Automated and Portable Machine Learning Systems

Byungsoo Jeon

CMU-CS-24-122
May 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Tianqi Chen (Co-Chair)
Zhihao Jia (Co-Chair)
Gregory R. Ganger
Luis Ceze (University of Washington)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2024 Byungsoo Jeon

This research was sponsored by: the National Science Foundation under award number CNS2211882, the University of Washington under award number UWSC13112, the National Science Foundation under award number CNS-2147909, the DARPA under the Real Time Machine Learning (RTML) project, the Qualcomm Innovation Fellowship, and the Kwanjeong Scholarship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.
Keywords: Machine Learning, Large Language Model, Distributed System, Compiler, Portability, Automatability, Parallelism, Operator Fusion, Deep Learning Backend
To my family.
Abstract

The landscape of ML ecosystem including models, software, and hardware evolves quickly due to phenomenal growth of Machine Learning (ML) and its application. Nevertheless, it remains challenging and labor-intensive to swiftly adapt existing ML systems to new models and hardware to maximize performance. We find that it is attributed to existing ML systems falling short in portability and automatability across several crucial layers of a system stack. However, building a portable ML system requires non-trivial modeling of intricate commonalities and differences of diverse ML models or platforms. In addition, automating ML system layers introduces the challenge of designing practical search space and search algorithms to customize optimizations to a given model and hardware.

In this thesis, we aim to tackle the challenges above of building an automated and portable ML system with a focus on crucial ML system layers. Specifically, the thesis explores ways to build an efficient system that automates 1) integration of ML backends and 2) ML parallelisms and makes them more portable. We develop a user interface and system stack to be more portable across different backends and underlying hardware. We also design practical search space and algorithms to automate backend placement and parallelism.

First, we built Collage, a DL framework that offers seamless integration of DL backends. Collage provides an expressive backend registration interface that allows users to precisely specify the capability of various backends. By leveraging the specifications of available backends, Collage automatically searches for an optimized backend placement strategy for a given workload and execution environment.

Second, we developed GraphPipe, a distributed system that enable performant and scalable DNN training. GraphPipe automatically partitions a DNN into a graph of stages, optimizes micro-batch schedules for these stages, and parallelizes DNN training. This generalizes existing sequential pipeline parallelism and preserves the inherent topology of a DNN, resulting in reduced memory requirement and improved GPU performance.

Lastly, we conducted a comparative analysis of parallelisms in distributed LLM inference for long sequence application. Specifically, we focused on Cache Parallelism (CP), a scheme to parallelize long KV cache in auto-regressive decoding step in LLM inference. We investigated trade-offs from different parallelisms for long context scenarios where we need to process tens of thousands tokens.
Acknowledgments

PhD journey has been truly humbling experience for me. I can’t believe how little I was when I dared to start this journey and how lucky I have been to meet many great people who helped me grow and reach this moment. Here, I would like to express my gratitude for those great people I met.

First and foremost, I would like to sincerely thank my advisors, Tianqi Chen and Zhihao Jia. I genuinely appreciated their passion, patience, support, and insightful comments throughout my PhD journey. I still remember TQ’s email about how to get started in the field of ML system. In that email, he asked me to start with having comprehensive understanding of the field rather than narrowing down to a specific project, which I believe is a way to grow as a great researcher. Still, he also emphasized how important it is to get my hand dirty with implementation and optimization details to realize an impactful ML system. It has been an invaluable time to learn how he approaches research problems including how to break down a big project into small action items and balancing between speculation and implementation.

I am also deeply grateful for having Zhihao as my advisor. When I worked on my second MLSys project, I had a lot of tough moments that makes me doubt the value of my work and my decision to dig deeper in this project. Without Zhihao’s encouragement and support, I couldn’t have wrapped up that project. He has always been there for me to spend time on discussing ideas, paper writing, and even debugging issues in our system. It has been a blessing to me to work closely with such a brilliant and warm-hearted researcher.

I would also like to thank my other thesis committee members, Greg Ganger and Luis Ceze, for their constructive feedback and encouragement. Greg’s advice on my second MLSys project guided me to spot motivation for my work and position it properly in the field. I will miss the moment in the meetings where Greg made cheerful jokes and lightened the mood for everyone. Luis’s feedback on my thesis proposal also guided me to shape my last research project and come up with a concrete action items. In addition, I won’t forget that Luis willingly wrote a recommendation letter for my green card application. His support for this meant the world to me and my family.

I also want to thank Jaime Carbonell, who advised me until he passed away in February 2020. He was not only brilliant, but also beloved advisor by all of his students. He has always been supportive of ambitious (but dumb) ideas that I had and helped me concretize it. I won’t forget his curiosity and passion towards ML research that he showed us until his last day.
I am thankful to the CMU Catalyst group members and visitors for the interesting discussions and conversations, including Sheng Xu, Neeraj Aggarwal, Peiyuan Liao, Mengdi Wu, Shiyi Cao, Xupeng Miao, Ameet Talwalkar, Pratik Fegade, Elias Jaasaari, Bohan Hou, Ruihang Lai, Hongyi Jin, Daiyaan Arfeen, Gabriele Oliaro, and many more.

I was fortunate to have fun with amazing collaborators and friends, who deserve my gratitude. I could not have come this far without their help and support. I am thankful to Yohan Jo, Dongyeop Kang, Jay-Yoon Lee, Namyong Park, Jinkyu Kim, Seojin Bang, Byungjoo Ahn, Juyong Kim, Hun Namkung, Dohyun Kim, Kijung Shin, Hyeju Jang, Shane Moon, Sunghyun Park, Sunghyun Kim, Sangkeun Choe, U Kang, Colin Unger, Bharathan Balaji, Saurabh Gupta, Chun Ye, Petros Christodoulou, Xiaoyu Lu, Jordan Bell-Masterson, Eyal Shafran, Luke Breitfeller, Jason Levin, Carolyn Rose, Graham Neubig, Qinlan Shen, and many more.

I would also like to thank Deborah Cavlovich, Catherine Copetas, Benjamin Cook, Nick Hernandez, and Tracy Farbacher for their administrative support and helping my studies at CMU run smoothly.

Finally, and most of all, I would like to thank my family members for their endless love, support, and belief in me throughout all stages of my life and career. I can’t wait to set my parents free from their life burden, and support them to enjoy their rest of life in every possible way! Lastly, I want to mention that I feel blessed to have Xilai Li as my wife, who listened to me whining, encouraged me whenever I doubted myself, and explored beautiful things in the world together with me.
# Contents

1 Introduction  ......................................................... 1
   1.1 Thesis Statement .................................................. 2
   1.2 Thesis Contributions .............................................. 3

2 Collage: Seamless Integration of Deep Learning Backends with Automatic Placement  ......................................................... 5
   2.1 Introduction .......................................................... 6
   2.2 Overview ............................................................ 9
   2.3 Backend Pattern Abstraction ........................................ 10
   2.4 Backend Placement Optimization .................................... 13
      2.4.1 Problem Definition ............................................ 13
      2.4.2 Op-level Placement Optimizer .................................. 13
      2.4.3 Graph-level Placement Optimizer ................................ 16
   2.5 Evaluation .......................................................... 17
      2.5.1 Experimental Setup ............................................ 18
      2.5.2 End-to-end Evaluation ........................................ 19
      2.5.3 Optimization Time ............................................... 20
      2.5.4 Backend Ablation Study ......................................... 22
      2.5.5 Case Study of Backend Operator Placement .................... 23
   2.6 Related Work ..................................................... 24
   2.7 Conclusion ........................................................ 26

3 GraphPipe: Improving the Performance and Scalability of DNN Training with Graph Pipeline Parallelism  ......................................................... 27
   3.1 Introduction ........................................................ 28
   3.2 Graph Pipeline Parallelism ........................................... 30
   3.3 Problem Formulation ................................................. 32
   3.4 System Overview .................................................... 34
   3.5 Pipeline Stage Partitioner ............................................ 35
   3.6 Static Micro-Batch Scheduler ........................................ 39
   3.7 Evaluation ........................................................ 42
      3.7.1 End-to-End Evaluation ........................................ 43
List of Figures

2.1 A comparison between existing DL frameworks and Collage. Existing frameworks (top) use rule-based heuristics to integrate different backends. In contrast, Collage provides an automatic search algorithm to find optimized placement of backends for a given hardware platform. New backends can be easily integrated into Collage through the backend registration interface. .......................................................... 7

2.2 Performance of various convolutions (C#) with different configurations (e.g., input tensor shape, kernel size) in ResNext-50 on NVIDIA RTX 2070; Note that there is no single backend that is the best for all convolutions. 8

2.3 System overview of Collage. By using our backend specification interface, users can efficiently register diverse backend patterns supported by diverse backends. Then, with its two-level optimization process, Collage automatically optimizes backend placement for an underlying execution environment. .......................................................... 9

2.4 Example illustrating how the backend pattern generator would automatically generate valid patterns with the pattern rule presented in Listing 1. . 12

2.5 Example of Dynamic Programming (DP) procedures. By visiting over each frontier node, DP algorithm matches backend patterns and update the optimized placement and its cost. For simplicity, optimized placement update is omitted. .......................................................... 14

2.6 Example of Evolutionary Search (ES) procedure. After pruning search space, it iterates over mutation, selection, and crossover until it reaches saturation or time limit. .......................................................... 17

2.7 End-to-end performance of state-of-the-arts DL frameworks and Collage in five real-life workloads on NVIDIA GPUs and Intel CPU. Throughput of each framework is normalized by the throughput of Collage. Following backends are employed for each framework according to target hardware and its capabilities: NVIDIA GPU (cuDNN, cuBLAS, TVM, TensorRT), Intel CPU (MKL, DNNL, TVM). .......................................................... 19

2.8 End-to-end performance with different batch sizes in ResNeXt50 on NVIDIA V100. Normalized throughput is the throughput normalized by the throughput of Collage. .......................................................... 20
2.9 (Left) On average, profiling overhead for operator cost measurements takes up 68% of the entire optimization time. Note that profiling is only necessary for unseen operators. Once the cost of a new operator is measured, its information will be saved in the logging database in Collage to avoid the repetitive profiling. If profiling log is available, op-level optimizer only takes less than a minute. (Right) The y-axis presents the speedup relative to the op-level placement optimization.

2.10 Performance improvement of graph-level placement optimization over time. The y-axis presents the speedup relative to the op-level placement optimization.

2.11 End-to-end performance of Collage with different number of backends on NVIDIA Tesla V100. Each throughput is normalized by the throughput of Collage (TVM, cuB, cuD, TRT). TVM, cuB, cuD, and TRT represents TVM, cuBLAS, cuDNN, and TensorRT.

2.12 Representative backend placements discovered by Collage on V100 (Figure 2.7a). Note that Collage leverages various backends given their unique strength to enhance performance.

3.1 Pipeline parallelism for DNN training with basic terms used in this paper.

3.2 A high-level comparison between existing (SPP) and our (GPP) approaches. SPP (top) produces sequential pipeline stages that miss the opportunity of parallelizing the branches in the DNN. In contrast, GPP (bottom) generates graphical pipeline stages that enable parallel execution of the branches. This leads to lower training iteration time (i.e., higher training throughput) and smaller memory footprint in pipeline-parallel DNN training.

3.3 Overview of GRAPHPIPE

3.4 Pipeline stage partitioner performing series-parallel decompositions. Black arrows indicate subproblem formulations. Red arrows indicate solutions of the subproblems.

3.5 A comparison between 1F1B and kFkB schedules when stages have different micro-batch sizes. F\{i, j\}, B\{i, j\} indicate forward and backward passes for a micro-batch including samples i and j. It showcases how kFkB can be better than 1F1B in terms of memory footprint.

3.6 End-to-end performance evaluation. GRAPHPIPE outperforms both PipeDream [NPS+21] and Piper [TNP21] in three different models: Multi-modal Transformer-based model [RKH+21], DLRM [NMS+19], and CANDLE-Uni [201] at all but one GPU count configurations tested. Missing data points indicate that no training strategy can be found within reasonable timeframes.

3.7 Throughput vs. different numbers of branches using 4, 8, 16 GPUs respectively (left). Throughput vs. different micro-batch sizes using 8 GPUs (right).
3.8 A synthetic Transformer-based two-branch DNN for case study. A sequence of one multi-head attention and two linear layers is repeated four times to compose a single branch. One concatenation layer at the end combines the branches. ........................................ 46

3.9 Pipeline schemes devised by SPP (top) and GRAPHPIPE (bottom). They produce an identical model partition. The selected micro-batch sizes are different: 2 (SPP) v.s. 4 (GRAPHPIPE), which results in a better compute efficiency for GRAPHPIPE. Both methods deem it unnecessary to employ data parallelism primarily because doing so would have split a smaller micro-batch size even further, which would have harmed compute efficiencies. The pipeline depths are also different: 8 (SPP) v.s. 4 (GRAPHPIPE), which results in a smaller pipeline depth for GRAPHPIPE. This improvement comes purely from the fact that GRAPHPIPE can produce a pipeline scheme that allows for concurrent execution of parallel branches. ........ 47

4.1 Total amount of data transfer per GPU for TP and CP ...................... 58
4.2 Throughput (tokens per second per layer) of varying parallelisms across different prefix sequence length. ................................. 59
4.3 Throughput (tokens per second per layer) of varying parallelisms across different decode sequence length. ............................... 60
4.4 Throughput and latency trade-offs of varying parallelisms across different sequence length. .................................................. 60
4.5 Breakdown of execution time into three different categories: GPU Compute, CPU Overhead, and GPU (to GPU) Communication. .............. 61
List of Tables

3.1 Solution search times (in seconds) for Piper, PipeDream, and Ours (GRAPHPipe) on the Apple M1 Max; ✗ indicates search cannot be completed. Numbers in parentheses indicate the search time ratio of the algorithm to that of GRAPHPipe. ................................................. 44
3.2 Computation of the number of in-flight samples. ................................................. 52
Chapter 1

Introduction

The explosive popularity of Machine Learning (ML) drives the rapid advance in ML models, systems, and hardwares. As recent ML models impress the world with human-level performance in a variety of tasks such as self-driving, SAT and bar exam, the diversity of models and their applications grows quicker than ever. Naturally, ML hardware and systems have been diversified and fast-evolving in response to fast-growing demand for new models and applications. However, it is still labor-intensive and challenging to quickly adapt ML systems to new models and HWs with a goal of maximizing performance. Imagine new ML models or HW generations come out. Hardware vendors would implement and release newly optimized operator kernels while ML system engineers would manually adjust system implementation and optimizations to accommodate them.

Still, we observe that existing ML systems often fall short in two essential aspects to keep up with the rapid evolution of ML: portability and automatability. While portability is loosely defined term in general, we define it as an ability of a ML system to adapt to a new ML model or underlying execution environment (e.g., HW cluster). For example, a portable ML system should be able to support new models and corresponding updates with minimal effort including code changes from system engineers and inputs from users. On the other hand, we refer automatability to an ability of a ML system to automate compilation of ML models specifically regarding optimization (e.g., operator fusion, parallelism).

It is challenging to build a ML system with both qualities. First, building a portable ML system requires effective modeling of commonalities and differences of different ML models or platforms. For instance, assume a scenario of building a ML system efficiently employing diverse ML backends\(^1\) [CWV+14, WZS+14, KFT+19, Tena, One] for different ML models and platforms (e.g., NVIDIA GPU, Intel CPU, etc). However, it is non-trivial how to capture and represent various capabilities of backends that vary depending on the

\(^1\)We define a backend as a kernel library [CWV+14, WZS+14] or a runtime framework [Tena, KFT+19] that takes DL workloads as inputs and provides an optimized low-level target code.
platform. The example capability includes complicate operator fusion rules with various constraints \([\text{CMJ}^{+18}, \text{NGW}^{+21}, \text{MXY}^{+20}, \text{ZZL}^{+20}, \text{ELB}^{+17}]\), e.g., fusing convolution ops with 3x3 kernel and data layout constraint.

In addition, automating optimizations in ML system introduces a challenge of designing 1) practical search space and 2) search algorithm. In ML systems, naive optimization space is often extremely large, whose size often grows exponentially in the number of operators in a ML model and the number of devices to use. Therefore, it is crucial to leverage domain-specific knowledge to narrow down search space while still being comprehensive enough not to leave the performance on the table. Then, we face a challenge of designing a reasonable cost model that efficiently evaluates possible optimization strategies. Lastly, there is a challenge of how to design an efficient search method that effectively prunes worthless candidates and leverages optimal substructure.

Specifically, we mainly focus on addressing such challenges in optimizing ML parallelisms. Parallelisms are fundamental for optimizing the performance of large-scale ML models. By distributing computations across multiple devices, we can perform training or inference of models faster and handle models or datasets too big to fit on a single machine. Data parallelism \([\text{Val}^{90}, \text{Kri}^{14}, \text{LAP}^{+14}, \text{GDG}^{+17}, \text{MHH}^{+21}]\), where we split the dataset across devices, and model parallelism \([\text{DCM}^{+12}, \text{SCP}^{+18}, \text{SPP}^{+19}, \text{LLX}^{+20}, \text{NSC}^{+21}, \text{ZLZ}^{+22}]\), where we split the model itself and pipeline them \([\text{HCB}^{+19}, \text{FRM}^{+21}, \text{NPS}^{+21}, \text{NHP}^{+19}, \text{TNP}^{+21}]\), are the foundational approaches. However, optimizing these strategies poses significant challenges due to the complex search space involved. Factors like model architecture, communication overhead, and hardware constraints all interact in intricate ways \([\text{ZLZ}^{+22}]\). It is a certainly non-trivial task to find the ideal balance between data distribution, model partitioning, and device synchronization that yields the best performance gains and resource efficiency \([\text{QCS}^{+21}]\).

1.1 Thesis Statement

The thesis centers around tackling the aforementioned challenges of building an automated and portable ML system with a focus on key ML system layers. Specifically, the thesis explores ways to build an efficient system that automates 1) integration of ML backends and 2) ML parallelisms and makes them more portable. We develop an user interface and system stack to be more portable across different backends and underlying hardwares. We also design search space and algorithms to automate backend placement and parallelism.

