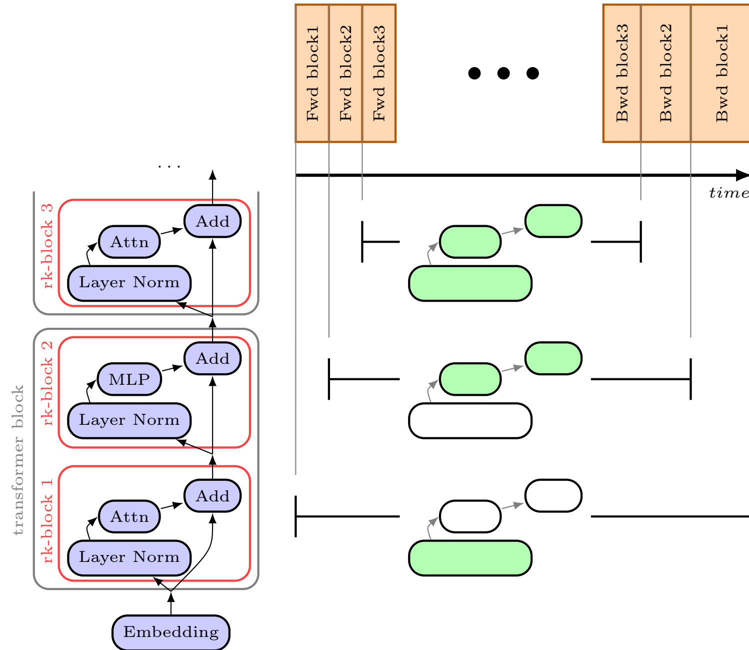


Figure 2.3: Simplified example of running ROCKMATE on a GPT model. **Left:** Dependency graph of the first part of the model, where transformer blocks are shown in gray, and ROCKMATE blocks are identified in red. **Right:** (top) a schedule corresponding to the first three blocks; (bottom) indication of which activations are saved (green) or not (white) for each block, and the intervals during which they are present in memory. Saving fewer activations leads to more recomputation and thus longer backward time.

processing times of all operations and the sizes of all intermediate data that are produced by these operations are measured. These measurements (graphs of each block, labeled with the execution times and memory footprints of the produced data) contain all the necessary information to find the re-materialization sequence.

In the second phase of ROCKMATE (lines 3-7 of Algorithm 3), we consider each single block independently. ROTOR fails to compute very good re-materialization strategies because it can only choose between two options: keep all or delete all activations in the block. In ROCKMATE, we use a refined version of CHECKMATE to generate a larger set of re-materialization strategies. This refined version is denoted as **RK-CHECKMATE** and described in Section 2.4.

A re-materialization strategy is characterized by (i) the memory peak during the execution of the block (either during forward or backward) and (ii) the total size of the internal activations of the block that are kept between the forward phase and the backward phase. The first one ensures that this strategy can be executed within a given memory limit. The second one allows the dynamic program to know how much memory will be left for the next blocks. The number of different options to consider is a parameter of ROCKMATE. Since RK-CHECKMATE is applied at the level of a block (and not on the whole network), the corresponding graph is small enough that the runtime remains small,

Algorithm 3: ROCKMATE

1 **Input:** *module*, *input*, M_{GPU}
2 [*blocks*] = RK-GB (*module*, *input*)
3 *budgets* = $[(M_{peak}, M_{save})]$ (quantized)
4 *sols* = []
5 **for** $b \in$ [*blocks*] **do**
6   **for** $(M_{peak}, M_{save}) \in$ *budgets* **do**
7   *sols*[*b*].add(RK-CHECKMATE (b, M_{peak}, M_{save}))
8 *Sequence* = RK-ROTOR (*sols*, M_{GPU})
9 *rkMod* = CodeGeneration(*Sequence*)
10 **Output:** *rkMod*

even for generating the whole family of strategies. Moreover, as RK-GB automatically detects identical blocks, RK-CHECKMATE is performed only on unique types of blocks (for instance, GPT2 models only involve five unique types of blocks).

The third phase of ROCKMATE (line 8 of Algorithm 3, described in Section 2.5) is called **RK-ROTOR** and computes the global re-materialization strategy. RK-ROTOR features an adapted dynamic program of ROTOR that, instead of having two solutions per block, can exploit the different re-materialization strategies computed during the second phase. The output of RK-ROTOR therefore consists in a schedule which describes which block should be computed, in which order, and with which re-materialization strategy. If necessary, some blocks can be computed without keeping any data at all, and thus be recomputed later (possibly several times).

Finally, this schedule is transformed into a new PyTorch `nn.Module`, which performs all the corresponding elementary operations in the correct order. The resulting module computes exactly the same gradients as the original version while respecting a global constraint on the memory usage of activations, at the cost of duplicating some computations. The execution of the forward phase is based on the Python code obtained via `torch.export`. In ROCKMATE, we detach the operations during the forward phase, so that the full network is represented as many small autograd graphs. This allows the backward operations to be performed separately, thus deletions of tensors can be easily inserted between two backward operations.

2.3 RK-GB, Graph Builder

The computational graph is explicit in TensorFlow, for which CHECKMATE was originally implemented. In PyTorch, however, graphs need to be obtained by certain tools. We developed a tool named rk-GraphBuilder (RK-GB) which takes as input a `nn.Module` and an example input for it, and builds the data-flow graph of the module. Having an example input is necessary to inspect the time and memory cost of all the operations used

during forward and backward phases.

Obtaining the graph rk-GB does not require any modification or annotation of the module source code, instead it uses `torch.export` to trace the forward execution of the module on the example input. This function executes the forward code and provides the list of all primitive operations used. Based on this list, we build a forward graph where each node represents one assignment. However, multiple variables may share the same memory space due to `view` and `in-place` operations in PyTorch. Such variables need thus be kept or removed together when performing re-materialization.

Since checkpointing consists of being able to forget and recompute data, we want each node to represent exactly one data tensor. Due to viewing and in-place operations, different assignments can refer to the same data. Therefore, rk-GB merges all the nodes sharing the same memory space to obtain a simplified forward graph. In a process we call *simplification.* In the simplified forward graph every node consists of one main operation which creates the data, and some secondary assignments concerning the shapes, views or in-place operations. For a 12-layer GPT model, the number of nodes decreases from 934 to 185 after simplification.

The simplified forward graph is further partitioned into a sequence of blocks: each block only depends on the previous block. This sequence is required by rk-Rotor. For a 12-layer GPT model, this results in 26 blocks, where each Transformer layer is separated into a Multi-Head Attention block and a MLP block.

Identical blocks Afterwards, rk-GB goes through all the blocks to recognize *identical blocks*, *i.e.* blocks whose computational graphs are the same. Since rk-GB is deterministic, two blocks representing the same function share the same graph structure, including the same topological ordering of nodes. Following this ordering, rk-GB checks equivalency node by node. A group of identical blocks can be measured and solved together to improve the solving time. Identifying identical blocks is an optimization of the Rockmate solving time but does not change its solution quality. In the case where two *blocks* are topologically identical but defined with different source code, Rockmate could wrongly declare them different. Although, this would not change the memory gains or the computational overhead. For a 12-layer GPT model, this procedure identifies only 5 identical blocks from the 26 rk-blocks produced after separation.

FB-Graph One underlying assumption in the original Checkmate graph model is that each operation has exactly *one* output data. However, when several forward operations share the same input, the corresponding backward operations contribute to the same data (by summing all the contributions). This means that removing the result of one of these backward operations has an impact on the other operations, which can not be taken into account in the graph model of Checkmate. Additionally, some elementary operations in PyTorch actually create intermediate data (they are called `saved_tensors`), which can be deleted independently of the output of the operation.

For these reasons, we introduce a new graph called `FB-Graph`, which contains two categories of nodes: Computation and Data. A `cNode` represents an operation, labeled with the time it takes and the temporary memory overhead during execution. An `aNode` represents a data tensor stored in the memory. An `aNode` can be forgotten to free memory, and restored by recomputing the corresponding `cNodes`. An edge between a `cNode` and an `aNode` represents the execution dependency between the operation and its output data tensors. As discussed in Section 1.3.1, there is a special kind of `aNodes`, which is generated in forward and only used in backward stage. They are called phantoms in this thesis. The phantoms are special in practice because we can avoid generating them with `torch.no_grad()` mode when executing the forward operation. Therefore, they will be handled differently from the other `aNodes`. We also define a loss node of each FB-Graph to represent the operations happening between the forward `cNodes` and the backward ones. If the FB-Graph represents the forward-backward propagation of a PyTorch `nn.Module`, the loss node represents the loss function (usually defined separately) of the output; if the FB-Graph represents a part of the `nn.Module`, the loss node represents not only the loss function, but also the forward and backward operations between the last forward `cNode` and the first backward `cNode`. The benefits of considering such a FB-Graph is to enable finer re-materializations, such as releasing memory from a subset of outputs of one operation. The final product of RK-GB is a sequence of FB-Graphs.

2.4 RK-CHECKMATE

Given a FB-Graph, it is a non-trivial problem to find the optimal execution schedule of all the operations within a given memory limitation. To solve this problem, we use RK-CHECKMATE, an Integer Linear Programming (ILP) adapted from CHECKMATE [25]. Just like CHECKMATE, RK-CHECKMATE requires a topological order of all the operations, which is provided by RK-GB.

RK-CHECKMATE provides several improvements over the original CHECKMATE formulation.

First, additional variables are introduced to represent the execution of each `cNode` separately from the memory allocation of each `aNode`. Constraints are also adapted to ensure that the execution order follows the dependencies between computational nodes and allocation nodes. In the case where one operation generates multiple outputs, there are multiple `aNodes` depending on the same `cNode`. Deleting these outputs is considered separately in RK-CHECKMATE, whereas they are grouped together in the CHECKMATE formulation. For example, this improvement is useful for an operation that produces two large outputs, each required by a different operation: with RK-CHECKMATE, it is possible to delete the second output before performing the operation that requires the first output, which reduces the memory usage.

Second, RK-CHECKMATE takes into account the temporary memory usage of all operations: because of temporary data allocated and deleted during the operation, the peak memory might be higher than the size of input and output. CHECKMATE ignores

this possibility, and thus may produce solutions whose actual peak memory is higher than the budget.

Finally, since RK-CHECKMATE is aware of the separation between forward and backward phases, it is possible to include a constraint on the memory usage when going from the forward to the backward phase. This constraint expresses the limit M_{save} on the size of the activations which are kept in memory between both phases of a block (and thus, during the execution of the following blocks). This memory occupancy is necessary to control the overall memory cost of all the blocks.

In the following sections, we provide the ILP formulation of RK-CHECKMATE.

2.4.1 Optimization Problem

The input to the optimization problem is a FB-Graph and budgets of the memory usage at the end of forward stage and the peak memory usage over the forward and backward stages. The FB-Graph is generated with inspection through real executions, thus each `cNode` has known time and memory overhead cost. The memory overhead of a `cNode` is defined as the difference between the peak memory usage during the execution of the corresponding operation and the memory saved at the end of execution. Similarly, the memory cost of saving each `aNode` is included with the FB-Graph. Within the ROCKMATE framework, the FB-Graph input to RK-CHECKMATE represents only a *block* of the model, but it could be any PyTorch module potentially. The target of RK-CHECKMATE is to find a schedule with the minimal computation time, which executes every node of the FB-Graph at least once with the correct dependencies within the memory budgets.

The FB-Graph input to RK-CHECKMATE is a directed acyclic graph, which contains:

- I allocation nodes $\{a_1, \ldots, a_I\}$ and T computational nodes $\{c_1, \ldots, c_T\}$,
- edges of type $a_i \rightarrow c_k$ and $c_k \rightarrow a_i$ that show dependencies between computational operations and allocation. For example, a_i is used to perform computation c_k, and computation c_k outputs allocation a_i as a result. One computational node can have several incoming allocation nodes and vice versa.

Main variables Following the definition of CHECKMATE, a schedule generated by RK-CHECKMATE is unrolled into T stages where every computational node can be executed at most once during each stage. We use $stage_t$ to represent the period starting after the result of computation c_{t-1} is obtained for the first time and ends when the computation c_t is firstly performed. During one stage, several computations from $\{c_k\}_{k \leq t}$ and deletions might happen.

The solution of the ILP formulation provides a schedule R (low-triangular binary matrix $T \times T$) that determines which computations should be performed during each stage.

$$R_{t,k} = \begin{cases} 0, & \text{otherwise} \\ 1, & \text{if we compute } c_k \text{ during the } stage_t \end{cases} .$$

If k corresponds to the forward nodes, we add a *fast-forward* option as

$$R'_{t,k} = \begin{cases} 0, & \text{otherwise} \\ 1, & \text{if we compute } c_k \text{ during the } stage_t \text{ with } \texttt{torch.no_grad} \end{cases}.$$

Note that we have $R_{t,k} + R'_{t,k} \leq 1$ so that one operation cannot be both fast-forwarded and forwarded normally at the same time. In CHECKMATE, only two options can be selected for a node, representing executing it or not. In ROCKMATE, we add one more option to represent the possibility of executing the operation with `torch.no_grad`. Note that phantoms nodes of the corresponding forward will not be generated at all. Comparing to the case where phantoms are generated and then immediately deleted, the fast-forward option will have lower memory cost within the execution. For the j-th pair of forward and backward operations, we use $fwd(j)$ and $bwd(j)$ to represent the indices of forward and backward nodes. Note that not every forward node has the corresponding backward (some operations do not require gradient), thus $fwd(j)$ is not j in most cases. The size of phantom j is denoted as $|phantom_j|$.

Each stage can be seen as a sequence of steps. During $step_{t,k}$ computation c_k is done (or not if the schedule doesn't require that, i.e. if $R_{t,k} = 0$) and some tensors are deleted.

Also, the solution of ILP provides an information S about allocation nodes saved during each stage. As discussed in Section 1.3.1, there can be multiple `cNodes` contributing the same `aNode`, and all the contributions must be complete before the `aNode` is used as an input. We define variable S to represent the dependencies between `cNode` and `aNode` in the FB-Graph.

$$S_{t,(k,i)} = \begin{cases} 1, & \text{if the contribution of computation } c_k \text{ to tensor } a_i \text{ is saved before starting } stage_t \\ 0, & \text{otherwise} \end{cases}.$$

Consider all edges in the FB-Graph that connect computation nodes with their children allocation nodes,

$$ChildrenOfComp := \{(k,i)|\ a_i \in children(c_k), k = 1, \ldots, T\},$$

and let their number equals $|ChildrenOfComp| = E^{c \to a}$. Then, S can be seen as binary matrix of size $T \times E^{c \to a}$. Since phantom nodes are handled differently, we do not consider them in $c \to a$. The dependencies of phantom nodes will be handled with an additional variable:

$$S^p_{t,j} = \begin{cases} 1, & \text{if phantoms } j \text{ is saved before starting } stage_t \\ 0, & \text{otherwise} \end{cases}.$$

Objectives Given memory budget, ILP finds schedule R, S such that the computational costs

$$\sum_{1 \leq k < t \leq T} C_k R_{t,k}$$

are minimized given feasibility and memory constraints (where C_k is a cost of computation c_k).

2.4.2 Feasibility Constraints

Consider all edges in the FB-Graph that connect allocation nodes with their parent computation nodes

$$ParentsOfAlloc := \{(k,i)|\ c_k \in parents(a_i), i = 1, \ldots, I\}$$

$$ChildrenOfAlloc := \{(k,i)|\ c_k \in children(a_i), i = 1, \ldots, I\}$$

then a set of edges, which connects each allocation node with its children and parent computation nodes, can be expressed as

The following constraints for ILP should hold

$$\sum_{t=1}^{T} \sum_{k=t+1}^{T} R_{t,k} + R'_{t,t} = 0, \tag{2.3}$$

which ensures that we can recompute only operations that have been executed during previous stages.

$$\sum_{e} \sum_{t=1}^{k'-1} S_{t,e} = 0, \tag{2.4}$$

where $e = (k', i), e \in ChildrenOfComp$. It ensures that before the first computation, allocation cannot be saved.

$$\sum_{t=1}^{T} R_{t,t} + R'_{t,t} = T, \tag{2.5}$$

which ensures that c_t is executed at the end of $stage_t$.

$$\sum_{t=1}^{T} R_{t,k_{loss}} = 1, \tag{2.6}$$

where k_{loss} is the index of the node that computes the loss. It ensures that the loss is computed only once during the forward-backward phase. The purpose and function of loss is discussed in Section 1.3.1.

$$S_{t+1,e} \leq S_{t,e} + R_{t,k} + R'_{t,k}, \tag{2.7}$$

where $e = (k,i) \in ChildrenOfComp$ and $t = 1, \ldots, T-1$. It ensures that the results of a tensor can only be generated from the execution of the source computational nodes.

$$R'_{t,k} + R_{t,k} \leq R'_{t,k'} + R_{t,k'} + S_{t,e}, \tag{2.8}$$

where $k \in children(i), e = (k', i), e \in ChildrenOfComp$, and $t = 1, \dots T$. It ensures that all computations c_k which are required for generation of allocation a_i are present, where a_i is an input allocation for computation c_t.

$$S^p_{t,j} \leq R_{t,fwd(j)}, \tag{2.9}$$

phantoms can only be generated by the corresponding forward.

$$R_{t,bwd(j)} \leq R_{t,fwd(j)} + S^p_{t,j}, \tag{2.10}$$

The backward operations will require phantoms to be alive or having forward operations to generate them during the same stage.

2.4.3 Memory Constraints

Note that variable S represents whether the computation of an allocation node is done, but not directly represents the memory allocation of the tensor. We introduce a binary matrix P of size $T \times I$, where

$$P_{t,i} = \begin{cases} 1, & \text{if we have tensor } a_i \text{ in memory at the end of } stage_t \\ 0, & \text{otherwise} \end{cases} .$$

with $\sum_{i=1}^{I} \sum_{t=1}^{k} P_{t,i} = 0$, where $k = \min\{k' | c_{k'} \in parents(a_i)\}$. The tensor should allocate memory if any of the source operations are executed and the result is stored:

$$P_{t,i} \geq S_{t,(k,i)} / |parents(a_i)|, \tag{2.11}$$

$$S_{t,e} \leq P_{t,i}, \tag{2.12}$$

where $e = (k, i) \in ChildrenOfComp$ and $t, k = 1, \dots, T$.

We remind that each stage can be seen as a sequence of steps, such that during one step, one computation (or not if the schedule doesn't require that) and some tensors are deleted.

To represent the presence of certain tensors at different steps $step_{t,k}$ of each stage $stage_t$, we introduce the following two **variables** *create* and *delete*:

$create_{t,i,k} \in \{0, 1\}, \forall k \in ParentsOfAlloc[i]$: whether tensor a_i is created during $step_{t,k}$ at stage $stage_t$.

$delete_{t,i,k} \in \{0, 1\}, \forall k \in (ParentsOfAlloc[i] + ChildrenOfAlloc[i])$: whether tensor i is deleted during $step_{t,k}$ at stage $stage_t$.

Comparing to CHECKMATE *delete* shares the same function as the FREE variable in CHECKMATE as described in Section 2.1.1. However, *create* is introduced only in RK-CHECKMATE because an `aNode` can now have multiple sources. Thus, it is not obvious which source `cNode` created the memory allocation of the `aNode` without knowing the schedule explicitly.

To make it easy to represent the memory occupancy of different tensors, we define an **expression** for $t = 1, \dots, T$ and $(k, i) \in ChildrenOfAlloc \cup ParentsOfAlloc$:

$$alive[t, i, k] = P_{t,i} + \sum_{k' \le k} create_{t,i,k'} - \sum_{k' \le k} delete_{t,i,k'} \in \{0, 1\}.$$

A tensor a_i is either alive or deleted immediately after the computation of parent nodes:

$$alive[t, i, k] + delete_{t,i,k} \ge R_{t,k}, \tag{2.13}$$

A tensor a_i is retained during $stage_t$ if it is alive during the last possible step of $stage_{t-1}$:

$$alive[t, i, k] = P_{t,i}, \tag{2.14}$$

where $k = max(ParentsOfAlloc[i], ChildrenOfAlloc[i])$

A tensor can only be created from the parent computation:

$$create_{t,i,k} \le R_{t,k} \tag{2.15}$$

A tensor should be deleted if it would not be needed or saved in the current stage:

$$delete_{t,i,k} = R_{t,k} * (1 - P_{t+1,i}) * \prod_{k' \in children(a_i) | k' > k} (1 - R_{t,k'}) \tag{2.16}$$

Similar to the FREE variable in CHECKMATE, the formulation of *delete* can be written in a linear form:

$$\begin{aligned} delete_{t,i,k} &\in \{0, 1\} \\ 1 - delete_{t,i,k} &\le \text{ num_hazards } (t, i, k) \\ \kappa \left(1 - delete_{t,i,k}\right) &\ge \text{ num_hazards } (t, i, k) \end{aligned}$$

where

$$\text{num_hazards } (t, i, k) = (1 - R_{t,k}) + (1 - P_{t+1,i}) * \sum_{k' \in children(a_i) | k' > k} (1 - R_{t,k'})$$

and κ is the maximum value that num_hazards can achieve.

No tensor should be alive after the final stage that it is concerned:

$$alive[T, i, k] = 0, \ \ k = max(ParentsOfAlloc[i] \cup ChildrenOfAlloc[i]) \tag{2.17}$$

Let $U_{t,k}$ denotes the memory saved at the end of $step_{t,k}$ during $stage_t$ and M_i is the memory required to store tensor a_i, then

$$U_{t,1} = \sum_{i=1}^{I} M_i P_{t,i} + \sum_{i=1}^{I} M_i \ create_{t,i,1} - \sum_{i=1}^{I} M_i \ delete_{t,i,1}.$$

$$U_{t,k} = U_{t,k-1} + \sum_{i=1}^{I} M_i \ create_{t,i,k} - \sum_{i=1}^{I} M_i \ delete_{t,i,k}.$$

If k is forward and it corresponds to the forward-backward pair j, then

$$U_{t,k} += |phantom_j|(S^p_{t,j} + R_{t,k})$$

.

If k is backward and it corresponds to the forward-backward pair j, then

$$U_{t,k} -= |phantom_j|(R_{t,k})$$

.

The peak memory at $step_{t,k}$ during $stage_t$ is within memory budget:

$$tmpM_k R_{t,k} + U_{t,k} + \sum_{\forall i} M_i delete_{t,i,k} \leq M_{peak} \forall t, t', \tag{2.18}$$

$$U_{k_{loss},k_{loss}} \leq M_{save}, \tag{2.19}$$

where $tmpM_k$ is the temporary memory overhead needed in the computation node c_k. M_{peak} and M_{save} are the given limits of peak memory within the schedule and the save memory at the loss. The values chosen for M_{peak} and M_{save} are discussed in Section 2.6.0.0.1.

2.4.4 Schedule Construction

A feasible schedule can be constructed from the solution of RK-CHECKMATE. With the variables t and k from 0 to $T-1$, each $step_{t,k}$ is considered to add operations to the schedule. $R_{t,k} = 1$ indicates the `cNode`$_k$ is added to the schedule. Also, if $delete_{t,i,k} = 1$ for any i, the deletion of `aNode`$_i$ is added to the schedule. The pseudocode to construct a schedule can be found in Algorithm 4.

2.5 RK-ROTOR

Models like GPT [44] can be considered as a sequence of *blocks* where each *block* consists of several operations. The complexity of the *blocks* introduces the motivation of adapting ROTOR to RK-ROTOR.

Algorithm 4: Construction of schedule based on the RK-CHECKMATE solution

1. Initialize $sched = []$
2. **for** $t = 1, \dots, T$ **do**
3.   **for** $k = 1, \dots, t$ **do**
4.   **if** $R_{t,k}$=*1* **then**
5.   Add **Compute** (`cNode`$_k$) to *sched*;
6.   **for** $i = 1, \dots, I$ **do**
7.   **if** $delete_{t,i,k}$=*1* **then**
8.   Add **Delete** (`aNode`$_i$) to *sched*;
9. **return** *sched*

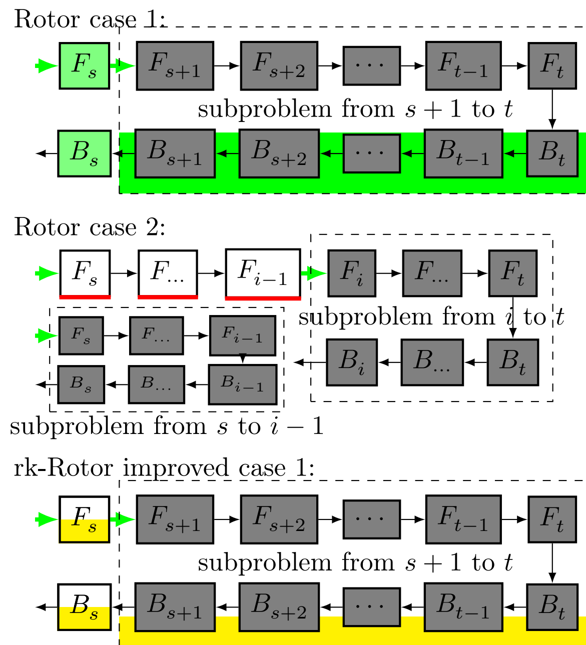


Figure 2.4: Diagram representing the different cases for the dynamic program. Green arrows represent materialized activations. Green, yellow and red *blocks* represent internal activations to the *blocks*, which are respectively completely, partially, or not saved. Colored backgrounds on the subproblems represent how much memory is occupied by these activations.

2.5.1 Formulation Notations

Assume our model is a sequence of L *blocks*, numbered from 0 to $L-1$. For each *block*, we have $1+B$ budget options, where option 0 does not save any intermediate `aNode`, and each of the other B options saves a different amount of `aNodes`. We denote by F_i^o the forward computation of *block* i with option o, and if $o>0$, B_i^o is the corresponding backward computation. Since F_i^0 does not store any intermediate `aNode`, we consider that it does not have a corresponding backward computation.

