Implicit parallelism for neural network acceleration

Haoran Wang

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE, THÉORIQUE ET INGÉNIERIE DES SYSTÉMES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE présentée par :

Haoran WANG

soutenue le : **27 Octobre 2022**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : **Informatique**

Implicit parallelism for neural network acceleration

JURY :

M. Serge PETITON	Professeur, Universite de Lille

RAPPORTEURS :

M. Noel DE PALMA (président de jury)	Professeur, University Grenoble Alpes
M. Denis BARTHOU	Professeur, ENSEIRB-MATMECA

THÈSE dirigée par :

M. Sébastien LIMET	Professeur, Université d'Orléans
Mme Sophie ROBERT	MCF, Université d'Orléans
M. Chong LI	Docteur, Huawei Technologies France

Remerciements

Les trois dernières années de mon doctorat apparaissaient sur l'écran de mon ordinateur comme un film alors que j'en tapais le dernier paragraphe. À 27 ans, j'étais passé d'un garçon qui pouvait jouer tout l'après-midi sur le terrain de basket à un jeune homme qui rentrait en Chine depuis l'aéroport de Paris-Charles de Gaulle avec une épaisse thèse sur le dos. Il n'a pas été facile de mener à bien cette thèse et je tiens à remercier tous ceux qui m'ont aidé pendant ces trois années, sans votre aide je ne peux pas croire que j'aurais pu terminer un travail aussi ambitieux.

Je tiens à remercier d'abord Monsieur Sébastien Limet et Madame Sophie Robert qui m'ont encadré tout au long de cette thèse. Qu'ils soient aussi remerciés pour leur gentillesse, leur disponibilité permanente et pour les nombreux encouragements qu'ils m'ont prodigués.

Je tiens à remercier tout particulièrement Monsieur Chong Li, qui est mon chef de travail et aussi mon frère dans la vie. L'idée de cette thèse de doctorat est venue de lui, et il m'a aidé tout au long de l'avancement et de l'évaluation de cette thèse. Je dois dire honnêtement que son aide a été un phare et un moteur pour moi afin de mener à bien cette thèse. Je ne peux pas oublier les moments amusants que nous avons passés ensemble à faire de l'accrobranche dans la banlieue de Paris, à regarder les phoques sur la plage en Normandie, et l'aide qu'il m'a apportée lorsque j'ai eu des difficultés à obtenir ma carte de séjour.

J'adresse tous mes remerciements à Monsieur Denis Barthou, Professeur à ENSEIRB-MATMECA, ainsi qu'à Monsieur Noel De Palma, Professeur à Université Grenoble Alpes, de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse. Je tiens à remercier Monsieur Serge Petiton d'avoir accepté de participer à mon jury de thèse.

Je tiens à remercier l'Association nationale de la recherche et de la technologie (ANRT) et le Centre de Recherche Huawei Paris (Huawei PRC) pour financier mes recherches de thèse. Je tiens à remercier le département des ressources humaines de Huawei et le département de la plateforme à Huawei PRC pour l'aide qu'ils m'ont apportée. Je tiens également à exprimer ma gratitude à tous mes collègues du Huawei Central Software Institute Paris pour toutes les discussions académiques et le brainstorming.

Je tiens à remercier tous les gens qui m'ont aidé directement dans le développement du projet de la thèse. Nous avons travaillé ensemble sur les discussions académiques, la conception du projet, le développement du code et la rédaction des papiers. En dehors du travail, Thibaut Tachon est la personne la plus ensoleillée que je connaisse, son sourire quotidien est contagieux. Pierre Leca et moi discutons beaucoup de la vie et des visions du monde, c'est amusant de comparer les idées chinoises et françaises, et il m'aide aussi avec beaucoup de problèmes de la vie française comme les déclarations d'impôts. Hongxing Wang et Ruiwen Wang sont les meilleures personnes que je connaisse pour créer l'ambiance. Sheng Yang, Zixi Chen et Quentin Petit ne parlent pas beaucoup mais sont très sérieux dans leur travail. Je tiens à dire que c'est un plaisir de travailler avec vous tous.

Je tiens à exprimer ma gratitude à l'équipe de Mindspore. J'ai reçu beaucoup d'aide pour le développement de l'ingénierie et j'ai appris à connaître la distance et les liens entre l'industrie et le monde universitaire. Je tiens à remercier tout particulièrement l'équipe d'AutoParallel, Teng Su, Xiaoda Zhang et tous les autres membres de l'équipe.

Il m'est impossible d'oublier Harenome Razanajato et Nelson Lossing pour leur aide précieuse pendant ma rédaction de thèse. Des conseils sur l'écriture, sur la révision du français, et surtout des astuces pour utiliser LaTeX afin de faciliter la rédaction de thèse.

Je tiens à remercier Gaétan Hains et Cédric Bastoul pour les conseils qu'ils m'ont prodigués lors de la rédaction des papiers et pour le bon exemple qu'ils m'ont donné dans le domaine académique. Je tiens également à remercier Hongbo Wang, Chunchun Yang, Weizhong Zhao, Lei Wang et Zhen Zhang pour les moments agréables que nous avons passés en tant qu'amis à la Huawei RPC et pour leur inspiration et leur aide dans mon parcours professionnel.

Il me sera très difficile de remercier tout le monde car c'est grâce à l'aide de nombreuses personnes que j'ai pu mener cette thèse à son terme.

Je tiens à remercier sincèrement mes parents pour m'avoir donné la vie, m'avoir élevé et m'avoir soutenu dans mes études. Ce sont leurs attentes qui m'ont permis d'arriver là où je suis aujourd'hui.

Enfin, je voudrais exprimer ma seule gratitude à la personne avec qui je passerai ma vie dans le futur, Yue Yang. Nous nous sommes rencontrés le premier mois où j'ai commencé ma thèse. Nous nous sommes soutenus mutuellement dans de nombreux moments difficiles et, plus encore, nous avons passé des moments heureux ensemble. Noël à Strasbourg, le yachting sur le lac de Sainte-Croix et la lavande

en Provence sont les plus belles notes de ce parcours doctoral. La tolérance de mon tempérament, et l'encouragement de mon travail, nos objectifs communs, nous ont permis d'aller plus loin. Juste parce que tu es toi, je suis moi.

Résumé

Ces dernières années, le domaine de l'Intelligence Artificielle (IA) s'est développé avec des succès spectaculaires et très médiatisés. En fait, l'IA est utilisé dans de nombreux domaines, de la vision par ordinateur au traitement du langage naturel. Parmi toutes les techniques d'intelligence artificielle; l'apprentissage profond basé sur les réseaux de neurones a montré des capacités d'apprentissage exceptionnelles avec de très bonnes performances dans de nombreux domaines. La conception et le développement de ces réseaux sont des tâches ardues qui nécessite des connaissances avancées en matière d'architectures parallèles modernes afin d'exploiter au mieux la puissance de calcul de ces machines. Une tendance notable des réseaux de neurones est augmentation exponentielle de leur taille afin d'obtenir des résultats de classification et de prédiction plus précis. Les gigantesques réseaux de neurones profonds (DNN, Deep Neural Networks) ont atteint des performances sans précédent dans des tâches d'IA difficiles et montrent une excellente capacité de généralisation à de nouveaux types de données non vue. Parmi ces DNN, on peut citer GPT-3 [9], GShard [42], Wide & Deep [94] et bien d'autres. Comme les DNN gigantesques peuvent avoir des trillions de paramètres (par exemple, Google Gopher à 600B paramètres [64]), l'entraînement d'un réseau étendu prend souvent des semaines, voire des mois. De plus, les plus grands réseaux peuvent généralement dépasser les limites de mémoire des accélérateurs de calcul individuels. Pour ces deux raisons, les milieux académiques et industriels commencent à utiliser des *clusters* d'ordinateurs pour distribuer l'entraînement des réseaux de neurones.

Les méthodes de partitionnement couramment utilisées pour distribuer un réseau de neurones comprennent le parallélisme de données, le parallélisme de modèles au niveau des opérateurs, le parallélisme de flux, etc. De nos jours, les performances optimales d'un réseau de neurones complexe sont généralement obtenues en utilisant un mélange des méthodes de parallélisme ci-dessus, appelé parallélisme hybride. L'élaboration d'un plan parallèle exige des chercheurs et des ingénieurs en IA qu'ils aient des connaissances en matière de calcul parallèle et nécessite également du temps et des efforts pour concevoir et vérifier les performances. Ce fait a donné lieu à un sujet de recherche essentielle: la génération automatique de plans parallèles. Le calcul d'un plan parallèle pour des DNNs gigantesques est un défi. Un DNN gigantesque peut contenir des dizaines de milliers d'opérateurs (par exemple, MatMul et Relu). Chaque opérateur peut avoir plusieurs tenseurs, et de nombreuses dimensions peuvent être choisies lors de leur partitionnement. Les combinaisons des dimensions de partitionnement potentiels sont énormes [35], et un planificateur de parallélisme doit évaluer toutes les combinaisons pour trouver un plan optimal. De plus, lors de la recherche d'un plan optimal, le planificateur doit prédire les performances des plans candidats. Le profilage des opérateurs DNN permet souvent d'atteindre cet objectif du point de vue de l'utilisation du calcul, de la mémoire et de la communication. Cependant, le profilage des opérateurs DNN avec tous les dispositifs matériels possibles à un coût prohibitif et de telles données de profilage sont souvent indisponibles.

Les planificateurs de parallélisme existants supportent mal les DNN gigantesques. OptCNN [36], FlexFlow [35], ToFu [87] et TensorOPT [12] explorent largement les configurations de parallélisme possibles mais ne sont pas en mesure de déterminer des plans optimisés pour les DNN. Les planificateurs optimisés pour les pipelines, tels que PipeDream [56], Dapple [20], et Piper [80], réduisent l'espace de recherche de la configuration en utilisant d'abord des règles manuelles pour trouver des plans réalisables pour le parallélisme de données et de modèles, puis en optimisant la configuration pour le parallélisme de flux. Cette conception empêche ces planificateurs d'optimiser conjointement le parallélisme de données, de modèle et de flux, ce qui entraîne la découverte de plans sous optimaux. Elle induit en outre un temps d'exécution considérable, ce qui rend l'utilisation des planificateurs fastidieuse et coûteuse dans la pratique.

Hormis les inconvénients distincts des travaux ci-dessus, ils présentent tous un inconvénient commun: les modèles de coups numériques proposés pour tous basés sur le temps d'exécution de l'opérateur de profilage sous un matériel particulier. Ce type d'approche introduit un effort de préparation coûteuse sans garantie d'optimalité. En outre, il faut des heures, voire des jours, pour trouver le plan optimal d'un réseau de neurones étendu. La méthode de ces travaux part de l'expérience existant d'experts, définit un espace de recherche du plan parallèle, propose un modèle de coup basé sur les données de profilage et transforme le problème en modélisation mathématique. Ils résolvent ensuite ce problème de recherche en concevant un solveur. Cependant, cette méthode présente les lacunes suivantes:

- Si on définit un espace de recherche étendu, la préparation du profilage sera très longue, et la recherche ultérieure du solveur prendra un temps inacceptable. Si la taille de l'espace de recherche

est limitée manuellement, il y a un risque de ne pas trouver le plan optimal réel.

- Comme ces méthodes réduisent le temps de recherche en restreignant l'espace de recherche, on ne peut jamais trouver de meilleurs plans que ceux définis par les experts.

- Le profilage est coûteux, et les chercheurs et ingénieurs en IA sans connaissance en parallélisme ne savent pas comment effectuer une analyse de profilage.

- La conception de l'espace de recherche est basée sur l'expérience des experts. Les méthodes doivent être repensées et reconstruites pour s'adapter à de nouvelles structures de réseaux qui n'ont pas été considérées auparavant.

En résumé, ces méthodes ne fournissent pas un bon équilibre entre généralité et précision dans ce problème de recherche NP-hard.

Afin de contourner le processus de profilage coûteux des méthodes de pointe et de fournir un algorithme permettant d'obtenir un plan parallèle hybride précis en peu de temps, cette thèse propose une machine abstraite hiérarchique symétrique et un modèle de coût symbolique qui découple le matériel de l'algorithme parallèle. Basée sur le modèle BSP, cette approche élimine le besoin de profilage sur du matériel spécifique pour chaque opérateur. En se basant sur la sémantique des réseaux de neurones informatiques, le modèle de coût symbolique peut être transformé et réduit. Cette thèse propose un algorithme qui réduit la complexité des problèmes de recherche NP-hard à une complexité linéaire et peut générer des algorithmes parallèles hybrides efficaces en quelques secondes. Les résultats visent à être intégrés dans l'environnement open-source MindSpore de Huawei et à contribuer aux produits et solutions d'IA de Huawei afin d'exploiter toute la puissance des puces Ascend de Huawei.

Cette thèse utilise des approches de parallélisme structuré pour optimiser l'apprentissage profond distribué afin que les cadres DL puissent gérer automatiquement les performances. Par conséquent, les concepteurs de DL pourraient se concentrer davantage sur le développement de DNN plus précis sans encourir de coûts supplémentaires. Pour décrire les caractéristiques d'une machine d'apprentissage d'IA moderne, HSM2DL a créé une machine abstraite hiérarchique et symétrique. HSM2DL a également proposé un modèle d'exécution pour caractériser le processus d'entraînement parallèle hybride et un modèle de coût symbolique comme métrique pour évaluer le coût produit par divers plans parallèles.

L'objectif principal de HSM2DL est la découverte de plans parallèles hybrides optimaux. Le partitionnement récursif et le graphe Flex-Edge Recursif (FER) sont deux contributions clés au niveau des opérateurs. Le partitionnement récursif partitionne le graphe de calcul en deux parties, étape par étape, jusqu'à ce que le graphe soit partitionné sur le nombre de dispositifs. Ce partitionnement récursif en deux parties est basé sur une caractéristique de machine abstraite symétrique. Au lieu de parcourir directement toutes les possibilités, la complexité de la recherche est devient linéaire par rapport au nombre de dispositifs. Le graphe FER permet de réordonner les opérateurs en fonction de leur importance, et le retour en arrière est évité. Par conséquent, avec une complexité de recherche linéaire, l'algorithme peut trouver un plan parallèle optimal au niveau des opérateurs. Le rôle et l'efficacité des techniques de chevauchement tout-réduit dans le parallélisme distribué sont discutés en tant que complément au parallélisme au niveau des opérateurs. Une méthode pour calculer le *tail factor* est également fournie dans cette thèse. Elle peut être utilisée conjointement avec l'algorithme D-Rec pour le parallélisme au niveau opérateur. Enfin, cette thèse présente une méthode de recherche conjointe pour le parallélisme hybride flux et modèle. Basé sur le modèle de *transformer* et le nombre de dispositifs 2^p, un algorithme de recherche d'une stratégie de parallélisme est fourni. En comparaison avec la méthode SOTA Alpa [97], l'efficacité de l'algorithme et la haute qualité du plan sont démontrées expérimentalement.

En conclusion, la principale contribution de cette thèse est de proposer HSM2DL pour l'apprentissage profond distribué. Ce modèle propose une machine abstraite hiérarchique et symétrique pour simuler la machine d'entraînement réel et le modèle d'exécution et le modèle de coût de l'entraînement distribué. Sur la base de ce modèle, cette thèse propose un algorithme de recherche efficace à la fois pour la recherche au niveau de l'opérateur et la recherche conjointe. L'avantage de cette méthode est qu'elle équilibre bien l'optimalité des résultats de la recherche et la portabilité de l'algorithme de recherche.

Ce document est organisé comme suit:

- Le Chapitre 2 présente d'abord l'état de l'art de l'apprentissage profond distribué et de la recherche automatique de plans parallèles. Ce chapitre aborde le contexte de l'apprentissage profond distribué,

les principes de base de entraînement des DNN distribués et les travaux connexes de la recherche automatique de plans parallèles.

- Le Chapitre 3 décrit d'abord la base des modèles de calcul parallèle et compare différents modèles représentatifs, notamment PRAM, LogP et BSP. Sur la base du modèle de BSP, le chapitre détaille HSM2DL. La machine abstraite de HSM2DL présente deux caractéristiques principales : hiérarchique et symétrique. Ensuite, le modèle d'exécution de HSM2DL décrit les méthodes de calcul de l'entraînement distribué du DNN. HSM2DL propose un modèle de coût symbolique qui décrit clairement les coûts des plans HP pour l'entraînement DNN distribué. Ce modèle offre une excellente opportunité pour des analyses et optimisations ultérieures. Ce modèle permet la recherche systématique de la HP optimale des DNNs. La définition des problèmes de recherche est également définie dans ce chapitre. Enfin, ce chapitre se termine par une comparaison entre HSM2DL et les travaux connexes.

- Le Chapitre 4 présente la méthodologie de recherche de plans parallèles optimaux au niveau des opérateurs. En se basant sur la machine abstraite hiérarchique et symétrique, HSM2DL propose un partitionnement récursif, qui réduit la complexité de la recherche en fonction du nombre de dispositifs. En outre, inspiré par le troisième homomorphisme, HSM2DL propose un graphe récursif Flex-Edge dont les opérateurs peuvent être réorganisés librement, évitant ainsi le retour en arrière du problème de recherche et réduisant donc la complexité de la recherche. L'optimalité du partitionnement récursif et du réordonnancement des sommets est prouvée dans ce chapitre. Les expériences montrent que la qualité des plans parallèles trouvés correspond aux plans SOTA avec un temps de recherche très faible.

- Le Chapitre 5 décrit le comportement overlap de la synchronisation des paramètres. Ce chapitre détaille comment le coût de synchronisation est généré et comment il peut être caché avec les techniques overlap. Ce chapitre traite de la modélisation symbolique du *tail factor* et de son modèle de coût. Un algorithme permettant de calculer les facteurs de queue pour un graphe de calcul donné et un plan parallèle est également présenté. Les expériences montrent qu'avec un *tail factor* précis, le résultat de la recherche de D-Rec peut être amélioré. L'exactitude de l'algorithme du *tail factor* est également validée.

- Le Chapitre 6 décrit une premier tentative très préliminaire de recherche de partitionnement pipeline. Cette tentative est basée sur deux hypothèses fortes. Tout d'abord, cette thèse suppose que le nombre de dispositifs et le nombre d'étages sont tous deux une puissance de deux. Une autre hypothèse est de ne considérer que le modèle DNN basé sur les *transformers*. Dans ce chapitre, HSM2DL complète le modèle de coût symbolique du les temps d'attente et propose un algorithme de recherche conjointe qui peut trouver les plans parallèles correspondant à SOTA en un temps très court. Des expériences sur des *clusters* de grande envergure montrent que HSM2DL peut surpasser les planificateurs SOTA jusqu'à 255× lorsqu'il traite des modèles à grande échelle basés sur des *transformers*.

Contents

Liste des figures viii

Liste des tables ix

1 Introduction 1
- 1.1 Summary . . . 1
- 1.2 Overview . . . 3
- 1.3 Publications . . . 3

2 State Of the Art 5
- 2.1 Introduction . . . 5
- 2.2 Background of distributed deep learning . . . 5
  - 2.2.1 Deep learning . . . 5
  - 2.2.2 Training a deep neural network . . . 7
  - 2.2.3 Distributed deep learning . . . 8
- 2.3 Basics of distributed DNN training . . . 9
  - 2.3.1 Computational graph, operator and tensor . . . 9
  - 2.3.2 Standard parallelism plans . . . 11
  - 2.3.3 Parallel cost of standard plans . . . 15
  - 2.3.4 Hybrid parallelism plans . . . 15
  - 2.3.5 Challenges of training with hybrid parallel plans . . . 17
- 2.4 Automatic parallel plan search . . . 19
  - 2.4.1 Introduction of automatic parallel plan search . . . 19
  - 2.4.2 Problem Positioning . . . 19
  - 2.4.3 Related works . . . 19
  - 2.4.4 Limitations of existing approaches . . . 22
- 2.5 Conclusion . . . 23

3 Parallel computing model for distributed deep learning 25
- 3.1 Introduction . . . 25
- 3.2 Basics of parallel computing model . . . 25
- 3.3 Typical parallel computing models . . . 28
  - 3.3.1 Parallel random access machine (PRAM) . . . 28
  - 3.3.2 Bulk synchronous parallel model (BSP) . . . 31
  - 3.3.3 LogP model (LogP) . . . 32
  - 3.3.4 Comparison of LogP model and BSP model . . . 34
- 3.4 Hierarchical and symmetric model for distributed deep learning (HSM2DL) . . . 34
  - 3.4.1 Abstract Machine . . . 35
  - 3.4.2 Execution model of HSM2DL . . . 36
  - 3.4.3 Symbolic cost model . . . 39
- 3.5 Inputs and parallel plan range of HSM2DL . . . 39
  - 3.5.1 Planner input . . . 39
  - 3.5.2 Hybrid parallelism plan . . . 40
- 3.6 Comparison between HSM2DL and the related works . . . 41

3.6.1 Related works' methodology . . . 42
3.6.2 HSM2DL's methodology . . . 43
3.6.3 Conclusion . . . 44

4 Operator Level Parallel Plan Search **45**
4.1 Introduction . . . 45
4.2 Symbolic simplification based on the abstract machine . . . 46
4.2.1 Symbolic simplification . . . 46
4.2.2 Recursive partitioning . . . 48
4.2.3 Optimality proof . . . 50
4.2.4 Primitive configurations and their costs . . . 51
4.3 Functional recursive cost analysis . . . 54
4.3.1 Symbolic transformation based on the homomorphism . . . 54
4.3.2 Flex-Edge Recursive graph . . . 58
4.4 Double recursive algorithm . . . 61
4.5 Experiments . . . 62
4.5.1 Environment setup . . . 62
4.5.2 Searching efficiency . . . 64
4.5.3 Parallel plan quality . . . 64
4.6 Conclusion . . . 66

5 Parameter synchronization overlap **67**
5.1 Introduction . . . 67
5.2 Hybrid parallelism and parameter synchronization overlap . . . 67
5.3 Tail factor calculation based on HSM2DL . . . 69
5.3.1 Modeling the tail factor . . . 69
5.3.2 Tail Factors Algorithm . . . 69
5.4 Experiments . . . 70
5.4.1 Experiments Setup . . . 70
5.4.2 Results . . . 71
5.5 Conclusion . . . 72

6 Graph level plan search **73**
6.1 Introduction . . . 73
6.2 Graph-level partitioning analyses . . . 74
6.2.1 Scope of HSM2DL's pipeline partitioning . . . 74
6.2.2 Pipeline partitioning analysis . . . 76
6.3 Joint search algorithm for pipeline and operator parallelism . . . 77
6.3.1 Operator-level candidate plans searching . . . 78
6.3.2 Pipeline search based on the candidate plans . . . 78
6.3.3 Time complexity and practical considerations . . . 79
6.4 Experiments . . . 80
6.4.1 Environment setup . . . 80
6.4.2 Planning time . . . 81
6.4.3 Plan quality . . . 82
6.5 Conclusion . . . 83

7 Conclusion **85**
7.1 Conclusion . . . 85
7.2 Perspectives . . . 85

Bibliographie **87**

List of Figures

2.1 DNN Model Training Process . . . 7
2.2 Data Parallelism . . . 11
2.3 Operator-level model parallelism . . . 12
2.4 Model (graph-level) Parallelism . . . 13
2.5 Pipeline execution process . . . 14
2.6 Model (graph-level) parallelism Training Process . . . 16
2.7 Minimal neural network example . . . 18
2.8 Possible distribution of a 3D tensor over 4 devices . . . 18
2.9 The positioning of automatic parallel plan generation . . . 20

3.1 Compositions of parallel computing model . . . 27
3.2 Parallel random access machine (PRAM) . . . 29
3.3 Bulk synchronous parallel machine (BSP) . . . 30
3.4 A SuperStep of BSP . . . 31
3.5 A typical DNN cluster . . . 35
3.6 Operator level Modeling . . . 36
3.7 Pipeline Modeling . . . 36
3.8 Double Level Execution Model . . . 38
3.9 Example of a hybrid parallelism plan. . . . 41
3.10 Comparison between related works and HSM2DL . . . 42

4.1 A typical GPU architecture described by a recursive tree . . . 47
4.2 Example of primitive configurations of a 2D MatMul . . . 49
4.3 Partitioning MatMul along output-independent dimension . . . 53
4.4 Partitioning MatMul along output-dependent dimension . . . 53
4.5 Redistribution cost between operators . . . 54
4.6 Leftward and rightward processing of $cost_{all\_rdst}$ over a vertex list . . . 56
4.7 Flex-Edge Graph Traversal: the number after the column denotes the order of the vertex. $v2:3$ means $v2$ is ordered at the third place. . . . 59
4.8 Searching Time of ResNet101 . . . 63
4.9 Searching Time of BERT . . . 64

5.1 Timeline of Distributed DNN Training . . . 68
5.2 Tail Time in Backward Propagation . . . 70
5.3 HP speedup with the number of classes . . . 71

6.1 Pipeline partition example and cluster mapping . . . 74
6.2 Two types of the pipeline scheduler and two examples of each . . . 75
6.3 A running example of Algorithm 3 . . . 79
6.4 A running example of Algorithm 4 . . . 81
6.5 HSM2DL planning time for GPT-3. . . . 82

List of Tables

2.1 Tensor attributes 10

3.1 Some fundamental concepts of parallel programming 28

4.1 Performance on varieties of DNN models 65
4.2 Portability w.r.t. the scale of cluster 65
4.3 Portability w.r.t. hardware architecture 66

6.1 DNN configurations and statistics 82
6.2 BERT planning time with Piper and HSM2DL 82
6.3 Plan quality of transformer-based DNN models 83

Chapter 1

Introduction

1.1 Summary

The Artificial Intelligence (AI) field has been growing with spectacular, high-profile successes in recent years. In fact, AI is applied in many fields, from computer vision to natural language processing. Among all the techniques of artificial intelligence, neural network-based deep learning has shown outstanding learning capabilities with outstanding performance in many areas. The design and development of such networks is an arduous task that requires advanced knowledge of modern parallel architectures in order to make the best use of the computing power of such machines. A noticeable trend in neural networks is their exponential increase in size in order to search for more accurate classification and prediction results. Gigantic deep neural networks (DNNs) have achieved unprecedented performance in challenging AI tasks and show the excellent capability of generalizing to unseen data. Examples of such DNNs include GPT-3 [9], GShard [42], Wide & Deep [94] and many others. As gigantic DNN can have trillions of parameters (e.g., Google Gopher has 600B parameters [64]), training an extensive network often takes weeks or even months. Moreover, the larger networks may usually exceed the memory limits of individual computational accelerators. For these two reasons, both academia and industry are beginning to use computer clusters to train neural networks in a distributed way.

Commonly partition methods used to distribute a neural network include data parallelism, operator-level model parallelism, pipeline model parallelism, etc. The optimal performance of a complex neural network nowadays is usually obtained using a mixture of the above parallelism methods called hybrid parallelism. Building a parallel plan requires AI researchers and engineers to have parallel computing knowledge and also needs time and effort to design and verify performance. This fact resulted in a hot research topic: automatic parallel plan generation. The computation of a parallel plan for gigantic DNNs is challenging. A gigantic DNN can contain tens of thousands of operators (e.g., MatMul and ReLu). Each operator can have multiple tensors, and many dimensions can be chosen when partitioning tensors. The combinations of the potential partitioning dimensions are enormous [35], and a parallelism planner needs to evaluate all combinations to find an optimal plan. In addition, in the search for an optimal plan, the planner must predict the performance of candidate plans. Profiling DNN operators often achieve this from the perspective of computation, memory, and communication usage. Profiling DNN operators with all possible hardware devices are prohibitively expensive, and such profiling data is often unavailable.

Existing parallelism planners exhibit poor support for gigantic DNNs. OptCNN [36], FlexFlow [35], ToFu [87] and TensorOPT [12] explore possible parallelism configurations extensively, but they failed to return optimized plans for gigantic DNNs. Pipeline-optimized planners, such as PipeDream [56], Dapple [20], and Piper [80], reduce configuration search space by first using manual rules to return feasible plans for data and model parallelism and then optimizing the configuration for pipeline parallelism. This design prevents these planners from jointly optimizing for data, model, and pipeline parallelism, resulting in finding sub-optimal plans. It further incurs tremendous execution time, making the planners tedious and expensive to use in practice.

Except for the separate disadvantages of the related work, they all have a common drawback: the numerical cost models proposed by the above methods are all based on the execution time of the profiling operator under particular hardware. This kind of approach introduces an expensive preparation effort without optimality guarantees. Besides, it needs hours or even days to find the optimal plan for an

extensive neural network. The related work's methodology starts from the existing expert experience, defines a parallel plan search space, proposes a cost model based on profiling data, and transforms the problem into mathematical modeling. Then they solve this search problem by designing a solver. However, this methodology has the following shortcomings:

- If defining an ample search space, the profiling preparation will be very time-consuming, and the subsequent solver search will take an unacceptable amount of time. If the search space size is restricted manually, there is a risk of missing the actual optimal plan.
- Because these methods reduce the search time by restricting the search space, the search can never find better plans than those defined by experts.
- Profiling is expensive, and AI researchers and engineers without parallel domain knowledge do not know how to perform profiling analysis.
- The search space design is based on expert experience, and the method needs to be redesigned and constructed for new structures of new networks that have not been considered before.

In a word, these methods do not provide a good balance between generality and accuracy in this NP-hard search problem.

In order to circumvent the state-of-the-art method's expensive profiling process and provide an algorithm to give an accurate hybrid parallel plan in a short time, this thesis proposes a symmetric hierarchical abstract machine and a symbolic cost model that decouples the hardware from the parallel algorithm. Based on the BSP model, this approach eliminates the need for profiling each operator on specific hardware. Based on the semantics of computing neural networks, the symbolic cost model can be transformed and reduced. This thesis proposes an algorithm that reduces the complexity of NP-hard search problems to linearity and can generate efficient hybrid parallel algorithms in seconds. The results aim to be integrated into Huawei's MindSpore open-source environment and contribute to Huawei's AI products and solutions to explore the full potential power of its hardware of Ascend chips.

This thesis employs structured parallelism approaches to optimize distributed deep learning so that DL frameworks can automatically handle performance. As a result, DL designers could focus more on developing more precise DNNs without incurring additional costs. To describe the features of a modern AI training machine, HSM2DL created a hierarchical and symmetric abstract machine. HSM2DL also proposed an execution model to characterize the hybrid parallel training process and a symbolic cost model as a metric to evaluate the cost produced by various parallel plans.

The main objective of HSM2DL is the discovery of optimal hybrid parallel plans. Recursive partitioning and the Flex-Edge Recursive (FER) graph are two key contributions to operator-level partitioning. The recursive partitioning partitions the computational graph into two-parts step by step until the graph is partitioned onto the number of devices. This recursive two-part partitioning is based on a symmetric abstract machine feature. Instead of directly traversing all possibilities, the searching complexity is reduced to linear in relation to the number of devices. The FER graph allows the operators to be reordered based on their importance. As a result, backtracking is avoided during the searching process, reducing the searching complexity in relation to the number of operators to linear. As a result, with a linear searching complexity, the algorithm can find a near-optimal operator-level parallel plan. The role and effectiveness of all-reduce overlap techniques in distributed parallelism are discussed as a complement to operator-level parallelism. A method for computing the tail factor is also provided in this thesis. It can be used in conjunction with the D-Rec algorithm for operator-level parallelism. Finally, this thesis presents a joint search method for pipeline plus operator-level hybrid parallelism. Based on the transformer model and the number of 2^p devices, an algorithm for searching for a triple-hybrid parallel plan is provided. In comparison to the SOTA method Alpa [97], the algorithm's efficiency and high plan quality are experimentally demonstrated.

In conclusion, the main contribution of this thesis is to offer a bridging model HSM2DL for distributed deep learning. This model proposed a hierarchical and symmetric abstract machine to simulate the actual training cluster and distributed training's execution model and cost model. Based on this model, this thesis proposes an efficient searching algorithm for both operator-level search and joint search, which combines op-level and graph-level. The advantage of this method is that it nicely balances the quality of the search results and the portability of the search algorithm.

1.2 Overview

This document is organized as follows:

- Chapter 2 first introduces the state-of-the-art of distributed deep learning and automatic parallel plan search. This chapter discussed the background of distributed deep learning, the basics of distributed DNN training, and related works of automatic parallel plan search.
- Chapter 3 first describes the basis of the parallel computing models and compares different representative models, including PRAM, LogP, and BSP. Based on the BSP bridging model, the chapter details HSM2DL. The abstract machine of HSM2DL has two main features: hierarchical and symmetric. Then the execution model of HSM2DL describes the computing procedures of the distributed DNN training. HSM2DL proposed a symbolic cost model that clearly describes HP plans' costs for distributed DNN training. This model provides an excellent opportunity for further analysis and optimizations. This model enables the systematic search for optimal HP of DNNs. The definition of the searching problems is also defined in this chapter. Finally, this chapter ends by giving a comparison between HSM2DL and the related works.
- Chapter 4 introduces the methodology to search for near-optimal operator-level parallel plans. By taking advantage of HSM2DL, symbolic simplification and functional reduction are applied in order to control the searching complexity without losing the optimal results. Based on the hierarchical and symmetric abstract machine, HSM2DL proposed recursive partitioning, which reduces the searching complexity with regard to the number of devices. Besides, inspired by the third homomorphism, HSM2DL proposed a Flex-Edge Recursive graph whose operators can be reordered freely, thus avoiding back-tracking of the searching problem and therefore reducing the searching complexity. The optimality of recursive partitioning and vertices reordering is proved in this chapter. The experiments show that the found parallel plans' quality can match the SOTA plans with a very small search time.
- Chapter 5 describes the parameter synchronization overlap behavior. This chapter details how the synchronization cost is generated and how it can be hidden with the overlap techniques. This chapter discussed the symbolic modeling of the tail factor and its cost model. An algorithm to calculate the tail factors for a given computational graph and a parallel plan are also introduced. Experiments show that with an accurate tail factor, the search result of D-Rec can be improved. The correctness of the tail factor algorithm is also validated.
- Chapter 6 describes a very preliminary attempt at pipeline partitioning search. This attempt is based on two strong assumptions. First, this thesis assumed that the number of devices and the number of stages are both a power of two. Another one is that this thesis only considers the transformer-based DNN model. In this chapter, HSM2DL completes the symbolic cost model of the bubble ratio and proposes a joint-search algorithm that can find the SOTA-matched parallel plans in a very short time. Large cluster experiments show that HSM2DL can outperform SOTA planners by up to 255× when handling large-scale transformer-based models.

1.3 Publications

This section presents the accepted papers which are extended in this thesis:

- H. Wang, C. Li, T. Tachon, H. Wang, S. Yang, S. Robert, S. Limet. Efficient and Systematic Partitioning of Large and Deep Neural Networks for Parallelization. European Conference on Parallel Processing EuroPar 2021, 2021, Lisbon, Portugal. pp.201-216.
- H. Wang. Freeing hybrid distributed ai training configuration. in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 16201624.
- H. Wang, T. Tachon, C. Li, S. Robert, S. Limet. SMSG: profiling-free parallelism modeling for distributed training of DNN. International Journal of Parallel Programming, Springer Verlag, 2022, to appear.

Chapter 2

State Of the Art

2.1 Introduction

The main topic of this thesis is the automatic parallel plan generation for Deep Neural Networks (DNNs). More precisely, it focuses on how to systematically choose an optimal parallel plan to train a DNN model on a specific cluster without user interaction.

This chapter first introduces the background of the distributed training of DNNs in Section 2.2, including the general knowledge of deep learning and its applications. Then this chapter introduces a general example of the DNN training process. The necessity and bottleneck of distributed DNN training are also stressed in this chapter. In Section 2.3, the basics of distributed DNN training are described, including the definition of computational graphs, operators, and tensors. This section also introduces the basic parallelism plans with their costs during the distributed training. Section 2.4 introduces the state of the art of existing automatic plan searching approaches and their limitations. This chapter presents a global view of distributed deep learning and introduces the key points when choosing an optimal hybrid parallel plan

2.2 Background of distributed deep learning

In this section, the background of distributed deep learning will be presented. In Section 2.2.1, the boom of deep learning and its application during the past decades will be introduced. Then the necessity of distributed DNN training is described in Section 2.2.3. Next, in Section 2.3.2, commonly used parallel plans of distributed training, including their advantages and disadvantages as well as their suitable applied domain, are presented.

2.2.1 Deep learning

Artificial intelligence (AI) has been the ultimate dream that humans, especially computer scientists, have wanted to realize for a long time. Since the 21st century, it can be seen artificial intelligence has played a pivotal role in our daily life. Artificial intelligence can easily classify images, accurately realize face recognition in traffic and security, and improve the automation ability in medical image processing. Besides, people enjoy more accurate translation, text comprehension, and automatic article generation. In addition, human beings can have real conversations with robots. Last but not least, AI surpasses the level of professional players in Go [76] and computer games.

The take-off of AI discussed above can be attributed to the in-depth research of deep learning academia and the wide application of deep learning in industry. Currently, deep learning (DL) as a term can be synonymous with deep neural networks (DNN). When it points to a technical field, it refers to a subdomain of machine learning [26].

The neural network is not a new word. It has been more than half a century since McCulloch [51] put forward the mathematical model of M-P neurons in 1943. A neuron is the primary computing unit of a neural network. It receives the output signals from other neurons and performs summation by weight amplification. If the sum exceeds a threshold, the neuron is "activated" and outputs signals. The so-called

neural network is a mathematical system composed of a series of input-output connected neurons. The development of neural networks has experienced many twists and turns. In 1958, Rosenblatt proposed the first generation of neural network single-layer perceptrons [68], which can distinguish basic shapes such as triangles and squares. People began to think that neural networks would be the future of artificial intelligence. In 1969, Minsky [52] proved the limitation of a single-layer perceptron: it could not solve the XOR problem. Enthusiasm for neural networks began to decline until 1986 when Hinton et al. [70] proposed the second generation of neural networks: replacing the original single fixed feature with multiple hidden layers, using the sigmoid function as the activation function, and training the model using a loss backpropagation algorithm. Such a neural network can effectively solve nonlinear classification problems. In 1989, Cybenko and Hornik [31] proved the universal approximation theorem: any function can be approximated by a three-layer neural network with any precision. In the same year, LeCun et al. [40] invented convolutional neural networks to recognize handwriting. These studies have revived researchers' enthusiasm for neural network research. In 1991, the backpropagation algorithm was pointed out to have the problem of gradient disappearing [30]. Since then, the research on artificial intelligence turned to shallow machine learning (e.g., support vector machine [63]), and the research on deep neural networks was shelved. The third wave of research and application of neural networks took off in 2006 when Hinton et al. [29] proposed to use pre-training to train deep belief networks quickly to suppress the problem of gradient disappearance. Deep learning has since shown its powerful effects in various fields. Deep learning was the first to make breakthroughs in speech recognition. Microsoft and Google [28, 15] successively used deep learning to reduce speech recognition error rate from 30% to 20%, the most significant breakthrough in this field in the past ten years. Hinton and his students reduced the top 5 error rate for Imagenet [17] image classification problems from 26% to 15% [38], and deep learning entered an explosion period.

In addition to the key breakthroughs such as backward propagation, which is the gradient disappearance resolution, the development of neural networks and their powerful effects are also because of the increase in the number of their hidden layers. In fact, deep neural networks are profound neural networks with multi-layers. Why are deep neural networks more effective? One may say that a more complicated structure can learn more information and features to produce a more effective neural network. However, a complicated, vast neural network is not as effective as a deep neural network with the same number of neurons in practice.

Actually, deep learning is effective because it relies on a 'representation learning' [7] approach, which is a consensus in the machine learning world. In traditional machine learning, data features need to be manually extracted first. This step is called 'feature engineering.' Then classifier learning step takes place on the data produced by the feature engineering steps. In many research fields, feature engineering is an important step and directly determines the quality of machine learning algorithm output. However, deep learning can be regarded as a black box. When data is input from one end of the black box, a well-trained model can be obtained at the other end. The deep neural network can automatically extract the features used in the process, which is so-called 'representation learning.' For representation learning, the most crucial thing is layer-by-layer processing. For example, when inputting an image, pixels can be seen at the bottom of the neural network, and layer by layer, more and more abstract descriptions, such as edges and contours, are gradually displayed. Although there may not be such precise layering in neural networks, there is a tendency to abstract from the bottom up. It is widely believed that "layer-by-layer processing" is the key to learning and one of the critical factors for the success of deep learning. As a result, deep neural networks can learn a high-level abstract representation of data and automatically extract features from data [39, 24]. In addition, the hidden layers in deep learning are equivalent to a linear combination of input features [6]. The model efficiency of deep learning increases exponentially with its depth [54].

In conclusion, deep learning is a vital research part of artificial intelligence. It solves automation problems in computer vision, natural language processing, recommendation system, and intelligent speech by training a deep neural network. By calculating the hidden layers of deep learning, deep learning can automatically extract features of input data. These features are abstracted as hidden layer parameters and updated. Theoretically, the effect of deep learning becomes better as the scale of network models grows and the number of hidden layers increases. In the last decade, the practices in this field also show an exponentially increasing trend regarding the DNNs' scale and better effect in various application fields.