Automated and portable integration of ML backends. To keep up with the fast advancement of Deep Learning (DL) backends, it is crucial for modern DL frameworks to efficiently integrate a variety of optimized tensor algebra libraries \([\text{CWV}^{+14}, \text{WZS}^{+14}, \text{KFT}^{+19}]\) and runtimes \([\text{Tena}, \text{KFT}^{+19}]\) as their backends and generate the fastest possible executable using these backends. However, current DL frameworks \([\text{PGM}^{+19a}, \text{ABC}^{+16}, \text{XLA}, \text{RFA}^{+18}, \text{CBB}^{+18}, \text{RKBA}^{+13}]\) require significant manual effort and expertise to integrate every new backend while failing to unleash its full potential. Given the fast-evolving nature of the DL ecosystem, this manual approach often slows down continuous
innovations across different layers; it prevents hardware vendors from the fast deployment of their cutting-edge libraries, DL framework developers must repeatedly adjust their hand-coded rules to accommodate new versions of libraries, and machine learning practitioners need to wait for the integration of new technologies and often encounter unsatisfactory performance. We propose Collage, a DL framework that offers seamless integration of DL backends. Collage provides an expressive backend registration interface that allows users to precisely specify the capability of various backends. By leveraging the specifications of available backends, Collage automatically searches for an optimized backend placement strategy for a given workload and execution environment.

Automated and portable ML parallelisms. Pipeline parallelism \cite{HCB+19,FRM+21,NPS+21,NHP+19,TNP21} is commonly used in existing DL systems to support large-scale Deep Neural Network (DNN) training by partitioning a DNN into multiple stages, which concurrently perform DNN computation for different micro-batches of training samples in a pipeline fashion. However, existing pipeline-parallel approaches only consider sequential pipeline stages and thus ignore the topology of a DNN, resulting in missed model-parallel opportunities. To overcome such limitation, we present graph pipeline parallelism (GPP), a new pipeline-parallel scheme that partitions a DNN into pipeline stages whose dependencies are identified by a directed acyclic graph. GPP generalizes existing sequential pipeline parallelism and preserves the inherent topology of a DNN to enable concurrent execution of computationally-independent operators, resulting in reduced memory requirement and improved GPU performance. In addition, we develop GraphPipe, an automated distributed system that exploits GPP strategies to enable performant and scalable DNN training on device clusters with any network topology.

Parallelism for Emerging LLM Application. Many emerging large language model (LLM) tasks, like writing long documents or translating books, involve processing huge amounts of text. However, the memory-intensive way LLMs work and the unpredictable length of generated text create performance bottlenecks. Existing methods for parallelizing LLM inference across devices have drawbacks, especially with long sequences. We introduce cache parallelism (CP), a new approach that partitions the KV cache for attention for better workload balance and less communication overhead. We implement custom CUDA kernels that boosts CP’s efficiency. CP demonstrates improved performance over other methods when handling text generation, particularly for long sequences.

1.2 Thesis Contributions

In this section, we summarize the contributions of the thesis. We present three pillar works that contribute to building an automated and portable ML system across chapters in the thesis.

In Chapter 2, we propose a ML system, Collage, that makes following research contributions:
• We identify system and optimization challenges in integration of diversified DL backends and build Collage to tackle these challenges.
• We provide a pattern-based interface for quick registration of various backends and their updates with significantly less user efforts and expertise in performance landscape of varied backends and the placement heuristic in the framework codebase.
• We develop a two-level search method to automatically optimize placement of diverse backends for a given hardware.

In Chapter 3, we introduce our ML system, GraphPipe. The key contributions are
• We introduce graph pipeline parallelism, a new pipeline-parallel scheme that enables concurrent execution of stages, reduces memory requirement, and improves GPU performance compared to existing SPP strategies.
• We design algorithms to partition a DNN into a graph of stages and schedule micro-batches for these stages, which jointly discover efficient GPP strategies.
• We develop GraphPipe, a distributed runtime that enables fast and scalable DNN training leveraging GPP; it outperforms existing pipeline-parallel systems by up to $1.6\times$ on a variety of DNNs.

In Chapter 4, we analyze and compare different parallelism strategies for efficient distributed LLM inference in long sequence applications:
• We explore Cache Parallelism (CP) as a strategy to improve the efficiency of LLM inference when dealing with extremely long input sequences.
• We compare how various forms of parallelism impact LLM inference performance in long context scenarios, requiring the processing of tens of thousands of tokens.
• We provide insights for factors that impact performance of long context LLM inference. We also analyze trade-offs between different parallelism approaches.
Chapter 2

Collage: Seamless Integration of Deep Learning Backends with Automatic Placement

The strong demand for efficient and performant deployment of Deep Learning (DL) applications prompts the rapid development of a rich DL ecosystem. To keep up with this fast advancement, it is crucial for modern DL frameworks to efficiently integrate a variety of optimized tensor algebra libraries and runtimes as their **backends** and generate the fastest possible executable using these backends. However, current DL frameworks require significant manual effort and expertise to integrate every new backend while failing to unleash its full potential. Given the fast-evolving nature of the DL ecosystem, this manual approach often slows down continuous innovations across different layers; it prevents hardware vendors from the fast deployment of their cutting-edge libraries, DL framework developers must repeatedly adjust their hand-coded rules to accommodate new versions of libraries, and machine learning practitioners need to wait for the integration of new technologies and often encounter unsatisfactory performance.

In this paper, we propose **Collage**, a DL framework that offers seamless integration of DL backends. **Collage** provides an expressive backend registration interface that allows users to precisely specify the capability of various backends. By leveraging the specifications of available backends, **Collage** automatically searches for an optimized backend placement strategy for a given workload and execution environment. Our evaluation shows that **Collage** outperforms the best existing framework for each hardware by 1.26×, 1.43×, 1.40× on average on NVIDIA’s RTX 2070 GPU, V100 GPU, and Intel’s Xeon 8259CL CPU, respectively. Collage has been open-sourced and deployed in Apache TVM.

[^github]: https://github.com/cmu-catalyst/collage
2.1 Introduction

Due to the explosive popularity of Deep Learning (DL) applications, there are tremendous demands for performant and efficient software/hardware stacks for DL computations. These strong demands have driven both industry and academia to invest a significant amount of effort in developing various hardware devices [JYP+17, ANE, NVD], software libraries [CWV+14, One, Tena, Opeb, KFT+19], compilers [AMA+19, BRR+19, ZJS+20, KKC+17, PSS+21, MYS20, CGG+20, RSG19, BZR+22, CCC+19, LCC+21], and DL frameworks [PGM+19a, ABC+16, XLA, RFA+18, CBB+18, RKBA+13]. Both the hardware and software stacks for DL have been diversified, resulting in a rich and fast-evolving ecosystem.

Within this ecosystem, today’s DL frameworks can leverage a variety of optimized software libraries [CWV+14, WZS+14] and runtimes [Tena, KFT+19] as their backends\(^1\) to deliver fast execution. Existing backends can be grouped into two categories based on their capabilities. First, **operator kernel libraries** [CWV+14, WZS+14, KFT+19] provide efficient low-level kernel API for individual DL operators (e.g., convolution). These libraries often support **operator fusion**, which combines multiple operators into a single kernel based on certain fusion rules (e.g., cuDNN fusion engine) [CMJ+18, CWV+14, NGW+21, MXY+20, ZZL+20, ELB+17]. Second, **graph inference libraries** [Tena, One] take an entire DL model as input and produce efficient run-time code. In addition to the optimizations that operator kernel libraries provide, the graph inference libraries also consider graph-level cross-kernel optimizations, such as memory optimizations [SRSA21].

There are strong demands for high-performance DL backends in both industry and academia. However, seamless integration of diverse and rapidly advancing DL backends requires addressing two key challenges: (1) incorporating a wide variety of available backends with different programming models and performance characteristics, and (2) optimizing placement of backends to effectively assign DL computations to various backends by leveraging the performance advantages of each backend. We refer to this overall problem as **backend integration problem**.

To solve the backend integration problem, existing DL frameworks [ABC+16, PGM+19a] rely on rule-based heuristics manually designed by experts (Figure 2.1). These heuristics often directly offload the entire workload to a single backend (e.g., TensorRT) whenever applicable. Otherwise, DL frameworks lower individual operators to different backends based on a fixed priority-based strategy; for example, in PyTorch, cuDNN has the highest priority for convolution, while cuBLAS is the first choice for matrix multiplication.

However, even for the same type of operators, the optimal backend varies depending on the hardware (e.g., different types of GPUs) and operator configuration (e.g., tensor shape, padding) as depicted in Figure 2.2. As a result, the hand-coded heuristics in current DL frameworks may leave substantial performance on the table. Besides, existing frameworks require significant expertise in both framework and performance

\(^1\) We define a backend as a kernel library or a runtime framework that takes DL workloads as inputs and provides an optimized low-level target code.
Figure 2.1: A comparison between existing DL frameworks and Collage. Existing frameworks (top) use rule-based heuristics to integrate different backends. In contrast, Collage provides an automatic search algorithm to find optimized placement of backends for a given hardware platform. New backends can be easily integrated into Collage through the backend registration interface.

landscape of diverse backends as developers need to directly modify the complex lowering heuristics (e.g., more than ten thousand lines of code in PyTorch) in a framework to introduce a new backend or reflect any backend updates. These handcrafted heuristics are hard to maintain and keep up with the rapid developments in backends. This is a major bottleneck for various machine learning personas, since the integration workflow requires repetitive manual efforts to accommodate new backends. This integration overhead hinders hardware vendors from deploying their cutting-edge libraries and delays machine learning practitioners from employing newest system-level supports.

In this paper, we aim to design a system that can provide seamless backend integration workflow with high performance. Building such a solution requires addressing two key challenges. First, it is non-trivial to integrate diverse backends with different characteristics into a system while maintaining their full capabilities. Often times, backend capability is intricate to capture accurately since today’s DL backends generally support sophisticated operator fusion with various constraints (e.g., fusing convolution ops with 3x3 kernel). Second, the search space of backend placement is extremely large, whose size grows exponentially in the number of operators in a DNN and the number of available backends. The search space is also highly irregular due to diverse backend capabilities and operator fusion patterns.

In Collage, we advocate for a new approach to tackling these challenges, as shown
in the bottom of Figure 2.1. **Collage** contains two key components. First, to integrate diversified backends, **Collage** provides a descriptive backend registration interface to specify a backend’s capability based on its supported operator type (e.g., conv), configurations (e.g., kernel size), and its fusion rule. This interface only requires basic understanding of our pattern language and backend capability in contrast to existing frameworks that require considerable expertise in both the performance landscape of varied backends and the coding skills for backend placement rules in existing frameworks. **Collage** allows easy backend registration for a new backend (e.g., 100 LoC for all possible operators) or a new operator pattern support (e.g., 1 LoC in most cases). Second, to efficiently optimize backend placement, **Collage** employs a two-level optimization to deal with unique characteristics of two backend categories (i.e., operator kernel library and graph inference library). Our system automatically explores possible matches between an input computation graph and backend operator patterns to find optimized placements by taking available backends and an underlying hardware into consideration.

To sum up, **Collage** significantly lowers the bar in the current backend integration workflow by eliminating the need to modify the placement heuristic. With simple registration from users, **Collage** can immediately launch the automatic placement optimizer without any intricate manual consideration for the capability of new backend and its performance relation with other backends across different workloads and hardware architectures.

This paper makes the following contributions:

- We identify system and optimization challenges in integration of diversified DL backends and build **Collage** to tackle these challenges.
- We provide a pattern-based interface for quick registration of various backends and their updates with significantly less user efforts and expertise in performance landscape of varied backends and the placement heuristic in the framework codebase.
- We develop a two-level search method to automatically optimize placement of
Figure 2.3: System overview of Collage. By using our backend specification interface, users can efficiently register diverse backend patterns supported by diverse backends. Then, with its two-level optimization process, Collage automatically optimizes backend placement for an underlying execution environment.

Our evaluation shows that Collage stably outperforms existing DL frameworks across a variety of models and hardware architectures by effectively mix-using multiple backends with their own unique strengths. On average, Collage brings $1.26 \times$, $1.43 \times$, and $1.40 \times$ speedup on two different NVIDIA GPUs and an Intel CPU respectively, compared to the best framework for each hardware.

### 2.2 Overview

Figure 2.3 illustrates the overarching design of Collage, which takes a DNN model and the specifications of available backends as inputs, and optimizes backend placement for the underlying hardware. Note that Collage considers different sets of backends based on a given target environment (e.g., Intel CPU, NVIDIA GPU) and reflects performance characteristics of backends via the measurer component ($M$). Collage consists of two key components.

**Backend pattern abstraction.** Existing backends provide a variety of programming models for performing DL computations. To decouple backend capability from the placement algorithm and eliminate the manual effort for backend integration, we introduce backend pattern, a new abstraction for capturing the capability of varied backends. Specifically, a backend pattern defines a set of operators and their possible fusion combinations (e.g., Conv+ReLU) that can be deployed on each backend. Based on this pattern abstraction, Collage provides a straightforward interface to register a backend and specify supported operator patterns.

Accurate specification is crucial to leverage the full capability of diverse backends. To achieve this goal, Collage offers two levels of abstraction. For simple patterns, Collage allows users to enumerate the supported operator patterns. However, this approach may
not cover the full capability of backends with advanced operator fusion engines \([\text{CMJ}^{+}18, \text{NGW}^{+}21, \text{CWV}^{+}14, \text{Tena}]\). To enable more flexible specification, \textit{Collage} also allows users to bring their pattern rules that specify supported operator kinds and complex operator fusion rules. When those rules are provided, the \textit{pattern generator} automatically identifies all legitimate operator fusion patterns on a given computation graph and adds them into the backend pattern registry. §2.3 provides details.

**Backend placement optimizer.** Once all available patterns are registered in the pattern registry, \textit{Collage} uses a \textit{two-level optimization} approach to discovering an optimized backend placement strategy for a given execution environment. As existing operator libraries offer operator-level point of view while graph inference libraries additionally apply cross-kernel optimizations, \textit{Collage} takes two different optimization strategies to exploit their differences. First, the \textit{op-level placement optimizer} explores promising candidates for individual operators, without considering cross-kernel optimizations. By adopting a Dynamic Programming (DP) algorithm, the op-level placement optimizer can efficiently find an optimized backend placement strategy within a minute. Second, the \textit{graph-level placement optimizer} fine-tunes the optimized backend placement using evolutionary search \([\text{FDRG}^{+}12]\). This approach compensates for the missing opportunities from the op-level placement optimizer by examining the impact of cross-kernel optimizations. §2.4 discusses the two optimizers in detail.

### 2.3 Backend Pattern Abstraction

As an important component of DL ecosystem, there are diverse fast-evolving DL backends with different programming models and performance characteristics. Depending on their target hardware and design principles, each backend has its own unique strength and coverage. In addition, many backends support various complex operator fusion rules \([\text{CWV}^{+}14, \text{CMJ}^{+}18, \text{Tena}, \text{CMJ}^{+}18, \text{NGW}^{+}21]\), which add significant complexity in their integration with the full capability. Under the hood, existing operator fusion engines often fuse operators based on heuristic fusion rules that examine the type of each operator and the relationship between different types. For instance, a fusion engine may combine multiple operators across different branches into a single kernel as long as they satisfy its fusion rule.

For an adoption of various backends, our system provides two levels of abstraction: \textit{pattern} and \textit{pattern rule}. Pattern is a direct way to specify all supported operator patterns in \textit{Collage}'s pattern language, which extends the Relay pattern language \([\text{RLW}^{+}18]\). However, supported patterns can be too complicated to explicitly specify. To incorporate sophisticated patterns, pattern rules offer an expressive way to specify a valid set of operator fusion rules in the form of Python; users can use any Python features to describe complex fusion algorithms. Each pattern rule is used to generate valid patterns for the input workload with our automatic pattern generator. With two levels of abstraction, users can easily incorporate an additional backend by specifying its patterns and pattern rules with an intuitive programming interface. By default, \textit{Collage} provides built-in patterns and pattern rules for popular backends \([\text{CWV}^{+}14, \text{cuB}, \text{Tena}, \text{WZS}^{+}14, \text{CMJ}^{+}18]\).
import collage

# [Method 1] Explicit pattern specification
# Pattern language to describe conv2d + add + relu.
conv = is_op('conv2d')(wildcard(), wildcard())
conv_constr = conv.has_attr({"data_layout": "NCHW"})
conv_add = is_op('add')(conv_constr, wildcard())
conv_add_relu = is_op('relu')(conv_add)

# Introduce new backend pattern to Collage.
collage.add_backend_pattern(backend='cuDNN',
                           pattern=conv_add_relu)

# [Method 2] Pattern rule specification
class MyPatternRule(collage.BasePatternRule):
  # Define variables
  kFusable = 0
  kElemwise = 1
  # ...
  # Checker for the supported operators.
  @staticmethod
  def op_rule(op):
    if op.name == "dense":
      # Dense operator is always supported.
      return True
    elif op.name == "conv2d":
      # Constraints can be verified as well.
      return op.attr["data_layout"] == "NCHW"
    # ... rest of the op rule ...
    return False

  # Checker for fusion patterns.
  @staticmethod
  def fusion_rule(cur_type, src, sink):
    # If current fusion group contains
    # at least one conv/matmul (kFusable)
    if cur_type == MyPatternRule.kFusable:
      # Helper functions can be defined.
      def fchecker(node_pattern):
        return (node_pattern == MyPatternRule.kElemwise)
      # Check if every operator between src and sink.
      # Helper function can be passed as a checker.
      if collage.check_path(src, sink, fchecker):
        return True
    # ... rest of the fusion rule ...
    return False

# Introduce new pattern generation rule to Collage.
collage.add_backend_pattern_rule(backend='TVM',
                                  pattern_rule=MyPatternRule())

Listing 1: Example of the backend registration interface. To register a new backend, users
can directly enumerate patterns or write a pattern rule that consists of valid operator
checker and fusion rule in Python classes.

Listing 1 presents an example of use-case scenarios. If a backend only supports a few
simple patterns, users may enumerate those patterns and add them directly to the
backend pattern registry (line 3-12). Users can easily check the operators (line 5), their
configurations such as data layouts and kernel sizes (line 6), and the the relationship
Figure 2.4: Example illustrating how the backend pattern generator would automatically generate valid patterns with the pattern rule presented in Listing 1.

between operators (line 7-8). A wildcard operator is a special placeholder that matches any operator.