The input activation of F_i is x_i, and its output is x_{i+1}. Similar to the ROTOR paper [3], for each option $o>0$, we denote by $\bar{x}_i^o$ the union of x_i and of all the intermediate `aNode` generated by F_i^o. For ease of notation, we will also use x_i and $\bar{x}_i^o$ to denote the size of the corresponding `aNode`.

For any computation, we use $tmp(\cdot)$ to denote the temporary memory usage of this computation: this is the amount of memory that needs to be available for this computation to succeed, and that is released afterwards. We also use $r(\cdot)$ to denote the running time of a computation mode. As an example, since the input and output `aNodes` need to be in memory, the memory usage for running F_i^0 is $x_i + x_{i+1} + tmp(F_i^0)$, and this takes time $r(F_i^0)$. Note that both $r(\cdot)$ and $tmp(\cdot)$ depend on the computation mode of the given layer.

We follow the memory persistence assumption 2.1.1 from ROTOR [7], which significantly reduces the complexity of the optimization algorithm by excluding solutions that are rarely optimal in practical cases.

2.5.2 Algorithms

ROTOR In the dynamic programming algorithm of ROTOR, an optimal solution for the forward-backward computation from *block* s to t with memory m can be of two different types: either the first *block* s is computed only once, or more than once. In the first case, the computation starts with computing *block* s and keeping all intermediate `aNodes` that are needed during backward, and continues with an optimal solution for *blocks* $s+1$ to t (with less memory available). In the second case, the computation starts with computing *blocks* s to $s+i$ for some i, stores the result of $s+i$, continues with an optimal solution for *blocks* $s+i$ to t, and finally recomputes from s to $s+i$ with an optimal solution for this part. Note that no intermediate `aNode` is saved for *blocks* s to $s+i$.

In each case, the subproblems that need to be solved have a smaller value of $t-s$. Assuming that the solutions to these smaller problems are known, the algorithm can make the choice that leads to the smallest overhead among all valid choices, *ie* those for which the memory usage is not higher than the budget m. We can thus iteratively compute optimal solutions until we find the solution for the complete model.

RK-ROTOR In the RK-ROTOR context, we have several options for the first case: instead of storing all the intermediate `aNodes`, a selected subset of intermediate `aNodes` can be stored for the first *block* s. Different choices of subsets lead to different memory usage

for storing the intermediate aNodes and for computing the backward operation. There is thus a larger set of options to choose from, but the main idea is still there: assuming that solutions to all smaller problems are known, we can select the option that yields the lowest overhead among all options which respect the memory budget. An illustration of comparing ROTOR with RK-ROTOR is represented on Figure 2.4, where the bottom shows the improved case of RK-ROTOR.

We use the fact that *blocks* are identical to avoid solving several times the same ILP. Although RK-CHECKMATE generates the same set of solutions for the identical *blocks*, they are treated differently in RK-ROTOR, which happen to have the same available options. RK-ROTOR can indeed choose different solutions for these layers. In practice, different solutions are usually chosen: the conditions are different for the first layers of the model compared to the last layers. In addition, one single layer might be recomputed several times. Before the last forward stores (a subset of) intermediate aNodes for backward, the previous forward execution stores no intermediate aNode at all. This is exemplified in Figure 2.3: the left shows the dependency graph, where each color represents a different type of operation. The right shows a possible execution, where a layer is shown transparent if its output is not saved. The bottom-most *block* and the top-most *block* in this Figure are identical (same operations), but they use different solutions on the right (save a different set of aNodes).

2.5.3 DP Formulation

We denote by $\mathrm{Opt}(s,t,m)$ the optimal execution time for computing the sequence from *block* s to *block* t, assuming that the input x_s will be kept in memory. There are two possible cases for the start of this computation:

Case 1 If *block* s is only computed once in this sequence, then it is computed with one of the F_i^o options for $o > 0$ so that it is possible to perform the backward computation. This requires to have at least $tmp(F_i^o)$ available memory for the forward and at least $tmp(B_i^o)$ available for the backward. The corresponding execution time is $r(F_i^o) + r(B_i^o)$, and the memory available for the rest of the computation is $m - \bar{x}_i^o$. The best choice is given by:

$$\mathrm{Opt}_1(s,t,m) = \min_{\text{valid option } o} r(F_i^o) + r(B_i^o) + \mathrm{Opt}(s,t,m-\bar{x}_i^o). \tag{2.20}$$

In this equation, an option is considered valid only if the temporary memory requirements for the forward and backward computations are satisfied. Note that memory requirements for any option generated by RK-CHECKMATE is known.

Case 2 If *block* s is computed more than once, then its first computation does not need to keep any intermediate aNode. It is thus computed with F_s^0, and the choice now is about which is the next activation to be kept in memory. Let us denote by i the index activation kept in memory, so that activations $x_{s+1}, x_{s+2}, \ldots, x_{i-1}$ are discarded just after being used. It is possible to compute x_i by performing $F_s^0, F_{s+1}^0, \ldots, F_{i-1}^0$. Once this

activation is computed and stored in memory, optimizing the rest of the computation becomes a subproblem: we need to compute the optimal execution time from *block* i to t. Afterwards, since no activation was stored between *blocks* s and i, this corresponds to another subproblem, from s to i. The best choice is given by:

$$\mathrm{Opt}_2(s,t,m) = \min_{\text{valid } i \text{ with } s<i<t} r(F_s^0)+r(F_{s+1}^0)+\cdots+r(F_{i-1}^0)+\mathrm{Opt}(i,t,m-x_i)+\mathrm{Opt}(s,i,m) \tag{2.21}$$

In this equation, a choice is considered valid if the temporary memory requirements for all computations $F_s^0, F_{s+1}^0, \dots, F_{i-1}^0$ are satisfied.

In both cases, if there is no valid choice, the corresponding min value is considered to be $+\infty$. Finally, the optimal decision for our problem is computed with:

$$\mathrm{Opt}(s,t,m) = \min\big(\mathrm{Opt}_1(s,t,m), \mathrm{Opt}_2(s,t,m)\big) \tag{2.22}$$

Additionally, if $s = t+1$, only the first case can be considered, but this time the rest of the computation is empty. We can thus compute $\mathrm{Opt}(s, s+1, m)$ for all s and all m. The resulting algorithm is close to the ROTOR algorithm, using the updated Equation (2.20), and is provided in Algorithm 5.

Algorithm 5: RK-ROTOR for L *blocks* with memory m.

1 **for** $m = 1, \dots, M$ **do**
2   **for** $k = 1, \dots, L$ **do**
3   **for** $s = 1, \dots, L+1-d$ **do**
4   Compute $\mathrm{Opt}(s, s+k, m)$ with Equation (2.22)

5 **return** *RK-ROTOR-Build*$(Opt, 1, L+1, m-x_0)$ *(See* `Alg. 6`*)*

Algorithm 6: RK-ROTOR-Build(Opt, s, t, m) – Computation of the schedule

1 **if** $Opt(s,t,m) = \infty$ **then**
2   **return** *Infeasible*

3 **else if** $s = t+1$ *and* $Opt(s,t,m) = Opt_1(s,t,m)$ *with option* o **then**
4   **return** (F_s^o, B_s^o)

5 **else if** $Opt(s,t,m) = Opt_2(s,t,m)$ *with choice* i *(Equation* (2.21)*)* **then**
6   **return** $(F_s^0, F_{s+1}^0, \dots, F_{i-1}^0,$ *RK-ROTOR-Build*$(Opt, i, t, m - x_i)$, *RK-ROTOR-Build*$(Opt, s, i, m))$

7 **else**
8     $o \leftarrow$ option such that $\mathrm{Opt}(s,t,m) = \mathrm{Opt}_1(s,t,m)$ (Equation (2.20))
9   **return** $(F_s^o,$ *RK-ROTOR-Build*$(Opt, s+1, t, m - \bar{x}_s^o), B_s^o)$

2.6 Performance analysis

Complexity With a model that contains L *blocks*, and a memory of size M, the ROTOR algorithm has a complexity in $O(L^3 M)$: for each value of s, t and m, there are $O(L)$ choices to consider. In the ROCKMATE case, with B budget options, the dynamic programming algorithm considers $O(L+B)$ choices at each step, and thus has a complexity in $O(L^2 M(L+B))$.

Sub-optimality of the solution Although both RK-CHECKMATE and RK-ROTOR obtain optimal solutions for the given sub-tasks, the final ROCKMATE solution is not always optimal on the overall network. Two reasons can lead to sub-optimality in ROCKMATE: (i) since the number of memory budgets is finite, only a limited number of execution schedules are produced by RK-CHECKMATE. (ii) in RK-ROTOR, intermediate `aNodes` are only stored to improve the execution time of the backward phase. However, if the forward phase of a *block* is executed several times, it might be beneficial to store some intermediate tensors on the first pass, and use it to compute the output faster on subsequent passes. This possibility is not considered in RK-ROTOR: forward passes in case 2 do not save any intermediate `aNode`.

2.6.0.0.1 Budget Selection For each block, RK-CHECKMATE will be applied with different values for the memory budgets M_{peak} and M_{save}, as explained Algorithm 3. We first compute the minimum and maximum possible values for M_{peak}. The maximum value M_{peak} is the same as the peak memory usage of the PyTorch autograd schedule, since the memory usage of our re-materialization schedule should not exceed the original module. The minimum value of M_{peak} is chosen as the maximum temporary memory usage over all `cNodes`. Note that this minimum value may not be feasible, and we do not have an efficient algorithm to find the minimal feasible M_{peak} for any given FB-Graph. The number of budgets is a hyperparameter of ROCKMATE. The values of M_{peak} are **evenly** spaced within $[min_peak; max_peak]$. Given one value for M_{peak}, the values of M_{save} are evenly spaced within $[output_size; M_{peak}]$. This ensures that all pairs (M_{peak}, M_{save}) given to RK-CHECKMATE are relevant. Note that the number of schedules generated from RK-CHECKMATE can be smaller than the number of given budget pairs for two reasons: (i) different budgets may lead to the same optimal solution, especially when the pairs of (M_{peak}, M_{save}) have similar values. In this case, the same solutions will be combined as one. (ii) some selected M_{peak} may not be feasible for the given FB-Graph, leading to no solution for the corresponding budget pairs.

The identical blocks are solved only once with the same budgets, so that all identical blocks have the same set of block-level execution options provided to RK-ROTOR. However, RK-ROTOR sees all of these blocks as different parts of the sequence, which just happen to have the same set of options. In the resulting sequence, each identical block may be executed with a different option in the output of RK-ROTOR.

2.7 Experiments

In this section, we present empirical results to demonstrate the capability of ROCKMATE on different models and machines.

Two types of machines are used in our experiments: an NVIDIA Tesla V100 16GB GPU with a 32-core Intel Skylake CPU, or an NVIDIA Tesla P100 16GB GPU with 32-core Intel Broadwell CPU. All the training computations are performed with the GPU, thus all the experiments will refer to the GPU types in the latter sections. Since our experiments are conducted thoroughly on the same machine, the ILP/DP solving are corresponding to the CPU type. We use the open source PULP library to build the ILP models, but the commercial solver GUROBI is called to solve them.

For each experiment, we report the average iteration time of 15 iterations with standard deviations. The measured memory cost includes only the activations. In practice, we measure the size of the model parameters and their gradients, and remove them from the real peak memory measured by `torch.cuda.max_memory_allocted()`. Without specified, all the model parameters and sample data are in single precision (float point 32).

2.7.1 Budgets Selection

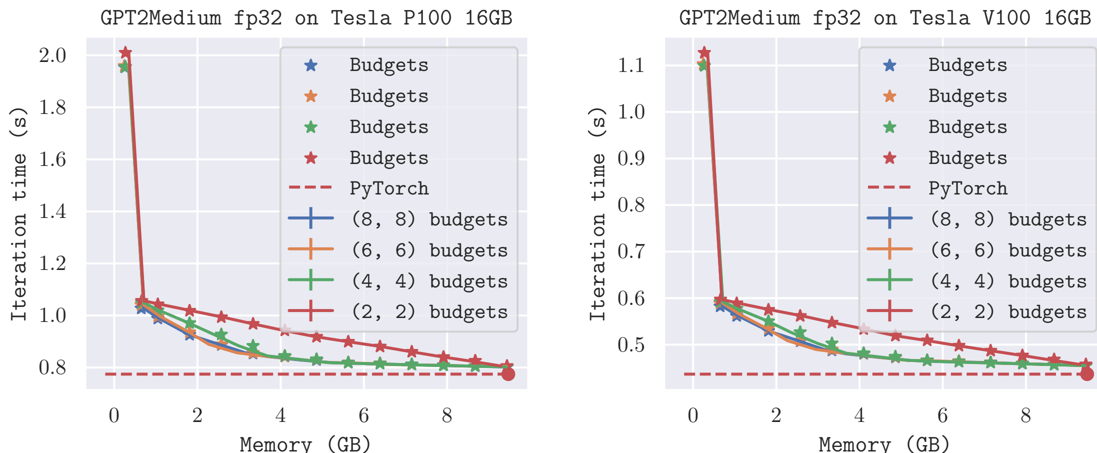


Figure 2.5: Iteration time with recomputation overhead versus peak memory required by activations. For different plots, we control the number of peak memory and save memory of RK-CHECKMATE solving. For example, (6,6) means 6 peak budgets each combined with 6 save budgets.

The key difference between ROTOR and ROCKMATE is that ROCKMATE considers different options to execute a *block*. The number of options is controlled by the number of budgets used in RK-CHECKMATE, which directly affect the times RK-CHECKMATE is called and the overall solving time. Figure 2.5 shows how the number of budget options

influences the quality of the ROCKMATE solution. We observe that even for the networks with complicated *blocks*, the budget selection of (6,6) should allow RK-CHECKMATE to find fine-grained options for RK-ROTOR. Even though raising the number of budgets will increase the RK-CHECKMATE solving time, we choose to use higher number of budgets to guarantee performance of ROCKMATE. In the rest of the chapter, we use the number of budgets as (10,10) in other experiments. In the next section, we study the results of ROCKMATE efficiency where we find the RK-CHECKMATE solving time is currently acceptable, where proper efficiency can be achieved through this setting.

2.7.2 Performance vs. Solving Time

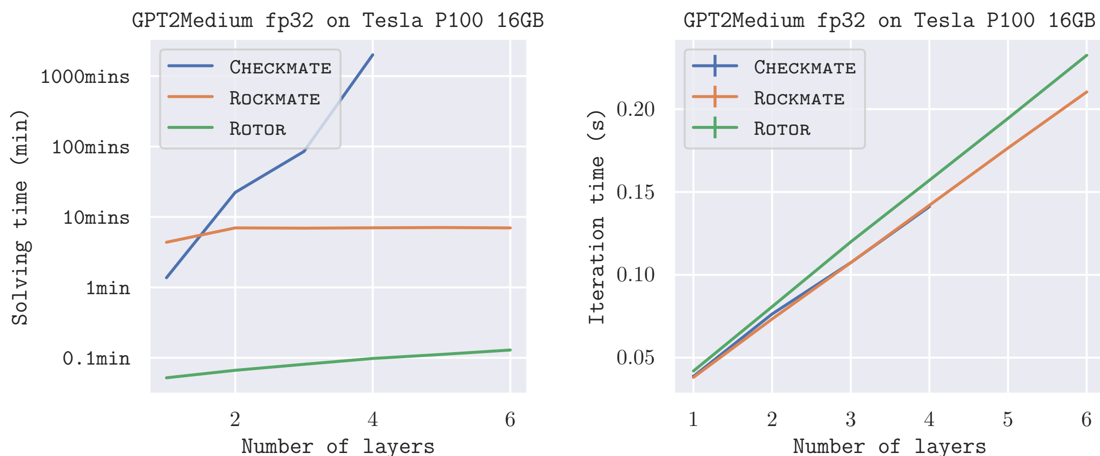


Figure 2.6: Iteration time in logarithmic scale with recomputation overhead versus peak memory required by activations. Budgets are set as the half of maximum peak memory usage for PyTorch execution. For this experiment, ILP solving time is not limited.

In this section, we compare the solving time and performance of ROCKMATE, CHECKMATE and ROTOR on GPT2 networks with 1-6 layers. For each network, we choose the budget as half of the peak activation memory used in PyTorch execution. No limitation of ILP solving is set in this experiment, and CHECKMATE is solved until the optimal solution is found.

For all three algorithms, we present the solving time including all the procedures in the framework, including RK-GB inspection time to obtain the time and memory cost of the operations. For ROCKMATE, both solving time of RK-CHECKMATE and RK-ROTOR are included in the reported solving time. As shown in Figure 2.6, ROCKMATE achieves very similar throughput to CHECKMATE with the same budget, while ROTOR is worse. The solving time of CHECKMATE is exponential in the number of blocks, exceeding 30 hours on the 10-blocks GPT2. On the other hand, the solution time of ROCKMATE remains almost constant because the same RK-CHECKMATE models are applied to all identical

Transformer blocks. Despite the significant difference in solution time, ROCKMATE achieves similar or better overhead than CHECKMATE within the same budget.

2.7.3 Performance on Different Networks

In this section, we compare the performance of ROCKMATE and ROTOR on different networks and machines over a range of memory budgets. Figure 2.7 and 2.8 show the computational overhead in terms of peak memory usage during the forward-backward computations. For the same memory peak, ROCKMATE has a lower overhead than ROTOR in most cases. For ResNet models, ROCKMATE does not show a significant improvement over ROTOR, especially when the neural networks are deep enough, in which cases ROTOR has more re-materialization options. On the other hand, ROCKMATE shows much better performance than ROTOR on GPT2 networks. For GPT2-large, it is noteworthy that ROCKMATE saves 50% memory by introducing only 5% overhead, while ROTOR has more than 10% overhead for the same budget. In addition, ROCKMATE allows training with a smaller memory budget.

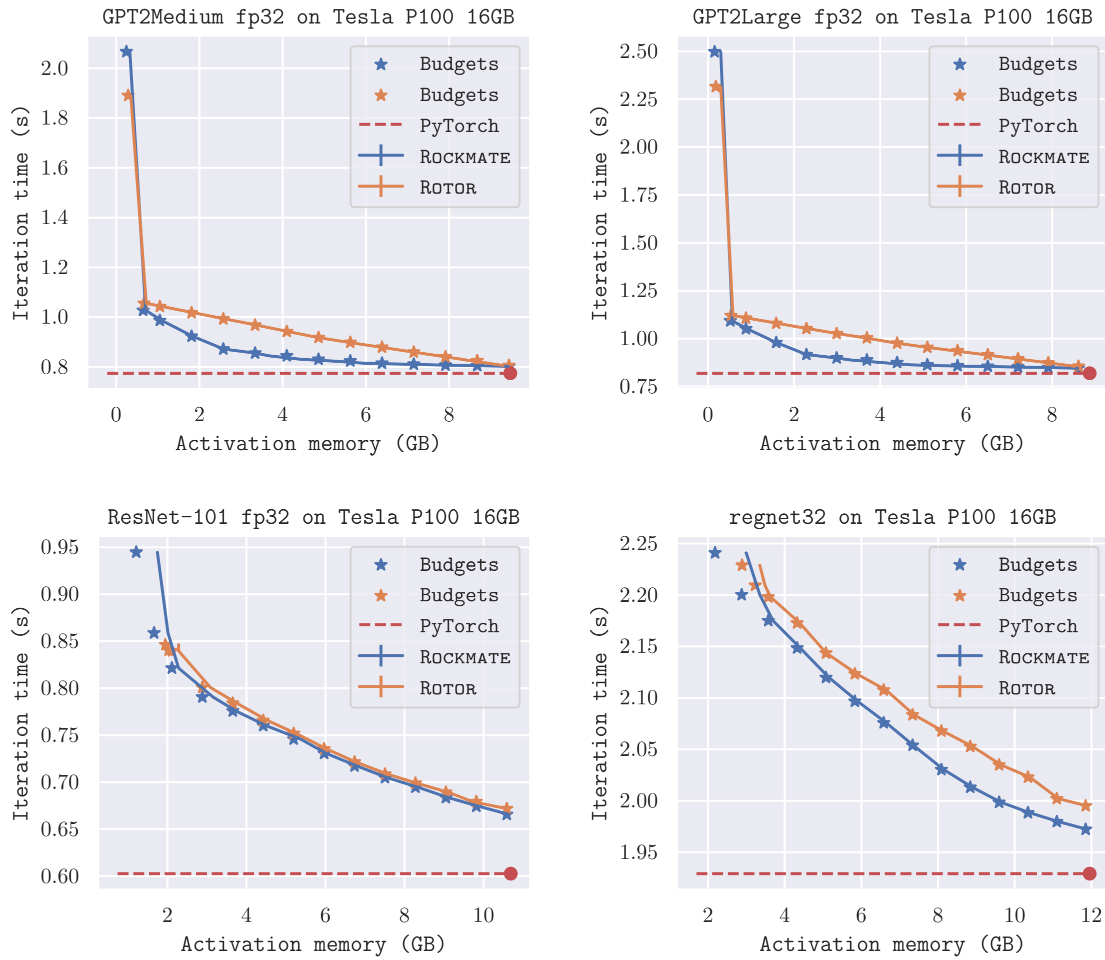


Figure 2.7: Iteration time with recomputation overhead versus peak memory required by activations. The experiments are conducted on NVIDIA Tesla P100 GPU.

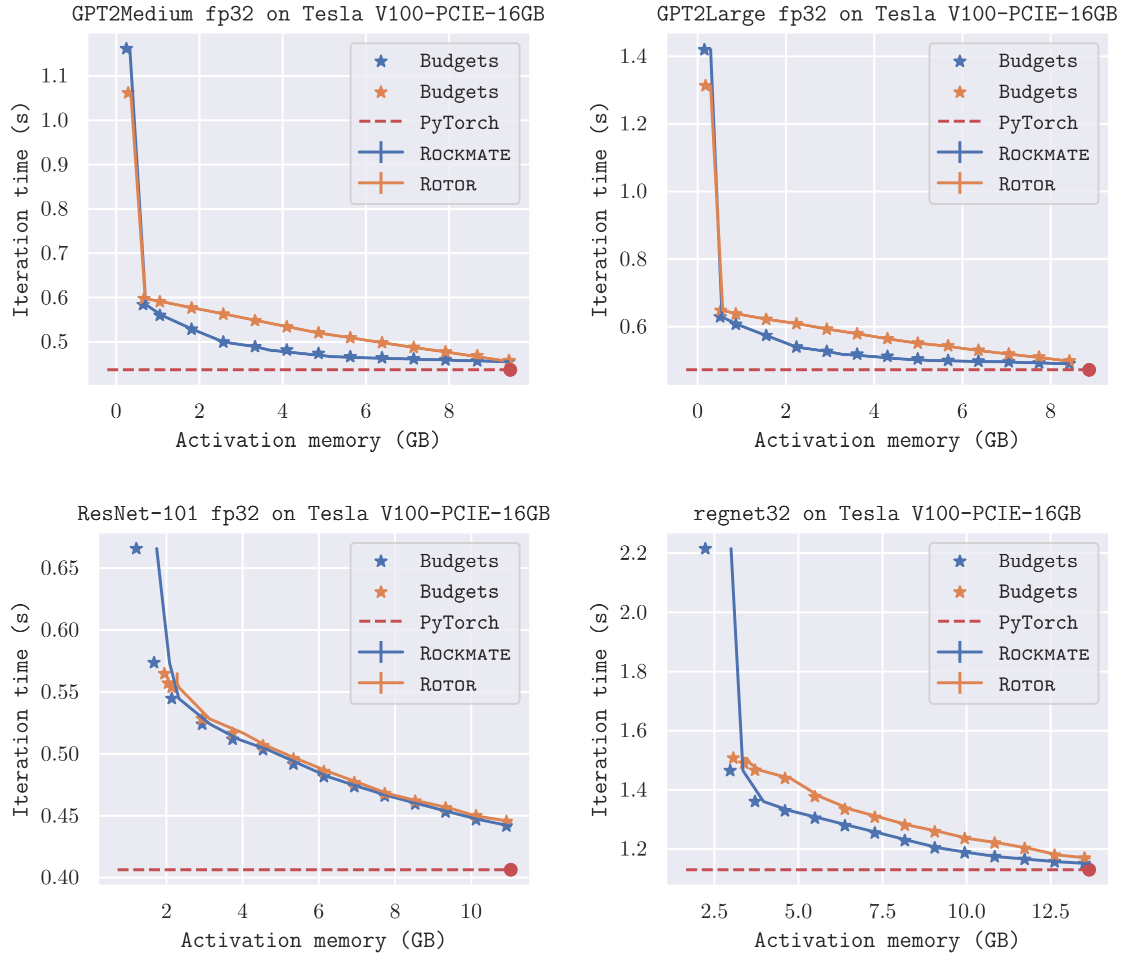


Figure 2.8: Iteration time with recomputation overhead versus peak memory required by activations. The experiments are conducted on NVIDIA Tesla V100 GPU.

The reason why ROCKMATE significantly outperforms ROTOR is that there are "cheap" operations inside a Transformer block, such as `dropout` and `gelu`. The tensors generated by these operations consume a lot of memory, but there is almost no cost to recompute these operations. Because ROTOR rematerializes one block at a time, it cannot take advantage of the "cheap" operations to optimize performance. ROCKMATE works particularly well on models with a sequential-like structure, where each part complicated structure. Also note that a minimum feasible budget is applied to every model, which is lower for ROCKMATE than for ROTOR. With more memory available, ROTOR finds a solution with less time consumed.