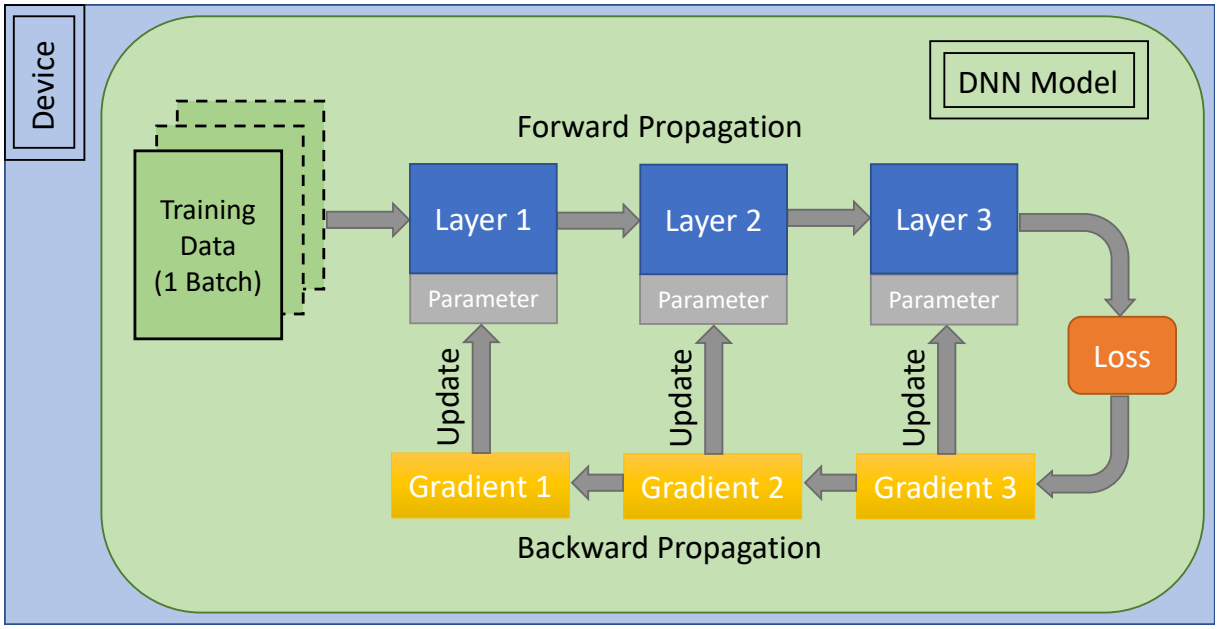


Figure 2.1: DNN Model Training Process

2.2.2 Training a deep neural network

There are various types of neural networks, such as multi-layer perceptrons, auto-encoders, convolutional neural networks, and transformers. These neural networks apply to different application scenarios, including anomaly detection, classification, clustering, dimensionality reduction, etc. For example, convolutional neural networks are more suitable for image processing, and transformers play an important role in natural language processing. The training method of deep neural networks includes supervised, unsupervised, self-supervised learning, reinforcement learning, etc. Supervised learning learns a function that maps an input to an output based on an example of an input-output pair. This function is inferred from the labeled training data consisting of a set of training examples. In supervised learning, each instance is a pair of an input object (usually a vector) and the desired output value (also called a supervised signal). The supervised learning algorithm analyzes the training data and generates an inference function that can be used to map new examples. In contrast, unsupervised models use unlabeled data, and the DNN model must extract features and patterns to understand the data. Self-supervised learning learns from unlabeled sample data. It can be regarded as an intermediate form between supervised and unsupervised learning. The neural network learns in two steps. First, the task is solved based on pseudo-labels which help to initialize the network weights. Second, the actual task is performed with supervised or unsupervised learning.

Although there are ever-changing neural networks, the training process of deep neural networks has common characteristics, which can be summarized as the process shown in Figure 2.1.

A deep neural network is composed of multiple consecutive **hidden layers**. A trained DNN model can provide the expected output based on input data after the computation of the hidden layers. The **weights** and **biases** of each hidden layer are the keys to determining the input-output relationship. The weights and biases are also named the **parameters** of a neural network. Training the whole input dataset one time is called an **epoch**. There are multiple epochs in the neural network training process to obtain the best accuracy. In practice, the neural networks are trained with multiple mini-batches of the dataset. In Figure 2.1, a green rectangle represents a data **mini-batch** (the total dataset is split into three mini-batches in this example).

The training process of the neural network includes **forward propagation** (FPG) and **backward propagation** (BPG). In forward propagation, a mini-batch of data goes through each neural network's hidden layer (blue rectangle in Figure 2.1). Each hidden layer computes its output based on the mini-batch of data and its current parameters. The output of the last layer will be compared with the labeled data.

The quantification of the gap between them is called the **Loss** (orange rectangle in Figure 2.1). Backward propagation computes the gradient (yellow rectangle in Figure 2.1) of each hidden layer based on Loss and then updates local parameters based on the gradient. Backward propagation begins from the last hidden layer and ends in the first hidden layer. The end of the backward propagation also indicates the end of training the current small batch of data. The DNN model computes the next mini-batch and updates the model's parameters until the model can return an accurate enough output.

2.2.3 Distributed deep learning

As discussed in Section 2.2.1, deep learning is a representational learning method where more hidden layers and a higher number of parameters are positively correlated with a more accurate output. In recent years, the blossoming of artificial intelligence is closely related to the exponential upward trend of neural network model size, hidden layer number, parameter number, and data set size. Currently, the largest DNNs have trillion-level parameters. For example, OpenAI GPT-3 [9] has 175 billion parameters, while Google's Gophe [64] has 600 billion parameters.

Two main reasons restrict training such large-scale models: limited memory and extraordinarily long training times. Although the size and scale of DNNs have increased exponentially, the memory growth of training accelerators has been linear: from a few GBs to tens of GBs, such as Nvidia's A100 and H100 with 80 GBs of memory. Limited memory forces the training of such large-scale models to rely on large clusters of multiple devices. On the other hand, DNN training is an iterative process involving hundreds or thousands of epochs. Training a large neural network can take weeks or even months. For example, OpenAI uses 1024 Nvidia V100s to train GPT-3, which takes 34 days. Training GPT-3 costs millions of dollars if renting the equipment from the cloud service provider. Note that it is only the cost of training the DNN model for one time.

Training a good AI model is not easy. In recent years, more researchers have begun to conduct in-depth research on parallelized and distributed deep learning technologies. That is, how to divide training data, allocate training tasks, allocate computing resources, and integrate distributed training results to achieve a perfect balance between training speed and training accuracy.

A. Krizhevsky [38] splits the kernel of the convolution neural network into two groups and uses two GPUs to train AlexNet in parallel, which is an initiative and successful attempt for model parallelism. Another typical early work is DistBelief [16], which employs data parallelism and model parallelism. DistBelief utilizes the parameter server for data parallelism and applies model parallel for each sub-model. In DistBelief's experiments, the authors used up to 144 model chunks for parallel training. In such a distributed model, each worker node aggregates the intermediate data from other nodes to complete its local computation. Then it passes the generated data to the consecutive nodes. The experimental results show that when the model is large, the speedup ratio will increase with the increase of the number of machines, and the best case is using 128 machines to achieve a speedup ratio of 12 times. Moreover, when the model is small, using more than 16 machines to implement model parallelism does not produce speedup and sometimes is even slower than a single machine because of communication. Gpipe [32] pioneered the concept of pipeline parallelism, which divides a neural network into multiple subnets called stages. Each stage alternately trains different micro-batches of data on different devices. Pipeline parallelism has proven to be a successful way to parallelize extensive models (e.g., transformer [85]-based networks) which can be regarded as one kind of model parallelism. Megatron [74] is an expert-designed parallel method proposed by NVIDIA that reasonably mixes pipeline, data, and model parallelism. It divides the model into multiple stages. It takes advantage of the characteristics of Matrix Multiplication in each stage. It proposes a parallel plan for a series of consecutive MatMul: the parallel plan of these operators is along the row dimension and column dimension on the parameter tensor alternatively.

The research in distributed deep learning mainly includes two focuses: data and model partitioning and communication control. These two parts' specific implementation and interrelationships may vary with different algorithms and systems, but they have some common basic principles. A brief introduction of these two parts is given below.

- **Data and model partition** determines a reasonable and efficient parallel plan for a given DNN model and training hardware. A parallel plan signifies how to distribute the data and DNN model onto distributed devices. Different parallelism plans have different features and may be more suitable for different kinds of DNN models. For example, determining how many stages are partitioned

for pipeline parallelism and selecting data parallelism or model parallelism for specific operators. Detailed descriptions and comparisons of different parallel plans are given in Section 2.3.2.

- **Communication control** concerns how to optimize the efficiency of communication caused by partition. After computing nodes finish their local computation, they need to communicate with the other nodes to finish the training process. The implementation of communication determines the efficiency of the system. For data parallelism, at the end of each training epoch, communication is required to make each node synchronize the parameters of the neural network. Common examples include AllReduce-based[72] collective communication and ParameterServer-based communication [16]. For model parallelism, in order to complete the calculation, point-to-point communication is required between different nodes. The choice of an appropriate communication pattern determines the speed of training.

This thesis considers the actual communication implementation but focuses more on the data and model partition module. The objective is to free the users from the heavy tasks of deciding on an efficient data and model partition plan.

2.3 Basics of distributed DNN training

2.3.1 Computational graph, operator and tensor

Now training neural networks in academics and industries usually use modern deep learning frameworks (DL frameworks)[62, 2]in practice. Users can write Python code to define their own DNN models. The DL frameworks process input data and determine training configurations of the DNN models. These deep learning frameworks transform a high-level model declaration into a low-level computational graph that can be trained on accelerators (e.g., GPU, TPU, and NPU). In order to perform deep learning training, a deep learning system still needs to solve many problems:

- how to execute a complex deep learning model efficiently?
- how to identify parameters to be trained in a deep learning model?
- how to calculate the gradient needed to update the model automatically?

Modern deep learning frameworks solve these problems by using computational graph techniques. This section will introduce the basic definitions of distributed training. These definitions include *computational graphs*, which are the abstracted DNN models, *tensors*, which is the basic data structure in DNN training, and *operator*, which is the computation unit in DNN training.

A **tensor** is the basic data structure in the DL frameworks. The definition of tensor in mathematics is based on a generalization of vectors and matrices, covering the concepts of scalars, vectors, and matrices. A scalar can be considered a zero-order tensor, a vector is a first-order tensor, and a familiar RGB color image is a third-order tensor. In the DL framework, a tensor stores data and multiple attributes such as data type, shape, dimension or rank, and gradient transfer status. For example, Table 2.1 lists the main attributes and functions of a tensor. The *dimension* represents the number of tensor axes. The tensor's *shape* is an important property that records the length of each axis, which is the number of elements per dimension of the tensor. Tensors can typically hold boolean, floating-point, integer, complex, and string data. Each tensor has a unique data type, represented as *dtype*. During calculation, the framework checks the types of all tensors involved in the operation. If the types do not match, an error will be reported. Some special calculations must use the specified data type. For example, logical operations must be of the Boolean type. In DL frameworks, an attribute of a tensor may indicate a *device* location where the tensor is stored, for example, in a CPU or a GPU. A storage state of tensor data may be classified into variable and immutable. An immutable tensor is generally used for data initialized by a user or input by a network model. A variable tensor stores the network weight parameters and updates its own data according to the gradient information.

A scalar is a zero-order tensor that contains a single value but no axis information. A vector is a first-order tensor with an axis. A second-order tensor has two axes, which is known as a matrix. Usually, tensors are "regular" with the same number of elements on each axis, like a 'matrix' or 'cube.' Special types of tensors, such as irregular and sparse tensors, are also used in particular environments. This

Attributes	Functions
shape	Length of each dimension of the storage tensor, for example, [3,3,3]
dimension	Indicates the number of tensor dimensions. The scalar value is 0, the vector value is 1, and the matrix value is 2.
dtype	Indicates the data type of the storage, such as bool, int8, int16, float32, and float64.
device	When creating a tensor, you can specify the storage device location, such as CPU and GPU.
name	Identifier of tensor

Table 2.1: Tensor attributes

thesis will concentrate on the regular tensors because the majority of important DNN models only treat regular tensors.

An **operator** is the primary computing unit of computational graphs. Based on their functionality, operators can be classified into tensor operations, neural network operations, and control flow operations. Tensor operations include tensor structure operations and mathematical tensor operations: tensor creation, index slicing, dimension transformation, and split are structure operations, while mathematical operations include scalar operations, vector operations, and matrix operations. Scalar operators feature element-by-element operations on tensors. The vector operator computes only on a specific axis, mapping a vector to a scalar or another vector. Matrix operation includes matrix multiplication, matrix normalization, matrix determinant, matrix eigenvalue, matrix decomposition, etc. Neural network operations include feature extraction, activation function, loss function, optimization algorithm, etc. Feature extraction is a standard operation in deep learning. Its critical point is to extract more representative tensors than the original input. For example, a standard convolution operation (Conv) is a feature extraction operator. An activation function is responsible for mapping the input of the neural network layer to the output end. The activation function is responsible for increasing the nonlinearity of the neural network model. Common activation functions include an S-shaped growth curve (Sigmoid), a linear correction unit (Rectified Linear Unit, ReLU), etc. The loss function estimates the degree of inconsistency between the predicted value and the actual value of a model. Optimization algorithms use different strategies to update parameter weights to minimize the loss function based on gradients. Standard optimization algorithms include Stochastic Gradient Descent (SGD), Adaptive Moment Estimation (Adam), etc. Data flow operations include operators related to data preprocessing and data loading. Data preprocessing operators are mainly used to cut, fill, normalize, and enhance image and text data. Data loading usually performs shuffle, batch, and pre-loading operations on data sets. Finally, the control flow operation controls the data flow direction in the calculation diagram. Control flows are required when representing flexible and complex models like the Mixture of Expert (MoE). Use frequently used control flow operators, conditional operators, and loop operators. The control flow operation affects the forwarding operation's data flow direction and the reverse gradient operation's data flow direction.

A **computational graph** represents the forward and backward propagation of the DNN model in the back-end of the DL frameworks, describing the computation logic and state of DNN models during training. The computational graph consists of a fundamental data structure: *tensor* and a basic operation unit: *operator*. In computational graphs, nodes are usually used to represent operators, and the directed lines between nodes represent dataflow directions and describe the dependency between computations. The data flow is updated by performing forward, and backward gradient computation based on the data flow direction and operator in the graph to update the tensor status in the graph, thereby training the model.

Computational graphs also implement the strategy of distributed parallel training. A computational graph can be split into multiple sub-graphs to implement graph-level model parallelism, for example, pipeline parallelism. The tangent sub-graph can be realized by appointing the sub-graph number to which each operator belongs. In addition, an operator-level parallelism plan may be specified by assigning either data parallelism or model parallelism to each operator in the computation graph. The following section describes the segmentation dimensions of different parallel strategies and the distributed training process through examples. For clarity, these examples are based on Figure 2.1, and the computational graphs are partitioned into different dimensions of the hidden layer. Although the examples are demonstrated with layers, the reader can understand that the hidden layers are equivalent to operators. Actually, a hidden

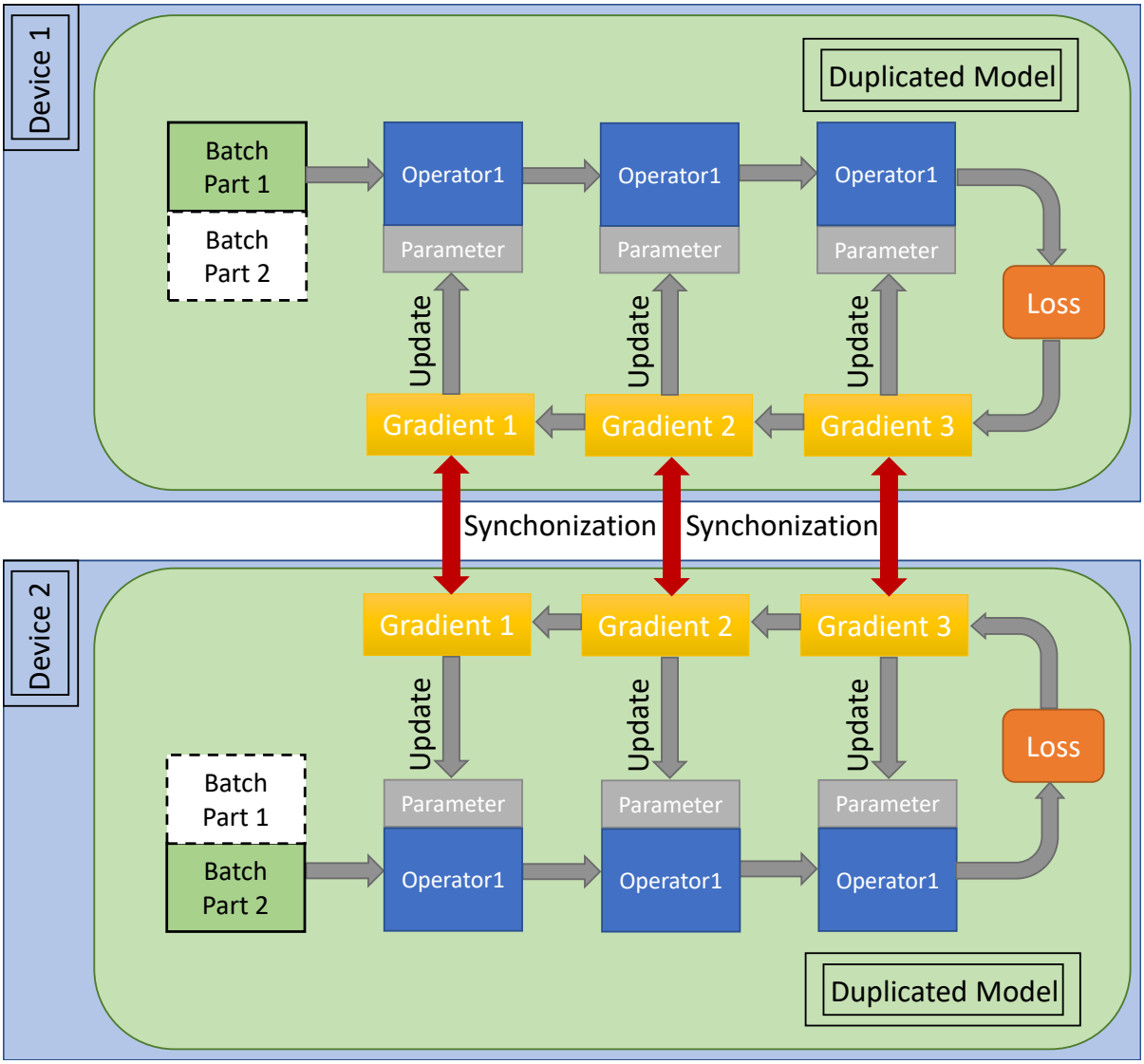


Figure 2.2: Data Parallelism

layer is usually composed of a set of operators.

2.3.2 Standard parallelism plans

The necessity of distributed deep learning has been discussed in Section 2.2.3. In this section, different parallelism plans to realize distributed training are discussed.

The emergence of large machine learning models has brought about a rapid increase in the demand for computing power and memory, giving rise to distributed training. Distributed training systems significantly improve computational performance allowing us to both reduce the training time and increase the size of neural network models while using the limited memory of computing devices. The target of developing *parallelism plan* is to correctly allocate models that exceed the device's memory to distributed clusters to train efficiently without compromising training accuracy.

Standard distributed parallel plans can be divided into *operator-level* and *graph-level* parallel plans. In deep learning, most of the tensors are vectors of data. The dimension of this vector is called the batch dimension of the tensor. At the operator level, the tensor may be partitioned - either in the batch dimension, called *data parallelism* - or in the other dimensions, called *operator-level model parallelism*. Another type of partition plan is graph-level parallel partitioning, where the computational graph is partitioned into multiple sub-graphs. Graph-level parallel partitioning is also a kind of *model parallelism*

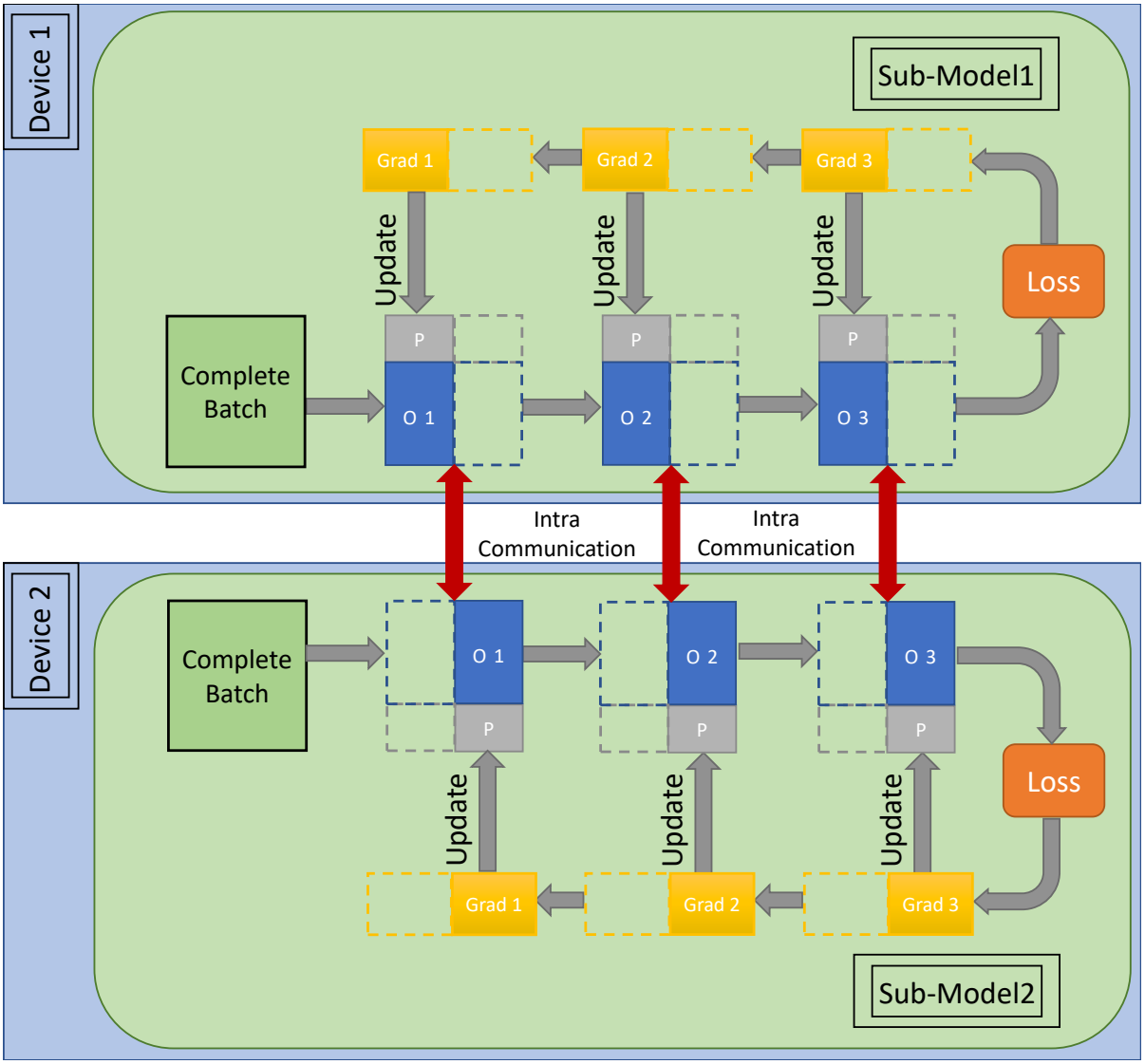


Figure 2.3: Operator-level model parallelism

(graph-level).

Data parallelism (DP) is a simple, efficient parallelism plan that is often used to address the lack of computing power of a single node. Common DL frameworks have specific implementations of data parallelism, including TensorFlow Distributed Strategy [2], PyTorch Distributed [62], Horovod Distributed Optimizer [72], etc. Data parallelism distributes a user-given batch size N equally across M devices, with each device being allocated N/M training samples. An identical duplicate of the computational graph is shared on each device, which independently trains its own N/M samples and computes the gradient. Different devices will compute the gradient G_i based on the local N/M training samples. To ensure consistency of the training program parameters, the local gradients G_i need to be aggregated to calculate the average gradient $(\sum_{i=1}^{N} G_i)/N$. Eventually, the training procedure uses the average gradient to update the model parameters and complete the training for the N samples.

Figure 2.2 shows an example of data parallelism consisting of 2 devices. Each device is allocated half of the training samples but has the same neural network configurations (duplicated model). The local training samples are sequentially passed through the operators in this computational graph duplication, completing forward and backward propagation. During the backward propagation process, each computational graph duplication generates the local gradients (Gradient1, 2, 3). The corresponding local gradients on different devices (e.g., Gradient1 on each of Device1 and Device2) are aggregated to cal-

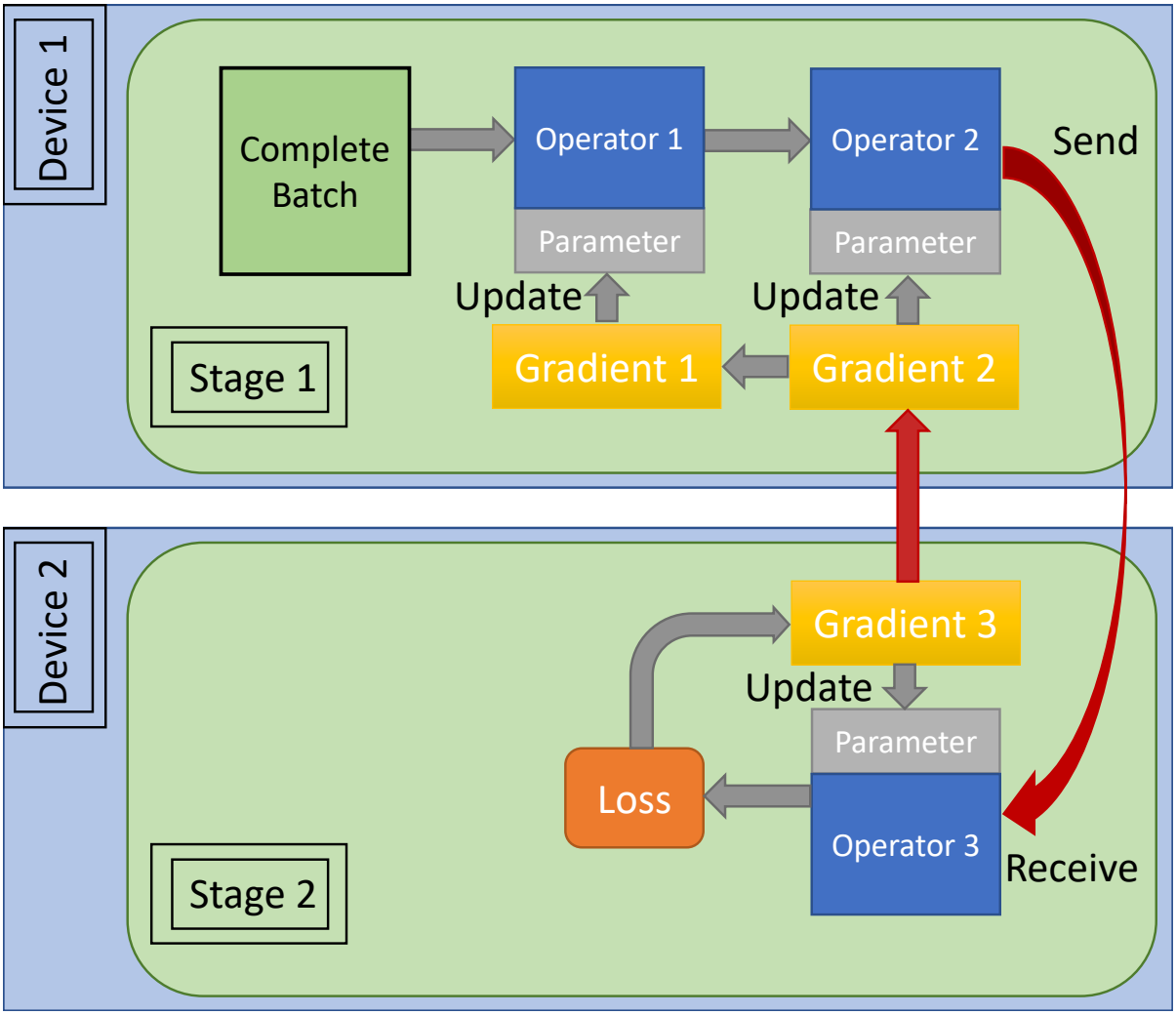
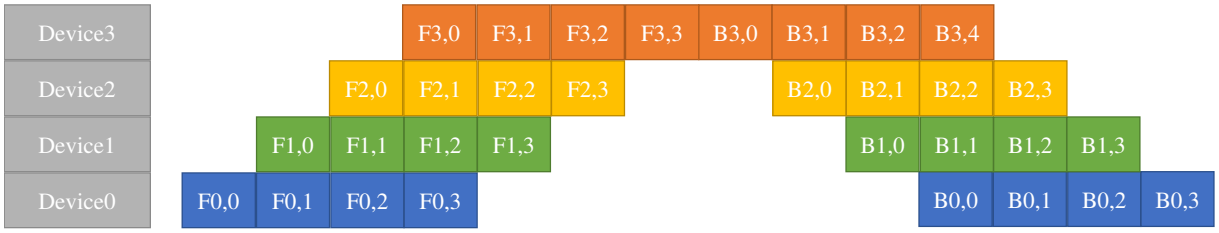


Figure 2.4: Model (graph-level) Parallelism

culate the average gradient. The aggregation process is usually realized through All-Reduce collective communication and is also referred to as *parameter synchronization*.

Operator-level model parallelism (OMP) is often used to solve the memory shortage problem for single nodes [38]. The reason for applying OMP is that data parallelism can significantly improve the training speed but is neither efficient nor solves the problem of insufficient memory when the number of model parameters is large. A common under-memory scenario is when a model contains large operators, such as a Fully Connected Layer in a deep neural network, which must compute many classifications. The memory required to complete such large operators' computations may exceed a single device's memory capacity. Then this large operator needs to be partitioned. Suppose this operator has P parameters and is to be distributed on N devices. The P parameters can be evenly partitioned among the N devices (P/N parameters per device) so that each device is responsible for less computation and can perform the required computation in forward and backward propagation within the memory capacity limit.

Figure 2.3 gives an example of OMP implemented by two devices. In this example, assuming a neural network with three operators, the memory required to store their parameters is 40GB, and the memory needed to perform forward and backward propagation calculations are 10GB, i.e., a total of 50GB of memory is required. Assuming a single computing device has 32GB of memory, a single card cannot complete the training of this neural network. Data parallelism between the two devices would also require storing an entire computational graph duplication on each device, i.e., each device would need the same 40GB to store the parameters, so data parallelism cannot solve the memory shortage problem. The OMP divides the parameters of the three operators equally between the two devices, requiring only 20GB



$F_{i,j}$ represents the j_{th} micro-batch of the i_{th} stage of the forward propagation
$B_{i,j}$ represents the j_{th} micro-batch of the i_{th} stage of the backward propagation

Figure 2.5: Pipeline execution process

of stored parameters per device plus the 10GB needed for computation, which is sufficient for training the network. As shown in the Figure 2.3, OMP requires inter-device communication to obtain the missing half of the parameter information to complete the operators' computation when performing forward and backward propagation. OMP does not require parameter synchronization because each sub-model trains with the entire batch data.

Graph-level pipeline parallelism (PP) is a vital parallelism plan in natural language processing neural networks which contains large-scale parameters. The way to partition the computational graph of pipeline parallelism is also model parallelism. Unlike OMP, pipeline parallelism partitions the computational graph into multiple sub-graphs with a subset of the operators (also called *stage*) distributed on each device and is called graph-level model parallelism.

An example of graph-level parallelism implemented by two devices is given in Figure 2.4. In this example, assuming a neural network with three operators, operator1 and operator2 require 10GB of memory each to complete the computation while operator3 need 20GB, the model requires a total of 40GB of memory. Each device can only provide 32G of memory. In this example, the user can place operator1, operator2 on device1, and operator3 on device2. In forward propagation, the output of operator2 is sent to device2 downstream. Device2 receives the data from upstream and completes the forward computation of operator3. In backward propagation, device2 sends the backward computation of operator3 to device1. device1 completes the backward computation of operators1, operator2 and then finishes this training.

However, during the execution of graph-level parallelism, the device assigned to the operators positioned in the back part of the computation graph must be continuously idle for a long time, waiting for the previous operators to complete the computation, as shown in the figure above. The idle time dramatically reduces the average usage of the devices. This phenomenon is known as the Bubble.

GPipe[32] proposes to divide a data *mini-batch* into multiple pipeline *micro-batches*. Suppose a data mini-batch has D training data, and this mini-batch can be divided into M micro-batches, then the micro-batch size is D/M. Each micro-batch enters the training system accordingly and completes forward and backward propagation to compute the gradient. The gradients corresponding to each micro-batch will be cached, and when all micro-batches are completed, the cached gradients will be summed up to calculate the average gradient and update the model parameters.

Figure 2.5 further gives an example of pipelined parallel execution. In this example, the model parameters need to be partitioned into four devices for storage. The mini-batch is cut into four micro-batches to fully use these four devices. After device0 completes the forward propagation of the first micro-batch (denoted as F0,0), it sends the intermediate results to device1, triggering the forward propagation task (denoted as F1,0) in response. At the same time, device0 can also start the forward propagation task for the second micro-batch (denoted as F0,1). The forward propagation is completed at the last device3 in the pipeline. The system then starts backward propagation. Device3 starts the backpropagation task for the 1st micro-batch (denoted as B3,0). After this task is completed, the intermediate result is sent to device2, triggering the backward propagation task (denoted as B2,0) in response. At the same time, device3 caches the gradient corresponding to the 1st micro-batch and starts the 2nd micro-batch calculation next (denoted as B3,1). When device3 has finished all the backward propagation computations, he will sum the locally cached gradients and divide them by the number of micro-batches to calculate the average gradient, which is used to update the model parameters. The *bubble* in the

pipeline parallelism is the critical element of pipeline parallelism. When the device finishes forward propagation, it must wait until complete backward propagation begins, during this time the device will be idled. In Figure 2.5, it can be seen that device 1 has a long much time to start the second backward propagation task after completing the second forward propagation task. In order to reduce the waiting time, a common practice is to increase the number of micro-batches so that the backward propagation starts as early as possible. However, a too-small micro-batch size may cause the devices to be underutilized. The optimal micro-batch size is, therefore, a compromise between several factors. The core factors are the pipeline bubble's size and the devices' computational power.

2.3.3 Parallel cost of standard plans

The previous section described the training process of standard parallelism plans and briefly introduced their characteristics and application scenarios. This section analyzes the costs generated by different parallelism plans during training to explain why they apply to different scenarios. It can also be used as a guide to quantify costs in subsequent chapters.

The cost discussed here mainly refers to the extra cost incurred by distributed training compared to training on a single node, such as communication due to data partitioning or idle time due to data dependencies. Common costs like operator computation to both stand-alone and parallel training, are not discussed.

In data parallelism, each device holds the complete model. Each sub-dataset is calculated independently on each device when performing forward and backward computation, and there is no extra cost in this computation process. Each device independently computes the gradients with a different sub-dataset. It is necessary to exchange the gradients and parameters on different devices through inter-device communication to calculate their average value (the average of gradients is the same gradient as training on a single node), as shown in the red arrow in Figure 2.2. This exchange of the gradients is named "parameter synchronization". Two well-known implementations of parameter synchronization are Parameter Server and AllReduce collective communication. The computation overlap technique can hide the cost generated by collective communication to improve efficiency. Thanks to the overlap technique, data parallelism is a very efficient distributed training mode. Unfortunately, data parallelism is insufficient to deal with large-scale neural networks with a large memory footprint since each device must implement the entire model.

For OMP, because each device is trained on the complete data set, the parameters on each device are inherently updated. However, as shown in Figure 2.3, each operator and its parameter are partitioned into different devices, thus each operator must communicate internally (the communication happens inside the operator computation, distinguish from tensor redistribution or parameter synchronization) to complete the forward and backward computations. Different operator-level model partition plans incur different communication costs, which are large compared to the parameter synchronization of data parallelism after being overlapped. So, choosing the partition plan with the minimum communication cost is the primary concern of OMP.

At the graph level operator partitioning, as shown in Figure 2.4, for forward propagation, the output of opeartor2 on device1 needs to be transmitted to opeartor3 on device2 via point-to-point communication *Send, Receive*, and symmetrically, the reverse gradient of opeartor3 on device2 needs to be communicated to operator2 on device1. This communication cost is generally small or even negligible compared to the data-parallel, model-parallel communication cost associated with the number of parameters. However, because the computation of operator3 depends on the output of operator2, device2 remains idle until the computation of operator2 on device1 is completed. Slicing the batch into a micro-batch and performing pipeline training (as shown in Figure 2.5) can reduce the waiting bubble time, but the bubble time cannot be completely removed, so the main cost of operator slicing at the graph level is the idle time.

2.3.4 Hybrid parallelism plans

All the standard parallel approaches help speed up the training of large-scale neural networks, but the optimal results cannot be achieved by applying only a few of them. When training large AI models, users often face both insufficient computing power and insufficient memory. Therefore, they need to use a mixture of data parallelism and model parallelism. This approach is called hybrid parallelism.

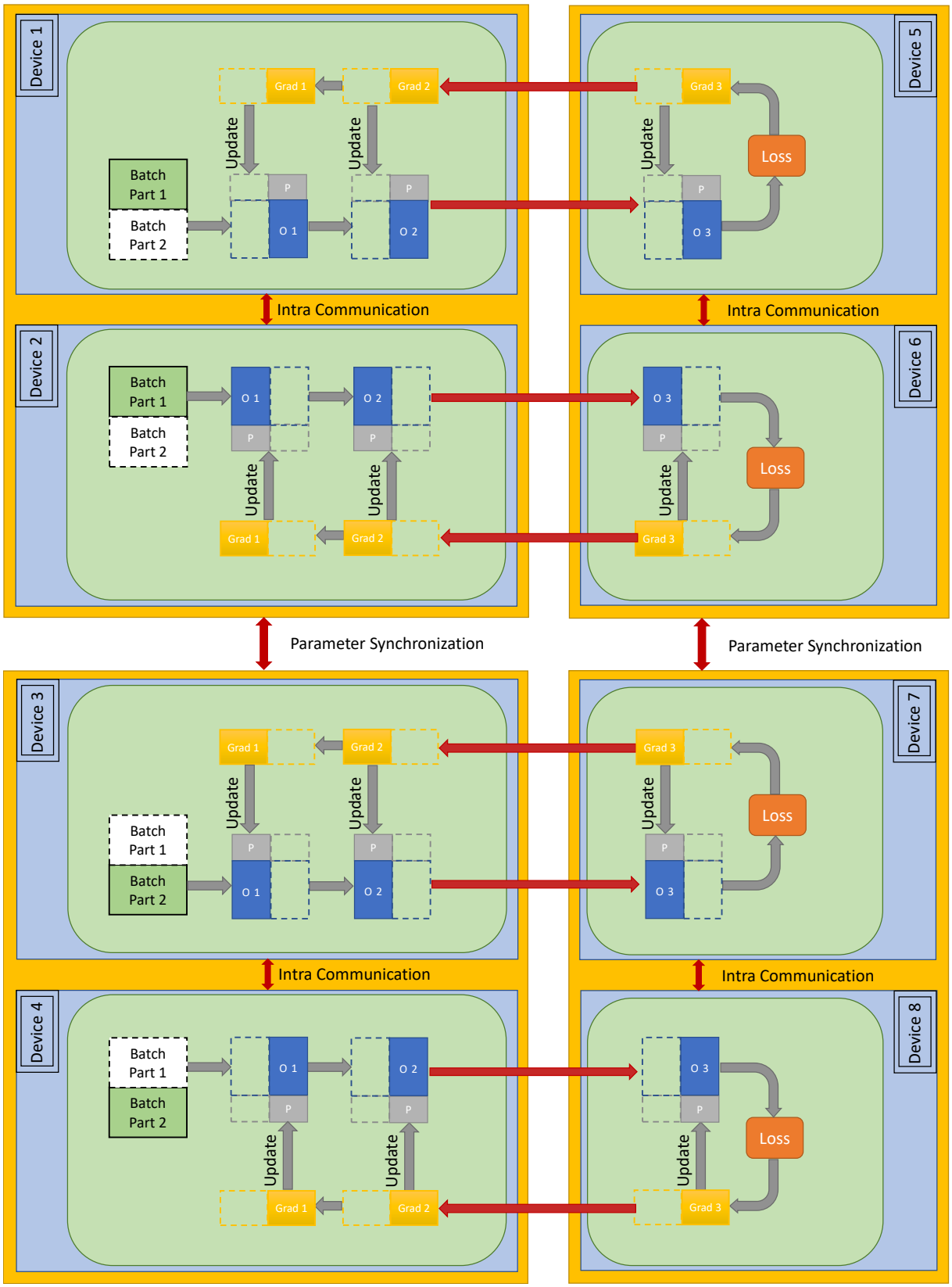


Figure 2.6: Model (graph-level) parallelism Training Process

Representative previous work on hybrid parallelism includes OWT (One Weird Trick) [37], Megatron-LM [74] and 3D-parallelism [67].

OWT applies data parallelism for the convolution layers and model parallelism for the last fully connected layer because it found that the two parallel plans are respectively suitable for the two layers.

Developed by NVIDIA, Megatron-LM[74] is a large, powerful transformer with a mixture of DP, OMP, and PP. It requires a user-given number of degrees for the three parallel plans and partitions the DNN model according to the parallel plan degrees. Degrees denote the number of partitions of DP, OMP, and PP. For example, to partition a transformer onto eight devices with the degrees of DP, OMP, and PP all equal to 2 means that the computational graph is partitioned into two pipeline stages respectively trained on four devices, and every four devices are partitioned into two parts along DP dimension once and MP dimension once. It can also be extended and favors the other transformer-based DNN models like BERT [18], GPT [10], T5 [65]. 3D-parallelism from DeepSpeed also combines the data parallel, model parallel (operator-level), and pipeline parallel simultaneously. It assumes three different axes *x, y, z* which respectively represent the three parallel plans and analyze their trade-offs on the axes.

Figure 2.6 provides an example of hybrid parallelism implemented on eight devices. The computational graph in this example consists of three operators. The eight devices are first divided into two device groups: devices1-4 and devices5-8. The two groups launched the graph-level pipeline parallelism, with stage1 containing operator1,2 assigned to the first group of devices and stage2 containing operator3 assigned to the second group of devices. Devices1-4 first complete the forward propagation of stage1 and pass it to devices 5-8. Devices 5-8 complete the forward and backward propagation of stage2 on the second group of devices and pass the backward propagation result back to the first group of devices to complete the backward propagation of the stage1. Each group of four devices also applies data parallelism and OMP. Devices1-2 and 3-4 are grouped to perform data parallelism. Within the device group of devices1-2, the two devices perform OMP via intra-communication to perform OMP to complete the computation of operator1,2.

