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

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

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 Feburary 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

List of Figures

List of Tables

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 correpsonding 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 [CMJ+18, NGW+21, MXY+20, ZZL+20, 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 [Val90, Kri14, LAP+14, GDG+17, MHH+21], where we split the dataset across devices, and model parallelism [DCM+12, SCP+18, SPP+19, LLX+20, NSC+21, ZLZ+22], where we split the model itself and pipeline them [HCB+19, FRM+21, NPS+21, NHP+19, TNP21], 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 [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 [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 [CWV+14, WZS+14, KFT+19] and runtimes [Tena, KFT+19] as their backends and generate the fastest possible executable using these backends. However, current DL frameworks [PGM+19a, ABC+16, XLA, RFA+18, CBB+18, 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 [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 paritions 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× 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 based on work published at PACT 22.

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 $1.26\times$, $1.43\times$, $1.40\times$ on average on NVIDIA's RTX 2070 GPU, V100 GPU, and Intel's Xeon 8259CL CPU, respectively. Collage has been open-sourced [a] and deployed in Apache TVM.

[a]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.

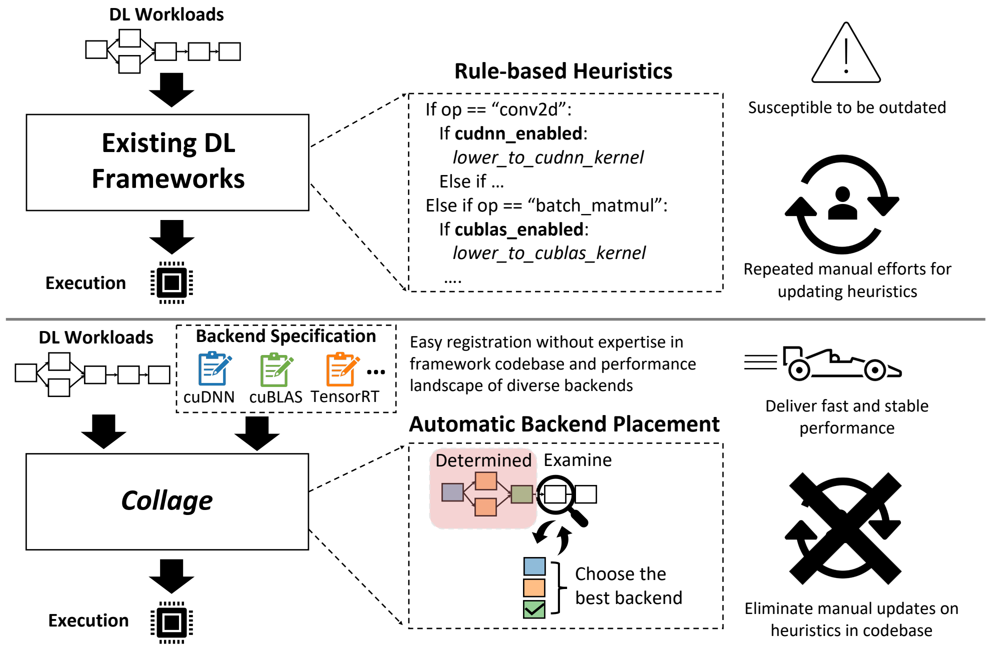

DL Workloads
Existing DL Frameworks
Execution
Rule-based Heuristics
If op == "conv2d":
If cudnn_enabled:
lower_to_cudnn_kernel
Else if ...
Else if op == "batch_matmul":
If cublas_enabled:
lower_to_cublas_kernel
....
Susceptible to be outdated
Repeated manual efforts for updating heuristics
DL Workloads
Backend Specification
cuDNN
cuBLAS
TensorRT
Easy registration without expertise in framework codebase and performance landscape of diverse backends
Deliver fast and stable performance
Automatic Backend Placement
Determined
Examine
Choose the best backend
Collage
Execution
Eliminate manual updates on heuristics in codebase

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

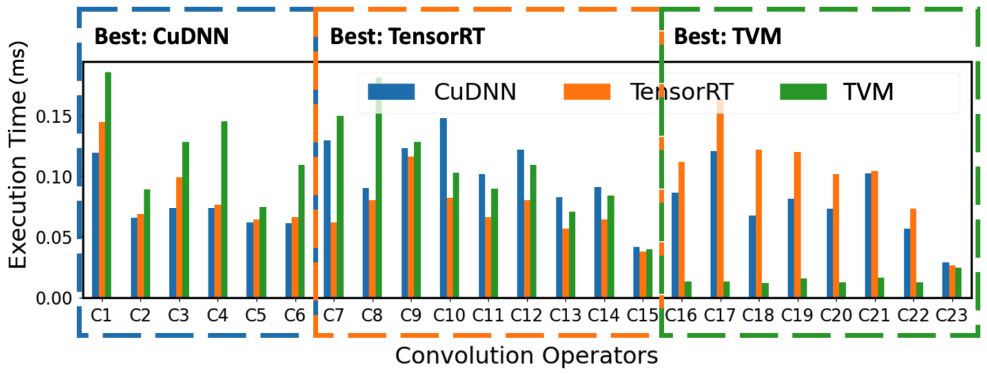


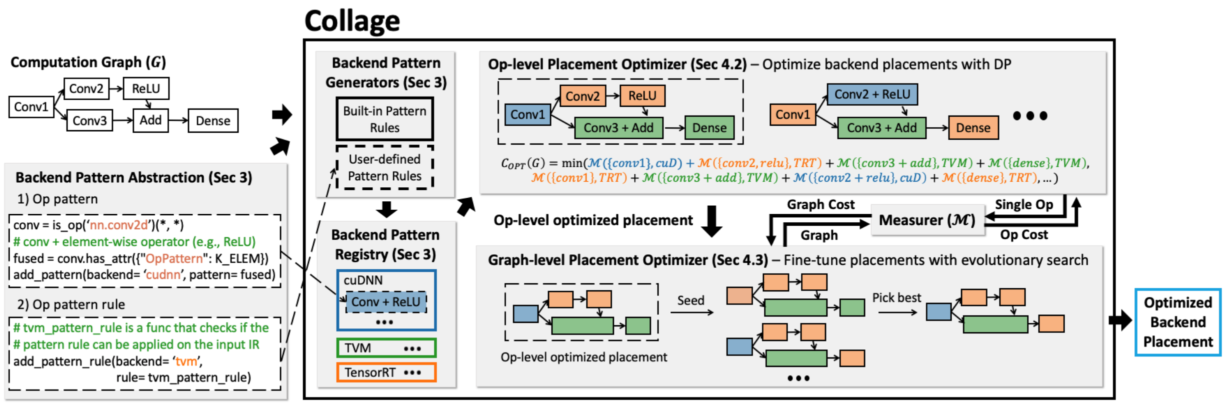
Figure 2.2: Performance of various convolutions (C#) with different configurations (e.g., input tensor shape, kernel size) in ResNext-50 on NVIDIA RTX 2070; Note that there is no single backend that is the best for all convolutions.

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 chacteristics 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


Collage
Computation Graph (G)
Conv1
Conv2
ReLU
Conv3
Add
Dense
Backend Pattern Abstraction (Sec 3)
1) Op pattern
2) Op pattern rule
Backend Pattern Generators (Sec 3)
Built-in Pattern Rules
User-defined Pattern Rules
Backend Pattern Registry (Sec 3)
cuDNN
Conv + ReLU
TVM
TensorRT
Op-level Placement Optimizer (Sec 4.2) – Optimize backend placements with DP
Conv2 + ReLU
Conv3 + Add
Op-level optimized placement
Graph Cost
Graph
Measurer (ℳ)
Single Op
Op Cost
Graph-level Placement Optimizer (Sec 4.3) – Fine-tune placements with evolutionary search
Seed
Pick best
Op-level optimized placement
Optimized Backend Placement

```
conv = is_op('nn.conv2d')(*, *)
# conv + element-wise operator (e.g., ReLU)
fused = conv.has_attr({"OpPattern": K_ELEM})
add_pattern(backend= 'cudnn', pattern= fused)
```

```
# tvm_pattern_rule is a func that checks if the
# pattern rule can be applied on the input IR
add_pattern_rule(backend= 'tvm',
                 rule= tvm_pattern_rule)
```

$$C_{OPT}(G) = \min(\mathcal{M}(\{conv1\}, cuD) + \mathcal{M}(\{conv2, relu\}, TRT) + \mathcal{M}(\{conv3 + add\}, TVM) + \mathcal{M}(\{dense\}, TVM), \mathcal{M}(\{conv1\}, TRT) + \mathcal{M}(\{conv3 + add\}, TVM) + \mathcal{M}(\{conv2 + relu\}, cuD) + \mathcal{M}(\{dense\}, TRT), \dots)$$

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.

diverse backends for a given hardware.