2.8 Conclusion

In this chapter, we propose ROCKMATE, a fully automatic tool that takes as input a PyTorch model in the form of a `nn.Module` and a memory limit for activations and automatically generates another `nn.Module`, perfectly equivalent from the numerical point of view, but that fulfills the memory limit for activations at the cost of a small computational overhead. This work is published at the International Conference on Machine Learning conference 2023 and was accepted as an oral presentation [66]. Through experiments on various models, we show that the computation time of the resulting `nn.Module` is negligible in practice and that the computational overhead is acceptable, even for drastic reductions in memory footprint. ROCKMATE is therefore a tool that can transparently allow increasing model size, data resolution and batch size without having to upgrade GPUs. This work opens several new scientific questions. First, ROCKMATE is very efficient for graphs that can be written as a sequence of blocks, which corresponds to numerous models in practice but not to all of them, which raises the question of its extension to any type of graph. Then, the combination of ROCKMATE with data parallelism is trivial, but the question of finding a partition of the model adapted to model parallelism that balances well the computational load and the memory footprint on the different nodes is also an open problem.

Chapter 3

HIREMATE

As we have observed, training modern neural networks poses a significant memory challenge, as storing intermediate results during the forward and backward passes requires considerable memory resources. To address this issue without affecting model accuracy, re-materialization techniques have been introduced to recompute selected intermediate results instead of storing them, thus fulfilling the memory size constraint. ROCKMATE introduced in the last chapter can efficiently generate re-materialization schedules which satisfy the memory constraints with minimized time overhead. However, ROCKMATE is adapted to sequential structure: the size of a *block* is not limited in the framework, which raises uncertainty for the solving time when applying RK-CHECKMATE to it. In this chapter, we introduce a new framework HIREMATE, based on a new *hierarchical* approach that provides generality and quality: we can handle any class of network graphs and satisfy the memory constraint with a low computational overhead during training. The framework exhibits low algorithmic complexity, making it possible to scale up and handle very large models. The framework automatically builds a dataflow graph from a PyTorch model, decomposes the graph hierarchically, and then builds an nn.Module that executes forward and backward passes within the given memory budget.

3.1 Introduction

Modern Neural Networks (NN) undergo several important evolutions which have consequences on the computation and memory requirements, from the first vision networks like ResNet-50 [62] to Natural Language Processing transformer-based models [58] like GPT. The increasing size of models and the resolution of data pose significant challenges for storing both weights and activations. Initially, models had chain-like structures, such as sequences of convolutional layers. These evolved into chains of complex blocks, like chains of Transformer blocks in GPT-like models. Recently, neural networks exhibit increasingly complex dependency graph structures between layers, such as UNO [59], which is structured as a U-Net of complex blocks, and encoder-decoder transformers [58], which feature very long skip connections.

Re-materialization is an efficient technique to limit the memory requirements related

to activations during training. As discussed in Chapter 2, the idea of re-materialization is to avoid storing all the necessary activations because of the subsequent dependencies during the Forward phase and the Backward phase (those related to the backpropagation mechanism). Some activations are computed and then deleted to free up memory, and they will be re-computed later when needed. For simple chains [4] and chains of complex blocks (see Chapter 2), re-materialization has shown its efficiency: it is often possible to save 50% of memory for a computational overhead of about 10 to 15%.

In practice, training hardware often has fixed memory limit. Consequently, the practical objective is to minimize the computational time required to execute the forward and backward passes while adhering a predefined memory budget. This problem has been proven NP-Hard in [43] in the case of general dependency graphs represented as general data-flow graphs. Some solutions, such as TW-REMAT [32] or CHECKMATE [24] have nevertheless been designed to deal with the case of general graphs. However, both face limitations related to the computational cost and scalability of the algorithm that generates the re-materialization strategy, as well as the overhead introduced during the training phase. Another line of research is to propose re-materialization strategies whose computational cost and overhead are controlled, as in ROTOR [4] and ROCKMATE introduced in Chapter 2. These strategies, while effective, are generally limited in scope as they are designed for specific classes of dependency graphs where a chain structure (or potentially complex blocks) can be identified.

The work presented in this chapter is at the convergence of these research lines and we propose a computationally efficient, low-overhead solution that can address general graphs. Our framework HIREMATE is based on a **hierarchical decomposition** approach of the computation graph, to find a re-materialization strategy for **any graph of dependencies** between layers. When the graph is too large to be handled directly by Integer Linear Programming approaches, we decompose it into a graph of complex clusters, potentially with several levels in the hierarchy to manage very large graphs. Efficient solutions for different memory budgets are generated for each of the clusters at the bottom of the hierarchy, using different approaches from the literature. Then, we provide a **new** Integer Linear Programming (ILP) formulation to efficiently recombine these low-level solutions into candidate solutions for the higher levels of the hierarchy. Furthermore, HIREMATE is fully compatible with the `autograd` mechanism of PyTorch, so that no modification of the code is required to use it. With a single line, the user can automatically control the memory usage of their neural network: `model = HRockmate(model, sample, memory_budget)`.

To achieve this result, we rely on the following main contributions:

- A data-flow graph decomposition algorithm **H-Partition** that builds a hierarchy of blocks of reasonable sizes (to keep an acceptable computational complexity) while minimizing the memory size of the interfaces between the blocks.
- A new linear programming solver H-ILP adapted to this hierarchical decomposition.
- A general **framework** for integrating any existing (or future) re-materialization strategy at any level of the hierarchy, and combining their strengths.

The rest of the chapter is organized as follows. The framework of HIREMATE is

presented in Section 3.2 which covers graph decomposition, partial problems resolution and global re-materialization strategy reconstruction. Section 3.3 introduces the partitioning algorithm in details. Section 3.4 covers the ILP formulation of our major solver H-ILP. The experimental results of HIREMATE are presented in Section 3.5.

3.2 HIREMATE

Problem statement We associate each neural network with a *data-flow graph*, where each node represents a tensor-level computation. A *schedule* is a sequence of elementary computations and tensor deletions that performs the forward and backward passes for one training iteration.

Let us consider a model with L layers, where layers may have non-sequential dependencies (the i-th layer may depends on the j-th layer with any $j < i$.) Let F_i and B_i denote the forward pass and backward pass associated with layer i and D_i denotes the deletion of the output of F_i. We assume a user-defined loss function L is conducted between the forward and backward pass of the last layer. Then, for a model that is a sequence of 3 layers, a schedule corresponding to the standard autodiff training can be written as $F_1F_2F_3LB_3B_2B_1$, while $F_1F_2D_1F_3LB_3D_3F_1F_2B_2D_2F_1B_1$ is another valid schedule, yielding a smaller peak memory since it does not require storing all intermediate activations simultaneously. **The re-materialization optimization problem is to find a schedule whose peak memory is below a given budget and whose execution time is as small as possible.**

Building on the ideas presented in ROCKMATE introduced in Chapter 2, we propose the following assumption: **The global optimal or near-optimal re-materialization schedule can be constructed by combining re-materialization schedules from different parts of the graph.** While ROCKMATE focuses on sequential networks, we extend this idea to non-sequential networks, introducing two key challenges: (1) partitioning the graph into distinct components, and (2) integrating the local re-materialization schedules into a cohesive global schedule. The first challenge is relatively straightforward in sequential models but requires more sophisticated solutions in non-sequential ones. Similarly, RK-ROTOR addresses the second challenge efficiently in sequential models, but not in non-sequential ones. In the following sections, we briefly introduce the approaches in HIREMATE to tackle both challenges for graphs of arbitrary structure.

Figure 3.1 outlines the main steps of the HIREMATE approach. In the first step, using the graph building tool adapted from Chapter 2, a data flow graph of the input module is obtained. In the second step, the **H-partition** algorithm (see Section 3.2.1) recursively partitions the graph into subgraphs of manageable sizes. Steps 3, 4, and 5 of Figure 3.1 describe the **H-Solver** algorithm (see Section 3.2.2), which builds a training schedule. Starting at the lowest level of decomposition, the algorithm computes schedules for each subgraph under different memory budgets to explore various time-memory tradeoffs, providing several *options* (*i.e* ways to (re)compute operations and store activations during

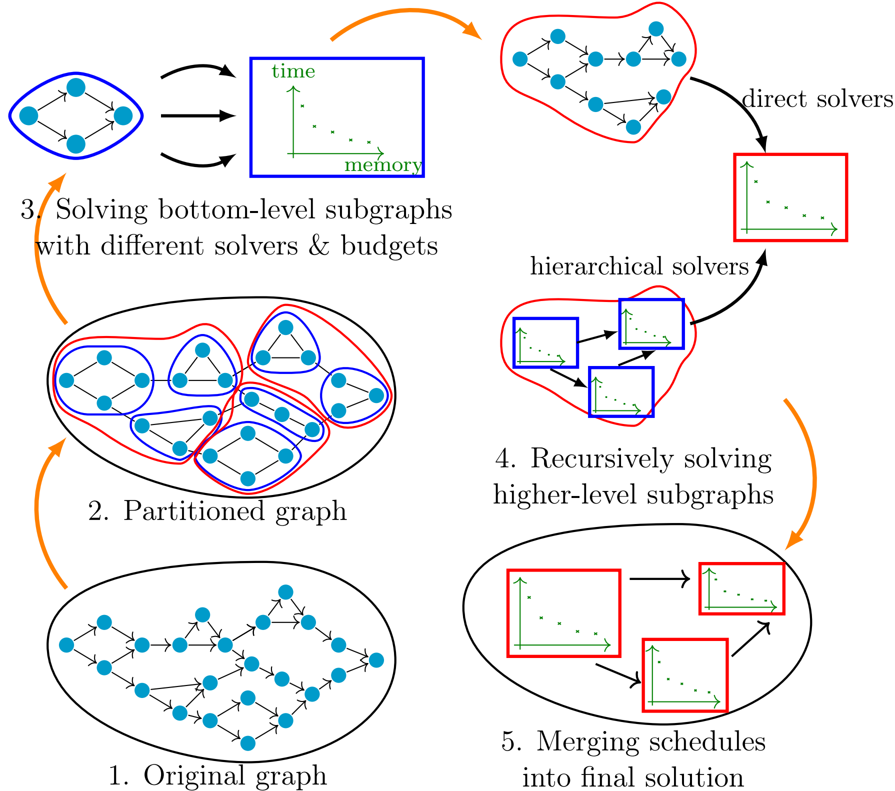


Figure 3.1: **HIREMATE** takes a PyTorch `nn.Module` and creates a `nn.Module`, which provides same outputs while satisfying peak memory budget constraints B during training. The solving procedure of HIREMATE can be divided into the following steps: (1) Obtain data-flow graph G from the PyTorch module. (2) Recursively partition G with **H-Partition** (Section 3.2.1) into small-size subgraphs. (3)-(5) Obtain a re-materialization schedule from G with budget B using **H-Solver** (Section 3.2.2). (6) Produce a new `nn.Module` whose execution follows the schedule.

forward and backward passes through the subgraph) for the nodes at higher levels. This procedure continues until the top-level graph is solved (*i.e* a schedule is found) for a single memory budget corresponding to the overall memory available for activations.

Note that the current implementation of the general scheme described above and in Figure 3.1 is fully modular. While an efficient and effective approach is implemented and tested for both partitioning and solving, we encourage users to propose new algorithms and apply them within the HIREMATE framework.

3.2.1 Partition Framework

In this section, we introduce the process of partitioning the computational graph. In the ROCKMATE context, a computational graph consists of two types of nodes: computation nodes (`cNode`) that represent PyTorch atomic operations and allocation nodes (`aNode`)

that represent PyTorch tensors. In HIREMATE, we address the challenge of managing large graphs by grouping nodes to create a more abstract representation of dependencies. The goal of HIREMATE's partitioning step is to reduce the size of the problems to be solved without compromising the overall solution quality. The outcome of this step is a hierarchical decomposition into subgraphs, where each node at a given level represents an set of nodes at the level below.

Representation Instead of keeping the computational graph with thousands to tens of thousands of nodes, we seek for representation of graphs with tens of nodes. To achieve that, we define the following concepts to establish the partitioning framework:

Definition 3.2.1. `H-Cluster`: a set of PyTorch forward operations and their corresponding backward operations.

Definition 3.2.2. `H-Graph`: a directed bipartite graph contains `cNodes` and `aNodes`, where each edge connects a `cNode` and a `aNode`. A `cNode` represents either a single PyTorch atomic operation or a set of PyTorch atomic operations, and a `aNode` represents either a single PyTorch tensors or a set of PyTorch tensors. The inputs of a `cNode` are all the `aNodes` which are used in the operations within the `cNode`, and the outputs of a `cNode` are all the `aNodes` produced by the operations within the `cNode`.

Within this chapter, we slightly update the definition of `cNode` and `aNode` which were defined for FB-Graphs in Section 1.3.1. The major difference is about the granularity of their representations. In ROCKMATE, a `cNode` represents the smallest group of computation operations obtained through the RK-GB, while in this chapter we intentionally merge several operations together and represent them as one node in some `H-Graphs`. Also, one `H-Cluster` can be partitioned in different ways, and thus can be represented by multiple `H-Graphs`. Our goal is to develop the algorithm that can represent a `H-Cluster` with thousands to tens of thousands of operations using a `H-Graph` with only tens of nodes, while preserving the ability to find a well-performing schedule. Figure 3.2 provides an example of one `H-Cluster` being partitioned into two different `H-Graphs`.

Forward and backward Typically, a `H-Cluster` contains both forward and backward parts. [1] The forward and backward parts of the `H-Cluster` are represented by a pair of `cNodes` in the higher level `H-Graph`. As illustrated in Figure 3.2, the inputs and outputs of a forward `cNode` correspond to all the inputs and outputs of PyTorch operations it contains. Within a `H-Cluster`, many tensors are created by forward `cNodes` and used **only** by the backward `cNodes`. They are handled together in a highly efficient manner, which is achievable only when the forward and backward `cNodes` are paired.

The `cNodes` in `H-Graph` are ordered topologically. In a ROCKMATE graph, $\texttt{cNode}_i$ does not depend on the output of $\texttt{cNode}_j$ when $i < j$. Similarly, in a `H-Graph`, we maintain the property for `cNode`:

[1] An exception occurs when all the forward operations in the `H-Cluster` do not require gradients, resulting in no backward operation being included.

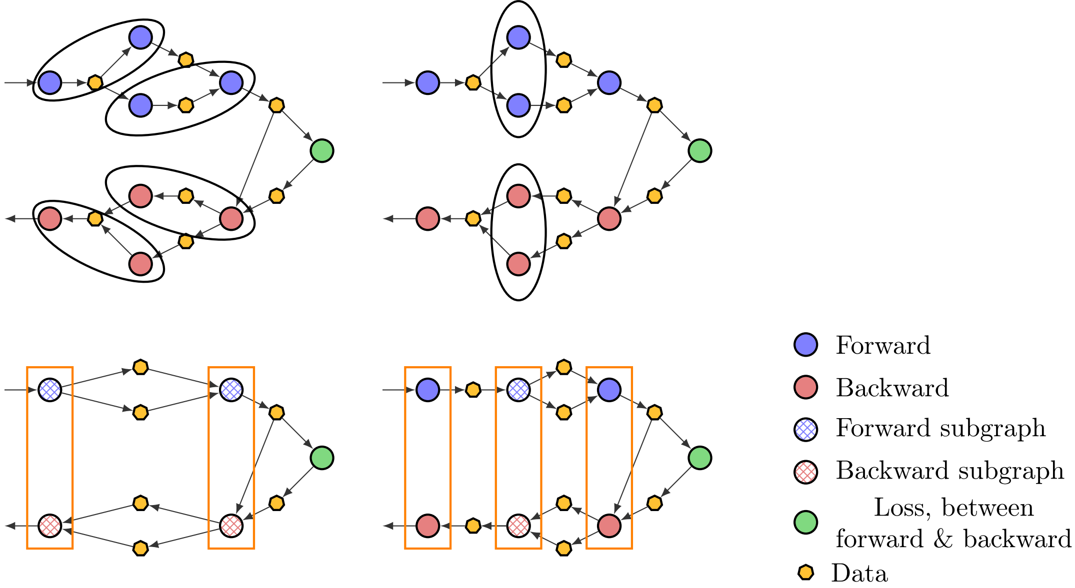


Figure 3.2: One `H-Cluster` can be partitioned in different ways as shown in the top figures. They will make different `H-Graphs` as shown in the bottom figures. Rectangles in the bottom figures show the clusters of the higher level.

Property 1. For $\texttt{cNode}_i$ and $\texttt{cNode}_j$ with $i < j$, we denote their corresponding `H-Clusters` as C_i and C_j. There is no forward operation f_a in C_i and f_b in C_j such that f_a depends on the output of f_b. Also, there is no backward operation b_a in C_i and b_b in C_j such that b_b depends on the output of b_a.

Loss node Similar to the *block* in ROCKMATE, every `H-Graph` in HIREMATE contains a loss node, which represents the operations that happen between the end of forward part of the graph and the beginning of the backward part. Figure 3.3 shows the example of one `H-Graph` and what its loss node represents. Viewed as a single node in the current subgraph, loss node acts as a placeholder between forward and backward operations, while excluding the dependencies outside the graph.

Property 2. There is one and only one loss node in a `H-Graph`. It is a descendant of all the forward `cNodes` and an ancestor of all the backward `cNodes`.

In the latter sections, we introduce the detailed algorithms used for partitioning the graph.

3.2.2 Solving Framework

In this section, we introduce the framework of combining re-materialization schedules hierarchically. A `H-Cluster` contains a subset of the computational operations. Similar to the schedule defined in the ROCKMATE, we define a re-materialization schedule as:

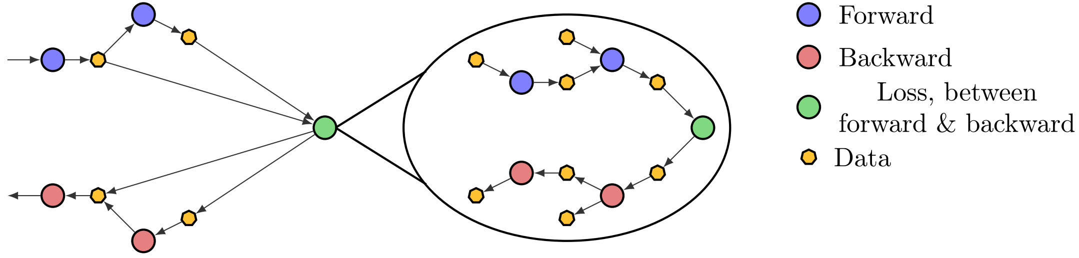


Figure 3.3: Visualization of the leftmost `H-Cluster` in the bottom of Figure 3.2

Definition 3.2.3. Schedule: a sequence of `cNodes` and `aNodes`. An `aNode` is alive when all the input `cNodes` have been executed, and is removed when itself appears in the sequence. A schedule is valid if: (1) every `cNode` in the graph appears at least once in the sequence; (2) whenever a `cNode` is computed, all its input `aNodes` are alive. When all the `cNodes` in the sequence represent only one PyTorch operation, the schedule is called a bottom schedule.

The solving framework of HIREMATE aims to provide an effective re-materialization schedule for the main `H-Cluster`, which contains all the PyTorch operations in the model. There are two approaches to build the re-materialization schedule for the computational graph:

1. Construct from all computational nodes it contains, requiring no partitioning;
2. Combining the re-materialization of the subsets of the graph, requiring a partitioned subsets of computational nodes.

The first approach, requiring no hierarchical structure, can be implemented using existing re-materialization algorithms like Rotor [7] and Checkmate [24]. The simplest method for creating a re-materialization schedule for a computational graph is to compute each node (representing one PyTorch atomic operation) exactly once, with tensors being deleted as soon as they are no longer needed. This naive re-materialization approach represents the PyTorch autograd implementation.

When the number of computational nodes becomes too large for a solver to efficiently handle, the second approach become a more viable alternative. We first demonstrate that combining re-materialization schedules from subgraphs is an effective strategy.

Definition 3.2.4. Interface nodes $Inter(G)$: given a `H-Cluster` C, $Inter(C)$ are all the `aNodes` that have dependencies with `aNode`, which do not belong to C. They can also be seen as the inputs or the outputs of the `H-Cluster`.

Lemma 3.2.1. *Assume a graph G is partitioned into sorted `H-Clusters` $C_1, \dots, C_m$ with property 1, and $S_1, \dots, S_m$ are valid bottom schedules of $C_1, \dots, C_m$ containing no deletion of $Inter(C_i)$. For any given valid schedule S of G consisting of a sequence of $\texttt{cNode}_i^f$ and $\texttt{cNode}_i^b$ denote the forward and backward `cNodes` of the i-th `H-Cluster`, replacing $\texttt{cNode}_i^f$ and $\texttt{cNode}_i^f$ in the schedule by S_i^f and S_i^b makes a valid bottom schedule of the `H-Cluster` of G.*

Proof. 1. Every computational node in G belongs to one of the subgraph C, so that the computation appears in the final schedule at least once. 2. A PyTorch operation o in the final schedule may depend on two types of `aNodes`: (1) the ones appearing in only one C_i, (2) the ones appearing in more than one C_i. For the first type, it is guaranteed by the valid schedule S_i that it is alive; for the second type, it appears explicitly in G so that the original schedule S guarantees that it is alive. Note that no deletion of the second type appears in S_i. □

In addition to the naive approach, we employ various solvers to combine the re-materialization schedules from subgraphs. Each solver, operating under specific constraints and objectives, aims to provide a valid schedule of the given `H-Graph`. The *applicability* of a solver decides whether it can solve a given `H-Graph` in a reasonable time. The solving procedure illustrated in Figure 3.1 works from bottom to top: each `H-Cluster` is solved by applying all the solvers (if applicable) to all the possible `H-Graphs` of the `H-Cluster`. The produced schedules are stored along with the `H-Cluster` and may be selected and used by solvers at higher level.

While all solvers can be used at any level, not all are capable of operating hierarchically. A solver is called *hierarchical* when it can be applied on different levels of `H-Graph` only if it can read and combine **multiple** schedules of every `cNode` in the given `H-Graph`. When a `cNode` represents a big set of operations, the solver is required to consider different schedules to realize the `cNode` in its solution. On the other hand, a **direct solver** allows only a single way to realize a `cNode`. Hence, they can only solve the bottom `H-Graph` where each `cNode` represents a single PyTorch operation.

Lemma 3.2.1 indicates that valid schedule of the overall computational graph can be produced by combining the schedules of subgraphs. Sophisticated solvers can be designed to make the smart combination. Furthermore, since the combination can be conducted recursively, the size of each `H-Graph` can be limited according to the complexity of the solver. We use this framework to balance the efficiency and the performance of the solving algorithm.

Specifically, we propose H-ILP as the main hierarchical solver. H-ILP is an Integer Linear Program (ILP) formulation that provides an optimal schedule within a given memory budget. It can be applied to subgraphs of arbitrary structure, but using it on large subgraphs may lead to unreasonably long solution times. Therefore, it is only applied to subgraphs with a sufficiently small number of nodes, like 10 to 20. This algorithm is inspired by RK-CHECKMATE introduced in Chapter 2 and CHECKMATE [24]. Extending an ILP-based approach to a graph of arbitrary size and structure, which is hierarchically decomposed into subgraphs of manageable size, is a significant contribution of this work and is detailed in Section 3.4.

Additionally, two pre-existing algorithms are adapted to the HIREMATE framework. First, H-TWREMAT is a wrapper around the TW-REMAT implementation [52] of a heuristic based on a **treewidth decomposition** approach [32]. This wrapper enables the heuristic to be used with PyTorch, whereas it was previously available only for TensorFlow. Second, RK-ROTOR is the **dynamic programming** algorithm from Chapter 2, which

provides very effective schedules for (forward) graphs consisting of a sequence of potential complex subgraphs. Although limited in general applicability, it has low solving time.

In the next two sections, we introduce the detailed methods of **H-Partition** and H-ILP.

3.3 H-Partition Algorithm

3.3.1 Sketch of Partitioning Algorithm

In this section, we introduce the algorithm used in HIREMATE to partition a large size graph into smaller subgraphs. Note that the algorithm is applied on the forward graph so that all their corresponding backward operations will automatically be grouped into the same `H-Cluster`. The subgraph sizes are bounded by two main parameters: M^l denotes the maximum number of nodes in a lower-level subgraph, and M^t denotes the maximum number of nodes in the top-level graph. Since it is advantageous to allow a longer solution time for the top-level graph, we use $M^t \geq M^l$. Our partitioning algorithm is a greedy heuristic described in Algorithm 7. Each iteration consists of four main steps: forming *candidate* groups, selecting the best candidate according to our evaluation criterion, merging the selected candidate, and updating the candidates. Each step is described below.

Forming candidate groups For each node x in G, we consider $a(x)$, the closest common ancestor of all direct predecessors of x (a common global ancestor is added in case G has multiple entry points). We create four candidate groups with all nodes on all paths from $a(x)$ to x, depending on whether x and/or $a(x)$ are included. Any candidate group with more than M^l nodes is discarded.

Selecting the best candidate When selecting the best group among all candidates, our goal is to avoid incurring too much memory pressure when the group is used as a subgraph. The memory pressure depends directly on the size of the input and output values. It also depends, although not as directly, on the length of the schedule during which they will be alive, which we evaluate by the number of original nodes in the subgraph. We use the following score function $s(C)$ for a candidate C:

$$s_\alpha(C) = \left(\sum_{\substack{x \text{ input or output} \\ \text{value of } C}} \begin{matrix}\text{memory size} \\ \text{of } x\end{matrix} \right) \cdot \begin{pmatrix}\#\text{ of original} \\ \text{nodes in } C\end{pmatrix}^\alpha, \tag{3.1}$$

where α is a hyperparameter whose default value is 0.5.