To fully support advanced backends [CMJ+18, NGW+21, CWV+14, Tena], users can bring their pattern rules to incorporate more complicated patterns with Collage’s pattern generator (line 14-53). To use this feature, users need to provide operator checkers with their potential constraints (line 20-30) and a fusion rule (line 32-49) in the form of Python methods. Then, the automatic pattern generator in Collage will search for valid operator patterns satisfying these rules and add them to the backend pattern registry before optimizing backend placement.

Figure 2.4 exhibits how our pattern generator searches for legitimate patterns using given pattern rules on an input computation graph. By visiting every operator in an input computation graph, the pattern generator investigates how far a pattern can grow without breaking the pattern rule. For each operator, the pattern generator first validates whether the operator can be executed on a backend (line 20-30). If valid, it enlarges the scope one step further and validates whether a set of operators satisfies the fusion rule (line 32-49). For instance, line 40-47 specify that the assumed backend can fuse element-wise operators following an operator of type kFusable, which includes convolution and matrix multiplication. Whenever a group of operators satisfying the rule is found, the pattern generator produces a corresponding pattern and adds it to
the backend pattern registry. Then, it enlarges the scope of interests one step further again to see if a bigger pattern can be found. This approach allows Collage to incorporate advanced backends, such as TVM, cuDNN, DNNL and TensorRT, without missing any pattern.

2.4 Backend Placement Optimization

2.4.1 Problem Definition

Collage attacks the backend placement problem to find the best use of available backends and maximize performance. Consider a computation graph $G$ and a set of backend patterns $B$ in Collage’s backend pattern registry. $G$ is a Directed Acyclic Graph (DAG) where each node represents a tensor operator (e.g., convolution, matrix multiplication). $b = (p, d) \in B$ is a pair of an operator pattern $p$ and a backend identifier $d$, such as cuDNN, cuBLAS, etc.

With $M$ matched subgraphs $g_i$ and backend patterns $b_i$ for $i \in \{1, 2, \cdots, M\}$, let $\mathcal{P}(G) = \{(g_i, b_i)|b_i \in B, \bigcup_{i=1}^{M} g_i = G, g_i \cap g_j = \emptyset \text{ for all } i, j \in \{1, 2, \cdots, M\} \text{ where } i \neq j\}$ be a backend placement strategy on a computation graph $G$ and $Cost(\mathcal{P}(G))$ be the execution time of a placement $\mathcal{P}(G)$. In this work, we aim to find a backend placement strategy $\mathcal{P}_{opt}$ that minimizes $Cost(\mathcal{P}(G))$. This problem can be formalized as follows:

$$\mathcal{P}_{opt}(G) = \arg\min_{\mathcal{P}(G)} Cost(\mathcal{P}(G))$$

(2.1)

2.4.2 Op-level Placement Optimizer

To efficiently evaluate numerous candidates with different placement and prune the search space, Collage conducts an op-level placement optimization as the first step. Its goal is to map all operators on the computation graph to the most efficient set of low-level kernel implementations from available backends fast without considering cross-kernel optimizations in graph inference libraries. As discussed earlier, the graph-level placement optimizer (§2.4.3) would make up for the possible performance loss from this simplification.

With this simplification, low-level kernel executions become independent to each other in a single device execution. Let $s_1$ and $s_2$ be subgraphs of $G$ where $s_1 \cup s_2 = G, s_1 \cap s_2 = \emptyset$. Then, the following additive relationship [JPT+19] between the run-time cost of $\mathcal{P}(s_1)$ and $\mathcal{P}(s_2)$ can be used to determine $Cost(\mathcal{P}(G))$:

$$Cost(\mathcal{P}(G)) = Cost(\mathcal{P}(s_1)) + Cost(\mathcal{P}(s_2)) + \epsilon$$

(2.2)

where $\epsilon$ is a context switching cost (e.g, driver overhead), which is nearly constant empirically. Note that Collage avoids data transfers between different backends on the same device by only exchanging data pointers to the tensors (e.g., $s_1$ and $s_2$) using the zero-copy mechanism. With this cost model, it is possible to cheaply approximate the cost of a graph by partitioning a graph into smaller subgraphs and summing up their
Figure 2.5: Example of Dynamic Programming (DP) procedures. By visiting over each frontier node, DP algorithm matches backend patterns and update the optimized placement and its cost. For simplicity, optimized placement update is omitted.

cost. Despite the efficient cost model, excessively large number of possible placement strategies and a variety of fusion patterns make search non-trivial.

To address this challenge, we propose a Dynamic Programming (DP) method for optimizing backend placement at the operator level. By using the additive relation (Equation 2.2), we deduce the following recurrence relation of optimized backend placement $P_{opt}(s)$ and its cost $C_{opt}(s)$ for any subgraph $s \subset \mathcal{G}$. This breaks down a problem of finding $P_{opt}(\mathcal{G})$
into smaller problems of finding $P_{opt}(s)$.

$$P_{opt}(s) = P_{opt}(s_{min}) \cup P(g_{min})$$

$$C_{opt}(s) = \begin{cases} 0 & \text{if } s = \emptyset \\ C_{opt}(s_{min}) + M(P(g_{min})) + \epsilon & \text{otherwise} \end{cases}$$  \hspace{1cm} (2.3)

where $s_{min}$ and $g_{min}$ are

$$\arg\min_{s' \cup g' = s, s' \cap g' = \emptyset} \{C_{opt}(s') + M(P(g')) + \epsilon\}$$  \hspace{1cm} (2.4)

$s'$ represents a subgraph that is already examined while $g'$ is a subgraph that is going to be evaluated with a measurer $M(\cdot)$, which takes a backend placement strategy and returns its actual run-time cost on the execution environment. We query the measurer at the granularity of a backend pattern that matches with $g'$, which is either single or multiple operators (operator fusion) that will be lowered to a single low-level kernel. This approach ensures that we always measure a single kernel and add it up to compute the cost of larger subgraphs. To avoid the repetitive and expensive measurement overhead (i.e., compilation + multiple runs on the actual hardware), we cache the result to the log for the future usage. With this approach, we can efficiently explore possible backend placements and evaluate them.

Figure 2.5 illustrates a simplified walkthrough example of our DP method. By traversing a computation graph $G$, it solves smaller problems of finding $P_{opt}(s)$ for a subgraph $s \subset G$ and eventually discovers $P_{opt}(G)$. First, it puts a root node in the priority queue as an initial frontier node; we define a frontier node as a node that has the lowest depth among unvisited nodes on a path from the root. Then it pops a frontier node with the lowest depth from the queue and examines if any subgraph rooted at the current frontier node can match any valid backend pattern. Once a matching is found, we add new frontier nodes to the priority queue and measure the cost of the subgraph matched with the backend pattern. If a better placement strategy is found, we update the optimized cost and backend placement strategy based on Equation (2.3). We repeat these steps until the priority queue is empty. Given that graph inference libraries, such as TensorRT, can also provide competitive operator-level implementations (Figure 2.2), we also include them in the op-level optimization. Algorithm 1 formalizes our DP method.

**Time complexity.** We derive the time complexity of Algorithm 1. Let $N$ be the number of nodes (operators) in computation graph $G$, $P$ be the average number of backend pattern matches per frontier, $F$ be the maximum possible number of frontiers for a single match, and $S$ be the maximum number of subgraphs in $S$ (line 19). In Algorithm 1, the outermost while loop (line 3) takes $O(N)$ times to traverse each frontier node in $G$. For each frontier, there can be $O(P)$ matches (line 6-8). For each match, the algorithm iterates over its $F$ (line 10) and $S$ (line 19) and takes $O(F + S)$. Therefore, the overall time complexity of our op-level placement optimizer is $O(NP(F + S))$. In all workloads that we have investigated, $N < 1000$, $P < 20$, $F < 10$, $S < 200$. As a result, our DP method optimizes placement within a minute by effectively pruning candidates.
**Algorithm 1** Op-level Placement Optimization: DP

**Input:** Computation graph $G$ and set of backend patterns $B$

**Output:** Optimized placement $P_{opt}(G)$

1: // $v_0$: a root of $G$, $Q$: a priority queue sorted by node depth
2: $Q = \{v_0\}$
3: while $Q = \emptyset$ do
4:   // $v_s$ is a frontier node
5:   $v_s = Q$.dequeue()
6:   for $b_i \in B$ do
7:     // Find a subgraph $g$ rooted at $v_s$ that matches $b_i$
8:     if $g = \text{get_match}(v_s, b_i)$ then
9:       // $F$ is a set of new frontier nodes after matching
10:      for $v_j \in F$ do
11:        if $v_j$ has never been added to $Q$ then
12:          $Q$.enqueue($v_j$)
13:       end if
14:     end if
15:   end for
16:   // $P(g) = \{(g, b_i)\}$
17:   // $M$ is a measurer
18:   // $S$ is a set of subgraphs, each of which includes all nodes
19:   // before $v_s$ in post-order and does not include $g$
20:   // $\epsilon$ is a constant for context switching cost
21:   for $s_j \in S$ do
22:     if $C_{opt}(s_j \cup g) > C_{opt}(s_j) + M(P(g)) + \epsilon$ then
23:       $C_{opt}(s_j \cup g) = C_{opt}(s_j) + M(P(g)) + \epsilon$
24:       $P_{opt}(s_j \cup g) = P_{opt}(s_j) \cup P(g)$
25:     end if
26:   end for
27: end if
28: end for
29: end while
30: return $P_{opt}(G)$

---

**2.4.3 Graph-level Placement Optimizer**

As the op-level placement optimization ignores the effect of cross-kernel optimizations (e.g., scheduling and memory optimizations) in graph inference libraries, Collage introduces the graph-level placement optimizer to fine-tune the potentially sub-optimal backend placement strategies from the op-level. To do so, we need to identify additional operators that are not assigned to graph inference libraries but can benefit from cross-kernel optimizations. Once identified, we offload them to graph inference libraries to extract further improvement. However, a key challenge we must address in this approach is deciding which operators to offload to graph inference libraries among a myriad of candidates.

To address this challenge, we represent each backend placement strategy by using a sequence of digits. Each digit implies whether to offload to graph inference libraries. Since our goal is to offload more operators that can benefit from the cross-kernel optimization,
we exclude operators already mapped with a graph inference library from this encoding. This straightforward state representation eliminates the complexity from various graph partitions and their topology.

We adopt an evolutionary search algorithm [FDRG+12] for graph-level placement optimization. Figure 2.6 describes the procedure of our evolutionary search method. For state representation, 0 indicates keeping the decision of the op-level optimizer and 1 means overriding the decision and offloading it to a graph inference library (e.g., TensorRT). To facilitate the search process, we include the op-level optimized placement strategy as one of the seeds to provide a good starting point. The evolutionary algorithm iterates over rounds of mutation, selection, and two-point crossover to fine-tune the backend placement.

2.5 Evaluation

This section aims to answer the following questions:

- Can Collage effectively optimize real-world DL model execution over diverse backends and target devices compared to the existing DL frameworks? (§2.5.2)
- Is optimization time affordable? How much time does each optimization take?
• Does adding more backends improve the performance of Collage? (§2.5.4)
• How does backend placement optimized by Collage look like? (§2.5.5)

2.5.1 Experimental Setup

Implementation. We built the core of Collage in the form of a portable Python library and leveraged diverse backends in different hardware architectures: cuDNN [CWV+14], cuBLAS [cuB], TVM [CMJ+18], TensorRT [Tena], MKL [WZS+14] and DNNL [One]. To orchestrate a runtime execution with multiple backends, Collage uses DLPack to minimize data movement (e.g., tensor) across different backend runtimes by efficiently exchanging pointers of data with zero-copy approach [DLP]. Still, even such optimized communications incur certain run-time overhead (e.g., deserialization overhead of the engine in graph inference libraries [Tenb]). Thus, Collage takes this run-time overhead into account when measuring execution time of various placement candidates. If such run-time overhead is too excessive, Collage will choose another candidate with better performance. To leverage full capabilities of backends, their supported patterns and pattern rules are provided based on their official documentation and codebases. Each backend specification with full operator supports only takes about 100 LoC with Collage API.

Baselines. We examine TensorFlow (TF) [ABC+16], TF-XLA [XLA], PyTorch [PGM+19a], TVM [CMJ+18], and TensorRT [Tena] as DL framework baselines. For TVM, we use AutoTVM to automatically generate the optimized operator schedules for each target. Note that we also integrate TensorRT and TVM as high-performance graph inference libraries in this experiment.

Workload. We evaluate five popular real-world DL inference workloads that cover a wide range of application. BERT [DCLT18] is a transformer-based language model that achieved the state-of-the-art performance on a spectrum of natural language processing tasks. DCGAN [RMC15] is an extension of the GAN [GPAM+20] with an unsupervised representation learning mainly for image generation. NasNet-A [ZVSL18] is one of the most popular machine-generated DL workloads that show strong performance on popular image recognition tasks. 3D-ResNet50 [HKS18] is an extension of widely adopted ResNet50 [HZRS16] for 3D image tasks such as action recognition. ResNeXt50 [XGD+17] introduces a grouped convolution to ResNet50 architecture and improves its model accuracy and computational complexity for image recognition.

Each workload has its own characteristics in terms of its operators and structure. Most of recent models for language application such as BERT are basically a series of the Transformer layers that consist of batch matrix multiplication, layer normalization, softmax, etc. On the other hand, models for vision application such as ResNeXt50 and NasNet-A has a series of layers that has operators including convolutions and non-linear activation functions (e.g., ReLU). In these models, operator configuration (e.g., number of channels and hidden nodes) varies across different layers as you see in Figure 2.2, which leads to performance diversity of DL backends.
2.5.2 End-to-end Evaluation

To discuss the effectiveness of our approach, we evaluate the end-to-end performance of Collage against the baseline frameworks; note that we omit error bars from our figures because we observe marginal standard deviation (less than 3%) for all results. Note that the performance of TF-XLA is missing for some pairs of workload and targets (e.g., 3D-ResNet50 and NVIDIA GPU) because it has issues with some 3D convolutions for GPU targets and certain image resizing operators.

Figure 2.7a and Figure 2.7b presents the end-to-end normalized throughput of Collage and existing DL frameworks on two different NVIDIA GPU architectures, Tesla V100 and GeForce RTX2070. Normalized throughput is the throughput of each framework normalized by the throughput of Collage. Overall, Collage consistently produces the most efficient executable across different workloads and hardware architectures: In terms of
Figure 2.8: End-to-end performance with different batch sizes in ResNeXt50 on NVIDIA V100. Normalized throughput is the throughput normalized by the throughput of Collage.

gemetric mean, Collage outperforms the state-of-the-arts by $1.43 \times$ on V100 and $1.26 \times$ on RTX 2070, respectively. This improvement comes from Collage’s backend placement optimization that effectively leverages the unique strength of various backends.

Figure 2.7c exhibits the experimental results on the Intel CPU. Likewise, Collage showcases the most stable performance across different workloads on this Xeon architecture while beating the state-of-the-arts by $1.40 \times$ in the geometric mean. However, on BERT and 3D-ResNet50, TF-XLA and TF are faster possibly due to their optimizations customized for Intel CPU such as data layout optimization with non-uniform memory access, which is orthogonal to backend placement.

As the representative case, different batch sizes are also examined with ResNeXt50 on V100. Figure 2.8 indicates that Collage consistently outperforms the state-of-the-art frameworks across different batch sizes as well.

Since backends and their performance vary depending on the underlying execution environment, backend placement should be carefully customized by considering their performance landscape. Our experimental results indicate that Collage can stably offer a faster DL execution than existing frameworks with the rigid hand-written heuristics across different hardware architectures.

2.5.3 Optimization Time

To evaluate the overhead from our automated optimizer, this subsection studies the overall optimization cost of the two-level approach. For this section, we use NVIDIA V100 as our target.

Figure 2.9a shows the breakdown of our operator-level optimization time. If the optimization is launched from scratch, the entire optimization process takes up to two minutes. This optimization time consists of two parts: measurement of the operator cost and
(a) The breakdown of op-level placement optimization time.

(b) Performance improvement of graph-level placement optimization over time.

Figure 2.9: (Left) On average, profiling overhead for operator cost measurements takes up 68% of the entire optimization time. Note that profiling is only necessary for unseen operators. Once the cost of a new operator is measured, its information will be saved in the logging database in Collage to avoid the repetitive profiling. If profiling log is available, op-level optimizer only takes less than a minute. (Right) The y-axis presents the speedup relative to the op-level placement optimization.

overhead from the DP algorithm. Due to the high evaluation cost, the optimization time is dominated by the profiling overhead. However, as discussed in §2.4.2, the repetitive profiling for operator cost can be avoided by saving the cost of each operator. When the cost of every operator is profiled in advance, our op-level placement optimization takes less than a minute on all of the five networks.

Figure 2.10 exhibits how our graph-level placement optimization gradually improves from the op-level placement optimization over time. The evolutionary searcher could boost the performance by leveraging more cross-kernel optimizations as it goes through several generations of mutations and crossovers. In BERT and DCGAN, the effect of cross-kernel optimization is quite notable and thus, our graph-level placement optimizer accelerates its execution by $1.09 - 1.20 \times$ from the op-level optimization. For the rest of the workloads, graph-level placement optimization cannot improve any further since the placement from the op-level optimization is already hard to beat. Overall, most of workloads are observed to reach the saturation within thirty minutes.

Due to the lack of the efficient cost model that can factor in the cross-kernel optimization effect, graph-level placement optimization has expensive evaluation overhead that leads to the longer optimization time compared to the op-level. Given that our op-level placement optimizer can identify high-performance backend placement for the most workloads within just a minute, we recommend the graph-level placement optimization as the optional tool for the users interested in squeezing the last drop of performance.
2.5.4 Backend Ablation Study

To assess the impact of integrating backends, we conduct an ablation study by adding backends one-by-one to Collage.

Figure 2.11 shows the experimental result on V100. Overall, Collage monotonically improves performance as we integrate more backends. This reinforces the importance of smart mixed-use of multiple backends and also corroborates the robustness of our backend placement optimization. It is worth noting that the performance improvement from a new backend varies depending on a network. In the case of BERT and DCGAN, we see relatively consistent enhancement from each backend. This is because Collage identifies a way to utilize every backend for the different part of the workload depending on its own unique strength. In case of NasNet-A and ResNeXt50, TVM offers the majority of the performance improvement while cuDNN significantly benefits Collage for the 3D-ResNet50.

These observations show that Collage can stably improve performance by having more
Figure 2.12: Representative backend placements discovered by Collage on V100 (Figure 2.7a). Note that Collage leverages various backends given their unique strength to enhance performance.

backends. By leveraging the unique strength of available backends, our automated optimizer delivers the performance with a set of backends that surpasses or guarantees the performance with its subset.