2.3.5 Challenges of training with hybrid parallel plans

Hybrid parallelism combining data and model parallelism is necessary to accelerate the training of giant DNN models. However, acquiring highly distributed training benefits faces two significant problems: system implementation and parallel strategy selection.

System implementation:

Because of the lack of available automatic differentiation modules [5] and auxiliary development APIs, early research in deep neural networks required researchers to write their computational processes from scratch. Researchers had to write parallel code manually if they wanted to perform distributed parallel computation on the networks [37].

Later, auxiliary frameworks such as Caffe [34], Chainer [82], and Theano [4] emerged, allowing the users to design their neural networks by means of building blocks that relieve them from the burden of writing neural network code. However, these frameworks do not support distributed parallel computing well and lack support for the accelerators such as GPUs.

TensorFlow [2] and PyTorch [62] support custom hybrid parallelism by explicitly assigning operators to specific devices. Recent works have proposed different high-level primitives based on existing frameworks for manual parallel configuration to facilitate the implementation of hybrid parallelism. With the help of these high-level primitives, the users only need to specify the parallel plans but do not need to consider the code generation, scheduling, mapping, etc., to realize distributed training. Mesh-Tensorflow [73] implements a domain-specific language for specifying the parallelism of giant DNN models in Tensorflow. However, this approach is intrusive and requires rewriting the entire model code using this language. Moreover, it has some limitations. E.g., it does not allow applying different partitioning strategies to the same tensor in different layers. DeepSpeed [66] provides simple APIs in Pytorch to configure and initiate hybrid parallelism. However, it requires the user to modify the model to a sequential structure to achieve pipeline parallelism and relies on external libraries such as Megatron-LM [74] or custom implementations to support tensor model parallelism. Such programming work is still too complex for AI researchers.

Artificial intelligence researchers and engineers are now writing and training their neural network models with the latest AI frameworks [2, 62, 1], which have their built-in machine learning compilers

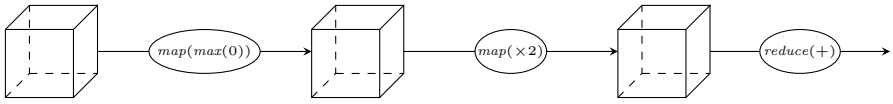


Figure 2.7: Minimal neural network example

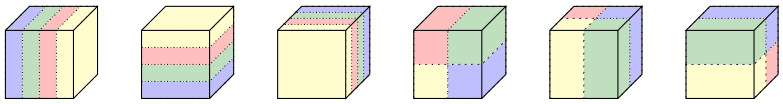


Figure 2.8: Possible distribution of a 3D tensor over 4 devices

such as XLA, ONNX, TVM, etc [45]. These compilers can automatically convert user-defined neural network programs in Python into equivalent computational graphs, which can then be efficiently executed on accelerated devices such as GPUs, Ascend, etc. The compiler applies multiple intermediate passes to transform the computational graph before executing the training on the accelerator. Typical examples of the passes include an automatic differentiation stage that automatically generates the operators and edges needed to back-propagate the computational gradient; a computational graph simplification stage that does not affect the computational result: graph pruning. These passes are applied sequentially by the compiler and are orthogonal to each other, allowing the developer to extend the other stages to achieve the desired functionality.

The GSPMD [91], Mindspore [1], PaddlePaddle [49], and Oneflow [92] all provide parallel processing stages integrated into their compilers and can be called transparently in the compiler without affecting the execution of the DNN model. These stages are currently capable of automatically partitioning the computational graph according to a user-defined plan for each operator in Python, inserting the required communication operators, and smoothing the sliced computational graph to the device cluster. These stages are also being developed in an iterative phase to support more distributed operators, more parallel paradigms, and the ability to search for parallel plans automatically.

Parallel strategy selection:

Deciding a suitable hybrid parallelism strategy is difficult for the following reasons:

- Different types of operators may prefer different parallelism strategies. For example, it is not suitable to partition the kernel tensor of a Conv op because the shape of the kernel is usually $3 * 3$ or $5 * 5$. A common practice for Conv op is to partition it along its *batch dimension* (data parallelism) or *channel dimension* (one of the possible operator parallelism strategies) [25].

- Besides, the shape of the operators also affects the strategy choices [12]. A MatMul op is more suitable for data parallelism when the shape of the parameter tensor is small and should be configured as operator parallelism when the shape of the parameter tensor is very large.

- The operators are not executed separately. They are all connected in the computational graph via the edges. The output tensor of an operator is also the input tensor of its successive operator. If the parallel plans are different for these two operators, the tensor needs to be redistributed in the cluster, which generates additional cost and should also be taken into consideration.

Possibilities of hybrid parallel plans For operator-level, to split one tensor with x dimensions into 2^p parts, it is equivalent to an unordered sampling with replacement, so there are C^p_{x+p-1} possibilities. Then to distribute a computational graph with n operators into 2^p devices where each op includes two x dimensions tensors. There are $(C^p_{x+p-1}{}^2)^n$ possibilities. Graph-level pipeline partitioning is also an NP-hard problem. If searching graph-level and operator-level parallel plans together, it is a mixture of two NP-hard problems. Therefore, parallelizing a simple NN to only a few accelerators leads to already near uncountable choices.

The complexity of the operator-level problem is illustrated through the following minimal example. Consider a toy neural network represented in Figure 2.7 of 3 operators on 3-dimensional tensors that aimed to be distributed over four devices. Operators may be, for example, a RELU followed by an element-wise twofold increase followed by the sum of all elements. Possible distributions of a 3D tensor over four devices are all illustrated in Figure 2.8, and the number is 6. As there are three operators, the total number of possibilities for this whole tiny graph is $6^3 = 216$.

2.4 Automatic parallel plan search

2.4.1 Introduction of automatic parallel plan search

Manual parallelization becomes highly infeasible treating huge DNN models. First, parallelizing a model just for a specific hardware platform (e.g., NVIDIA DGX-2) requires a considerable engineering effort because there are diverse hardware environments like cloud platforms, data centers, and local machines. In addition, manually considering all possible parallelization opportunities for a giant model is increasingly difficult. Even strategies designed by experts for small-scale models are considered sub-optimal because they do not make good use of some of the parallelization opportunities in the model [35]. For large-scale neural networks, optimal performance is usually obtained by applying hybrid parallelism, but it is not always possible to apply any hybrid parallelism strategy to obtain better performance than the standard strategy. In fact, in some cases, the distributed training time may even become longer if the hybrid parallelism strategy is not appropriately configured, incurring a higher communication cost or large waiting time. Choosing a sensible, efficient hybrid parallelism strategy is therefore crucial to the effectiveness of distributed parallelism. However, this is not an easy task, and developing a reasonable parallel strategy requires researchers to understand both the semantic nature of neural network operator computation and the structure of neural networks. In this context, researchers have to master both neural networks and parallel computing, which is seldom the case.

Based on the exponential complexity of hybrid plans and the inadequacy of manual expert-defined policies, automated search for parallel plans has become an important area of research.

Automatic parallel plan search algorithms can be divided into two categories. One is based on machine learning, especially reinforcement learning methods, such as ColocRL [53], REGAL [61], Auto-MAP [88], etc. They have the same disadvantages of requiring large amounts of training data, long time, poor interpretability, and poor scalability, so they will not be discussed in this thesis. The following three sections present representative related work on operator-level search (DP+OMP), graph-level search (DP+PP), and three-mix joint search (DP+PP+OMP), respectively.

2.4.2 Problem Positioning

When talking about the "automatic parallelism" of compiling DNN models, people may think of machine learning model compilers, such as XLA and TVM [45], which are designed for optimizing the computation on a single accelerator. Another possible thinking of automatic parallelism is a system like GSPMD[91], which enables parallelism in XLA with user-defined parallel strategies for operators. Such a system fails to achieve full automation, especially for users without parallel computing knowledge.

Orthogonal to the above two kinds of "automatic parallelism" optimizations, this thesis focuses on the scope of generating parallel plans automatically. A distributed DNN parallel planner/generator is able to eliminate the time and labor-consuming process of determining the parallel plans. Output can be given to the above two approaches to achieve further optimizations. In addition, the planner should be able to be implemented in one of the state-of-art Deep Learning frameworks [2, 62, 1] to allow AI researchers and engineers to write neural network code without the need for additional parallel code. After generating the parallel plan, the DL frameworks will take charge of the rest part of distributed training, including the code generation and run-time execution. The positioning of the planner/generator is shown in Figure 2.9.

2.4.3 Related works

Operator-level plan search methods

- **OptCNN** OptCNN [36] provides an automatic parallel plan searching solution for layer-level (a more coarse-grain operator-level, a layer is normally composed of a series of operators) parallel

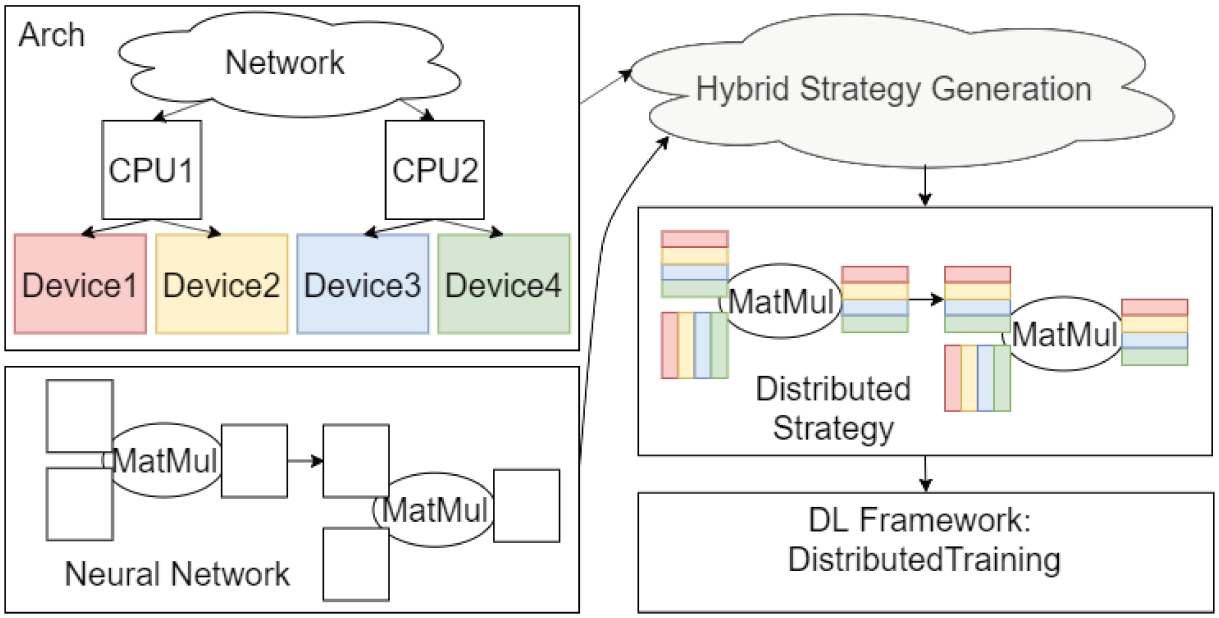


Figure 2.9: The positioning of automatic parallel plan generation

plans for convolution neural networks. OptCNN considers the majority of the possible dimensions for partitioning, including batch size, width, height, and the number of channels for a convolution layer. All these dimensions can be partitioned and distributed into distributed cluster devices. OptCNN constructs a computational graph $\mathcal{G}$ to represent the model and a device graph $\mathcal{D}$ for the cluster. It also builds a cost model to estimate the cost under different plans. OptCNN proposes a dynamic programming graph search algorithm based on node/edge elimination, and it can determine the combination of partition dimensions for each layer.

- **ToFu** OptCNN's dynamic programming algorithm can only solve linear graph plans[1]. ToFu [87] proposes a graph coarsening method, which can fuse the graph's nonlinear parts (such as the back-propagation operator) into other corresponding operators to form a group. Finally, the dynamic programming algorithm of OptCNN is applied to the linear group for policy searching. ToFu uses a recursive algorithm to speed up the searching on top of the dynamic programming algorithm. For the cost model, ToFu only considers the communication cost to reduce the evaluation difficulty. Compared with OptCNN, which spends 8 hours searching for an 8-machine parallel plan of WResNet-152, ToFu's experiments show that it only takes 8.3 seconds to search for a parallel plan for the same DNN model.

- **Flexflow** Based on the setting of OptCNN, FlexFlow [35] proposes four possible operator-level cut dimensions, namely *sample, operation, attribute, and parameter*, denoted SOAP. In SOAP, the sample dimension partitioning is data parallelism, and the other dimensions are model parallelism (operator-level). The attribute dimension refers to each dimension of the input tensor. FlexFlow can handle non-linear computational graphs that OptCNN cannot take using a stochastic Monte Carlo algorithm, which defines a suitable parallel plan for each operator in the computational graph. The accuracy of the Monte Carlo algorithm depends on the size of the search space listed randomly, which makes its time on large-scale model policy search unacceptable. It took 37 minutes to search for a plan for the NMT model [89] on 16 servers equipped with 4 P100 GPUs.

- **TensorOpt** Unlike ToFu, TensorOpt [11] does not have a graph coarsening operation but extends the dynamic planning algorithm of OptCNN and names it FTElimination (Frontier Tracking Elimination), which solves the problem that OptCNN cannot handle non-linear graphs. This FTElimination

[1]ToFu defines a graph G is linear if it is homeomorphic to a chain graph G', meaning there exists a graph isomorphism from some subdivision of G to some subdivision of G [8]. Note that a fork-join style graph is linear by this definition.

is not very efficient and can incur high search times. Therefore, TensorOpt also tries to group operators and apply the FT-LDP (Frontier Tracking Linear Dynamic Programming) algorithm to help reduce the time complexity. In addition, the FT-LDP algorithm can be further enhanced by multi-threaded parallelism to improve search efficiency. FT-LDP with multiple threads applied can find the best strategy for WResnet [94] in 22 minutes, while FT-Elegation takes 5.5 hours. Because ToFu's plan does not make full use of the device's memory, TensorOpt could find a higher quality parallel plan (with much higher throughput than ToFu).

Graph-level plan search methods

- **PipeDream** PipeDream [56] provides an automatic parallel solution supporting asynchronous pipeline training with data and pipeline parallelism. PipeDream first profiles the model to be partitioned, obtaining the runtime, size of activations, and model parameter size for each layer. Then, based on the profiling results, they create a Profiling-based cost model and design a dynamic planning algorithm. The algorithm prioritizes load balancing, partitions the pipeline stages, and determines the DP degree for each stage. PipeDream-2BW [57] is an extension of PipeDream. It optimizes memory consumption by applying activation recomputation and reduces the number of parameter weight buffers that store different versions of the computed gradients. PipeDream-2BW groups the repeated structures (e.g., transformer) of the model and only considers configurations where all stages of the model are replicated the same number of times. Another shortcoming of PipeDream is that only linear graph structures are supported.

- **DNN-partitioning** To solve the problem of PipeDream only supporting linear graphs, Fiddle's researchers proposed DNN-partitioning [81]. DNN-partitioning extends the dynamic scheduling algorithm in PipeDream and proposes an integer-based scheduling solution to solve the nonlinear graph search problem.

- **DAPPLE** PipeDream-based networks are based on asynchronous computing. Asynchronous computing may not guarantee the accuracy of training and prediction, although some measures are taken to minimize the effects of asynchrony. However, in large-scale industrial production, accuracy is an important metric, so this thesis only considers the synchronous case. DAPPLE [20] is a practical example of synchronous data and pipeline hybrid parallelism. It proposes a new parallelized policy planner to solve the partitioning and placement problem and explores the best hybrid strategy for data and pipeline parallelism. DAPPLE also proposes a new runtime scheduling algorithm to reduce device memory usage. This scheduling method is orthogonal to the recomputation approach and does not come at the expense of training.

Joint search methods

- **DistIR** DistIR [71] is an efficient intermediate representation for parallel training neural networks, used to describe distributed DNN computation explicitly. It defines feasible cuts, including data parallelism, pipeline parallelism, and limited model parallelism (operator-level) partition dimension. It uses a linear regression model to model the operator's cost for different model sizes to ease the profiling effort. To finite cut-off possibilities, DistIR uses a simple grid search to find the minimum cost strategy. Although DistIR is very effective in finding the best parallel plan in its search space, as the search space is so limited, its optimal strategy may be too coarse to use compared to other solutions.

- **Piper** Piper [80] uses a two-level dynamic programming approach to search for DP, MP, and PP parallel plans. The external dynamic programming algorithm deals with pipeline parallel plans, and the internal dynamic programming searches for operator-level plans by storing data profiled in advance into a lookup table. External dynamic programming generates hundreds of NP-hard knapsack sub-problems when computing the minimum throughput of a sub-graph for a given configuration. Piper uses a bang-per-buck heuristic to speed up the process of solving the generated backpack sub-problems, reducing the computational complexity. The internal dynamic programming of Piper takes 2 hours to generate a partitioning plan for training 64 layers of BERT on 2048 devices. However, the current implementation of the algorithm is sequential and inefficient, and the heuristic algorithm has no plan optimality guarantees. A potential advantage of Piper is that

some of the processes in Piper can be executed in parallel, which can linearly scale the runtime of the algorithm on a multi-core CPU server and further reduce the search time.

- **Alpa** Alpa [97] uses a two-level hierarchical algorithm to search for parallel plans and supports DP, MP, and PP searching together for arbitrary directed acyclic graphs (DAGs). A formal integer linear programming method solves the operator parallelism problem in the two-level hierarchy. The inter-operator pipeline parallel partition problem is solved by a dynamic programming algorithm based on TeraPipe[46], and additionally considers how device mesh is partitioned. During the execution of the dynamic programming algorithm, Alpa invokes ILP to search for the best operator-level parallel strategy for each subgraph. The operators are also clustered using another dynamic planning algorithm. Alpa was able to find the best strategy for GPT-39B on 64 GPU devices in 40 minutes. However, searching for a strategy for GPT-175B[9] on 2048 GPU devices could take thousands of hours, indicating poor scalability given the complexity and cost of this approach.

2.4.4 Limitations of existing approaches

To find the optimal hybrid parallel strategy, the above existing approaches are all following this methodology:

1) Defining the problem and search space;

2) Creating a cost model to predict the execution time;

3) Profiling the execution time of different parallel plans on the specific hardware

4) Searching the results with an algorithm based on 2) and 3);

5) Outputting the optimal solution.

The only difference is that they differ in the search spaces, cost models, and search algorithms. The quality of the outputted parallel plan of step 5) depends on the joint effect of steps 2), 3), and 4). These approaches construct a generic cost model to estimate the actual execution time of the distributed training process as detailed as possible. Step 3) profiles the time of a specific parallel plan on a given hardware platform for a DNN model (or for an operator). These approaches invoke the time obtained by step 3) in the cost model of step 2). Step 4) will select the optimal strategy with the lowest estimated time.

Example of existing approaches' cost model

Let us take OptCNN[36] as an example for operator-level plan search. All the existing solutions are extensions of this simple example.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a computational graph. Vertices set $\mathcal{V}$ is a set of operators. The directed edges set $\mathcal{E}$ represents the dataflow direction between operators. In the following denote $e_{ij} \in \mathcal{E}$, the edge between two connected vertices v_i and v_j. $\mathcal{P}$ is a hybrid parallelism plan of the computational graph $\mathcal{G}$, which is the set of operator-level strategies for operators.

The following gives an equation of the estimated training time T for the graph $\mathcal{G}$, the training hardware is represented as $\mathcal{D}$ and the strategy is:

$$T(\mathcal{G}, \mathcal{P}, \mathcal{D}) = \sum_{v_i \in \mathcal{V}} (t_e(v_i, p_{v_i}, \mathcal{D}) + t_s(v_i, p_{v_i}, \mathcal{D})) + \sum_{e_{ij} \in \mathcal{E}} t_r(e_{ij}, p_{v_i}, p_{v_j}, \mathcal{D}) \tag{2.1}$$

- $t_e(v_i, p_{v_i}, \mathcal{D})$ is the execution time to process the operator v_i under strategy p_{v_i} on hardware $\mathcal{D}$. It comprises the local computation time and the communication time caused by parallel computing. t_e is obtained through profiling the operator's execution v_i with its parallel plan under hardware environment $\mathcal{D}$. It is an average time for several experiments. $t_e(v_i, p_{v_i}, \mathcal{D})$ is the execution time to process the operator v_i under strategy p_{v_i} on hardware $\mathcal{D}$. It comprises the local computation time and the communication time caused by parallel computing. t_e is obtained through profiling the operator's execution v_i with its parallel plan under hardware environment $\mathcal{D}$. It is an average time for several experiments.

- t_s denotes the parameter synchronization time at the end of each iteration. It happens after the BPG. Parameter updating is usually implemented by collective communication like all-reduce and can be optimized with communication/computation overlap techniques. t_s requires the same profiling procedure as t_e.
- t_r represents the cost of the same tensor in two connected operators (the output tensor of the first operator is the input tensor of the second operator) are partitioned along different dimensions. This tensor needs to be redistributed. The redistribution cost t_r cost is usually estimated by the multiplication of the data size and the known bandwidth.

Limitations

The above methodology seems understandable and practical: it only requires building an accurate cost model and performing a profiling task to search for the optimal strategy. Experiments show that optimal hybrid strategies of specific DNN models can be found in this way.

Though promising, the following facts limit the generality of these approaches:

- For one operator, all the dimensions of its tensors are splittable. The possible partition dimensions for a DNN model increase exponentially with regard to the number of devices. This fact caused high searching complexity for step 4) and an unacceptable profiling task for step 3) for large-scale training (e.g., on 256-2048 devices cluster).
- The profiling results of an operator are heavily dependent on the hardware. Re-profiling is needed when the DNN model and training accelerators change. If the unacceptable profiling task (e.g., more than 24 hours for 64 devices [97]) is repeated at every training attempt, it is a disaster for the AI researchers.
- It is hard for these approaches to model the new enhancement techniques like communication and computation overlap. Based on the profiling result without modeling, such kind of optimization may lead to sub-optimal results.

These methods can obtain good results on small-scale models thanks to the accuracy of the cost model. But this genuine cost model depends on the precise profiling task. When the search space and the profiling task become large, it is not easy to balance accuracy and generality with this methodology. To make these methods more general, either the accuracy of the strategy is pursued, which brings in NP-hard complexity and unacceptable profiling efforts, or the search space and the cost model are simplified, with simulated value instead of profiling. If this methodology aims to guarantee the accuracy of the strategy, it will face an NP-hard complexity and unacceptable profiling work to the search. But to finish the plan search within acceptable time limits, it needs to reduce the search space, abstract the cost model, and add simulation value instead of profiling. For example, [97] reduces its searching space by limiting the operator-level hybrid parallel plan to a mixture of only two 2-parts dimensions (e.g., MatMul has three possible partition dimensions and Conv has 7) which significantly reduces the searching space but may miss the optimal results. It announced that the profiling task for a 64 accelerator cluster for a GPT-like model is more than 24 hours. To reduce that, it applies a simulation function to estimate the execution time instead of profiling all the cases. These two examples help reduce the profiling time and searching complexity but introduce inaccuracies.

Due to the strict limitation caused by the pre-defined cost model and combined profiling task, the existing approaches lack portability and generality. Building a concrete and accurate cost model is simple, but it throws the complex problem into the hands of the search algorithm and profiling work. The increased number of devices results in a polynomial increase in the complexity of the search and the amount of profiling work. Not to mention, profiling is not reusable; changing devices and networks requires re-profiling. As a result, a more general method for new DNNs with different environments is demanded.

2.5 Conclusion

This chapter first introduced the background of distributed deep learning, starting with the rise and development of deep learning, and gave a basic definition and characteristics of deep neural networks. This

chapter then represents the basic definitions of distributed parallelism, including computational graphs, operators, tensors, etc. In addition, several standard parallel training plans are also described, including data parallelism, OMP, and graph-level model parallelism (pipeline parallelism). Manually designing parallel plans is labor-intensive and challenging, so automatic parallel plan search methods have become a hot research topic in academia. This chapter introduces representative operator-level, graph-level, and joint plan search schemes in recent years and explains the limitations of the related approaches. Based on the above descriptions, this thesis proposes an automatic policy generation method based on a parallel computing model, which will be described in the subsequent chapters.

Chapter 3

Parallel computing model for distributed deep learning

3.1 Introduction

In Chapter 2, representative automatic parallel plan search approaches in the scientific community are presented, and the shortcomings of these profiling-based methods are presented in Section 2.4.4. Their main limitation is that they don't create an analyzable model for the parallel plan search problem. The cost model is unsplittable, which mixes the parallel details with the hardware characteristics. Therefore, the variables in the cost model can only be obtained through profiling the high-complexity possibilities on real devices, which invokes unacceptable searching time. These approaches may miss the optimal parallel plan by rule-based reducing the search complexity.

In order to overcome the shortcomings of these methods, this thesis aims to propose a method that can efficiently generate hybrid parallel plans without the need for expensive profiling preparation. The main idea is to create a specific parallel computing model to describe the parallel machine and distributed training details. The parallel computing model proposed in this thesis is based on the well-known model in the field of parallel computing: Bulk Synchronous Parallel (BSP) model [83]. BSP models the distributed training process and the training cluster individually, thus avoiding the profiling process by decoupling the hardware and software. The model also creates a symbolic cost model as an evaluation criterion for the parallel plan decision. According to the abstract machine and AI domain-specific knowledge, this symbolic cost model will first be transformed and reduced. The profiling values will be substituted in the reduced-cost model, simplifying the profiling task and increasing search complexity. The main advantage of this model is that it offers a flexible model that can balance the search complexity and accuracy well.

his chapter is organized as follows: First, the background of the parallel computing model is introduced and followed by the typical computing models, including PRAM, BSP, and LogP model. Next, the abstract machine of this thesis is introduced for DNN distributed training, whose structure is hierarchical and symmetric for modern AI training clusters. The execution model built for the distributed training process and the underlying cost model are presented in Section 3.4.2. Finally, Section 3.4.3 introduces a basic cost model for distributed deep learning.

3.2 Basics of parallel computing model

A *computing model* helps algorithms to be designed, analyzed, and efficiently compiled in a high-level language and finally implemented in hardware. The von Neumann model [69] is a serial computing model that effectively separates the work of the hardware designer from that of the software engineer. The hardware designer can design a variety of von Neumann machines without regard to the software being executed, and the software engineer can write a variety of programs that can be executed efficiently on the von Neumann model without regard to the hardware being used.

A *parallel computing model* usually refers to the abstraction and formalization of the analysis of parallel computing. Under the conception of a parallel computing model, the *abstract machine* describes the essential characteristics of various parallel computers (or at least a particular category of parallel

computers); the *execution model* describes the design and behaviors of parallel algorithms in the model; the *cost model* helps to evaluate the performance of the parallel computing model. In a broader sense, a parallel computing model provides an 'interface' for hardware and software in parallel computing. This interface is actually an abstraction that hides the complicating details but just keeps the main characteristics. Under the convention of this 'interface,' parallel system hardware designers and software designers can develop mechanisms to support parallelism and thus improve the system's performance.

However, there is no parallel computing model as widely accepted and used as von Neumann's machine for parallel computers. The popular parallel computing models are either too simple and abstract or too specialized, and none of them can effectively promote the development of parallel computing. Therefore, either a more practical and general parallel computing model that can reflect modern parallel computers' performance can be developed, or a precise parallel computing model for a specific application domain can be created.

The roles of parallel computing models are:

- **The fundamental of the conception of the parallel algorithms** Usually, a parallel algorithm designer can conceive many different algorithms for the same problem to fit the problem's solution on different models and analyze and evaluate the merits of parallel algorithms. The design and analysis of parallel algorithms depend on parallel computing models.
- **Provide a simple and convenient framework for analyzing parallel algorithms** The parallel computing model abstracts the basic features of a class of parallel computers, avoiding the limitation of too many tedious details of hardware structures, ensuring its generality in a considerable range while reflecting the main features of different algorithms, providing inspiration, guidance and evaluation basis for the design of algorithms.
- **Enabling the design of parallel algorithms with both flexibility and accuracy** Computing model allows parallel algorithm designers to avoid a wide variety of specific parallel computing structures. On the one hand, this allows the developer to concentrate on developing the parallelism inherent in the application problem itself and analyzing the algorithm's performance; on the other hand, the designed algorithm has generality, thus making the research of parallel algorithms a relatively independent activity.

As shown in the Figure 3.1, the relationship between the parallel computing model and the other models in the parallel system is as follows.

- *Machine model*: the lowest level of the parallel model, including the hardware and operating system description.
- *Architecture model*: describes the interconnection network and its role and the form of communication completion, but not the implementation details. It also defines whether the computer is synchronous or asynchronous, SIMD or MIMD architecture [22], or other architectural features. It is a higher level of abstraction than the machine model.
- *Computing model*: describes the cost and resources of the abstract machine for designing and analyzing algorithms and predicting algorithm performance; it is a higher level of abstraction.
- *Programming model*: the highest level of abstraction, describing computation in terms of the semantics of some programming language. For example, in the deep learning domain, the DL frameworks [2, 1] offer the domain-specific language based on python which allows the users to create their DNN model easily.

The purpose of building a computational model is to be able to decouple the parallel hardware from the distributed algorithms. It allows the designer to analyze the parallel hardware and distributed algorithms 'independently'. Here, 'independently' means the parallel experts can analyze the performance algorithm based on a good abstraction of the hardware. The advantages of this 'independent' analysis are that it is more general because it is not only for specific hardware and it is easier to analyze because there is no need to have the profiling value of all the cases. The guidelines to be followed in establishing the parallel computing model: (1) It should be simple and easy to use for users. (2) It can correctly reflect architectural characteristics.

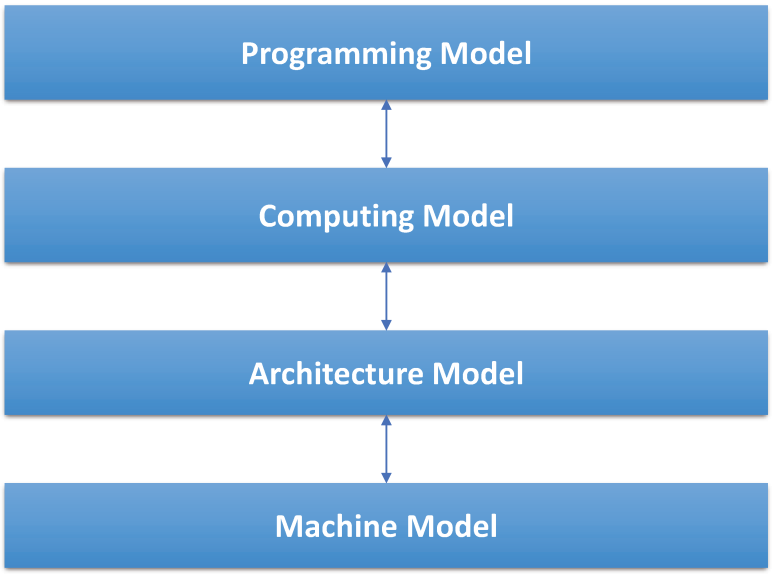


Figure 3.1: Compositions of parallel computing model

A variety of parallel computing models have been proposed in academia, but there is still no universal parallel computing model because of a wide variety of parallel computer architectures. The ideal parallel computing model [78] should have the following characteristics: (1) easy to understand and program; (2) independent from hardware architecture (e.g., DNN training accelerators); (3) reliable performance; and (4) predictable cost. It can also be summarized as simple to use and better reflect the characteristics of architecture. It is important to note that there are opposites and contradictions between the above evaluation criteria, and the design and use of parallel computing models require a trade-off between these characteristics.

- *Independent hardware architecture*: Due to the rapid development of processor and interconnection network technologies, computer system architectures have only a short life cycle. Parallel computing models should be able to abstract parallel computers of different architectures and isolate parallel algorithm design from the update changes of the underlying parallel computer. The more abstract the model is, the stronger the architecture's independence, leading to better algorithm portability.

- *Easy to understand and program*: Parallel computing models must be easy to learn and understand. Otherwise, it is difficult for software developers to understand and apply the model. The programming details are hidden as much as possible: thread decomposition, thread mapping on processors, inter-thread communication, and inter-thread synchronization.

- *Reliable performance*: Models should be able to have performance guarantees on a variety of parallel system architectures. Since system performance degradation is usually due to communication congestion, the communication needs to be reduced proportionally to the parallel computer. In addition, the computation is required to be as regular as possible to ensure scalability.

- *Predictable cost*: The performance of a computational model should be predictable and close to the true performance of the implementation on a concrete architecture. The cost of an algorithm consists mainly of the algorithm's execution time, processor utilization, and the cost of software development. The costs are combinable. The total cost can be calculated from its partial cost.

Parallel Algorithm	A parallel algorithm is a collection of multiple processes that can be executed simultaneously, which interact and coordinate their work to achieve the solution of a given problem. From different perspectives, parallel algorithms can be classified into different categories: numerical and non-numerical parallel algorithms; synchronous, asynchronous and distributed parallel algorithms; shared-storage and distributed-storage parallel algorithms; deterministic and stochastic parallel algorithms, etc.
Numerical Computing	It is a class of numerical computation problems such as matrix computation, polynomial valuation, and solving systems of linear equations based on algebraic relational operations. The algorithm for solving numerical computation problems is called *Numerical Algorithm*.
Non-numerical Computing	It is a class of computational problems based on comparative relational operations, such as symbolic processing problems like sorting, selection, searching, and matching. The algorithm for solving non-numerical computational problems is called a *Non-numerical Algorithm*.
Synchronized Algorithm	Synchronized Algorithm is a class of algorithms in which the execution of individual algorithm processes must wait for each other.
Asynchronized Algorithm	Asynchronized Algorithm is a class of algorithms in which the execution of each algorithm process does not have to wait for each other.
Communication	Communication is the spatial exchange of data between multiple concurrently executing processes.
Synchronization	Synchronization is the temporal forcing of a group of executing processes to wait for each other at a certain point. During the asynchronous execution of processes in a parallel algorithm, the programmer needs to set the synchronization point at the right place in the algorithm to ensure the processors' correct working order and access to the shared resources (mutually exclusive access to resources). Synchronization can be achieved by software, hardware, or firmware methods.

Table 3.1: Some fundamental concepts of parallel programming

3.3 Typical parallel computing models

Some essential and typical parallel computing models include the PRAM (Parallel Random Access-Machine) model [90], the BSP (Bulk Synchronous Paralle1) model [83], and the LogP model [14]. A significant question that needs to be answered through their development is how to accurately reflect the communication overheads specific to parallel machines. These models are generally used for homogeneous systems. The computational model of the recent development of heterogeneous systems, e.g., grid systems, is generally based on the extension of the BSP model.

3.3.1 Parallel random access machine (PRAM)

The most influential early computing model is the PRAM model. Because it ignores synchronization and communication overheads, the PRAM model is a theoretical model that cannot be used to model memory hierarchies or message-passing systems and is difficult to apply in real-world scenarios. PRAM is a natural extension of the sequential von Neumann memory program model, consisting of several processors with local and shared memory with unlimited capacity, which is controlled by a standard clock and operates synchronously. This is shown in Figure 3.2.

In the PRAM model, it is assumed that there exists a shared memory with infinite capacity and a finite or infinite number of processors with the same function, and they all have simple arithmetic operations and logical evaluation functions. The processors can interact with each other at any moment through the shared memory unit. The PRAM model can be divided into the following categories according to the restrictions on the simultaneous reading and writing of the shared memory unit by the processors:

- *PRAM-EREW*: Exclusive-Read and Exclusive-Write PRAM

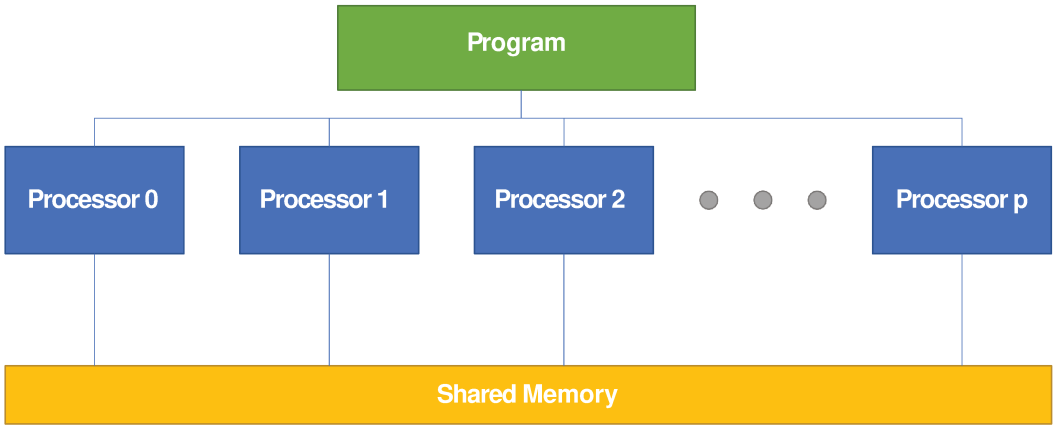


Figure 3.2: Parallel random access machine (PRAM)

- *PRAM-CREW*: Concurrent-Read and Exclusive-Write PRAM
- *PRAM-CRCW*: Concurrent-Read and Concurrent-Write PRAM

Obviously, it was not practical to allow simultaneous writing, so further conventions were made to the PRAM-CRCW model as follows:

- *CPRAM-CRCW*: Only allow all processors to write the same number at the same time, which is called common PRAM-CRCW
- *PPRAM-CRCW*: Only allow the highest priority processor to write first, which is called Priority PRAM-CRCW
- *APRAM-CRCW*: Allow any processor to write freely, which is called arbitrary PRAM-CRCW
- *SPRAM-CRCW*: The actual content written to the memory is the sum of the numbers written by all processors, which is called the summation (Sum) PRAM-CRCW

Advantages of the PRAM model

The PRAM model is particularly suitable for representing, analyzing, and comparing parallel algorithms. Extra considerations (e.g., synchronization and communication) can be added depending on the requirements.

Disadvantages of the PRAM model

- Global shared memory is used in the PRAM model, and the local memory capacity is too small to describe the performance bottleneck of a multiprocessor with distributed main memory. Also, the assumption of shared single memory is unsuitable for MIMD [22] machines with distributed storage structures.
- The PRAM model is synchronous, meaning all instructions operate in a lock-step manner. Although the user does not feel the presence of synchronization, the presence of synchronization is indeed time-consuming and does not reflect the asynchronous nature of many systems in reality.
- The PRAM model assumes that each processor can access any unit of shared memory per unit of time and therefore requires latency-free infinite bandwidth and overhead-free inter-processor

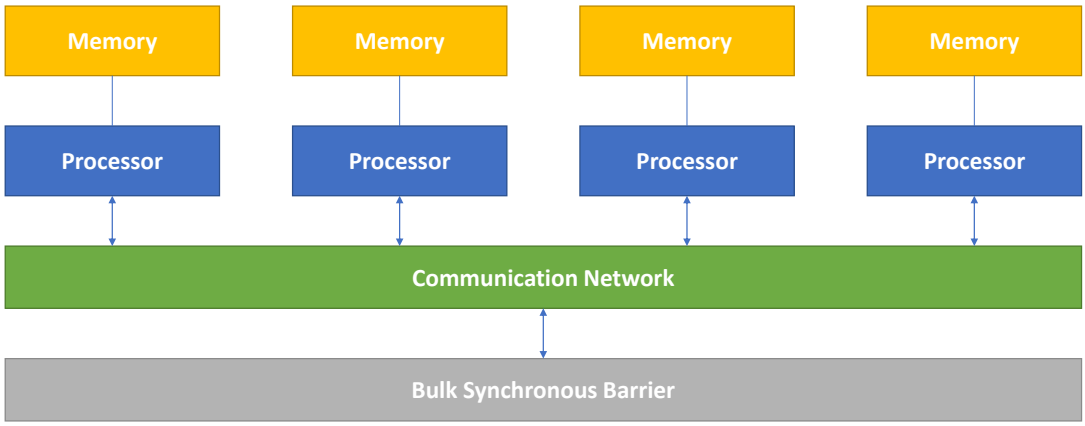


Figure 3.3: Bulk synchronous parallel machine (BSP)

communication. It is unrealistic to assume that each processor can access any memory cell per unit of time while omitting practical, reasonable details such as resource contention and limited bandwidth.

- The PRAM model assumes an infinite number of processors and no overhead for increasing the number of parallel tasks.

- It fails to describe the threaded and pipeline prefetching techniques, the two most common techniques used in parallel architectures today.

Generalizations of the PRAM model

As the understanding of the PRAM model deepened, some generalizations were made so that it could be closer to the actual parallel computers. The main ones are

- The memory contention model divides memory into modules, each of which can handle one access at a time so that memory contention can be handled at the storage module level.

- The latency model takes into account the communication delay between the moment when information is generated and the moment when it can be made available.

- The local PRAM model considers communication bandwidth and assumes that each processor has unlimited local memory and that accessing global memory is relatively expensive.

- The hierarchical storage model treats memory as hierarchical storage modules, each characterized by size and transfer time, with multiprocessors organized into a module tree and individual processors as leaf nodes of the tree.

- The asynchronous PRAM model, where there is no uniform global clock among the processors.