Updating the candidates Once the best candidate C has been chosen, it becomes a group: all its nodes are considered together from now on. However, it is not yet a subgraph: if it is not too large, it can be merged with other groups later in the same phase to reduce the number of groups. The remaining candidates are updated: in each candidate C' whose intersection with C is not empty, we add all other vertices of C to

ensure that all vertices of C are kept together. If this union contains more than M^l nodes, this candidate C' is no longer acceptable and is removed.

After partitioning, HIREMATE identifies subgraphs that correspond to the execution of the same piece of code to avoid solving the same optimization problem multiple times.

Algorithm 7: H-Partition algorithm

```
1  Input: data-flow graph G
2  Result: a recursive partition of G
3  Parameters: max high-level size M^t, max lower-level size M^l, score parameter α
4  while G has more than M^t nodes do
5      C ← ⋃_{x∈G} candidate group containing all nodes between x and a(x) ;
6      while C is non empty and G has more than M^t nodes do
7          Select candidate C which minimizes s_α (eq. 3.1);
8          Wrap the nodes of C into a group;
9          Update C;
10     Consider all groups as subgraphs ;
11     Update G so that each subgraph is considered as a node;
12 return partitioned graph
```

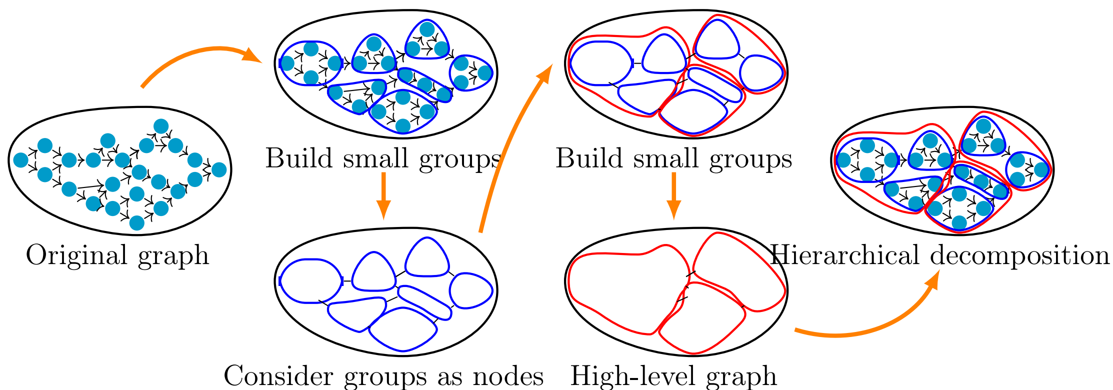


Figure 3.4: Visualization of recursive partitioning of the forward graph with 3 levels of hierarchy

In the rest of this section, we analyze the H-Partition algorithm, whose description is reproduced in Algorithm 7. We prove that the partition computed by this algorithm is always valid, in the sense that the resulting subgraphs do not contain any cycle. When merging a subgraph into a node on the higher level, all edges related to a vertex of the subgraph are attached to the resulting node. To ensure that this does not result in a cycle, we guarantee that all of the subgraphs are *convex* in the graph theoretic sense, as defined in Section 3.3.2.

3.3.2 Convexity

We first provide some graph notations. Given two nodes a and b, we write $a \to b$ if there is a direct edge from a to b, and $a \rightsquigarrow b$ if there is a path of any length from a to b. Paths of length 0 are also valid, so that $a \rightsquigarrow a$ is always true. With these notations, we can define the *convexity* of a subgraph:

Definition 3.3.1. A subgraph C of a graph G is *convex* if for any two elements a, b in C, C contains all nodes of G on any path from a to b. This can be written as:

$$\forall a, b \in C, \forall u \in G, (a \rightsquigarrow u \text{ and } u \to b) \Rightarrow u \in C$$

Merging a convex subgraph C into a node n does not create new cycles into the graph: such a new cycle would be a path starting at n and going back to n, going through another node $u \notin C$. If C is convex, any path from a node of C to another node of C only goes through nodes of C, which ensures the absence of cycles.

3.3.3 Candidate Groups

The candidate groups C_x formed on line 5 of Algorithm 7 contain all nodes on all paths from $a(x)$ to x, where $a(x)$ is the common ancestor to all direct predecessors of x. They have the following property, where $h(C_x) = a(x)$:

Property 3. A subset C of nodes is a *valid* candidate group, if and only if there exists a *head* $h(C)$ such that:

$$\text{If } u \in C \text{ and } v \to u, \qquad \text{then } v \in C \text{ or } v = h(C) \tag{3.2}$$

$$\text{If } u \in C, \qquad \text{then } h(C) \rightsquigarrow u \tag{3.3}$$

This property ensures their convexity:

Lemma 3.3.1. *Any candidate group C which satisfying Property 3 is convex.*

Proof. Consider a and b in C, and u in G such that $a \rightsquigarrow u$ and $u \to b$. There are two cases:

- If $u \neq h(C)$, then since $b \in C$ and $u \to b$, according to (3.2) we have $u \in C$.
- If $u = h(C)$, then since $a \in C$, by (3.3), we have $u \rightsquigarrow a$. Since G is acyclic, this implies $u = a \in C$.

□

3.3.4 Update of Candidates

As discussed in Section 3.3.1, once the best candidate C has been chosen, it becomes a group: all its nodes will be considered together from now on. The remaining candidates are updated: in any candidate C' whose intersection with C is nonempty, we add all the other nodes of C to ensure that all nodes of C remain together. The following results show that the resulting set of nodes is still a valid candidate group; in particular it is also convex.

Lemma 3.3.2. *If C and C' are valid candidate groups with $C \cap C' \neq \emptyset$, then $h(C) \rightsquigarrow h(C')$ or $h(C') \rightsquigarrow h(C)$. Furthermore, if $h(C) \neq h(C')$, then the first case implies $h(C') \in C$ and the second case implies $h(C) \in C'$.*

Proof. Let $u \in C \cap C'$, and consider $v \in G$ such that $v \to u$. If no such v exists, then u is the source of G and $u = h(C) = h(C')$. If $v \in C \cap C'$, we can start over with $u = v$.

We now have $u \in C \cap C'$, and $v \notin C \cap C'$ with $v \to u$. We have three cases:

- If $v \in C$ and $v \notin C'$: from (3.2) applied to C', we have $v = h(C') \in C$, and from (3.3) applied to C we get $h(C) \rightsquigarrow h(C')$.
- Symmetrically, if $v \notin C$ and $v \in C'$, we get $h(C') \rightsquigarrow h(C)$.
- If $v \notin C$ and $v \notin C'$: from (3.2) applied to both C and C', we get $v = h(C) = h(C')$.

□

Theorem 3.3.2. *If C and C' are valid candidate groups with $C \cap C' \neq \emptyset$, then $D = C \cup C'$ is a valid candidate group.*

Proof. From Lemma 3.3.2, we know that $h(C) \rightsquigarrow h(C')$ or $h(C') \rightsquigarrow h(C)$. We define the head of D as $h(D) = h(C)$ in the first case, and $h(D) = h(C')$ otherwise. For simplicity, we assume in the following that $h(C) \rightsquigarrow h(C')$; the other case is symmetrical.

It is clear that D satisfies (3.3): consider any $u \in D$. If $u \in C$, then $h(D) = h(C) \rightsquigarrow u$ by (3.3) applied to C. If $u \in C'$, then $h(D) \rightsquigarrow h(C')$ by assumption and $h(C') \rightsquigarrow u$ by (3.3), so that in both cases $h(D) \rightsquigarrow u$.

We now prove that D satisfies (3.2). Let $u \in D$ and $v \to u$ with $v \notin D$. We distinguish two cases:

- If $u \in C'$, then since $v \notin C'$, by (3.2) applied to C' we get $v = h(C')$; and since $v \notin C$, the contrapositive of Lemma 3.3.2 yields $h(C') = h(C) = h(D)$. Thus $v = h(D)$.
- If $u \in C$, then $v \notin C$ and (3.2) applied to C yields directly $v = h(C) = h(D)$.

□

This completes the validity proof of Algorithm 7: all candidate groups in $\mathcal{C}$ satisfy Property 3 all along the execution of the algorithm, both when they are created (line 5) and when they are updated (line 9). This implies that all subgraphs created line 11 are convex.

3.3.5 Identification of Identical H-Clusters

To further improve the efficiency of HIREMATE, we adapt an idea from ROCKMATE to reuse the solutions on identical problems. In ROCKMATE, we identify the *blocks* that share the same operations of `cNodes`, and reuse the solutions from RK-CHECKMATE for identical *blocks*. In HIREMATE, each `H-Cluster` also contains a topologically sorted list of `cNodes` which follows the order of PyTorch source code. Similarly, we can easily identify two `H-Clusters` to be identical when all the `cNodes` in their list contain the same functions. This does not require to solve the difficult graph isomorphism problem. All the identical `H-Clusters` share the same set of re-materialization solutions from different solvers.

3.3.6 Complexity Analysis of H-Partition

The complexity of HIREMATE depends mostly on the number N of nodes in the graph. Obtaining the dataflow graph with RK-GB has a complexity $O(N)$ and is very fast in practice. With our current implementation, recursively partitioning the graph with H-partition has a complexity of $O(N^2 \log N)$. This step is also fast for graph sizes up to $N = 1000$, but handling very large graphs would require more work on the graph algorithms: recursively partitioning a graph of size $N = 10^5$ takes 2 hours on an Intel Xeon Gold processor. For graphs with larger sizes, further optimization in the algorithm and its realization is required.

3.4 H-ILP Hierarchical Formulation

In this section, we provide details on H-ILP formulation, the hierarchical re-materialization approach based on solving linear programming problem.

3.4.1 Context

The input to H-ILP optimization is an arbitrary graph H, where each `cNode` represents a subgraph. Similar to RK-CHECKMATE from ROCKMATE 2, dependencies are carried by ***aNodes***, that represent *values* that can be saved in memory. A value is said to be *alive* at some time in a schedule if it is stored in memory at that time. The memory usage at a given time in a schedule is the sum of the memory sizes of all values alive at that time, and the *peak memory* of a schedule is the largest memory usage over the length of the schedule. The H-ILP formulation computes the schedule with minimum running time whose peak memory remains below a specified memory budget B. We denote by T the number of `cNodes`, by I the number of `aNodes`. `cNodes` are numbered in a topological order.

3.4.1.1 Compute options and phantom nodes

The novelty of H-ILP compared to RK-CHECKMATE is that each `cNode` can represent a subgraph of the original graph. Such a `cNode` can be computed with one of several options. Each of these options represents a possible schedule for the forward and backward phases of the subgraph.There is a strong link between the forward and the respective backward computations, and each backward computation should be performed with the same option as its corresponding forward computation. As defined in Section 3.2.1, a pair of a forward and the corresponding backward nodes is a `H-Cluster`. For a `H-Cluster` j, its forward and backward `cNodes` are denoted F_j and B_j respectively.

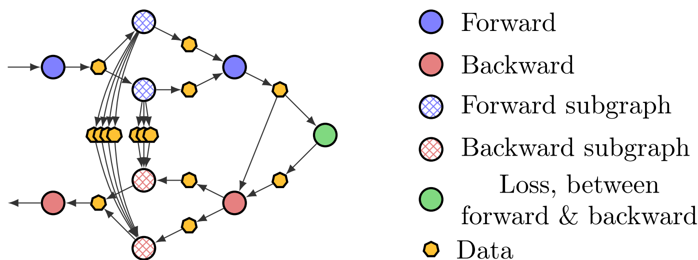


Figure 3.5: HIREMATE recursively finds schedules for each subgraph. Different schedules correspond to different memory budget constraints (options) and hence have different values for memory/time ratio, peak memory, size of the saved data, and time for the backward computation.

In H-ILP, we also introduce an explicit representation of the data saved in memory between a forward computation and its corresponding backward. We call them *phantom nodes*, and we update the formulation by considering them as special `aNodes`, with two specificities:

- a phantom node is always created by its forward computation, can only be deleted by its backward computation, and is not required by any other `cNode`. In the formulation, we can take advantage of this by not including additional variables expressing whether the phantom node is deleted or not.
- values saved in an phantom node (and thus the associated memory size) depend on the option used for the forward and backward computations. For this reason, the formulation contains additional variables that specify which option of each phantom node exists in memory during each stage.

Even though a phantom node represents multiple tensors, it is considered as a single node in the graph. In this formulation, we consider schedules in which for a given phantom node, only one option is present in memory at a given time. We do not consider the phantom node being incrementally generated by several forward options: once a phantom node is generated, it must be used in backward, and either completely deleted or completely saved for the recomputation of backward.

3.4.1.2 Note about input dependencies

3.4.1.2.1 Option-specific dependencies An output value of the forward computation (i.e., an `aNode` which is computed during forward and used by another `cNode`) is never included in the phantom node. However, it happens that an output value is also used within the forward computation to produce other results. An example could be:

```
def compute(a):
  x = f(a)
  y = g(x)
  return x, y
```

In this example, the value `x` is both an output of the `H-Cluster` and used to produce `y`. In that case, the backward schedule might choose either to use `x` as input to be able to perform the backward of `g()` (if having it in memory between forward and backward fits in the budget), or to recompute it during backward. The implication is that for a given `H-Cluster`, each option leads to specific dependencies for the backward `cNode`, depending on which inputs is used by the corresponding schedule. If option o of a computation node k depends on value d, we denote this as $d \xrightarrow{o} k$.

3.4.1.2.2 Multiple predecessors An `aNode` can have several predecessors. This happens in backward when computing gradients: each computation is a *contribution* to the same memory slot (gradients are accumulated). A successor of such an `aNode` can only be processed if all its contributions have been computed.

3.4.2 Formulation

The schedule is divided into T stages. The goal of stage t is to compute `cNode` t for the first time. In the following, we denote `cNodes` with index k, `aNodes` with index d, options with index o, stages with index t and `H-Cluster` (a pair of forward and corresponding backward nodes) with index j.

In contrast to RK-CHECKMATE, phantom nodes have adaptive memory size in the context of H-ILP. With respect to different options chosen for each `H-Cluster`, different number of tensors can be saved for the same phantom node. For any `aNode` d, let s_d be the amount of memory required to store d; and for any `H-Cluster` j and option o, let $p_{j,o}$ be the amount of memory required to store option o of the phantom node of `H-Cluster` j. Phantom nodes are not considered as `aNode` in the formulation, since they are treated with their own variable constraints.

$\mathcal{F}$ is the set of final `aNodes`. The graph contains a specific *loss* node, which represents the computations that take place between the forward and backward passes of our graph. If G is the main highest-level graph, this represents the computations of the loss for the training; if G is any subgraph, loss node also contains other computations from the rest of the graph, as shown in Figure 3.5. The index of the loss node is l.

3.4.2.1 Variables

The H-ILP formulation only contains binary variables, which can take the value 0 or 1. The values of some variables are represented in Figure 3.6.

$Comp^t_{k,o}$ is 1 if and only if node k is computed with option o during stage t.

P^t_d is 1 if and only if `aNode` d is present in memory before stage t.

$S^t_{k,d}$ for k predecessor of d is 1 if and only if the contribution of `cNode` k has been included in `aNode` d before stage t.

$Sp^t_{j,o}$ is 1 if and only if the phantom node of `H-Cluster` j is saved with option o before stage t.

$C^t_{k,d}$ is 1 if and only if `aNode` d is created when computing node k during stage t

$D^t_{k,d}$ is 1 if and only if `aNode` d is deleted after computing node k during stage t

For any `cNode` k and any stage t, we denote by $sumComp^t_k = \sum_o Comp^t_{k,o}$ the equivalent of the $R_{t,k}$ variable of RK-CHECKMATE introduced in Section 2.4, which is equal to 1 if node k is computed during stage t (with any option). Note that the solution of $sumComp^t_k \in \{0,1\}$ since there can only be one way of executing the `cNode` at each step. The time cost of executing `cNode` k with option o is marked as $time_{k,o}$.

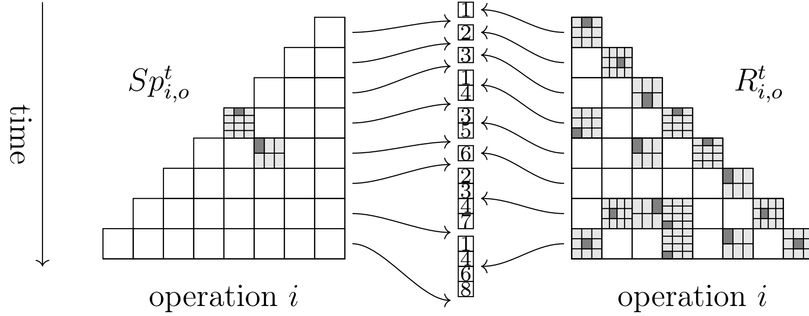


Figure 3.6: An adapted view of H-ILP formula, corresponds to Figure 2.1.

3.4.2.2 Objective

The objective is to minimize the total running time, expressed as

$$\min \sum_{i,t,o} Comp^t_{i,o} * time_{k,o}$$

3.4.3 Validity Constraints of H-ILP

The principles for a schedule to be valid, as introduced in Section 3.2.2, is that (i) every `cNode` is computed at least once and (ii) all computation has valid inputs in memory. In the ILP formulation,

3.4.3.1 Boundary constraints

In this section, we add the boundary conditions for ILP variables. These constraints ensure that the phantoms will be properly generated for the backward.
Node $k > t$ can not be computed in stage t following the topological order:

$$\forall t, \forall k > t, \quad sumComp_k^t = 0. \tag{3.4}$$

No phantom j from node $F_j > t$ is saved before stage t:

$$\forall t, \forall j \text{ s.t. } F_j > t, \forall o, \quad Sp_{j,o}^t = 0. \tag{3.5}$$

No result of `cNode` k can be saved in any stage before stage k:

$$\forall k \to d, \forall t \leq k, \quad S_{k,d}^t = 0. \tag{3.6}$$

`aNode` d is not in memory before any of its predecessors:

$$\forall d, \forall t \leq \min\{k | k \to d\}, \quad P_d^t = 0. \tag{3.7}$$

After the last stage, all *final* `aNodes` should be in memory:

$$\forall d \in \mathcal{F}, \forall k \to d, \quad S_{k,d}^T + sumComp_k^T = 1. \tag{3.8}$$

Node t is executed in stage t:

$$\forall t, \quad sumComp_t^t = 1. \tag{3.9}$$

The *loss* node is executed only once:

$$\sum_t sumComp_l^t = 1. \tag{3.10}$$

3.4.3.2 Data dependencies

We introduce the data dependency constraints in this section, corresponding to the second valid condition mentioned above. Note that each `aNode` may have different contributions and all of them should be alive and kept in memory when the `aNode` is used as an input.

`aNode` d with at least one contribution k is alive:

$$\forall t, \forall k \to d, \quad S_{k,d}^t \leq P_d^t. \tag{3.11}$$

New contribution k to `aNode` d only appears by being computed:

$$\forall t < T, \forall k \to d, \quad S_{k,d}^{t+1} \leq S_{k,d}^t + sumComp_k^t. \tag{3.12}$$

`cNode` k' requires all contributions k to input node d:

$$\forall t, \forall k \to d \to k', \quad sumComp_{k'}^t \leq sumComp_k^t + S_{k,d}^t. \tag{3.13}$$

Option-specific dependencies:

$$\forall t, \forall j, \forall o, \forall k \to d \xrightarrow{o} B_j \quad Comp_{B_j,o}^t \leq sumComp_k^t + S_{k,d}^t. \tag{3.14}$$

3.4.3.3 Options-related valid constraints

The H-ILP formulation contains constraints relative to the choice of options and the management of phantom nodes. Namely:
At most one option for each computation

$$\forall t, \forall k, \quad \sum_o Comp^t_{k,o} \leq 1 \tag{3.15}$$

Only one option of a phantom node is in memory

$$\forall t, \forall j, \quad \sum_o Sp^t_{j,o} \leq 1 \tag{3.16}$$

A phantom node is only created by its F_j

$$\forall t, \forall j, \forall o, \quad Sp^{t+1}_{j,o} \leq Sp^t_{j,o} + Comp^t_{F_j,o} \tag{3.17}$$

A phantom node is only deleted by its B_j

$$\forall t, \forall j, \forall o, \quad Sp^{t+1}_{j,o} \geq Sp^t_{j,o} + Comp^t_{F_j,o} - Comp^t_{B_j,o} \tag{3.18}$$

Computing B_j requires the phantom node

$$\forall t, \forall j, \forall o, \quad Comp^t_{B_j,o} \leq Sp^t_{j,o} + Comp^t_{F_j,o} \tag{3.19}$$

3.4.3.4 Alive status of values

A `cNode` k is *related* to an `aNode` d if $k \to d$ or $d \to k$. We denote this with $k \leftrightarrow d$. For an `aNode` d, only `cNodes` k that are related to d can affect its alive status. For any t, if k is related to d, we denote with $A^t_{k,d}$ the alive status of node d during stage t after `cNode` k (and also after performing all deletions mandated by variables D). In stage t after `cNode` k, an `aNode` d is alive if it was stored before stage t or created in stage t before node k, and not deleted until then, so that we can write:

$$\forall t, \forall k \leftrightarrow d, \quad A^t_{k,d} \doteq P^t_d + \sum_{k' \to d, k' \leq k} C^t_{k',d} - \sum_{k' \leftrightarrow d, k' \leq k} D^t_{k',d} \tag{3.20}$$

In the above equation and in the following, we use $\doteq$ to denote an alias definition, so that $A^t_{k,d}$ can be replaced by the right-hand side in any constraint, whereas the $=$ sign is used to denote a constraint that is added to the formulation.

3.4.3.5 Constraints related to liveness

As in RK-CHECKMATE in Chapter 2, liveness of an `aNode` and its contributions from source `cNodes` are considered separately in the ILP formulation. Here, we introduce the liveness constraints, preparing for the memory usage representation in the next section.

`aNode` d is either alive or not:

$$\forall t, \forall k \leftrightarrow d, \quad 0 \leq A^t_{k,d} \leq 1 \tag{3.21}$$

d is alive if computed and not deleted

$$\forall t, \forall k \to d, \quad A^t_{k,d} \geq sumComp^t_k - D^t_{k,d} \tag{3.22}$$

Value d can only be created by a node k that is really computed

$$\forall t, \forall k \to d, \quad C^t_{k,d} \leq sumComp^t_k \tag{3.23}$$

Value d is alive after stage t if and only if it is alive after its last related node k

$$\forall t < T, \forall d, \quad P^{t+1}_d = A^t_{\max\{k|k \leftrightarrow d\},d} \tag{3.24}$$

One additional constraint states that a value d is deleted after `cNode` k in stage t if it is not used afterwards: neither by later `cNodes` $k' > k$ in the same stage t, nor in the next stage $t+1$. This can be stated as:

$$\begin{aligned} \forall t, \forall k \leftrightarrow d, \quad & D^t_{k,d} = 1 \\ & \text{if and only if } sumComp^t_k = 1 \\ & \quad \text{and } P^{t+1}_k = 0 \\ & \quad \text{and } \sum_{d \to k', k' > k} sumComp^t_{k'} = 0 \end{aligned}$$

However, this constraint is not linear. It can be linearized in the similar way as in the original CHECKMATE paper [24]: if we denote by $h_{k,d} = 2 + |\{k' | d \to k', k' > k\}|$ the number of equalities in the above statement, it is equivalent to:

$$\forall t, \forall k \leftrightarrow d, \quad D^t_{k,d} \geq sumComp^t_k - P^{t+1}_k - \sum_{d \to k', k' > k} sumComp^t_{k'} \tag{3.25}$$

$$\forall t, \forall k \leftrightarrow d, \quad h_{k,d}(1 - D^t_{k,d}) \geq 1 - sumComp^t_k + P^{t+1}_k + \sum_{d \to k', k' > k} sumComp^t_{k'} \tag{3.26}$$

3.4.4 Memory Usage Constraints

We denote by U^t_k the memory usage after computing node k in stage t, similar to RK-CHECKMATE.

Then, U^t_k can be expressed as a linear combination of the formulation variables. Indeed, the memory usage before starting stage t is

$$M_t \doteq \sum_d s_d \cdot P^t_d + \sum_{j,o} p_{j,o} \cdot Sp^t_{j,o}. \tag{3.27}$$

Then, the increment when computing a forward node $k = F_j$ during stage t is

$$IF_k \doteq \sum_{k \to d} s_d \cdot C^t_{k,d} - \sum_{k \leftrightarrow d} s_d \cdot D^t_{k,d} + \sum_o p_{j,o} Comp^t_{k,o}. \tag{3.28}$$

When computing a backward node $k = B_j$ during stage t, the increment is

$$IB_k \doteq \sum_{k \to d} s_d \cdot C^t_{k,d} - \sum_{k \leftrightarrow d} s_d \cdot D^t_{k,d} - \sum_o p_{j,o} \left(Comp^t_{F_j,o} + Sp^t_{j,o} - Sp^{t+1}_{j,o} \right) \tag{3.29}$$

The expression within the parenthesis is equal to 1 only if the allocated node j is deleted, and 0 otherwise. Indeed, since B_j is the only `cNode` that can use it, $Sp^{t+1}_{j,o} = 0$ means that phantom node j can be deleted right after B_j. Constraint (3.18) ensures that if $Comp^t_{B_j,o} = 0$, then the expression within the parenthesis is also 0.