2.5.5 Case Study of Backend Operator Placement

To understand the source of performance improvement from Collage, we examine two representative workloads in detail. Figure 2.12 illustrates Collage’s final backend placement for ResNeXt50 and BERT on V100.

Even within a single network, we observe that the same type of operator is mapped to different backends due to the performance diversity depending on its configuration, such as data shape and kernel size, and the operator fusion with its neighbor nodes. For example, batch matrix multiplication operators in BERT are assigned to two different backends (cuBLAS and TVM) while convolution operators in ResNeXt50 are assigned to three different backends (cuDNN, TVM, and TensorRT). Interestingly, the graph inference library (e.g., TensorRT) can be a competitive choice even for a single operator as observed with some convolution operators in ResNeXt50.

This figure also demonstrates that Collage is capable of leveraging various fusion patterns from each backend. For instance, we discover a variety of operator fusion patterns selected by Collage such as Conv+ReLU, Conv+Add+ReLU, and Add+ReLU. Although it is omitted from this figure for simplicity, we observe the fusion pattern involved with
more than ten operators. Again, as in a single operator, Collage chooses the different backends for the identical fusion pattern of Conv+Relu in ResNeXt50 because the best backend choice varies depending on specific operator configurations.

This study confirms that Collage can accelerate DL workload execution by leveraging diverse operator patterns from multiple backends given their performance characteristics.

2.6 Related Work

**Diversified Backend Ecosystem.** To extract the best performance from the underlying hardware, there have been substantial efforts to design high-performance DL backends. Hardware vendors have released various specialized optimized libraries and inference engines. NVIDIA has actively developed cuDNN [CWV+14] to deliver optimized implementations of DL operators, cuBLAS [cuB] to offer efficient BLAS kernels, and TensorRT [Tena] to create fast execution plans for DL workloads. Particularly, TensorRT considers various graph-wide cross-kernel optimizations for scheduling, memory footprint and etc. Meanwhile, Intel has released oneDNN [One] for optimized DL operator kernels and OpenVINO [Opeb] as an inference engine for Intel CPUs. AMD also has driven MIOpen [KFT+19], a GPU library for DL primitives.

Today’s DL frameworks exploit tensor compilers [RKBA+13, AMA+19, CMJ+18, BRR+19, FCGM21, ZJS+20, LAB+21, KKC+17, PSS+21, MYS20, CGG+20, RSG19, BZR+22, JKC+22, TBT+16, VZT+18, JDL21, CCC+19, LCC+21, AKV+14, ZLW+20] as their backends to generate operator kernels for various target devices. While some tensor compilers rely on manual scheduling [RKBA+13, BRR+19, ZLW+20], automatic approaches [CMJ+18, CZY+18, JDL21, AMA+19, ZJS+20, PSS+21, RKBA+13, KKC+17, CGG+20, MYS20, JDL21, RSG19, BZR+22, AKV+14, VZT+18] has been actively studied to optimize tensor operator kernels for a given DL workload and device. For instance, Tensor Comprehension [VZT+18] uses black-box auto-tuning to optimize CUDA kernels along with polyhedral optimizations. To speed up the optimization time, cost model has been also widely examined together with automated approaches [CZY+18, KPB19, ZJS+20, ZLW+20]. By providing an expressive registration interface and automatic placement optimizer, Collage enables seamless integration of a wide variety of DL backends without any expertise in complex performance dynamics of varied backends.

**DL Frameworks.** To provide easy and powerful platform of running a variety of DL workloads, different frameworks have been continuously released and improved. Google maintains TensorFlow [ABC+16] and XLA [XLA] to optimize the execution on various hardware devices including TPUs [JYP+17]. Facebook develops Pytorch [PGM+19a] that supports dynamic eager execution for usability while preserving compelling DL execution performance. For NVIDIA GPUs, TensorRT [Tena] is developed as a runtime framework that optimizes DL model execution. As an open-source C++ library and compiler suite for CPUs, Intel has launched nGraph [CBB+18]. Also, TVM [CMJ+18] offers the efficient compilation pipeline that is designed to support diverse hardware devices and DL workloads. On the other hand, Glow [RFA+18] is proposed to efficiently
generate the optimized code for multiple targets of heterogeneous hardware. While such existing DL frameworks employ handwritten rules to integrate new backend, Collage reduces the manual effort with the backend pattern abstraction and extracts further performance gain with the automated backend placement.

**Operator Fusion.** Fusion is one of the most efficient techniques to optimize DL workloads by combining multiple high-level operators on the computation graph into a single kernel. To maximize the benefit, advanced fusion techniques [MXY+20, FCGM21, NGW+21, ZZL+20, XLA, ELB+17, BRH+18, ATB+15, CMJ+18, JDL21, LZPL22, DSC+16, AXW+19, SCSZ19] introduce their own unique fusion rules to apply this optimization beyond a few special cases. For instance, by iterating over every operator, TVM seeks for an opportunity to merge each operator with its neighbors by using the union-find algorithm [CMJ+18]. To efficiently explore the fusion opportunities, DNNFusion [NGW+21] employs a detailed classification of operation type and makes the fusion decisions. To identify the best fusion plan, FusionStitching [ZZL+20] conducts Just-In-Time tuning. NVIDIA has actively improved the fusion engine in cuDNN to merge certain patterns of operators at runtime [CWV+14]. Internally, TensorRT [Tena] also actively apply the fusion to optimize the memory access and scheduling overhead. By offering the highly flexible user interface for the pattern rules, Collage can support such complicated fusion patterns from a variety of such backends. With fusion patterns and their rules, Collage naturally considers diverse fusion possibilities in multiple backends.

**Graph Rewriting.** To accelerate a DL execution, DL frameworks can rewrite an input computation graph by considering a number of graph substitution rules. Most DL frameworks such as TensorFlow [ABC+16], TensorRT [Tena], and TVM [CMJ+18] rely on the greedy approach by opportunistically applying a few important hand-coded rules. In contrast, MetaFlow [JTW+19] suggests an automated graph rewriting approach that optimizes an input graph using backtracking search. TASO [JPT+19] extends MetaFlow’s backtracking search and further automates graph substitution generation for every new input graph. To further improve graph substitution search efficiency, sampling-based approach [FSWC20] has also been explored. To overcome the inefficiency in making sequential rewriting decisions, [YPW+21] proposes e-graph and equality saturation method. As these graph rewriting techniques are orthogonal to Collage, Collage can improve the performance of a rewritten computation graph by optimizing the backend placement.

**Device Placement.** There are two major categories of work that investigates how to place DL operators across devices. One category is to learn a placement policy [MPL+17, MGP+18, GCL18] that places each operator onto one of given set of devices and generalize it to new workloads via transfer learning [ZRA+19, AVG+18, PGN+20]. Another category is to algorithmically find good graph partitions of DL workloads and their schedules [JZA19a, JLQA18, NHP+19, TPD+20, ZLZ+22]; for example, FlexFlow [JZA19a] uses stochastic search method with delta simulation to partition a single operator into multiple computation and place them on devices. Compared to device placement, backend placement itself has its unique challenges of modeling complicated and fast-evolving
operator fusion patterns and constraints from diverse backends in addition to different backend characteristics (e.g., cross-kernel optimization of graph inference library). To tackle this challenge, Collage provides an expressive backend pattern abstraction and a two-level optimizer, each level of which considers different characteristics of backends. Our work is complementary to existing device placement works.

2.7 Conclusion

This work investigates an efficient DL backend integration system, called Collage. For the seamless integration of various backends, Collage offers an user interface that allows the flexible specification of diverse backend capabilities. To find the best uses of available backends, Collage introduces a two-level optimization method and automatically customizes the best possible backend placement for the underlying execution environment. The experimental results demonstrate that Collage outperforms the best manual approach in the state-of-the-arts DL framework by up to $1.43 \times$ on average over real-life DL models and various hardware architectures. More importantly, unlike existing approaches, it offers stable performance across diverse hardware architectures and models by selecting the most beneficial backends for each part of workload.
Chapter 3

GraphPipe: Improving the Performance and Scalability of DNN Training with Graph Pipeline Parallelism

Deep neural networks (DNNs) continue to grow rapidly in size, making them infeasible to train on a single device (e.g. GPU). Pipeline parallelism is commonly used in existing DNN systems to support large-scale DNN training by partitioning a DNN into multiple stages, which concurrently perform DNN computation for different micro-batches of training samples in a pipeline fashion. However, existing pipeline-parallel approaches only consider sequential pipeline stages and thus ignore the topology of a DNN, resulting in missed model-parallel opportunities.

We present graph pipeline parallelism (GPP), a new pipeline-parallel scheme that partitions a DNN into pipeline stages whose dependencies are identified by a directed acyclic graph. GPP generalizes existing sequential pipeline parallelism and preserves the inherent topology of a DNN to enable concurrent execution of computationally-independent operators, resulting in reduced memory requirement and improved GPU performance. In addition, we develop GRAPHPIPE, a distributed system that exploits GPP strategies to enable performant and scalable DNN training. GRAPHPIPE partitions a DNN into a graph of stages, optimizes micro-batch schedules for these stages, and parallelizes DNN training using the discovered GPP strategies. Evaluation on a variety of DNNs shows that GRAPHPIPE outperforms existing pipeline-parallel systems such as PipeDream and Piper by up to 1.6×. Despite the fact that GPP involves a much larger search space of parallelization strategies, GRAPHPIPE reduces the search time by 9-21× compared to PipeDream and Piper.
3.1 Introduction

Deep neural networks (DNNs) continue to grow more rapidly in size against hardware developments, making them computationally costly to train [opea, PGL+21]. A recent language model GPT-4 [Ope23] supposedly uses a much larger number of parameters [Hea] compared to the previous model GPT-3 with 175 billion parameters [BMR+20]. As a result, training modern DNNs requires distributing the model architecture across multiple devices.

To address this challenge, existing DNN systems apply model parallelism [DCM+12, SCP+18, SPP+19, LLX+20, NSC+21, ZLZ+22] where a DNN is partitioned into smaller pieces each of which fits into the memory of a single device.¹ Pipeline parallelism [HCB+19, FRM+21, NPS+21, NHP+19, TNP21] is a particular form of model parallelism that improves DNN training throughput and device utilization. As shown in Figure 3.1, a key idea of pipeline parallelism is to split both a DNN and a mini-batch of samples into smaller pieces. First, the DNN is partitioned into multiple disjoint stages, each of which is a sub-model and links to other stages to form a pipeline. Second, a mini-batch of samples is further divided into multiple micro-batches, which are executed on different stages in a pipeline fashion. This approach reduces device idle time in training iterations, during each of which a single data mini-batch is processed, and therefore improves throughput.

![Figure 3.1: Pipeline parallelism for DNN training with basic terms used in this paper.](image)

**Shortcomings of existing sequential pipeline parallelism.** Existing schemes of applying pipeline parallelism form a sequential pipeline from partitioned stages, which we refer to as sequential pipeline parallelism (SPP). SPP is simple to construct and operate but has three key limitations.

First, opportunities to exploit the inherent topology of a DNN are left unseized. Many DNN applications such as chatbot [Ope23], recommendation [NMS+19], and healthcare [KTB16, TTP+20, SUV20] interact with heterogeneous data types (e.g., text, images, different tables in tabular data). DNNs employed therein can be designed to feature multiple branches to jointly process the different types of data. These branches are often computationally-independent and thus can be processed concurrently. However, existing DNN systems with SPP first linearize the computation graph of a DNN in order

¹ More specifically, tensor parallelism refers to a form of model parallelism in which an operator (e.g., matrix multiplication) is split into smaller sub-operators and spread across devices. While this form of intra-operator model parallelism can also reduce per-device memory footprints, we focus only on inter-operator model parallelism in which more coarse-grained splits occur, i.e., a group of operators is assigned to each device.
to construct the stages of a sequential pipeline and process these stages sequentially. This manner of applying pipeline parallelism, despite its simplicity, falls short in harnessing the opportunity to leverage such branch-level parallelism in combination with pipeline parallelism.

Second, pipeline depth (i.e., number of sequential stages in SPP) is unduly increased and so is memory consumption as a consequence. In pipeline-parallel DNN training, a micro-batch traverses the pipeline’s stages to perform the computations dictated by the DNN (forward pass), and traverses in reverse for all stages to update their assigned model weights (backward pass), as shown in Figure 3.1. Each stage needs to store the intermediate activations of a forward pass until its corresponding backward pass is completed, creating the need to reserve GPU memory for activations in addition to weights. For a given stage, a micro-batch is in-flight until its backward pass finishes. As micro-batches are continuously injected into the pipeline, there is a warm-up of in-flight micro-batches for each stage. The earlier the stage in the pipeline, the longer the warm-up. The elongated pipeline formed by SPP increases pipeline depth, particularly for multi-branch DNNs. This in turn increases memory requirement especially for early stages in the pipeline. The tight memory constraint in training large DNNs is a primary reason to apply pipeline parallelism, thus it is critical to curb the increased memory consumption.

Third, today’s devices for DNN training (e.g., GPUs) have high parallel-computing capabilities, requiring a large micro-batch of training samples to achieve peak performance. However, due to the increased memory consumption, applying SPP impedes doing so. As a consequence, devices perform computations at lower operational intensity than the desired capacity, resulting in suboptimal training performance.

Our approach. To address the above challenges, we introduce graph pipeline parallelism (GPP), a new scheme of applying pipeline parallelism that enables performant and scalable DNN training. Figure 3.2 highlights the key difference between GPP and SPP. Instead of enforcing a strictly sequential execution order of pipeline stages, GPP allows partitioning a DNN into stages whose dependencies are identified by a directed acyclic graph. GPP includes SPP as a special case and can preserve the inherent topology of the DNN during stage partitioning. As a result, GPP enables concurrent execution of computationally-independent components, resulting in reduced memory requirement and improved GPU performance compared to SPP.

GPP involves a significantly larger and more complicated search space of parallelization strategies compared to the SPP strategies considered by existing DNN systems. Discovering GPP strategies with superior performance over existing SPP baselines requires weighing subtle trade-offs between pipeline depth, memory consumption, and micro-batch schedule. To unleash the power of GPP, we develop GRAPHPIPE, a system that automatically discovers efficient GPP strategies to enable performant and scalable DNN training. GRAPHPIPE includes three key components. First, a pipeline stage partitioner automatically determines how to partition the operators of a DNN into a graph of stages,
while balancing the computational load among these stages and minimizing inter-stage communication. Second, a static micro-batch scheduler schedules the forward and backward passes of different micro-batches within a mini-batch to minimize the peak GPU memory requirement of a GPP strategy. The stage partitioner and micro-batch scheduler jointly partition a DNN into stages and determine the micro-batch schedules for each stage. Finally, a distributed runtime uses the discovered GPP strategy to enable performant and scalable DNN training.

Through experiments on three multi-branch DNNs (e.g., Multi-Modal Transformer \([\text{VSP}^+17, \text{WCQ}^+23, \text{RKH}^+21, \text{Ope}23, \text{RPG}^+21, \text{JYX}^+21]\), DLRM \([\text{NMS}^+19]\), and CANDLE-Uno \([201]\)), we show that GRAPHPIPE can achieve up to \(1.61 \times\) training throughput improvements over existing pipeline-parallel systems such as PipeDream and Piper. Despite the fact that GPP involves a much larger search space of parallelization strategies, GRAPHPIPE reduces the search time by 9-21 \(\times\) compared to PipeDream and Piper.

To summarize, we make the following contributions:

- We introduce graph pipeline parallelism, a new pipeline-parallel scheme that enables concurrent execution of stages, reduces memory requirement, and improves GPU performance compared to existing SPP strategies.
- We design algorithms to partition a DNN into a graph of stages and schedule micro-batches for these stages, which jointly discover efficient GPP strategies.
- We develop GRAPHPIPE, a distributed runtime that enables fast and scalable DNN training leveraging GPP.
- We show that GRAPHPIPE outperforms existing pipeline-parallel systems by up to \(1.6 \times\) on a variety of DNNs.

### 3.2 Graph Pipeline Parallelism

Figure 3.2 compares sequential pipeline parallelism (SPP) employed by existing DNN systems \([\text{TNP}21, \text{NPS}^+21]\) and graph pipeline parallelism (GPP). Given a DNN and a set of devices, SPP and GPP produce strategies with different partitioning of stages and pipeline schedules. We next describe the key differences between SPP and GPP.

**Concurrent execution of stages.** SPP linearizes all operators of a DNN while preserving data dependencies between these operators, and then partitions the linearized DNN into a sequence of pipeline stages. As a result, each stage has at most one predecessor and one successor. The execution order of these stages is thus strictly sequential.

In contrast, GPP preserves the topology of a DNN when partitioning it into pipeline stages. To avoid circular dependencies between pipeline stages, the relationships between these stages form a directed acyclic graph. The execution order of the stages can be thus more general compared to SPP. This topology-aware partitioning and pipeline stage execution provides GPP a clear advantage: (potentially) concurrent execution of stages that are computationally-independent.

For the GPP strategy in Figure 3.2, three stages \(S_1, S_2,\) and \(S_3\) are computationally-
Figure 3.2: A high-level comparison between existing (SPP) and our (GPP) approaches. SPP (top) produces sequential pipeline stages that miss the opportunity of parallelizing the branches in the DNN. In contrast, GPP (bottom) generates graphical pipeline stages that enable parallel execution of the branches. This leads to lower training iteration time (i.e., higher training throughput) and smaller memory footprint in pipeline-parallel DNN training.

independent. Accordingly, the forward and backward passes of the three stages can be executed concurrently. However, in the SPP strategy, two stages $S_2$ and $S_3$ are partitioned such that they have a sequential data dependency (due to the dependency between operator $o_6$ in $S_2$ and operator $o_7$ in $S_3$) since the SPP partitioner does not consider the topology of the DNN and fails to exploit it. Moreover, while two stages $S_1$ and $S_2$ in the SPP strategy should be computationally-independent according to the original DNN, the SPP scheduler executes the forward and backward passes of these two stages sequentially. This is because new data dependencies are imposed between them when linearizing the operators of a DNN to construct a sequential pipeline.