Although PRAM is now a very impractical model for parallel computing, it has been widely accepted and used in algorithmic analysis. Thanks to the PRAM model, the classical parallel algorithms are defined. The following are very successful BSP use cases

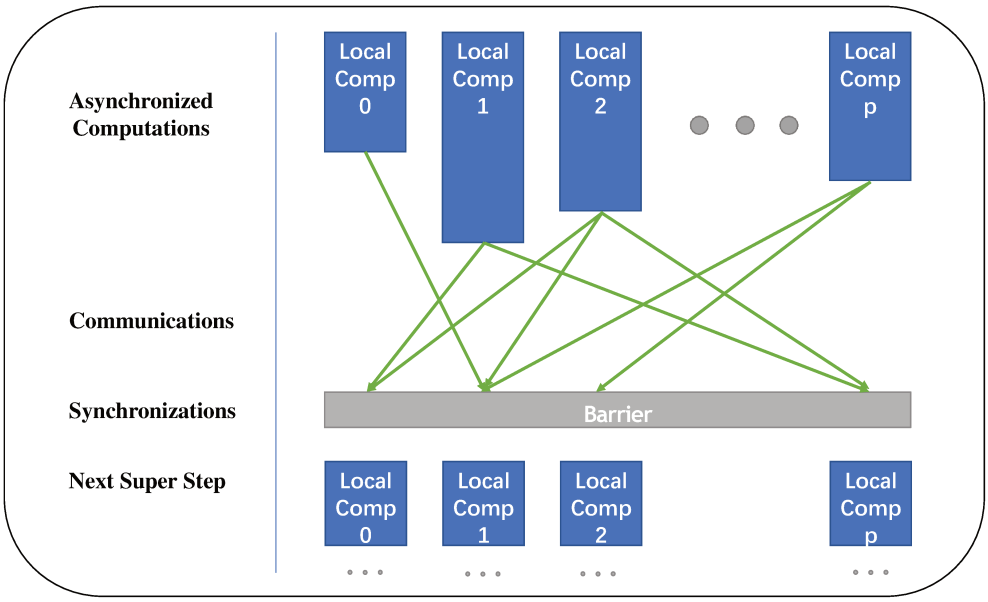


Figure 3.4: A SuperStep of BSP

3.3.2 Bulk synchronous parallel model (BSP)

Basic parameters in the BSP model The BSP model was proposed by Leslie Valiant [83]. This model serves as a bridge between the computer language and the architecture and is a multi-computer distributed storage model described by the following three parameters.

- *Processor/memory modules* The number of processor/memory units is denoted by p in the BSP model.
- *Routers communication modules* Routers are used for the point-to-point delivery of messages between processors/memory modules, denoted by g in the model as the router throughput rate (also called bandwidth factor).
- *Barrier synchronizer modules* The modules perform barrier synchronization with a time interval, which denotes the time interval between global synchronizations.

Procedures in the BSP model

The following figure Figure 3.4 can represent the procedures in the BSP model. In the BSP model, the computation consists of a series of a period separated by a global synchronization, called *SuperStep*. In each SuperStep, each processor performs a local computation and receives and sends messages through a router; then, a global synchronization is made to determine if all processors have completed the SuperStep; if so, it continues to the next SuperStep. Otherwise, the synchronization barriers are assigned to the uncompleted SuperStep.

Cost analysis in the BSP model

The cost model of the BSP can be abstracted in one superstep of the BSP as follows:

$$Cost = max_{all\ process}(w_i) + max(h_i \times g) + L$$

where w_i is the local computation time of a process, h_i is the maximum number of communication packets sent or received by the process, g is the inverse of the bandwidth (time step/communication

packet), and L is the barrier synchronization time (note that in the BSP cost model, the I/O time is not taken into account). Therefore, in the BSP calculation, if s supersteps are used, the total running time is

$$T_{BSP} = \sum_{i=0}^{s-1} w_i + g \times \sum_{i=0}^{s-1} + s \times L$$

This performance equation is simple and convenient for algorithm and program analysis.

Evaluations of the BSP model

- In parallel computing, Valiant also tries to build a bridge between software and hardware similar to a von Neumann machine. It argues that the BSP model can serve such a purpose, which is why the BSP model is often called the bridge model.
- In general, the MIMD distributed storage model [22] is difficult to program. Still, the BSP model presents significant advantages in programmability if computation and communication can be suitably balanced (e.g., $g = 1$).
- Some important algorithms (such as matrix multiplication, parallel preorder operations, FFT, and sorting) have been directly implemented on the BSP model, and they all avoid the additional overhead of automatic storage management.
- The BSP model can be effectively implemented on hypercube networks and optical cross-switch interconnect technologies, showing that the model is independent of the particular technology implementation, as long as the routers have a special communication throughput rate.
- In the BSP model, the length of the superstep must be able to accommodate arbitrary h-relation, which is the least preferred adequate.
- In the BSP model, a message sent at the beginning of a superstep can only be used at the next superstep, even if the network delay is shorter than the length of the superstep.
- the assumption of global barrier synchronization in the BSP model is supported with special hardware, which may not have corresponding hardware in many parallel machines.
- The programming simulation environment proposed by Valiant may not have a small constant during algorithm simulation. This constant may be extensive if inter-process switching is considered (which may involve setting registers and possibly some cache).

  Based on all the advantages discussed above, the BSP model is considered the most promising model for parallel computing BSPLib [27], Apache Hama [75], and Pregel [50] from Google.

3.3.3 LogP model (LogP)

The LogP model was proposed by Culler et al. in 1993 [14]. It considers the shared storage paradigm and the message passing paradigm as consisting of multiple processors with local memories connected through a network. In the case of shared storage, messages are generated by accessing non-local memory. This model has had a significant impact, and many parallel algorithms have been designed using it.

According to the trends of technological development, one of the mainstays of parallel computer development in the late 1990s and in the future is the massively parallel machines, or MPCs (Massively Parallel Computers), which consist of thousands of powerful processor/memory nodes connected through interconnected networks with limited bandwidth and considerable latency. Therefore, we should model parallel computing with this situation in mind so that the model-based parallel algorithms can run efficiently on existing and future parallel computers. According to the existing programming experience, existing programming approaches such as shared storage, message passing, and data parallelism are popular. However, there is not yet an accepted and dominant programming approach, so a computing model independent of the above programming approaches should be sought. Furthermore, according to the existing theoretical models, the shared-storage PRAM model and the SIMD model [22] of interconnected networks are not yet appropriate for developing parallel algorithms. Because they neither include the

case of distributed storage nor take into account practical factors such as communication and synchronization, thus they also do not accurately reflect the behavior of algorithms running on real parallel computers. Therefore, in 1993, D. Culler et al. proposed a multi-computer model with peer-to-peer communication based on analyzing the characteristics of distributed storage computers. It fully illustrates the performance characteristics of interconnection networks without involving specific network structures and assumes that algorithms must be described in terms of realistic message-passing operations.

The LogP model is a multiprocessor model for distributed storage, and point-to-point communication, in which four main parameters describe the communication network.

- L *(Latency)* denotes the upper limit of the waiting or delay time required for a message (one or several words) to be communicated between the source processor and the destination processor.
- o *(Overhead)* indicates the time overhead (including operating system core overhead and network software overhead) for the processor to prepare to send or receive each message. During this time, the processor cannot perform other operations.
- g *(Gap)* The minimum time interval that a processor sends or receives messages twice in a row, the reciprocal of the microprocessor communication bandwidth.
- P *(Processor)* processor/memory module number

Assuming that one cycle completes a local operation and is defined as a unit of time, then L, o, and g can all be expressed as integer multiples of processor cycles.

Features of the LogP model

- Captures the performance bottleneck between the network and the processors. g reflects the communication bandwidth, with at most L/g messages per unit time capable of inter-processor transmission.
- The processors are asynchronous, and synchronization is accomplished by inter-processor messaging.
- Some reflection of multi-threading technology. Each physical processor can emulate multiple virtual processors (VPs), and computation does not terminate when a VP has an access request. Still, the number of VPs is limited by the communication bandwidth and the overhead of context switching. The network capacity limits VPs, and there are at most L/g VPs.
- The message delay is uncertain, but the delay is not greater than L. The waiting time experienced by the message is unpredictable, but in the absence of blocking, the maximum does not exceed L.
- The LogP model encourages programmers to adopt good plans such as job allocation, computation and communication overlap, and balanced communication patterns.
- The actual running time of the algorithm can be predicted.

Disadvantages of the LogP model

- The communication patterns in the network are not described deeply enough. For example, retransmitted messages may occupy total bandwidth, and intermediate router cache saturation is not described.
- The LogP model mainly applies to the design of message-passing algorithms. The shared storage model considers the remote read operation as equivalent to two message passes without considering the pipeline prefetching technology, data inconsistency caused by Cache, and the impact of the Cache hit rate on computation.
- The contextual overhead of multi-threading techniques is not considered.
- The LogP model assumes communication with a point-to-point message router, which increases the burden on the programmer to consider the relevant communication operations on the router.

3.3.4 Comparison of LogP model and BSP model

BSP treats all computation and communication as an overall behavior rather than an individual behavior of separate processes and communications. BSP uses delayed communication of each process to combine individual messages into a communication entity as large as possible, routed and transmitted, called large overall synchronization. It simplifies the design and analysis of algorithms (programs) but simultaneously sacrifices runtime because delayed communication means that all processes must wait for the slowest. An improved approach is to use subset synchronization, i.e., to divide all processes into subsets according to their speed so that the large overall synchronization evolves into synchronization within subsets. If the subsets are so small that each set contains only senders/receivers of messages, it becomes asynchronous individual synchronization, as described by the LogP model. That is, if the overhead caused by individual communication is taken into account in BSP and the barrier synchronization is removed, it becomes LogP, which the following equation can illustrate:

$$BSP + Overhead - Barrier = LogP$$

L.G. Valiant, the author of the BSP model, theoretically proved that parallel computing does not need to be optimized at the single-message level (e.g., LogP model). The gains in time do not outweigh the drawbacks in computational performance that are difficult to analyze and predict. Currently, questions about the BSP model focus on two points. The first point is whether the latency to a particular communication point can be the leading cause of performance degradation. Another questioned point is about the synchronization barrier frequency. Proponents of the BSP model have investigated these two issues, arguing that latency provides more opportunities to optimize communication. The use of communication aggregation and global communication synchronization can reduce congestion and competition. The main reason the synchronization barrier is more expensive for systems with shared storage structures is that most of the current underlying software does not support access to the corresponding hardware. However, in any case, the cost of the synchronization barrier can be partially offset by discounting it to global communication.

The BSP and LogP models are essentially equivalent and can be simulated with each other: the computations performed by LogP are usually a constant multiple slower when using BSP to simulate those performed by LogP and logarithmically slower when using LogP to simulate those performed by BSP. Intuitively, BSP offers more algorithm design and analysis conveniences, while the LogP model offers more control over machine resources. The loss in accuracy that BSP brings is acceptable compared to the advantages it can offer in a more structured programming style. In summary, the BSP model wins more favor due to its simplicity, performance predictability, portability, and structured nature of programming.

3.4 Hierarchical and symmetric model for distributed deep learning (HSM2DL)

This section introduces HSM2DL, which stands for the hierarchical and symmetric model for distributed deep learning. HSM2DL is an extension of the BSP parallel computing model designed for the hybrid parallel policy generation problem for neural networks. This BSP-based extension is designed for the problem of hybrid parallel plan generation. In the Section 3.5, the problem that HSM2DL aims to solve is defined, including the inputs and the search range of parallel plans. The Section 3.4.1 introduces the computer clusters used for large-scale training in deep learning and their architectures and features. This thesis proposes a hierarchical and symmetric abstract machine model to describe the features and connections of physical machines. Based on this abstract machine, the parallel algorithm analysis can be independent of the actual physical hardware and thus can circumvent the expensive profiling process. Section 3.4.2 describes the execution model of HSM2DL based on the BSP model. The execution model describes data, operator-level model parallelism, and pipeline parallelism. Section ref introduces a simple cost model of HSM2DL and describes how it simplifies and transforms the operator-level cost model based on the abstraction machine. Two typical operators, MatMul and Conv, are given as examples to demonstrate the symbolic cost model.

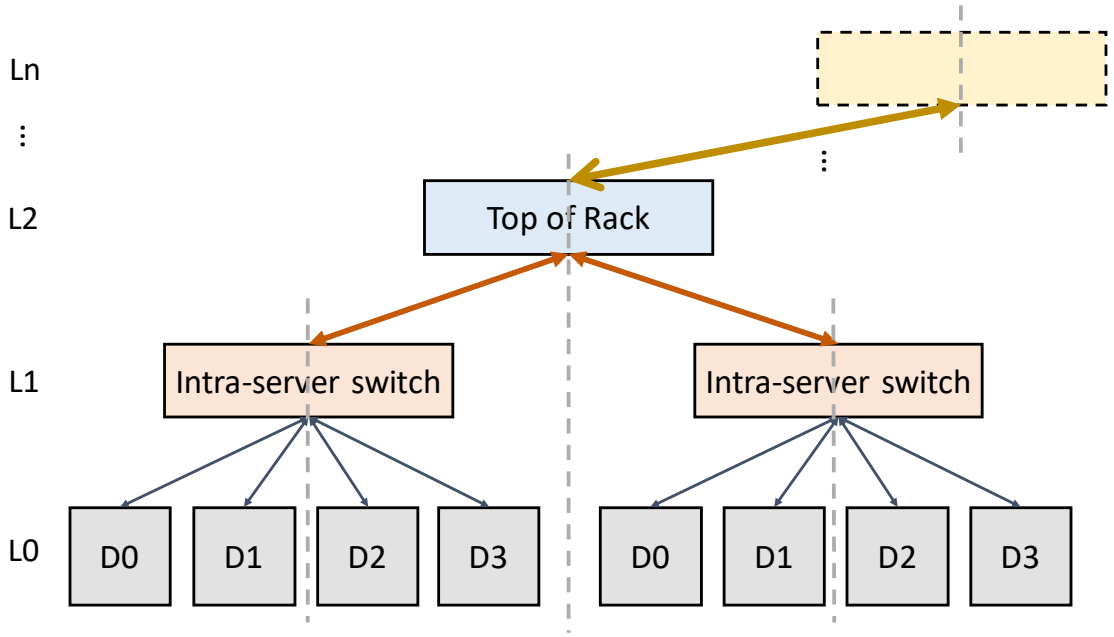


Figure 3.5: A typical DNN cluster

3.4.1 Abstract Machine

The design of abstract machines is based on the characteristics of real physical hardware. This thesis summarizes the characteristics of computer clusters used in academia and industry to train large-scale neural networks. This thesis will follow the two key observations that serve as the basis for the abstract machine. First, because of the intensive computation in neural networks, the training of the DNN model requires large-scale cluster computers, even up to cluster sizes of 1024 or 2048. The computation of distributed training is regular, and in addition, distributed training requires parameters to be synchronized between devices, so these clusters are usually using homogeneous accelerators and devices as a way to ensure load balancing. The second key point is that the network topology of the clustered computers has two main characteristics: symmetric and hierarchical characteristics [86]. These two characteristics are a common practice in data center network design [3] and also can be seen in supercomputer [19, 93] or AI training accelerator clusters[47, 59]. Figure 3.5 shows the architecture of a classic AI training accelerator cluster. The homogeneous devices within the server are connected to the first-level switch (such as a PCIe switch or NVSwitch). The second-level switch is typically named top-of-rack (ToR), and the network connections within each layer are also often homogeneous.

Motivated by this architecture, HSM2DL proposes an abstract machine for distributed training clusters of neural networks. The primary feature of this abstract machine is that it is symmetric and hierarchical. In this abstract machine, we assume that the network components (accelerators, switches, etc.) at each level of the device tree are isomorphic. The characteristics of this abstract machine can be summarized in the following three points:

- The computational capacity (FLOPS) of the homogeneous devices is the same.
- The communication capability inside each hierarchical level is the same.
- The communication capability between hierarchical levels is different, and the higher the level, the worse the communication capability.

3.4.2 Execution model of HSM2DL

This section will introduce the execution model of HSM2DL. Since the purpose of HSM2DL is to analyze hybrid parallel plans, the execution model here refers to the training process of an abstract neural network. Specifically, it is the neural network training execution process under different parallel strategies. This section first recalls the logic of neural network training and the three basic parallel modes of computation.

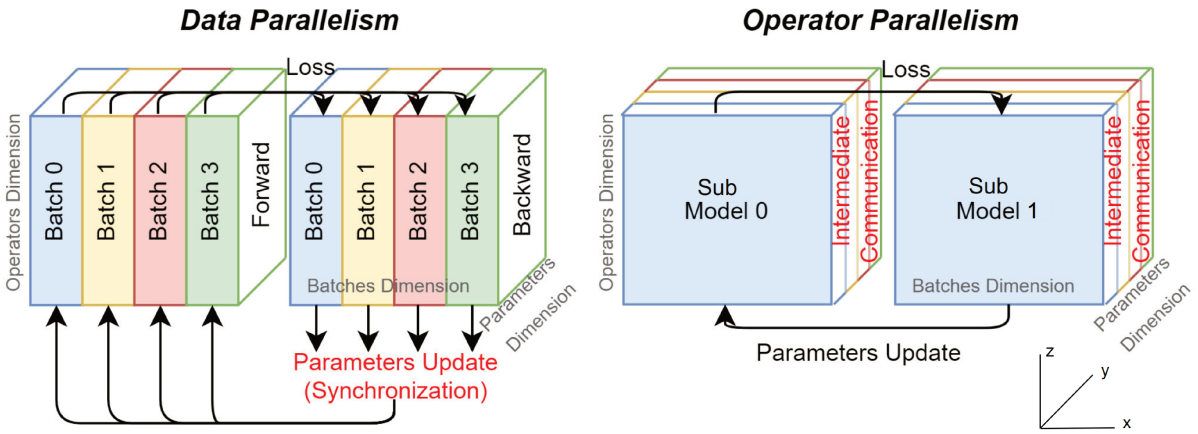


Figure 3.6: Operator level Modeling

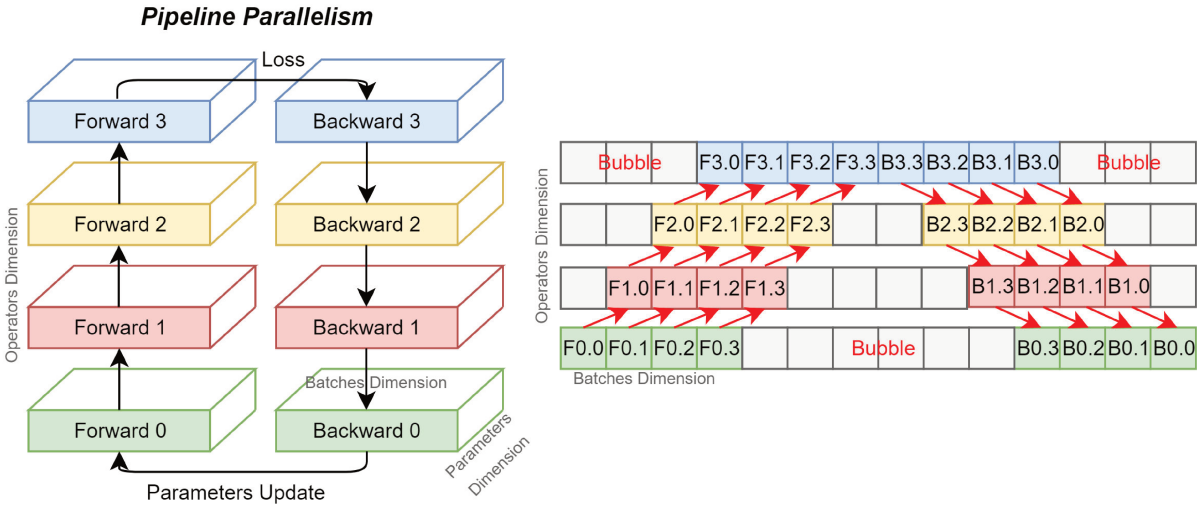


Figure 3.7: Pipeline Modeling

The DL frameworks use a computational graph to represent the DNN model. The graph is composed of several *operators*. Some of the operators have their *parameters*. Finding proper parameters that allow the DNN model to return more accurate results close to the ground truth is the objective of DNN training. A well-trained DNN model can correctly return expected results from new inputs. At the end of the DNN model, the metric to evaluate the distance between the outputs and the expected results is called *Loss Function*. Concerning the data, the minimal unit of input data is called a *sample*, and we call the process of traversing all the samples once an *epoch*. The samples are not trained one by one but are put together in a *mini-batch*. The size of the mini-batches impacts the accuracy of the DNN. After training the samples many times, i.e., through many epochs, the DNN model would find proper

parameters. The iterative process of training one mini-batch is called *Forward/Backward Propagations (FPG/BPG)*. An FPG computes a mini-batch of inputs across the DNN to get outputs. A BPG starts from the last operator back to the first operator and updates the parameters according to the Loss Function.

Figure 3.6 and Figure 3.7 abstract the neural network training as a partitionable square, where its x-axis represents the input data of the neural network, and partitioning x is equivalent to cutting the data of a batch. Y-axis represents the intra-operator dimension, and the partitioning of the y-axis is equal to the intra-operator partitioning. Z-axis represents the whole computational graph from the input to the output, which denotes the inter-operator dimension. Partitioning the z-axis is equivalent to partitioning different operators onto different subgraphs. The cubes in the two figures include the process of forward propagation, loss function, backward propagation, and operator update. The areas marked in red in the diagram are the main cost areas arising from the cuts. The following list describes each base cut's execution logic and cost generation.

The description of the three standard parallel plans is the following:

- *Data Parallelism (DP)*: In data parallel, each device possesses a DNN model replication. A mini-batch is distributed equally across the devices. Each device computes a subset of the mini-batch separately. Because each device has a complete replica of the DNN model, no communication cost is incurred in the forward and backward propagation. Because each device trains a different subset of the mini-batch, the updated parameters are different on each device at the end of the back-propagation. In order to obtain the same results as training a mini-batch on a single machine, parameter synchronization needs to be performed for devices that have different mini-batch subsets (as indicated by the red markings in the left panel of Section 3.6), and standard methods are parameter server [72] and all-reduce set communication [96]. The computation can hide the communication cost of parameter synchronization with the help of all-reduce overlap techniques Chapter 5, thus improving training efficiency. Because each device has the complete model and parameters, data parallelism does not significantly reduce memory usage. DP is inefficient for DNNs with large parameter sizes because of the time required to synchronize the updated parameters. Parameters of the DNN will be synchronized at the end of each BPG. Note that asynchronous parameter updates will not be considered, as it is impractical in critical industrial applications for convergence reasons [20].

- *Model Parallelism - operator level (OMP)*: Unlike data parallelism, operator-level model parallelism, also called operator parallelism, allows each device to have a complete mini-batch during training so that no additional communication occurs when parameters are updated. By assigning operators and parameters to different devices, operator parallelism also helps to solve the problem of saving memory usage and supporting training on larger devices. However, because the operators are partitioned, and additional point-to-point and rescheduling communication is generated during forwarding and backward propagation (as shown in the labeling of Section 3.6(b), the computational performance is heavily dependent on the quality of the slicing. In practical partitioning, there may be several different operator parallelism plans for a given operator, and the resulting cost needs to be analyzed on a case-by-case basis, as described in detail later in the cost model section.

- *Pipeline Parallelism (PP)*: The Pipeline model parallelism parallelizes the computational graph of a DNN model into several computational subgraphs, each of which is called a stage, and each stage consists of a set of connected operators. In addition to partitioning the stages, PP minimizes the latency due to computation dependencies by dividing mini-batches into micro-batches and training them in a pipelined manner. In Fig.3.7(b), *F1.0* (i.e., FPG of the second sub-model's first micro-batch) is trained in the pipeline with *F0.1*. PP helps large models adapt to limited memory. Its communication occurs only at the boundaries between sub-models. PP leads to non-negligible *Bubble Time* unlike traditional pipelining due to data dependencies. The calculation of *F1.0* requires waiting for the result of *F0.0*. In addition, the calculation of *B0.3* (*i.e.* the BPG of *F0.3*) cannot start before *B1.3* is finished. The ratio between bubble time and computation time is referred to as *Bubble Ratio*. The scalability of the PP is affected by the number of operators in the DNN. Communication between different stages will be ignored in our modeling because they are by nature small. Modeling this communication is feasible but would add additional inaccuracies and reduce search efficiency, which is therefore not discussed here.

Double Level Execution Model

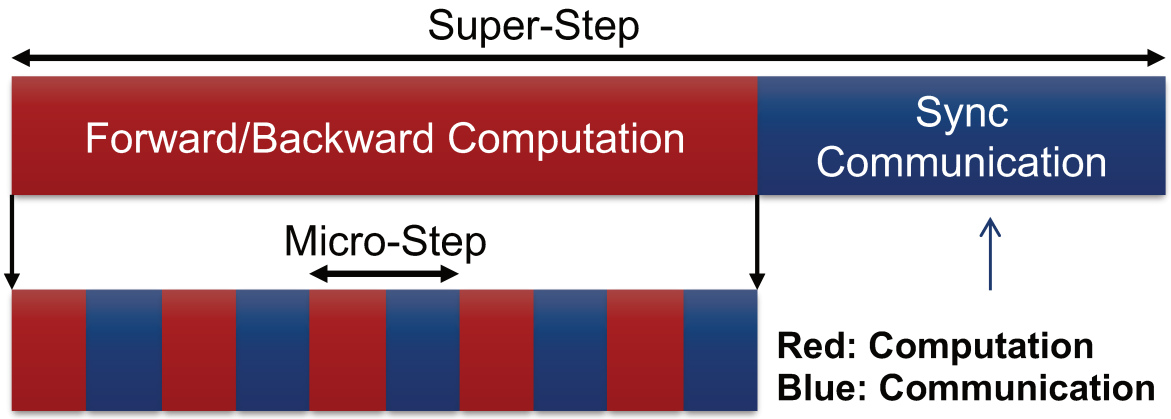


Figure 3.8: Double Level Execution Model

A BSP-based execution model for the DNN training process is represented as follows.

1) When first considering distributed training with the DP plan, the communication (parameter synchronization) occurs after the local computation (FPG/BPG), so training a mini-batch of samples can be abstracted as a SuperStep. One observation of the DL frameworks is that the BSP-defined barrier time after one *step* is minimal, which is negligible, so we model the cost of it as 0. DNN training is an iterative process of FPG/BPG, and since the operator and batch sizes are unchangeable, the training and computation of different SuperStep are identical, and therefore their cost is equal. When performing parallel plan analysis, we can evaluate the runtime performance of DNN distributed training by analyzing the cost of a super-step.

2) When considering only a mixture of DP and PP, the computation and communication in the SuperStep are naturally separated. In a SuperStep, the amount of communication is determined by the DP and has no effect on the FPG/BPG computation; the PP partitions the FPG/BPG computation across multiple stages, forming multiple pipeline SuperSteps. The effects of DP and PP on the SuperStepare independent and can be analyzed using a simple cost model.

3) However, when considering OP, FPG/BPG becomes a hybrid process of communication and computation, and independent cost analysis of SuperStep is no longer applicable. None of the existing profiling-based approaches build distributed DNN training creating specific parallel computation and execution models [87, 20, 88, 36], but evaluate the computational and total communication cost. If different kinds of communication are not distinguished, only suitable HPs can be obtained by traversal/tuning, and opportunities for further optimization are lost.

   To include the three kinds of parallelism in a unified model and to clearly describe the communication/computation in the basic model within a SuperStep, HSM2DL proposes *MicroStep* to abstract the computational process of each operator in FPG/BPG, as shown in Figure 3.8. Each MicroStep is also a basic BSP-based model, consisting of the communication and computation of operators. Same reason as for SuperStep, we idealize the assumption that the barrier time of each MicroStep is 0.

According to this two-layer execution model, the iterative training process can be described by a sequence of SuperSteps, including the pipelined FPG/BPG (the pipeline is not shown in Figure 3.8) and the communication of parameter synchronization. Besides, the FPG/BPG can be described by consequent MicroSteps, which detail the computation and communication of each operator. MicroSteps are used to represent a hybrid execution of MP/DP, while SuperSteps are used to represent a hybrid execution of a sequence of MicroSteps of DP/PP. With this two-level execution model, the training process is clearly

described and can also be used as a basis for designing a cost model that can be used as a criterion to select the optimal hybrid parallel plan.

3.4.3 Symbolic cost model

HSM2DL proposes the following **double level symbolic model** where *MS* and *SS* denotes respectively *MicroStep* and *SuperStep*:

$$Cost_{MS} = \alpha q_x + \beta(q_c + q_r) \tag{3.1}$$

$$Cost_{SS} = R_b \sum Cost_{MS} + \gamma\beta \sum q_s, \tag{3.2}$$

- α is the abstract variable that reflects the computation capacity ratio of the training accelerator.
- β is the abstract variable that reflects the communication capacity ratio of the training accelerator.
- γ is an abstract parameter reflecting the relationship between synchronization with FPG/BPG. It varies with the structure of the DNN model, and the implementation of the communication computation overlap technique of the DL framework will affect its value. Section 5.2 is an exploration of it.
- R_b is the Bubble Ratio of the SuperStep, which is vital for evaluating graph-level pipeline parallel plan search. It is detailed in Chapter 6.
- q_x signifies the amount of local computation excluding the communication time.
- q_c denotes the quantity of data that needs to be transferred between devices, known as the communication quantity of each operator.
- q_r denotes tensor redistribution quantity when one tensor is shared by two connected operators and is assigned a different parallel plan by the two operators, respectively.
- q_s denotes the number of parameters to be synchronized, which is also a kind of communication.

$Cost_{MS}$ describes the execution cost of an operator, including the computing cost and the communication cost. The communication cost comprises the cost caused by the partitioned computation and the redistribution cost caused by two operators assigned different plans. On the other hand, $Cost_{SS}$ is composed of the summation of the $Cost_{MS}$ of all the operators multiplied by the bubble ratio and the synchronization cost.

This *double level symbolic model* could be used as a metric for parallel plan searching: For operator-level parallel plan search, without pipeline bubble, R_b is fixed to 1, and this cost model can be used as the main metric for operator-level search and is detailed in Chapter 4. Chapter 6 will introduce how to use $Cost_{SS}$ for graph-level parallel plan search.

The above symbolic cost model can be used as a metric to evaluate the cost generated by the distributed training and helps to choose the optimal parallel plans. However, to use this cost model directly, it goes back to the methodology of the related works. The advantage of this symbolic cost model is that the hardware and implementation parameters (α, β, γ) and the parallel parameters $(R_b, q_x, q_c, q_r, q_s)$ are distinguished. All these parameters can be regarded as a mathematical function that could be redefined, simplified, and transformed according to abstract machine and AI domain-specific knowledge. The simplification and transformation first help reduce the search complexity without losing accuracy and also avoid profiling each operator. A detailed introduction and comparison of HSM2DL's methodology are presented in the next section.

3.5 Inputs and parallel plan range of HSM2DL

3.5.1 Planner input

The hybrid parallelism planner of HSM2DL takes the following inputs:

- A directed **computational graph** given by DNN frameworks.
  - The graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges.
  - A vertex $v \in \mathcal{V}$ represents a DNN operator (e.g., Matrix Multiplication, Convolution and ReLU), which contains a set of tensors (denoted by $\mathcal{T}_v$)
  - A tensor is denoted by t. Each tensor has a shape: *t.shape*, which is an integer vector where each integer represents the size of a dimension of the tensor
  - Each edge $e = (u, v) \in \mathcal{E}$ corresponds to a data transfer: operator v requires the output of u as its input
- **Device parameters** that describe the device's computation, memory and communication performance characteristics. These parameters include
  - the number of devices N
  - device memory capacity M
  - device floating-point-operations-per-second (FLOPS) α
  - device network bandwidth β
  - the ratio of overlapping gradient synchronization with computation r where $0 \leq r \leq 1$, and this ratio is decided based on the computational graph and the devices being used
  - the number of micro-batches K in a pipeline parallelism system

The above inputs are designed to be model-agnostic and can be obtained from DL frameworks directly. We can thus avoid requesting users to define model-specific parameters when using the planner, making this planner easy to be adopted by a large number of framework users.

3.5.2 Hybrid parallelism plan

Given the above inputs, the planner aims to compute a hybrid parallelism plan to minimize the epoch time of training a DNN. A hybrid parallelism plan $\mathcal{P} = (\mathcal{P}_g, \mathcal{P}_o)$ comprises:

- A **graph-level plan** $\mathcal{P}_g = (\mathcal{S}_0, \ldots, \mathcal{S}_{l-1})$ that partitions the input computational graph into a collection of l continuous pipeline stages $\mathcal{S}_0 \cup \cdots \cup \mathcal{S}_{l-1}$. Each stage contains a series of consecutive operators.
- An **operator-level plan** $\mathcal{P}_o$ is the set of p_v, where p_v is an operator configuration (formally defined in Definition 3.5.2) that denotes how to partition the multiple dimensions (e.g., data and model dimension) of each operator $v \in \mathcal{V}$.

Definition 3.5.1 (Tensor configuration)**.** For an m-dimensional tensor t, the tensor configuration $p^t = \langle d_0, \ldots, d_{m-1} \rangle$ is a vector of m integers, where $d_i, 0 \leq i < m$ denotes the number of partitions along dimension i.

Definition 3.5.2 (Operator configuration)**.** For an operator v with n input tensors, the operator configuration $p_v = \langle p_0^t, \ldots, p_{n-1}^t \rangle$ is a vector of n tensor configurations.

Note that not all operator configurations are valid. For example, $\langle \langle 1, 2 \rangle, \langle 1, 1 \rangle \rangle$ is an invalid configuration for Op 0 in Figure 3.9 because the dimension of the partitioned first tensor mismatch with the corresponding dimension of the second tensor.

Figure 3.9 shows an example where the planner takes a graph and device parameters as inputs. This graph contains two MatMul operators (i.e., Op 0 and Op 1). The planner needs to generate a plan that combines pipeline, data, and model parallelism. The plan consists of graph-level sub-plan p_g and operator-level sub-plan p_o. $p_g = (\mathcal{S}_0 = \{v0\}, \mathcal{S}_1 = \{v1\})$ implies that the computational graph is divided into $l = 2$ pipeline parallelism stages, where Op 0 is assigned to stage 0 and Op 1 to stage 1. Along with pipeline parallelism, p_o describes the operator-level plan for each operator $v \in \mathcal{V}$. $p_{v0} = \langle \langle 2, 1 \rangle, \langle 1, 1 \rangle \rangle$ indicates that the first tensor of Op 0 is partitioned on the batch dimension (first dimension) across two devices (i.e. data parallelism). Similarly, $p_{v1} = \langle \langle 1, 1 \rangle, \langle 1, 2 \rangle \rangle$ indicates that the model dimension (second dimension) of the second tensor of Op 1 is partitioned across two devices (i.e. model parallelism).

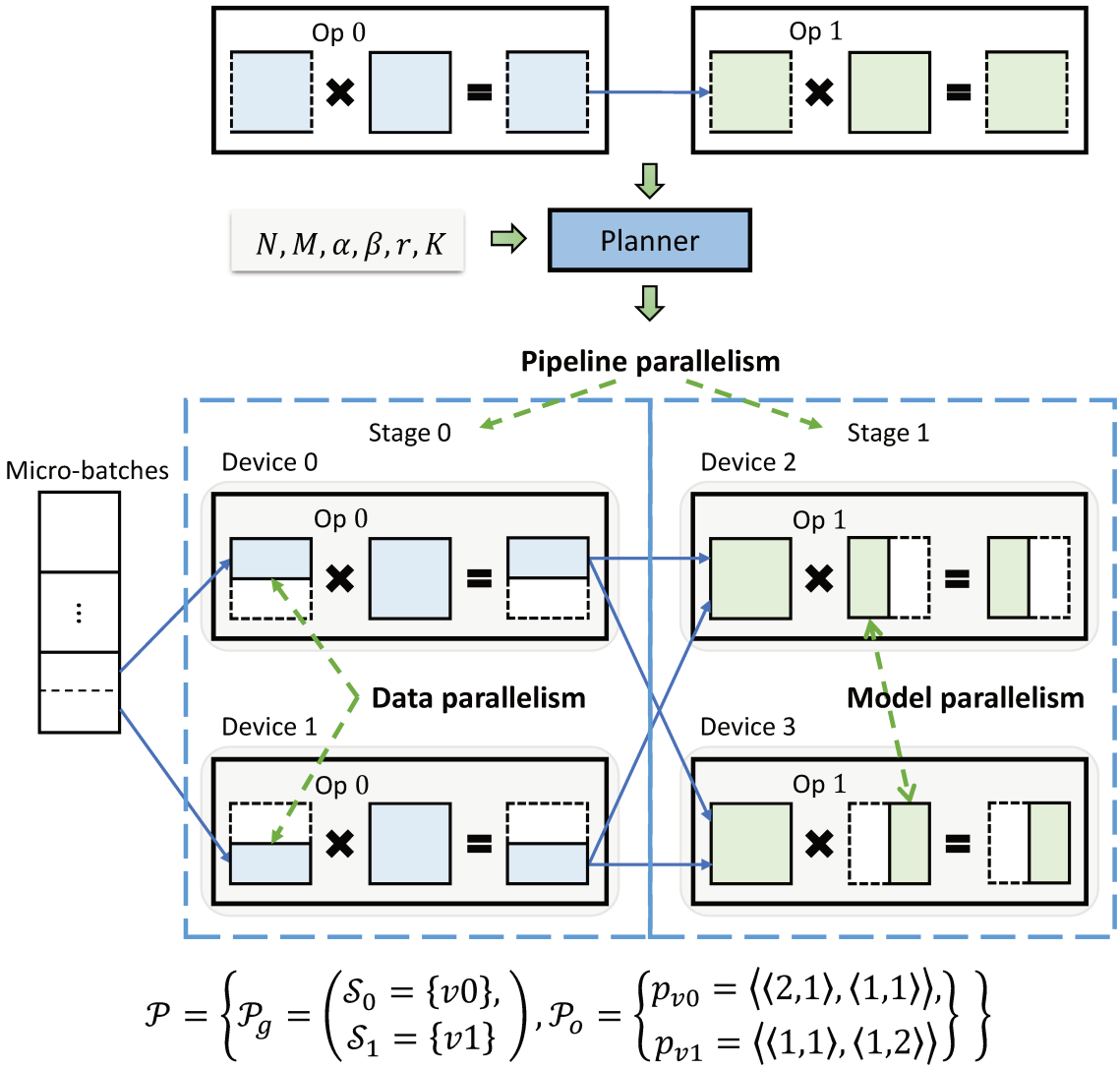


Figure 3.9: Example of a hybrid parallelism plan.

Many prior related studies [35, 20] have proven that exploring all configurations for graph-level and operator-level plans is an NP-hard problem that requires exponential time complexity. We are thus interested in polynomial-time algorithms that can efficiently explore the search space and find plans that can effectively reduce the training time of DNNs.

3.6 Comparison between HSM2DL and the related works

In this section, we try to compare the difference in methodologies between related works Section 2.4.3 and HSM2DL. This section will use the operator-level cost model as an example to show the similarities between the two methodologies and their differences to present the advantage of HSM2DL.

Both the related works and HSM2DL take the same computational graph and hardware cluster as inputs, and therefore theoretically, they should define the same searching space. And both of them expect to output the optimal parallel plan regarding the inputs.

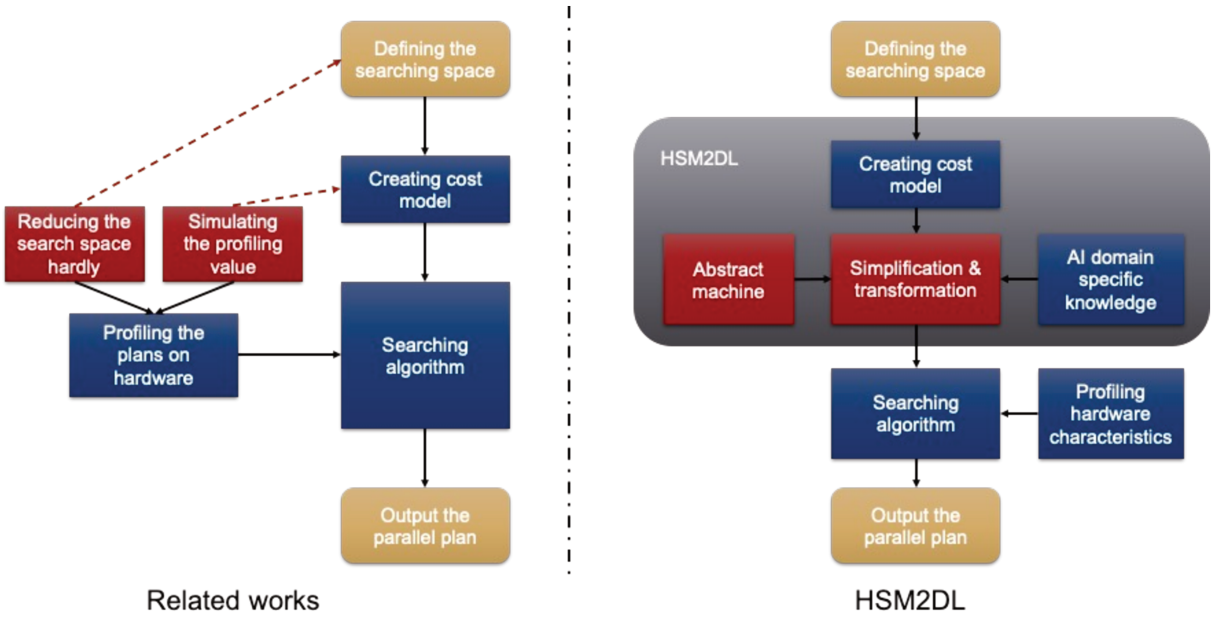


Figure 3.10: Comparison between related works and HSM2DL

3.6.1 Related works' methodology

The difference comes from how they model and treat the searching problem. The related works regard the solving problem totally as a mathematical problem. This methodology first defines the searching space and builds a numerical cost model composed of the execution time of different parallel plans. In order to improve the searching accuracy, the cost model and the searching algorithm need to be built as detailed as possible by adding all the mathematical and physical conditions.

After building such a kind of operator-level cost model, the next step is to profile t_e, t_e, t_r under different parallel plans inside the searching space. Finally, a searching algorithm would select the optimal parallel plan according to the cost model.

The advantage of this methodology is that building the cost model is relatively easy because it is only a mathematical problem describing the execution time and the redistribution time. Recall the operator-level cost model example introduced in Equation (2.1). Detailed definitions of the variables in the equation can be found in the previous section.

$$T(\mathcal{G}, \mathcal{P}_o, \mathcal{D}) = \sum_{v_i \in \mathcal{V}} (t_e(v_i, p_{v_i}, \mathcal{D}) + t_s(v_i, p_{v_i}, \mathcal{D})) + \sum_{e_{ij} \in \mathcal{E}} t_r(e_{ij}, p_{v_i}, p_{v_j}, \mathcal{D})$$

On the other hand, the drawback of the related works is brought by the predetermined exact cost model that consists of execution times. In Section 2.3.5 we mentioned that for operator-level, to distribute a computational graph with n operators into 2^p devices where each op includes two x dimensions tensors. There are $(C_{x+p-1}^{p}{}^{2})^n$ possibilities. Such a high complexity will generate both a high complexity in the search time and make the search time unacceptable. In addition, it generates an equally expensive profiling preparation because the profiling results are not reusable when different network models and hardware change, essentially making this seemingly accurate approach to search on large-scale datasets and large-scale clusters an intractable problem.