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 ($\mathcal{M}$). *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 [CMJ+18, NGW+21, CWV+14, Tena]. To enable more flexible specification, *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 *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, *Collage* uses a *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, *Collage* takes two different optimization strategies to exploit their differences. First, the *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 *graph-level placement optimizer* fine-tunes the optimized backend placement using evolutionary search [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 [CWV+14, CMJ+18, Tena, CMJ+18, 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: *pattern* and *pattern rule*. Pattern is a direct way to specify all supported operator patterns in *Collage*'s pattern language, which extends the Relay pattern language [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, *Collage* provides built-in patterns and pattern rules for popular backends [CWV+14, cuB, Tena, WZS+14, CMJ+18].

```python
 1 import collage
 2 
 3 # [Method 1] Explicit pattern specification
 4 # Pattern language to describe conv2d + add + relu.
 5 conv = is_op('conv2d')(wildcard(), wildcard())
 6 conv_constr = conv.has_attr({"data_layout": "NCHW"})
 7 conv_add = is_op('add')(conv_constr, wildcard())
 8 conv_add_relu = is_op('relu')(conv_add)
 9 
10 # Introduce new backend pattern to Collage.
11 collage.add_backend_pattern(backend='cuDNN',
12                             pattern=conv_add_relu)
13 
14 # [Method 2] Pattern rule specification
15 class MyPatternRule(collage.BasePatternRule):
16    # Define variables
17    kFusable = 0
18    kElemwise = 1
19    # ...
20    # Checker for the supported operators.
21    @staticmethod
22    def op_rule(op):
23       if op.name == "dense":
24          # Dense operator is always supported.
25          return True
26       elif op.name == "conv2d":
27         # constraints can be verified as well.
28         return op.attr["data_layout"] == "NCHW"
29       # ... rest of the op rule ...
30       return False
31 
32    # Checker for fusion patterns.
33    #   -- cur_type: type of current fusion group
34    #   -- src: seed operator node
35    #   -- sink: post-dominator of src
36    @staticmethod
37    def fusion_rule(cur_type, src, sink):
38       # If current fusion group contains
39       #   at least one conv/matmul (kFusable)
40       if cur_type == MyPatternRule.kFusable:
41         # Helper functions can be defined.
42         def fchecker(node_pattern):
43            return (node_pattern == MyPatternRule.kElemwise)
44         # Check if every operator between src and sink.
45         # Helper function can be passed as a checker.
46         if collage.check_path(src, sink, fchecker)):
47            return True
48       # ... rest of the fusion rule ...
49       return False
50 
51 # Introduce new pattern generation rule to Collage.
52 collage.add_backend_pattern_rule(backend='TVM',
53                                  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

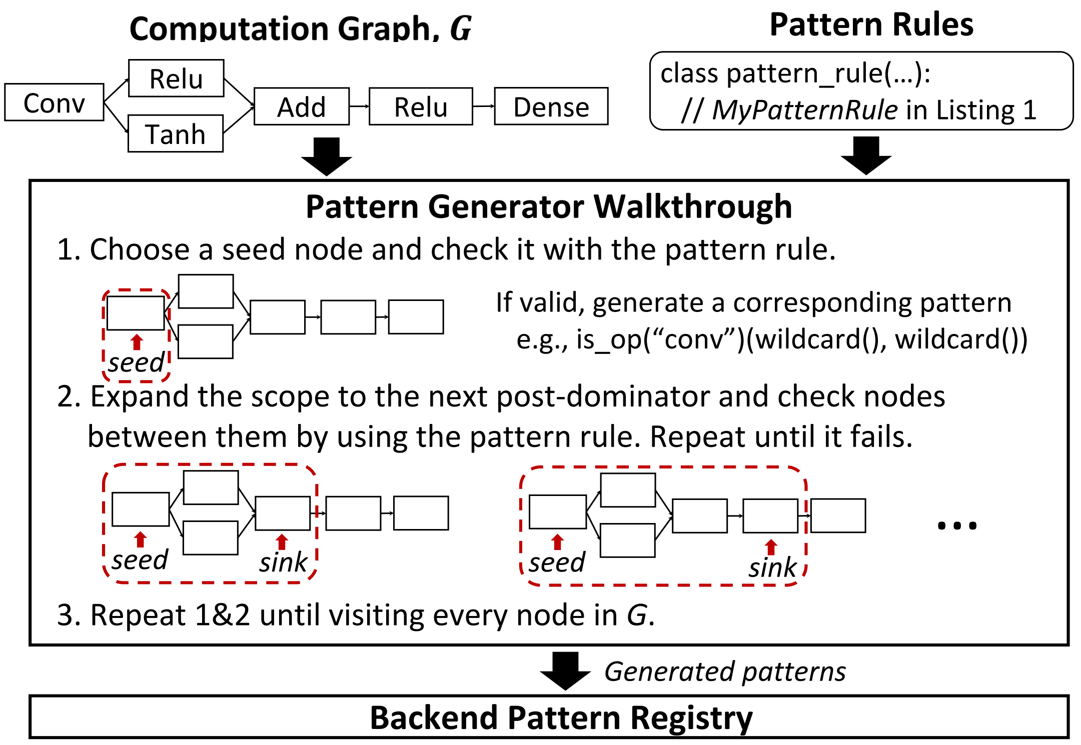

Computation Graph, G
Conv
Relu
Tanh
Add
Relu
Dense
Pattern Rules
class pattern_rule(...):
// MyPatternRule in Listing 1
Pattern Generator Walkthrough
1. Choose a seed node and check it with the pattern rule.
seed
If valid, generate a corresponding pattern
e.g., is_op("conv")(wildcard(), wildcard())
2. Expand the scope to the next post-dominator and check nodes between them by using the pattern rule. Repeat until it fails.
seed
sink
seed
sink
...
3. Repeat 1&2 until visiting every node in G.
Generated patterns
Backend Pattern Registry

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 $\mathcal{G}$ and a set of backend patterns $\mathcal{B}$ in *Collage*'s backend pattern registry. $\mathcal{G}$ is a Directed Acyclic Graph (DAG) where each node represents a tensor operator (e.g., convolution, matrix multiplication). $b = (p, d) \in \mathcal{B}$ 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 $\mathcal{P}(\mathcal{G}) = \{(g_i, b_i) | b_i \in \mathcal{B}, \bigcup_{i=1}^{M} g_i = \mathcal{G}, g_i \cap g_j = \emptyset$ for all $i, j \in \{1, 2, \cdots, M\}$ where $i \neq j\}$ be a backend placement strategy on a computation graph $\mathcal{G}$ and $Cost(\mathcal{P}(\mathcal{G}))$ be the execution time of a placement $\mathcal{P}(\mathcal{G})$. In this work, we aim to find a backend placement strategy $\mathcal{P}_{opt}$ that minimizes $Cost(\mathcal{P}(\mathcal{G}))$. This problem can be formalized as follows:

$$\mathcal{P}_{opt}(\mathcal{G}) = \text{argmin}_{\mathcal{P}(\mathcal{G})} Cost(\mathcal{P}(\mathcal{G})) \tag{2.1}$$

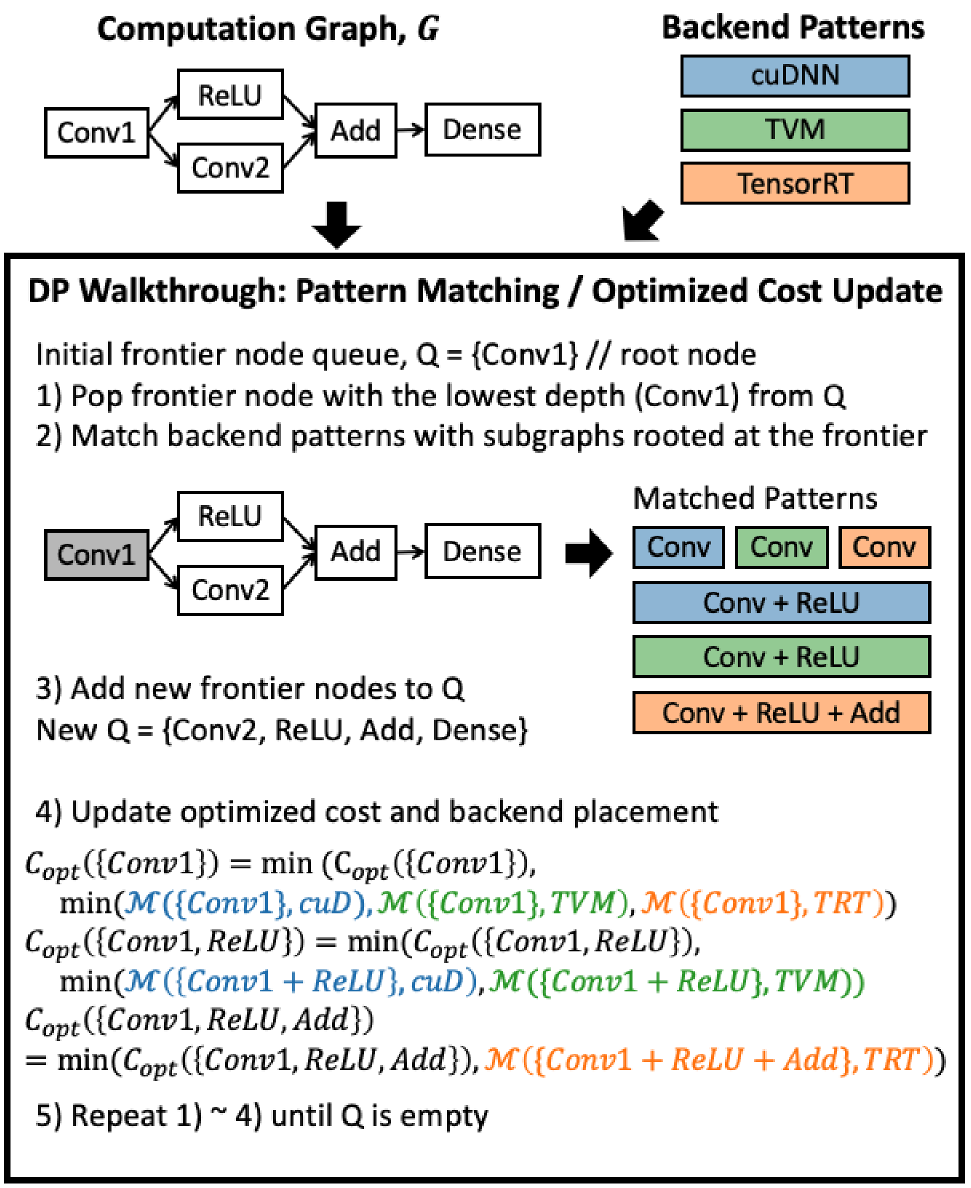
2.4.2 Op-level Placement Optimizer

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 $\mathcal{G}$ where $s_1 \cup s_2 = \mathcal{G}$, $s_1 \cap s_2 = \emptyset$. 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 $Cost(\mathcal{P}(\mathcal{G}))$:

$$Cost(\mathcal{P}(\mathcal{G})) = Cost(\mathcal{P}(s_1)) + Cost(\mathcal{P}(s_2)) + \epsilon \tag{2.2}$$

where ϵ 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., s1 and s2) 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



Computation Graph, G
Conv1
ReLU
Conv2
Add
Dense
Backend Patterns
cuDNN
TVM
TensorRT
DP Walkthrough: Pattern Matching / Optimized Cost Update
Initial frontier node queue, Q = {Conv1} // root node
1) Pop frontier node with the lowest depth (Conv1) from Q
2) Match backend patterns with subgraphs rooted at the frontier
Matched Patterns
Conv
Conv
Conv
Conv + ReLU
Conv + ReLU
Conv + ReLU + Add
3) Add new frontier nodes to Q
New Q = {Conv2, ReLU, Add, Dense}
4) Update optimized cost and backend placement
$C_{opt}(\{Conv1\}) = \min(C_{opt}(\{Conv1\}), \min(\mathcal{M}(\{Conv1\}, cuD), \mathcal{M}(\{Conv1\}, TVM), \mathcal{M}(\{Conv1\}, TRT))$
$C_{opt}(\{Conv1, ReLU\}) = \min(C_{opt}(\{Conv1, ReLU\}), \min(\mathcal{M}(\{Conv1 + ReLU\}, cuD), \mathcal{M}(\{Conv1 + ReLU\}, TVM))$
$C_{opt}(\{Conv1, ReLU, Add\}) = \min(C_{opt}(\{Conv1, ReLU, Add\}), \mathcal{M}(\{Conv1 + ReLU + Add\}, TRT))$
5) Repeat 1) ~ 4) until Q is empty

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 $\mathcal{P}_{opt}(s)$ and its cost $\mathcal{C}_{opt}(s)$ for any subgraph $s \subset \mathcal{G}$. This breaks down a problem of finding $\mathcal{P}_{opt}(\mathcal{G})$

into smaller problems of finding $\mathcal{P}_{opt}(s)$.