This distinction directly leads to a performance gap. Specifically, both SPP and GPP involve a warm-up phase during which micro-batches are injected into the pipeline until all stages can perform work concurrently. However, as shown in Figure 3.2, the warm-up phase of GPP (i.e., 2) is shorter than that of SPP (i.e., 4). This performance improvement also applies to the cool-down phase during which in-flight micro-batches are resolved. As a result, GPP achieves a shorter per-iteration training time (hence, a higher throughput) than SPP. Note that the topology-aware stage partitioning and scheduling of GPP address the first shortcoming of SPP (§3.1).

**Reduced memory requirement.** There is a close relationship between the memory requirement of a pipeline-parallel strategy and its pipeline depth, which is defined as the diameter of its stage graph. A micro-batch moves along the pipeline in the topological order during the forward pass and in the reserve order during the backward pass. A micro-batch is in-flight for a stage $S$ if $S$ has finished the forward pass of the micro-batch
but hasn’t performed its backward computation. Due to the dependencies between the forward and backward computation of a micro-batch, each stage must store the intermediate activations of all in-flight micro-batches, which creates a non-trivial memory requirement. As pipeline depth increases, activation memory pressure increases and falls disproportionately upon early stages of the pipeline.

In Figure 3.2, GPP and SPP have a pipeline depth of 2 and 4, respectively. As a result, the first stage with the highest activation memory pressure needs to store the forward pass results for 2 micro-batches in GPP and those for 4 micro-batches in SPP. All else being equal (i.e., an identical model partition by both), GPP has a lower total memory footprint than SPP. Note that memory saving grows as model size grows since a bigger model requires deeper pipeline depth. The activation memory saving by GPP addresses the second shortcoming of SPP in §3.1.

**Improved GPU utilization.** Devices employed in DNN training (e.g., GPUs) are designed to parallelize DNN computation of a micro-batch efficiently. Thus, larger micro-batches (i.e., more training samples within a micro-batch) can improve the operational intensity, thus GPU utilization of DNN operators. Note that larger micro-batches lead to reduced numbers of micro-batches, which in turn increases the warm-up and cool-down time of pipeline that GPP can significantly reduce.² For simplicity of presentation, Figure 3.2 assumes that the same micro-batch size is used by GPP and SPP. However, a lower device memory requirement of GPP over SPP allows integrating more training samples in a micro-batch, which increases the operational intensity and overall GPU utilization, and therefore further reduces the per-iteration training time. We evaluate this aspect in more detail in §3.7.

**Fine-grained micro-batch size adjustments across stages.** We design GPP such that it offers the practitioner the option to use different micro-batch sizes across stages. Granted, this design complicates scheduling of forward and backward passes since different stages process varied micro-batch sizes and as a consequence the data dependencies within and across stages become convoluted. However, this added complexity is beneficial in several cases (we discuss these cases in §3.6 and provide an example in Figure 3.5). The execution time of an operator depends on a variety of factors such as the GPU type, kernel implementation, and micro-batch size. The practitioner can opportunistically use different micro-batch sizes across stages to balance their execution times in a fine-grained manner at the cost of the added complexity.

### 3.3 Problem Formulation

In this section, we formulate the problem of devising a GPP strategy for distributed DNN training. As input, we are given (a) a computation graph $G_C = (V_C, E_C)$ that represents the neural architecture of a DNN model, (b) a mini-batch size $B$, and (c) a device topology.

²Note that the accuracy of a DNN is not affected by varied micro-batch sizes since we still collect gradients from a fixed number of samples (identified by the user-provided mini-batch), and varying micro-batch size only affects the number of micro-batches within a mini-batch.
graph $\mathcal{D} = (\mathcal{V}_D, \mathcal{E}_D)$ where each node $v \in \mathcal{V}_D$ represents a device with memory budget $M_v$ and each edge $e \in \mathcal{E}_D$ represents a communication link with bandwidth $C_e$ between the two adjacent devices.\(^3\)

As output, we generate a pipeline stage graph $\mathcal{G}_S = (\mathcal{V}_S, \mathcal{E}_S)$ that optimizes for the performance metric of interest. In this work, we limit the scope to strategies that combine pipeline-parallel and data-parallel techniques, and aim to minimize the Time-Per-Sample (TPS) of the bottleneck pipeline stage since the pipeline throughput performance hinges upon the straggler stage.\(^4\) The stage graph $\mathcal{G}_S = (\mathcal{V}_S, \mathcal{E}_S)$ is a directed acyclic graph (DAG), where each node $S_i \in \mathcal{V}_S$ specifies a pipeline stage and each directed edge $(S_i, S_j) \in \mathcal{E}_S$ indicates that stage $S_i$ must precede $S_j$ for forward passes and that $S_j$ must precede $S_i$ for backward passes.

The goal is to solve the min-max optimization problem:

$$\begin{align*}
\min_{S_i \in \mathcal{V}_S} & \max_{S_i \in \mathcal{V}_S} \text{TPS}(S_i; \mathcal{G}_C, B, \mathcal{D}) \\
\text{s.t.} & \max_{S_i \in \mathcal{V}_S} \text{DeviceMemoryUsage}(S_i; \mathcal{G}_C, B, \mathcal{D}) \leq M.
\end{align*}$$

(3.1) \quad (3.2)

Formally, GPP devises a strategy $\mathcal{G}_S$ as follows. We define $S_i \in \mathcal{V}_S$ in further detail as a four-element tuple: $S_i = (\mathcal{G}_i, b_i, \mathcal{D}_i, \Pi_i)$:

1. $\mathcal{G}_i$ represents a subgraph of $\mathcal{G}_C$,
2. $b_i$ is the micro-batch size of $S_i$ (i.e., there are $B/b_i$ micro-batches for each mini-batch),
3. $\mathcal{D}_i$ is a set of devices allocated to process the forward and backward passes of $S_i$ (we apply data parallelism within $S_i$ if $|\mathcal{D}_i| > 1$), and
4. $\Pi_i$ is a micro-batch schedule that specifies the order in which the $B/b_i$ forward and $B/b_i$ backward passes are processed. We use $f_{w,j}^i$ (or $b_{w,j}^i$) to denote the forward (or backward) pass of the $j$-th micro-batch for $S_i$.

$\mathcal{G}_S$ is a valid GPP strategy if and only if the memory constraint (Equation 3.2) and all following conditions are met:

\begin{enumerate}
\item[C1.] $\mathcal{G}_i$ is a convex subgraph of $\mathcal{G}_C$\(^5\), and $\mathcal{G}_1, \ldots, \mathcal{G}_{|\mathcal{V}_S|}$ form a partition of $\mathcal{G}_C$.
\item[C2.] If there exists $(u, v) \in \mathcal{E}_C$ such that $u \in \mathcal{G}_i$ and $v \in \mathcal{G}_j$, then $(S_i, S_j) \in \mathcal{E}_S$.
\item[C3.] $\mathcal{D}_1, \ldots, \mathcal{D}_{|\mathcal{V}_S|}$ form a partition of $\mathcal{D}$.
\item[C4.] For each micro-batch schedule $\Pi_i$, $f_{w,k}^i$ precedes $f_{w,k+1}^i$, $b_{w,k}^i$ precedes $b_{w,k+1}^i$, and $f_{w,k}^i$ precedes $b_{w,k}^i$.
\item[C5.] For each $(S_i, S_j) \in \mathcal{E}_S$, the following must hold.
   \begin{align*}
   \text{If } \beta_{ij} := b_j/b_i \geq 1, \text{ then } & f_{w,j}^{\beta_{ij},k+1} \preceq f_{w,j}^{\beta_{ij},k+1} \in \Pi_i \preceq f_{w,j}^{\beta_{ij},k+1} \in \Pi_j \text{ and } b_{w,j}^{\beta_{ij},k+1} \preceq b_{w,j}^{\beta_{ij},k+1} \in \Pi_j
   \end{align*}
\end{enumerate}

\(^3\)In this work, we consider the homogeneous case of equal device memory $M_v = M$ for all $v \in \mathcal{V}_D$ and equal link bandwidth $C_e = C$ for all $e \in \mathcal{E}_D$.

\(^4\)While our optimization seeks to generate a strategy that minimizes the TPS metric, we can evaluate end-to-end performance with respect to different metrics, such as training iteration time, using the strategy generated (see §3.7).

\(^5\)For a graph $\mathcal{G}$, $\mathcal{G}'$ is a convex subgraph of $\mathcal{G}$ if for any pair of nodes $u$ and $v$ in $\mathcal{G}'$, any path from $u$ to $v$ is also in $\mathcal{G}'$. 
precedes $bw^i_{\beta_{ij},k+1}, \ldots, bw^i_{\beta_{ij},k+\beta_{ij}} \in \Pi_i$. Otherwise ($\beta_{ji} > 1$), $fw^j_{\beta_{ji},k+1}, \ldots, fw^j_{\beta_{ji},k+\beta_{ji}} \in \Pi_j$, and $bw^j_{\beta_{ji},k+1}, \ldots, bw^j_{\beta_{ji},k+\beta_{ji}} \in \Pi_j$ precede $bw^i_{k+1} \in \Pi_i$.

In words, $C_1$ mandates that all operators be covered by stages that do not overlap with each other, and $C_2$ mandates that a strict sequential execution order between two stages be established if according to the computation graph there exists a data dependency between two operators each in either of the stages. $C_3$ ensures that at least one device is allocated to every stage. $C_4$ dictates the orderings of forward and backward passes within a stage and $C_5$ dictates such orderings across different stages for correctness of pipeline parallelism.

3.4 System Overview

Figure 3.3 illustrates an overview of our distributed system GRAPHPIPE that accelerates DNN training at scale using GPP. Taking as input (a) the computation graph of a DNN, (b) mini-batch size, and (c) the topology of assigned GPUs, GRAPHPIPE produces an optimized GPP strategy for parallel DNN training. GRAPHPIPE includes three key components: a pipeline stage partitioner, a static micro-batch scheduler, and a distributed...
runtime. The first two components jointly discover a high-performance GPP strategy for a given DNN model, mini-batch size, and assigned devices, which will be executed by the distributed runtime.

**Pipeline stage partitioner.** The partitioner performs three tasks. First, it partitions a DNN, aimed at achieving an effective distribution of workloads across stages. It examines the amount of computation and communication needs associated with the operators in each stage. Importantly, it leverages the inherent topology of the DNN at hand in order to exploit concurrent execution opportunities. To this end, it performs a sequence of series-parallel decompositions of the given DNN. Second, it adjusts the micro-batch size for each stage. This fine-grained adjustment aims to exploit heterogeneous compute efficiencies of different types of operators. Finally, it determines how many devices to assign to each stage to achieve an effective allocation of resources. Note that all three functions are jointly performed, as no one function is independent of the others. We provide further details in §3.5.

**Static micro-batch scheduler.** The scheduler performs two tasks. First, it optimizes micro-batch schedules for forward and backward passes while ensuring the integrity of distributed DNN training. This involves examining both intra- and inter-stage data dependencies between the passes (see C4–C5 in §3.3). Next, it checks if the memory usage that results from the schedule is within the given device memory constraint (see Equation 3.2). Memory usage is closely related to the numbers of in-flight micro-batches of a stage, which can be computed based on the schedule of the forward and backward passes of the stage. §3.6 provides further details.

**Distributed runtime framework.** We develop a distributed DNN runtime system that executes GPP training strategies generated by the optimizer of GRAPHPIPE. Using the distributed runtime as the testbed, we compare the performance of the generated GPP strategies against existing SPP strategies for various DNNs. We provide further details in §3.7.

### 3.5 Pipeline Stage Partitioner

The pipeline stage partitioner of GRAPHPIPE aims to minimize Time-Per-Sample (TPS) of the bottleneck pipeline stage as in §3.3. It takes as input a DNN computation graph $G_C$, a mini-batch size $B$, and a device topology graph $G_D$, and generates an optimized stage graph $G_S$ by searching over different model partitions, device assignments, and micro-batch sizes simultaneously. A key challenge we must address is the large and complex search space of potential GPP strategies. To reduce the complexity of the search task, we employ a binary search method combined with series-parallel decomposition and dynamic programming. We next describes these three components.

**Binary search.** Given the large search space of potential solutions, GRAPHPIPE does not attempt to directly find an optimal solution. Instead, GRAPHPIPE employs binary search to iteratively narrow down the target performance range and examines whether there exist valid solutions within the range. By iteratively reducing the range, GRAPHPIPE
Algorithm 2 Pipeline stage partitioner.

Input: Computation graph $G_C$, number of devices $|V_D|$
Output: Optimized stage graph $G_S$

1: // MAXTPS: safe upper-bound for TPS of bottleneck stage.
2: $t_l = 0$, $t_r = \text{MAXTPS}$, $G_S = \emptyset$
3: while $t_r - t_l > \epsilon$ do
4: $t_m = (t_l + t_r)/2$
5: $G_S^{\text{best}} = \text{SEARCHSTAGEGRAPH}(G_C, |V_D|, t_m, B)$
6: if $G_S^{\text{best}} == \emptyset$ then
7: $t_l = t_m$
8: else
9: $t_r = t_m$
10: $G_S = G_S^{\text{best}}$
11: return $G_S$
12:
13: function $\text{SEARCHSTAGEGRAPH}(G_C, |V_D|, t_m, B)$
14: // $C$ is a set of candidate schedule configurations ($c$)
15: for $c \in C$ do
16: $G_S^{\text{new}} = \text{DP}(G_C, c_f, c_b, |V_D|, t_m)$ // $c_0$: dummy schedule configuration
17: // PICKBESTER() picks one with less memory
18: $G_S^{\text{best}} = \text{PICKBESTER}(G_S^{\text{best}}, G_S^{\text{new}})$
19: return $G_S^{\text{best}}$
20:
21: function $\text{DP}(G, c_f, c_b, d, t_{max})$
22: if this DP state has been visited then
23: return corresponding $G_S^{\text{best}}$ to this DP state
24: $G_S^{\text{best}} = \emptyset$ // Consider a given DP state as a SINGLE stage
25: if ESTIMATETPS($G, c_f, c_b, d) \leq t_{max}$ then
26: // Optimize schedule via Algorithm 3
27: $\Pi_{opt} = \text{SCHEDULESTAGE}(G, c_f, c_b, d)$
28: $G_S^{\text{best}} = \text{STAGEGRAPH}(G, \Pi_{opt}, d)$
29: // Decompose a given DP state into two stages
30: if $G$ can be decomposed in series then
31: for $(G_1, G_2) \in \text{SERIESDECOMPOSE}(G)$ do
32: for $d_2 \leftarrow 1$ to $d - 1$ do
33: $d_1 = d - d_2$
34: for $c_m \in C$ do
35: $G_S^{\text{new}} = \text{DP}(G_2, c_m, c_b, d_2, t_{max})$
36: Update $i_m$ based on $G_S^{\text{new}}$
37: $G_S^{\text{new}} = \text{DP}(G_1, c_f, c_m, d_1, t_{max})$
38: else if $G$ can be decomposed in parallel then
39: for $(G_1, G_2) \in \text{PARALLELEDUCESTATE}(G)$ do
40: for $d_1 \leftarrow 1$ to $d - 1$ do
41: $d_2 = d - d_1$
42: $G_S^{\text{new}} = \text{DP}(G_1, c_f, c_b, d_1, t_{max})$
43: $G_S^{\text{new}} = \text{DP}(G_2, c_f, c_b, d_2, t_{max})$
44: $G_S^{\text{best}} = \text{PICKBESTER}(G_S^{\text{best}}, G_S^{\text{new}})$
45: return $G_S^{\text{best}}$
discovers solutions arbitrarily close to an optimal one, and thus there is little difference in performance for practical purposes. Lines 2–11 of Algorithm 2 shows GRAPHPIPE’s binary search process.

**Series-parallel decomposition.** Since most DNNs structurally reflect series-parallel graphs [TNS82], GRAPHPIPE applies series-parallel decomposition to an input graph \( G_C \) in order to break it down into smaller, manageable subgraphs, and perform model partitioning, device allocation, and task scheduling for each subgraph. In the unusual cases where a DNN does not possess such a structural property, GRAPHPIPE bypasses this issue by converting the DNN to an arithmetically identical one whose structure is a series-parallel graph.

**Dynamic programming (DP).** GRAPHPIPE adopts a dynamic programming algorithm where the value of each DP state indicates the existence of a strategy achieving a throughput within a target range (Lines 13–19 of Algorithm 2). At each DP level, GRAPHPIPE applies series-parallel decompositions to split an input graph (say \( G \)) into two new subgraphs (say \( G_1, G_2 \)), each of which serves as the input computation graph of a new DP subproblem at one DP level below. GRAPHPIPE recursively solves the DP subproblems to construct a solution of the original problem where the input computation graph is \( G_C \) (Lines 21–45 of Algorithm 2).

**DP subproblem.** We ensure that each DP subproblem maintains a certain structure (i.e., having a unique pair of source and sink nodes and a subgraph \( G \) comprised of them). The input to a DP subproblem includes a computation graph \( G \subseteq G_C \), the number of devices \( d \), and some schedule-related information for its predecessor and successor stages, which we furnish by enumeration if not available.

The solution of a DP subproblem involves devising a training strategy such that (1) the number of in-flight micro-batches for the source stage (i.e., the pipeline stage that includes the source node) is minimized; and (2) the Time-Per-Sample (TPSes) for all stages do not exceed the target TPS range. These results are returned back to the parent DP subproblem at one DP level above where the results are gathered for the parent DP subproblem to produce its own.

We consider three cases in a DP subproblem:

- **Base case:** We consider the entire subgraph \( G \) as a single stage and apply data parallelism with data-parallel degree \( d \) (Line 25 in Algorithm 2). We check if the target TPS range is achievable while meeting the memory constraint, and compute the number of in-flight micro-batches according to Algorithm 3 (see §3.6).
- **Series decomposition:** We perform a series decomposition to create two subgraphs \( G_1 \) and \( G_2 \), where the sink node of \( G_1 \) coincides with the source node of \( G_2 \) (Line 30 in Algorithm 2).

\(^6\) We define a sink node of a subgraph as the last node among all nodes of the subgraph in their execution order. We define a source node of a subgraph as the sink node of the subgraph associated with the previous stage.
we enumerate all feasible schedules for the source node of $G_2$. We then solve the subproblem associated with $G_1$.