Finally, we can express U^t_k iteratively (similar to CHECKMATE and RK-CHECKMATE formulations):

$$\forall t, \quad U^t_0 \doteq M_t + IF_0 \tag{3.30}$$

$$\forall t, \forall k = F_j, \quad U^t_k \doteq U^t_{k-1} + IF_k \tag{3.31}$$

$$\forall t, \forall k = B_j, \quad U^t_k \doteq U^t_{k-1} + IB_k \tag{3.32}$$

Thanks to the U^t_k definitions, we can express constraints to ensure that the memory usage is always within the memory budget B. If we detail a single step k of some stage t, it corresponds to (a) allocating memory for the newly created values (according to $C^t_{k,d}$ variables), (b) computing `cNode` k, (c) freeing the memory of the deleted values (according to variables $D^t_{k,d}$). The variable U^t_k corresponds to the saved memory after (c), while the real peak memory usage should be found during (b). In addition, the computation of node k with some option o might incur a memory overhead (by allocating temporary values), which we denote by $m_{k,o}$. In total, in RK-CHECKMATE, the memory budget constraints are written as:

$$\forall t, \forall k, \quad U^t_k + \sum_o m_{k,o} \cdot Comp^t_{k,o} + \sum_{k \leftrightarrow d} s_d \cdot D^t_{k,d} \leq B. \tag{3.33}$$

3.4.4.1 Correction Terms

The formulation described so far expresses correctly the memory usage *between* `cNodes`. This is sufficient in the context of RK-CHECKMATE, where the `cNodes` represent basic operations. For H-ILP however, each node represents a complete *sequence* of basic operations, whose peak memory can not be estimated in isolation: it depends on whether the input values are needed after its execution. After the H-ILP formulation is solved, the actual schedule is modified: the memory deallocations for the values freed in step (c) above are performed as early as possible, possibly during the schedule of node k (in the middle of step (b)).

This means that the memory overhead $m_{k,o}$ during the computation of option o of node k might depend on whether some values are alive before or after computing `cNode` k. For

example, if the corresponding schedule deletes a value in the middle of computation, its memory overhead $m_{k,o}$ assumes that the deletion is delayed until the end of the schedule. If that value is actually not needed later in the higher-level schedule computed by H-ILP, it will be deleted within the schedule, what may or may not change the memory overhead.

In the following, we present how to modify the memory budget constraint to account for this kind of situation. Consider a specific stage t, and an option o (and thus a schedule) for node k. For simplicity of presentation, let us consider only inputs; the situation with outputs is similar and symmetric. Consider a substep i of the schedule. We compute the memory overhead at this substep as $m^i_{k,o}$, assuming that value deletion happens after the computation of this schedule of node k. We denote by $\mathcal{F}_i$ the set of values which are not used in the following substeps of that schedule. Within the schedule computed by H-ILP, values in $\mathcal{F}_i$ are deleted after substep i if and only if they are deleted after step k. Hence, the *actual* memory usage of substep i is $m^i_{k,o} - \sum_{d\in\mathcal{F}_i} s_d \cdot D^t_{k,d}$. The corresponding memory constraint is given by

$$\forall t, \forall k, \forall o, \forall i, \quad U^t_k + m^i_{k,o} \cdot Comp^t_{k,o} + \sum_{k\leftrightarrow d} s_d \cdot D^t_{k,d} - \sum_{d\in\mathcal{F}_i} s_d \cdot D^t_{k,d} \leq B \tag{3.34}$$

We write one constraint for each option and each substep. Since all the correction terms are negative, and all $m_{k,o}$ are at least 0, if $Comp^t_{k,o}$ is 0, this constraint is weaker than (3.33). We write such a constraint for each substep of the schedule, and this provides a more precise assessment of the memory usage of the solution. The case of output values is the same, except that we care whether the output value is created during the computation of node k, which is represented with variable $C^t_{k,d}$.

An interesting remark is that it is not necessary to write one constraint for each substep: if the set of inputs not needed after substeps i and j are the same ($\mathcal{F}_i = \mathcal{F}_j$), we can keep only one of both constraints (the one with the larger memory usage $m^i_{k,o}$). The number of constraints is thus bounded by $\min(\text{number of substeps}, 2^{|\{\text{inputs}\}|+|\{\text{outputs}\}|})$. In practice, the number of different constraints remain low enough. In addition, these constraints are only introduced when solving the top-level graph, where the constraint to remain under budget B is required to be as accurate as possible.

3.4.5 Option Selection

The number of binary variables in the H-ILP formulation scales linearly with the total number of options across all nodes. To control the size of variables and make the hierarchical solving more efficient, we include in H-ILP a hyperparameter N_o that imposes a limit on the total number of options in the input `H-Graph`. To stay within this limit, we may need to select only a few options from all the schedules generated at the lower level.

To do so, we first decide how many options to assign to each `cNode` based on the total number of N_o. The number assigned to `cNode` k is: $N_k = min(1, N_o * \frac{\#\text{ options in }\texttt{cNode k}}{\#\text{ options in }\texttt{H-Graph}})$. We assume that `cNode` with more operations have more complicated structure, thus it is more meaningful to assign more options to them.

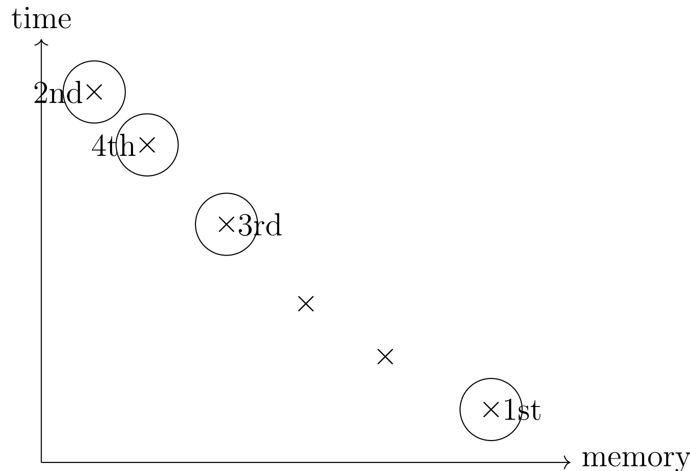


Figure 3.7: Schedules selection for a `H-Cluster`. Each × represents a schedule with certain time and memory cost, and the circled one are selected in the given order.

To avoid wasting resources when several very similar options are available for a `cNode`, we then greedily select the schedules whose memory peaks are farther apart from each other: starting with the schedule with the highest memory peak, then the one with the lowest memory peak, then the one closest to the middle, and so on. An example of schedule selection is provided in Figure 3.7.

3.4.6 Complexity Analysis of H-ILP

Solving H-ILP is the most time-consuming step of the HIREMATE framework. Thanks to our hierarchical approach, this step actually has linear complexity. In fact, the hierarchical decomposition has a logarithmic depth, and the total number of subgraphs is $O(N/M^t)$, where N is the total number of nodes and M^t is the size of each subgraph. Solving a subgraph of size M^l for a number O of budget options only requires solving O Integer Linear Programs, whose sizes and solving times do not depend on N. Moreover, all ILPs at the same level are completely independent and can be run in parallel. We do not have the implementation of running multiple ILP solving processes on the same machine but this can be feasible in practice.

3.5 Experimental Evaluation

In this section, we present empirical results to demonstrate the capability of HIREMATE on different models and machines.

Two types of machines are used in our experiments: an NVIDIA Tesla V100 16GB GPU with a 32-core Intel Skylake CPU, or an NVIDIA Tesla P100 16GB GPU with 32-core Intel Broadwell CPU. Since HIREMATE is improved based on ROCKMATE, which can efficiently solve different architectures of AI models, we provide experimental results

on ILP solving time comparison which is affected by the CPU type. We use the open source PuLP library to build the ILP models, with the commercial solver GUROBI to solve them up to optimal states. We do not run experiments on non-optimal solutions since it highly depends on the random initial states.

For each experiment, we report the average iteration time of 15 iterations with standard deviations. The measured memory cost includes only the activations. In practice, we measure the size of the model parameters and their gradients, and remove them from the real peak memory measured by `torch.cuda.max_memory_allocted()`. Since HIREMATE is a framework that includes different algorithms, we use the specific names for algorithms and their combinations in the experiments, such as H-ILP or ROCKMATE. When H-ILP is applied to a subgraph, the budgets are automatically computed according the mechanism introduced in Section 2.6.0.0.1.

3.5.1 Visualization of Partition

In this section, we demonstrate how a Transformer model is partitioned in HIREMATE. Unlike, GPTs, a Transformer model is not typically considered as sequential. The architecture of Transformer is shown in Figure 3.8. For a n-layer model, there are n encoder layers and n decoder layers. It cannot be partitioned into $2n$ *blocks* since there are dependencies between the encoders and decoders.

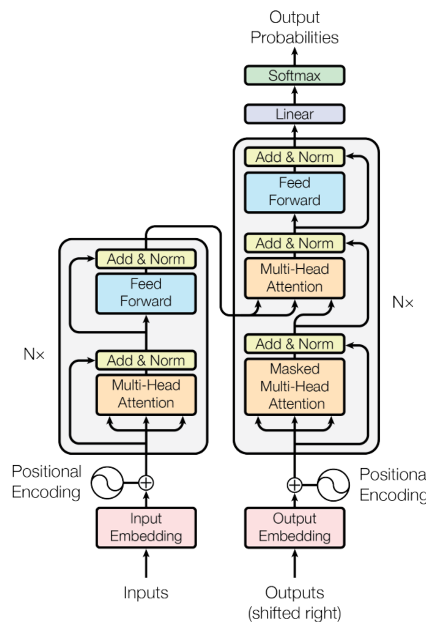


Figure 3.8: Demonstration of Encoder-Decoder Transformer from [58]. This graph shows only the forward pass of the neural network.

If we try to follow the partition in ROCKMATE which separates different parts of the model sequentially, the result is shown in Figure 3.9. When n is large, there will be a *block* containing hundreds of nodes, which makes it impossible to apply RK-CHECKMATE on it.

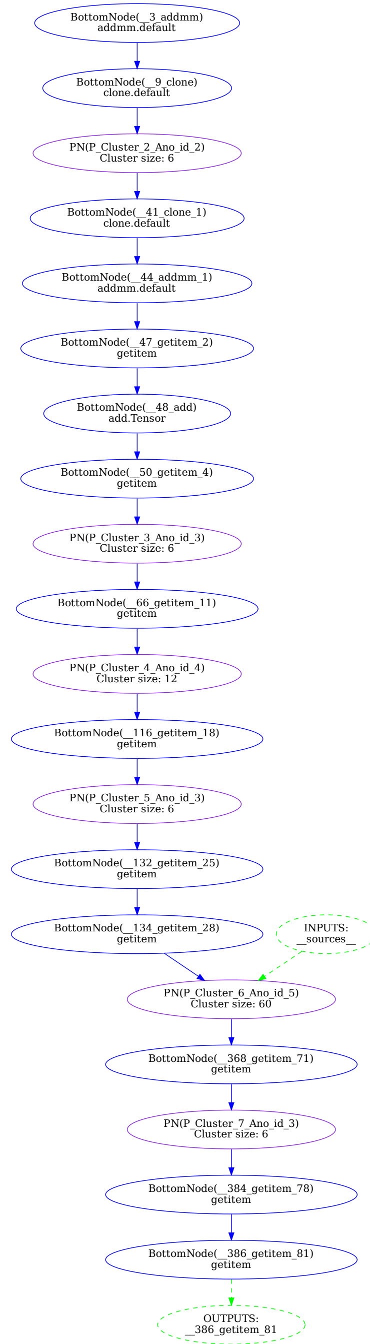


Figure 3.9: Visualization of 2-layer Transformer partitioned sequentially, using the rule introduced in ROCKMATE. Note that two decoder layers are wrapped into one (Cluster_6_Ano_id5) and its dependencies from other encoder layers are not shown in the graph. Due to the limit of space, we do not present the dependencies between those 60 nodes.

In HIREMATE, we partition the graph recursively so that the every graph in the final structure has limited number of nodes. An example of partitioning a 6-layer Transformer is shown in Figure 3.10 and 3.11. As introduced in Section 3.3, the partitioning proceeds from bottom to top. Whenever the graph is too large, some nodes will be chosen to group as one which creates a subgraph. In our example, the encoder side of the model is grouped as one subgraph, which is presented in the right plot of Figure 3.10. This subgraph contains several `H-Clusters`, one is particularly large which contains 139 nodes. This `H-Cluster` is shown in Figure 3.11. After the partitioning, every graph in the structure has a limited size thus can be efficiently solved by H-ILP.

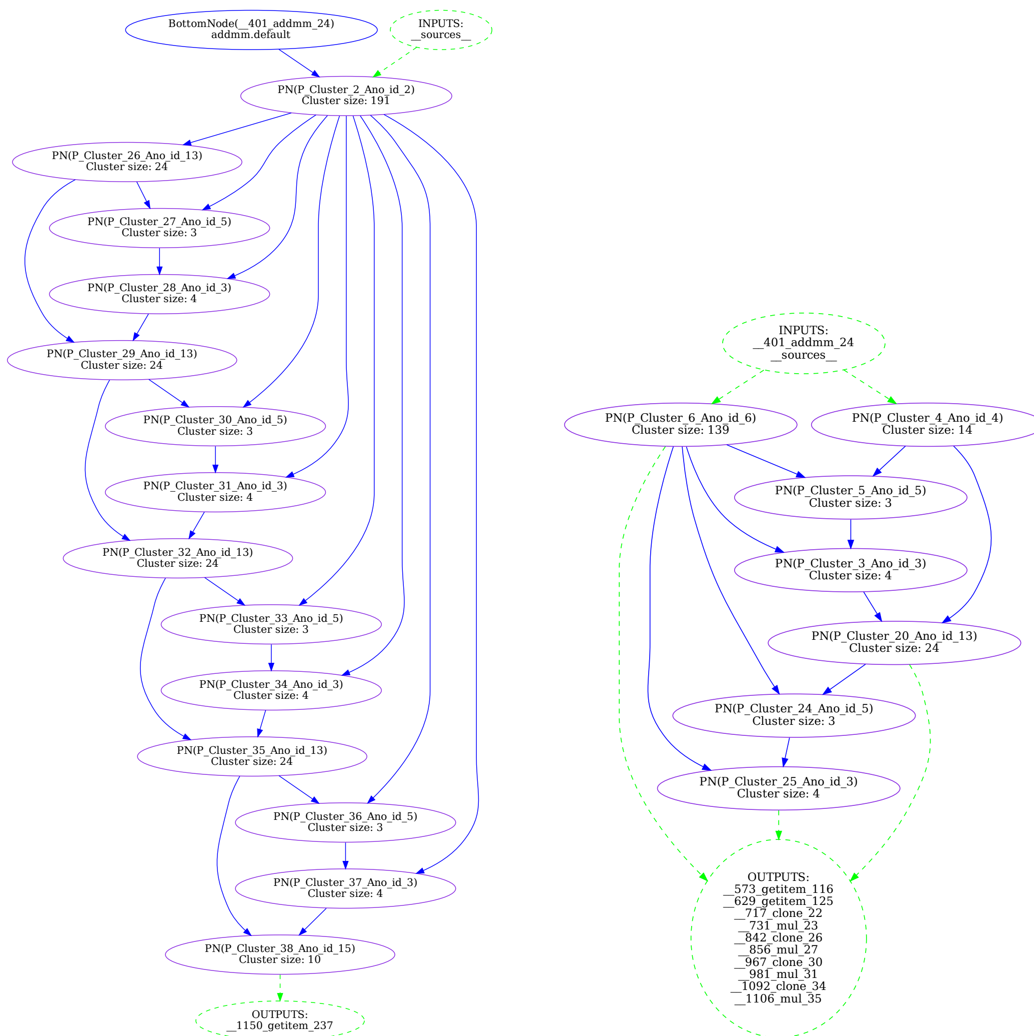


Figure 3.10: Left: visualization of the top level graph for a 6-layer Encoder-Decoder Transformer. Right: visualization of the Cluster_2_Ano_id2 in the left figure.

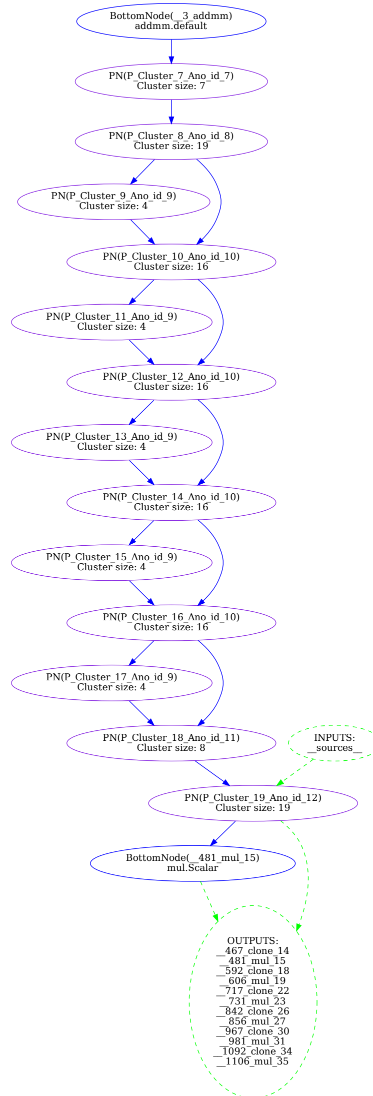


Figure 3.11: Visualization of the Cluster_6_Ano_id6 in Figurse 3.10.

3.5.2 Efficiency

In Figure 3.12, we show the solving time of H-ILP on Transformers. Note that H-ILP can solve 24-layer Transformer within 2 hours. Note that HIREMATE is run once before the entire model training, so 12 minutes is clearly acceptable. By controlling the total number of nodes in each graph with the H-partition algorithm, H-ILP is able to efficiently generate the solution for complex model.

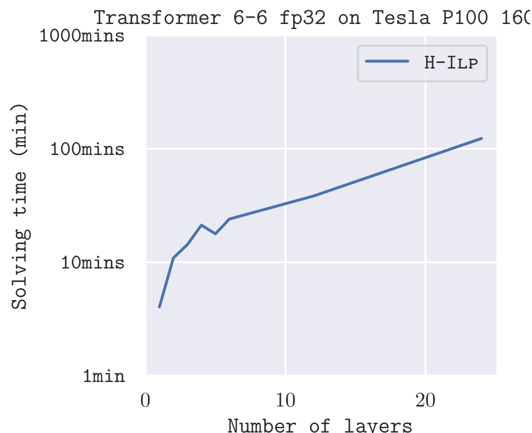


Figure 3.12: Solving time of H-ILP on different layers of Transformers. For the model with 6 layers, 6 encoder layers and 6 decoder layers are involved. The reported preparing time includes partitioning and solving of H-ILP on all the subgraphs.

3.5.3 Performance

In this section, we compare the performance of H-ILP with CHECKMATE and ROCKMATE. We only use the models that can be efficiently solved by those algorithms. Figure 3.13 shows the performance of H-ILP on sequential networks. Those networks can be efficiently solved by ROCKMATE, which makes it a good benchmark for the performance of H-ILP. Another baseline of the re-materialization performance is obtained from ROTOR. Overall, H-ILP finds good quality solutions on different sequential models.

Overall, HIREMATE acts as a general solution that includes H-ILP, ROTOR, ROCKMATE, and CHECKMATE. Our experimental results consistently show that HIREMATE performs on par with these baseline algorithms, demonstrating its versatility and effectiveness in optimizing memory utilization for a wide range of network architectures.

3.5.4 Ablation Study

Without graph partitioning, H-ILP is equivalent to RK-CHECKMATE, which provides an optimal solution for a given topological order of operations. In another experiment, we control the depth of the hierarchy in the H-partition by limiting the size of each subgraph. The results in Figure 3.14 show that H-ILP maintains similar results in terms of solution quality as the depth of the hierarchy increases. This is a critical issue in practice for scalability. Indeed, when the number of nodes in the graph becomes very large, the solution is to increase the depth of hierarchical decomposition in order to maintain reasonable graph sizes at each level (and thus reasonable execution times). Figure 3.14 demonstrates that this increase in depth, even for a fixed model size, does not degrade

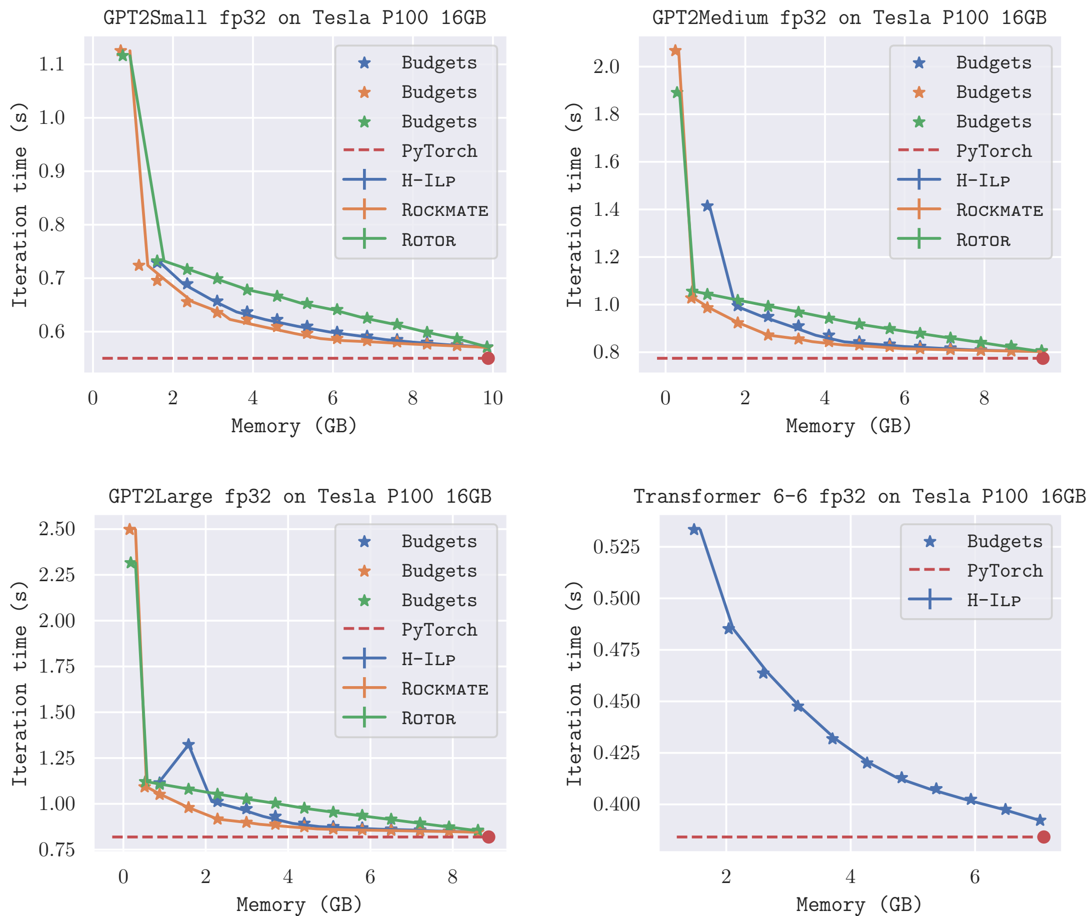


Figure 3.13: Experiments on sequential networks: ROCKMATE are tailored for those models, as discussed in Chapter 2.

performance significantly.

Specifically, we found that the performance improvement gained by increasing the subgraph size is less obvious when the budgets are small. For Transformer, we observe that the H-ILP solution with 20 nodes limitation may even perform worse than the ones with less nodes. A possible explanation is that we control the same budget selection mechanism for all the experiments, which is equal distributed intervals between the minimum and maximum feasible budgets. The minimum budget is related to the usage of the maximum memory usage of single `cNode`, while the maximum budget depends on summation of them. Therefore, when the same number of intervals are applied, the graphs with larger size will have larger interval width which makes less options towards the lower memory side. It also suggests that a better budget selection mechanism may improve the performance of H-ILP.

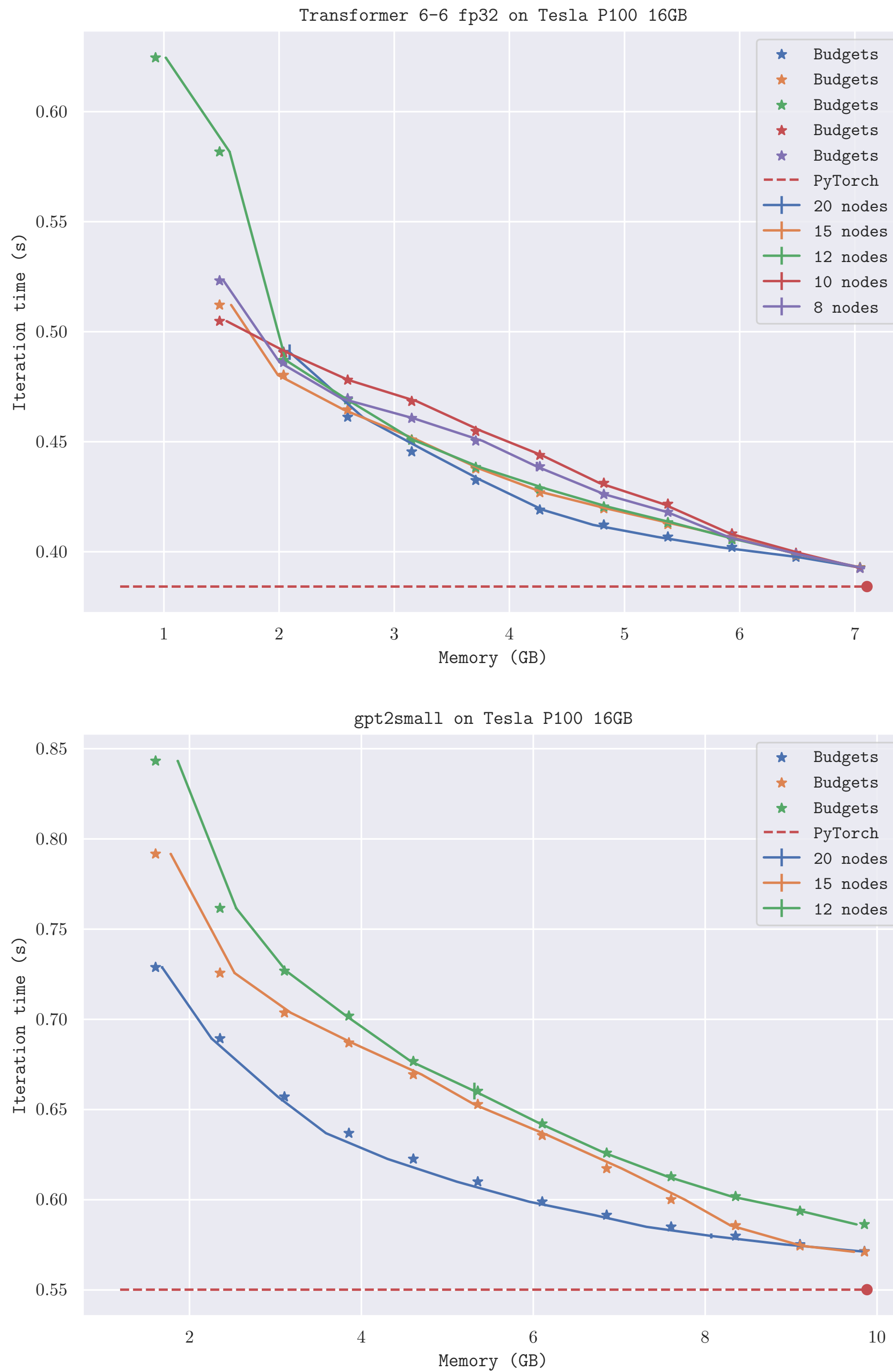


Figure 3.14: Experiments on 6-layer encoder-decoder Transformer and GPT2-small with different nodes limitations of subgraphs in the hierarchical structure.