$$\begin{aligned} &\mathcal{P}_{opt}(s) = \mathcal{P}_{opt}(s_{min}) \cup \mathcal{P}(g_{min}) \\ &\mathcal{C}_{opt}(s) = \begin{cases} 0 & \text{if } s = \emptyset \\ \mathcal{C}_{opt}(s_{min}) + \mathcal{M}(\mathcal{P}(g_{min})) + \epsilon & \text{otherwise} \end{cases} \end{aligned} \tag{2.3}$$

where s_{min} and g_{min} are

$$\text{argmin}_{s' \cup g' = s, s' \cap g' = \emptyset} \{\mathcal{C}_{opt}(s') + \mathcal{M}(\mathcal{P}(g')) + \epsilon\} \tag{2.4}$$

s' represents a subgraph that is already examined while g' is a subgraph that is going to be evaluated with a measurer $\mathcal{M}(\cdot)$, 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 an simplified walkthrough example of our DP method. By traversing a computation graph $\mathcal{G}$, it solves smaller problems of finding $\mathcal{P}_{opt}(s)$ for a subgraph $s \subset \mathcal{G}$ and eventually discovers $\mathcal{P}_{opt}(\mathcal{G})$. 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 $\mathcal{G}$, 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 $\mathcal{S}$ (line 19). In Algorithm 1, the outermost while loop (line 3) takes $\mathcal{O}(N)$ times to traverse each frontier node in $\mathcal{G}$. For each frontier, there can be $\mathcal{O}(P)$ matches (line 6-8). For each match, the algorithm iterates over its $\mathcal{F}$ (line 10) and $\mathcal{S}$ (line 19) and takes $\mathcal{O}(F+S)$. Therefore, the overall time complexity of our op-level placement optimizer is $\mathcal{O}(NP(F+S))$. 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 $\mathcal{G}$ and set of backend patterns $\mathcal{B}$
Output: Optimized placement $\mathcal{P}_{opt}(\mathcal{G})$

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

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,

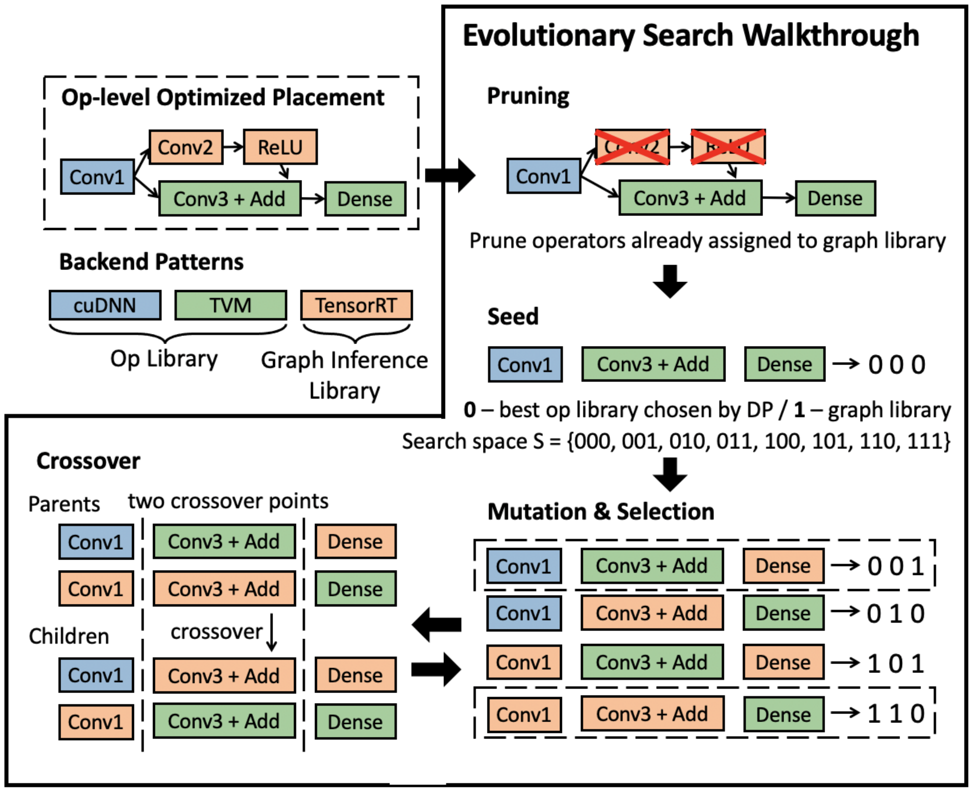

Op-level Optimized Placement
Conv1
Conv2
ReLU
Conv3 + Add
Dense
Backend Patterns
cuDNN
TVM
TensorRT
Op Library
Graph Inference Library
Evolutionary Search Walkthrough
Pruning
Prune operators already assigned to graph library
Seed
0 0 0
0 – best op library chosen by DP / 1 – graph library
Search space S = {000, 001, 010, 011, 100, 101, 110, 111}
Mutation & Selection
0 0 1
0 1 0
1 0 1
1 1 0
Crossover
Parents
two crossover points
Children
crossover

Figure 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.

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?

(§2.5.3)
- Does adding more backends improve the performance of *Collage*? (§2.5.4)
- How does backend placement optimized by *Collage* look like? (§2.5.5)

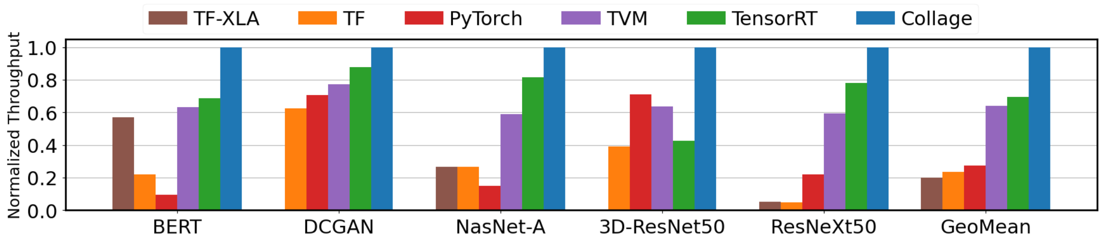
2.5.1 Experimental Setup

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.

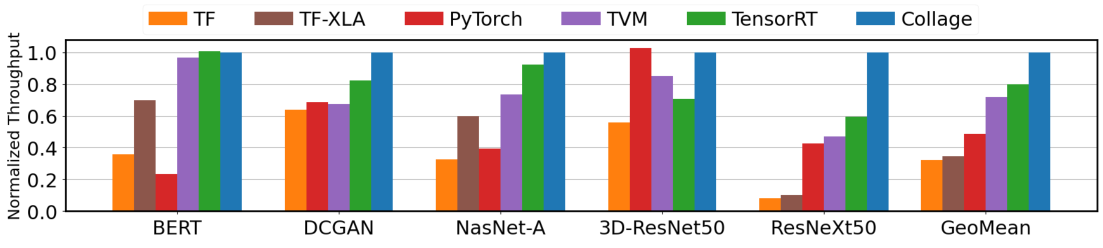
Baselines. We examine TensorFlow (TF) [ABC+16], TF-XLA [XLA], PyTorch [PGM+19a], TVM [CMJ+18], and TensorRT [Tena] as DL framework baselines. For TVM, we use AutoTVM to automatically generate the optimized operator schedules for each target. Note that we also integrate TensorRT and TVM as high-performance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

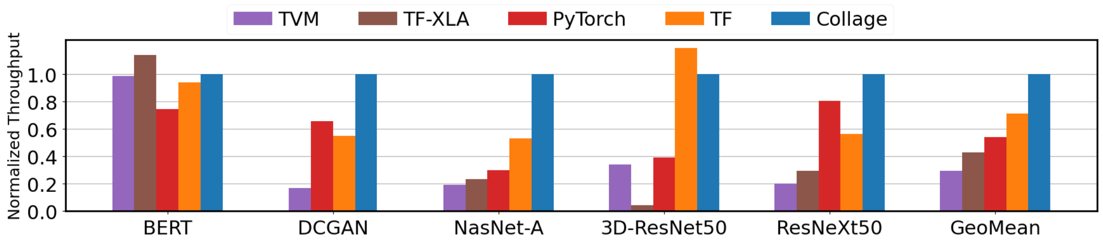
Each workload has its own characteristics in terms of its operators and structure. Most of recent models for language application such as BERT are basically a series of the Transformer layers that consist of batch matrix multiplication, layer normalization, softmax, etc. On the other hand, models for 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.



(a) NVIDIA Tesla V100



(b) NVIDIA GeForce RTX 2070



(c) Intel Xeon Platinum 8259CL

Figure 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).

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

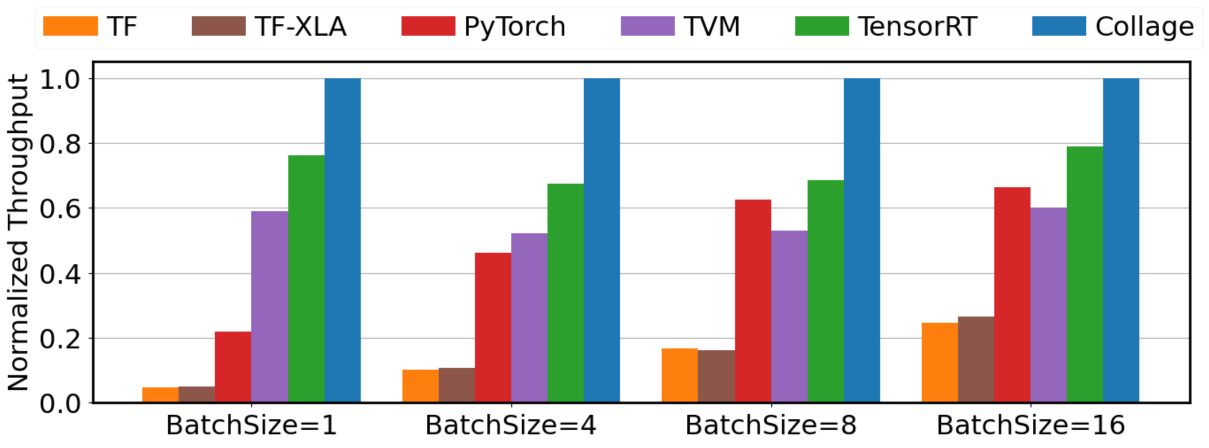


Figure 2.8: End-to-end performance with different batch sizes in ResNeXt50 on NVIDIA V100. Normalized throughput is the throughput normalized by the throughput of *Collage*.

geometric 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.

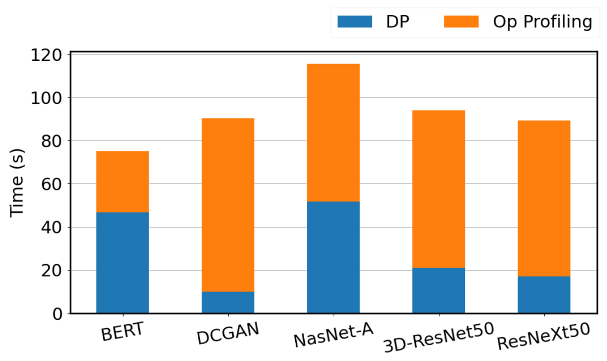
As the representative case, different batch sizes are also examined with ResNeXt50 on V100. Figure 2.8 indicates that *Collage* consistently outperforms the state-of-the-art frameworks across different batch sizes as well.

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.

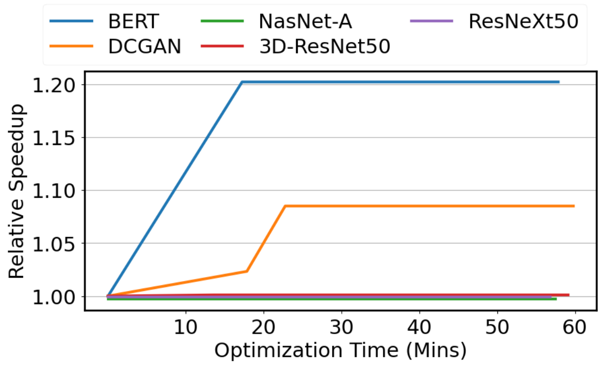
2.5.3 Optimization Time

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 accelerate 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.

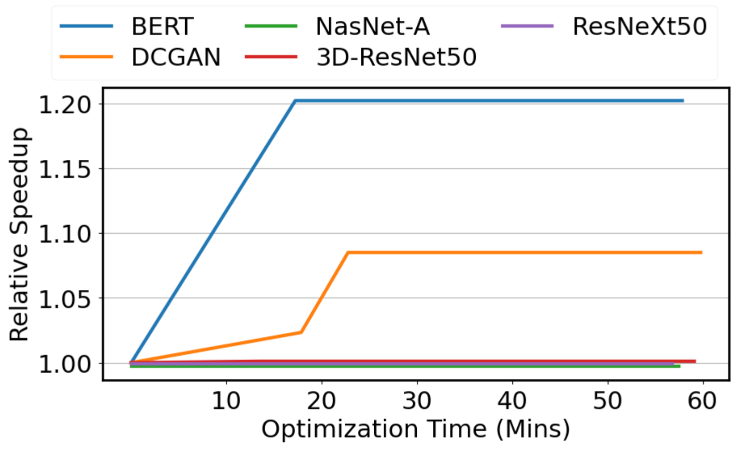


Figure 2.10: Performance improvement of graph-level placement optimization over time. The y-axis presents the speedup relative to the op-level placement optimization.

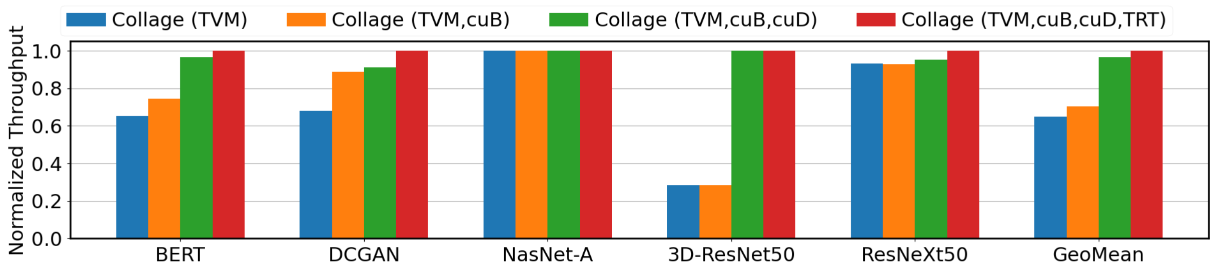


Figure 2.11: End-to-end performance of *Collage* with different number of backends on NVIDIA Tesla V100. Each throughput is normalized by the throughput of *Collage* (TVM,cuB,cuD,TRT). TVM, cuB, cuD, and TRT represents TVM, cuBLAS, cuDNN, and TensorRT.

2.5.4 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

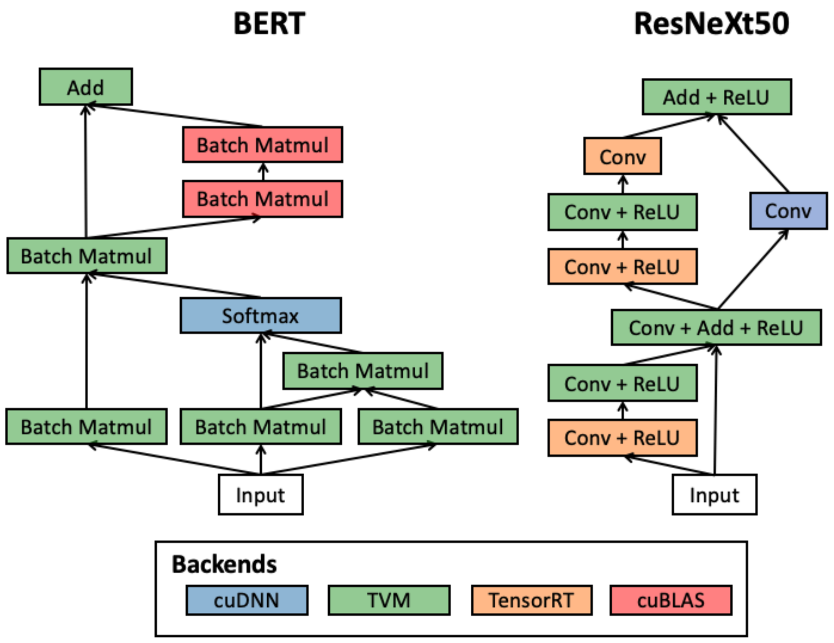


Figure 2.12: Representative backend placements discovered by *Collage* on V100 (Figure 2.7a). Note that *Collage* leverages various backends given their unique strength to enhance perigrmance.

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+21, 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 based on the work in submission to USENIX ATC 24.

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 GPPstrategies 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 GPPstrategies. 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 GPPinvolves 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.

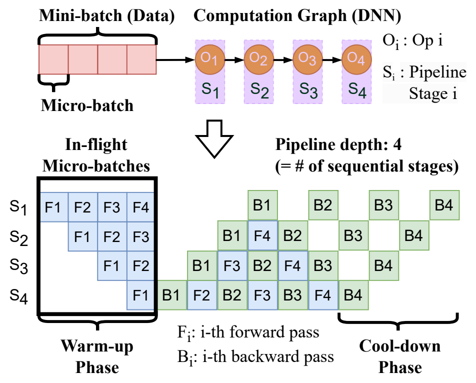


Figure 3.1: Pipeline parallelism for DNN training with basic terms used in this paper.

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 [KTW16, 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 GPPand SPP. Instead of enforcing a strictly sequential execution order of pipeline stages, GPPallows partitioning a DNN into stages whose dependencies are identified by a directed acyclic graph. GPPincludes SPP as a special case and can preserve the inherent topology of the DNN during stage partitioning. As a result, GPPenables concurrent execution of computationally-independent components, resulting in reduced memory requirement and improved GPU performance compared to SPP.