In order to make this methodology more practical, researchers reduce the search complexity and profiling efforts by manually narrowing down the search space based on their expertise. However, this may lead to losing the possibility of finding the optimal solution. On the other hand, to further reduce the workload of profiling, the researchers used a particular symbolic model to simulate the execution time. This simulation brings inaccuracy to the parallel plan search.

In a nutshell, simplifying modeling throws the problem's complexity to the search algorithm and the profiling effort. This approach is systematically customized and has no way to balance accuracy and generality, thus losing the possibility of future optimization.

3.6.2 HSM2DL's methodology

The related works cannot find the balance between accuracy and complexity because, from the point of view of an AI expert, the operators are the minimal unit to be evaluated in the cost model analysis. Besides, they only model the cost but not the hardware machine. When the hardware characteristics and the parallel plans are combined, the variables in the cost model can only be obtained through profiling. Therefore, either they go through the unrealizable search space to get an accurate result, or they reduce the search space to lose the possibility of finding the optimal parallel plan.

The insight of HSM2DL is to take advantage of the computing parallel model to decouple the hardware characteristics and the parallel analysis. As shown in Figure 3.10, after creating the cost model of the execution time (similar to that of the related works), HSM2DL will apply the features of its abstract machine and AI domain-specific knowledge to simplify and transform the cost model. HSM2DL doesn't need the exact execution time to make a parallel plan decision but to compare the extra cost caused by different parallel plans. In fact, the objective is to choose a better parallel plan instead of an accurate estimation of the execution time.

Because of such a modeling methodology, the parallel cost analysis is split from the hardware information. The heavy profiling task is avoided. Also, the search space is reduced according to the abstract machine and AI domain-specific knowledge. From the Figure 3.10, it can be conducted that the computing parallel model ensures that the searching space of the searching algorithm is relatively tiny. And the profiling task is also limited to profiling some fundamental information about the hardware like the floating-point-operations-per-second(FLOPS) and the actual communication capacity. In fact, the optimality of the founded parallel plan depends on how the computing parallel model is built. Chapter 4 and Chapter 6 detailed how HSM2DL searches the operator-level plan and joint search of op-level and graph-level, respectively.

An example demonstrates how a basic operator-level symbolic cost model is created. Note that this example doesn't do any symbolic simplification so it is actually equal to the cost model of the related works. The simplification in order to reduce the searching complexity and guarantee the results' optimality is introduced in Chapter 4.

This equation is an example of an operator-level symbolic cost model ($R_b = 1$) to compare the performance of two plans $\mathcal{P}_o$ on a computational graph G:

$$C(\mathcal{G}, \mathcal{P}_o) = \sum_{v_i \in \mathcal{V}} (\alpha \times q_x(v_i, p_{v_i}) + \beta \times (q_c(v_i, p_{v_i}) + \gamma \times q_s(v_i, p_{v_i}))) + \sum_{e_{ij} \in \mathcal{E}} \beta \times q_r(e_{ij}, p_{v_i}, p_{v_j}) \tag{3.3}$$

The variables in the above equation are:

- α is an abstract parameter representing the accelerator's real-time computation capacities.
- β is an abstract parameter representing the hardware cluster's communication capacity.
- γ is an abstract parameter for the ratio of the parameter synchronization overlap.

  The value of γ is between zero and one: $\gamma \in [0, 1]$. If parameter synchronization is not activated or some DL frameworks do not support this enhancement method, in this case, $\gamma = 1$; if the synchronization quantity is small, which could be totally overlapped under the computation, $\gamma = 0$. Chapter 5 describes how to calculate γ.

- $q_x(v_i, p_{v_i})$ denotes the symbolic function of the computation quantity for an operator v_i under the plan S_{v_i}.
- $q_c(v_i, p_{v_i})$ denotes the symbolic function of the intra-communication quantity caused by model (op-level) parallel.

- $q_r(e, p_{v_i}, p_{v_j})$ denotes the symbolic function of the quantity of data redistribution caused by conflicting plans of two connected operators.
- $q_s(v_i, p_{v_i})$ denotes the symbolic function of the parameter synchronization caused by data parallel.

HSM2DL classes the variables into two categories:

- **Profiled hardware parameters:** α, β, γ

  During the distributed training, the calibrated FLOPS and bandwidth usually cannot be fully used, and these two values are obtained through profiling the hardware environment. The parameter synchronization ratio's value varies regarding the DNN model and DL framework implementation. The profiled value of these variables would be substituted into the cost model in the last step of the search.

- **Symbolic function of data quantity:** $q_x(v_i, p_{v_i}), q_c(v_i, p_{v_i}), q_r(e, p_{v_i}, p_{v_j}), q_s(v_i, p_{v_i})$

  These symbolic functions reflect the data computing or movement caused by the distributed training, which are the critical points for evaluating the performance of parallel plans. They are created in advance of the search for each operator type.

  Diverse modern DNN models have come to earth recently, but all of them are composed of the 20+ basic computational operators like MatMul, Conv, etc., and 100+ element-wise operators like elementary arithmetic Add. HSM2DL classes these operators into 20+ types and analysis their semantics to build the symbolic cost model. The cost models are built for each operator type under different plans.

From the above analysis, we can conduct that the symbolic cost model is essentially an abstraction of the numerical cost model proposed by the related works. The two models 2.1 and 3.3 are basically the same cost but described in different scope: $t_e(v_i, p_{v_i}, D)$ in the profiling-based model is actually the summation of $\alpha \times q_x(v_i, p_{v_i}) + \beta \times q_c(v_i, p_{v_i})$ in our symbolic model. $t_s(v_i, p_{v_i}, D)$ equals to $\beta \times q_s(v_i, p_{v_i})$ and $t_r(e, p_{v_i}, p_{v_j}, D) = \beta \times \gamma \times q_r(e, p_{v_i}, p_{v_j})$.

Therefore, the symbolic cost model can help to find the optimal parallel plan as accurately as the related works without heavy profiling tasks. HSM2DL quantitatively evaluates the cost of computation, intra-communication, and redistribution with the defined symbolic cost model. HSM2DL only profiles the communication and computation capacity of the hardware. However, until now, HSM2DL faces the same searching complexity. The following chapters Chapters 4 and 6 will introduce how HSM2DL deals with the searching complexity without losing accuracy. In other words, how HSM2DL overcomes the disadvantage of the related works is that they cannot balance optimality and feasibility well.

3.6.3 Conclusion

This section first introduces the parallel computing model and illustrates the advantages and shortcomings of typical models such as PRAM, LOGP, and BSP. Then this section proposes a parallel computing model HSM2DL for deep neural network training is introduced, including the definition of the abstract machine, the description of the execution model, and the cost model. Finally, by comparing with the methodology of related work, it is shown that HSM2DL can well handle the balance between complexity and accuracy in large-scale model parallel plan search.

Chapter 4

Operator Level Parallel Plan Search

4.1 Introduction

Chapter 2 discusses the standard parallel plans in distributed DNN training and the difficulties of hybrid parallel plan searching. Chapter 2 presents current academic work on hybrid plan search. Chapter 3 discusses the parallel computing model and HSM2DL, the bridging-model-based model for deep learning parallel computing, including its abstract machine, execution, and preliminary cost model.

In a hybrid parallel plan searching, the related works first build a cost model to estimate the execution time for distributed training on different hybrid parallel plans. These works profile operators' performance under different parallel plans on specific hardware. The profiled execution times are directly substituted into the cost model to select the hybrid parallel plan with the shortest estimated execution time. This 'top-down' tuning method may find accurate results on small-scale models, but as the networks become larger and the number of devices increases, the parallel search time and the amount of profiling work become unacceptable (>24 hours). The hybrid plan's quality is reduced if the workload is reduced by manually compressing the search space and omitting particular possibilities. In other words, this methodology cannot find a balance between parallel plan search accuracy and complexity and is, therefore, difficult to apply to industrial production. Another drawback of the methodology of the correlation method is its poor extensibility. For example, building an operator-level cost model and wanting to extend this model to support more parallelisms, such as pipeline parallelism, is equivalent to rebuilding a new model from the top down for the whole problem.

This chapter will describe how we take advantage of the HSM2DL bridge model for hybrid parallel plan search. In fact, unlike related work that first proposes a concrete cost model and then simplifies it, HSM2DL starts from a bridging model and builds a symbolic cost model based on an abstract machine. HSM2DL's methodology is a bottom-up solution like 'putting together building blocks'. This bottom-up modeling methodology allows HSM2DL to balance the search complexity with the optimality of the parallel plan; on the other hand, the analysis of HSM2DL is modularly extensible and not as customizable as related work.

A symbolic cost model, equivalent to the cost model of the related work, is introduced in Chapter 3 and can circumvent the expensive profiling effort. However, the giant search space remains a pressing problem in operator-level plan searches. It is pointed out in Section 2.3.5 that this search space is exponentially related to the number of operators and polynomially related to the logarithm of the number of devices. Section 4.2 will present how HSM2DL proposes a recursive partitioning method based on the symmetry and hierarchical properties of the abstract machine, which transforms the directly partitioning 2^n devices into n times 2-parts partitioning, compressing the complexity associated with the number of devices to linear with guaranteed accuracy.

DL frameworks [2, 62, 1] provide good functionalities on data loading, computational graph execution, and fault tolerance. Some of the frameworks [1, 49] support hybrid parallelism with a manually configured parallel plan. However, the biggest challenge for these frameworks is how to decide the optimal hybrid parallel plan automatically. Starting with this chapter, the main focus is on how to use the features of HSM2DL to perform parallel plan searches and send the results to the DL frameworks.

This chapter gives examples of 2-parts possible partition dimensions and the cost model for the two typical operators, MatMul and Conv. The cost of redistribution between different operators brings

exponential complexity w.r.t the number of operators for the search. Symbolic transformation and simplification based on the third homomorphism are presented in Section 4.3. Furthermore, a flex-edge graph structure is proposed that can be flexibly ordered to search the high-quality parallel plan in a heuristic way and within as much control as possible. Section 4.4 demonstrates a double recursive algorithm based on recursive 2-parts partitioning and functional recursive searching, which can compress the search complexity to linearity and give a near-optimal hybrid parallel plan in a search time of seconds while ensuring the accuracy of the parallel plan. Section 4.5 demonstrates experimentally that the double recursive algorithm of HSM2DL can find high-quality parallel plans in linear search time while balancing search accuracy and complexity.

4.2 Symbolic simplification based on the abstract machine

As shown in Figure 3.10, the most significant advantage of HSM2DL is its ability to simplify the symbolic cost model based on abstract machine and AI domain-specific knowledge. This feature-based abstraction does not decrease the accuracy of the search but can significantly reduce the complexity. This section will describe the recursive 2-parts partitioning proposed by HSM2DL based on the features of symmetric and hierarchical abstraction machines. This recursive partitioning converts directly partitioning the computational graph onto 2^n devices into partitioning n times the graph into two parts. On the other hand, this section discusses how the cost model is reduced based on AI domain-specific knowledge and that only additional communication costs need to be considered in an operator-level parallel plan search. Examples of classical operators MatMul and Conv in possible partition dimensions are given in Section 4.2.2.

4.2.1 Symbolic simplification

As discussed in Equation (3.3), for operator-level parallel plan searching, the cost model is :

$$C_o(\mathcal{G}, \mathcal{P}_o) = \sum_{v_i \in \mathcal{V}} (\alpha \times q_x(v_i, p_{v_i}) + \beta \times (q_c(v_i, p_{v_i}) + \gamma \times q_s(v_i, p_{v_i}))) + \sum_{e_{ij} \in \mathcal{E}} \beta \times q_r(e_{ij}, p_{v_i}, p_{v_j})$$

For a parallel program, the total cost is the summation of the local computation and communication costs. The local computation is the process executed locally on each device without external data. Communication denotes data transferring between devices caused by the distribution.

Before symbolically reducing this cost model by integrating the AI domain-specific knowledge and the abstract machine characteristics, this thesis focuses on application domains that are widely used in industries, including computer vision, natural language processing, and recommended systems. Other computational graph parallelization problems (e.g., scientific computing) are out of the scope of this thesis.

Within the scope of this thesis, the following are some AI domain-specific facts and assumptions:

- Tensors in DNNs are dense multi-dimensional arrays.
- The operators (e.g., *Matmul*, *Conv*, *Add*, etc.) are massively parallelizable computations. The dense computation (in contrast to the sparse computation) is able to be evenly parallelized among the devices. Therefore, it is possible to ensure load-balancing of the computation.
- DL platforms [1] automatically and intelligently control the operators' mapping and scheduling, which ensures load-balancing.

Based on the above facts and assumptions, it can be conducted that the number of operations to perform is constant for any parallel plan and a fixed number of devices. Therefore $q_x(v_i, p_{v_i})$ is equal for any chosen parallel plan on a given number of devices. Recall that the objective of HSM2DL's cost model is not to predict the real execution value of a given parallel plan but to be used as a metric to compare the performance of different parallel plans. It is unnecessary to calculate the real value of $q_x(v_i, p_{v_i})$ to compare different partition plans but the relative costs just need to be compared. Taking the assumption of the abstract machine that the computational capacity of the training devices is the

same by load-balanced mapping, α can be regarded as a constant. As a result, our asymptotic analysis can focus only on communication.

$$\hat{C}_o(\mathcal{G},\mathcal{P}) = \cancel{\sum_{v_i\in\mathcal{V}} \alpha \times q_x(v_i,p_{v_i})} + \sum_{v_i\in\mathcal{V}} \beta \times (q_c(v_i,p_{v_i}) + \gamma \times q_s(v_i,p_{v_i})) + \sum_{e_{ij}\in\mathcal{E}} \beta \times q_r(e_{ij},p_{v_i},p_{v_j})$$

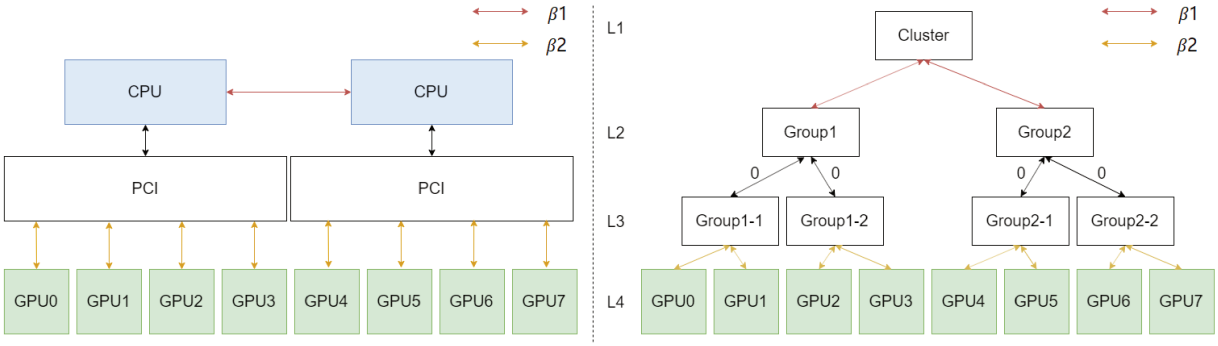


Figure 4.1: A typical GPU architecture described by a recursive tree

The communication cost is determined by two factors: the communication capacity of the chosen machines, denoted by β, and the quantity of data needed to be transferred, denoted by q_c, q_s, q_r. To achieve better performance, modern computer clusters, like supercomputers [19, 93], and AI accelerator clusters [47, 59], have a hierarchical architecture. A unique β cannot describe precisely the communication capacity of modern machines. Valiant [84] proposed to use different β for each hierarchical level. The communication cost can be calculated by the summation of each level contribution: $\hat{C}_o(\mathcal{G},\mathcal{P}) = \sum_j \beta_j \times (q_{c_j} + q_{r_j} + \gamma \times q_{s_{v_j}})$, where j is the hierarchical level. From the definition of HSM2DL's abstract machine, the hierarchical architectures are also typically symmetric [43]. Inspired by SGL[44], a hierarchical and symmetric machine can be abstracted in a recursive way. For example, a typical GPU architecture shown on the left of Figure 4.1, can be described by an abstract machine on the right. The abstract machine has a tree structure, where the leaves are the computing devices, and the branch nodes model the hierarchical structure. The communication is analyzed by a recursion. Each recursion step is a level of the tree whose communication capacity is shared as β_j. For each level, β_j does not affect the choice of the parallel plan.

$$\hat{C}_o(\mathcal{G},\mathcal{P}) = \cancel{\sum_{v_i\in\mathcal{V}} \alpha \times q_x(v_i,p_{v_i})} + \sum_{v_i\in\mathcal{V}} \cancel{\beta} \times (q_c(v_i,p_{v_i}) + \gamma \times q_s(v_i,p_{v_i})) + \sum_{e_{ij}\in\mathcal{E}} \cancel{\beta} \times q_r(e_{ij},p_{v_i},p_{v_j})$$

The original cost model C is simplified by symbolic derivation, which is formally introduced in the following theorem.

Theorem 4.2.1 (Symbolically simplified cost model)**.** *Given that the system model is hierarchical, the plan $\mathcal{P}_o$ which minimizes the original cost model $C_o(\mathcal{G},\mathcal{P}_o)$ is the same $\mathcal{P}_o$ which minimizes the following symbolically simplified cost model*

$$\hat{C}_o(\mathcal{G},\mathcal{P}_o) = \sum_{v_i\in\mathcal{V}} (q_c(v_i,p_{v_i}) + \gamma \times q_s(v_i,p_{v_i})) + \sum_{e_{ij}\in\mathcal{E}} q_r(e_{ij},p_{v_i},p_{v_j}) \tag{4.1}$$

Formally, given a computational graph $\mathcal{G} = (\mathcal{V},\mathcal{E})$, $\forall \mathcal{P}_o, \mathcal{P}'_o$, $C_o(\mathcal{G},\mathcal{P}_o) \geq C_o(\mathcal{G},\mathcal{P}'_o)$ if $\hat{C}_o(\mathcal{G},\mathcal{P}_o) \geq \hat{C}_o(\mathcal{G},\mathcal{P}'_o)$.

Proof. The theorem is proved from the assumptions of HSM2DL.

First, by substituting the definition of the original cost model, $C_o(\mathcal{V},\mathcal{P}_o) \geq C_o(\mathcal{V},\mathcal{P}'_o)$ can be expended as

$$\sum_{v_i\in\mathcal{V}} \alpha \times q_x(v_i,p_{v_i}) + \beta \times (q_c(v_i,p_{v_i}) + \gamma \times q_s(v_i,p_{v_i})) + \sum_{e_{ij}\in\mathcal{E}} \beta \times q_r(e_{ij},p_{v_i},p_{v_j}) \geq$$
$$\sum_{v_i\in\mathcal{V}} \alpha' \times q_x(v_i,p'_{v_i}) + \beta' \times (q_c(v_i,p'_{v_i}) + \gamma' \times q_s(v_i,p'_{v_i})) + \sum_{e_{ij}\in\mathcal{E}} \beta' \times q_r(e_{ij},p'_{v_i},p'_{v_j})$$

Because the computation cost C_c and the computation capability α are independent of the partition plan (i.e., $\alpha{=}\alpha'$ and $q_x(v_i, p_{v_i}) = q_x(v_i, p'_{v_i})$), we eliminate the first term on both sides to obtain

$$\sum_{v_i \in \mathcal{V}} \beta \times (q_c(v_i, p_{v_i}) + \gamma \times q_s(v_i, p_{v_i})) + \sum_{e_{ij} \in \mathcal{E}} \beta \times q_r(e_{ij}, p_{v_i}, p_{v_j}) \geq$$
$$\sum_{v_i \in \mathcal{V}} \beta' \times (q_c(v_i, p'_{v_i}) + \gamma' \times q_s(v_i, p'_{v_i})) + \sum_{e_{ij} \in \mathcal{E}} \beta' \times q_r(e_{ij}, p'_{v_i}, p'_{v_j})$$

While generally, network bandwidths vary in modern DNN training clusters, cost models are only used at the same level in our algorithm, which implies that network capacities are equal for different partition plans (i.e., $\beta = \beta'$). Therefore, we can further simplify the above equation to

$$\sum_{v_i \in \mathcal{V}} q_c(v_i, p_{v_i}) + \gamma \times q_s(v_i, p_{v_i}) + \sum_{e_{ij} \in \mathcal{E}} q_r(e_{ij}, p_{v_i}, p_{v_j}) \geq$$
$$\sum_{v_i \in \mathcal{V}} q_c(v_i, p'_{v_i}) + \gamma' \times q_s(v_i, p'_{v_i})) + \sum_{e_{ij} \in \mathcal{E}} q_r(e_{ij}, p'_{v_i}, p'_{v_j})$$

The proof is finished by recalling the definition of $\hat{C}_o(\mathcal{G}, \mathcal{P}_o)$. □

In this simplified cost model, the device parameters α, β are eliminated and would not impact the operator partition plan searching and let the results be robust.

4.2.2 Recursive partitioning

Given the simplified cost model in Equation (4.1), the cost analysis can only focus on the communication cost. However, it is still too expensive to list symbolic cost functions for all possible configurations because of the high search complexity. Therefore, this section introduces recursive partitioning, which partitions a computational graph into 2^p parts by recursively partitioning the graph into 2 parts through p recursive steps.

The parallel plan determines how tensors are distributed into devices. The parallel plan search can be formalized by logically setting how to input, and output tensors of operators are evenly distributed among the devices (e.g., GPU0-7 in Figure 4.1). The parallel plans for each level of the recursive tree (GPU architecture) in Figure 4.1 depend on the number of branches of the architecture. A specific number of branches acquires a specific set of cost functions. For the related works Section 2.4.3, it is required to profile each possibility on targeted devices. For HSM2DL, the same number of possible cost functions could be defined for each operator, but it is an unrealistic task.

However, in the abstract machine defined by HSM2DL, each level of the recursive tree is homogeneous, like GPU0-3 in Figure 4.1. It can be transformed again into a multi-level tree. Besides, in real academic and industrial practices, the number of devices is usually a power of 2 to achieve the best performance. Therefore, the recursive tree can be transformed into a full binary tree, and the partitioning of the symmetric architectures can be realized by recursive dichotomy. As a result, the cost function defined for the operators is the cost for each primitive configuration that partitions the operators into two parts. The examples of cost functions of primitive configurations are given in Section 4.2.4.

Instead of predicting the execution time, the goal is to find the best parallel plan. As a result, it is sufficient to assume that all devices operate consistently, and minor performance differences between the same devices can be ignored. Furthermore, the hierarchical abstract machine can be used to decompose the heterogeneity of symmetric architecture. Based on the assumptions stated above, homogeneity is applied to all of the 2-part analyses in this thesis.

Definition 4.2.2 (Composition)**.** Let two tensors t, t' and their configurations $p^t = \langle d_0, \ldots, d_{m-1} \rangle, p^{t'} = \langle d'_0, \ldots, d'_{m-1} \rangle$ of tensor t. The composition of p^t and $p^{t'}$ is the element-wise multiplication of the two vectors. Correspondingly, the composition of two operator configurations $p_v = \langle p^{t_0}, \ldots, p^{t_{n-1}} \rangle, p_{v'} = \langle p^{t'_0}, \ldots, p^{t'_{n-1}} \rangle$ of operator v is the element-wise composition of $p^{t_i}, p^{t'_i}, 0 \leq i < n$.

Definition 4.2.3 (Primitive Configuration)**.** A primitive configuration p^ρ is an operator configuration that cannot be composed by other configurations. Also referred to as the 2-parts partitioning configuration in this thesis. $\mathbb{P}_v$ denotes the set of all primitive configurations of v.

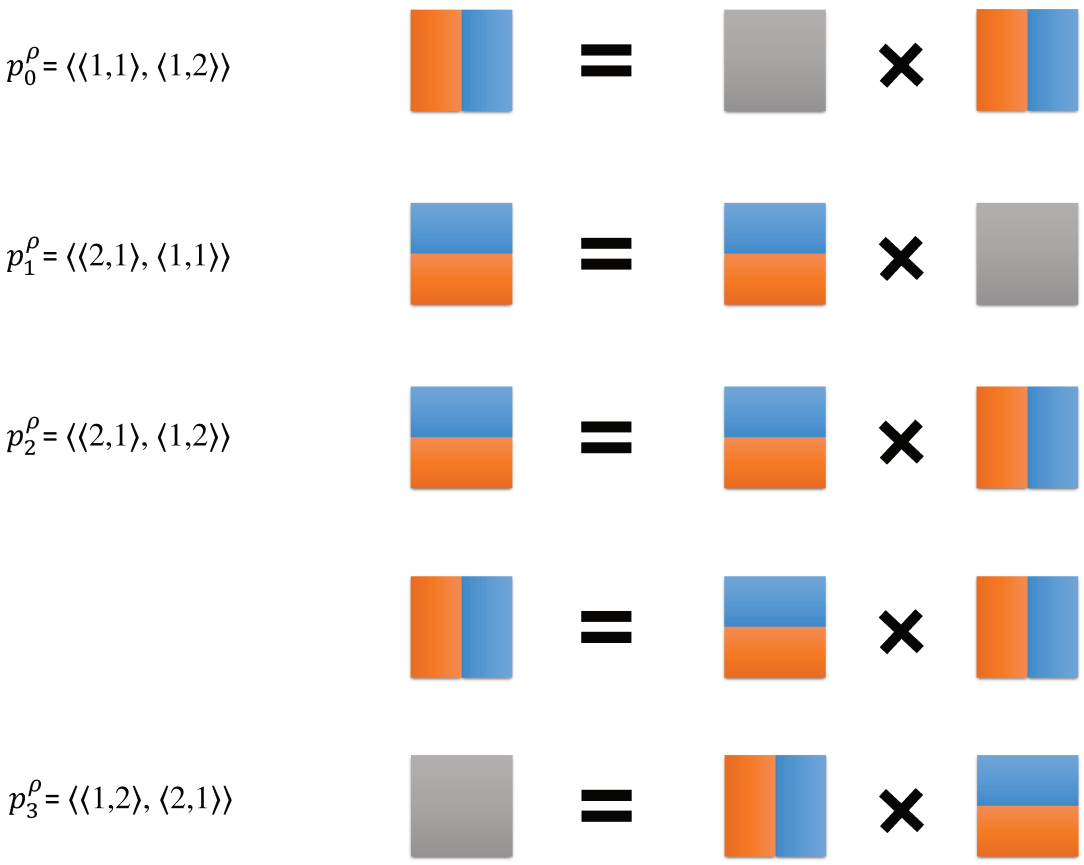


Figure 4.2: Example of primitive configurations of a 2D MatMul

For example, the primitive configuration set of a 2D MatMul operator is

$$\mathbb{P} = \{p_0^\rho = \langle\langle 1,1\rangle, \langle 1,2\rangle\rangle, p_1^\rho = \langle\langle 2,1\rangle, \langle 1,1\rangle\rangle, p_2^\rho = \langle\langle 2,1\rangle, \langle 1,2\rangle\rangle, p_3^\rho = \langle\langle 1,2\rangle, \langle 2,1\rangle\rangle\} \tag{4.2}$$

Note that not all the primitive configurations are included in the set $\mathbb{P}$ because some random primitive configurations are always not optimal. Take the above 2D MatMul as an example, one possible primitive configuration $p^\rho = \langle\langle 1,2\rangle, \langle 1,2\rangle\rangle$ always requires communicating the whole two tensors which is always more expensive than $p_4^\rho = \langle\langle 1,2\rangle, \langle 2,1\rangle\rangle$. Therefore $p^\rho = \langle\langle 1,2\rangle, \langle 1,2\rangle\rangle$ would not be included into the set $\mathbb{P}$. The primitive configurations in the set $\mathbb{P}$ are also referred to as *Possbile Partition Dimensions(PPDs)* in this thesis.

Definition 4.2.4 (Recursive Partitioning)**.** Recursive partitioning means that when searching a parallel plan for a computational graph assigned to a 2^p device, instead of choosing directly from all tangent possibilities, optimal 2-parts partitioning configurations (i.e., primitive configuration Definition 4.2.3) will be recursively chosen. After p recursion steps, the optimal parallel plan is obtained by combining (i.e., composition Definition 4.2.2) the optimal dichotomous configuration selected.

Here, leveraging two properties of distributed DNN training:

- The training clusters of DNN models are hierarchical and symmetric as introduced in Section 3.4.1. These properties enable us to evenly divide the problem into two sub-problems where the solution of the sub-problem can be used by the other.

- The composition (Definition 4.2.2) of two optimal parallelism configurations is still optimal. The proof can be found in Section 4.2.3. Intuitively, this is because the composition of the cost function is commutative and associative.

Given the above two properties, the idea is to find the optimal configurations step by step. For example, to find an optimal configuration to partition an operator into 4, first find the optimal primitive configuration (formally defined in Definition 4.2.3) to partition an operator into two sub-operators and then find an optimal primitive configuration to partition one sub-operator into 2. Finally, the combination of two optimal primitive configurations is optimal.

Example 4.2.5. *It is obvious that partitioning an operator evenly into 2^p parts can be done by dichotomy with p recursions. Let us take a matrix as an example of a tensor, a matrix partitioned into eight parts along columns can be the result of partitioning into two parts along columns recursively twice; a matrix can be partitioned into 2×2 grids by firstly partitioning along columns and then recursively along rows. Therefore the recursive 2-part partitioning can be well-mapped to the symmetric architecture.*

Definition 4.2.6 (Split)**.** *Split* specifies the change in variation of the tensor in the operator during the *Composition* process. We define the process of applying a configuration c_t to tensor t as

$$split(t, p^t) = (t.shape[0]/d_0, ..., t.shape[m-1]/d_{m-1}),$$

where $d_i \in p^t, 0 < i < m - 1$.

4.2.3 Optimality proof

The section tries to show that the recursive partitioning in Section 4.2.2 can continuously optimize the quality of a plan by iteratively creating and composing primitive configurations. The initial computational graph describes the primitive configuration of all the operators. The complexity of this algorithm is reduced to linear w.r.t the logarithm of the number of devices because, at each step (from 0 to $\log_2 N - 1$), only the optimal primitive plan is kept. The recursive partitioning is based on the assumption that per-step optimality can lead to global optimality. In other words, the optimal operator-level plan combines the optimal primitive configurations.

Recall the simplified cost model Equation (4.1) here:

$$\hat{C}_o(\mathcal{G}, \mathcal{P}_o) = \sum_{v_i \in \mathcal{V}} (q_c(v_i, p_{v_i}) + \gamma \times q_s(v_i, p_{v_i})) + \sum_{e_{ij} \in \mathcal{E}} q_r(e_{ij}, p_{v_i}, p_{v_j})$$

At the operator level, the cost functions q_c, q_s are pre-defined for the operators v according to their semantics and shapes. q_r is pre-defined to the edge $e = (u, v)$ according to the shape of the output tensor of u and one of the input tensor shapes of v, and the two tensors are actually the same tensor in the DNN model. Therefore q_r can be bound from the operator v. All these cost functions can be represented as the following equations:

$$q_x(v, p_v^\rho) = \mu_x^v \frac{\prod_{p^\rho \in \mathbb{P}_v} L(v, p^\rho)}{L(v, p_v^\rho)}, \tag{4.3}$$

where $x \in \{c, s, r\}$ and μ_x^v denotes coefficients related to q_c, q_s, q_r for operator v. They vary with different types of operators. $L(v, p^\rho)$ denotes the length (shape) of the dimension partitioned by p^ρ of v, where p^ρ is the primitive configuration (Definition 4.2.3).

Theorem 4.2.7 (Per-step optimality leads to global optimality)**.** *The optimal operator-level partition plan $\mathcal{P}_o$ can be obtained by the composition of optimal primitive configurations $p_{\mathcal{V}}^{\rho_i}$ for the set of vertices $\mathcal{V}$. which is the set of $p_v^{\rho_i}$ for all the vertices v at each step i.*

Proof. From Equation (4.1) and Equation (4.3), it can be deduced that the cost of the graph can be represented as a summation of the q_x. The minimum of the summation of independent terms is the summation of the minimum value of each term. If the per-step optimality for a single operator is proved, it could be extended to the whole graph.

Therefore, the following proof only concentrates on the per-step optimality of an operator by binding the redistribution cost q_r with its related q_c:

$$\hat{c}(v, p_v^\rho) = q_c(v, p_v^\rho) + \gamma \times q_s(v, p_v^\rho) \tag{4.4}$$

Let $L^0(v, p^\rho)$ be the original length of the dimension corresponding to the primitive configuration p^ρ, i.e. at step 0; let $f_c(p_v^i, p_v^{\rho_i})$ be the function used to calculate the number of appearances of $p_v^{\rho_i}$ in the previous step i; and let $p_v^i = \prod_i p_v^{\rho_i}$ be the configuration of v which is the composition of the primitive configurations $p_v^{\rho_i}$ chosen at step i of the recursive partitioning.

In Equation (4.3), L is the length of the dimension which would be *split* (value updation) defined in Definition 4.2.6:

$$L(v, p_v^{\rho_i}) = \frac{L^0(v, p_v^{\rho_i})}{2^{f_c(p_v^i, p_v^{\rho_i})}}$$

.

Transform Equation (4.3) with the above function L for specific step i :

$$q_x(v, p_v^{\rho_i}) = \frac{2^{f_c(p_v^i, p_v^{\rho_i})}}{2^{i+1}} \frac{\mu_x^v \prod_{p^\rho \in \mathbb{P}_v} L^0(v, p^\rho)}{L^0(v, p_v^{\rho_i})}$$

Let X_v be a new constant as follows:

$$X_v = \frac{(\mu_m^v + \mu_s^v * r)}{2} * \prod_{p^\rho \in \mathbb{P}_v} L^0(v, p^\rho)$$

By substituting the above equations into Equation (4.4), the cost function of an operator v with primitive configuration $p_v^{\rho_i}$ at step i is represented as:

$$\hat{c}(v, p_v^{\rho_i}) = q_c(v, p_v^{\rho_i}) + \gamma \times q_s(v, p_v^{\rho_i}) = \frac{2^{f_c(p_v^i, p_v^{\rho_i})}}{2^i L^0(v, p_v^{\rho_i})} X_v \tag{4.5}$$

Due to the symmetry of each workgroup, the cost function of operator v with configuration p_v^i can be calculated by summing the intra-group costs incurred by each composition primitive:

$$\hat{c}(v, p_v^i) = \sum_{j=0}^{i} 2^j \hat{C}_o(v, p_v^{\rho_j}) = \sum_{j=0}^{i} \frac{2^{f_c(p_v^i, p_v^{\rho_j})} X_v}{L^0(v, p_v^{\rho_j})}$$

Function f_c is the number of occurrences of one primitive configuration so that we can change the base of the summation from the recursion steps to the possible primitive configurations $p^\rho \in \mathbb{P}_v$:

$$\hat{c}(v, p_v^i) = X_v \sum_{p^\rho \in \mathbb{P}_v} \frac{\sum_{j=1}^{f_c(p_v^i, p^\rho)} 2^{j-1}}{L^0(v, p^\rho)} = X_v \sum_{p^\rho \in \mathbb{P}_v} \frac{2^{f_c(p_v^i, p^\rho)} - 1}{L^0(v, p^\rho)} \tag{4.6}$$

Therefore, from the formulation Equation (4.6), we can deduce that minimizing the cost of a configuration $\hat{c}(v, p_v^i)$ is equivalent to minimizing $\sum_{p^\rho \in \mathbb{P}_v} 2^{f_c(p_v^i, p^\rho)} / L^0(v, p^\rho)$. As $\sum_{p^\rho \in \mathbb{P}_v} f_c(p_v^i, p^\rho) = i$, the searching targets can be transformed to find a number of partition times for each partition primitive configurations, which are approximately proportional to the length of the related dimensions.

On the other hand, to find the per-step optimal primitive configuration $p_v^{\rho_i}$ in Equation (4.5), the target is to find the primitive configuration which minimizes $2^{f_c(p_v^i, p_v^{\rho_i})} / L^0(v, p_v^{\rho_i})$ which equals to find the primitives whose related dimension has the largest remaining length. By recursively applying the per-step optimal searching to the algorithm, it is obvious that we could find the same number of partition times for each primitive configuration approximately proportional to the dimensions.

We can conclude that the minimize goals of Equation (4.5) and Equation (4.6) are the same, so the per-step optimality can lead to global optimality is thus proved. □

4.2.4 Primitive configurations and their costs

This section aims to show how the symbolic cost functions are designed via examples of representative operators, MatMul, Convolution, and Elementwise operators.

For each primitive configuration, a cost function is defined to return the communication data quantity. The training of DNN is an iterative process that consists of forwarding and backward propagations.

Forward Propagation computes the operators with the intermediate *parameters* from input to output and calculates the *loss*, which estimates the distance between the output and the expected value. *Backward Propagation* will update the intermediate parameters from output to input based on the loss using an optimizer like Adam [13].

There exist two kinds of data communication during the DNN training:

- *Communication inside operators* is the quantity of data needed to be transferred inside an operator. It is composed of q_c and q_s. q_c is the communication quantity between two groups of devices during forward propagation. q_s denotes the communication for updating the parameters during the backward propagation process.
- *Communication between operators* is the communication quantity between two connected operators. In fact, the output tensor of the previous operator is the same as the input tensor of the second operator. However, these tensors may have different parallel plans for the two connected operators, the data may need to be redistributed. q_r models this specific communication.

Communication Inside Operators

An operator is defined by a type that describes its computational task (e.g., *Type* = *Add*, *Conv*, *MatMul*, *Relu*, etc.). It takes tensors as input and produces tensors as output. Denote $\mathbb{P} = \{p_0^\rho, p_1^\rho, \dots p_2^\rho\}$ the set of possible partition dimensions (PPDs) for each type of operator. Each PPD can be converted to the partition dimensions of all the tensors in an operator.

Define the following notions:

- $shape(tensor)$ denotes the shape of a tensor (e.g., $shape(t) = [4; 4]$).
- $shape(d)$ denotes the shape of a dimension of a tensor (e.g., $shape(d_0) = 4$).
- $input_n$ denotes an operator input tensor according to its index.
- $output$ denotes the operator output tensor of the operator.

Forward communication q_c

Communication occurs when an operator is executed on multiple devices. Each device possesses only a part of the data. Therefore, data need to be moved between devices to perform the whole computation. We detail here the most representative operators.

MatMul OP is the operator for Matrix Multiplication, described as:

$$output[i][j] \rightarrow \sum_k input_0[i][k] \times input_1[k][j].$$

Recall the set of primitive configurations of MatMul in Definition 4.2.3 here:

$$\mathbb{P} = \{p_0^\rho = \langle\langle 1, 1\rangle, \langle 1, 2\rangle\rangle, p_1^\rho = \langle\langle 2, 1\rangle, \langle 1, 1\rangle\rangle, p_2^\rho = \langle\langle 2, 1\rangle, \langle 1, 2\rangle\rangle, p_3^\rho = \langle\langle 1, 2\rangle, \langle 2, 1\rangle\rangle\}$$

If we cut along the output-independent dimension $p_3^\rho = \langle\langle 1, 2\rangle, \langle 2, 1\rangle\rangle$, a reduction still needs to occur to combine the partial results. The dimension cut is not specified for the output (represented with a diagonal dashed line on purple in Figure 4.3) but still exists. Hence, for any dimension cut, each device is responsible for computing half of the output tensor. To this end, each device preserves half of its data and communicates the other half to the other device. As we assume both communications can happen simultaneously, the communication cost will be proportional to the amount of data communicated by one of them. The 2-part cost function is as follows:

$$q_c(v, p_4^\rho) = \frac{shape(i) \times shape(j)}{2}.$$

If we cut along an output-dependent dimension p_2^ρ, partial results simply have to be concatenated. However, one input is wholly needed by each device to compute their partial result. As this input will be

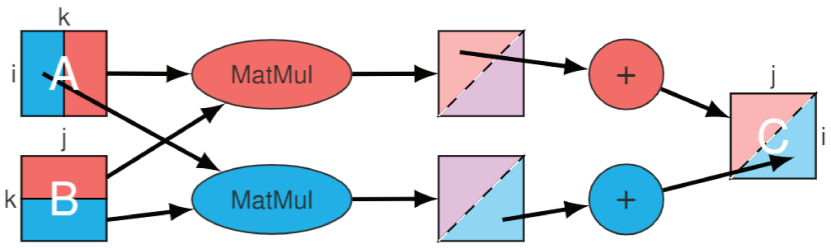


Figure 4.3: Partitioning MatMul along output-independent dimension

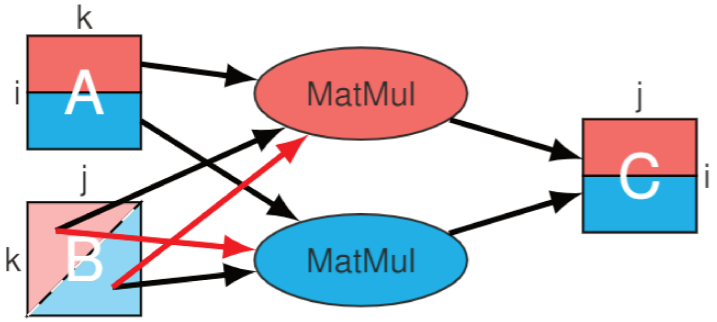


Figure 4.4: Partitioning MatMul along output-dependent dimension

cut eventually, each device must receive the half that it does not possess. The 2-part cost functions are presented below:

$$q_c(v, p_2^\rho) = \begin{cases} \frac{shape(i) \times shape(k)}{2} & \textit{if the first tensor is transferred} \\ \frac{shape(j) \times shape(k)}{2} & \textit{if the second tensor is transferred, shown in Figure 4.4} \end{cases}.$$

If we cut one of the two input tensors but duplicate another one among devices, there would be no inside communication cost. For these cases, the trade-off is situated in the q_s and q_r.

$$q_c(v, p_0^\rho) = q_c(v, p_1^\rho) = 0.$$

Conv OP represents N-dimension convolution operators. We name one of the inputs as *kernel*. b, c_i, c_o are batch, input channel, and output channel dimensions respectively. $\vec{x}$ and $\vec{z}$ are computing dimensions. $\vec{s}$ is stride and $\vec{d}$ is dilation rate. Bold italic refers to vector. The description is as follows:

$$output[b][\vec{x}][c_o] \rightarrow \sum_{\vec{z}c_i} (kernel[\vec{z}][c_i][c_o] \times input_0[b][x_0 s_0 + d_0 z_0]...[x_{n-1} s_{n-1} + d_{n-1} z_{n-1}][c_i]).$$

The set of primitive configuration $\mathbb{P}$ is represented with the name of the dimension (e.g., $p_1^\rho = b$ denotes to partition along the batch dimension of the operator as $\langle\langle 2, 1, ..., 1\rangle, \langle 1, 1, ..., 1\rangle\rangle$) because it is difficult to enumerate all the possibilities for an N-dimension operator:

$$\mathbb{P} = \{p_1^\rho = b, p_2^\rho = c_o, p_3^\rho = \forall x \in \vec{x}, p_4^\rho = \forall z \in \vec{z}, p_5^\rho = c_i, \}$$

$p_3^\rho = \forall x \in \vec{x}$: the costs of partitioning all the dimensions x belonging to $\vec{x}$ are equal so that all the dimensions are classified as one primitive configuration p_3^ρ. The same situation for p_4^ρ.