3.6 Discussion and Conclusion

This chapter introduces the HIREMATE framework, which offers both theoretical and practical advances in re-materialization for PyTorch models. This work has been summarized in the paper HIREMATE [18]. HIREMATE provides a solution that can find very effective solutions in terms of overhead during training, comparable to those of the literature for problems of small size or with a particular graph structure, but without these limitations. On the practical side, HIREMATE integrates seamlessly with PyTorch, improving the memory-time tradeoff and efficiency, and is fully compatible with PyTorch Autograd. The theoretical contributions include a hierarchical approach and an original linear programming formulation H-ILP. Although HIREMATE focuses on computational graphs with primitive operations, the hierarchical approach with linear complexity is potentially applicable to other intensive tasks targeting graphs resulting from tiling. The framework also incorporates previous approaches, ensuring state-of-the-art results. The only remaining limitation of HIREMATE is that it is not adapted to dynamic neural network architectures, where the structure of the computational graph depends on the input. An important feature of HIREMATE is that the code is fully modular, and it is possible for external contributors to introduce a new graph partitioner or solver at any level of the hierarchical decomposition. We hope that this modularity will stimulate research and further improvement of HIREMATE. Future research may include exploring how to integrate offloading into the optimization problem to reduce the need for recomputation, and how to optimally combine re-materialization with pipelined model parallelism. Advances in these areas hold promise for improving the performance and efficiency of deep learning systems.

Chapter 4

OFFMATE

4.1 Introduction

In the previous chapters, we have introduced the methods to reduce the activation memory cost of DNNs training. In this chapter, we introduce OFFMATE, which reduces the overall memory cost of training a large model, including the cost related to model parameters. The motivation of targeting at memory cost of parameters comes from the development of Large Language Models (LLMs). Inspired by the transformer structure [58], various LLMs such as GPT [44], Bloom [60], Llama [56] and Mistral [27] have been developed and demonstrate excellent performance on general tasks. These models are typically trained on multiple GPU devices [48, 53], and the memory footprint is distributed across the GPUs. Based on the large pre-trained models, which require a significant amount of resources to train, fine-tuned models have been proposed and contribute to a variety of fields such as law [22], medicine [61], and finance [35]. Compared to training an LLM from scratch, fine-tuning is more accessible to broader communities of artificial intelligence (AI) enthusiasts because it requires less data, less computing power, and less training time.

However, a common challenge for fine-tuning LLMs is the memory bottleneck of the training process. Most billion-parameter models can hardly be stored on consumer graphics cards, which typically have between 8GB and 24GB of video RAM (VRAM). Depending on the choice of hyperparameters and optimization settings, the training process can require far more memory than the VRAM available on a single GPU. While most LLMs are pre-trained using parallel approaches across hundreds of GPUs, many works like QLoRA or ZeRO [14, 49] have been proposed to allow fine-tuning of LLMs on a single GPU. Most methods reduce memory requirements by simplifying the training process, such as lowering the data precision or reducing the number of trainable parameters. These strategies have proven to be efficient without significantly degrading the accuracy of the resulting model.

In this chapter, we focus on **fine-tuning on a single consumer-grade GPU** without degrading the performance of the model at all. We introduce OFFMATE to reduce memory requirements for LLMs fine-tuning with a very low overhead in training time. OFFMATE efficiently reduces the memory footprint of both activation and weights

during training iterations by selectively combining recomputation of some operations and offloading data and computations to CPU. This combination makes it possible to train an entire large model within the memory footprint of a single transformer block. Without changing the data precision or the optimization settings, OFFMATE enables fine-tuning of billion-size models on a consumer-grade GPU such as RTX 3060 12GB. Unlike Parameter Efficient Fine-Tuning (PEFT) methods that reduce memory requirements by simplifying the training task, OFFMATE preserves the exact numerical results of the training. This makes it compatible with other memory-saving techniques including PEFT.

This chapter presents the following contributions:

- an efficient, fully asynchronous PyTorch framework for full-duplex communication between GPU and RAM, overlapped with independent computations on both GPU and CPU;
- an optimization algorithm based on an Integer Linear Programming formulation, which optimizes over all techniques for reducing memory requirements;
- the OFFMATE tool, which takes any PyTorch model (including from HuggingFace) and with a one-line instruction seamlessly modifies it to fit into memory without approximation;
- an extensive experimental study highlighting the low overhead of our approach compared to the state-of-the-art solutions.

4.2 Motivation

4.2.1 Memory Requirement

The memory requirement in training AI models is presented in Section 1.1.3. We show the definition and example sizes of each part of memory requirements again, as the work of OFFMATE aims to tackle the usage of all different components of memory footprint.

The memory footprint when training a large model consists of several parts:

- intermediate activations with size M_{act}. M_{act} depends on the input batch size, which can be rebuilt using re-materialization to reduce the associated peak memory usage;
- model parameters with size M_{param};
- parameter gradients with size M_{p_grad}. M_{p_grad} is equal to the size of all trainable parameters;
- optimizer states with size M_{opt_st}. M_{opt_st} depends on the optimizer chosen for the task.

Once the model is selected, the size of M_{param} depends only on the data type of the parameters. Both M_{p_grad} and M_{opt_st} are proportional to the number of trainable parameters. If methods like weight-freezing or PEFT are applied, M_{p_grad} can be significantly smaller than M_{param}. M_{opt_st} also depends on the optimizer. Specifically, Adam optimizer [28] and its derived optimizers family store momentum and variance for each parameter gradient, so that $M_{opt_st} = 2 \times M_{p_grad}$ in the case when the same data type is used.

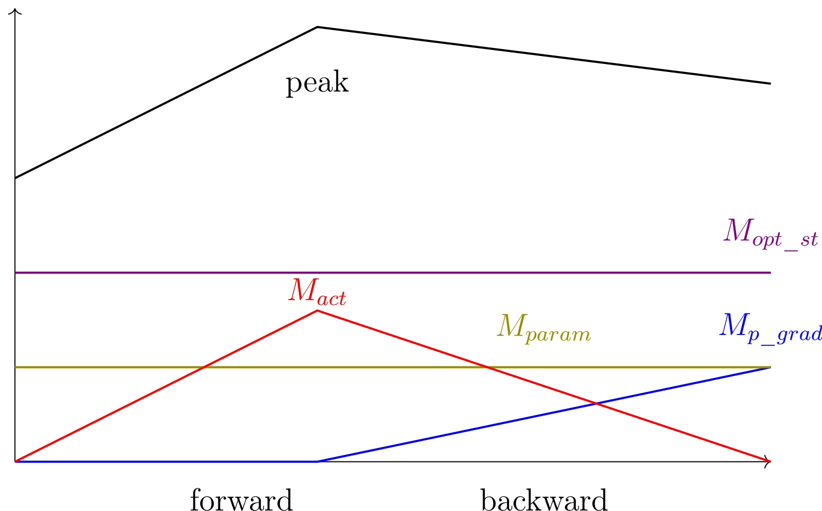


Figure 4.1: Schematic representation of memory requirement of different parts. Black line shows the summation of all part.

If parameter gradients are accumulated through multiple iterations (usually called gradient accumulation), they should be kept in memory during the whole training iteration. In this case, the memory peak is $M_{act} + M_{param} + M_{p_grad} + M_{opt_st}$. On the other hand, it is also common to update parameters and release their gradients after every iteration. In this case, parameter gradients size increases during backward phase while intermediate activation size decreases. As depicted in a schematic manner in Figure 4.1, the peak memory is then smaller than $M_{act} + M_{param} + M_{p_grad} + M_{opt_st}$.

For example, the relative sizes of each part when training a full-precision Llama2-7B model with batch size 2 and sequence length 512 are $M_{param} = M_{p_grad} =$ 25GB and $M_{act} =$ 24GB. If the model is optimized with the Adam optimizer, $M_{opt_st} =$ 50GB, so that the total memory requirement is 124GB. If gradient accumulation is not used, parameter gradients can be removed from memory at each iteration, and M_{p_grad} and M_{act} are not always required simultaneously. Still, the peak memory usage for such a training task is expected to be around 100GB, which is not feasible on any consumer GPU. OFFMATE efficiently reduces the memory usage of all the parts and manages to train this model on a machine with 12GB GPU VRAM and 128GB CPU RAM.

4.2.2 Memory Efficient Solutions

Multiple approaches have been proposed to reduce the memory footprint in neural network training. Parameter Efficient Fine-Tuning (PEFT) methods have been proven effective in reducing the memory requirements of LLM tuning. By efficiently simplifying the training task, many algorithms have demonstrated that it is possible to achieve comparable performance as full model fine-tuning while using substantially lower resource. In particular, the LoRA family [20, 14] has shown that using a small fraction of the training parameters can achieve good performance on various tasks. Quantization has also been

found useful to significantly reduce memory requirements in LLM training [13, 36]. Other methods include training only the input embedding layer [1], training hidden states [38], and training with a sparse mask over the weights [55].

However, the PEFT methods may suffer from lower performance, requiring extensive experiments across various models and fine-tuning tasks to justify their applications. On the other hand, there are other memory-efficient methods which do not compromise the training results. The main approaches are outlined as follows:

Re-materialization Inspired by gradient checkpointing, other re-materialization strategies [25, 6, 7, 2] have been proposed to build efficient schedules for recomputation by solving optimization problems. The previous chapters in our thesis are also providing solutions in re-materialization. Those methods can significantly reduce the memory usage of intermediate activations (M_{act}) at the cost of recomputing time.

Activation offloading Approaches that offload activations from GPU to CPU memory have also been found to be effective in both training [49, 48, 5] and inference [51]. A hybrid combination of re-materialization and offloading has been proposed in [6, 45] to reduce the memory usage of activations.

Parameter offloading ZeRO-Offload [49] reduces the GPU memory usage of parameters (M_{param}) by offloading them to CPU RAM. Furthermore, ZeRO-Infinity [48] extends the approach by offloading parameters to both CPU RAM and NVMe disk during training. This method makes it possible to handle unprecedented large models.

CPU optimization ZeRO-Offload [49] reduces the memory usage of optimizer states (M_{opt_st}) by storing full-precision optimizer states in CPU RAM and performing Adam optimization steps directly on the CPU. The optimizer states are thus generated and updated only on CPU, saving the bandwidth to move them between CPU and GPU. This method reduces the memory footprints on GPU without increasing communication overhead, but the CPU optimization speed may become a bottleneck to limit the training efficiency.

Optimizer states offloading In QLoRA [14], the authors proposed Paged Optimizer which offloads the optimizer states generated on GPU to CPU RAM. This reduces the memory usage of optimizer states (M_{opt_st}) by leveraging the bandwidth between CPU RAM and GPU VRAM.

A comparison of different approaches is presented in Table 4.1. Our work, OFFMATE, integrates all the approaches mentioned above, providing a comprehensive solution to reduce memory usage during training. The framework and implementation details of OFFMATE are explained in the following sections.

Methods	M_{act}	M_{param}	M_{p_grad}	M_{opt_st}
ROCKMATE	$\frac{1}{n}\times$	$1\times$	$1\times$	$1\times$
ZeRO-Offload	$\frac{1}{n}\times$	$1\times$	$\frac{1}{n}\times$	$\frac{1}{n}\times$
PEFT	$1\times$	$1\times$	$a\times$	$a\times$
OFFMATE	$\frac{1}{n}\times$	$\frac{1}{n}\times$	$\frac{1}{n}\times$	$\frac{1}{n}\times$

Table 4.1: Theoretical reduction of memory usage by different methods. Assuming the same precision is used in all methods. n is the number of layers to apply those methods. a is the fraction of trainable parameters in PEFT.

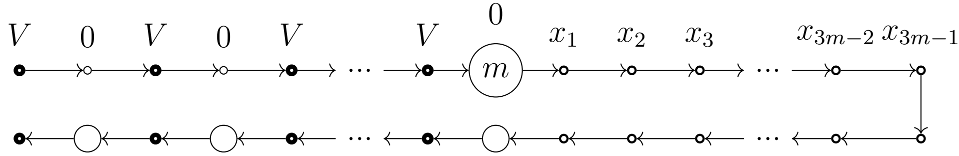


Figure 4.2: Instance used in the NP-completeness proof of Theorem 4.2.1. Node size represents execution times: ∘ for $u_i = 0$, ◯ for $u_i = 1$; the size of the parameters is represented by node thickness and written above.

4.2.3 Complexity

Finding optimal strategies for the memory-efficient solutions listed in Section 4.2.2 is inherently complex. Indeed, it has been proven in Rotor [3] that the re-materialization problem for a sequential neural network is NP-Hard. Similarly, it has been proven in POFO [5] that the activation offloading is strongly NP-Complete. In this section, we analyze the complexity of parameter offloading problem without re-materialization or CPU optimization. We show that the parameter offloading problem itself is NP-Complete in a strong sense so that introducing simplifying assumptions is necessary to OFFMATE.

The decision problem associated to the parameter offloading optimization problem is the following:

Problem 1 ($OFF(L, M_{GPU}, \beta)$)**.** Consider the training phase of an L-layer network. Let the operation time of the forward and backward phase of the i-th layer be denoted by u_{F_i} and u_{B_i}. Given a GPU with memory M_{GPU} and bidirectional bandwidth β between the GPUs and the main memory, is there a valid schedule of offloading, deleting, and prefetching operations such that the training cycle can be completed with execution time at most T?

Theorem 4.2.1. *$OFF(L, M_{GPU}, \beta, 1)$ is NP-complete in the strong sense.*

Proof. Obviously, $OFF(L, M_{GPU}, \beta, 1)$ is in NP: given a schedule of all operations and communications, one can check in polynomial time that the execution is valid and fits within the execution time T.

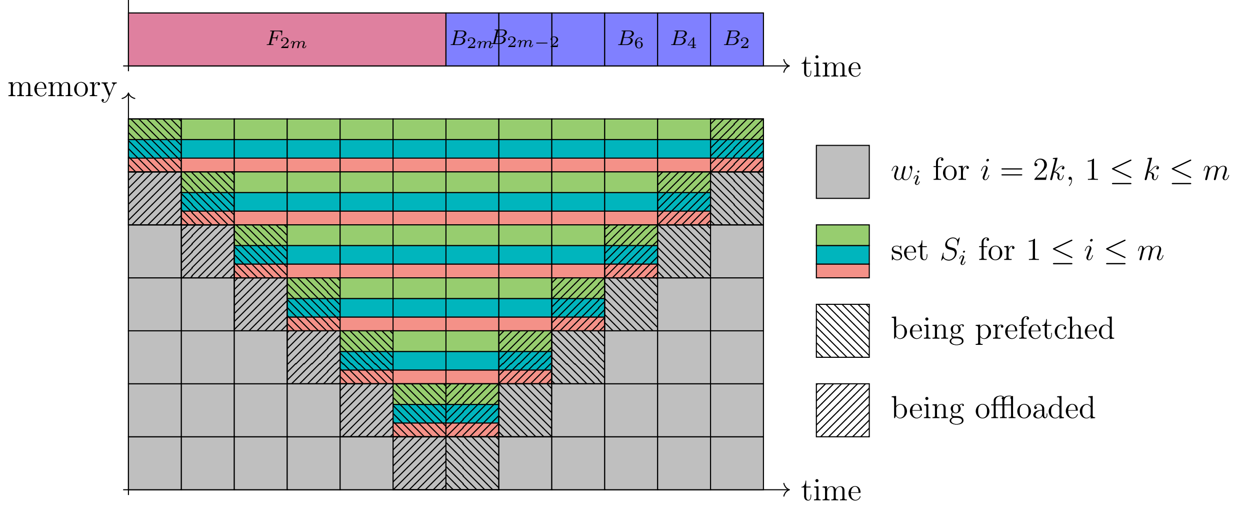


Figure 4.3: Instance and schedule used in the NP-completeness proof of Theorem 4.2.1.

We establish that this problem is NP-complete using a reduction to the 3-Partition problem, which is known to be NP-complete in the strong sense.

The 3-Partition problem can be formulated as: given a set of integers $\{x_0, x_2, \dots, x_{3m-1}\}$ such that $\sum_i x_i = mV$, decide if it is possible to partition it into m parts $\{S_1, \dots, S_m\}$ such that for any $j \le m, |S_j| = 3$ and $\sum_{i \in S_i} x_i = V$.

Given an instance for $OFF(L, M_{GPU}, \beta, 1)$, for which the existence of a valid schedule without any idle time implies the existence of a solution to the 3-partition problem, and a solution to the 3-partition problem leads to a feasible schedule for this instance.

Given an instance of 3-partition, we build the following instance of Problem 1 depicted in Figure 4.3:

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

We claim that this instance can be scheduled in time $T = 2m$ if and only if there exists a solution for the 3-partition instance.

Let us first assume that there exists a solution to the 3-partition instance, i.e., sets $(S_j)_{1 \le j \le m}$ such that $\sum_{i \in S_j} x_i = V$, and let us build a feasible schedule, as illustrated in Figure 4.3.

Before F_1, the GPU memory contains all w_i for $i \le 2m$, so the available memory is $M_{GPU} - \sum_{i=1}^{2m} |w_i| = V$. The forward operations for the first $2m$ layers take no time. With V free memory at the beginning of F_{2m}, prefetching w_{i+2m} for $i \in S_1$ and offloading w_1 can start simultaneously. After 1 time unit, both transfers are over, since $|w_1| = \sum_{i \in S_1} |w_{i+2m}| = V$. Prefetching w_{i+2m} for $i \in S_2$ and offloading w_3 can start immediately.

By the end of F_{2m}, all w_i for $i > 2m$ are prefetched into memory and all w_{2k-1} for $k \leq m$ are offloaded.

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

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

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

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

4.3 Method

4.3.1 Our Approach

In OFFMATE, we focus on reducing the memory footprint without any modification to the training result in any way. There are two major benefits: (1) this ensures that the model retains its original generalization capabilities, making fine-tuning applicable to all use cases where the original architecture is competitive. (2) In cases where lower precision or fewer trainable parameters are known to be valid, our solution can be seamlessly applied to the new model, further reducing the memory footprint.

Therefore, OFFMATE performs the same training task as the original model and ensures maximum compatibility. OFFMATE leverages the HIREMATE framework [18] and an Integer Linear Programming (ILP) formulation to efficiently combine all techniques: re-materialization, weight and activation offloading, CPU optimization, and optimizer state paging.

To reduce M_{param}, parameters can be offloaded onto CPU RAM. To reduce M_{p_grad}, parameters are optimized immediately when the associated gradients are generated and no gradient accumulation is applied. To reduce M_{opt_st}, for parameters which are optimized on GPU, the optimizer states can either be kept on GPU or offloaded to CPU RAM and moved back to GPU before the next optimization step. OFFMATE also considers *CPU optimization*: this allows parameters to be optimized on CPU, in which case optimizer states will always be kept in CPU RAM. This idea was originally proposed in ZeRO-Offload [49], where it was noted that since it is possible to (i) offload the gradient of a parameter, (ii) optimize on the CPU, and (iii) prefetch the new optimized parameter, CPU optimization involves the same amount of data transfers as regular offloading.

The main contribution of OFFMATE is to perform a holistic optimization of all these techniques simultaneously, so that different parameters can be handled by different techniques to minimize the iteration time within a limited memory budget. This strategy makes it possible to train the entire model within the memory requirements of a single block: in the context of very tight memory budgets, all data related to other blocks can indeed be offloaded to the CPU RAM. When more memory is available, OFFMATE automatically reduces the amount of offloaded data.

4.3.2 High Complexity

While combining different approaches maximize the memory reduction capability of OFFMATE, it does induces a high complexity of the problem. In Section 4.2.3, we have shown that many approaches are NP-Hard when they are considered independently. The exceptional high complexity of OFFMATE arises due to two key factors:

Synchronize operations We consider the training device is equipped with both a CPU and a GPU, allowing multiple operations to occur concurrently: (i) GPU computing, (ii) CPU computing, (iii) copying data from CPU to GPU, (iv) copying data from GPU to CPU. However, these operations are not completely independent in practice. Certain resources such as memory bandwidth between CPU and RAM is shared by both CPU computing and offloading. Synchronizing these diverse operations requires careful consideration of memory consumption and dependencies between tasks. For example, offloading an activation cannot occur until the forward pass has finished generating it, and optimizing a parameter on the CPU must wait until its corresponding gradient has been offloaded from the GPU to the CPU. This interdependency adds significant complexity to managing operations efficiently.

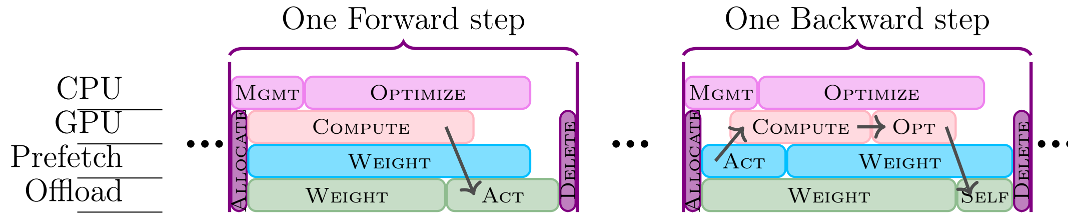


Figure 4.4: All operations performed in a step. MGMT represents management operations performed on CPU, ACT represents activation offloading or prefetching, and SELF represents offloading the weights of the current layer.

Large size of model OFFMATE targets at models that cannot fit into the single GPU memory. In practice, the targeted models generally contain billions of parameters which are represented in hundreds to thousands of tensors. Consequently, the number of operations involved is of a similarly high order of complexity.

Finding an optimal solution that combines multiple approaches becomes a significant challenge due to this complexity. Therefore, it is crucial to balance solving efficiency and memory reducing effectiveness within the OFFMATE framework. In the following sections, we discuss the decisions made to enhance the efficiency of the solution process while maintaining its overall effectiveness.

4.3.3 Framework

OFFMATE follows the similar pattern of solving proposed in Chapter 2 (ROCKMATE) and Chapter 3 (HIREMATE). We assume the the graph consists of different *blocks*, each representing an arbitrary computational task graph from part of the module. These *blocks* are also sorted in a topological order, even though their dependencies are not necessarily sequential. Unlike HIREMATE, no hierarchical solving is applied for re-materialization, therefore the internal structure of each *block* is not a main concern to us. Each *block* is provided with some execution schedules. For the top-level main graph, we use a different ILP solver O-ILP to include offloading and CPU optimization. The detailed formulation of O-ILP is described in Section 4.3.

The solution of O-ILP is a set of *steps* associated with the *blocks*, where each step involves four operations: executing a forward or backward schedule of a *block*, offloading some parameters, optimizer states or activations to RAM, prefetching some other previously offloaded data back to GPU memory, and performing an optimization step for some parameters on the CPU. The total time required for each step is determined by the longest duration among these four types of operations. A visual sketch of the operations performed in a step is provided in Figure 4.4, highlighting the dependencies between some of these operations, the memory allocation and deletion, and the management overhead incurred on the CPU.

The execution of these steps is *cyclic*: after all steps have been executed, the execution

resumes from the first step with a new input batch. This implies that a valid schedule must ensure the consistency of the set of parameters that are offloaded into RAM between the start and end of the schedule. For notational simplicity, when summing values over an interval of time steps, we assume that the index wraps around at 0, so that for $a > b$, $\sum_{t=a}^{b} Y_t$ is $\sum_{t=a}^{2L} Y_t + \sum_{t=0}^{b} Y_t$.

The overall framework of OFFMATE works as follows: we use RK-GB introduced in Chapter 2 (ROCKMATE) to obtain the computational graph, and run the recursive partitioning proposed in Chapter 3 (HIREMATE) to generate hierarchical graph representations. Next, the H-ILP solver is applied to every subgraph to generate the re-materialization schedule according to different memory requirements. Finally, the O-ILP solver combines these re-materialization schedules incorporating offloading and CPU optimization approaches. The solution of O-ILP is further post processed for code generation. The recompiled module is presented to the users, delivering the same outputs as the original module while keeping the memory consumption within the predefined budget.