GPPinvolves a significantly larger and more complicated search space of parallelization strategies compared to the SPP strategies considered by existing DNN systems. Discovering GPPstrategies 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 GPPstrategies 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 GPPstrategy. 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 GPPstrategy to enable performant and scalable DNN training.

Through experiments on three multi-branch DNNs (e.g., Multi-Modal Transformer [VSP+17, WCQ+23, RKH+21, Ope23, RPG+21, JYX+21], DLRM [NMS+19], and CANDLE-Uno [201]), we show that GRAPHPIPE can achieve up to 1.61× training throughput improvements over existing pipeline-parallel systems such as PipeDream and Piper. Despite the fact that GPPinvolves a much larger search space of parallelization strategies, GRAPHPIPE reduces the search time by 9-21× 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 GPPstrategies.
- 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× on a variety of DNNs.

3.2 Graph Pipeline Parallelism

Figure 3.2 compares sequential pipeline parallelism (SPP) employed by existing DNN systems [TNP21, NPS+21] and graph pipeline parallelism (GPP). Given a DNN and a set of devices, SPP and GPPproduce 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 linearzied 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, GPPpreserves 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 GPPa clear advantage: (potentially) concurrent execution of stages that are computationally-independent.

For the GPPstrategy in Figure 3.2, three stages S_1, S_2, and S_3 are computationally-

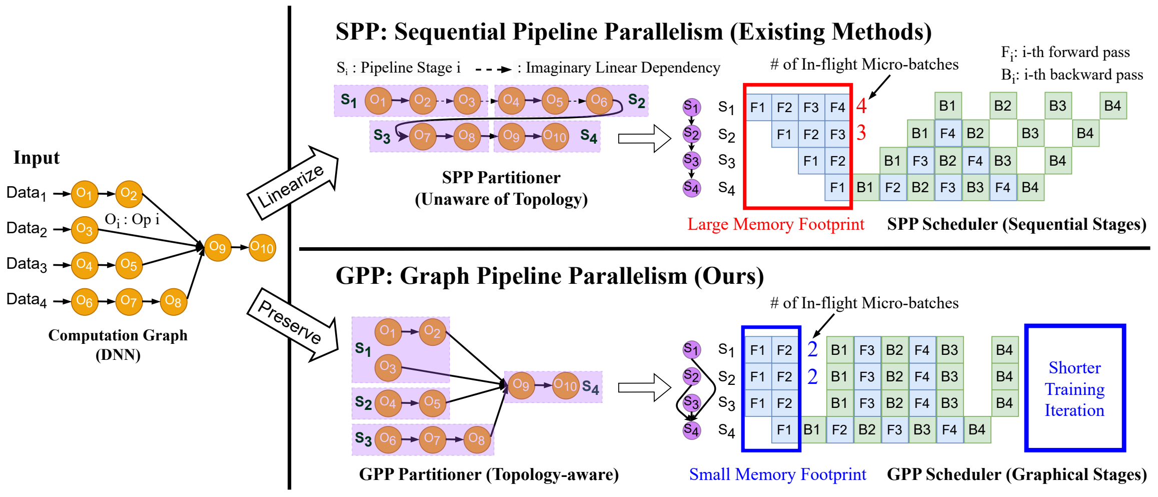

Input
Data1
Data2
Data3
Data4
Oi : Op i
Computation Graph (DNN)
Linearize
Preserve
SPP: Sequential Pipeline Parallelism (Existing Methods)
Si : Pipeline Stage i
: Imaginary Linear Dependency
SPP Partitioner (Unaware of Topology)
# of In-flight Micro-batches
Fi: i-th forward pass
Bi: i-th backward pass
Large Memory Footprint
SPP Scheduler (Sequential Stages)
GPP: Graph Pipeline Parallelism (Ours)
GPP Partitioner (Topology-aware)
Small Memory Footprint
GPP Scheduler (Graphical Stages)
Shorter Training Iteration

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 GPPinvolve 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, GPPachieves a shorter per-iteration training time (hence, a higher throughput) than SPP. Note that the topology-aware stage partitioning and scheduling of GPPaddress 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, GPPand 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 GPPand those for 4 micro-batches in SPP. All else being equal (i.e., an identical model partition by both), GPPhas 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 GPPaddresses 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 GPPcan significantly reduce.[2] For simplicity of presentation, Figure 3.2 assumes that the same micro-batch size is used by GPPand SPP. However, a lower device memory requirement of GPPover 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 GPPsuch 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 uses 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 GPPstrategy for distributed DNN training. As input, we are given (a) a computation graph $\mathcal{G}_C = (\mathcal{V}_C, \mathcal{E}_C)$ that represents the neural architecture of a DNN model, (b) a mini-batch size B, and (c) a device topology

[2]Note 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:

$$\min \quad \max_{S_i \in \mathcal{V}_S} \mathsf{TPS}(S_i; \mathcal{G}_C, B, \mathcal{D}) \tag{3.1}$$

$$\text{s.t.} \quad \max_{S_i \in \mathcal{V}_S} \mathsf{DeviceMemoryUsage}(S_i; \mathcal{G}_C, B, \mathcal{D}) \leq M. \tag{3.2}$$

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

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. Π_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 fw_j^i (or bw_j^i) to denote the forward (or backward) pass of the j-th micro-batch for S_i.

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

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 $\mathcal{G}_C$.
C2. If there exists $(u, v) \in \mathcal{E}_C$ such that $u \in \mathcal{G}_i$ and $v \in \mathcal{G}_j$, then $(S_i, S_j) \in \mathcal{E}_S$.
C3. $\mathcal{D}_1, \ldots, \mathcal{D}_{|\mathcal{V}_s|}$ form a partition of $\mathcal{D}$.
C4. For each micro-batch schedule Π_i, fw_k^i precedes fw_{k+1}^i, bw_k^i precedes bw_{k+1}^i, and fw_k^i precedes bw_k^i.
C5. For each $(S_i, S_j) \in \mathcal{E}_S$, the following must hold.
If $\beta_{ij} := b_j/b_i \geq 1$, $\mathsf{fw}^i_{\beta_{ij}\cdot k+1}, \ldots, \mathsf{fw}^i_{\beta_{ij}\cdot k+\beta_{ij}} \in \Pi_i$ precede $\mathsf{fw}^j_{k+1} \in \Pi_j$, and $\mathsf{bw}^j_{k+1} \in \Pi_j$

[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 $\mathsf{bw}^i_{\beta_{ij}\cdot k+1}, \ldots, \mathsf{bw}^i_{\beta_{ij}\cdot k+\beta_{ij}} \in \Pi_i$. Otherwise ($\beta_{ji} > 1$), $\mathsf{fw}^i_{k+1} \in \Pi_i$ precedes $\mathsf{fw}^j_{\beta_{ji}\cdot k+1}, \ldots, \mathsf{fw}^j_{\beta_{ji}\cdot k+\beta_{ji}} \in \Pi_j$, and $\mathsf{bw}^j_{\beta_{ji}\cdot k+1}, \ldots, \mathsf{bw}^j_{\beta_{ji}\cdot k+\beta_{ji}} \in \Pi_j$ precede $\mathsf{bw}^i_{k+1} \in \Pi_i$.

In words, *C1* mandates that all operators be covered by stages that do not overlap with each other, and *C2* 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. *C3* ensures that at least one device is allocated to every stage. *C4* dictates the orderings of forward and backward passes *within* a stage and *C5* dictates such orderings *across* different stages for correctness of pipeline parallelism.

3.4 System Overview

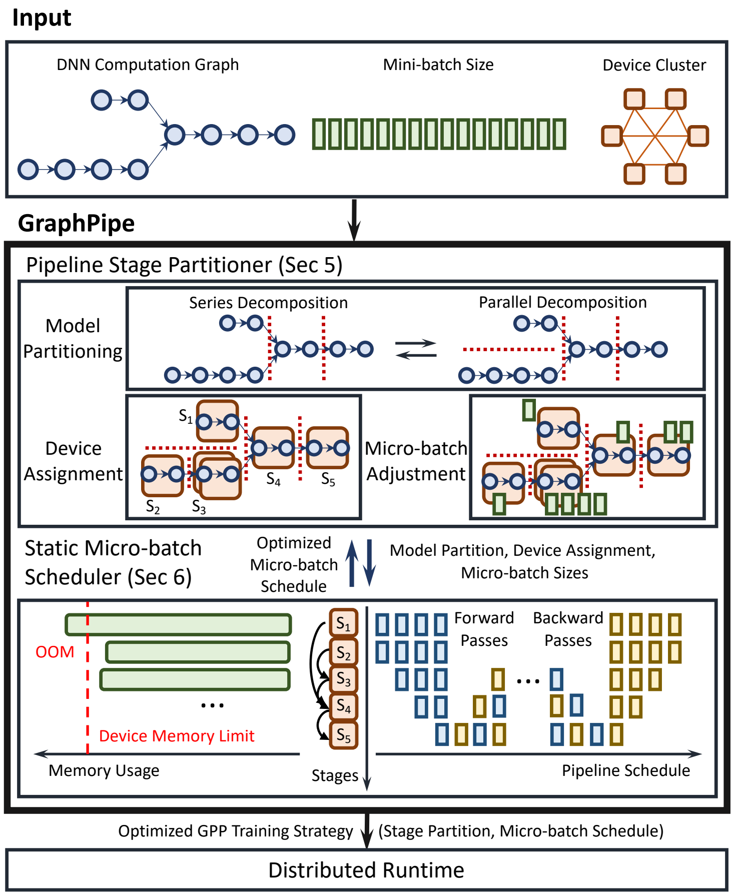


Figure 3.3: Overview of GRAPHPIPE

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 GPPstrategy 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 GPPstrategy 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 GPPtraining strategies generated by the optimizer of GRAPHPIPE. Using the distributed runtime as the testbed, we compare the performance of the generated GPPstrategies 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 $\mathcal{G}_C$, a mini-batch size B, and a device topology graph $\mathcal{G}_D$, and generates an optimized stage graph $\mathcal{G}_S$ 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 GPPstrategies. 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 $\mathcal{G}_C$, number of devices $|\mathcal{V}_D|$
Output: Optimized stage graph $\mathcal{G}_S$