- **Parallel decomposition:** We perform a parallel decomposition to create $G_1$ and $G_2$, where $G_1$ and $G_2$ share the same source and sink nodes (Line 38 in Algorithm 2). As there is no data dependency between these subgraphs, the pipelines can be executed in parallel. The subproblems associated with $G_1$ and $G_2$ may produce different optimal numbers of in-flight micro-batches for the shared source node. When there exists a subgraph $G'$ preceding both $G_1$ and $G_2$, the data dependency between $G'$ and $G_1$, and that between $G'$ and $G_2$ should be respected. To ensure those dependencies, we take the larger number of in-flight micro-batches as the solution.

**Overall process.** Figure 3.4 visualizes the overall process. At the top, a DP subproblem is provided with its initial conditions: computation graph $G$, the number of available devices $d$, and the target TPS range $[0, t_{max}]$. Suppose the number of in-flight micro-batches for the sink node is $i_b$, the micro-batch sizes for the source and sink nodes are $b_f$ and $b_b$, the stage containing the source node (i.e., source stage) uses the $k_f F_k B$ schedule, and the stage containing the sink node (i.e., sink stage) uses the $k_b F_k B$ schedule (we introduce GRAPHPIPE’s micro-batch schedules in §3.6). These supposed conditions comprise a schedule configuration denoted by $c := (i, b, k)$ in Algorithm 3. They are either available as the results of some other DP subproblems solved previously, or furnished by enumeration. The solution of this DP subproblem computes the smallest possible number of in-flight micro-batches for the source stage (i.e., $i_f$ in Figure 3.4) that meets the target TPS range $[0, t_{max}]$.

**Time complexity.** We analyze the time complexity of the stage partitioner to gauge the impacts of design parameters. Let $N$ be the number of series-parallel subgraphs of $G_C$, $B$ be the set of possible micro-batch sizes, $D$ be the set of possible data-parallel degrees. The maximal element of $B$ is upper-bounded by $B$. We consider powers of 2 for micro-batch sizes (i.e., $|B| < \log_2 B$). Likewise, the maximal element of $D$ is upper-bounded by $|V_D|$ and $|D| < \log_2 |V_D|$ holds.

The number of candidates for $G$ is $O(N)$, that for $c_f = (b_f, k_f)$ is $O(|B|^2)$, that for $c_b = (i_b, b_b, k_b)$ is $O(B|B|^2)$, and that for $d$ is $O(|D|)$ in each DP subproblem. To compute a DP value, it takes $O(|D||B|^2)$ time for series decompositions and $O(|D|)$ time for parallel decompositions. Therefore, the time complexity for a single DP run is $O(NB|B|^6|D|^2)$ and the overall time complexity is

$$O((\log \text{MAXTPS})NB|B|^6|D|^2) = O((\log \text{MAXTPS})NB(\log_2 B)^6(\log_2 |V_D|)^2)$$

We see that the inherent topological complexity of a given DNN $G_C$ (denoted as $N$ in the above analysis), mini-batch size $B$, and device budget $|V_D|$ largely determine the theoretical time complexity. For the practical time complexity, however, reducing the number of candidates for micro-batch sizes (i.e., $|B|$) and data-parallel degrees (i.e., $|D|$) has considerable impacts. In addition, by considering only a subset of candidates, as opposed to sweeping over all of them, we can reduce the practical time complexity at the
cost of potentially degraded performance. Moreover, instead of applying series-parallel decompositions at a finest granularity of individual operators, we can apply them at a relatively coarse granularity of small groups of operators to reduce the overall time complexity (i.e., reducing $N$). These are choices the practitioner can make in practice to strike a balance between performance and time complexity. §3.7 shows that our stage partitioner is $9\times-40\times$ faster than the partitioning algorithms of existing pipeline-parallel systems, while considering a significantly larger search space.

### 3.6 Static Micro-Batch Scheduler

The static micro-batch scheduler optimizes micro-batch schedules and ensures their validity by meeting the data dependency requirements between all forward and backward passes within a stage ($C4$ in §3.2) and those across stages ($C5$ in §3.2). The scheduler takes as input (1) a configuration of model partition $G$, (2) current and next stage schedule configurations $c_f$, $c_b$, and (3) the number of devices $d$ from the pipeline stage partitioner, and produces an optimized micro-batch schedule configuration $c_{opt}$ for a given stage configuration. As shown in Figure 3.3, the input is fed by the stage partitioner, and the output is returned back to the stage partitioner to form a stage graph with an optimized

---

Figure 3.4: Pipeline stage partitioner performing series-parallel decompositions. Black arrows indicate subproblem formulations. Red arrows indicate solutions of the subproblems.
micro-batch schedule.
Once the micro-batch schedules are determined, the memory requirement of each device, which consists of model weights, optimizer states, and intermediate activations, can be calculated. As a result, the micro-batch scheduler can examine whether the device memory constraint (Equation 3.2) is satisfied. As presented in §3.5, the micro-batch scheduler is a key subroutine of GRAPHPIPE’s pipeline stage partitioner. Algorithm 3 describes its algorithm.

Algorithm 3 Static micro-batch scheduler.

Input: Model partition $G$, initial current and next stage schedule configurations $c_f, c_b$, number of devices $d$

Output: Optimized schedule $\Pi_{opt}$

1: function SCHEDULESTAGE($G, c_f, c_b, d$)
2:     // Optimize schedule to minimize number of
3:     // in-flight micro-batches using Table 3.2
4:     // while respecting data dependencies
5:     $i_f = \text{COMPUTEINFLIGHT}(k_f, b_f, k_b, b_b, i_b)$
6:     $c_{opt} = (i_f, k_f, b_f)$
7:     if $c_{opt}$ violates device memory constraint then
8:         $c_{opt} = \emptyset$ // Invalidate schedule $c_{opt}$
9:     $\Pi_{opt} \leftarrow \text{SCHEDULETASK}(c_{opt})$
10:    return $\Pi_{opt}$

Algorithm 2 first calls Algorithm 3 to discover an optimized micro-batch schedule for the last stage. It then traces back all directed edges $(S_i, S_j) \in E_S$ of the stage graph $G_S$ in the reverse direction and determines a schedule for each stage $S_i$ until a schedule for the first stage is determined. The reason for backward traversal is that computing the activation memory usage, and thus the total usage, for a stage $S_i$ requires complete schedule information of its subsequent stages $S_j$.

COMPUTEINFLIGHT(·) is a key function in Algorithm 3 to optimize a schedule. It aims to effectively minimize the number of in-flight micro-batches for a given stage without increasing per-iteration training time. While there are numerous possible micro-batch schedules, we consider two widely-adopted classes: 1F1B and $kFkB$. Note that they both ensure the data dependency requirements C4–C5 in §3.2.

1F1B schedules a single forward pass for a given micro-batch followed by a single backward pass. In contrast, $kFkB$ schedules $k$ forward passes followed by $k$ backward passes.$^7$ Recent work [WCS+23] explores performance implications in cases where all stages use the same micro-batch size and employ either 1F1B or $kFkB$. In practice, computations for forward and backward passes take different amounts of time, and

$^7$A schedule $\Pi_i$ is said to be $kFkB$ when there exist $\ell$ and $k$ such that $\Pi_i$ starts with $\ell$ forward passes (for warm-up), alternates between $k$ backward and $k$ forward passes, and ends with $\ell$ backward passes (for cool-down).
so do communications of their results. Thus, employing \( kFkB \) over \( 1F1B \) can enable finer-grained adjustments of these varied execution times to overlap them, hiding the smaller cost. It comes at the expense of increased memory usage, but can be justified when the execution time saving benefit outweighs it.

In contrast, our setup generalizes conventional pipeline parallelism so that it allows for different micro-batch sizes over stages. This extension opens up a new opportunity of using different micro-batch schedules across distinct stages as opposed to applying the same schedule (e.g., \( 1F1B \)) for all stages. We find it beneficial in terms of memory footprint to use different schedules for different stages in our setup unlike conventional pipeline parallelism. As shown in Figure 3.5, \( S_2 \) has a different micro-batch size (=1) from that of \( S_1, S_3 \) (=2). \( S_2 \) can employ either \( 1F1B \) or \( 2F2B \) without degrading training iteration time. If \( S_2 \) employs \( 2F2B \), it can save the activation memory footprint of \( S_1 \) by reducing the number of in-flight micro-batches from 4 to 3 in comparison to \( 1F1B \).

We also develop an algorithm where we schedule forward and backward passes in an arbitrary sequence. Intuitively, for a given stage, it is desirable to schedule a backward pass as early as possible since it quickly resolves the corresponding in-flight forward pass, reducing both memory consumption and training iteration time. This method of greedily scheduling backward passes for a stage has a beneficial cascading effect for its predecessor stages. According to C5 in §3.3, it is required that a stage can start processing a backward pass only after all of its successors finish processing it. This requirement means that when a stage aggressively schedules backward passes, it benefits all of its predecessors in addition to itself. Since we allow for different micro-batch sizes across stages, the data dependencies of forward and backward passes to meet become convoluted. Yet the intuition is still valid. Our algorithm based on the intuition turns out to be optimal in terms of iteration time and memory footprint. We provide a detailed description and the proof in Section 3.10.1.

In practice, there is a trade-off between performance and time complexity in choosing which scheduling algorithm to employ. \( kFkB \) has an advantage over \( 1F1B \) in terms of memory consumption as shown in Figure 3.5, and our method of greedily scheduling backward passes guarantees optimal performance. Improved performance comes with the cost of increased solution search time. This trade-off is non-trivial to quantify in practice, thus we enable all options in GRAPHPIPE. However, we employ \( 1F1B \) by default in our evaluations since except for some corner cases, we observe that performance improvements are incremental to justify the significantly increased search times.

On a final note, the idea of using different micro-batch sizes across different stages is to some extent similar to the idea of intentional uneven model partitioning in recent work [FRM+21]. The shared goal is to improve throughput by distributing workloads across stages unevenly and then exploiting heterogeneous compute efficiencies to carry out fine-grained adjustments of stage execution times. The difference is the dimension along which the workloads are split unevenly. It is along the batch (data samples) dimension in our work, and the model (operations) dimension in [FRM+21].
Figure 3.5: A comparison between 1F1B and $kFkB$ schedules when stages have different micro-batch sizes. $F\{i, j\}$, $B\{i, j\}$ indicate forward and backward passes for a micro-batch including samples $i$ and $j$. It showcases how $kFkB$ can be better than 1F1B in terms of memory footprint.

3.7 Evaluation

We develop GRAPHPIPE on top of FlexFlow [JZA19b], a distributed multi-GPU runtime for DNN training. Major modifications are to replace FlexFlow’s partitioner and scheduler with ours described in §3.5 and §3.6 respectively. We evaluate GRAPHPIPE on the Summit supercomputer [sum]. For each compute node of Summit, we use 2 IBM POWER9 CPUs and 4 NVIDIA V100 GPUs with 512GB of main memory. GPUs within a node are interconnected via NVLink while nodes are connected via Mellanox EDR 100Gb InfiniBand. Note that we omit error bars for our plots in this section, as we observe marginal standard deviations (less than 3%) for all results.

DNNs. We explore three multi-branch DNNs: Multi-Modal Transformer-based model (MMT) [VSP+17, RKH+21], DLRM [NMS+19], and CANDLE-Uno [201]. Multi-Modal Transformer (MMT) is a backbone of most state-of-the-art multi-modal models [WCQ+23, RKH+21, Ope23, RPG+21, JYX+21]. DLRM is a popular deep learning recommendation model for personalization and ads recommendation. CANDLE-Uno is a specialized model in the medical domain (i.e., precision medicine). We describe the detailed model configurations in Section 3.10.3. Despite different applications, all these models share a structural similarity: they all feature parallel branches, each processing a different type of data.
3.7.1 End-to-End Evaluation

We compare the training throughput of GRAPHPIPE with existing pipeline-parallel systems: PipeDream [NPS+21] and Piper [TNP21]. We choose these two baselines since their combined search space encompasses all possible model partitions covered by other approaches [FRM+21, ZLZ+22, NHP+19]. We implement their stage partitioning algorithms in the FlexFlow runtime to conduct fair comparisons. We use the synchronous 1F1B schedule [NPS+21] since it avoids gradient staleness with the same pipeline latency and lower activation memory footprint in comparison with other alternatives (e.g., GPipe [HCB+19]).

Figure 3.6 shows the results. We measure the training throughput (number of samples processed per second) as we increase the number of GPUs and mini-batch sizes. Note that Piper does not generate training strategies for DLRM and CANDLE-Uno since its time and space complexity increases exponentially with respect to the number of parallel branches. GRAPHPIPE outperforms PipeDream and Piper at all but one GPU configuration. Moreover, the performance gap widens as the number of GPUs increases.8

Our analysis reveals that we can attribute the widening performance gap to the pipeline depths greatly reduced by GRAPHPIPE compared to PipeDream and Piper for the multi-branch models. As we use more devices, the number of sequential pipeline stages tends to increase to achieve a higher throughput, particularly when the model size is too large to apply data parallelism at the cost of weight memory footprint and all-reduce weight synchronization. With a larger number of stages, sequential pipeline schemes by either PipeDream or Piper suffer from extended warm-up and cool-down phases. Directly, these extended pipeline bubbles negatively affect training throughput. Indirectly, these bubbles increase activation memory footprints, which in turn impede effective model

8We were not able to conduct experiments beyond 32 GPUs due to the limitation of the runtime backend we rely on. It inherently suffers from the bloated overhead of managing and analyzing large numbers of data dependencies for forward and backward tasks as we increase the GPU count.
partitioning. We visualize this analysis in detail via a case study (see §3.7.4).

### 3.7.2 Search Time

Table 3.1: Solution search times (in seconds) for Piper, PipeDream, and Ours (GRAPHPIPE) on the Apple M1 Max; ✓ indicates search cannot be completed. Numbers in parentheses indicate the search time ratio of the algorithm to that of GRAPHPIPE.

<table>
<thead>
<tr>
<th># GPUs</th>
<th>MMT</th>
<th>DLRM</th>
<th>CANDLE-Uno</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Piper</td>
<td>PipeDream</td>
<td>Ours</td>
</tr>
<tr>
<td>4</td>
<td>52.9</td>
<td>2.57</td>
<td>0.12</td>
</tr>
<tr>
<td>8</td>
<td>126</td>
<td>11.9</td>
<td>0.76</td>
</tr>
<tr>
<td>16</td>
<td>304</td>
<td>44.3</td>
<td>3.00</td>
</tr>
<tr>
<td>32</td>
<td>745</td>
<td>151</td>
<td>10.11</td>
</tr>
</tbody>
</table>

Table 3.1 presents the search times by the three optimizers (GRAPHPIPE, PipeDream, and Piper) for the three models (Multi-Modal Transformer, DLRM, and CANDLE-Uno). The Multi-Modal Transformer-based model has two branches and the DLRM and CANDLE-Uno models have eight branches.

GRAPHPIPE is at least 9× faster than the baselines irrespective of the models or GPU configurations. In addition, GRAPHPIPE’s efficient partitioner produces a strategy within a minute for all configurations. The SPP baselines are much slower by comparison, and this search time discrepancy can be attributed in large part to the fact that the baselines rarely leverage DNN topology in expediting search. Note that Piper does not produce strategies for the DLRM and CANDLE-Uno models for the aforementioned reasons.

To see the large search space of each SPP baseline, it is helpful to approximate their time complexities. Let us consider a simple multi-branch model with each branch having \( k > n \) operators, where \( n \) is the number of branches. Recall that Piper considers model partitions in which cross-branch stages exist. This level of granularity of model partitions significantly increases the number of model partitions to examine. Piper’s optimizer runs in \( O(|D|^2) \) time (Appendix D in [TNP21]), where \( D \) is the set of downsets (Definition 4.1 in [TNP21]). According to the definition, model partitions in which one stage spans multiple branches and all other stages are formed within a branch are valid candidates. Since we can choose one operator out of \( k \) from each branch to form a cross-branch stage, the number of such model partitions is at least \( |D| \geq \prod_{i=1}^{n} k = k^n \). Thus, Piper’s time complexity is lower-bounded by \( O(k^{2n}) \). This time complexity implies that unless we employ a set of clever heuristics, Piper’s time complexity can be significantly high for multi-branch DNNs.

On the other hand, PipeDream considers a converted DNN that linearizes all branches and the operators within. Thus, it deals with a single chain of operators, where the number of model partitions to consider is much smaller than Piper.

Still, GRAPHPIPE considers significantly fewer model partitions than PipeDream (and hence Piper) particularly when a given DNN features multiple branches. Instead of
solving a single long chain of \( nk \) operators as in PipeDream, \textsc{GraphPipe} solves \( n \) short chains of \( k \) operators separately. As empirically shown in Figure 3.6, \textsc{GraphPipe} barely demonstrates throughput degradation, which could have resulted from examining much fewer model partitions. Explicitly leveraging DNN topology in examining model partitions in search for a training strategy turns out to be critical to reducing the search space and time complexity.

### 3.7.3 Different Numbers of Branches and Micro-Batch Sizes

Figure 3.7 shows the results of two experiments in which we change the number of parallel branches for the CANDLE-Uno model (left) and change the number of micro-batch sizes for the two-branch multi-modal Transformer-based model (right). The purpose of the experiments is to investigate the effects of main parameters on the performances of \textsc{GraphPipe} and the SPP baselines (PipeDream and Piper).

The left sub-figure depicts the throughput performances normalized by that of PipeDream with respect to the number of parallel branches for the CANDLE-Uno model.\(^9\) We see that the performance gap achieved by \textsc{GraphPipe} scales with the number of branches, reaching up to \( 2 \times \) at 16 branches. Intuitively, the performance gain mostly stems from the fact that \textsc{GraphPipe} is able to reduce the pipeline depths at all configurations allowing concurrent execution of parallel branches, reducing the inefficient pipeline warm-up and cool-down phases significantly. The gain scales because the larger the number of branches, the larger the differentials of the phases between \textsc{GraphPipe} and SPP. This experiment result demonstrates that (1) reducing pipeline depth is critical to throughput performance; and (2) \textsc{GraphPipe} is better at it than SPP especially when multiple branches of non-negligible workload are present. The larger the number of branches in a given DNN to train, the more promising opportunities for \textsc{GraphPipe} to exploit by reducing pipeline depths.

The right sub-figure depicts the throughput performances for the multi-modal Transformer-\( ^9\)Piper was not able to produce a strategy for the CANDLE-UNO model.