4.3.4 Assumptions

As discussed in Section 4.3.2, the major challenge for OFFMATE is balancing the efficiency and effectiveness. To simplify the optimization problem, we make the following assumptions:

No parameter offloading below the top level OFFMATE uses a hierarchical ILP solving similar to HIREMATE. Schedules of each *block* handles only activations, detailed in Section 4.4. Also, no hierarchical solving is applied to each *block* to make complicated re-materialization schedules. Since activation is only part of the memory footprint concerned in OFFMATE, we further reduce its flexibility to balance the complexity and performance.

Activations are offloaded at the step when they are generated Activations are offloaded at the step in which they are produced, reducing the number of variables in the ILP formulation by restricting offloading decisions to sub-schedules. The algorithm is detailed in Section 4.4.

Only one schedule selected for every *block* To avoid introducing quadratic number of binary variables to represent re-materialization, we use only one schedule for each *block* in OFFMATE. There are $2L$ steps in the schedule, each with known dependencies of activation/parameter groups. Note that using single schedule does not exclude the usage of re-materialization since the selected schedule may include recomputation within the backward step.

All parameters are on GPU for executing a *block* Multiple parameter tensors can be required for executing a *block*. In practice, they can be swapped between RAM

and VRAM during the execution such that part of the memory is allocated on VRAM. However, representing the status of every single tensor will significantly increase the complexity of O-ILP. Therefore, we assume that all required tensors must be present on the GPU for the duration of executing the *blocks*. This simplifies the management of parameters during computation.

Memory allocation once per step In line with the previous assumption, we assume that memory allocation and deallocation occur only once at every step. Thus, the offloading/prefetching operations within a step require full memory allocation at the beginning. Both assumptions help us to use a single variable to represent each type of operations (offloading/prefetching) on a parameter or activation group in any given step, which will be variable Ofl_t^w and Prf_t^w in Section 4.5.

Represent fractional offloading in ILP During a step, the amount of parameters/activations offloading/prefetching is represented as the fraction of a group. Each group includes multiple tensors used within the same *blocks*. This fractional representation enables the use of continuous variables in ILP to capture the amount of offloading/prefetching, providing simpler representation comparing to employing a variable for each tensor.

Offloading is implemented only on whole tensors Although fractional offloading is represented in ILP, the actual implementation of offloading and prefetching is applied only for whole tensors. While it is technically possible to copy parts of a tensor between devices, the memory allocation can become expensive when they are done multiple times for every tensor. After solving the ILP, a greedy algorithm is applied to convert these fractional decisions into a group of full tensors out of the candidates group to manage the memory copying/allocation. The algorithm is described in details in Section 4.6.

No gradient accumulation Parameters are optimized immediately after their gradients are obtained, avoiding keeping parameter gradients on GPU for a long time. However, when gradients for a parameter are generated across different *blocks* (such as in the case of weight-sharing), partially generated gradients may still occupy GPU memory temporarily until optimization is complete.

No memory deallocation on CPU To enhance efficiency and enable asynchronous transfers between CPU and GPU, pinned memory is utilized to avoid incurring costly memory allocation overhead. The memory buffers are kept on RAM for the whole training process. This also ensures that optimizer states for parameters optimized on the CPU remain in CPU memory throughout the entire training process.

4.4 Preparing the *blocks*

Following the assumptions made in the previous section, we prepare the *blocks* with execution schedules. Three different re-materialization schedules are provided to each *block*, without solving optimization problems:

1. Executing as PyTorch, without re-computation.
2. Executing forward, while deleting activations as soon as they have no more users during forward. This schedule stores nothing at the end of forward except the outputs. Therefore, all the forward operations will be called again at the beginning of backward.
3. Executing forward, deleting activations that are generated by the *cheap* operations. We identify operations whose computational load is smaller than the time required to offload and prefetch their output data. All the stored activations will be **offloaded** to CPU at the end of forward. In practice, those offloading operations can start right after the tensors are generated. This algorithm is inspired by the *selective re-computations* from Megatron-LM [53].

Each schedule offers a choice to execute the given *block* with different requirements for the activations. Either computing time or bandwidth can be sacrificed to reduce the memory cost. Those schedules appear as options in O-ILP. They provide the standard solutions to handle the activation memory, without adding high complexity to the ILP formulation.

4.5 O-ILP

4.5.1 Optimization Problem

The target of our optimization is to minimize the time cost of each training iteration, including the synchronization of different operations. With the assumptions stated above, the resulting optimization problem to be solved by O-ILP is the following: given a set of L *blocks*, and for each *block* i, (1) a list of n_i re-materialization and activation offloading schedules (called *options*) without taking into account the memory cost of the parameters, and (2) the set P_i of all parameters used by that *block*. We distinguish between *interface activations*, which are passed from one *block* to another, and *internal activations*, which are stored in the forward phase of a *block* to be used during the backward phase. Each option o of *block* i may store on the GPU a different set of internal activations whose memory usage is denoted as S_i^o, and may offload another set of internal activations whose memory usage is denoted as O_i^o. We denote by T_t^o the computation time of option o of the t-th computation (the forward computation if $t \leq L$, otherwise the backward computation). For options that involve activation offloading, this computation time also includes the delay incurred by waiting for communications to complete (offload for forward computations, prefetch for backward computations).

The total size of a parameter w is denoted by $|w|$, and the size of its trainable part is denoted by $|w|_g$. If several tensor parameters are used by the same *blocks*, they are

considered together as a single parameter, where some parts may be trainable and others not. We denote by $B(t)$ and A_t the executed *block* and the size of the interface activations and gradients present in memory during step t. For a given parameter w, we refer to f_w as the first step that makes use of w, and to g_w as the step that computes the last gradient related to w.

We denote with α_G and α_C the update speeds of optimizing a parameter on the GPU and on the CPU respectively. We denote with β the bandwidth of the communication link between the CPU and the GPU, and with H the time required for the CPU to handle the management of all the other operations in the step (submitting the kernels to the GPU), which we estimate as a constant in all steps.

4.5.2 Linear Programming Formulation

The formulation involves the following variables:

$$\forall i \leq L, \forall o \leq n_i, \quad Comp_i^o \in \{0,1\} \tag{4.1}$$
$$\forall t \leq 2L, \quad Time_t \geq 0 \tag{4.2}$$
$$\forall t \leq 2L, \forall w, \quad Sto_t^w, Ofl_t^w, Prf_t^w, StoO_t^w, OflO_t^w, PrfO_t^w \in [0,1] \tag{4.3}$$
$$\forall w, \forall f_w < t < g_w, Opt_t^w \in [0,1] \tag{4.4}$$

$Comp_i^o$ is 1 if *block* i is executed with option o, and 0 otherwise. These variables satisfy $\forall i, \sum_o Comp_i^o = 1$. $Time_t \geq 0$ represents the duration of step t. Sto_t^w, Ofl_t^w, Prf_t^w, Opt_t^w are the fractions of parameter w that is respectively stored on GPU, offloaded, prefetched and optimized on CPU during step t. Finally, $StoO_t^w$, $OflO_t^w$ and $PrfO_t^w$ represent similar decisions on the optimizer states linked to w. We denote as $X_w = \sum_{t=g_w}^{f_w} Opt_t^w$ the fraction of parameter w which are optimized on CPU at some point, which is also the fraction of optimizer states stored on the CPU.

We now enumerate the constraints. First, the computation of a *block* i requires all its parameters (4.5) and the optimization is performed with the last backward step (4.6).

$$\forall t \leq 2L, \forall w \in P_{B(t)}, \; Sto_t^w \geq 1 \tag{4.5}$$
$$\forall w, \; StoO_{g_w}^w \geq 1 - X_w \tag{4.6}$$

Second, parameters and optimizer states not optimized on CPU must either be in the GPU or offloaded (4.7, 4.8), and bringing a parameter or optimizer state back to the GPU requires prefetching it (4.9, 4.10).

$$\forall t, w,\ Sto_t^w + \sum_{t'=g_w}^{t} Ofl_{t'}^w \geq 1 \tag{4.7}$$

$$StoO_t^w + \sum_{t'=g_w}^{t} OflO_{t'}^w \geq 1 - X_w \tag{4.8}$$

$$\forall t, w,\ Sto_{t+1}^w \leq Sto_t^w + Prf_t^w \tag{4.9}$$

$$StoO_{t+1}^w \leq StoO_t^w + PrfO_t^w \tag{4.10}$$

Third, performing the optimization on CPU requires fetching the optimized part of the parameter (4.11), the number of parameters optimized on CPU is at most the number of offloaded parameters (4.12), and parameters waiting to be optimized cannot be prefetched (4.13).

$$\forall w,\ |w|_g \sum_{t'=g_w}^{f_w} Opt_{t'}^w \leq |w| \sum_{t'=g_w}^{f_w} Prf_{t'}^w \tag{4.11}$$

$$\forall t, \forall w,\ |w|_g \sum_{t'=g_w}^{t} Opt_{t'}^w \leq |w| \sum_{t'=g_w}^{t} Ofl_{t'}^w \tag{4.12}$$

$$\forall t, \forall w,\ |w|_g(X_w - \sum_{t'=g_w}^{t} Opt_{t'}^w) \leq |w|(1 - Prf_t^w - Sto_t^w) \tag{4.13}$$

We denote by k the number of optimizer states per trainable value, which depends on the optimizer. We can express the global memory constraint as (4.17), where W_t, O_t, and S_t represent respectively the memory usage during step t of parameters, optimizer states, and stored internal activations:

$$W_t = \sum_w |w|\,(Sto_t^w + Prf_t^w) \tag{4.14}$$

$$O_t = \sum_w k|w|_g\,(StoO_t^w + PrfO_t^w) \tag{4.15}$$

$$S_t = \sum_{i \leq B(t)} \sum_o S_i^o \cdot Comp_i^o \tag{4.16}$$

$$\forall t,\ W_t + O_t + S_t + A_t \leq M_{GPU} \tag{4.17}$$

We also express a memory constraint for the memory of the CPU. It contains all the parameters of the model, the offloaded activations and optimizer states, as well as the optimizer states and gradient of each parameter optimized on CPU:

$$\sum_w |w| + \sum_{i,o} O_i^o \cdot Comp_i^o + \sum_w |w|_g \sum_t (k \cdot OflO_t^w + (k+1) \cdot Opt_t^w) \leq M_{CPU} \tag{4.18}$$

To avoid GPU idle time, we allow optimization operations on the CPU to be performed during the training iteration, overlapping with computation and communication operations. Our formulation also allows a parameter to be offloaded in the same step as the one it is used, but these operations cannot overlap. By denoting as L_t the time spent offloading parameters and optimizer states for the *block* of step t, we can express the time $Time_t$ of step t as:

$$L_t = \frac{1}{\beta} \sum_{w \in P_{B(t)}} |w| Ofl_t^w + |w|_g OflO_t^w$$

$$\text{(GPU Fwd)} \quad \forall t \leq L, Time_t \geq \sum_o Comp_{B(t)}^o T_t^o \tag{4.19}$$

$$\text{(GPU Bwd)} \quad \forall t > L, Time_t \geq \sum_o Comp_{B(t)}^o T_t^o + \frac{1}{\alpha_G} \sum_{w \in P_{B(t)}} (1 - X_w)|w|_g + L_t \tag{4.20}$$

$$\text{(Prefetch)} \quad \forall t, Time_t \geq \frac{1}{\beta} \sum_w (|w| Prf_t^w + k|w|_g PrfO_t^w) \tag{4.21}$$

$$\text{(Offload)} \quad \forall t, Time_t \geq \frac{1}{\beta} \sum_w (|w| Ofl_t^w + k|w|_g OflO_t^w) \tag{4.22}$$

$$\text{(CPU)} \quad \forall t, Time_t \geq H + \frac{1}{\alpha_C} \sum_w |w|_g Opt_t^w \tag{4.23}$$

The objective is then to minimize the overall duration $\sum_t Time_t$.

4.5.3 Adaptation: Batch Size Selection

The formulation introduce above, like all other approaches for re-materialization, assumes that the batch size is fixed, so that the forward and backward computation times are measured using samples of real size. For re-materialization problems, we assume that all the time and memory costs are proportional to the batch size (although time is usually not strictly proportional in practice). Therefore, changing the batch size is equivalent to changing the memory limit. In O-ILP, on the other hand, only T_t^o, S_t^o and O_t^o are affected by the batch size of the training task. All parameter-related operations (offload, prefetch, optimization, update) only depend on the model size and the bandwidth between the GPU and RAM. This makes batch size of the input task an interesting factor to optimize.

In this section, we present a variant of OFFMATE ILP, which also considers batch size as a variable to be optimized, under the assumption that computation times is proportional to the batch size. Users interested in fine-tuning tasks whose final accuracy is not sensitive to batch size can thus enable batch size tuning to achieve better overall throughput.

The natural way to do this would be to add a variable b to the ILP formulation and multiply all computation times by b. However, this would result in a quadratic (non linear) formulation. See for example the global memory constraint 4.17, which would

become:

$$\forall t,\, W_t + O_t + b \sum_{i \leq B(t)} \sum_{o} S_i^o \cdot Comp_i^o + b \cdot A_t \leq M_{GPU},$$

where both b and $Comp_i^o$ are variables in the formulation. Instead, we change the scale and divide all parameter-related times and memory by b. For this equation, this yields:

$$\forall t,\, \frac{1}{b} W_t + \frac{1}{b} O_t + \sum_{i \leq B(t)} \sum_{o} S_i^o \cdot Comp_i^o + A_t \leq \frac{1}{b} M_{GPU},$$

$$\text{where} \quad \frac{1}{b} W_t = \sum_{w} |w| \left(\frac{1}{b} Sto_t^w + \frac{1}{b} Prf_t^w \right)$$

This can be done while keeping a linear formulation by introducing a continuous variable $r \in [0,1]$, which represents $\frac{1}{b}$ and is interpreted as a fraction of the parameters to be used. We then view all parameter-related variables as bounded in $[0,r]$ instead of $[0,1]$: instead of writing $\frac{1}{b} Sto_t^w$ with $Sto_t^w \in [0,1]$, we write Sto_t^w with $Sto_t^w \in [0,r]$. The resulting formulation remains linear, with nearly the same constraints as before.

Some constraints where the constant 1 is used to represent the entirety of the parameters need to be modified, as follows:

$$\forall t \leq 2L, \forall w,\, Sto_t^w, Ofl_t^w, Prf_t^w, StoO_t^w, OflO_t^w, PrfO_t^w \leq r \tag{4.3'}$$

$$\forall w, \forall f_w < t \leq g_w,\, Opt_t^w \leq r \tag{4.4'}$$

$$\forall w \in P_{B(t)},\, Sto_t^w \geq r - X_w \tag{4.5'}$$

$$\forall w,\, StoO_{g_w}^w \geq r \tag{4.6'}$$

$$\forall t,\, Sto_t^w + \sum_{t'=g_w}^{t} Ofl_{t'}^w \geq r \tag{4.7'}$$

$$\forall t,\, StoO_t^w + \sum_{t'=g_w}^{t} OflO_{t'}^w \geq r - X_w \tag{4.8'}$$

$$\forall t, \forall w,\, |w|_g (X_w - \sum_{t'=g_w}^{t} Opt_{t'}^w) \leq |w| (r - Prf_t^w - Sto_t^w) \tag{4.13'}$$

$$\forall t,\, W_t + O_t + S_t + A_t \leq M_{GPU} \cdot r \tag{4.17'}$$

$$\forall t > L,\, Time_t \geq \sum_{o} Comp_{B(t)}^o T_t^o + \frac{1}{\alpha_G} \sum_{w \in P_{B(t)}} (r - X_w)|w|_g + L_t \tag{4.20'}$$

This adapted formulation is evaluated in Section 4.8.3.

4.6 Post-processing

The schedule obtained with ILP is feasible because it fits within the memory constraints, but the assumption of offloading fractional tensors (discussed in Section 4.3.4) introduces

Algorithm 8: Greedy grouping for parameter w

```
1  Input: Sto_t^w, Ofl_t^w for all t, T_w = set of all tensors in w
2  Output: Offload, Prefetch, Delete
3  def SELECT(set, value) = knapsack()
4  Initialize ofl = ∅
5  Initialize alive = T_w
6  for t ∈ steps from g_w do
7      if |w| · Σ_{t'=g_w}^{t} Ofl_{t'}^w > |ofl| then
8          set = T_w \ ofl  Offload[t] = SELECT(set, |w| · Σ_{t'} Ofl_{t'}^w − |ofl|)
           ofl = ofl ∪ Offload[t]
9      if |w| · Sto_t^w < |alive| then
10         set = ofl ∩ alive  Delete[t] = SELECT(set, |alive| − |w| · Sto_w^t)
           alive = alive \ Delete[t]
11     if |w| · Sto_t^w > |alive| then
12         set = T_w \ alive
13         kept = SELECT(set, |set| − |w| · Sto_w^t + |alive|)
14         Prefetch[t] = set \ kept
15         alive = alive ∪ Prefetch[t]
```

challenges in practice. Even though it is possible to copy a part of a tensor from one device to another, it requires a buffer to maintain the other part before releasing the memory. Both allocating the buffer and copying the data can lead to significant overhead in memory and execution time. To optimize execution, we choose to offload and prefetch whole tensors instead of fractional ones, leveraging the fact that parameters within a *block* are generally stored across multiple tensors. The goal of the post-processing step is as follows: for each parameter w and each time step t, to select a subset of tensors from w, ensuring that at each step, we offload and delete at least as much memory as required by the ILP solution, and prefetch at most what the solution requires. These constraints will ensure that the resulting schedule still fits in the GPU memory budget.

To achieve this, we use a greedy algorithm which repeatedly solves a particular case of the knapsack problem: given a set of tensors and a target value, find a subset whose size is larger but as close as possible to the target value. Algorithm 8 presents the main algorithm of this post-processing step.

For a given parameter w, Algorithm 8 starts with step g_w where all tensors are present in GPU memory, and successively selects tensors to offload so that the total offloaded size at step t is at least $|w| \sum_{t'=g_w}^{t} Ofl_t^w$. At each step, if some memory needs to be freed, some tensors are chosen from the set of already offloaded but not yet deleted tensors, again ensuring that at least the required amount of data is deleted. On the contrary, if the amount of available data is too low, some tensors are chosen from the set of deleted tensors, ensuring that at most the required amount of data is prefetched. This results in

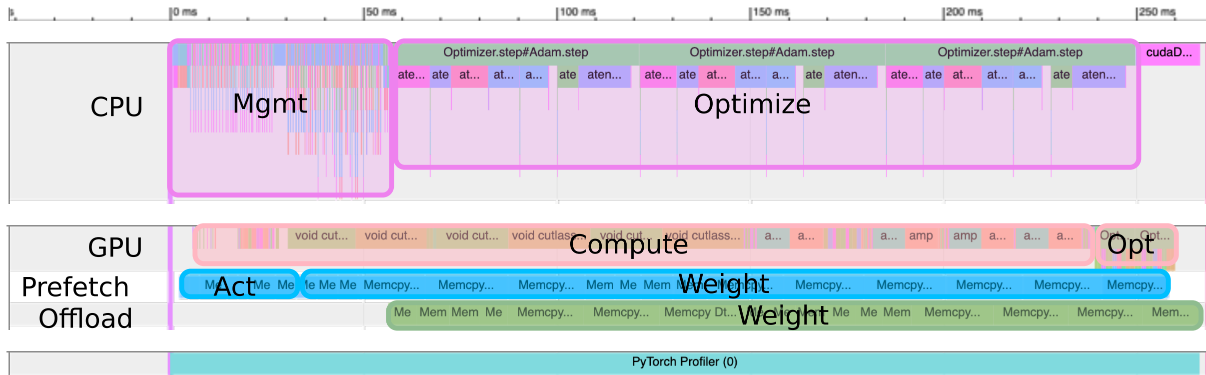


Figure 4.5: Tracing step from torch.cuda.profiler().

a new solution whose memory usage is never larger than the memory usage of the original ILP solution, and which only transfers entire tensors.

Once Algorithm 8 has selected which tensors to offload and prefetch, we identify candidate tensors for CPU optimization: those that are offloaded to the CPU after the backward computation, and are prefetched before the forward computation. We use the SELECT function once to globally select a sufficiently large set of tensors that will be optimized on CPU. Once this is done for all parameters w, the optimization operations of the selected tensors are greedily scheduled into the time steps in order of increasing *block* index, since *blocks* with lower index have a smaller range between backward and forward computation.

Finally, the optimizer states are also grouped to offload and prefetch. For this purpose, we run Algorithm 8 on optimizer states, with $StoO_t^w$ and $OflO_t^w$ as input, and where $\mathcal{T}_w$ is the set of optimizer states that have not been selected for optimization on CPU.

4.7 Additional Improvements

4.7.1 CPU Management

In Section 4.3 we assumed that different types of operations can be synchronized at the same time. In practice, different types of operations will interact with each others, which results in time overhead comparing to the simulation. Specifically, we do not optimize the single-threaded Python code, thus all the operations happening in a single step are submitted to CPU at the beginning of the step. Indeed, many of the operations are realized through CUDA, which occupies little CPU time, but they are not negligible comparing to the makespan of one step. The CPU optimization operations are scheduled after the CUDA operations to be handled through CPU, which differs from the assumption that CUDA operations are independent with the CPU optimization ones. Therefore, we adjust the O-ILP algorithm to include a constant CPU time cost at each step as term H in constraint 4.23. This is justified in the visualization of the tracing of one step as shown in Figure 4.5.

4.7.2 Grouping of CPU Memory Allocations

In the ILP formulation of Section 4.3, Constraint 4.18 expresses the constraint for CPU memory usage. The left side of Constraint 4.18 only contains data that need to be present in memory during an unmodified PyTorch execution, so the total amount of memory cannot be higher than the extrapolated memory usage on an infinite-memory GPU. However, we observed in practice that the CPU memory usage can be higher. This is due to the way PyTorch allocates pinned memory on CPU: to increase the possibility of reusing pinned memory buffers, PyTorch rounds up the data size to the next power of 2, which leads to high memory overhead in allocation when the desired tensor size is not a power of 2.

To obtain the best possible communication performance in OFFMATE, all the offloading is realized by moving data to a pinned memory buffer on the CPU memory, which is not reallocated between iterations. Indeed, unlike parameter gradients and activations, which may appear in different time periods on GPU memory, all the CPU memory allocations are performed at the first iteration and never released until the training is over. This PyTorch behavior is thus not useful for OFFMATE, and can incur a significant CPU memory overhead.

To overcome this issue and to allow OFFMATE to run on unmodified distribution of PyTorch for easier adoption, we include a heuristic in OFFMATE to reduce this memory overhead on CPU. Except for the parameters scheduled to be optimized by CPU, all other CPU allocations correspond to tensors that need to be offloaded from the GPU: they are not involved in computation but merely used for storing the data. Therefore, we analyze the sizes of all the required CPU allocations and merge them with a heuristic algorithm to obtain groups of size almost 2^n, and directly ask PyTorch to allocate the grouped data. All offload and prefetch operations are then performed with the corresponding parts of the allocations on CPU. This heuristic grouping allows OFFMATE to use at most as much CPU memory as an original, unmodified PyTorch execution on CPU.

4.8 Experiments

In this section, we evaluate the performance of OFFMATE on real fine-tuning scenarios. OFFMATE relies on `torch.export()` to obtain the task graph of the model. All experiments are run on PyTorch 2.3 and LLMs are loaded from HuggingFace Transformers v4.36. Library `peft` v0.11 is used to apply PEFT methods like LoRA on those models. To avoid memory fragmentation issues, we set the target memory budget to 2GB less than the total available GPU memory for ILP solving. Unless otherwise specified, all experiments are performed on a computer with an NVIDIA RTX 3060 GPU and an Intel Core i3-10105F CPU, where the GPU-to-CPU bandwidth is measured to be 10.4GB/s. The Integer Linear Program is solved using the default solver provided in the open source PuLP library[1]. For all experiments in this section, we use three options for each block

[1] https://github.com/coin-or/pulp

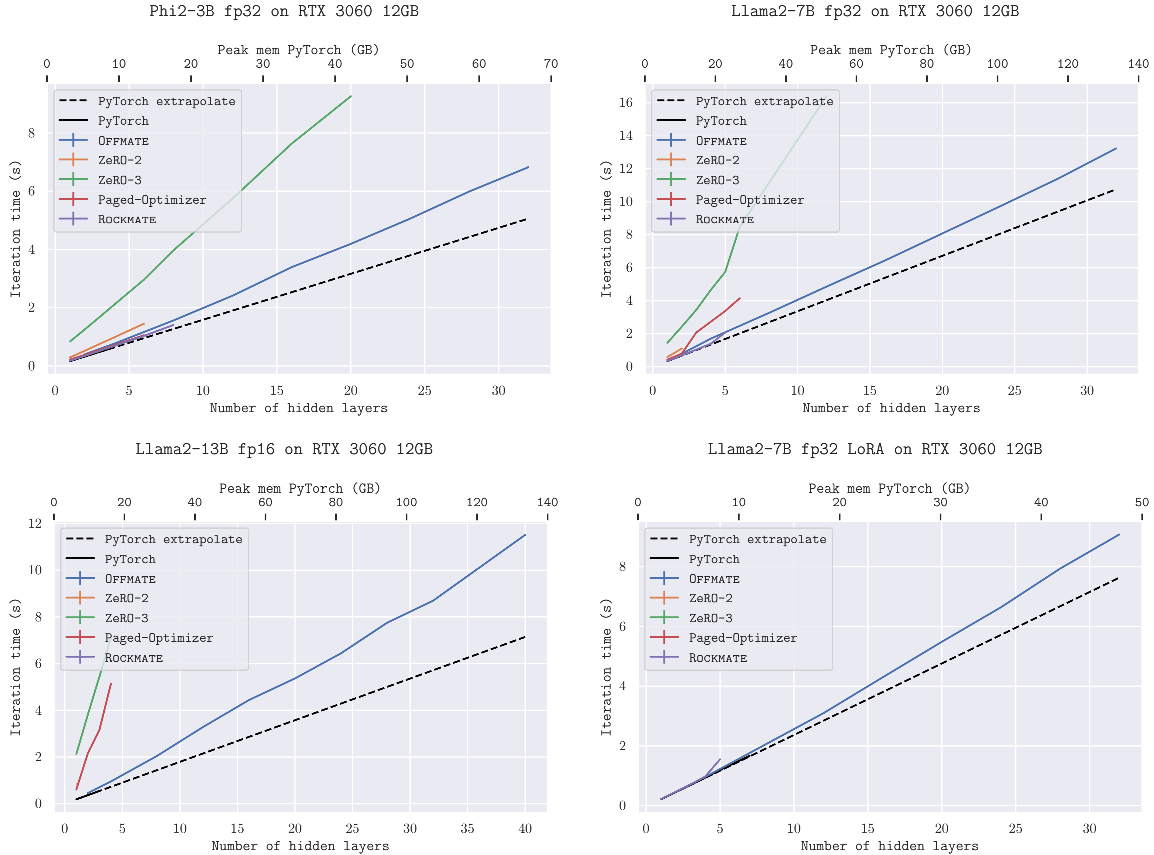


Figure 4.6: Average iteration time vs. number of hidden layers (full size models, with 32 or 40 layers, correspond to the right-most point in each plot). Batch size is 4 and sequence length is 512. The extrapolated PyTorch memory usage is marked on top of each graph. Adam optimizer is used for all the experiments. With LoRA, less than 1% parameters are trainable thus M_{p_grad} and M_{opt_st} are negligible. The LoRA model from `peft` is not compatible with ZeRO.

and limit the ILP solving time to 20 minutes to provide ready-to-use results. We also assume all the trainable parameters are updated with the Adam optimizer [28] during each iteration.