```
1:  // MAXTPS: safe upper-bound for TPS of bottleneck stage.
2:  $t_l = 0$, $t_r =$ MAXTPS, $\mathcal{G}_S = \varnothing$
3:  while $t_r - t_l > \epsilon$ do
4:      $t_m = (t_l + t_r)/2$
5:      $\mathcal{G}_S^{best}$ = SearchStageGraph($\mathcal{G}_C, |\mathcal{V}_D|, t_m, B$)
6:      if $\mathcal{G}_S^{best} == \varnothing$ then
7:          $t_l = t_m$
8:      else
9:          $t_r = t_m$
10:         $\mathcal{G}_S = \mathcal{G}_S^{best}$
11: return $\mathcal{G}_S$
12:
13: function SearchStageGraph($\mathcal{G}_C, |\mathcal{V}_D|, t_m, B$)
14:     // $C$ is a set of candidate schedule configurations ($c$)
15:     for $c \in C$ do
16:         $\mathcal{G}_S^{new}$ = DP($\mathcal{G}_C, c_0, c, |\mathcal{V}_D|, t_m$) // $c_0$: dummy schedule configuration
17:         // PickBetter($\cdot$) picks one with less memory
18:         $\mathcal{G}_S^{best}$ = PickBetter($\mathcal{G}_S^{best}, \mathcal{G}_S^{new}$)
19:     return $\mathcal{G}_S^{best}$
20:
21: function DP($\mathcal{G}, c_f, c_b, d, t_{max}$)
22:     if this DP state has been visited then
23:         return corresponding $\mathcal{G}_S^{best}$ to this DP state
24:     $\mathcal{G}_S^{best} = \varnothing$ // Consider a given DP state as a SINGLE stage
25:     if EstimateTPS($\mathcal{G}, c_f, c_b, d$)$\leq t_{max}$ then
26:         // Optimize schedule via Algorithm 3
27:         $\Pi_{opt}$ = ScheduleStage($\mathcal{G}, c_f, c_b, d$)
28:         $\mathcal{G}_S^{best}$ = StageGraph($\mathcal{G}, \Pi_{opt}, d$)
29:     // Decompose a given DP state into two stages
30:     if $\mathcal{G}$ can be decomposed in series then
31:         for $(\mathcal{G}_1, \mathcal{G}_2) \in$ SeriesDecompose($\mathcal{G}$) 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:                     $\mathcal{G}_{S_2}^{new}$ = DP($\mathcal{G}_2, c_m, c_b, d_2, t_{max}$)
36:                     Update $i_m$ based on $\mathcal{G}_{S_2}^{new}$
37:                     $\mathcal{G}_{S_1}^{new}$ =DP($\mathcal{G}_1, c_f, c_m, d_1, t_{max}$)
38:     else if $\mathcal{G}$ can be decomposed in parallel then
39:         for $(\mathcal{G}_1, \mathcal{G}_2) \in$ ParallelDecompose($\mathcal{G}$) do
40:             for $d_1 \leftarrow 1$ to $d - 1$ do
41:                 $d_2 = d - d_1$
42:                 $\mathcal{G}_{S_1}^{new}$ = DP($\mathcal{G}_1, c_f, c_b, d_1, t_{max}$)
43:                 $\mathcal{G}_{S_2}^{new}$ = DP($\mathcal{G}_2, c_f, c_b, d_2, t_{max}$)
44:     $\mathcal{G}_S^{best}$ = PickBetter($\mathcal{G}_S^{best}, \mathcal{G}_{S_1}^{new} \cup \mathcal{G}_{S_2}^{new}$)
45:     return $\mathcal{G}_S^{best}$
```

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 $\mathcal{G}_C$ 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 $\mathcal{G}$) into two new subgraphs (say $\mathcal{G}_1, \mathcal{G}_2$), 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 $\mathcal{G}_C$ (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 $\mathcal{G}$ comprised of them). The input to a DP subproblem includes a computation graph $\mathcal{G} \subseteq \mathcal{G}_C$, 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 $\mathcal{G}$ 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 $\mathcal{G}_1$ and $\mathcal{G}_2$, where the sink node of $\mathcal{G}_1$ coincides with the source node of $\mathcal{G}_2$ (Line 30 in Algorithm 2).[6] We first solve the subproblem associated with $\mathcal{G}_2$. To do so,

[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 $\mathcal{G}_2$. We then solve the subproblem associated with $\mathcal{G}_1$.

- *Parallel decomposition:* We perform a parallel decomposition to create $\mathcal{G}_1$ and $\mathcal{G}_2$, where $\mathcal{G}_1$ and $\mathcal{G}_2$ 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 $\mathcal{G}_1$ and $\mathcal{G}_2$ may produce different optimal numbers of in-flight micro-batches for the shared source node. When there exists a subgraph $\mathcal{G}'$ preceding both $\mathcal{G}_1$ and $\mathcal{G}_2$, the data dependency between $\mathcal{G}'$ and $\mathcal{G}_1$, and that between $\mathcal{G}'$ and $\mathcal{G}_2$ 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 $\mathcal{G}$, 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_fFk_fB schedule, and the stage containing the sink node (i.e., sink stage) uses the k_bFk_bB schedule (we introduces 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 $\mathcal{G}_C$, $\mathcal{B}$ be the set of possible micro-batch sizes, $\mathcal{D}$ be the set of possible data-parallel degrees. The maximal element of $\mathcal{B}$ is upper-bounded by B. We consider powers of 2 for micro-batch sizes (i.e., $|\mathcal{B}| < \log_2 B$). Likewise, the maximal element of $\mathcal{D}$ is upper-bounded by $|\mathcal{V}_D|$ and $|\mathcal{D}| < \log_2 |\mathcal{V}_D|$ holds.

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

$$O((\log \mathsf{MAXTPS})NB|\mathcal{B}|^6|\mathcal{D}|^2) = O((\log \mathsf{MAXTPS})NB(\log_2 B)^6(\log_2 |\mathcal{V}_D|)^2)$$

We see that the inherent topological complexity of a given DNN $\mathcal{G}_C$ (denoted as N in the above analysis), mini-batch size B, and device budget $|\mathcal{V}_D|$ largely determine the theoretical time complexity. For the practical time complexity, however, reducing the number of candidates for micro-batch sizes (i.e., $|\mathcal{B}|$) and data-parallel degrees (i.e., $|\mathcal{D}|$) 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

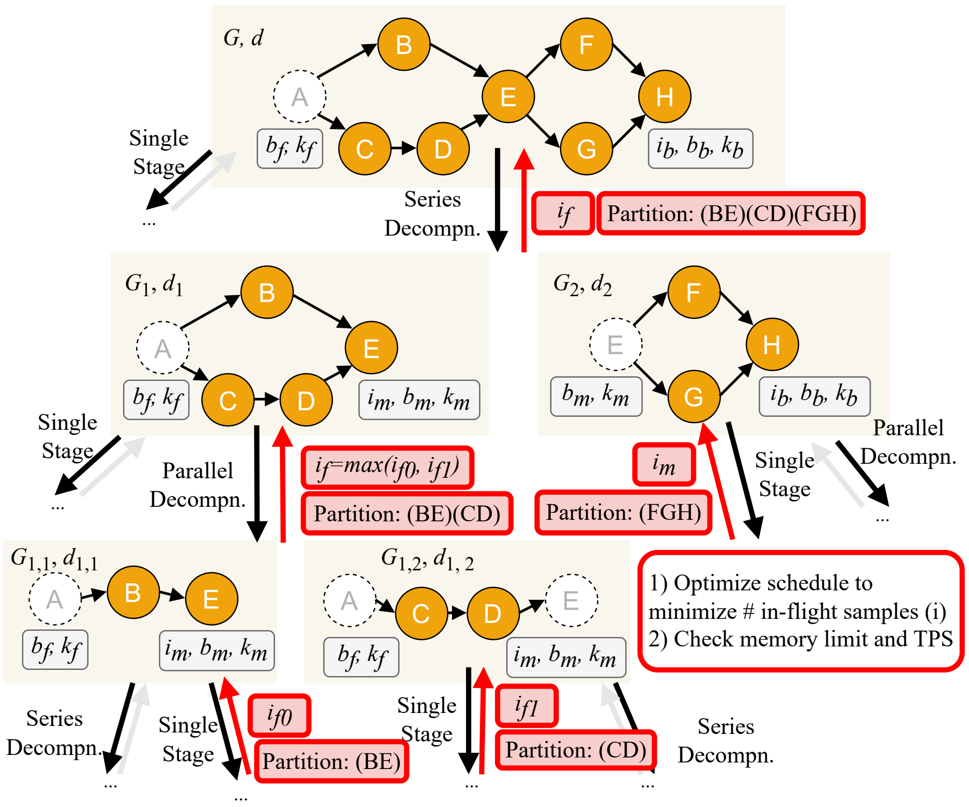


Figure 3.4: Pipeline stage partitioner performing series-parallel decompositions. Black arrows indicate subproblem formulations. Red arrows indicate solutions of the subproblems.

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 9-40$\times$ 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 $\mathcal{G}$, (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 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 $\mathcal{G}$, initial current and next stage schedule configurations $c_f, c_b$,
       number of devices $d$
Output: Optimized schedule $\Pi_{opt}$