If we cut according to an output-dependent dimension:

$$q_c(p_1^\rho) = \frac{\prod shape(kernel)}{2}, \; q_c(p_2^\rho) = \frac{\prod shape(input_0)}{2},$$

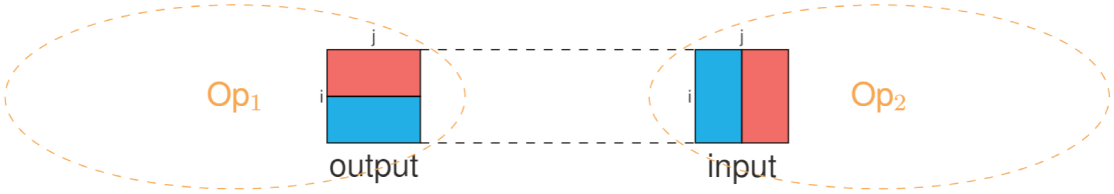


Figure 4.5: Redistribution cost between operators

$$q_c(p_3^\rho) = \frac{\prod shape(kernel)}{2} + \frac{\prod shape(input_0)}{2}.$$

If we cut according to an output-independent dimension:

$$q_c(p_4^\rho) = \frac{\prod shape(input_0)}{2} + \frac{\prod shape(output)}{2},$$

$$q_c(p_5^\rho) = \frac{\prod shape(output)}{2}.$$

Elementwise OP computes each element independently without any communication, for example, *Add*, *Sub*, *Mul*, *ReLU*, *Log*, etc. The description of *Add* is: $output[\vec{\mu}] \rightarrow input_0[\vec{\mu}] + input_1[\vec{\mu}]$ and the cost will be $\forall d \in \vec{\mu},\ q_c(d) = 0$.

Backward communication q_s

Parameter synchronization cost appears at the end of backward propagation. q_s is the communication quantity at the end of each backward propagation. Certain operators need to update one of their tensors during the training. These tensors are referred to as *Parameters*. When the *batch* dimension b is chosen, parameters hosted by each device need to be communicated to compute the average value. We group the parameter tensors as *param*. The communication quantity of a 2-part partitioning is defined $q_s(v, p^\rho) = \frac{\prod shape(param)}{2}$ when $p^\rho = b$.

Communication between operators: q_r

For two connected operators, the output tensor of the first one is the same tensor as the input tensor of the second one. If the two operators choose different partition dimensions for this tensor, it needs to be redistributed, which induces the communication cost named q_r. Figure 4.5 shows two simple situations of redistribution cost.

If the partition parallel plan of an operator's input tensor and its connected output tensor of the previous operator are equal, $q_r(e, d_0, d_0) = 0$. Otherwise, $q_r(e, d_0, d_1) = (shape(d_0) \times shape(d_1))/4$. As shown in the second part of Figure 4.5, the blue part and red part respectively represent the data stored in the first and second devices. In this situation, half of the blue part needs to be transferred to the second device while half of the red must be transferred to the first device. The cost equals the max value between them because both transfers happen simultaneously. More generally, for the typical operators in DNNs, q_r is computed from the tensor's shape.

4.3 Functional recursive cost analysis

4.3.1 Symbolic transformation based on the homomorphism

Homomorphism transformation

The cost of a vertex depends on its own configuration (computation amount q_x, intra-communication q_c, and parameter updating q_s) but also on the configuration of its neighbors (redistribution q_r). Choosing a configuration for a vertex will influence its neighbors that will influence their neighbors until the whole

graph recursively. Finding the optimal parallel plan for real-life deep neural networks is impossible in a reasonable amount of time.

The complexity of this problem is reduced by taking decisions based on local contexts and not questioning them afterward, which is a greedy method. To mitigate the difference between our cost and the optimal one, vertices are treated by order of decreasing importance. This way, the most critical vertices will have a configuration that benefits them the most. Although the less important ones may not benefit from the best configuration, the global impact on performances will be smaller. In order to justify this reordering, the algorithm is formulated as a homomorphism that presents the benefit of being computable in any order.

To do so, the following first introduces how the cost may be computed from a homomorphism of a vertex list in Section 4.3.1. However, the data-flow representation of the computation is not a list but a directed acyclic graph (DAG). Morihata [55] showed that the homomorphism theory (and especially its third theorem) might be extended to trees. The only difference between a tree and a DAG lies in the number of parents (outputs) that may be more than 1 in a DAG which leads to the existence of several possible paths from one vertex to another. To remove the existence of these different paths, HSM2DL propose to consider the spanning tree of the DAG that would select only one of those paths for each case.

Notations

The operator-level parallel plan $\mathcal{P}_o$ is the set of all parallel plans of the vertices in $\mathcal{G}$ and p_{v_i} is the parallel plan of vertex v_i in $\mathcal{P}_o$. The cost of the computational graph is shown as follows:

$$\begin{aligned} cost_{op}(v_i, p_{v_i}) &= \alpha \times q_x(v_i, p_{v_i}) + \beta \times \big(q_c(v_i, p_{v_i}) + \gamma \times q_s(v_i, p_{v_i})\big) \\ cost_{rdst}(v_i, v_j, p_{v_i}, p_{v_j}) &= \beta \times q_r(e_{ij}, p_{v_i}, p_{v_j}) \end{aligned} \tag{4.7}$$

This way, equation Equation (4.7) can be rewritten

$$\hat{C}_o(\mathcal{G}, \mathcal{P}_o) = \sum_{v_i \in \mathcal{V}} cost_{op}(v_i, p_{v_i}) + \sum_{(v_i, v_j) \in \mathcal{E}} cost_{rdst}(v_i, v_j, p_{v_i}, p_{v_j}) \tag{4.8}$$

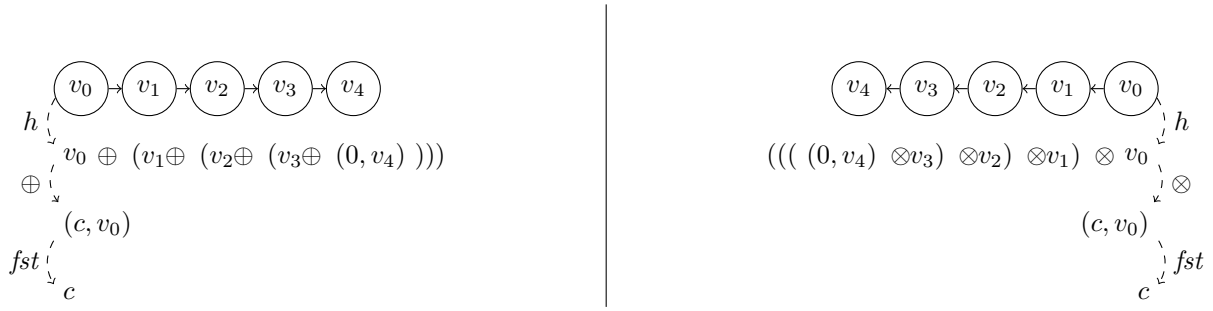
Remark that if $p_{v_j} \notin \mathcal{P}_o$ (because v_j has not been seen yet) then $cost_{rdst}(v_i, v_j, p_{v_i}, p_{v_j}) = 0$. Assume function $plan(v)$ gives the set of possible parallel plans for a given vertex v. We note $\mathcal{P}_{o\_i<|V|}$ the configurations of the first i vertices visited. The plan generation consists of taking for each vertex the parallel plan that minimizes its cost. It may be expressed recursively as

$$\begin{aligned} \mathcal{P}_{o\_0} &= \emptyset \\ \mathcal{P}_{o\_i<|\mathcal{V}|} &= search(v_i, p_{v_i}, \mathcal{G}) \\ cost_v(v_i, p_{v_i}, \mathcal{P}_o, \mathcal{G}) &= cost_{op}(v_i, p_{v_i}) + \textstyle\sum_{(v_i, v_j) \in \mathcal{E}} cost_{rdst}(v_i, v_j, p_{v_i}, p_{v_j}) \\ search(v_i, \mathcal{P}_o, \mathcal{G}) &= \mathcal{P}_o \cup \{p_{v_i}\} \text{ such that } p_{v_i} \in plan(v_i) \text{ minimizes } cost_v(v_i, p_{v_i}, \mathcal{G}, \mathcal{P}_o) \end{aligned} \tag{4.9}$$

The remaining of this section will express equations Equation (4.9) as a homomorphism. As a preliminary, the following notations are taken from [23]. $hom(\oplus, f, a)(l)$ is a homomorphism that maps function f on each element of l before reducing with the binary operator $\oplus$ whose first application will be with initialization element a. For example

$$hom(\oplus, f, a)([x, y, z]) = a \oplus f(x) \oplus f(y) \oplus f(z)$$

Let us note μ-*list*, the type of list of elements of type μ. The function will be noted in Curry notation. For example, $f : A \to B \to C$ is the function f that, when applied to an argument of type A, will produce a function of type $B \to C$ that when applied to an argument of type B will produce a value of type C. Denoting $+\!\!+ : \mu\text{-}list \to \mu\text{-}list \to \mu\text{-}list$ as the concatenation operator. The function composition is noted $(f \circ g)(x) = f(g(x))$. To access elements of a pair, we use function first $fst(x, y) = x$ and function second $snd(x, y) = y$



(a) Leftward processing of $cost_{all\_rdst}$

(b) Rightward processing of $cost_{all\_rdst}$

Figure 4.6: Leftward and rightward processing of $cost_{all\_rdst}$ over a vertex list

Redistribution cost as a homomorphism

We define the leftward operation $cost_{all\_rdst}$ to compute all redistribution costs of a linear graph (list). A leftward operation is a recursive operation for which the elements are added to the left (input) side (see fig. 4.6(a)). To be a leftward operation, $cost_{all\_rdst}$ needs to be defined in the form

$$\begin{aligned} & h([v] +\!\!+ x) = v \oplus h(x) \\ & \text{with} \\ & \qquad h : \mu\text{-}list \to \omega\text{-}list \\ & \qquad \oplus : \mu \to \omega \to \omega \end{aligned}$$

The form with the operation $cost_{all\_rdst}$ leftward is instantiated as follows

$$\begin{aligned} cost_{all\_rdst} &= fst \circ h \\ h([v] +\!\!+ x) &= v \oplus h(x) \\ h([v_0]) &= (0, v_0) \\ v_i \oplus (c, v_j) &= (c + cost_{rdst}(v_i, v_j, p_{v_i}, p_{v_j}), v_i) \end{aligned} \tag{4.10}$$

Suppose that $\mathcal{P}_o$ such that $p_{v_i}, p_{v_j} \in \mathcal{P}_o$, is a global constant fixed for the whole cost computation. As the direction of the edges has no influence on the redistribution cost, the function $cost_{all\_rdst}$ is also computable rightward (in the output direction, as illustrated in Figure 4.6(b)). A rightward operation must respect the form

$$\begin{aligned} & h(x +\!\!+ [v]) = h(x) \otimes v \\ & \text{with} \\ & \qquad h : \mu\text{-}list \to \omega\text{-}list \\ & \qquad \otimes : \omega \to \mu \to \omega \end{aligned}$$

The form with the operation $cost_{all\_rdst}$ rightward is instantiated as follows

$$\begin{aligned} cost_{all\_rdst} &= fst \circ h \\ h(x +\!\!+ [v]) &= h(x) \otimes v \\ h([v_0]) &= (0, v_0) \\ (c, v_j) \otimes v_i &= (c + cost_{rdst}(v_i, v_j, p_{v_i}, p_{v_j}), v_i) \end{aligned} \tag{4.11}$$

Thus, by the third homomorphism theorem [23], $cost_{all\_rdst}$ is a homomorphism because it is defined both leftward (Equation (4.10)) and rightward (Equation (4.11)).

Cost and plan generation as a homomorphism

The intra-communication cost is defined directly as the homomorphism

$$cost_{all\_op} = hom(+, cost_{op}, 0)$$

Hence, the whole cost of the linear graph ($cost_l$) may be defined as the addition of the two homomorphisms

$$cost_l(l) = cost_{all\_op}(l) + cost_{all\_rdst}(l)$$

The two may also be computed together as

$$\begin{array}{lcl} cost_l & = & fst \circ h \\ h([v] +\!\!+ x) & = & v \oplus h(x) \\ h([v_0]) & = & (cost_{op}(v_0, p_{v_0}), v) \\ v_i \oplus (c, v_j) & = & (c + cost_{rdst}(v_i, v_j, p_{v_i}, p_{v_j}) + cost_{op}(v_i, p_{v_i}), v_i) \end{array} \tag{4.12}$$

and symmetrically for the rightward notation.

$$\begin{array}{lcl} cost_l & = & fst \circ h \\ h(x +\!\!+ [v]) & = & h(x) \otimes v \\ h([v_0]) & = & (cost_{op}(v_0, p_{v_0}), v) \\ (c, v_j) \otimes v_i & = & (c + cost_{rdst}(v_i, v_j, p_{v_i}, p_{v_j}) + cost_{op}(v_i, p_{v_i}), v_i) \end{array} \tag{4.13}$$

In this case, the third homomorphism theorem tells us that $cost_l$ is a homomorphism because it is defined both leftward (Equation (4.12)) and rightward (Equation (4.13)). Now that we have shown how the cost may be formulated as a homomorphism, the plan generation may be in turn written leftward

$$\begin{array}{lcl} cost_l & = & fst \circ h \\ h([v] +\!\!+ x) & = & v \oplus h(x) \\ h([v_0]) & = & \big(search(v_0, \emptyset, [\,]), [v_0]\big) \\ v_i \oplus (\mathcal{P}_o, L) & = & \big(search(v_i, \mathcal{P}_o, L), v_i +\!\!+ L\big) \end{array} \tag{4.14}$$

and rightward

$$\begin{array}{lcl} cost_l & = & fst \circ h \\ h(x +\!\!+ [v]) & = & h(x) \otimes v \\ h([v_0]) & = & \big(search(v_0, \emptyset, [\,]), [v_0]\big) \\ (\mathcal{P}_o, L) \otimes v_i & = & \big(search(v_i, \mathcal{P}_o, L), v_i +\!\!+ L\big) \end{array} \tag{4.15}$$

The plan generation being a homomorphism allows us to state that the vertex list (linear graph) may be processed in any order. This homomorphism may be extended to trees thanks to the work of Morihata [55] thereby enabling us to consider a much more complex graph topology than a mere vertex list.

Tree homomorphism

For the sake of understanding, this section will simply consider homomorphisms of binary trees defined in Equation (4.16).

$$\textbf{data}\ T_{bin} = Node\ (V \times T_{bin} \times T_{bin}) \mid Leaf \tag{4.16}$$

In order to define homomorphisms, we need to be able to describe contexts and paths in the tree. These will be represented thanks to a zipper [33]. A path in the binary tree would be a sequence of left or right choices that we represent in Equation (4.17).

$$\textbf{data}\ zip = \big(Left\ (V \times T_{bin})\ \mid\ Right\ (V \times T_{bin})\big)\ list \tag{4.17}$$

Applying the third theorem of homomorphism on trees requires the definition of an upward and downward computation. The downward version is represented Equation (4.18) and the upward version in Equation (4.19).

$$
\begin{array}{lcl}
cost_{\downarrow} & = & fst \circ h_{\downarrow} \\
h_{\downarrow}([\,]) & = & (\emptyset, Leaf) \\
h_{\downarrow}\big([Left(v_i, l)] +\!\!+ x\big) & = & \textbf{let } (\mathcal{P}_o, T) = h_{\downarrow}(x) \\
 & & \textbf{in } \big(search(v_i, \mathcal{P}_o, T), Node\ (v_0, l, T)\big) \\
h_{\downarrow}\big([Right(v_i, r)] +\!\!+ x\big) & = & \textbf{let } (\mathcal{P}_o, T) = h_{\downarrow}(x) \\
 & & \textbf{in } \big(search(v_i, \mathcal{P}_o, T), Node\ (v_0, T, r)\big)
\end{array}
\tag{4.18}
$$

$$
\begin{array}{lcl}
cost_{\uparrow} & = & fst \circ h \\
h(x +\!\!+ [v]) & = & h(x) \otimes v \\
h([\,]) & = & (\emptyset, Leaf) \\
(\mathcal{P}_o, T) \otimes [Left(v_i, l)] & = & \big(search(v_i, \mathcal{P}_o, T), Node\ (v_i, l, T)\big) \\
(\mathcal{P}_o, T) \otimes [Right(v_i, r)] & = & \big(search(v_i, \mathcal{P}_o, T), Node\ (v_i, T, r)\big)
\end{array}
\tag{4.19}
$$

Remark that going from a binary tree to any (bounded) degree trees requires a minor adjustment to the tree, zipper, and computation definition by adding the required number of cases.

Extension from trees to DAGs

Trees differ from DAGs in the sense that a node may have several parents in the case of a DAG. To address this issue, we treat the DAG as its spanning tree (only picking one parent of each node) while computing the redistribution cost with respect to all of the DAG edges. The number of parents (inputs) of each node can easily be bounded by the maximum number of inputs an operator in neural networks can have. This allows us to treat any DAG cases, but our proof that the reordering of the vertices preserves the cost cannot be extended to general DAGs in their current form.

4.3.2 Flex-Edge Recursive graph

This section introduces how we take advantage of the homomorphism-based symbolic transformation presented in the previous section. Homomorphism theory states that the order of operators can be arbitrarily swapped when computing the cost of the whole graph. This section proposes a topology-independent graph structure named Flex-Edge Recursive graph(FER graph), which is used for the cost analysis of the neural network parallel plan. FER graph supports arbitrary reordering of the operators. The costs of different parallel plans on vertices and edges can be computed through the defined *concatenation* rules, and finally, the parallel plan can be decided. With the reordering feature of the Flex-Edge graph, backtracking when traversing the graph is eliminated and thus reducing the searching complexity from exponential to linear w.r.t. the number of operators.

FER Graph

Let us recall the definition of a computational graph here:

A computation graph is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a set of *Vertices* and $\mathcal{E}$ is a set of *Edges*. A vertex $v \in V$ is an operator. An edge e is defined as a tuple (u, v) where $u, v \in V$.

Definition 4.3.1. Let σ denote an order on the vertices $v \in \mathcal{V}$ such that $\sigma_i(\mathcal{V}) \in \mathcal{V}$ is the i^{th} visited vertex. From σ order, the Flex-Edge Recursive Graph (FER Graph) $\mathcal{G}_f$ can be redefined as

$$\mathcal{G}_f = (\sigma(\mathcal{V}), \mathcal{E})$$

Associated with the FER graph $\mathcal{G}_f$ a list of sub-graphs is defined in order to establish a traversal rule. This list is built thanks to a concatenation operator denoted $+\!\!+$.

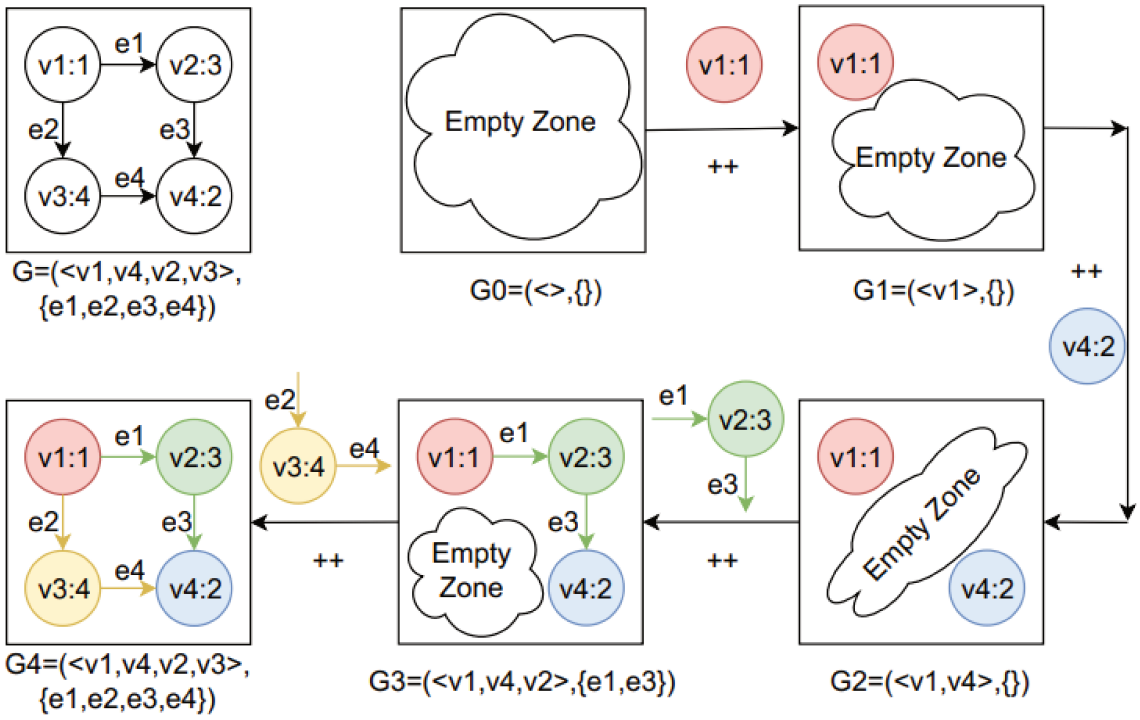


Figure 4.7: Flex-Edge Graph Traversal: the number after the column denotes the order of the vertex. $v2:3$ means $v2$ is ordered at the third place.

Definition 4.3.2. Let the FER graph $\mathcal{G}_f = (\sigma(\mathcal{V}), \mathcal{E})$, the list of sub-graphs $\mathbb{G} = [\mathcal{G}_f^i = (\sigma(\mathcal{V}_i), \mathcal{E}_i)]$ is defined as $\mathcal{G}_f^0 = (\langle\rangle, \{\})$ and $\mathcal{G}_f^i = \mathcal{G}_f^{i-1} +\!\!+ \sigma_i(\mathcal{V})$ with

$$\left|\begin{array}{c} \mathcal{V}_i = \mathcal{V}_{i-1} \cup \sigma_i(\mathcal{V}) \\ \mathcal{E}_i = \mathcal{E}_{i-1} \cup \bar{\mathcal{E}}_i \\ \bar{\mathcal{E}}_i = \{e_j \in \mathcal{E} \mid j < i, e_j = (\sigma_i(\mathcal{V}), \sigma_j(\mathcal{V}))\} \\ \cup \{e_j \in \mathcal{E} \mid j < i, e_j = (\sigma_j(\mathcal{V}), \sigma_i(\mathcal{V}))\} \end{array}\right.$$

Figure 4.7 illustrates an example where the upper left corner is a FER Graph with ordered vertices $\langle v_1, v_4, v_2, v_3\rangle$. The traversal is a process of reconstructing the original FER Graph from an empty one. The vertex v_1 is added first. None of the other vertices connected to $v1$ is added, so no edge is added to $\mathcal{G}_f^1$, then vertex v_4 is added. Similarly, no edge is together visited with v_4. When v_2 is added, their neighbors v_1 and v_4 are already in the graph. Therefore, e_1, e_3 are added with v_2. After all the concatenations, $\mathbb{G} = [(\langle\rangle, \{\}), (\langle v_1\rangle, \{\}), (\langle v_1, v_4\rangle, \{\}), (\langle v_1, v_4, v_2\rangle, \{e_1, e_3\}), \mathcal{G})]$.

Traversing Order

Let $\mathcal{G}_f^{i-1}, \mathcal{G}_f^i \in \mathbb{G}$ such that $\mathcal{G}_f^i = \mathcal{G}_f^{i-1} +\!\!+ \sigma_i(\mathcal{V}) = (\sigma(\mathcal{V}_i), \mathcal{E}_i)$, let q_m the communication quantity inside an operator: $q_m(\sigma_i(\mathcal{V}), p^\rho_{\sigma_i(\mathcal{V})}) = q_c(\sigma_i(\mathcal{V}), p^\rho_{\sigma_i(\mathcal{V})}) + \gamma \times q_s(\sigma_i(\mathcal{V}), p^\rho_{\sigma_i(\mathcal{V})})$.

Definition 4.3.3 (Cost function for operators in FER Graph)**.**

$$\hat{c}(\sigma_i(\mathcal{V}), \bar{\mathcal{E}}_i, p^\rho_{\sigma_i(\mathcal{V})}) = q_m(\sigma_i(\mathcal{V}), p^\rho_{\sigma_i(\mathcal{V})}) + \sum_{e \in \bar{\mathcal{E}}_i} q_r(e, p^\rho_{\sigma_i(\mathcal{V})})$$

Before discussing the traversal order, we first define the function *OptP*() to compare the parallel plans' quality and choose the minimum one. The function takes a vertex and its together visiting edges as inputs, then searches the possible partition dimensions and finds the optimal configuration, which minimizes the communication cost.

Definition 4.3.4 (Function *OptP*()). *OptP*$(\sigma_i(\mathcal{V}), \mathcal{E}_i)$ returns the chosen primitive configuration p^ρ_{opt} which minimizes the cost $\hat{c}$:

$$OptP(\sigma_i(\mathcal{V}), \mathcal{E}_i) = p^\rho_{opt} \ s.t. \ \min_{p^\rho \in \mathbb{P}} \hat{c}(\sigma_i(\mathcal{V}), \bar{\mathcal{E}}_i, p^\rho) \ \textit{is reached for} \ p^\rho = p^\rho_{opt}$$

The idea of our traversal order is to find the optimal parallel plan for the new sub-graph $\mathcal{G}^i_f$ when concatenating a vertex $\sigma_i(\mathcal{V})$ to a sub-graph $\mathcal{G}^{i-1}_f$. So that by finding the optimal parallel plan for every sub-Graph recursively, we can ensure the near-optimal parallel plan for the whole graph.

For a vertex v, p^ρ_{mopt} denotes a dimension in $\mathbb{P}$, such that

$$q_m(v, p^\rho_{mopt}) = \min_{p^\rho \in \mathbb{P}} q_m(v, p^\rho)$$

If there is no redistribution cost between $\sigma_i(\mathcal{V})$ and $\mathcal{G}^{i-1}_f$ (i.e., $q_r = 0$), the optimal parallel plan of $\mathcal{G}^i_f$ is the union of the optimal parallel plan of $\mathcal{G}^{i-1}_f$ and the p^ρ_{mopt} of $\sigma_i(\mathcal{V})$. However, if the redistribution cost q_r is large, either $\sigma_i(\mathcal{V})$ or $\mathcal{G}^{i-1}_f$ needs to change its configuration.

In order to avoid backtracking, define the order $\sigma(\mathcal{V})$ ensures that it is always the configuration of $\sigma_i(\mathcal{V})$ that needs to be changed. This change of configuration is referred to as a *compromise*.

Recall that q_r is either 0 or a fixed positive value. The *compromise* consists in changing the p^ρ_{mopt} to a parallel plan p^ρ_{ropt} *s.t.* $q_r = 0$. In this way, the price of reducing an operator's q_r to zero is the increment of its q_m. Therefore, the *compromise price* of an operator (i.e., the price to change the configuration of an operator) is defined as $\lambda_{\sigma_i(\mathcal{V})} = q_m(p^\rho_{ropt}) - q_m(p^\rho_{mopt})$. The order $\sigma(\mathcal{V})$ of the operators is in descending order of their *compromise price* $\lambda_{\sigma(\mathcal{V})}$.

Definition 4.3.5. Let $\mathcal{G}_f = (\sigma(\mathcal{V}), \mathcal{E})$ a FER graph where $|\mathcal{V}| = n$, such that

$$\forall \ 1 \leq j < k \leq n, \ \sigma_j(\mathcal{V}) \ \textit{is ordered before} \ \sigma_k(\mathcal{V}) \ \textit{if} \ \lambda_{\sigma_j(\mathcal{V})} < \lambda_{\sigma_k(\mathcal{V})}$$

The list of sub-graphs of $\mathcal{G}_f$ is referred as $\mathbb{G} = [\mathcal{G}^i_f = (\sigma(\mathcal{V}_i), \mathcal{E}_i)]$, where $1 \leq i \leq n$.

Definition 4.3.6 (compromise price). The *compromise price* of the sub-graph $\mathcal{G}^{i-1}_f$ is $\lambda_{\mathcal{G}^{i-1}_f}$.

It is obvious that $\lambda_{\mathcal{G}^{i-1}_f} \geq \lambda_{\sigma_{i-1}(\mathcal{V})} \geq \lambda_{\sigma_i(\mathcal{V})}$. As a result, if we can order the vertices in descending order according to its *compromise price*, the minimized communication cost can be guaranteed.

However, it is not trivial to find the p^ρ_{ropt} because q_r relies on the connected vertices. It seems that we return to the original complexity problem, but the features of DNN help us handle it. Actually, what we really need is the value of $q_m(p^\rho_{ropt}))$ instead of p^ρ_{ropt}. For typical operators, we can find their *compromise price* λ because of the characteristics of their semantics.

MatMul OP The primitive configuration of MatMul needs to *compromise* when its p^ρ_{mopt} leads to a large q_r. However, no matter $p^\rho_{ropt} = \forall p^\rho \in \mathbb{P}$, when it *compromises* to the other two dimensions, $q_r = 0$. The *compromise price* of MatMul is defined as $\lambda = \min_{p^\rho \in \mathbb{P} - p^\rho_{mopt}} q_m(p^\rho) - q_m(p^\rho_{mopt})$.

Conv OP Although Conv has many possible partition dimensions, in current real Convolution Neural Networks (e.g., VGG[77], ResNet[25]), only batch dimension b and input channel dimension k will be actually chosen. The reason is that in a DNN, the size of the kernels is very small, so partitioning kernel tensor usually leads to a super large communication cost. Besides, the channel number increases from input to output of DNN, so the size of the output tensor is always much bigger than the input. As there remain only two possible partition dimensions, let p^ρ_0 denotes the other dimension except p^ρ_{mopt}. The *compromise price* is defined as $\lambda = q_m(p^\rho_0) - q_m(p^\rho_{mopt})$.

Elementwise OP q_m of Elementwise OP is always 0. It is evident they do not have *compromise price*. When an Elementwise OP is located between two operators that have redistribution costs between them. This elementwise OP can compromise its plan to any of its neighbors, so the q_r for both sides is 0. However, this elementwise OP has zero redistribution cost with the neighbor that it compromises to but may have a redistribution cost with another neighbor. As a result, the real redistribution cost between the two no-elementwise operators will not be correctly considered. To avoid this problem, Elementwise OPs are eliminated before the parallel plan searching. They will reuse one of the neighbor's configurations.

Other OP Except MatMul, Conv, and Elementwise OP, all the other operators (MaxPool, ReduceMean, ReduceSum, ReduceMax, Squeeze... etc.), we noticed in the real DNNs, may have multiple

dimensions but they only have two values of q_m. In other words, q_m of several dimensions has the same value. Let p_0^ρ denote the dimension which has a different q_m as p_{mopt}^ρ. The *compromise price* is defined as $\lambda = q_m(p_0^\rho) - q_m(p_{mopt}^\rho)$.

4.4 Double recursive algorithm

Based on the recursive partitioning Section 4.2.2 and FER Graph Section 4.3.2, this section proposes the double recursive algorithm (D-Rec algorithm). Algorithm 1 describes the D-Rec algorithm composed of *Inner Recursion* and *Outer Recursion*. The traversing of the FER Graph is called Inner Recursion, which takes charge of the choice of a dimension in each vertex to partition it into two parts. At the same time, Outer Recursion is responsible for extending this 2-part partitioning to all devices. Both the two recursions are functional recursions and represented in a *for-loop* format.

Outer Recursion takes a FER Graph $\mathcal{G}_f$ with an empty parallel plan and the number of partition times n as inputs and returns the parallel plan assigned Graph as the output. The initial n is obtained from the logarithm of the number of devices $n = log_2 N$. The function *Reorder* sorts the vertex in FER Graph $\mathcal{G}_f$ according to the *compromise price* (see Section 4.3.2). At each Outer Recursion step, all the operators in the graph are partitioned into two parts with Inner Recursion. The function *ShapeUpdate* updates the *Shape* of each *Vertex* in G_f according to the chosen parallel plan with *split()* (Definition 4.2.6). N is decreased by one at each recursion step. Outer Recursion ends when $N = 0$.

Inner Recursion takes the sub-graph list $\mathbb{G}$ and an empty FER Graph $\mathcal{G}_{f_{in}}$ as inputs at each Outer Recursion step. *pop_end*() denotes the operation on $\mathcal{G}_f$ that pops the last graph in the list: $\mathcal{G}_f = pop\_end(\mathbb{G}), \mathbb{G} \leftarrow \mathbb{G} - \mathcal{G}_f$. In Algorithm 1, $v_{\mathcal{G}_f}$ denotes the visited vertex to construct $\mathcal{G}_f$ from its predecessor and $\bar{\mathcal{E}_{\mathcal{G}_f}}$ denotes the added new edges. At each step of Inner Recursion, a sub-graph $\mathcal{G}_f$ is popped, and the configurations of its vertices is chosen by $OptP(v_{\mathcal{G}_f}, \bar{\mathcal{E}_{\mathcal{G}_f}})$ according to the symbolic cost model. The reconstructed Graph $\mathcal{G}'_{f_{in}}$ is composed by concatenating the configuration updated vertex $v_{\mathcal{G}_f}$. The process is recursively applied on the sub-graph list $\mathbb{G}$. The recursion ends when all vertices have been visited.

Algorithm 1 Double Recursive Algorithm

Require: Initial FER Graph $\mathcal{G}_f$. The number of partition times $n = log_2 N$.
Ensure: Operator parallel plan $\mathcal{P}_o$.

function OuterRecursion($\mathcal{G}_f, n$)
  if $n = 0$ **then**
    return $\mathcal{G}_f$
  else
    $(\sigma, \mathbb{G}) = Reorder(\mathcal{G}_f)$
    $\mathcal{G}_{f_{in}} =$ InnerRecursion($\mathbb{G}, (\varnothing, \varnothing)$)
    $\mathcal{G}'_f = ShapeUpdate(\mathcal{G}_{f_{in}})$ // updates the vertices' shape according to the chosen parallel plan
    return OuterRecursion($\mathcal{G}'_f, n-1$)
  end if
end function
function InnerRecursion($\mathbb{G}, \mathcal{G}_{f_{in}}$)
  if $\mathbb{G} = \varnothing$ **then**
    return $\mathcal{G}_{f_{in}}$
  else
    $\mathcal{G}_f = pop\_end(\mathbb{G})$ // *pop_end*() will pop the last FER graph $\mathcal{G}_f$ from the list $\mathbb{G}$
    $p_{opt}^\rho = OptP(v_{\mathcal{G}_f}, \bar{\mathcal{E}_{\mathcal{G}_f}})$
    $\mathcal{P}_o \rightarrow p_{v_{\mathcal{G}_f}}^\rho \mathrel{+}= p_{opt}^\rho$
    $\mathcal{G}'_{f_{in}} = \mathcal{G}_{f_{in}} +\!\!+ v_{\mathcal{G}_f}$
    return InnerRecursion($\mathbb{G}, \mathcal{G}'_{f_{in}}$)
  end if
end function

4.5 Experiments

This section aims to evaluate the search efficiency and the quality of the parallel plans found by D-Rec for operator-level searching. The analysis of HSM2DL is currently based on synchronous distributed training, which ensures that the loss and accuracy in each training step are the same as in the stand-alone training. On the other hand, the D-Rec algorithm only finds the theoretically near-optimal parallel plans, and the DL framework executes the real distributed parallel training. The competitive methods are also trained on the same DL framework, so even if the framework's implementation impacts accuracy, the impact is the same for both. Therefore, it is not needed to compare the accuracy when we conduct the experimental analysis again. The experimental results also show that the parallel plan has no effect on the accuracy under our experimental framework.

4.5.1 Environment setup

DNN models:

The experiments are executed on real-world DNN models:

- Computer Vision:
  - ResNet50 with Cifar10 dataset,
  - ResNet50/101/152 [25] with ImageNet dataset,
  - Fully Convolution Network (FCN) [48];
- Recommendation Systems:
  - Wide&Deep [94] with Criteo dataset;
- Neural Language Processing:
  - BERT [18],
  - PanGu-Alpha 2.6B and 13B [95] (B stands for billion, which signifies the size of parameters),
  - T5 [65] (Text-to-Text Transfer Transformer) with dedicated text dataset.

Evaluation metric:

Training throughput, often defined as the capacity of processing *Items Per Second* (IPS), is used to evaluate the quality of a parallel plan. The higher IPS denotes that the training process can compute more items within one second which means better performance.

Hardware environments:

The experiments are conducted on an Atlas900 AI cluster [47]. Each Atlas node is composed of eight Ascend910 accelerators. The operator-level experiments test until 64 accelerators where the four nodes are connected with a 64-port switch. All the Ascend910 clusters are interconnected directly, even from a different node. An 8 NVIDIA-V100 GPU cluster is also implemented as a control group to show the better portability of HSM2DL and D-Rec algo. All GPUs of a node communicate with each other via the PCIe (e.g., Figure 4.1).

Deep Learning Framework:

The D-Rec algorithm[1] is implemented on the SOTA DL framework MindSpore[2]. The experiments are conducted on MindSpore, which supports automatically distributed training with a given plan. The parallel plans found by the D-Rec algorithm will be taken as input for Mindspore, and the training will be conducted on this DL framework.

[1] `https://github.com/mindspore-ai/mindspore/tree/master/mindspore/ccsrc/frontend/parallel/auto_parallel/rec_core`

[2] `https://www.mindspore.cn/en`

Baseline:

The research of automatic parallel plan search is still in its early stages: the generality of related methods is not complete, such as OptCNN, Flexflow, and TensorOpt's search algorithms (described in Section 2.4.3) can only handle linear computational graphs; in addition, on classical neural networks, it is difficult for automatic parallel plan search to find manual parallel plans that experts have intensively studied for months. Therefore, the comparison baseline chosen in this section is the Expert-Designed parallel plan. The following lists the **Expert-Designed parallel plans** used for different networks:

- **ResNet:** For all variants of ResNet, parallel plans are respectively analyzed for the convolutional and fully connected layers by taking into account the cost of redistribution. Fine-tuned OWT [37] for networks with different layers and input data is chosen as the Expert-Designed parallel plan here.
- **FCN:** The FCN network tested here is trained with a high-precision image such that model parallel (operator-level) should have more importance. For this high-precision-image-targeted FCN model, there is no published paper studying how to parallel train it. What we have implemented here is one Expert-Designed parallel plan that related researchers have studied for more than one month (it is confidential work without publishment).
- **Wide&Deep:** HugeCTR [60] is a dedicated distributed training framework design developed by NVIDIA that can support typical CTR network deployments such as Wide & Deep [94]. The Expert-Designed parallel plan here follows the HugeCTR idea.
- **BERT, T5:** Here implements a fine-tuned expert-defined parallel plan based on Megatron-LM [74], the well-known transformer-based manual parallel plan.
- **PanGu-alpha:** Pangu-alpha's Expert-Designed parallel plan is introduced in its published paper [95].

Competitive approach:

To show the better portability of HSM2DL and the D-Rec algorithm compared with the previous approaches, we choose TensorOpt [12] as the competitive approach. TensorOpt is a representative approach proposed in 2021. The dynamic programming searching algorithm of TensorOpt can be regarded as the SOTA approach for operator-level search. The dynamic programming search algorithm compares the possible configurations for operator-level parallelism, making it efficient for relatively small DNNs.

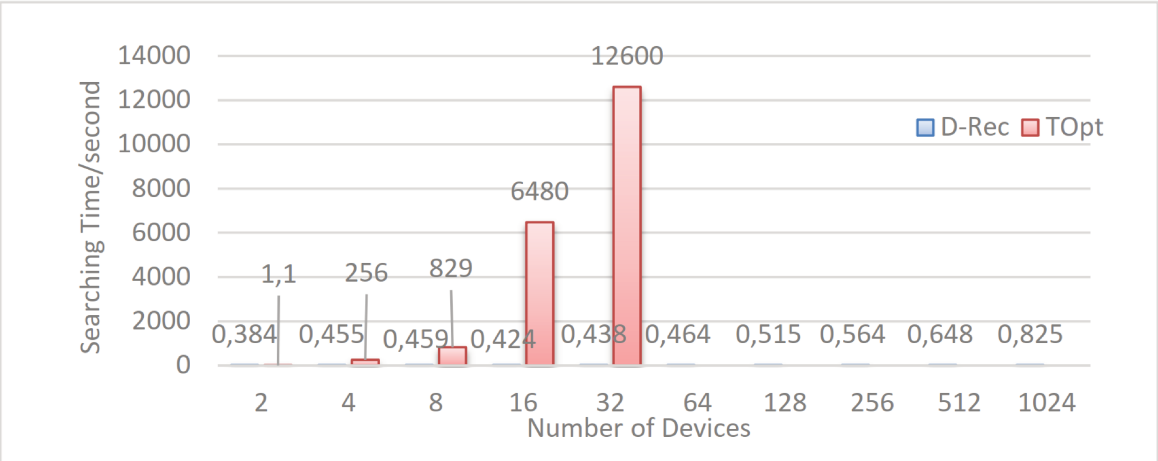


Figure 4.8: Searching Time of ResNet101

4.5.2 Searching efficiency

This section aims to prove the efficiency and linear time complexity of HSM2DL and the D-Rec algorithm. ResNet101 [25] and BERT [18], two representative DNNs, are taken to validate the parallel plan searching speed of D-Rec.