4.8.1 Fine-tuning Tasks

In this experiment, we study the performance of OFFMATE on four fine-tuning tasks: Phi-2-3B [26] and Llama2-7B [56] in floating-point 32 precision, Llama2-13B in bfloat16, and Llama2-7B with LoRA [20]. We evaluate the performance of OFFMATE compared to the following approaches:

1. PyTorch (Extrapolated): assuming the peak memory and iteration time of PyTorch execution depends linearly on the number of hidden layers, we measure the PyTorch

execution on small number of layers until it is out of memory, then extrapolate to predict the expected result on the full-size model.

2. Paged Optimizers proposed in QLoRA [14], using the implementation from HuggingFace Trainer. This experiment does not include quantization or low-rank adapters, only swapping the optimizer states between GPU and RAM.
3. ZeRO-2 [47], which uses ZeRO stage 2 and offload the optimizer states to RAM; ZeRO-Infinity [48], which uses ZeRO stage 3 to offload optimizer states and parameters, with default configuration. NVMe offload is not used in our experiments. Gradient checkpointing is also enabled through HuggingFace Trainer.
4. Rockmate introduced in Chapter 2, which can significantly reduce the memory usage from activations.

Since not all approaches can perform fine-tuning on the complete model, we present on Figure 4.6 the iteration time as a function of the number of hidden layers included in the model, where the right-most point corresponds to the full-size model. Figure 4.6 shows that OFFMATE is able to perform all the fine-tuning tasks with significantly lower time overhead. Specifically, OFFMATE is able to fine-tune a full-size Llama2-7B on a consumer-grade GPU, reducing the memory usage from 120 GB to 10 GB with only 20% overhead in the execution time comparing to the expected time without memory constraint.

Paged Optimizer, ZeRO-2 and Rockmate do not reduce all sources of memory and thus cannot perform fine-tuning as soon as the model size gets too big. ZeRO-Infinity has a more aggressive approach where all data are indiscriminately offloaded; this enables processing larger models, but induces a significant time overhead: the overhead of ZeRO-Infinity can reach more than 200%. It also induces Out of memory (OOM) on CPU RAM. OFFMATE avoids this problem by applying the CPU RAM constraint 4.18 in the ILP formulation.

4.8.2 Execution Tracing

We provide on Figure 4.5 a trace obtained from `torch.profiler` for the execution of one backward step on Llama2-7B, to be compared with the theoretical Figure 4.4. This figure highlights how OFFMATE is able to efficiently overlap both computation and communication to limit the idle time in practice. We also provide in Figure 4.7 a trace of the complete execution of all steps, which shows the high resource utilization and the similarity between the expected schedule and the actual execution. A CPU management time of $H = 50ms$ is used in ILP constraint 4.23.

The visualization of the execution trace shows that OFFMATE handles the operations in practice properly even though multiple assumptions were made in Section 4.3.4.

4.8.3 Batch Size

In this section, we evaluate the quality of the batch selection function of O-ILP introduced in Section 4.5.3. When the batch selection is turned on, a dynamic batch size is used in

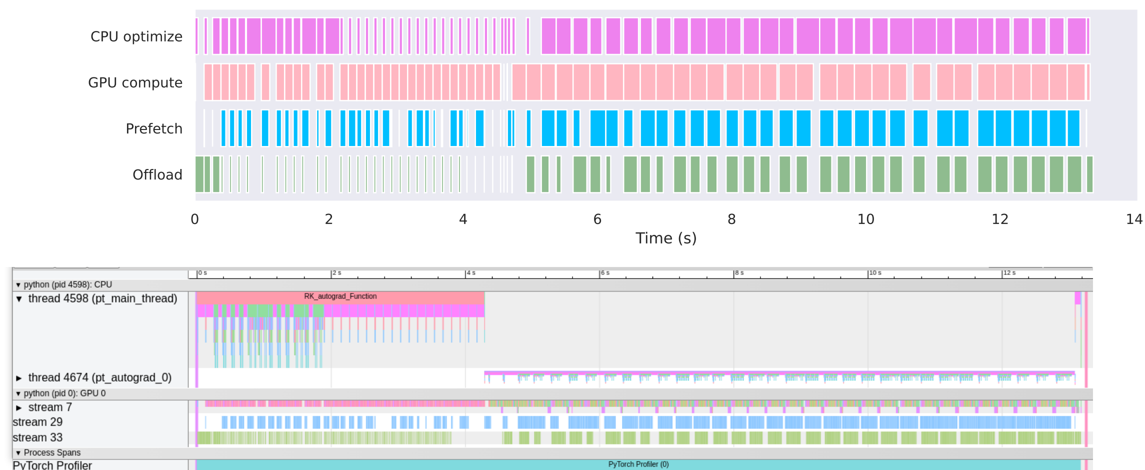


Figure 4.7: Annotated trace of a train iteration from Llama2-7B. This result corresponds to the rightmost point in Figure 4.6 of Llama2-7B.

RK-GB to allow for dynamic size in each operation. The dynamic_shapes function in `torch.export` is added in PyTorch 2.3. In the left of Figure 4.8, we present the selected batch size for different layers of Llama2-7B. Note that the optimized value of batch size is a continuous variable, which is rounded in our experiment to get the best value. In the right of Figure 4.8, we present the throughput of training 32-layer Llama2-7B when different batch sizes are used without selection. In O-ILP with batch size selection, number 4 is the rounded value for the optimized value. As shown in the throughput experiment, the batch size 4 is indeed the optimal value in ILP, but the measured results favors batch size 5 a bit more. The difference between the ILP throughput and the Scheduled throughput is due to the post-processing introduced in Section 4.6, where ILP solutions with continuous values are translated to practical operations. The difference between Scheduled throughput and the measured throughput is due to different behaviors in executing the operations. For instance, CPU optimization may take longer time than expected when multiple types of operations are processed during the same step, as discussed in Section 4.7.1. Overall, we show that the result of batch size selection in O-ILP is insightful but not necessarily optimal in practice. This is inevitable when a series of complicated operations is being simplified in the theoretical formula.

4.8.4 Ablation Study

In Table 4.2 we show the ablation study about the performance of OFFMATE on Llama2-7B on two different machines. Specifically, we compare the results when CPU optimization or activation offload are disabled.

Note that CPU RAM usage can become a bottleneck without the tensor merging introduced in Section 4.7.2. The number reported in Table 4.2 is the usage by OFFMATE, which is close to the limit of available RAM in the tested machine.

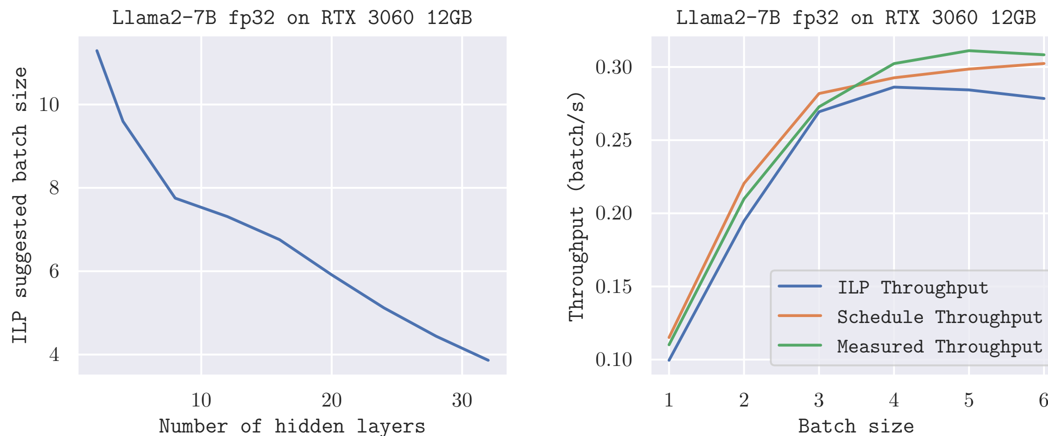


Figure 4.8: Batch size selection of Llama2-7B. Left: for models with different layers, suggested batch size value (continuous) from O-ILP. Right: For 32-layer model, throughput (batch/time) using fixed batch sizes.

GPU:VRAM	**CPU:RAM**	**Bandwidth**	$\alpha_{\mathbf{cpu}}$	$\alpha_{\mathbf{gpu}}$
RTX 3060 12GB	Intel i3-10105F 128GB	10.4GB/s	1.1GB/s	16.6GB/s
Method	**RAM usage**	**VRAM usage**	**Time**	**Overhead**
OffMate	105.7 GB	9.6 GB	13.1 s	120.3%
w/o CPU optim	96.1 GB	9.5 GB	15.2 s	139.5%
w/o Act offload	100.8 GB	9.9 GB	14.7 s	135.0%

Table 4.2: Comparing the results with Llama2-7B on two different machines (top and bottom). α_C and α_G represents the optimization speed on CPU and GPU. Input size is $(4, 512)$ which costs 40.8GB of activation size. Overhead is defined as the extra measured iteration time divided by PyTorch (Extrapolated) time shown in Figure 4.6. RAM usage does not include the system usage.

So far, we tested OFFMATE only with the RTX 3060 which is not very powerful as a consumer-grade GPU. When the GPU is slow, it gives sufficient space for the offloading and CPU optimization to overlap, which challenges the extension of OFFMATE to other platforms. In this experiment, we test OFFMATE with other GPUs that are more powerful. In Figure 4.9, we present the results of applying OFFMATE on other devices. This machine comes with a 128 GB AMD EPYC 7502 CPU and a 16 GB RTX 4080 GPU. Comparing to the RTX 3060 GPU, the RTX 4080 GPU offers approximately 8 times higher GPU computing speed, but its PCIe bandwidth is only twice as fast. As a result, re-materialization is a preferred choice comparing to activation offloading. O-ILP also takes into account the available CPU RAM, which is not sufficient for around 80 GB of activations. The overhead is mostly caused by re-materialization, which has a relative fixed overhead of 30 to 50%. Also note that in this context, CPU optimization is not

quite useful since the activations are much more significant than the parameters.

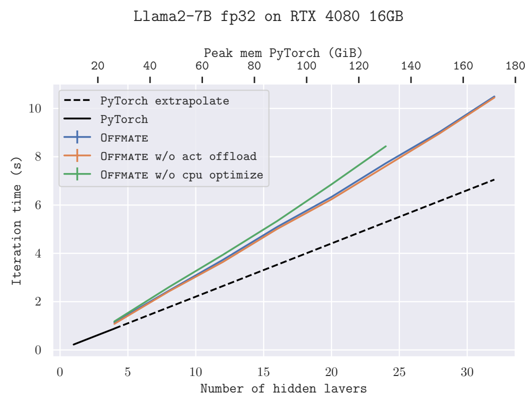


Figure 4.9: OFFMATE on different devices. The batch size chosen by O-ILP is 10 for the model on RTX 4080.

4.9 Conclusion

This chapter presents OFFMATE, a framework for efficiently reducing memory requirements when fine-tuning LLMs on a single consumer-grade GPU. This work has been summarized in the paper OFFMATE [65]. OFFMATE relies on integer linear programming to combine different memory-reducing approaches in a holistically optimized manner, ensuring efficient use of all available resource (computing, storage, communication).

It comes with certain limitations:

- OFFMATE does not support NVMe offload as ZeRO-Infinity. As shown in Table 4.2, the RAM size could become a bottleneck to fine-tune very large models.
- OFFMATE does not perform any approximation in training, but optimizing on CPU may change the results depending on the precision. Indeed, due to differences in machine computation, the same task running on CPU or GPU may produce slightly different results, and we confirm that the results obtained by OFFMATE are within this range.
- Furthermore, O-ILP requires that all parameters of a block are available when it is executed. A more fine-grained structure within blocks could help reduce the minimum memory required by offloading in the middle of block computations.
- Finally, we rely on a number of assumptions to limit the solving time of the ILP formulation. Different trade-offs between realism and tractability could be explored and may result in more efficient solutions. Specifically, we do not use a complicated H-ILP to generate the re-materialization solutions for every *block* in the experiments (which is available in the framework) for the consideration of solving time. If the ILP formulation is better optimized, it might allow for higher flexibility of different approaches.

Experiments presented in Section 4.8 show that OFFMATE significantly outperforms

the State-of-the-Art approaches and enables efficient fine-tuning of LLMs from HuggingFace Transformers. For Llama2-7B model, OFFMATE achieves a 10× reduction in GPU memory at the cost of a 20% increase in training time, without modifying the training task (e.g., lowering data precision or reducing the number of trainable parameters). We also show that OFFMATE can be combined with parameter-efficient fine-tuning methods such as LoRA [20]. OFFMATE is an easy-to-use framework that can enable fine-tuning on resource-constrained machines and is expected to have a significant impact for individual AI researchers. Future research efforts could add support for gradient accumulation or allow fine-grain offloading within a block to further reduce the minimum memory requirement.

Conclusion

In this thesis, we address the challenge of reducing memory requirements in neural network training. We achieve this by integrating several techniques, including re-materialization, offloading, and CPU optimization. Our final product is an automated tool that compiles a PyTorch `nn.Module` to fit within a specified memory budget. Unlike traditional model compression techniques, our approach ensures that the recompiled module produces the same outputs as the original, maintaining accuracy while managing memory efficiently. The trade-off is a slight increase in training iteration time, which we minimize through optimization algorithms designed for different scenarios.

Our first contribution, ROCKMATE, focuses solely on re-materialization. Given a PyTorch module, ROCKMATE decomposes the model into a sequence of *blocks*. We then apply an Integer Linear Programming (ILP) algorithm to generate multiple execution options for each *block*. These options are integrated using a Dynamic Programming (DP) model to create an overall schedule for the entire model, ensuring that memory usage remains within the given budget. This method is detailed in Chapter 2. ROCKMATE is particularly effective for popular large language models such as GPTs [44] and Llama [56].

To extend our work to accommodate more complex architectures where individual *blocks* are too large for ILP optimization, we developed HIREMATE. HIREMATE uses a recursive partitioning algorithm to break down the computational graph into multiple levels of smaller subgraphs, each with limited size. We then adapt the ILP solver to generate re-materialization schedules recursively, moving from the smallest subgraphs at the lower levels to the larger ones at the top. The adapted ILP solver allows for multiple execution options for each node in the graph, incorporating solutions from lower levels. This flexible framework, introduced in Chapter 3, supports various partitioning and solving algorithms, making it adaptable to any model architecture.

After the memory cost of activations is efficiently reduced, we focus on minimizing parameter-related memory usage. To achieve this, we propose OFFMATE, which leverages the computational power of both GPUs and CPUs as well as the communication bandwidth between them. We also simplify the optimization algorithm to ensure that solutions are generated within a reasonable time frame. The empirical results of OFFMATE, presented in Chapter 4, demonstrate a 10$\times$ memory reduction in a typical fine-tuning task, with only a 20% increase in iteration time.

In summary, we have developed efficient optimization algorithms and implemented

them within the Rockmate framework [2], making it easy-to-use for PyTorch models on NVIDIA GPUs. Our approach offers an effective solution for reducing memory requirements in neural network training across a wide range of scenarios. In the following sections, we explore potential applications of our work beyond the scope covered in this thesis, discussing how these techniques might be extended to other areas.

With the recursive partitioning proposed along with Hiremate in Chapter 3, our work is not strictly limited to a specific type of architectures. However, there are still different architectures that are challenging to us. First, our partitioning algorithms introduced in Section 3.3 are with $O(n^2 \log n)$ complexity, which is manageable for classical networks where thousands of operations are included in the graph. To solve the long-context issue in Transformers, state-space models are proposed including Mamba [16], where the operations are applied on each token instead of each sentence. As a result, the computational nodes in the graph can easily go up to 1 million. Models with those architectures will raise a great challenge to Hiremate: although H-Ilp can efficiently operate across multiple levels, the partitioning process would take an excessively long time to complete. To adapt Hiremate for these models, a more efficient partitioning algorithm is needed.

In this thesis, we focus on forward-backward training of neural networks, but our work is not limited to this scenario. For inference and non-backward training tasks, there is no activations saved from forward to backward, but it is still possible to have output of operations stored in memory which can be released by re-materialization. For example, in Transformers, the KV-cache is used during inference to store information generated for each token, which is then reused when processing future tokens. It is a common way to store the KV-cache in memory during the inference to avoid recomputing them. However, storing KV-cache can be memory consuming when the purpose is to produce a long sequence. Applying re-materialization in such cases could help manage memory usage without exceeding device limits. This approach can be extended to other models where significant memory is consumed by information that can be regenerated. Besides re-materialization, Offmate provides offloading solution for computational graph with known time/memory cost, which works on cases without backward as well.

Our work can also be extended to parallelism, while our experiments are conducted with a single device only. All our optimization algorithms consider only one memory budget, which makes it trivial to solve the parallelism problem with homogeneous memory requirements, like data parallelism. Regarding the other parallel training schemes, it is possible to consider the solutions provided in our framework and integrate them in a higher level to leverage the usage of different devices. The hierarchical solving approach could be especially useful in this case.

Within this thesis, we demonstrate all the experiments with PyTorch modules and NVIDIA GPUs. They are not necessary to apply our optimization algorithms. If one wishes to use a different deep learning framework, it is necessary to extract the computational graph with known time and memory cost of each node before using our

[2] https://github.com/topal-team/rockmate

optimization models. To recompile the model, it is also necessary to understand how to execute each node independently. For non-CUDA devices, a profiling tool must be available to measure the time and memory cost of different operations.

Overall, we have provided well-performed solutions to the problem of training neural networks with memory-limited device, and provided practical tools to users who wish to train PyTorch module on NVIDIA GPUs. Our work enables individual researchers and AI enthusiasts to contribute to the decentralized AI community without the need for expensive, cutting-edge hardware. This thesis also builds the framework to integrate further development of different partitioning and solving algorithms, allowing for extensions to different scenarios in deep learning.

Bibliography

[1] Shengnan An, Yifei Li, Zeqi Lin, Qian Liu, Bei Chen, Qiang Fu, Weizhu Chen, Nanning Zheng, and Jian-Guang Lou. Input-tuning: Adapting unfamiliar inputs to frozen pretrained models. *arXiv preprint arXiv:2203.03131*, 2022. 14, 86

[2] Burak Bartan, Haoming Li, Harris Teague, Christopher Lott, and Bistra Dilkina. MOCCASIN: Efficient tensor rematerialization for neural networks. In *International Conference on Machine Learning*, pages 1826–1837. PMLR, 2023. 16, 17, 86

[3] Olivier Beaumont, Lionel Eyraud-Dubois, Julien Hermann, Alexis Joly, and Alena Shilova. Optimal checkpointing for heterogeneous chains: How to train deep neural networks with limited memory, 2019. 2, 7, 42, 87

[4] Olivier Beaumont, Lionel Eyraud-Dubois, Julien Hermann, Alexis Joly, and Alena Shilova. Rotor, 2019. 27, 28, 53

[5] 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. 17, 86, 87

[6] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient Combination of Rematerialization and Offloading for Training DNNs. *Advances in Neural Information Processing Systems*, 34:23844–23857, 2021. 17, 19, 86

[7] 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. 16, 17, 19, 42, 58, 86

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. 16

[9] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023. 16, 19

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022. 16, 19

[11] 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. 14

[12] 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. 9

[13] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022. 86

[14] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023. 14, 19, 83, 85, 86, 103

[15] 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. 15

[16] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023. 109

[17] 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. 15

[18] Julia Gusak, Xunyi Zhao, Théotime Le Hellard, Zhe Li, Lionel Eyraud-Dubois, and Olivier Beaumont. hiremate: Hierarchical approach for efficient re-materialization of large neural networks. 2023. 82, 90

[19] 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. 16, 29, 30

[20] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 2, 6, 14, 85, 102, 107

[21] 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. 18

[22] Quzhe Huang, Mingxu Tao, Zhenwei An, Chen Zhang, Cong Jiang, Zhibin Chen, Zirui Wu, and Yansong Feng. Lawyer LLaMA Technical Report. *arXiv preprint arXiv:2305.15062*, 2023. 9, 83

[23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference, 2017. 13

[24] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020. 16, 17, 18, 53, 58, 59, 70

[25] 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. 2, 7, 25, 34, 86

[26] Mojan Javaheripi, Sébastien Bubeck, Marah Abdin, Jyoti Aneja, Sebastien Bubeck, Caio César Teodoro Mendes, Weizhu Chen, Allie Del Giorno, Ronen Eldan, Sivakanth Gopi, et al. Phi-2: The surprising power of small language models. *Microsoft Research Blog*, 2023. 102

[27] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023. 1, 5, 9, 83

[28] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 11, 84, 102

[29] 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. 16

[30] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023. 16

[31] 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. 9

[32] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient rematerialization for deep networks. *Advances in Neural Information Processing Systems*, 32, 2019. 53, 59

[33] 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. 18

[34] Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021. 15

[35] Yinheng Li, Shaofei Wang, Han Ding, and Hang Chen. Large Language Models in Finance: A Survey. In *Proceedings of the Fourth ACM International Conference on AI in Finance*, pages 374–382, 2023. 9, 83

[36] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. *arXiv preprint arXiv:2306.00978*, 2023. 86

[37] Junyang Lin, An Yang, Jinze Bai, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Yong Li, Wei Lin, et al. M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining. *arXiv preprint arXiv:2110.03888*, 2021. 18

[38] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022. 14, 86

[39] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023. 15

[40] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017. 13

[41] 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. 15

[42] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021. 10

[43] Uwe Naumann. Call tree reversal is NP-complete. In *Advances in Automatic Differentiation*, pages 13–22. Springer, 2008. 53

[44] OpenAI. ChatGPT: A Large-Scale Generative Language Model, 2022. 1, 2, 5, 7, 9, 25, 40, 83, 108

[45] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. POET: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*, pages 17573–17583. PMLR, 2022. 18, 86

[46] 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. 18

[47] 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. 19, 103

[48] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021. 19, 83, 86, 103

[49] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021. 18, 19, 83, 86, 90

[50] 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. 17

[51] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023. 86

[52] N. Shepperd. Fine tuning on custom datasets, 2021. 59

[53] 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. 15, 83, 94

[54] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014. 13

[55] Yi-Lin Sung, Varun Nair, and Colin A Raffel. Training neural networks with fixed sparse masks. *Advances in Neural Information Processing Systems*, 34:24193–24205, 2021. 14, 86

[56] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. 1, 4, 5, 8, 9, 83, 102, 108

[57] Albert Tseng, Jennifer J. Sun, and Yisong Yue. Automatic synthesis of diverse weak supervision sources for behavior analysis. *CoRR*, abs/2111.15186, 2021. 18

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 9, 52, 74, 83

[59] Gege Wen, Zongyi Li, Kamyar Azizzadenesheli, Anima Anandkumar, and Sally M Benson. U-FNO—An enhanced Fourier neural operator-based deep-learning model for multiphase flow. *Advances in Water Resources*, 163:104180, 2022. 52

[60] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022. 1, 5, 9, 83

[61] Chaoyi Wu, Xiaoman Zhang, Ya Zhang, Yanfeng Wang, and Weidi Xie. Pmc-llama: Further finetuning llama on medical papers. *arXiv preprint arXiv:2304.14454*, 2023. 9, 83

[62] Zifeng Wu, Chunhua Shen, and Anton Van Den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognition*, 90:119–133, 2019. 52

[63] 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. 15

[64] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021. 13

[65] Xunyi Zhao, Lionel Eyraud-Dubois, Théotime Le Hellard, Julia Gusak, and Olivier Beaumont. Offmate: full fine-tuning of llms on a single gpu by re-materialization and offloading. 2024. 106

[66] Xunyi Zhao, Théotime Le Hellard, Lionel Eyraud-Dubois, Julia Gusak, and Olivier Beaumont. Rockmate: An Efficient, Fast, Automatic and Generic Tool for Re-materialization in PyTorch. In *ICML 2023*, Honolulu (HI), United States, July 2023. 51

[67] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022. 15