 1: function SCHEDULESTAGE($\mathcal{G}, c_f, c_b, d$)
 2:     // Optimize schedule to minimize number of
 3:     // in-flight micro-batches using Table 3.2
 4:     // while respecting data dependencies
 5:     $i_f$ = 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:         $c_{opt} = \varnothing$ // Invalidate schedule $c_{opt}$
 9:     $\Pi_{opt} \leftarrow$ 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 $(S_i, S_j) \in \mathcal{E}_S$ of the stage graph $\mathcal{G}_S$ 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$(\cdot)$ 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 Π_i is said to be kFkB when there exist ℓ and k such that Π_i starts with ℓ forward passes (for warm-up), alternates between k backward and k forward passes, and ends with ℓ 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 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].

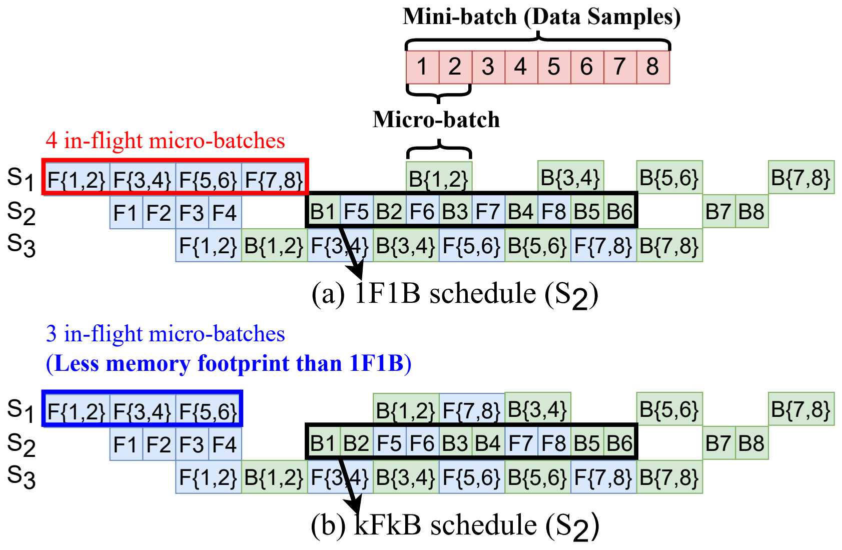
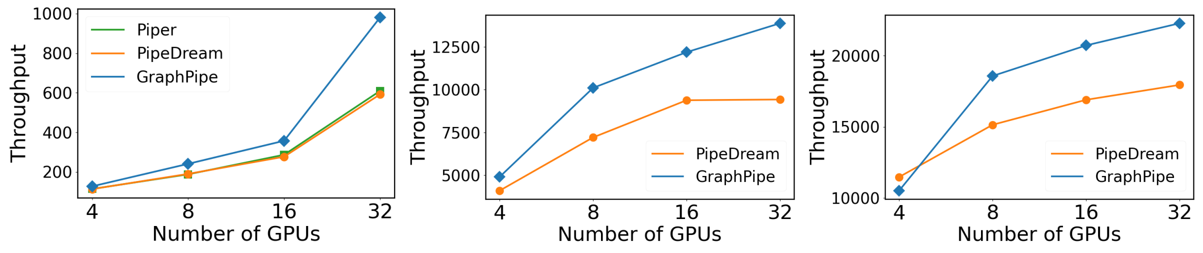

Mini-batch (Data Samples)
1 2 3 4 5 6 7 8
Micro-batch
4 in-flight micro-batches
S1 F{1,2} F{3,4} F{5,6} F{7,8} B{1,2} B{3,4} B{5,6} B{7,8}
S2 F1 F2 F3 F4 B1 F5 B2 F6 B3 F7 B4 F8 B5 B6 B7 B8
S3 F{1,2} B{1,2} F{3,4} B{3,4} F{5,6} B{5,6} F{7,8} B{7,8}
(a) 1F1B schedule (S2)
3 in-flight micro-batches
(Less memory footprint than 1F1B)
S1 F{1,2} F{3,4} F{5,6} B{1,2} F{7,8} B{3,4} B{5,6} B{7,8}
S2 F1 F2 F3 F4 B1 B2 F5 F6 B3 B4 F7 F8 B5 B6 B7 B8
S3 F{1,2} B{1,2} F{3,4} B{3,4} F{5,6} B{5,6} F{7,8} B{7,8}
(b) kFkB schedule (S2)

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.



(a) Multi-Modal Transformer (b) DLRM (c) CANDLE-Uno

Figure 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-Uno [201] at all but one GPU count configurations tested. Missing data points indicate that no training strategy can be found within reasonable timeframes.

3.7.1 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

[8]We 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.

# GPUs	MMT			DLRM			CANDLE-Uno		
	Piper	PipeDream	Ours	Piper	PipeDream	Ours	Piper	PipeDream	Ours
4	52.9 (440.5×)	2.57 (21.4×)	0.12	✗	6.39 (19.3×)	0.33	✗	3.84 (20.2×)	0.19
8	126 (165.7×)	11.9 (15.6×)	0.76	✗	31.3 (11.4×)	2.73	✗	17.0 (11.8×)	1.43
16	304 (101.3×)	44.3 (14.7×)	3.00	✗	131 (9.9×)	13.28	✗	66.10 (10.7×)	6.14
32	745 (73.7×)	151 (15.0×)	10.11	✗	505 (9.2×)	54.6	✗	234 (10.4×)	22.37

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 $9\times$ 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 $O(|\mathcal{D}|^2)$ time (Appendix D in [TNP21]), where $\mathcal{D}$ 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 $|\mathcal{D}| \geq \prod_{i=1}^{n} k = k^n$. 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

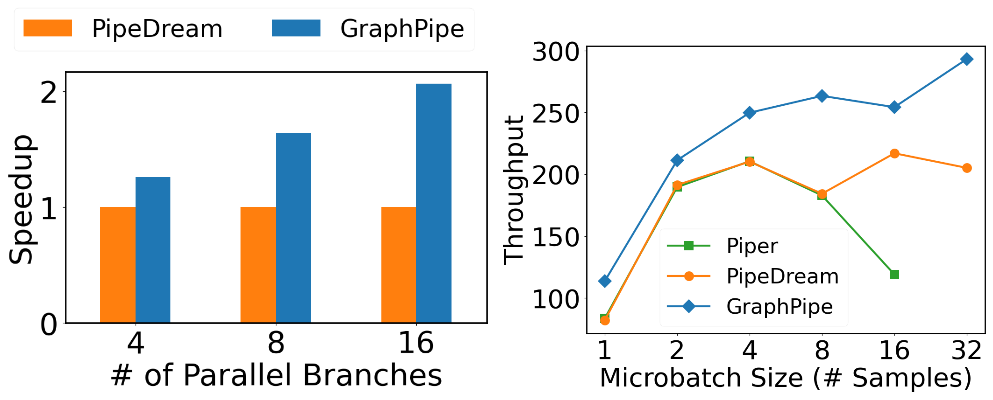


Figure 3.7: Throughput vs. different numbers of branches using 4, 8, 16 GPUs respectively (left). Throughput vs. different micro-batch sizes using 8 GPUs (right).

solving a single long chain of nk operators as in PipeDream, GRAPHPIPE solves n short chains of k operators separately. As empirically shown in Figure 3.6, 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 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 GRAPHPIPE scales with the number of branches, reaching up to 2× at 16 branches. Intuitively, the performance gain mostly stems from the fact that 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 GRAPHPIPE and SPP. This experiment result demonstrates that (1) reducing pipeline depth is critical to throughput performance; and (2) 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 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.

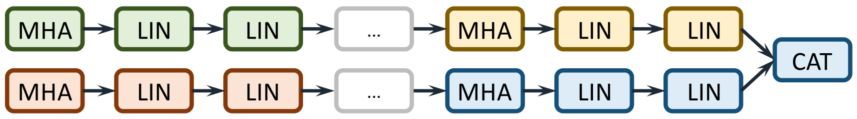


Figure 3.8: A synthetic Transformer-based two-branch DNN for case study. A sequence of one multi-head attention and two linear layers is repeated four times to compose a single branch. One concatenation layer at the end combines the branches.

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

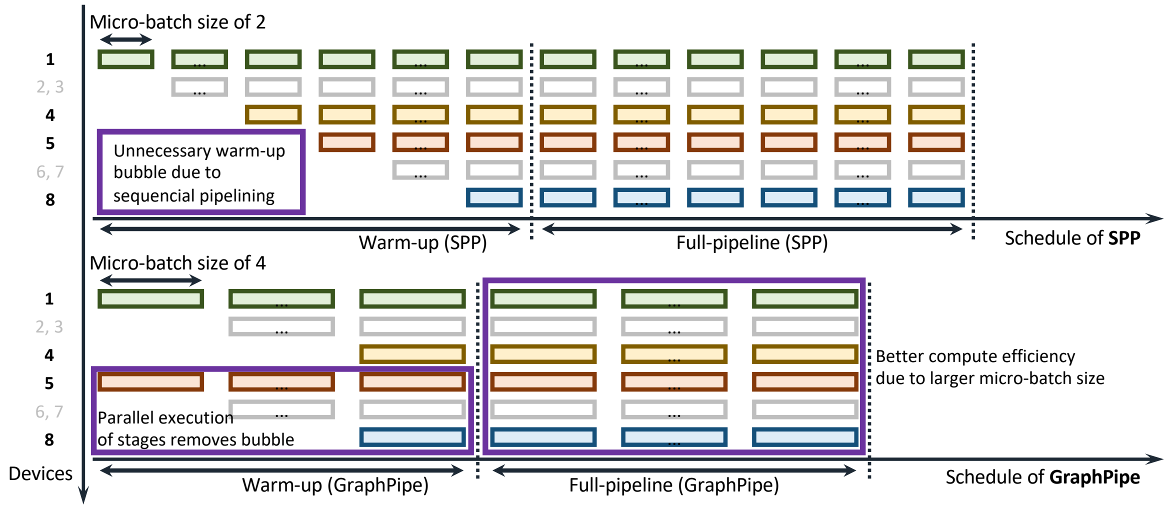

Micro-batch size of 2
1
2, 3
4
5
6, 7
8
Unnecessary warm-up bubble due to sequencial pipelining
Warm-up (SPP)
Full-pipeline (SPP)
Schedule of SPP
Micro-batch size of 4
Better compute efficiency due to larger micro-batch size
Parallel execution of stages removes bubble
Devices
Warm-up (GraphPipe)
Full-pipeline (GraphPipe)
Schedule of GraphPipe

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.

devices. Each stage contains one multi-head attention and two linear layers. There are eight such stages, four per branch, except that one stage necessarily contains the concatenation operator. A key difference between the two strategies, however, is the way the stages are pipelined. Figure 3.9 depicts the pipeline schedules. Note that the pipeline depth for SPP is eight since all eight stages form a sequential pipeline. In stark contrast, the pipeline depth for GRAPHPIPE is four. The two branches are computationally-independent, hence stage $1 + i$ and $5 + i$ for $0 \leq i \leq 3$ can be executed in parallel, and this is precisely what the training strategy produced by GRAPHPIPE suggests. This concurrent execution reduces the warm-up phase by half in terms of number of micro-batches from eight to four. This warm-up phase reduction leads to 10% performance improvement.

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, JJN+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$\times$ higher training throughputs and $> 9\times$ 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 Π_i that minimizes the running time for stage i per training iteration and also the peak memory usage for stage i. It iteratively augments Π_i by adding either a forward pass or a backward pass. The final Π_i is a sequence of forward and backward passes that constitutes a time- and memory-usage-optimal schedule. Function IsTimeOptimal checks if the micro-batch schedule leads to the minimum running time for stage i. In the following proof, we use *time-optimal* to describe a micro-batch schedule that minimizes the running time for a stage and use *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 Π_i