First, to show the search efficiency w.r.t. the number of devices, the computation graph of ResNet101 was fixed, and the number of devices varied from 2 to 1024 (Figure 4.8). The searching time of D-Rec on ResNet101 increases linearly from 0.384 seconds to 0.825 seconds. The dynamic programming of TensorOpt took nearly 2 hours to find a plan for 16 devices, and 3.5 hours for 32 devices, and failed to find any plan for 64 devices after hours. It can also be conducted from Figure 4.8 that the searching time of the DP algorithm increases exponentially w.r.t the number of devices. On the contrary, The searching time of D-Rec remains small and keeps the linear increase tendency.

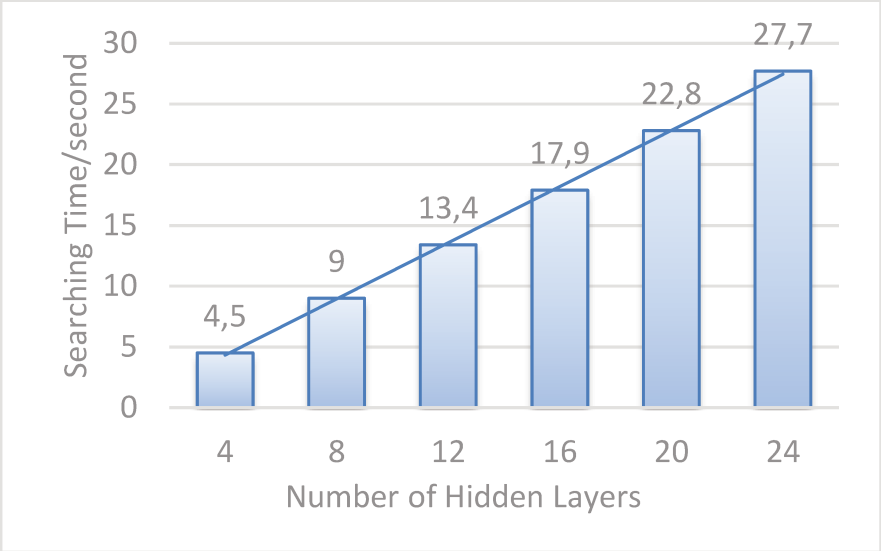


Figure 4.9: Searching Time of BERT

To validate the linearity of the searching time w.r.t. the number of operators, we conducted the following experiment: we fixed the number of devices to 8. It varied the number of hidden layers of BERT from 4 to 24 since the number of operators is in proportion to the number of hidden layers. The results are displayed in Figure 4.9 The searching time of D-Rec on the variants of BERT is between 4.5 sec and 27.7 sec. The searching time of DP hasn't been put in the Figure 4.9 because this algorithm does not work on these multi-input graph networks (non-linear graph). The experiments showed that D-Rec could handle general large computation graphs in a few seconds with a linear growth trend.

4.5.3 Parallel plan quality

Generality

An extensive range of real-world DNN models is tested to evaluate the quality of the hybrid parallel plan found by the D-Rec algorithm of HSM2DL. Table 4.1 shows the average step times of training varieties of DNN models with Expert-Designed parallel plans compared to the plans found by D-Rec. This table does not include the results of TensorOpt's DP algorithm for two reasons. First, the baseline of Expert-Designed parallel plans has a high plan quality because of the efforts spent by the researchers. Secondly, the DP algorithm actually lacks generality: 1) it cannot deal with non-linear graphs; 2) it requires manually configuring the search space for a new unknown DNN model.

The last column of Table 4.1 shows the performance percentage of D-Rec's founded plans to those of Baseline (high than 100% denotes a better performance). It can be found that the minimum performance percentage of D-Rec to the Baseline is 90.40% for the BERT model. The experiments show that D-Rec

Performance: step time/ms (8 Ascends)				
DNN models		Baseline	HSM2DL	Percentage
CV	ResNet50-cifar	48.58	45.91	**105.83%**
	ResNet50-ImageNet	57.53	61.18	**94.03%**
	ResNet101-ImageNet	86.73	93.38	**92.88%**
	ResNet152-ImageNet	120.57	127.46	**94.59%**
	FCN	485	512	**94.72%**
Rec.Sys.	Wide&Deep	21.6	22.38	**96.51%**
NLP	BERT	110.63	122.38	**90.40%**
	PanGu-Alpha 2.6B	4826	4876	**98.91%**
	PanGu-Alpha 13B	13990	13988	**100.01%**
	T5	1288	1279	**100.70%**

Table 4.1: Performance on varieties of DNN models

can find good hybrid parallel plans for varieties of real-world DNN models with a correct performance higher than 90% than the Expert-Designed parallel plans. The -10% variations are not statistically significant and can be said to be a good performance. Even though the results of TensorOpt are not given in Table 4.1, it can be deduced from Table 4.2 that a bad parallel plan leads to more than -60% performance decrease.

Portability

The experiments here demonstrate the portability of the HSM2DL algorithm and the competitive method when the training environments are changed. For TensorOpt, profiling the operators under different parallel configurations of typical DNN models usually takes more than one day. On the contrary, for HSM2DL, thanks to the recursive partitioning, the profiling tasks are avoided for operator-level searching. In Table 4.2 and Table 4.3, TensorOpt kept one profiling base and varied the training configurations. The results of TensorOpt show the impact of its profiling data. Two typical DNN models, ResNet152-ImageNet and Wide&Deep, are chosen to be evaluated because TensorOpt does not support transformer-based DNN models whose graph structure is linear.

Performance: Percentage w.r.t the Baseline			
DNN models	Dev. Num.	TensorOpt (Profiled with 8 Ascend)	HSM2DL
ResNet152-ImageNet	8	**93.16%**	**94.59%**
	16	**79.15%**	**110.74%**
	32	**63.25%**	**90.17%**
	64	**42.12%**	**91.22%**
Wide&Deep	8	**96.23%**	**96.51%**
	16	**77.96%**	**98.15%**
	32	**59.93%**	**102.82%**
	64	**39.12%**	**99.26%**

Table 4.2: Portability w.r.t. the scale of cluster

Table 4.2 shows the percentage performance of TensorOpt and HSM2DL w.r.t. the number of cluster devices. For both ResNet and Wide&Deep, it can be easily conducted that with the increase in device numbers, the quality of parallel plans found by TensorOpt decreases because they do not have enough profiling data on the possible partition dimensions, so they miss the optimal choices. However, the profiling-free approach HSM2DL can keep a good parallel plan quality because of the leveraged profiling time.

The same interpretation can be made from Table 4.3 that searching the parallel plan with different profiled data from a different architecture for TensorOpt, the decrease of parallel plan quality is evident, while HSM2DL keeps good results. The cost model of TensorOpt is based on the profiled execution

Performance: Percentage w.r.t the Baseline				
Profiling Base	DNN models	TensorOpt (8 Ascends)	TensorOpt (8 GPUs)	HSM2DL
8 GPUS	ResNet152-ImageNet	**62.12%**	**99.18%**	**99.53%**
	Wide&Deep	**49.55%**	**98.25%**	**98.33%**
8 Ascend	ResNet152-ImageNet	**98.16%**	**71.56%**	**94.59%**
	Wide&Deep	**97.23%**	**66.89%**	**96.51%**

Table 4.3: Portability w.r.t. hardware architecture

time of operators on the actual hardware. The execution time of an operator with the same parallel plan is different on GPUs and Ascends. That is why the parallel plan quality of TensorOpt decreases when executed on GPUs with profiling data on Ascends. Heavy profiling tasks (a few days) limit the portability of these profiling-based approaches, while HSM2DL is more practical thanks to the symbolic transformation and reduction to eliminate the requirement of the profiling data.

4.6 Conclusion

This chapter described how to apply HSM2DL for operator-level parallel plan search. Firstly, the proposed symbolic cost model is reduced using the symmetric multi-order property of abstract machines and deep learning domain-specific knowledge. Recursive partitioning is proposed to compress the search complexity associated with the number of devices to linearity. Also, it is shown that the computation of the cost model on computational graphs can be implemented in any order from the third homomorphism theory. Based on this, the Flex-Edge Recursive graph and the double recursive algorithm D-Rec are proposed for operator-level parallel plan search. The experiments demonstrated that D-Rec could find high-quality parallel plans with a search time in the order of seconds, which is much better than the related methods, reaching more than ninety percent of the baseline in terms of linear search complexity in a large range of DNN models. In addition, it outperforms competing methods regarding supported network breadth and portability.

Chapter 5

Parameter synchronization overlap

5.1 Introduction

Chapter 4 describes how to build a cost model based on HSM2DL and how to perform symbolic transformations and simplifications on the cost model to make operator-level parallel plan searching practical.

As a complement and enhancement to distributed data parallelism, parameter synchronization overlap has recently attracted plenty of academic attention [96, 72], and can significantly improve the efficiency of distributed training. It is difficult for cost in a profiling-based approach to describe the optimization of this technique, and in Section 3.6.2, HSM2DL uses γ to describe the role of overlap techniques in cost models. However, γ is a dynamic value that changes with the DL framework's implementation and the neural network's structure. This Chapter describes how HSM2DL calculates the value of γ for a specific case. The final experiments show an improvement in the results compared to data parallelism.

5.2 Hybrid parallelism and parameter synchronization overlap

Recall basic knowledge about distributed training first. DL frameworks use computational graphs to represent DNN models. The computational graphs are composed of operators, some operators (e.g. *Matmul, Conv, etc.*) have *parameters*. Figure 5.1(a) gives an example of a DNN model represented in computational graph format. Training a DNN aims to update its parameters so that the trained model can predict expected results from new inputs.

The DNN training comprises Forward/ Backward Propagations (FPG/BPG), an iterative process. FPG computes the operators in DNN from input to get the outputs. BPG updates the parameters of each operator from the last operator back to the first operator. An epoch is one iteration of FBG/BPG on the whole input data. These data are composed of one or several mini-batches. A mini-batch is a set of samples processed together by DNN. The training consists in updating the parameters through many epochs to reach close to optimal values. Figure 5.1(b) show the timeline of a training iteration on a single device: FPG is composed of the computations from $Op1$ to $Op5$, and the execution order of BPG is the reverse.

In general, pipeline parallelism and parameter synchronization overlap techniques are incompatible, as pipelines split mini-batch data into multiple micro-batches. There are dependencies on the data when back-propagating, so the overlap-able part is very small and can be ignored. Therefore, this section only discusses operator-level overlapping.

The timeline of training one mini-batch training of Data Parallel is shown in Figure 5.1(c). Under Data Parallelism, each accelerator possesses the entire DNN model and computes with a subset of the mini-batch. The sizes of each subset of data are equal, thus ensuring load balancing. The distributed training should follow the same semantics as training on a single device. Therefore, the parameters need to be computed and updated together within a mini-batch of data. Data Parallelism partitions each mini-batch into several subsets of data among devices and computes the parameters, respectively. In order to keep the same semantics as training on a single device, DL frameworks implement *parameter synchronization* to compute the average value of the parameters and broadcast it to all devices. In Figure 5.1, the process

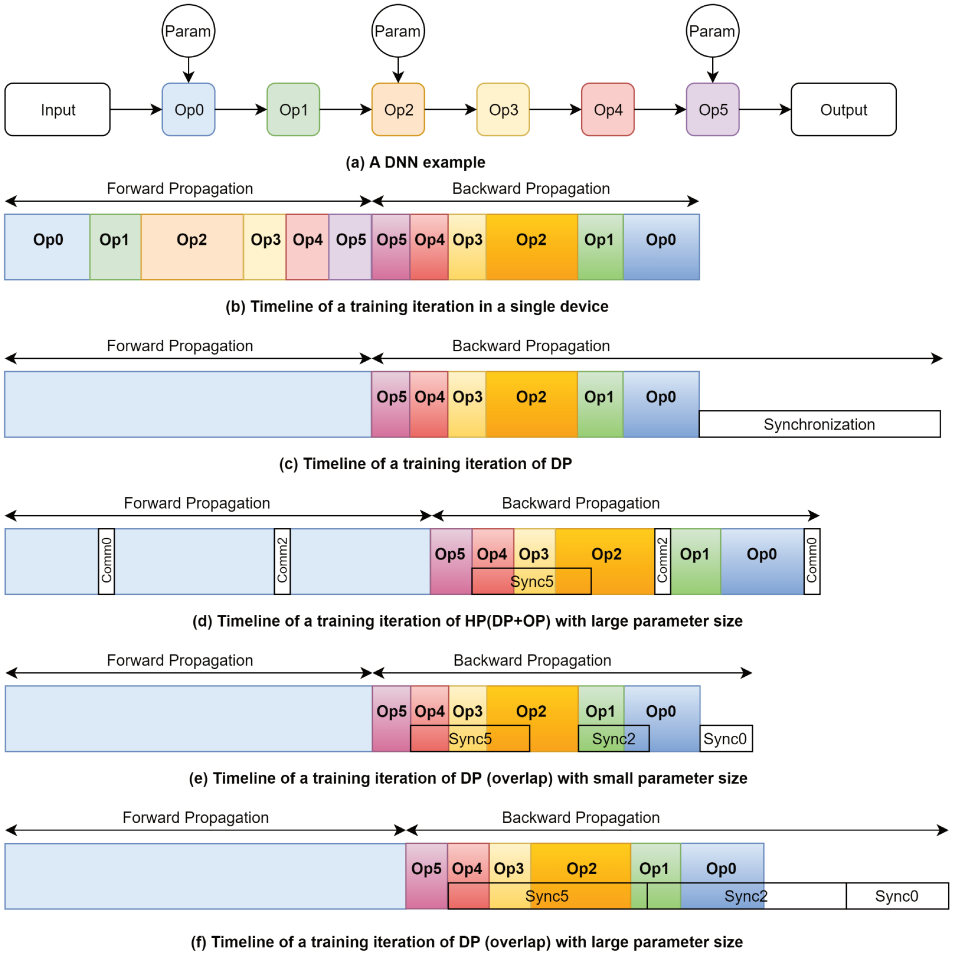


Figure 5.1: Timeline of Distributed DNN Training

of distributed FPG is the same as the FPG of single device training, so the computations from $Op1$ to $Op5$ are fused to the blue rectangle. Note that asynchronous parameter updating will not be considered because it is not realistic in important industrial application domains due to convergence concerns.

When applying model OMP for distributed training, each device possesses the whole mini-batch of data (also the whole parameters), so there is no need to do parameter synchronization in this case. On the other hand, the operators are distributed over the devices, so extra intermediate communications are needed to finish the computation semantics. Figure 5.1(d) shows the timeline of a hybrid parallelism training: OMP is applied on $Op0$, $Op2$ while DP is applied on the other operators. Mixing data parallelism and OMP allows for finding the optimal solution for distributed training.

In practice, the communication/computation overlap technique [72, 96] is implemented commonly by the DL frameworks to optimize training performance. Computations cannot overlap the intermediate communications caused by OMP because data dependencies exist between the communications and computations, so this technique mainly optimizes the parameters synchronization of data parallelism.

Only the operators with parameters need synchronization. Figure 5.1(e) represents the training iteration with overlap. The synchronization calculates and broadcasts the average value of the updated parameters stored on different devices. It cannot be executed before the end of the BPG computation of the related operator. As shown in Figure 5.1(e), $Sync5$, $Sync2$, $Sync0$ are executed respectively after the computation of $Op5$, $Op2$, $Op0$ and the synchronization of different operators are independent with each other. With overlap, data parallelism becomes more efficient (compared to Figure 5.1(d)). However, for the DNNs with large size of parameters, the synchronization may take much time which cannot be totally overlapped (as is shown in Figure 5.1(f)), resulting thus non-negligible tail time at the end of BPG.

5.3 Tail factor calculation based on HSM2DL

Here is a restatement of the problem we are trying to solve: This chapter aims to find a way to calculate the cost of parameter synchronization when using the computation/communication overlapping technique, i.e., this corresponds to γ defined in Equation (3.3). γ is called the *parameter synchronization ratio*, or simply the *tail factor*, which represents the proportion of the cost of parameter synchronization that cannot be overlapped.

5.3.1 Modeling the tail factor

Chapter 4 demonstrated that the computation cost, communication cost, and parameter synchronization cost could be abstracted and represented as math functions (i.e., in a symbolic way) from the semantics of operators. Their values can be inferred from the functions with input data shapes. D-Rec evaluates the symbolic cost model Equation (4.1) and decides parallel plans for each operator.

This chapter will concentrate on how to find proper γ in Equation (4.1) to improve its accuracy. Profiling γ as a global value for all the operators is not accurate enough because the tail factor varies for different operators. Besides, profiling needs to be re-executed each time for different DNN structures, which is time costly. Therefore, tail factors γ_i are defined as separate values for each operator *Opi*.

This section attempts to build a symbolic model of synchronization overlap based on the following observations

- The computation cost is a fixed value for each operator, whether it is assigned data parallel or model parallel (operator-level).
- The parameter synchronization is actually an overlap of communication and computation. Both the intra-communication cost of model parallel (operator-level) and synchronous communication occupy exclusive bandwidth, so they cannot be overlapped.

Based on these two observations, it can be inferred that the two kinds of time to be overlapped are actually the computational cost of all operators and the parameter synchronization cost. These values are fixed for a given neural network on a given piece of hardware. The tail factor for each operator can be calculated with an independent algorithm of a particular DNN model before applying the tail factor in D-Rec.

It is incorrect to calculate the tail factor directly by comparing the synchronization cost of an operator with the computation cost. Indeed the order of computation in BPG is fixed, as in Figure 5.1(e), the synchronization cost of *Op5* cannot overlap without its own computation being finished. It is essential to consider whether the computation cost of subsequent operators can overlap the synchronization cost of *Op5*.

From Figure 5(e), if the computation of subsequent operators can overlap the synchronization time, then the previous operators' tail factors are equal to 0, meaning they can completely overlap. Only the last operator in the BPG,op_0, has no subsequent operators to overlap. Its tail factor $\gamma_{Op_0} = 1$. Figure 5(f) shows the large trailing when the synchronization time is too large to be overlapped. In fact, this dragging is caused by each of the previous operators with a parameter: the synchronization cost of *Sync5* cannot be fully overlapped by the computational cost of *Op4*, *3*, *2*, so it takes up part of the computational part of *Op1*, resulting in *Sync2* should be postponed, and these contribute to the final trailing.

5.3.2 Tail Factors Algorithm

Algorithm 2 shows the searching algorithm for all γ_i. The key idea of this algorithm is to use an intermediate variable *Overlap* to summarize the computation that can be overlapped for the synchronization of the current operator.

Variables definition

- A DNN model is represented as a computational graph $G = \langle v_i \rangle$ which is a list of operators. The operators in the list are sorted by the reverse order of the BPG processing (i.e. from Op_0 to Op_5 in Fig.5.2).

- Operators Op_i are represented by the vertices v_i. A vertex $v_i = (S_i, C_i, \gamma_i)$ is a tuple composed of three elements: potential synchronization cost $S_i = \beta * q_{s_i}$, computation cost C_i and tail factor γ_i.

An example of tail time partition is shown in Fig.5.2, where $T_i = S_i * \gamma_i$ denotes the tail time caused by the i_{th} operator. In Fig.5.2, *Overlap* for Op_0 equals to 0 because the synchronization of the first operator cannot be overlapped. For Op_2, *Overlap* equals to the summation of the computations of Op_1 and Op_0. The *Overlap* for Op_5 depends on the possibility to totally overlap the synchronization of Op_2 by Op_1 and Op_2. It equals to the summation of the computation of Op_4, Op_3, Op_2 and the rest of Op_1, Op_0 after overlapping the synchronization of Op_2. The factor contributing to the tail of each operator can be calculated with its synchronization cost and the constantly updated intermediate variable *Overlap*: $\gamma_i = \frac{S_i - Overlap}{S_i}$. As shown in Fig.5.2, $S_0 = T_0$ so that $\gamma_0 = 1$; $T_2 = S_2 - C_2 - C_0$ thus $\gamma_2 = T_2/S_2$; $T_5 = S_5 - C_4 - C_3 - C_2$ thus $\gamma_5 = T_5/S_5$.

Algorithm 2 Tail Factors Algorithm

Require: Graph $G = \langle v_i \rangle$ ordered according to the reverse order of BPG processing
Ensure: γ_i for each operator v_i

```
function TailFactorsSearch(G)
    Overlap = 0
    for v_i in G do
        if S_i − Overlap ≤ 0 then
            γ_i = 0
            Overlap += C_i − S_i
        else
            γ_i = (S_i − Overlap) / S_i
            Overlap = C_i
        end if
    end for
return G
end function
```

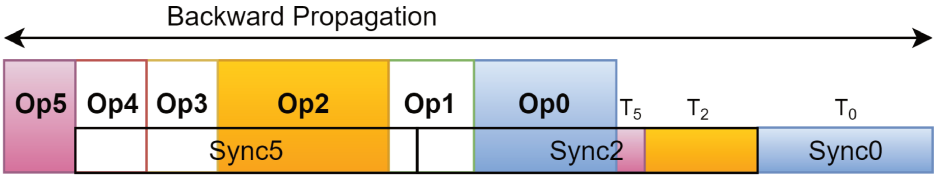


Figure 5.2: Tail Time in Backward Propagation

5.4 Experiments

These experiments address the performance improvement when applying an accurate tail factor into the symbolic cost model of HSM2DL.

5.4.1 Experiments Setup

The experiments were run on an Atlas 900 AI cluster equipped with 2 CPUs and 8 Ascend910 accelerators under the MindSpore DL training framework. The strategy searching algorithms were run on CPUs, and the DNN training was run on the accelerators.

The DNN model chosen for the experiments is Resnet50 [25], which is one of the most popular and commonly used CNN. The structure of Resnet50 is linear. Its main structure comprises a series of convolution layers and a fully connected layer at the output. Resnet50 is a network for image classification. *Number of classes* (NoC) indicates how many categories the input data is classified into. NoC is one

possible partition dimension in the fully connected layer parameter tensor. Therefore, the size of parameters can be changed by varying the NoC. In order to show the impact of γ on the symbolic cost model of HSM2DL, the experiments evaluate the training performance of Resnet50 with different NoCs. The dataset for the experiment is 'fake data' (generated randomly), for which the NoC can be easily tuned.

The purpose of the experiments is to validate the effect of γ and how much improvement can be brought about by Algorithm 2. The founded γ will be substituted into the cost model Equation (4.1) of HSM2DL, and the parallel plans will be searched with D-Rec Algorithm 1. D-Rec is tested in two situations: 1) $\gamma = 1$ and 2) determining γ by Algorithm 2. Data parallel is chosen as the baseline to find out the operator-level parallel trade-off where such Hybrid Parallelism provides better performance. Other factors like the number of layers are not considered in our experiments since their main impacts are on the network precision but not the Hybrid Parallel plan changing.

5.4.2 Results

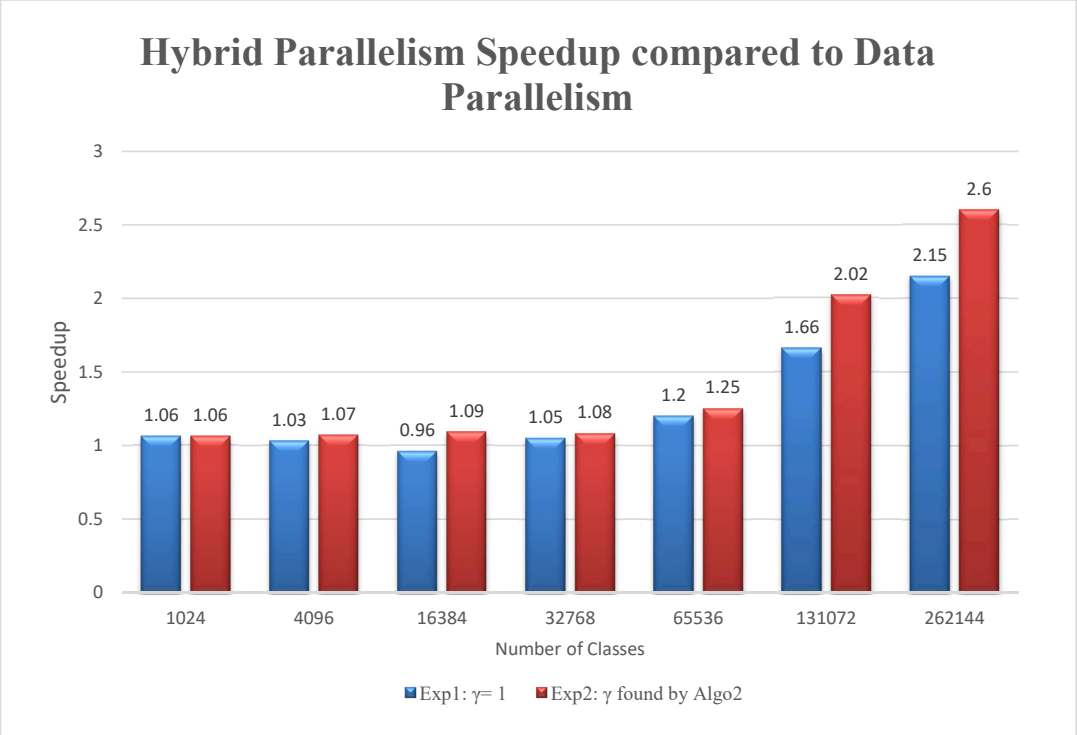


Figure 5.3: HP speedup with the number of classes

Training throughput denotes the training performance of distributed DNN training, and a higher training throughput signifies a better HP strategy. Here, the training throughput speedup compared to that of DP is used to evaluate the training performance of the hybrid parallel plan found by our approach. To show the effects of γ found by Algorithm 2, the NoC is varied in the experiments to change the size of the parameters.

Generally speaking, when the parameter size is small, DP is more efficient because the synchronization cost is small. When the parameter size is large, OMP is more efficient because of the high synchronization cost, which cannot be overlapped.

The results are shown in Fig.5.3. The NoC varies from 1K to 256K. The experiments of $\gamma = 1$ are called *Exp1* and the experiments of γ found by Algorithm 2 are called *Exp2*. For *Exp1*, $\gamma = 1$ means it does not take the overlap technique into consideration.

For small NoCs (less than 4K), the training performances of data parallel and the two *Exp*s found by D-Rec are similar. The reason is that the parameter size is relatively small. The synchronization cost is small, so the three methods (including the baseline) all choose a data parallel-based plan.

For NoC = $16k$, the performance of *Exp1* is worse than data parallel because when the synchronization cost increase, it tends to change part of data parallel to model parallel. However, in this case, the parameter synchronization can be majorly overlapped by the computation, DP is still better than model parallel. On the contrary, *Exp2* can accurately determine the tail factor γ and find a better parallel plan.

From 32K-256K, the training performance of DP is getting worse than hybrid parallel. This is because the size of the parameters increases with the NoC as well as the synchronization cost. The computation cannot overlap the expensive synchronization. Therefore D-Rec applies OMP strategy on the operators with big γ, thus avoiding the important synchronization cost and permitting a good training performance. Although both *Exp1* and *Exp2* return better performance than data parallel, it can be conducted that with a good tail factor found by Algorithm 2, the cost model is more accurate and can find higher quality parallel plans.

5.5 Conclusion

This Chapter introduced the parameter synchronization overlap, including how it is generated and how to quantitatively evaluate its impact on the cost model of HSM2DL. Section 5.3 represents an algorithm to determine the tail factor γ, which is the key point to evaluating the synchronization cost. Experiments show that with the help of Algorithm 2, HSM2DL can be more efficient and find parallel plans with higher performance.

Chapter 6

Graph level plan search

6.1 Introduction

Gigantic models have achieved unprecedented performance in challenging AI tasks and have shown the excellent capability of generalizing to unseen data. Chapter 4 introduced how to search operator-level parallel plans, including symbolic reduction of the cost model and linear complexity D-Rec algorithm. However, another trend of the DNN model today is that the scale of those gigantic models (including GPT-3 [9], GShard [41] etc.) may have trillions of parameters (e.g., Google Gopher has 600B parameters [64]). Their training needs to be parallelized across hundreds of distributed devices (e.g., CPUs, GPUs, and NPUs). However, the communication capacity between servers is very low, so operator-level parallelism is insufficient to train them efficiently. These large-scale models are usually trained through a parallelism plan that combines data parallelism [2], operator parallelism [36, 74], and pipeline parallelism [32].

This chapter first introduces the DL-domain-specific hypothesis of HSM2DL for pipeline parallelism and the cost model analysis based on that. Section 6.3 introduces a joint parallel plan search algorithm of HSM2DL for the gigantic DNN models. The main idea to generate an optimized parallel plan for gigantic DNN models on a heterogeneous cluster is the following: 1) Split the network into several stages. 2) The total latency of the stages should be optimized. 3) The stage partition point should be searched jointly with the operator-level parallel plan.

This joint parallel search algorithm is conceived with a hierarchical structure. The operator-level search takes the main idea of the D-Rec algorithm (Section 4.4). The difference between them is that D-Rec returns the optimal parallel plan for N devices, but here the algorithm saves a series of optimal candidate plans for different numbers of devices $2^n, n = 0, 1, 2, ..., N$. These candidate plans are used as the input for the pipeline search, and the performances of the global parallel plans are evaluated to select the minimum one. The property of recursive partitioning (Definition 4.2.4) that per-step optimality leads to global optimality also benefits pipeline partitioning. Indeed, HSM2DL can thus reduce the graph-level planning complexity to linear (i.e., $O(\log_2 N \times |\mathcal{V}|)$ where N is the number of devices and $|\mathcal{V}|$ is the number of DNN operators). Thus the efficiency of HSM2DL in handling gigantic DNNs with massive devices is proved.

Applying the D-Rec algorithm (Algorithm 1) to find the operator-level parallel plan. Supposing that the number of devices equals N, saving all the candidate optimal op-level plans for different numbers of devices of $2^n, n = 0, 1, ... \log_2 N$.

To verify the effectiveness of HSM2DL, extensive cluster experiments have been conducted on a large-scale accelerator cluster. For BERT and GPT-3, HSM2DL can return optimized parallelism plans for a cluster with thousands of devices in less than 50s seconds. Specifically, for GPT-3 with 39B parameters, HSM2DL can return a plan in 1.02 seconds while Alpa takes 804.5 seconds for extra profiling and 1582.66 seconds for computing a plan. Note that both the two times of Alpa for profiling and computing are optimized through simulations, decreasing the searching optimality. On the other hand, HSM2DL does not compromise the quality of plans. Its plans exhibit optimized performance matching the well-known Expert-Designed parallel system Megatron-LMv2 [58].

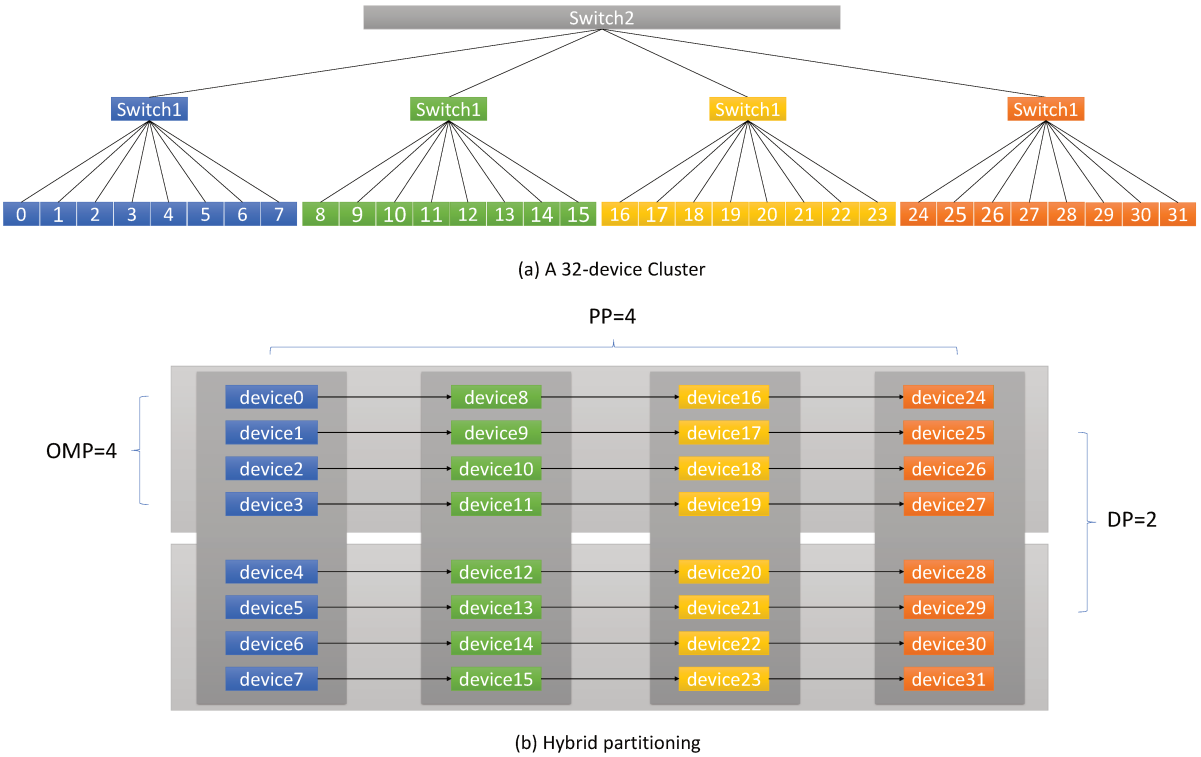


Figure 6.1: Pipeline partition example and cluster mapping

6.2 Graph-level partitioning analyses

Let us recall that the computation of a parallelism plan for gigantic DNNs is challenging: searching the operator-level parallelism alone is an NP-hard problem. As for pipeline parallelism, the computational graph is partitioned into several consecutive stages, while the partition point of stages can be put between every two operators. As a result, searching for pipeline parallelism is an NP-hard problem. In addition, in the search for an optimal plan, the related works in Section 2.4.3 predict the execution time of candidate plans. For pipeline parallelisms, every possible stage needs to be profiled. Besides, the profiling data often contain measurement errors that would be accumulated during the searching algorithm, affecting the optimality of a found plan. More importantly, profiling DNN operators with all possible hardware devices are prohibitively expensive, and such profiling data is often unavailable. Existing parallelism planners in Section 2.4.3 exhibit poor support for gigantic DNNs. Pipeline-optimized planners, such as PipeDream [56], Piper [80], and Alpa [97], reduce configuration search space by first using manual rules to return feasible plans for data and model parallelism and then optimize the configuration for pipeline parallelism. This design prevents these planners from jointly optimizing for data, model, and pipeline parallelism, resulting in finding sub-optimal plans. It further incurs tremendous execution time, making the planners tedious and expensive to use in practice.

6.2.1 Scope of HSM2DL's pipeline partitioning

This chapter is a preliminary attempt to investigate the search for pipelines. The aim is to initially validate the feasibility of a triple hybrid search and implement a working prototype. This chapter selects large models that currently play a critical transformer-based role in industrial production to conduct research to support the early implementation of automatic parallelism in the industry.

Since the transformer [85] was proposed in 2017, more and more large models based on the trans-

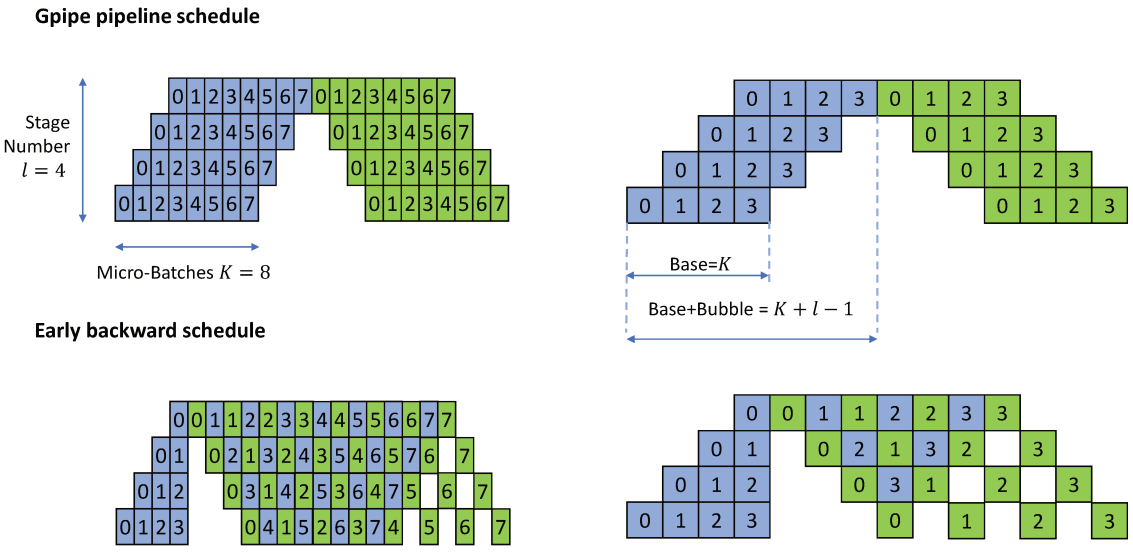


Figure 6.2: Two types of the pipeline scheduler and two examples of each

former have come out. These models play a crucial role in the field of NLP, such as language translation, human-machine quizzing, etc. For example, Google proposed BERT [18] in October 2018, and Nvidia proposed Megatron-LM [74] with 8.3 billion parameters in August 2019. GPT-3 [9] proposed by OpenAI in 2020 contains 175 billion parameters, and both Huawei Pangu-alpha [95] and Baidu ERNIE-3 [79] in 2021 contain over 200 billion parameters. MoEs [21], which is composed of sparse computing units, even have more than trillions of parameters. Wudao-2.0, proposed by the Beijing Academy of Artificial Intelligence in 2021, contains 1.75 trillion parameters. The large sparse parameter size does not directly lead to a better DNN model accuracy: the DNN model uses more computing devices, but the training results are only at the level of a dense transformer with 100 billion parameters. When the cluster network is composed of multi-layer switches, the bandwidth of the upper-layer switches is low. Operator-level parallelism causes intra-communication and parameter synchronization, which requires higher communication capacity. However, the bubble generated by pipeline slicing is due to the dependency between different micro-size, which is unrelated to inter-device communication. Therefore, applying pipeline parallelism in training transformer-based large models is important.

The following is the scope of this chapter:

- In order to guarantee the same computation semantics as in training on a single device, the design of pipeline partitions in this paper only considers synchronized pipelines but not asynchronized cases. Ensuring that the training accuracy is consistent with that of a single machine is also a requirement for industrial production.

- The current assumptions for DNN only consider the case of dense transformer computation. This chapter will not discuss the case of sparse matrix computation, such as MoE [21] and other types of networks. The extension for the other types of DNN models is one of the perspectives of the thesis.

- The abstract machine for pipeline partitioning follows the definition of HSM2DL in Section 3.4. This model is **hierarchical** and **symmetric** as it is built on top of typical DNN cluster architecture. In this model, each level of the tree is assumed to be comprised of homogeneous network components where the components in the same level have identical bandwidth to the consecutive levels. As a result, the tree topology can be treated approximately as a binary tree. The network topology's symmetry and hierarchy property can greatly simplify the algorithm's search space. Based on this

abstract machine, HSM2DL deals with 2^n devices. More precisely, graph-level pipeline partitioning splits the computational graph into 2^{n_1} stages, and each stage is partitioned into 2^{n_2} parts by operator-level partitioning. Here, the number of devices $N = 2^n = 2^{n_1} \times 2^{n_2}$.

An example of pipeline partitioning is illustrated in Figure 6.1. In the example sub-figure (a), a training cluster consists of four computing machines, each containing eight fully connected accelerators. There are a total of 32 accelerators. On the top level of the cluster, the communication bandwidth between the four computing machines is very poor. Therefore, a pipeline parallel plan is assigned between the machines because the bubble time does not rely on the communication bandwidth. In Figure 6.1, each color represents a stage. In sub-figure (a), each stage is arranged on a single computing machine. From sub-figure (b), it can be seen that the different stages are computed in order. On the other hand, the communication bandwidth between the eight accelerators inside the computing devices is higher, so that operator-level parallel is assigned to the accelerators, as shown in sub-figure (b): OMP=4, DP=2.

6.2.2 Pipeline partitioning analysis

In practice, the partition of pipeline parallelism focuses on two main points:

- How many stages to partition?
- At which point the computational graph is partitioned into the stages?

Balanced partition points

To ensure that the pipeline can be computed efficiently and to avoid generating extra idle time except for the inevitable bubble, it is necessary to control that the number of micro-batches K is equal within each stage and the computation time for each micro-batch (as in each small square in Figure 6.4) is as equal as possible. The user determines the number of micro-batches used as input to the algorithm.

Generally, the number of stages can be any integer instead of restricted to 2^n, and each stage can consist of a different number of devices. However, when different stages are assigned to different numbers of devices, it is complicated to guarantee load-balancing for each micro-batch. In the following discussion, the number of stages is fixed to 2^n, and the per-stage device number is set to the same integer. This may seem to reduce the search space significantly and miss the optimal result. Still, in reality, the load balancing of uneven partitioning is very difficult to control at the DL framework design level. Therefore, this assumption is actually consistent with the fact that transformer-based networks are used in practice. The uneven partitioning is also not treated in this thesis but is a perspective.