---

\(^{9}\)Piper was not able to produce a strategy for the CANDLE-UNO model.
Figure 3.8: A synthetic Transformer-based two-branch DNN for case study. A sequence of one multi-head attention and two linear layers is repeated four times to compose a single branch. One concatenation layer at the end combines the branches.

Based model with four branches. We use a mini-batch size of 128 and eight GPUs. We intentionally fix a micro-batch size (instead of using the best ones chosen by the optimizers) in comparing the performances, for the purpose of examining the benefits (or harms) of using large micro-batch sizes. If increasing micro-batch size turns out to be beneficial, then it is worth reducing pipeline depth so as to reduce activation memory footprints, and in turn create room for using a larger micro-batch size.

We can observe the key role of reduced pipeline depth by GRAPHPIPE in improving throughput. For each micro-batch size, GRAPHPIPE always outperforms SPP. Since there is no difference in operational intensity with the same micro-batch size used for both GRAPHPIPE and SPP, the performance gap can be solely attributed to the difference in pipeline depth. The reduced pipeline depth by GRAPHPIPE leads to a shorter execution time for the warm-up and cool-down phases, hence a higher throughput.

Also, the better throughput scalability of GRAPHPIPE with respect to increasing micro-batch size is also closely-related to reduced pipeline depth. With a fixed mini-batch size, the number of micro-batches decreases inversely proportional to micro-batch size. This means that the ratio of the duration of the full pipeline to the duration of the warm-up and cool-down phases diminishes as micro-batch size increases. This in turn means that as micro-batch size increases, the benefit of utilizing devices at improved operational intensity by using a larger micro-batch size is gradually offset by a shorter full-pipeline duration relative to the warm-up and cool-down phases. The reduced pipeline depth by GRAPHPIPE tames such an offsetting effect, hence a better throughput scalability.

3.7.4 Case Study

It is instructive to take a close look at the strategies produced by GRAPHPIPE and SPP. We run both GRAPHPIPE and SPP optimizers for a synthetic model, execute the strategies, and observe a 20% throughput improvement by GRAPHPIPE over SPP. Our analysis finds that the aggregate gain comes from two sources, and the contributions are nearly equal.

Figure 3.8 depicts the two-branch Transformer-based model synthesized for the experiment. Each branch consists of four repeated sequences of one multi-head attention and two linear (dense) layers, also known as Multi-Level Perceptron (MLP) layers. The branches are merged by a concatenation operator.

Both GRAPHPIPE and SPP produce the identical model partition on a budget of eight
Figure 3.9: Pipeline schemes devised by SPP (top) and GRAPHPIPE (bottom). They produce an identical model partition. The selected micro-batch sizes are different: 2 (SPP) v.s. 4 (GRAPHPIPE), which results in a better compute efficiency for GRAPHPIPE. Both methods deem it unnecessary to employ data parallelism primarily because doing so would have split a smaller micro-batch size even further, which would have harmed compute efficiencies. The pipeline depths are also different: 8 (SPP) v.s. 4 (GRAPHPIPE), which results in a smaller pipeline depth for GRAPHPIPE. This improvement comes purely from the fact that GRAPHPIPE can produce a pipeline scheme that allows for concurrent execution of parallel branches.

There is another subtle, yet key difference. Since GRAPHPIPE reduces the pipeline depth by half, the activation memory footprints for early stages are smaller for the GRAPHPIPE strategy. As a result, GRAPHPIPE can choose a micro-batch size from a wider range of candidates, and indeed selects a size of 4. The compute efficiency improvement from choosing a larger micro-batch size over SPP (which chooses a size of 2 due to larger activation memory footprints) leads to a larger number of samples processed per unit time. This means that when the pipeline operates at full capacity, it processes training samples at a faster rate for GRAPHPIPE than for SPP. Our measurements show that the
gain from this compute efficiency improvement is 10%. The two gain sources combined, GRAPHPIPE achieves 20% higher throughput over SPP.

### 3.8 Related Work

We provide an overview of related works for techniques to train large DNNs in the literature.

**Pipeline parallelism.** Existing DNN frameworks [ABC+16, PGM+19a, JZA19b, RRRH20, SCP+18] employ sequential pipeline parallelism (SPP) where pipeline stages are strictly sequential. As we discuss in Section 3.2, SPP hinders parallel execution of computationally-independent components of a DNN and memory savings from reduced pipeline depth. While this limitation still exists as long as SPP is adopted, there are a variety of pipeline parallelism approaches to improve pipeline performance in other ways. These approaches fall into one of two paradigms: synchronous and asynchronous pipeline parallelism.

*Synchronous pipeline parallelism* [HCB+19, NPS+21, FRM+21, ZLZ+22, UJW+22] refers to a set of techniques in which the model parameters spread across devices are updated synchronously after each mini-batch is processed in one training iteration. The DNN training semantics is preserved, thus statistical convergence issues do not arise. But the synchronous updates fill and drain the pipeline periodically over iterations, hurting throughput. Our graph pipeline parallelism mitigates this issue by reducing pipeline bubbles better than sequential pipeline parallelism.

*Asynchronous pipeline parallelism* [NHP+19, NPS+21, TNP21, YZZ+22] refers to a set of techniques in which the model parameters spread across devices are updated asynchronously. Although this mode may suffer from statistical convergence issues as devices execute their stages using out-of-sync model parameters, it keeps the pipeline full at nearly all times. Graph pipeline parallelism helps us reduce total device memory usage, thus use a larger micro-batch size to execute operators at a higher operational intensity compared to sequential pipeline parallelism. This enables us to process training data faster while the pipeline is full, thus improves training throughput.

**Multiple pipeline stages per device.** In the above pipeline-parallel techniques, each device contains only one pipeline stage. It has been shown that assigning multiple non-contiguous stages to a device can reduce pipeline bubbles [NSC+21, LP22] and reduces memory consumption imbalances across stages [LH21, LCZY23]. Earlier work GEMS [JAA+20] has a similar idea but does not utilize the pipeline well — devices are idle for most of the time and waiting for results from other stages. These techniques are orthogonal to graph pipeline parallelism, and thus can be applicable upon some modifications.

**Data parallelism.** Data parallelism [Val90, Kri14, LAP+14, GDG+17, MHH+21] is one of parallel DNN training techniques in which every device has a local copy of a DNN to train in its entirety and a batch of training data is split across devices. Each device updates
its model parameters based on its share of training data and synchronizes the parameters periodically with other devices. In our work, we apply data parallelism within a pipeline stage to which we assign multiple devices, in order to balance stage execution times in a more fine-grained manner compared to applying pipeline parallelism only.

**Automatic DNN parallelism.** There are a number of automated approaches [ZLZ+22, UJW+22, JZA19b, TNP21, NHP+19, MPL+17, WHL19] to combine data, pipeline, and tensor parallelisms [SPP+19]. Existing works first partition a DNN into sequential pipeline stages (SPP) and then apply other parallelisms to each stage. Our automated framework generalizes existing pipeline parallelism to form graph pipeline parallelism and combines it with data parallelism while it is also possible to combine tensor parallelism with GRAPHPIPE in the same way as existing works [ZLZ+22, TNP21].

**Memory optimizations.** Managing memory usage patterns is critical in training large DNNs. Thus, on top of pipeline-parallel techniques, memory optimization techniques [CXZG16, RGC+16, WYZ+18, ZYS+19, CSO+20, PSD+20, HJL20, JKN+20, RRA+21] can also be modified and applied in order to further improve the performance and scalability of graph pipeline parallelism.

### 3.9 Conclusion

We have developed graph pipeline parallelism where pipeline stages form a directed acyclic graph whose edges indicate execution orders of forward and backward passes in pipeline-parallel DNN training. This design encourages concurrent execution of parallel branches for superior performance. We have also developed a distributed system GRAPHPIPE, and through experiments using three multi-branch models, showed that GRAPHPIPE achieves up to 1.61× higher training throughputs and > 9× faster solution search times over existing baselines that operate in a strictly sequential manner.
3.10 Appendix

3.10.1 Greedy Schedule Implementation

The greedy scheduler described in Algorithm 4 produces a schedule \( \Pi_i \) that minimizes the running time for stage \( i \) per training iteration and also the peak memory usage for stage \( i \). It iteratively augments \( \Pi_i \) by adding either a forward pass or a backward pass. The final \( \Pi_i \) is a sequence of forward and backward passes that constitutes a time- and memory-usage-optimal schedule. Function \( \text{IsTimeOptimal} \) checks if the micro-batch schedule leads to the minimum running time for stage \( i \). In the following proof, we use \( \text{time-optimal} \) to describe a micro-batch schedule that minimizes the running time for a stage and use \( \text{optimal} \) to describe one that minimizes peak memory usage (= number of in-flight data samples) among time-optimal schedules.

Algorithm 4 Greedy Scheduler

Input: \( b_i, \{\Pi_j, b_j\} \) for each successor stage \( j \) of stage \( i \)
Output: Micro-batch schedule \( \Pi_i \)

1: \( \Pi_i \leftarrow fw_i^1 \)
2: for \( l \leftarrow 1 \ldots 2B/b_i - 1 \) do
3: \( c_{fw} \leftarrow \text{number of } fw_i^l \text{’s in } \Pi_i \)
4: \( c_{bw} \leftarrow \text{number of } bw_i^l \text{’s in } \Pi_i \)
5: if \( \text{IsTimeOptimal}(\Pi_i, bw_{c_b+1} \text{fw}_{c_f+1} \text{bw}_{c_b+2} \ldots \text{fw}_{B/b_i} \text{bw}_{B/b_i+c_b-c_f+1} \ldots \text{bw}_{B/b_i}, \{\Pi_j, b_j\}) \)
then
6: \( \Pi_i \leftarrow \Pi_i \text{bw}_{c_b+1} \)
7: else
8: \( \Pi_i \leftarrow \Pi_i \text{fw}_{c_f+1} \)

Definition 1.: A micro-batch schedule \( \Pi_i \) for stage \( i \) is \( \text{monotonous} \) if and only if it starts with \( \text{inflight}(\Pi_i)/b_i \) contiguous forward micro-batches, where we define \( \text{inflight}(\Pi_i) \) as the peak number of in-flight data samples for stage \( i \) according to schedule \( \Pi_i \).

Lemma 3.1.:
There exists an optimal micro-batch schedule \( \{\Pi_i\} \) such that \( \Pi_i \) is monotonous for each stage \( i \).

Proof. Let \( \{\Pi_i\} \) be an optimal micro-batch schedule. If \( \Pi_i \) is monotonous for each stage \( i \), the proof is complete. If not, let \( i_0 \) be the first stage for which \( \Pi_i \) is not monotonous. Construct \( \{\Pi'_i\} \) to be a micro-batch schedule where \( \Pi'_i = \Pi_i \) for \( i \neq i_0 \). Also, define \( l_g \) be the number of contiguous forward micro-batches that \( \Pi_g \) starts with, and construct \( \Pi'_{i_0} \) by scheduling \( \text{fw}_{l_g+1}^{i_0} \) in \( \Pi_{i_0} \) right after \( \text{fw}_{l_g}^{i_0} \). For each predecessor stage \( p \) of \( i_0 \), as \( \Pi_p \) is monotonous, \( b_{l_p} = \text{inflight}(\Pi_p) > \text{inflight}(\Pi_{i_0}) = \text{inflight}(\Pi'_{i_0}) \geq b_{i_0}(l_{i_0} + 1) \) holds, and so \( \{\Pi'_p\} \) does not increase the running time for stage \( i \). We repeat the above procedure until schedules of all stages are monotonous. \( \Pi'_{i_0} \) increases the number of contiguous forward micro-batches compared to \( \Pi_{i_0} \), and as the total number of forward passes is bounded,
the above schedule construction process is guaranteed to end in a finite time. As the final
result we obtain an optimal micro-batch schedule such that $\Pi_i$ starts with $\text{inflight}(\Pi_i)/b_i$
forward micro-batches (i.e., monotonous) for each stage $i$. This completes the proof by
construction.

\begin{theorem}

The greedy scheduler gives an optimal micro-batch schedule.

\end{theorem}

\begin{proof}

Let \( \text{ind}_{\Pi_i}(f_{w_i}) \) be the index of \( f_{w_i} \) in \( \Pi_i \). Let \( s \) be any time-optimal schedule
for stage \( i \) that starts with $\text{inflight}(s)/b_i$ contiguous forward micro-batches. Let \( s = rbw_{i+1}t \) be a decomposition representation, where \( r \) and \( t \) are subsequences of \( s \) and
there are $c_f$ forward micro-batches and $c_b$ backward micro-batches in \( r \). Then, \( s' = rbw_{i+1}f_{\ell+1}bw_{\ell+2}f_{\ell+2} \ldots f_{B/m_i} \ldots bw_{B/m_i} \) must be a time-optimal schedule, because
\( s' \) employs 1F1B schedule: $\text{ind}_{\Pi_i}(f_{w_k}) \leq \text{ind}_{\Pi'}(f_{w_k})$ and $\text{ind}_{\Pi_i}(bw_k) \geq \text{ind}_{\Pi'}(bw_k)$ for any \( k \).

Together with Lemma 3.1, we know that the greedy scheduler produces a schedule that
minimizes the numbers of in-flight micro-batches for stage \( i \).

Let us define that a schedule \( \Pi \) dominates a schedule \( \Pi' \) if $\text{ind}_{\Pi}(bw_k) \leq \text{ind}_{\Pi'}(bw_k)$ for any
\( k \). Note that if \( \Pi_i \) and \( \Pi_i' \) are both time-optimal schedules for stage \( i \) that minimize the numbers of in-flight micro-batches, and \( \Pi \) dominates \( \Pi_i' \), then we can always replace \( \Pi_i' \)
with \( \Pi_i \) while keeping the schedule for other stages valid. The remainder of the proof is
to show that for any time-optimal schedule \( \Pi_i' \) that minimizes the number of in-flight
micro-batches for stage \( i \) and starts with $\text{inflight}(\Pi_i')$ contiguous forward micro-batches,
the schedule \( \Pi_i \), given by the greedy scheduler dominates \( \Pi_i' \).

We prove by contradiction. we assume that \( bw_k \) is the first backward micro-batch such
that $\text{ind}_{\Pi_i}(bw_k) < \text{ind}_{\Pi_i}(bw_k)$. Let \( l = \text{ind}_{\Pi_i}(bw_k) \). There should be the same number of
backward micro-batches in the first \( l - 1 \) micro-batches in \( \Pi_i \), and \( \Pi_i' \), so we can obtain a
time-optimal schedule \( \Pi_i'' \) by replacing the first \( l - 1 \) micro-batches in \( \Pi_i' \) with those in
\( \Pi_i \). Since \( \Pi_i'' \) shares the first \( l - 1 \) micro-batches with \( \Pi_i \), and the \( l \)-th micro-batch in \( \Pi_i'' \)
is a backward micro-batch. According to the execution of the greedy scheduler, the \( l \)-th
micro-batch in \( \Pi_i \) should be a backward micro-batch, which is a contradiction.

\end{proof}

\subsection{3.10.2 \textit{kFkB} Schedule}

The \textit{kFkB} schedule of stage \( S_x \) is determined by

\[
\arg\min_{k_x} \max_{(S_x,S_y) \in V_S} \text{ComputeInFlight}(k_x, b_x, k_y, b_y, i_y),
\]

where \( i_y \) is the number of in-flight samples for stage \( S_y \). \text{ComputeInFlight}(k_x, b_x, k_y, b_y, i_y)
is computed according to Table 3.2:

\subsection{3.10.3 DNN Model Configurations}

The Multi-Modal Transformer-based model (MMT) for which we evaluate \textsc{GraphPipe}
consists of four parallel branches concatenated at the end and each branch consists of
eight Transformer layers (32 layers in total). Here, the input sequence length is 256. Each transformer layer has a hidden size of 1024, an embedding size of 1024, and 16 attention heads. The hidden size for a feed-forward layer following the attention layer has a hidden size of 4096.

The DLRM model for which we evaluate GRAPHPIPE consists of seven branches for dense features and seven branches for sparse features (embedding layers); these branches are concatenated at the end. Each branch for dense features includes four feed-forward layers. The hidden size of dense features and the following feed-forward layers is 4096. For sparse features, its hidden size is 64 and the embedding bag size is 100; embeddings in a single bag is concatenated. The number of entries in an embedding table is 1 million. Feed-forward layers post-processing the interaction also have the hidden size of 4096.

The CANDLE-Uno model for which we evaluate GRAPHPIPE consists of seven branches, each of which includes four feed-forward layers. All feed-forward layers have a hidden size of 4096.

For our end-to-end evaluations, we use the following ranges of mini-batch sizes for each device count to fit available unified GPU memory:

<table>
<thead>
<tr>
<th># Devices</th>
<th>MMT</th>
<th>DLRM</th>
<th>CANDLE-Uno</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>64</td>
<td>256</td>
<td>4096</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>512</td>
<td>8192</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>1024</td>
<td>16384</td>
</tr>
<tr>
<td>32</td>
<td>512</td>
<td>2048</td>
<td>32768</td>
</tr>
</tbody>
</table>

Note that we sweep over all possible micro-batch sizes given mini-batch sizes for each model to maximize training throughput.
Chapter 4

Cache Parallelism: Comparative Analysis of Parallelisms in Distributed LLM Inference for Long Sequence Application

A significant portion of large language model (LLM) applications (such as book writing, document summarization, and translation) requires a LLM serving system to process a long sequence of tokens (i.e., 100K words). However, the memory demands of the attention mechanism and the unpredictable nature of text generation pose performance challenges. Existing parallelism strategies, such as data and tensor parallelism, exhibit trade-offs in workload balance and communication overhead, especially for long sequences. We introduce cache parallelism (CP), a new scheme that partitions the attention module’s KV cache along the sequence length. CP aims to achieve the best of existing parallelisms by balancing workload while minimizing communication costs. We provide efficient CUDA kernels for CP and compare it with other parallelism strategies for distributed LLM inference during text generation.

4.1 Introduction

The ability of large language models (LLMs) to handle long sequences unlocks a range of valuable real-world applications. Examples include a customer service chatbot that can reference an entire conversation history to provide personalized solutions, or an AI translator that translates entire books while preserving the flow and style of the original text. Long-sequence capabilities push LLMs beyond simple tasks, enabling them to tackle complex, nuanced problems that were previously difficult for machines to solve.