```
1: $\Pi_i \leftarrow \mathsf{fw}^i_1$
2: for $l \leftarrow 1 \ldots 2B/b_i - 1$ do
3:     $c_f \leftarrow$ number of $\mathsf{fw}^i$'s in $\Pi_i$
4:     $c_b \leftarrow$ number of $\mathsf{bw}^i$'s in $\Pi_i$
5:     if IsTimeOptimal($\Pi_i \mathsf{bw}_{c_b+1} \mathsf{fw}_{c_f+1} \mathsf{bw}_{c_b+2}$
                 $\ldots \mathsf{fw}_{B/b_i} \mathsf{bw}_{B/b_i+c_b-c_f+1} \ldots \mathsf{bw}_{B/b_i}$, $\{\Pi_j, b_j\}$)
       then
6:         $\Pi_i \leftarrow \Pi_i \mathsf{bw}_{c_b+1}$
7:     else
8:         $\Pi_i \leftarrow \Pi_i \mathsf{fw}_{c_f+1}$
```

Definition 1.: A micro-batch schedule Π_i for stage i is *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 Π_i.

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

Proof. Let $\{\Pi_i\}$ be an optimal micro-batch schedule. If Π_i is monotonous for each stage i, the proof is complete. If not, let i_0 be the first stage for which Π_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 Π_g starts with, and construct Π'_{i_0} by scheduling $\mathsf{fw}^{i_0}_{l_{i_0}+1}$ in Π_{i_0} right after $\mathsf{fw}^{i_0}_{l_{i_0}}$. For each predecessor stage p of i_0, as Π_p is monotonous, $b_p l_p = \text{inflight}(\Pi_p) > \text{inflight}(\Pi_{i_0}) = \text{inflight}(\Pi'_{i_0}) \geq b_{i_0}(l_{i_0}+1)$ holds, and so $\{\Pi'_i\}$ does not increase the running time for stage i. We repeat the above procedure until schedules of all stages are monotonous. Π'_{i_0} increases the number of contiguous forward micro-batches compared to Π_{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 Π_i starts with $\mathrm{inflight}(\Pi_i)/b_i$ forward micro-batches (i.e., monotonous) for each stage i. This completes the proof by construction. ■

Theorem 3.1.:
The greedy scheduler gives an optimal micro-batch schedule.

Proof. Let $\mathsf{ind}_{\Pi_i}(\mathsf{fw}_k^i)$ be the index of fw_k^i in Π_i. Let s be any time-optimal schedule for stage i that starts with $\mathrm{inflight}(s)/b_i$ contiguous forward micro-batches. Let $s = r\mathsf{bw}_{c_b+1}t$ 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, $s' = r\mathsf{bw}_{c_b+1}\mathsf{fw}_{c_f+1}\mathsf{bw}_{c_b+2}\mathsf{fw}_{c_f+2}\dots\mathsf{fw}_{B/m_i}\dots\mathsf{bw}_{B/m_i}$ must be a time-optimal schedule, because s' employs 1F1B schedule: $\mathsf{ind}_{s'}(\mathsf{fw}_k) \leq \mathsf{ind}_s(\mathsf{fw}_k)$ and $\mathsf{ind}_{s'}(\mathsf{bw}_k) \geq \mathsf{ind}_s(\mathsf{bw}_k)$ for any k. Together with Lemma 3.1, we know that the greedy scheduler produces a schedule that minimizes the number of in-flight micro-batches for stage i.

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

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

3.10.2 kFkB Schedule

The kFkB schedule of stage S_x is determined by

$$\mathrm{argmin}_{k_x} \max_{(S_x,S_y)\in\mathcal{V}_S} \mathrm{ComputeInFlight}(k_x, b_x, k_y, b_y, i_y),$$

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

3.10.3 DNN Model Configurations

The Multi-Modal Transformer-based model (MMT) for which we evaluate GraphPipe consists of four parallel branches concatenated at the end and each branch consists of

Table 3.2: Computation of the number of in-flight samples.

Condition	Result
$\max\{b_x, b_y\} < k_x b_x < k_y b_y$	$i_y + 2\max\{b_x, b_y\}$
$\max\{b_x, b_y\} = k_x b_x < k_y b_y$	$i_y + \max\{b_x, b_y\}$
$b_x \le b_y < k_y b_y < k_x b_x$	$i_y + k_x b_x - k_y b_y + 2b_y$
$b_x \le b_y = k_y b_y < k_x b_x$	$i_y + k_x b_x$
$b_y \le b_x < k_y b_y < k_x b_x$	$i_y + k_x b_x - k_y b_y + 2b_x$
$b_y \le b_x = k_y b_y < k_x b_x$	$i_y + k_x b_x$
$\max\{b_x, b_y\} = k_y b_y = k_x b_x$	$i_y + k_y b_y$
$\max\{b_x, b_y\} < k_y b_y = k_x b_x$	$i_y + 2\max\{b_x, b_y\}$
$b_x \le k_x b_x < b_y \le k_y b_y$	$i_y + b_y$
$b_y \le k_y b_y < b_x \le k_x b_x$	$i_y + k_x b_x - k_y b_y + b_x$

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:

# Devices	MMT	DLRM	CANDLE-Uno
4	64	256	4096
8	128	512	8192
16	256	1024	16384
32	512	2048	32768

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, Transformers 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 lengthens. 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 at 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 middel (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.

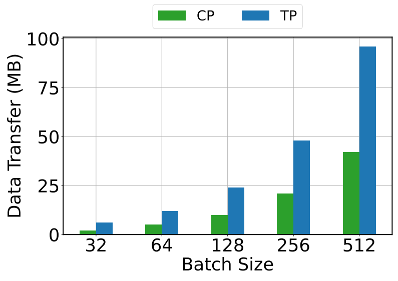
With CP, the attention module requires two steps: 1) local attention compute (for local sharded KV cache) and 2) reduce attention results from all KV cache across devices. This mechanism is equal to the FlashDecode [Dao23], but across multiple devices instead of multiple GPU warps. Note that this is 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 $BHL(\frac{E}{H})$. Let the index of L dimension be i and the index of $\frac{E}{H}$ dimension be j. For each request and attention head, we need to store three result tensors from local attention to perform reduce across devices:

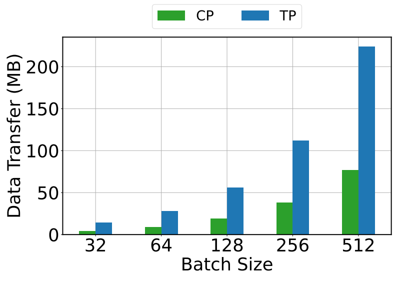
$$\begin{aligned} m_j &= \max_i Q[i,:]K^T[:,:] \\ s_j &= \sum_{i=1}^{n} e^{Q[i,:]K^T[:,:]-m_j} \\ o_{ij} &= \text{softmax}(QK^T)V \end{aligned}$$

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,

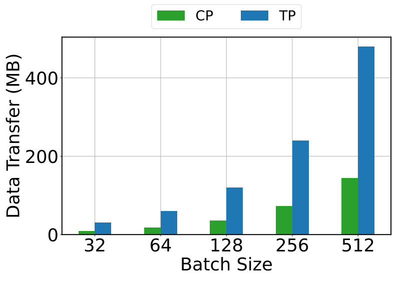
[1]Note that there is a concurrent work of ours [LPZ+24].



(a) 4 GPUs



(b) 8 GPUs



(c) 16 GPUs

Figure 4.1: Total amount of data transfer per GPU for TP and CP

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 incurs the following communication volume per GPU:

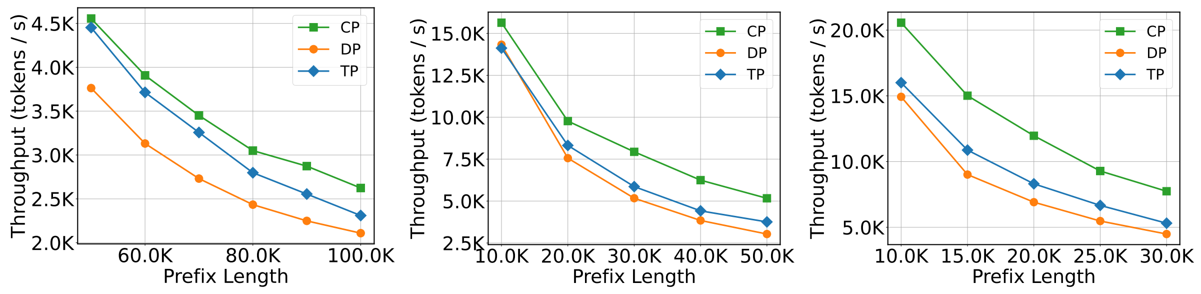
$$\text{(2 all-reduce)} = 4(N-1)BE$$

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?



(a) Low Latency (b) Medium Latency (c) High Latency

Figure 4.2: Throughput (tokens per second per layer) of varying parallelisms across different prefix sequence length.

4.5.1 Evaluation Setup

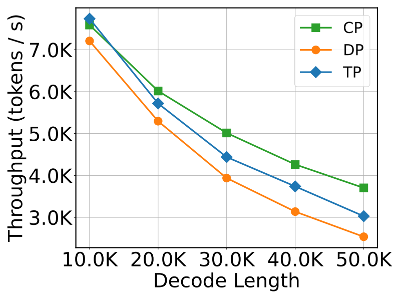
For compartive analysis, we evaluate three different parallelisms for Transformer layer: Cache Parallelism (CP), Data Parallelism (DP), and Tensor Parallelism (TP). Given that cache parallelism is specifically relevant to the attention operator with KV cache at decode step, we opt for data parallelism 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 paralleisms 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] confiuguration 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.

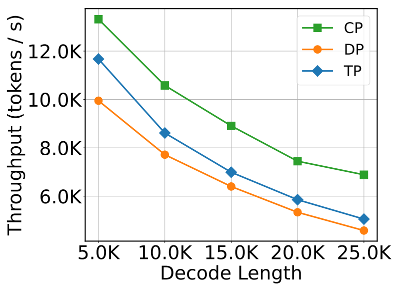
4.5.2 Scaling with Sequence Length

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.

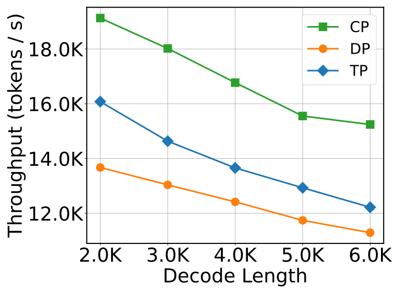
Regardless of maximum batch sizes, the general trend is that the throughput of cache



(a) Low Latency

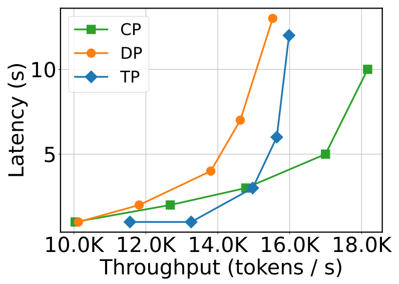


(b) Medium Latency

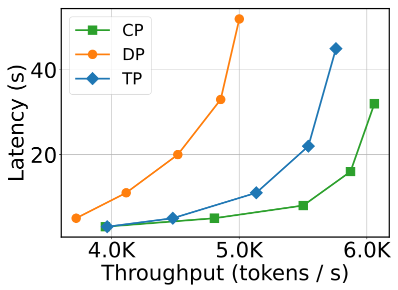


(c) High Latency

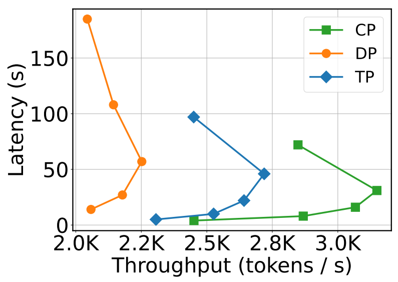
Figure 4.3: Throughput (tokens per second per layer) of varying parallelisms across different decode sequence length.



(a) Shorter Sequence (10K)



(b) Medium Sequence (40K)