The structure of the transformer is discussed below to illustrate the theoretical basis of even partitioning:

- The main structure of the transformer is composed of a large number of dense MatMul which computation and amount of communication can be computed relatively linearly. The communication bandwidth and the computational performance of the device can be used at a high utilization rate to maintain a stable value, so the cost model based on α, β, q_x, q_m can express the trade-off between different stages very well and therefore make the relatively accurate partitioning decision.
- A transformer is made up of several repeating layers. Generally, a transformer-based network has an equal amount of computation per layer. Load-balancing can be easily guaranteed.
- For other models, it is necessary to extend the computing model to support uneven partitioning. Because sparse or non-parallelizable operators result in inaccurate profiling parameters α, β and the uneven partitioning of stages need to improve the concept of 2-parts partitioning. The inaccurate value issue can be solved by adding some profiling information to the operator but this is not discussed in this thesis.

In summary, this chapter focuses on the typical DNN model transformer. For optimal pipeline partitioning, it is necessary to ensure that the computation time of each micro-batch is approximately equal. Based on the scope of this thesis and the practical needs of industrial production, two more substantial constraints are imposed:

- The number of stage cuts must be 2^n to guarantee better load-balancing.
- The number of devices per stage is equal, based on the relatively average computation of each layer of the transformer.

Pipeline stage numbers

The next problem to be solved is the number of stages to partition the computational graph. As introduced in $Cost_{SS}$ (Equation (3.2)), the cost of the waiting time generated by the pipeline is expressed by the bubble ratio R_b for graph-level partitioning, and pipeline partitioning does not impose additional communication costs (the smaller communication costs between stages can be ignored). Therefore, the costs of graph-level parallelism and operator-level parallelism are orthogonal. DP generates q_s, MP generates q_m, and PP generates R_b.

Because the operator-level partitioning cost is independent of the pipeline cost, the operator-level parallel plans will be searched first. Since the operator level guarantees per-step optimality, the joint algorithm will save all optimal partitions of the computational graph into $2^{n_2}, n_2 = 0, 1, ..., log_2 N$ parts. Different $n_1 = n/n_2$ stage numbers will be evaluated with the $Cost_{SS}$ by substituting the $Cost_{MS}$ (Equation (3.1)) under the assumption of ensuring load-balancing. The optimal hybrid parallel plan is the one with the minimized $Cost_{SS}$.

The cost model of the pipeline is an extension of $Cost_{SS}$. Bubble ratio R_b refers to the ratio of the pipeline execution time compared to the execution time of the same device numbers for data parallelism. Let us note that the bubble ratio R_b can be represented by the number of stages l and the micro-batch number K: $R_b = (l + K - 1)/K$.

There are different scheduling implementations for pipeline partitioning. Figure 6.2 gives two types of scheduling implementations: Gpipe pipeline schedule [32] and Early backward schedule from DAPPLE [20]. We give both types two examples with different micro-batches. With guaranteed load-balancing and a synchronous pipeline scope, GPipe proposes a straightforward implementation, in which backward-phase stages do not start until all forward-phase stages finish, as shown in Figure 6.2. While it is confirmed by DAPPLE that the early backward schedule has less peak memory consumption than the GPipe's schedule. As shown in Figure 6.2, the early backward schedule enforces the backward-phase stages to start as soon as possible so that the activation memories can be released early to mitigate the peak memory. From Figure 6.2, it can be deduced that R_b is always the same for any different scheduling implementations.

Thus in this chapter, the cost of the entire joint cut is as follows:

$$C_p(\mathcal{G}, \mathcal{P}_o, l) = \frac{l + K - 1}{K} \left(\alpha \times q_x \left(\mathcal{V}, \mathcal{P}_o\right) + \beta \times q_m \left(\mathcal{V}, \mathcal{E}, \mathcal{P}_o\right)\right) + \gamma\beta \times q_s(\mathcal{V}, \mathcal{P}_o) \tag{6.1}$$

where the first two terms calculate the cost of forward and backward passes for pipeline parallelism, and the last term calculates the cost of data parallelism. The algorithm returns the hybrid parallelism plan with minimal cost.

The joint search algorithm is represented in the following section.

6.3 Joint search algorithm for pipeline and operator parallelism

This section introduces a two-step joint search algorithm based on HSM2DL and the assumptions discussed in Section 6.2. The operator-level search of the joint algorithm (Algorithm 3) is based on the D-Rec algorithm (Algorithm 1), since D-Rec can guarantee the per-step optimality of the operator-level parallel plan for each 2-parts partitioning. Therefore, the optimal operator-level parallel plans for 2^{n+1} ($n = \{0, 1, ..., \log_2 N\}$) devices can be saved in a list with one execution of D-Rec. These saved operator-level parallel plans are called *candidate parallel plans*. The list of the different numbers of devices' optimal parallel plans is taken as input for the graph-level search.

The graph-level search iterates through the different stage numbers (from 1 to $\log_2 N$) and finds a balanced partition point for these stage numbers to guarantee the maximum efficiency of the pipeline. Finally, the optimal graph-level parallel plan is chosen by comparing the cost of the whole graph with different stage numbers.

6.3.1 Operator-level candidate plans searching

Algorithm 3 Operator parallelism plans generation (D-Rec Algorithm 1 serves for the joint search)

1: **Input:** Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of devices N.
2: **Output:** List of candidate plans $\mathcal{L}$
3: $\mathcal{L} = []$
4: **for** $i = 0, \ldots, \log_2 N$ **do** // Line 1 in Algorithm 1
5: $\sigma(\mathcal{V}) =$ order $\mathcal{V}$ by λ // Line 5 in Algorithm 1
6: $\mathcal{L}.append([])$
7: **for** $j = 0, \ldots, size(\mathcal{V}) - 1$ **do** // Line 11 in Algorithm 1
8: $v = \sigma_j(\mathcal{V})$
9: $p^{\rho}_{best} = \underset{p^{\rho} \in \mathbb{P}^{\rho}_v}{\operatorname{argmin}} \hat{c}(v, p^{\rho})$ // Line 16 in Algorithm 1
10: **if** $i > 0$ **then**
11: $p_v = \mathcal{L}[i-1][j] * p^{\rho}_{best}$
12: **else**
13: $p_v = \langle\langle 1, 1\rangle, ...\langle 1, 1\rangle\rangle$
14: **end if**
15: $\mathcal{L}[i].append(p_v)$
16: $\sigma_j(\mathcal{V}) = (split(v.t[0], p^{\rho}_{best}.p^t[0]), ..., split(v.t[m-1], p^{\rho}_{best}.p^t[m-1]))$ // Line 7 in Algorithm 1
17: **end for**
18: **end for**

In order to search the candidate parallel plans for the computational graph, Algorithm 3 takes the main idea of the recursion-style algorithm Algorithm 1 and rewrites it in a loop iteration style. Besides, the optimal parallel plans for different numbers of devices are all saved in a list $\mathcal{L}$. Algorithm 3 takes the computational graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as well as the cluster information (i.e., the number of devices N) as inputs and generates a list of candidate optimal plans for all possible cluster partitions. The outer loop (i.e., beginning at line 4) enumerates all possible cluster partitions (i.e., partitioning N devices into $2^{i+1}, i = 0, \ldots, \log_2 N - 1$ groups). In each step, all operators are first reordered according to their compromise price (i.e., lines 5-8,λ is defined in Definition 4.3.6) to ensure that operators with a larger impact on the cost will be considered before those with a smaller impact. The candidate plans are updated during the loop (i.e., line 16). The inner loop (i.e., beginning at line 10) searches the optimal primitive configuration p^{ρ}_{best} for each operator $v \in \mathcal{V}$ according to Equation (4.4). The optimal primitive configuration is combined with the previous optimal configuration to obtain the current optimal configuration. Finally, the shape of the operators in the computational graph is updated with *split* (Definition 4.2.6).

A running example of Algorithm 3 is shown in Figure 6.3, where the algorithm generates the candidate plans for two operators. Here, supposing the computational graph is composed of two MatMul Op0 and Op1 as shown in Figure 3.9. The algorithm starts with initial states where the computational graph is not partitioned. In the first step, the optimal primitive configurations are p^{ρ}_1 and p^{ρ}_0 (Equation (4.2)) for Op0 and Op1, respectively. After combining the optimal primitive configurations with $\mathcal{P}^0_o$, the first candidate plan $\mathcal{P}^1_o$ is obtained. Similarly, in step 2, the optimal primitive configurations are p^{ρ}_0 and p^{ρ}_3 for Op0 and Op1, respectively. After combining the optimal primitive configurations with $\mathcal{P}^1_o$, the second candidate plan $\mathcal{P}^2_o$ is obtained. Above all, the candidate plan list $\mathcal{L} = \left[\mathcal{P}^1_o, \mathcal{P}^2_o\right]$. Therefore, for the given computational graph $\mathcal{G}$, the optimal parallel plans for 2 devices $\mathcal{P}^1_o$ and the optimal parallel plans for 4 devices $\mathcal{P}^2_o$ is saved in $\mathcal{L}$ and can serve for the pipeline partitioning.

6.3.2 Pipeline search based on the candidate plans

After generating a list of candidate plans for operator parallelism using Algorithm 3, the next step is to explore the optimal hybrid parallelism plans that combine operator parallelism with pipeline parallelism. The goal is to find a graph-level partition such that the computational graph is partitioned into consecutive stages: $\mathcal{P}_g = (\mathcal{S}_0, ..., \mathcal{S}_{l-1})$. The key point is that the computational workloads are balanced among the different partitions to avoid resource underutilization.

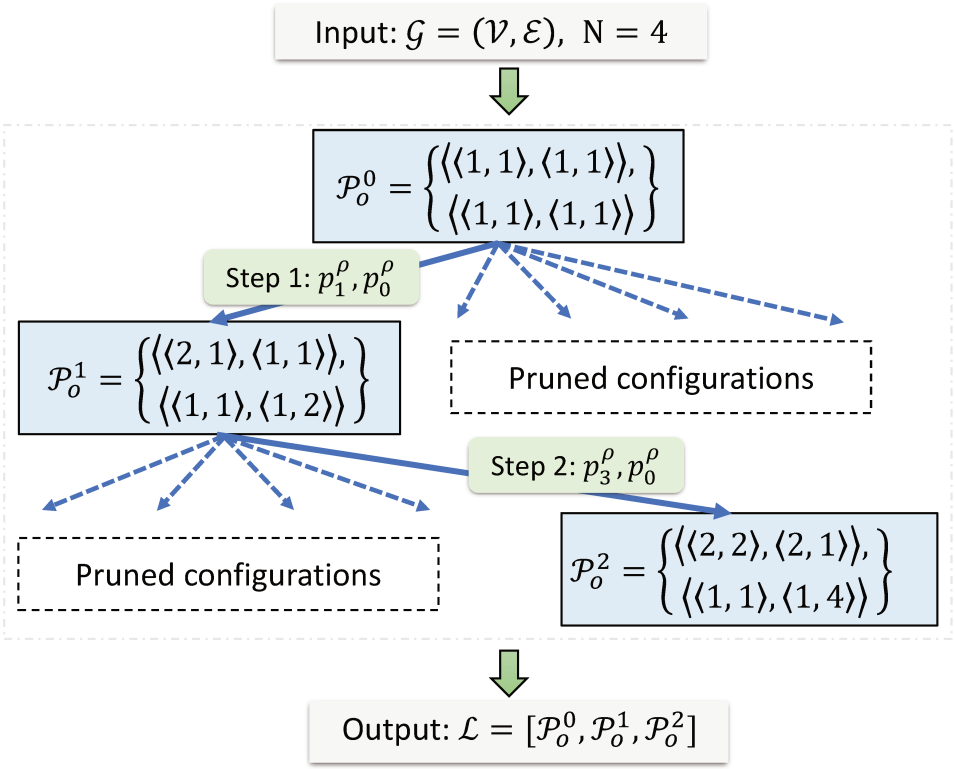


Figure 6.3: A running example of Algorithm 3

The procedure of the search algorithm is described in Algorithm 4 and a running example is given in Figure 6.4. Algorithm 4 enumerates the list $\mathcal{L}$ of the candidate plans $\mathcal{P}_o$. For each candidate plan, the balanced graph partition can be found in linear time using Algorithm 5 (line 6). Suppose the resulting hybrid parallelism plan $\mathcal{P} = (\mathcal{P}_g, \mathcal{P}_o)$ does not exceed the memory limitation M, $max\_memory\_usage(\mathcal{G}, \mathcal{P}_g, \mathcal{P}_o)$ (i.e., line 4) is applied to the computational graph as a security check. In this thesis, the memory cost model is not considered but used as the security check to ensure the parallel plans will not exceed the device memory. In that case, the algorithm computes the cost of the $\mathcal{P}_o$ with respect to pipeline parallelism and is evaluated by the Equation (6.1) (line 20). The optimal parallel plans $\mathcal{P} = (best\_\mathcal{P}_g, best\_\mathcal{P}_o)$ will be found with the minimized cost.

Algorithm 5 describes the procedure to find the balanced partition point for the different number of stages. The symbolic cost will be accumulated by the operators and the computational graph would be partitioned evenly according to the operator-level symbolic cost c_o.

6.3.3 Time complexity and practical considerations

Time complexity

In Algorithm 3, the outer loop enumerates $\log_2 N$ steps while the inner loop searches the optimal primitive configuration for each operator $v \in \mathcal{V}$ by calculating all possibilities in $\mathbb{P}_v^\rho$. Totally, the time complexity of Algorithm 3 is $O(\log_2 N \cdot |\mathcal{V}| \cdot |\mathbb{P}_v^\rho|)$, where $|*|$ denotes the size of the set. Similarly, Algorithm 4 enumerates $\log_2 N$ steps. During each step, it costs $O(|\mathcal{V}|)$ time to find the pipeline partition resulting in $O(\log_2 N \cdot |\mathcal{V}|)$ time complexity. Therefore, the total time complexity of our method, including Algorithm 3 and Algorithm 4, is $O(\log_2 N \cdot |\mathcal{V}| \cdot |\mathbb{P}_v^\rho|)$, where for common operators, $|\mathbb{P}_v^\rho| < 10$. In contrast, the time complexity of Piper [80] and Alpa [97] is larger than $O(|\mathcal{V}|^2 N^2)$ even though they already manually limit the searching space which is smaller than the search space of HSM2DL.

Algorithm 4 Joint search for pipeline and operator plans

Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, operator parallelism candidate plans $\mathcal{L}$.
Output: Parallelism plan $\mathcal{P} = (best\_\mathcal{P}_g, best\_\mathcal{P}_o)$
$min\_cost = 0$
for $i = 0, \ldots, \log_2 N$ **do**
    $\mathcal{P}_o = \mathcal{L}[\log_2 N - i]$
    $\mathcal{P}_g = find\_pipeline\_partition(\mathcal{V}, \mathcal{P}_o, 2^i)$ // Algorithm 5
    if $max\_memory\_usage(\mathcal{G}, \mathcal{P}_g, \mathcal{P}_o) < M$ **then** // M: available memory
        $cost = C_p(\mathcal{G}, \mathcal{P}_o, 2^i)$
        if $cost < min\_cost$ **then**
            $min\_cost = cost$
            $best\_\mathcal{P}_g = \mathcal{P}_g$
            $best\_\mathcal{P}_o = \mathcal{P}_o$
        end if
    end if
end for

Algorithm 5 Finding a balanced graph partition for pipeline parallelism

Input: Set of vertices $\mathcal{V}$, operator parallelism candidate plans $\mathcal{P}_o$, number of pipeline stages l.
Output: Graph level parallelism sub-plan $\mathcal{P}_g$
// Group the vertices to generate a balanced pipeline
$cost\_per\_stage = \sum_{v \in \mathcal{V}} \frac{c_o(v, \mathcal{P}_o)}{l}$
$c_{temp} = 0$
$i = 0$
for $v \in \mathcal{V}$ **do**
    $c_{temp} = c_{temp} + c_o(v, \mathcal{P}_o)$
    if $c_{temp} > cost\_per\_stage$ and $i < l$ **then**
        // the current stage is full
        $c_{temp} = c_o(v, \mathcal{P}_o)$
        $i = i + 1$
    end if
    add v to $\mathcal{P}_g.\mathcal{S}_j$
end for
return $\mathcal{P}_g$

6.4 Experiments

6.4.1 Environment setup

This section evaluates HSM2DL using real-world gigantic DNN models in large-scale clusters. In particular, this section compares the planning time and parallel plan quality achieved by HSM2DL with SOTA planners.

DNN models: The experiments consider a wide range of transformer-based DNN models, as shown in Table 6.1. These natural language processing models include three types of **GPT-3** [9] and **BERT** [18]. The different types of GPT-3 have different numbers of DNN layers, tailored for different kinds of tasks, and they are named $\text{GPT3}_{\text{SMALL}}$, $\text{GPT3}_{\text{MEDIUM}}$, and $\text{GPT3}_{\text{LARGE}}$, respectively. There are several dimensions to scale up/down transformer-based [85] models (e.g., the number of layers, the number of heads, etc.). $\text{GPT3}_{\text{LARGE}}$ and BERT models are scaled only along the number of layers, which are denoted by their parameter size (i.e., $\text{GPT3}_{\text{LARGE}}$-xxB and BERT-xxB).

Baselines: HSM2DL is compared with: (i) **Alpa** [97] is the SOTA planner for GPT-3 models; (ii) **Piper** [80] is another SOTA planner for BERT; (iii) **Megatron-LMv2** implements the NVIDIA Megatron-LMv2 [58], and (iv) **Megatron-LMv2-tuned** employs parallelism experts (who are working on real-world large-scale DNN clusters) to fine-tune the plan produced by Megatron-LMv2.

Hardware configuration: The evaluation is conducted within a deep learning cluster that has 32 servers, and each server has 8 Ascends, summing up to 256 Ascends devices. Each server has 512 GB

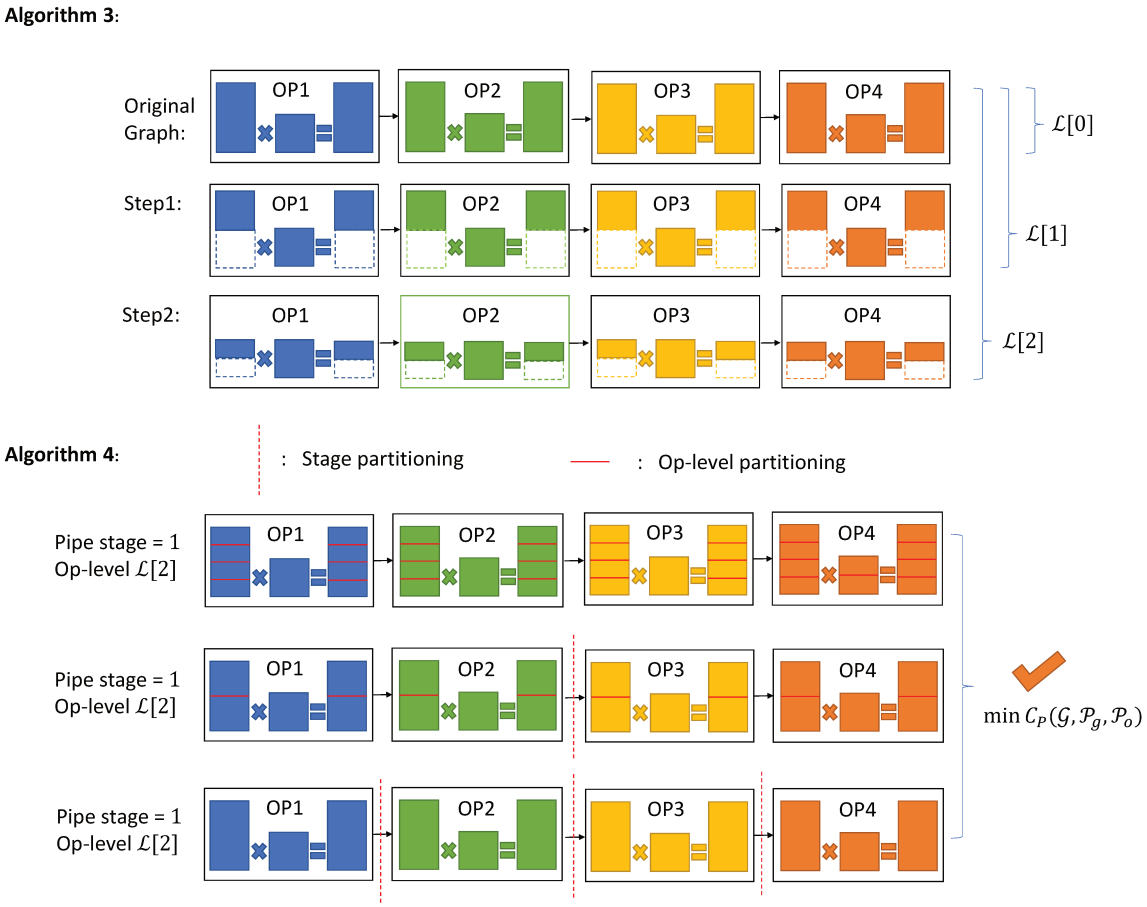


Figure 6.4: A running example of Algorithm 4

CPU memory, 128 CPU cores and 100 Gbps InfiniBand connectivity.

6.4.2 Planning time

The planning time experiments are conducted in an abstract DNN cluster (similar to the Pipe paper) where the scalability of planners is tested with a large number of devices. The planning is completed by a multi-core GPU server.

BERT. BERT is chosen to be tested for the planning time of Piper and HSM2DL because BERT is the largest DNN for which Piper can return a plan in a reasonable time (<96 hours). Results are summarized in Table 6.2. BERT with 64 layers is tested in the first set of experiments, and this model has 11.8B parameters. Different numbers of devices are tested to vary the complexity of the parallelism problem. Piper uses 208 seconds for 512 devices, and its time jumps to 6875 seconds for 2048 devices (These numbers are consistent with what can be observed in Figure 3 of the Piper paper[80]). HSM2DL is 255 times faster than Piper: it uses 104 seconds to return an optimized plan for 2048 devices, indicating the efficiency of its linear-time search algorithms.

In the second set of experiments, the number of devices is fixed at 1024, and the number of layers increased from 32 to 64. Piper uses 229 seconds for 32 layers and 1083 seconds for 64 layers, indicating a super-linear overhead in planning time w.r.t the number of layers. HSM2DL, however, uses only 18 seconds for 32 layers, and its time increases to 22 seconds for 64 layers. This shows linear-time complexity, mainly thanks to the hierarchical and symmetric system model defined in HSM2DL.

GPT-3. For GPT-3 with 39B parameters, HSM2DL can return a plan in 1.02 seconds while Alpa takes 804.5 seconds for extra profiling and 1582.66 seconds for computing a plan.

Table 6.1: DNN configurations and statistics

Model	#Layers	#Parameters
BERT	32, 48, 64	5.9B, 8.9B, 11.8B
$GPT3_{LARGE}$	32, 64, 128	100B, 200B, 400B
$GPT3_{MEDIUM}$	40	13B
$GPT3_{SMALL}$	32	2.6B

Table 6.2: BERT planning time with Piper and HSM2DL

Setting		Planning time (s)	
# Layers	# Devices	Piper	HSM2DL
64	512	208	**18**
	1024	1042	**22**
	2048	6875	**27**
32	1024	229	**18**
48		333	**21**
64		1083	**22**

GPT3-Large with 100B, 200B, and 400B parameters, one of the largest DNN models nowadays, is chosen to test HSM2DL's planning time. Both Alpa and Piper cannot return plans for such gigantic DNNs in a reasonable time (>96 hours), and thus only the results of HSM2DL are reported. Figure 6.5 shows the results. For GPT3-Large with 100B parameters, HSM2DL can return an optimized plan for 512 devices in 6 seconds, increasing to 9 seconds for 2048 devices. For GPT3-Large with 400B parameters, HSM2DL can return a plan for 2048 devices using less than 40 seconds.

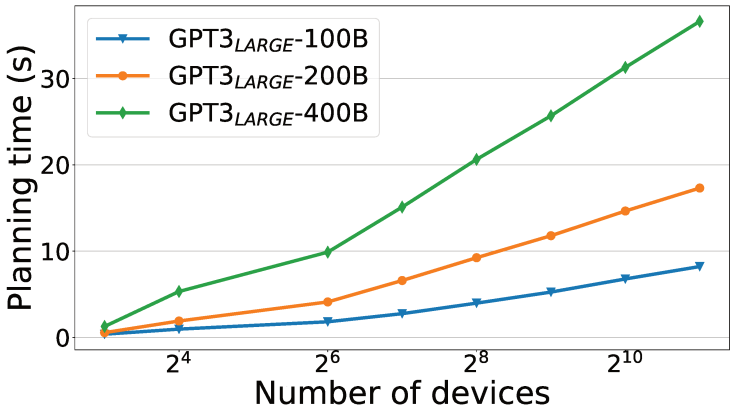


Figure 6.5: HSM2DL planning time for GPT-3.

6.4.3 Plan quality

The quality of the parallel plans computed by HSM2DL is then evaluated and compared with Megatron-LMv2 and Megatron-LMv2-Tuned. Piper and Alpa are excluded here. Piper is a prototype planner, and it is not implemented in DL frameworks. It is impossible to measure its performance in a real cluster. Alpa has a long profiling time for clusters with more than 64 GPUs. It thus fails to return a plan in a reasonable time. Also, according to their paper, Alpa exhibits similar performances as Megatron-LMv2, so it is sufficient to compare HSM2DL with Megatron-LMv2.

For the transformer-based DNN models, Megatron-LMv2 can be considered as the baseline for plan quality since experts have optimized it for years. However, the default setting of Megatron-LMv2 is customized for general configurations of the transformer-based DNN model. Its performance can be further fine-tuned as Megatron-LMv2-Tuned. HSM2DL is expected to return parallel plans that exhibit performance close to Megatron-LMv2-Tuned.

Table 6.3: Plan quality of transformer-based DNN models

Model	Setting	Throughput (sample/second)		
	# Devices	Megatron-LMv2	Megatron-LMv2-Tuned	HSM2DL
BERT-5.9B	8	297.75	**329.25**	320.25
BERT-8.9B	16	150.26	**168.28**	164.46
BERT-11.8B	64	86.80	91.01	**91.25**
GPT3-2.6B	8	13.12	**13.26**	13.12
GPT3-13B	16	4.57	4.57	**4.58**
GPT3-200B	256	3.15	5.17	**5.24**

Indeed, as discussed at the beginning of this chapter, for DNN models with overall model equilibrium, like BERT and GPT-3, finding the correct pipeline stage number and the stages' partition point is not the most critical challenge for parallel plan searching. As shown in Table 6.3, all three methods, in fact, find the same and an optimal number of stages (i.e., $\mathcal{P}_g$ is optimal for all of them). The difference in performance in Table 6.3 is caused by the difference in the operator-level plan for each stage.

According to Table 6.3, Megatron-LMv2-Tuned outperforms Megatron-LMv2 because the operator-level plan is optimized in all cases, particularly for the DNN model with a small size. The larger the size of the DNN model, the less evidence of performance improvement. This is because the difficulty of optimizing operator-level parallel plans increases exponentially as the network size grows, resulting in a tuning method that can not guarantee quality improvements. For example, for a very large network, an expert optimizing the performance of one module may cause the performance of other parts to drop, resulting in no overall performance improvement.

As expected, HSM2DL's performance matches Megatron-LMv2-Tuned with all BERT and GPT-3 configurations and even goes beyond their performance on large-scale models. This indicates that HSM2DL's joint search for the pipeline, model and data parallelism can indeed improve the quality of plans, matching those extensively optimized by experts. The most advantage of HSM2DL can be conducted here, it can infer the optimal parallel plans for the super large-scale DNN models in a reasonable time and even output the expert tuning. To the best of our knowledge, HSM2DL is the first planner to automatically produce a high-quality plan for a GPT-3 model with 200B parameters.

Here is an investigation into why HSM2DL performs better than Megatron-LMv2-Tuned: A key reason is that: the Megatron-LMv2-Tuned, tuning is at layer-level (more coarse-grain than operator-level) because of the limitation of human efforts. So the parallel plans for the operators in one layer must follow one specific rule, which is not universally optimal. The automatic joint-search algorithm helps to find what is hidden beyond the capacity of human efforts and can find better parallel plans for the operators than the specific rule.

6.5 Conclusion

This chapter proposes the graph-level partition method of HSM2DL, including the hypothesis, cost analysis, and algorithms. The graph-level searching space is also limited because of the hierarchical and symmetric abstract machine and also the restricted application domain of transformer-based DNN models. Large cluster experiments show that HSM2DL can outperform SOTA planners by up to 255 times when handling large-scale transformer-based models.

Chapter 7

Conclusion

7.1 Conclusion

The goal of this thesis is to use structured parallelism approaches to systematically optimize distributed deep learning so that DL frameworks can control the execution of training automatically. With the help of this work, DL designers could focus more on their expertise to create more precise DNNs without considering the implementation of parallel programming. This thesis introduced HSM2DL, a distributed DNN training bridging model based on the BSP bridging model. HSM2DL developed a hierarchical and symmetric abstract machine to describe the features of the modern AI training machine. HSM2DL also proposed an execution model to characterize the hybrid parallel training process, as well as a symbolic cost model as a metric to evaluate the cost produced by different parallel plans. HSM2DL avoids the pitfalls of existing methods that struggle to balance accuracy and portability.

HSM2DL aids in the discovery of optimal hybrid parallel plans. Recursive partitioning and the Flex-Edge Recursive graph are two important contributions to operator-level partitioning. Step by step, recursive partitioning partitionsăthe computational graph into two parts until the graph is partitioned into the number of devices. This recursive two-part partitioning is based on the feature of the symmetric abstract machine. The operators in the Flex-Edge Recursive graph can be reordered based on their compromise price. Therefore, backtracking is avoided during the searching process, reducing the searching complexity in relation to the number of operators to linear.ăăWith a linear searching complexity, the D-Rec algorithm can find an optimal operator-level parallel plan.

The role and effectiveness of all-reduce overlap techniques in distributed parallelism are discussed as a complement to operator-level parallelism. This thesis also provides a method for computing the tail factor, which can be used in conjunction with the D-Rec algorithm for arithmetic-level parallelism. Finally, this thesis presents a joint search method for pipeline plus operator-level hybrid parallelism. Based on the transformer model and the number of $2p$ devices, an algorithm for searching a triple-hybrid parallel strategy in conjunction with double recursion is provided.

Experiments in Chapter 4 show that HSM2DL can find high-quality parallel plans in seconds. The quality of the operator-level plans can match the SOTA expert-designed plans and is better than the related work TensorOpt [12]. The experiments in Chapter 5 show that an accurate tail factor γ can enhance the effectiveness of the D-Rec algorithm. In Chapter 6, under the hypothesis and strong assumptions of HSM2DL, the efficiency of the joint-search algorithm and the high plan quality are experimentally demonstrated in comparison with the SOTA method Alpa [97] and the plan quality can match the SOTA plans Megatron-lmv2 [58]. And it is very interesting and important further to develop the generality and portability of the pipeline partitioning.

7.2 Perspectives

Extend the generality of HSM2DL

Although HSM2DL proves its generality and portability in the operator-level search. But for the graph-level pipeline partitioning discussed in this thesis, the design of the model and algorithm is based on two stronger assumptions. First, this thesis assumed that the number of devices and the number of

stages are both a power of two. Another one is that this thesis only considers the transformer-based DNN model.

In production environments, users often give the number of devices as the power of two to best utilize hierarchical and symmetry data center networks. They, however, can also give non-prime numbers of devices. To address this, HSM2DL could decompose this nonprime number into several prime number levels. HSM2DL computes the plans for prime number level problems and combines the prime-level solutions. This thesis has demonstrated how to partition two parts as primitive. For a practical need, such primitive partitioning could be any number equal to or bigger than two, with cost model adjustment. Moreover, a cluster with a nonprime number of devices could always be decomposed to multiple prime number levels, and our algorithms can be adjusted for such settings. For a big cluster with a prime number of devices, we could approximate the number to the above possibilities by accepting a small bias of load balancing.

In order to support pipeline partitioning for the other types of DNN models, the execution model and cost model of pipeline parallelism need to be improved. When necessary, more profiling information may be added to keep the accuracy of the results.

Accurate memory estimation model

The cost model of HSM2DL mainly focuses on performance. The searching algorithm only checks if the computational graph trained with the found plan would exceed the device memory or not. Here, memory verification is used as a security check, but no memory cost model exists.

In practice, for large-scale DNN training, out-of-memory is one of the most frequent problems the users would meet. Trying to build an accurate memory cost model to estimate the expected memory usage and apply it to the joint search algorithm would be very helpful for HSM2DL. This is one of the most interesting perspectives for the auto parallel plan generation.

Guiding DL frameworks to support better parallel skeletons

The common industrial DL frameworks, all open source projects that are still in the development stage, have not been developed in a very long time. More specifically, the comparatively advanced DL frameworks first appeared in 2015, whereas most distributed training came to earth in 2019 and only supported data parallelism at that time. In fact, current DL frameworks do not yet fully support hybrid distributed parallelism. Their point-to-point communication and collective communication library is created case-by-case for expert-defined parallel plans. Because they first support parallel plans that have been demonstrated to be reliable. HSM2DL has the ability to guide the creation of more effective parallel skeletons in these areas and determine the best parallel plans.

Bibliography

[1] *MindSpore*. `https://www.mindspore.cn/`.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 2016.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[4] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv e-prints*, pages arXiv–1605, 2016.

[5] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[6] Y Bengio. Deep learning architectures for ai foundations trends mach. Learn, 2009.

[7] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[8] JA Bondy. Usr murty, graph theory with applications. *Published by T HE M ACM ILLAN*, 1976.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[11] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1967–1981, 2021.

[12] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1967–1981, 2022.

[13] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 571582, USA, 2014. USENIX Association.

[14] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 1993.

[15] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42, 2011.

[16] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[17] 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.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[19] Jack Dongarra. Report on the Fujitsu Fugaku system. *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06*, 2020.

[20] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[21] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2021.

[22] Michael J Flynn. Flynn's taxonomy., 2011.

[23] Jeremy Gibbons. Functional pearls: The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657665, 1996.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[26] JB Heaton, Nicholas G Polson, and Jan Hendrik Witte. Deep learning in finance. *arXiv preprint arXiv:1602.06561*, 2016.

[27] Jonathan MD Hill, Bill McColl, Dan C Stefanescu, Mark W Goudreau, Kevin Lang, Satish B Rao, Torsten Suel, Thanasis Tsantilas, and Rob H Bisseling. Bsplib: The bsp programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[28] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[29] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[30] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

[31] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[32] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[33] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[34] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[35] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In *International Conference on Machine Learning*, pages 2274–2283. PMLR, 2018.

[36] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In *International Conference on Machine Learning*, pages 2274–2283. PMLR, 2018.

[37] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[40] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[41] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[42] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[43] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE TPDS*, 31(1):94–110, 2019.

[44] Chong Li and Gaetan Hains. SGL: towards a bridging model for heterogeneous hierarchical platforms. *International Journal of High Performance Computing and Networking*, 7(2):139–151, 2012.

[45] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[46] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[47] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44. IEEE, 2019.

[48] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[49] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1):105–115, 2019.

[50] Grzegorz Malewicz, Matthew H Austern, AJ Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel. In *Proc. 28th ACM Symp. Princ. Distrib. Comput.-Pod*, volume 9, 2009.

[51] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[52] Marvin Minsky and Seymour Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 479:480, 1969.

[53] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[54] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. *Advances in neural information processing systems*, 27, 2014.

[55] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: Downward & upward lead to divide-and-conquer. *SIGPLAN Not.*, 44(1):177185, jan 2009.

[56] 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.

[57] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[58] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Nvidia. NVIDIA DGX-1 System Architecture White Paper. 2017.

[60] Even Oldridge, Julio Perez, Ben Frederickson, Nicolas Koumchatzky, Minseok Lee, Zehuan Wang, Lei Wu, Fan Yu, Rick Zamora, Onur Yilmaz, et al. Merlin: a gpu accelerated recommendation framework. *Proceeding s of IRS*, 2020.

[61] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. *arXiv preprint arXiv:1905.02494*, 2019.

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS '19)*. Curran Associates, Inc., 2019.

[63] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[64] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William S. Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *CoRR*, abs/2112.11446, 2021.

[65] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

[66] 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.

[67] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[68] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[69] Robert F. Rosin. *Von Neumann Machine*, page 18411842. John Wiley and Sons Ltd., GBR, 2003.

[70] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[71] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Andrew Fitzgibbon, and Tim Harris. Distir: An intermediate representation for optimizing distributed neural networks. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 15–23, 2021.

[72] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[73] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

[74] 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.

[75] Kamran Siddique, Zahid Akhtar, Edward J Yoon, Young-Sik Jeong, Dipankar Dasgupta, and Yangwoo Kim. Apache hama: An emerging bulk synchronous parallel computing framework for big data applications. *IEEE Access*, 4:8879–8887, 2016.

[76] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[77] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[78] David B Skillicorn and Domenico Talia. Models and languages for parallel computation. *Acm Computing Surveys (Csur)*, 30(2):123–169, 1998.

[79] Yu Sun, Shuohuan Wang, Shikun Feng, Siyu Ding, Chao Pang, Junyuan Shang, Jiaxiang Liu, Xuyi Chen, Yanbin Zhao, Yuxiang Lu, et al. Ernie 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation. *arXiv preprint arXiv:2107.02137*, 2021.

[80] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34:24829–24840, 2021.

[81] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators. *Advances in Neural Information Processing Systems*, 33:15451–15463, 2020.

[82] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[83] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[84] Leslie G Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154–166, 2011.

[85] 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.

[86] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ml, 2019.

[87] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[88] Siyu Wang, Yi Rong, Shiqing Fan, Zhen Zheng, LanSong Diao, Guoping Long, Jun Yang, Xiaoyong Liu, and Wei Lin. Auto-map: A dqn framework for exploring distributed execution plans for dnn workloads. *arXiv preprint arXiv:2007.04069*, 2020.

[89] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[90] James C Wyllie. The complexity of parallel computations. Technical report, Cornell University, 1979.

[91] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[92] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. Oneflow: Redesign the distributed deep learning framework from scratch. *arXiv preprint arXiv:2110.15032*, 2021.

[93] Anil Yuksel, Vic Mahaney, Chris Marroquin, Shurong Tian, Mark Hoffmeyer, Mark Schultz, and Todd Takken. Thermal and mechanical design of the fastest supercomputer of the world in cognitive systems: IBM POWER AC 922. In *ASME 2019 InterPACK*, 2019.

[94] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[95] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, et al. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. *arXiv preprint arXiv:2104.12369*, 2021.

[96] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, 2017.

[97] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 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, Carlsbad, CA, July 2022. USENIX Association.

Haoran WANG

Parallélisme implicite pour l'accélération de réseaux de neurones

Résumé:
L'intelligence artificielle (IA) s'est développé ces dernières années avec des succès spectaculaires et très médiatisés. Il est appliqué dans divers domaines. Les réseaux neuronaux a démontré des capacités d'apprentissage avec de très bonnes performances. Une tendance notable des réseaux neuronaux est leur augmentation exponentielle en taille. La formation d'un réseau étendu prend souvent des semaines, voire des mois. Les plus grands réseaux peuvent généralement dépasser les limites de la mémoire. Pour ces deux raisons, l'académie et l'industrie commencent à entrainer des réseaux neuronaux de manière distribuée. Les performances optimales d'un réseau neuronal complexe sont généralement obtenues en utilisant un mélange des méthodes de parallélisme ci-dessus, que l'on appelle parallélisme hybride. L'élaboration d'un plan de parallélisme requiert des connaissances en calcul parallèle pour les chercheurs en IA et nécessite également des efforts pour vérifier les performances. Des chercheurs ont proposé des méthodes telles que OptCNN, Tofu, Piper, Alpa, etc., qui peuvent donner automatiquement des stratégies hybrides quasi optimales. Cependant, leurs modèles de coût sont tous basés sur le temps d'exécution de l'opérateur de profilage sous une machine particulière. Ce type d'approche introduit un effort de préparation coûteuse sans garantie d'optimalité.
Cette thèse vise à contourner les inconvénients de la méthode de l'état de l'art et à fournir un moyen efficace de trouver un plan parallèle hybride précis. Basée sur le modèle BSP, cette thèse propose HSM2DL qui découple le matériel de l'algorithme parallèle, éliminant ainsi le besoin de profilage sur un matériel spécifique pour chaque opérateur. En se basant sur la sémantique des réseaux de neurones informatiques, le modèle de coût symbolique peut être transformé et réduit. Cette thèse propose un algorithme qui réduit la complexité des problèmes de recherche NP-hard à la linéarité et peut générer des algorithmes parallèles hybrides efficaces en quelques secondes.
Mots clés : Calcul parallèle, Entrainement réparti, Apprentissage profond

Implicit parallelism for neural network acceleration

Abstract :
The Artificial Intelligence (AI) field has been growing with spectacular, high-profile successes in recent years and is applied in varieties of fields. Neural network (NN) based deep learning has shown outstanding learning capabilities with very good performance. A noticeable trend in neural networks is their exponential increase in size. Training an extensive network often takes weeks or even months, and the larger networks may usually exceed the memory limits. For these two reasons, both academia and industry are beginning to train neural networks in a distributed way. Commonly partition methods used to distribute a neural network include data parallelism, operator-level model parallelism, pipeline model parallelism, etc. The optimal performance of a complex NN is usually obtained using a mixture of the above parallelism methods, which is called hybrid parallelism. Building a parallel plan requires parallel computing knowledge for AI researchers and also needs time and effort to design and verify performance. Academics have proposed methods such as OptCNN, Tofu, Piper, Alpa, etc., which can automatically give near-optimal hybrid plans. However, their cost models are all based on the execution time of the profiling operator under particular hardware. This kind of approach introduces an expensive preparation effort without optimality guarantees.
This thesis aims to circumvent the disadvantages of the state-of-the-art method and provide an efficient way to find an accurate hybrid parallel plan. Based on the BSP model, this thesis proposes HSM2DL that decouples the hardware from the parallel algorithm, thus eliminating the need for profiling on specific hardware for each operator. Based on the semantics of computing neural networks, the symbolic cost model can be transformed and reduced. This thesis proposes an algorithm that reduces the complexity of NP-hard search problems to linearity and can generate efficient hybrid parallel algorithms in seconds.
Keywords : Parallel computing, Distributed training, Deep learning

Laboratoire d'Informatique Fondamentale d'Orléans