Transformer model has been a key to a pleathora of LLM applications. However, Trans-
formers become more and more memory-hungry as input and output sequence of models are required to be longer for various applications such as document summarization, code completion, long multi-turn conversation, etc. Specifically, for such long-context inference, attention (self-attention) module can be a memory bottleneck since it requires maintaining a large KV cache over each step of auto-regressive inference. Besides, the large KV cache creates a performance bottleneck for auto-regressive text generation (decode step).

However, the way text generation works in LLMs makes it hard to optimize performance. Most of the parts works on a single token at a time, except for the attention module, which gets more demanding as the output lengths. It’s impossible to know in advance how much tokens the model will generate for a given input. These unpredictable characteristics make it difficult to choose the right parallelism methods to efficiently distribute the workload during text generation.

Each parallelism strategy has strengths and weaknesses when it comes to the LLM decode stage for long sequence scenarios. Data parallelism doesn’t require communication between devices, but it suffers from workload imbalance issue caused by variance in very long output sequence. Tensor parallelism saves memory and balances the workload well, but it has costly communication overhead and can sometimes lead to low compute utilization. These trade-offs become especially crucial when dealing with longer input sequences.

We introduce a new parallelism scheme called cache parallelism (CP), designed to leverage the advantages of existing parallelisms. It partition KV cache of the attention module along the sequence length dimension. We show this mitigates workload balance issue of DP while achieving smaller communication cost than TP. We also implement efficient CUDA kernels that locally compute attention for partial KV cache on each device and reduces them across devices. Lastly, we compare various parallelism strategies for distributed LLM inference during the decoding stage.

To sum up, our work make the following key contribution:

- We explore a new parallelism scheme, cache parallelism (CP). This scheme divides large KV cache along the sequence length dimension to effectively scale attention.
- We offer custom CUDA kernels that support efficient distributed attention computation with partial KV cache.
- We provide comparative analysis of diverse parallelism strategies for distributed LLM inference during the decode step.

4.2 LLM Inference and Parallelisms

In this section, we describe an overview of how LLM inference pipeline works and a design space of parallelisms for the distributed LLM inference.
4.2.1 LLM Architecture

The state-of-the-art LLMs are all variants of Transformer model. They all consist of multiple layers of the Transformer block. Each transformer block contains three key parts:

- **Projection**: Projection adds modeling power by translating state vectors into different subspace and there are two types: input and output projections. The input projection takes the input vector and projects it to three different vectors - query, key, and value. The output projection takes the output vector from the attention operator and projects it to a new output vector.

- **Multi-head Attention**: It is the key part that models relations between different tokens in sequence and their importance. This is effectively a matrix multiplication between query, key, value and a softmax operation to model the importance of relation between tokens. It has multiple heads, each of which encodes such information with different representation in parallel.

- **Feed-Forward Network (FFN)**: This processes hidden state vectors further and follows the attention operation in the Transformer block. It is usually two linear projections with an activation function in the middle (e.g., ReLU, SiLU, etc.).

4.2.2 LLM Inference

LLM inference pipeline consists of two key steps: 1) prefill and 2) decode.

- In the prefill step, LLM takes a prompt sequence as an input and generates the key-value cache (KV cache) for each Transformer layer on top of first output token.
- In the decode step, LLM generates a next token based on the prompt and the current output token. Note that this takes multiple auto-regressive iterations and a KV cache of new token will be added to the existing KV cache after every iteration.

4.2.3 Parallelisms for LLM Inference

Given the recent trend of larger models with longer sequence, LLM inference pipeline gets even hungrier for memory footprint and compute power. To handle this, distributed LLM inference is necessary and multiple parallelisms have been actively deployed.

**Data parallelism.** Data parallelism intend to manage lots of requests for large language models (LLMs) by replicating its models across devices. This means making copies of the LLM – each copy, called an instance, has the entire model. These instances then work independently on separate batches of requests in a concurrent fashion. Data parallelism does not require synchronization cost, but it entails heavy memory footprint from parameter replication. Besides, it also suffers from the workload imbalance of requests because each request have uncertainty in how much work it requires for decoding.

**Continuous batching.** In LLM inference, we do not have the information how long each request will take to finish. Besides, decode step has much lower compute efficiency than prefill step for the same batch size, so it is often beneficial to add more requests to a batch for decode steps to improve throughput. Thus, it is a common practice to
continuously batch more requests after every decode iteration in LLM inference. It is known as continuous batching [YJK+22].

**Tensor parallelism.** Tensor parallelism [SPP+19] is a technique where we take parameters for each layer of LLMs and split them across multiple GPUs. It allows the memory footprints of parameters to be distributed, saving memory footprint and making it possible to work with larger models and longer sequence. Under the assumption that the number of attention heads is multiple of the number of devices, it guarantees great workload balance regardless of how long each request takes. Still, it comes with significant synchronization cost (i.e., two all-reduce operations per layer).

### 4.3 Trade-offs of Existing Parallelisms for LLM Decode

We see that the unique characteristics and uncertainty of decode leaves significant performance on the table. First, the decode step only processes a single token per request per iteration no matter how long the prompt and output sequence are except for the attention module. The computation and memory footprint of attention module linearly grows with the increasing output sequence length though. Furthermore, we do not know how many decode iterations we would need for a given request. This makes it harder to choose parallelism methods for decode steps to achieve good workload balance.

Each of existing parallelism schemes has its own pros and cons for decode steps in LLM inference. The advantage of data parallelism is that it does not incur communication between different data-parallel instances. On the flip side, it replicates parameters on each instance. We find it critical especially in the long sequence regime since it causes more KV cache eviction to CPU due to memory shortage. Besides, it is vulnerable to workload balancing. For example, requests in each instance are very likely to end up having different number of output tokens (thus different workloads) even if the workload was previously balanced based on input sequence length.

On the other hand, tensor parallelism also has its own advantages and disadvantages. Its strength is memory footprint and workload balance. It shards parameters over devices, thus there is no duplicate memory footprint for parameters. It also achieves good workload balance since all devices jointly processes all requests and have same amount of workloads, under the condition that the attention head dimension is the multiple of the tensor-parallel degree (i.e., the number of devices). Its weakness is that it incurs two expensive all-reduce communication for every Transformer layer. It also leads to bigger block table management costs per device for PagedAttention since it has same batch size and sequence length for all devices. It could also lead to low compute utilization of attention kernel for models when head dimension divided by tensor-parallel degree is smaller.
4.4 Cache Parallelism

Given such limitations of existing parallelisms, we explore a new parallelism called cache parallelism (CP)\(^1\). The key observation behind it is that only the attention module computation in the decode step linearly scales with the KV cache size that increase over decode iterations. The other parts including projections and FFN always process a single token per requests regardless of KV cache size (or output sequence length). Naturally, we propose a parallelism method that partitions KV cache across devices and achieves good workload balance for the attention module compute.

The (KV) cache parallelism only applies to the attention module since it is about partitioning KV cache. Therefore, we need to combine it with other parallelisms for other components of LLMs (i.e., projection and FFN). We choose to combine it with data parallelisms for other parts. This is because data parallelisms for other parts saves communication cost while still achieving workload balance. The reason why we can achieve workload balance while still applying data parallelisms for other parts is because we exploit the fact that only attention module hinders from data parallelisms to achieve workload balance in LLM decode. Thus, the idea is to replace data parallelism for attention module with cache parallelism. To be concise, let us assume CP as CP for the attention and DP for the other parts.

With CP, the attention module requires two steps: 1) local attention compute (for local sharded KV cache) and 2) reduce attention results from all KV cache across devices. This mechanism is equal to the FlashDecode [Dao23], but across multiple devices instead of multiple GPU warps. Note that this is possible because attention operation is communicative and associative.

Let \(N\) be the number of devices, \(B\) be the number of requests in a batch, \(E\) be the hidden (or embedding) dimension, \(H\) be the number of attention heads, \(L\) be the KV cache length. Recall that the attention module is softmax\((QK^T)V\) where \(Q, K, V\) are vectors for query, key, and value. Thus, local output tensor shape is \(BHL(E/H)\). Let the index of \(L\) dimension be \(i\) and the index of \(E/H\) dimension be \(j\). For each request and attention head, we need to store three result tensors from local attention to perform reduce across devices:

\[
\begin{align*}
    m_j &= \max_i Q[i,:]K^T[;:]
    \\
    s_j &= \sum_{i=1}^n e^{Q[i,:]K^T[;:]} - m_j
    \\
    o_{ij} &= \text{softmax}(QK^T)V
\end{align*}
\]

The first term is for maximum value of \(QK^T\) to prevent exponential function of it from overflowing in softmax. The second term is the denominator term for softmax using the first term. The last term is the local attention result. After local attention,

\(^1\)Note that there is a concurrent work of ours [LPZ\(^+\)24].
we collect these local attention results for requests assigned to each device with the all-to-all communication. Lastly, we reduce attention results to get the final result as in FlashDecode [Dao23].

**Communication.** CP incurs one all-gather and one all-to-all operation per Transformer layer. We need one all-gather right before the attention module to gather query, key, value from all devices. Then we need another one all-to-all after local attention compute to gather and reduce attention results for each data-parallel instance for the following operations. Then CP takes the following communication volume per GPU assuming 1 byte per data for simplicity:

\[
\text{(all-gather) + (all-to-all)} = \frac{(N - 1)3BE}{N} + \frac{(N - 1)B(E + 2H)}{2}
\]

CP saves communication cost compared to TP since TP has two all-reduce operations that incur the following communication volume per GPU:

\[
(2 \text{ all-reduce}) = 4(N - 1)BE
\]

We also show that how large the gap of communication volume between CP and TP across different number of GPUs and batch sizes in Figure 4.1. As the analytical cost implies, CP has much lower communication volume and the benefit increases as we increase the batch sizes and the number of devices. Note that we use \(E = 4096\) from the Llama-7B model. Thus, this gap would be even larger with larger models.

### 4.5 Evaluation

This section aims to answer the following two questions:

- How does the throughput of various parallelism strategies scale with sequence length in long sequence regimes?
- What are the trade-offs among various forms of parallelism in terms of throughput and latency for varying sequence length?
- What are different factors that influence performance of different parallelisms and their varying trade-offs?
4.5.1 Evaluation Setup

For comparative analysis, we evaluate three different parallelisms for Transformer layer: Cache Parallelism (CP), Data Parallelism (DP), and Tensor Parallelism (TP). Given that cache parallelism is specifically relevant to the attention operator with KV cache at decode step, we opt for data parallelism for the FFN layer and other linear projections for CP evaluation. However, it’s worth noting that tensor parallelism is also a viable alternative, offering different trade-offs.

We implement all parallelisms on top of vLLM [KLZ+23]. In particular, we implement LLM implementation with parallelisms in 1000 lines of Python with Pytorch library [PGM+19b] and a CUDA kernel for cache parallelism in 1000 lines of CUDA codes. This kernel implementation includes local attention computation with global reduce operation while extending FlashDecoding and vLLM kernel optimization. We evaluate them on one compute node (in our group’s proprietary cluster) that has 4 NVIDIA RTX A5000 GPUs interconnected via NVLink. Since our focus is on a decode step of LLM inference, we evaluate a single decode step (i.e., generation of a single token per sequence). We use a popular Llama-7B model [TMS+23, TLI+23] configuration with 32 attention heads and hidden size of 4096. Specifically, we evaluate a single Transformer layer of it to maximize sequence length for our evaluation. Note that error bars have been excluded from the plots in this section due to the observation of minimal standard deviations (below 3%) across all results.

4.5.2 Scaling with Sequence Length

To compare how different parallelism strategies scale with increasing sequence length, we evaluate their throughput of decode steps across long sequence lengths for three different maximum batch sizes (32, 128, 512) in continuous batching scenarios. Small maximum batch size (e.g., 32) ensures low latency, but lower throughput while large batch size (e.g., 512) leads to high latency, but higher throughput. We have evaluation results for long prefix sequence in Figure 4.2 and long decode sequence in Figure 4.3.

Regardless of maximum batch sizes, the general trend is that the throughput of cache
parallelism is better than other parallelisms for both long prefix and decode regimes. DP is the slowest one among all parallelisms for long context scenarios because of the workload imbalance issue. Since we do not know how long final sequence will be, DP often suffers from the straggler issue and more CPU offloading during the decode step. Compared to TP, CP leads to better throughput because it has 1) lower communication volume and 2) less block table management overhead from PagedAttention [KLZ+23]. Note that block table size is linearly correlated with batch size and sequence length. In TP, we need redundant block table management and copy for every device because TP shard activation in head dimension, which does not reduce the size of block table per device.

4.5.3 Trade-off between Latency and Throughput

We assess the latency and throughput trade-off of varying parallelisms across different long prefix sequence length scenarios in Figure 4.4. Each dot for each parallelism represents the maximum batch size increasing from left to right (32, 64, 128, 256, 512). Regardless of sequence length, DP is strictly worse than other two parallelisms for the reasons mentioned above. CP is mostly strictly better than TP except for shorter sequence regime (10K) thanks to lower communication volume and smaller block table
management overhead. Still, TP can be better than CP for shorter sequence. It’s because CP suffers from lower compute utilization of attention kernel with shorter sequence since it shards attention computation in sequence length dimension. Note that for longer sequence scenario, performance actually degrades after certain batch sizes. This happens because it experiences a lot more sequence eviction to CPU from memory pressure with long sequence.

### 4.5.4 Execution Time Breakdown

We exhibit the execution time breakdown for parallelisms across different prefix sequence length. To make it easier to compare across different parallelisms, we categorize execution time into three: 1) GPU compute, 2) GPU communication, 3) CPU overhead. GPU compute only includes the execution time for GPU kernel execution. GPU communication involves collective communication across GPUs for TP and CP. CPU overhead covers data copy from CPU memory to GPU memory, block table management, or any other CPU execution.

For GPU compute, there are only a marginal difference across parallelisms while TP tends to be the slowest due to slightly worse compute utilization from small head dimension from partitioning. In terms of GPU communication, TP has larger communication volume than CP, thus it takes longer. DP takes the longest time for CPU overhead among all parallelisms due to the straggler issue and more sequence eviction to CPU during decode steps. Sequence eviction problem is worse from the straggler where we need to store a lot more tokens in the memory than others. Note that TP also takes more CPU overhead than CP because of larger block table to copy and manage per GPU.

### 4.6 Related Work

**Distributed Transformer Inference.** The explosive popularity of Transformer models has led to a variety of distributed systems specialized for Transformer inference: PaLM inference [PDC+23], DeepSpeed Inference [ARZ+22], FasterTransformer [Fas], EffectiveTransformer [Eff], Hugging Face Accelerate [HFA], Orca [YJK+22], and Light-
Many of such systems focus on scaling to gigantic Transformer models \cite{CND+, Ope23, BMR+20}. Given huge memory footprint of such models, it is crucial to design an efficient model parallelism strategy. Researchers have proposed several parallelism approaches tailored to Transformer inference. Megatron-LM \cite{SPP+19} suggests a simple, yet effective model parallelism method for a Transformer layer to minimize communication. Further research \cite{PDC+23} have investigated multi-dimensional sharding strategies for Transformer inference and their tradeoffs between latency and model FLOPS utilization on TPU-v4 \cite{JKL+23}.

**Efficient Transformer Inference.** Researchers have witnessed that high FLOP count and communication volume fundamentally limits Transformer inference performance \cite{PDC+23, IDBN+21}. To combat this issue, numerous efforts \cite{GA22, GKD+21} have been made in investigating techniques such as model design\cite{SMM+17, LLX+20, KHB+21, FDZ22, Sha19}, sparsity \cite{JCM+21, FA23, HABN+21}, quantization \cite{ZYYH18, ZBIW19, AWA+21, DLBZ22}, offloading \cite{SZY+23, WYZ+18, HJL20, RRA+21, RRR+21}, etc. Several studies have explored efficient attention \cite{DFE+22, CGRS19, SGBJ19, KKL20, CLD+20, RSVG21} and new architectures such as mixture of expert \cite{SMM+17} and multi-query attention \cite{Sha19} to reduce FLOP count per token. Quantization is also popular method to minimize memory footprint and communication volume. We also leverage some of quantization methods to further boost inference performance. Our method is orthogonal to these works and can be combined with them to maximize performance.

### 4.7 Conclusion

Our work introduces cache parallelism (CP) as a novel strategy to optimize LLM inference for long sequences. CP addresses the trade-offs of existing parallelism methods by partitioning the attention module’s KV cache, improving workload balance and reducing communication overhead. Our efficient CUDA kernels further enhance CP’s performance. Experimental results demonstrate that CP offers advantages for distributed LLM inference during the decoding stage, especially in long-sequence scenarios. This research opens avenues for further exploration into specialized parallelism schemes that can push the boundaries of LLM capabilities for complex real-world applications.
Chapter 5

Conclusions

This thesis tackles the challenge of building portable, automated ML systems. We focus on improving the integration of different DL backends and automating how DL computations are parallelized across device cluster. Our work includes designing a portable user interface, a flexible system design, and algorithms to automatically optimize performance. We highlight our contribution with each pillar work here:

- We introduce Collage, a novel DL framework to streamline the integration of different DL backends. Collage facilitates flexible backend placement optimization based on backend capabilities, workload characteristics, and the available execution environment.
- We present GraphPipe: This distributed system enables efficient and scalable DNN training by intelligently partitioning DNNs and optimizing micro-batch schedules. GraphPipe’s novel approach preserves DNN topology, leading to significant improvements in memory efficiency and GPU utilization.
- We offer comparative analysis of various parallelism strategies for distributed LLM inference in long sequence applications. This work focused on Cache Parallelism (CP) and established valuable insights into trade-offs involved in long context scenarios.

These contributions collectively demonstrate a meaningful advancement in the design of automated and portable ML systems. The research findings provide a foundation for building more portable and efficient machine learning system.
Bibliography


[KHB⁺21] Sneha Kudugunta, Yanping Huang, Ankur Bapna, Maxim Krikun, Dmitry Lepikhin, Minh-Thang Luong, and Orhan Firat. Beyond distil-


[MGP+18] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In International Conference on Learning Representations, 2018. 25


73


Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023. 61, 62

[PGL+21] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training, 2021. 28


[SZY+23] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez,


In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 267–284, 2022. 48, 49


Pengcheng Yang, Xiaoming Zhang, Wenpeng Zhang, Ming Yang, and Hong Wei. Group-based interleaved pipeline parallelism for large-scale DNN training. In International Conference on Learning Representations (ICLR), 2022. 48


Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini,

[ZVSL18] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. 2018. 18