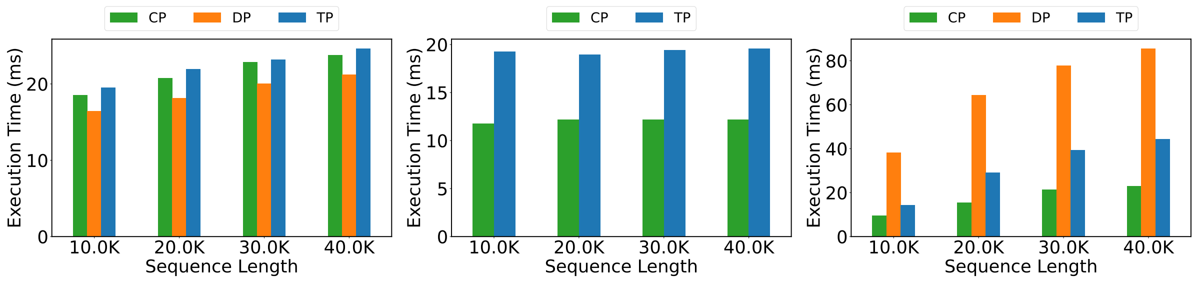
(c) Longer Sequence (80K)

Figure 4.4: Throughput and latency trade-offs of varying paralleisms across different sequence length.

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



(a) GPU Compute (b) GPU Communication (c) CPU Overhead

Figure 4.5: Breakdown of execution time into three different categories: GPU Compute, CPU Overhead, and GPU (to GPU) Communication.

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-

Seq [WXW+21]. Many of such systems focus on scaling to gigantic Transformer models [CND+22, 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 [SPP+19] suggests a simple, yet effective model parallelism method for a Transformer layer to minimize communication. Further research [PDC+23] have investigated multi-dimensional sharding strategies for Transformer inference and their tradeoffs between latency and model FLOPS utilization on TPU-v4 [JKL+23].

Efficient Transformer Inference. Researchers have witnessed that high FLOP count and communication volume fundamentally limits Transformer inference performance [PDC+23, IDBN+21]. To combat this issue, numerous efforts [GA22, GKD+21] have been made in investigating techniques such as model design[SMM+17, LLX+20, KHB+21, FDZ22, Sha19], sparsity [JCM+21, FA23, HABN+21], quantization [ZYYH18, ZBIW19, AWA+21, DLBZ22], offloading [SZY+23, WYZ+18, HJL20, RRA+21, RRR+21], etc. Several studies have explored efficient attention [DFE+22, CGRS19, SGBJ19, KKL20, CLD+20, RSVG21] and new architectures such as mixture of expert [SMM+17] and multi-query attention [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

[201] https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/Uno. Accessed: 2023-05-15. xii, 30, 42, 43

[ABC$^+$16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016. 2, 6, 18, 24, 25, 48

[AKV$^+$14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014. 24

[AMA$^+$19] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize Halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, 2019. 6, 24

[ANE] Apple neural engine (ane). https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/. Accessed: 2021-08-25. 6

[ARZ$^+$22] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv preprint arXiv:2207.00032*, 2022. 61

[ATB$^+$15] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices*, 50(8):173–182, 2015. 25

[AVG+18] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Efficient progressive device placement optimization. In *NIPS Machine Learning for Systems Workshop*, 2018. 25

[AWA+21] AmirAli Abdolrashidi, Lisa Wang, Shivani Agrawal, Jonathan Malmaud, Oleg Rybakov, Chas Leichner, and Lukasz Lew. Pareto-optimal quantized resnet is mostly 4-bit. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3091–3099, 2021. 62

[AXW+19] Amirali Abdolrashidi, Qiumin Xu, Shibo Wang, Sudip Roy, and Yanqi Zhou. Learning to fuse. In *NeurIPS ML for Systems Workshop*, 2019. 25

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. 28, 62

[BRH+18] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V Evfimievski, and Prithviraj Sen. On optimizing operator fusion plans for large-scale machine learning in systemml. *arXiv preprint arXiv:1801.00829*, 2018. 25

[BRR+19] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019. 6, 24

[BZR+22] Cedric Bastoul, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Adilla Susungi, Javier de Juan, Etienne Filhol, Baptiste Jarry, Gianpietro Consolaro, and Renwei Zhang. Optimizing gpu deep learning operators with polyhedral scheduling constraint injection. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 313–324. IEEE, 2022. 6, 24

[CBB+18] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018. 2, 6, 24

[CCC+19] Anupama Chandrasekhar, Gang Chen, Po-Yu Chen, Wei-Yu Chen, Junjie Gu, Peng Guo, Shruthi Hebbur Prasanna Kumar, Guei-Yuan Lueh, Pankaj Mistry, Wei Pan, et al. Igc: The open source intel graphics compiler. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 254–265. IEEE, 2019. 6, 24

[CGG+20] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. Automatic generation of multi-objective polyhedral compiler transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 83–96, 2020. 6, 24

[CGRS19] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019. 62

[CLD+20] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020. 62

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018. 2, 6, 10, 12, 18, 24, 25

[CND+22] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022. 62

[CSO+20] Esha Choukse, Michael B Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W Keckler. Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 926–939. IEEE, 2020. 49

[cuB] Nvidia cublas. `https://developer.nvidia.com/cublas`. Accessed: 2021-08-05. 10, 18, 24

[CWV+14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014. 1, 2, 6, 10, 12, 18, 24, 25

[CXZG16] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. 49

[CZY+18] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*, 2018. 24

[Dao23] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023. 57, 58

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 18

[DCM$^+$12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012. 2, 28

[DFE$^+$22] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022. 62

[DLBZ22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022. 62

[DLP] Dlpack: Open in memory tensor structure. `https://github.com/dmlc/dlpack`. Accessed: 2022-04-05. 18

[DSC$^+$16] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*, pages 2024–2033. PMLR, 2016. 25

[Eff] Effectivetransformer. `https://github.com/bytedance/effective_transformer`. Accessed: 2023-06-13. 61

[ELB$^+$17] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. Spoof: Sum-product optimization and operator fusion for large-scale machine learning. In *CIDR*, 2017. 2, 6, 25

[FA23] Elias Frantar and Dan Alistarh. Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*, 2023. 62

[Fas] Fastertransformer. `https://github.com/NVIDIA/FasterTransformer`. Accessed: 2023-06-13. 61

[FCGM21] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems*, 3, 2021. 24, 25

[FDRG$^+$12] Félix-Antoine Fortin, Franccois-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175, 2012. 10, 17

[FDZ22] William Fedus, Jeff Dean, and Barret Zoph. A review of sparse expert models in deep learning. *arXiv preprint arXiv:2209.01667*, 2022. 62

[FRM+21] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021. 2, 3, 28, 41, 43, 48

[FSWC20] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. Optimizing dnn computation graph using graph substitutions. *Proceedings of the VLDB Endowment*, 13(12):2734–2746, 2020. 25

[GA22] Manish Gupta and Puneet Agrawal. Compression of deep learning models for text: A survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 16(4):1–55, 2022. 62

[GCL18] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*, pages 1676–1684. PMLR, 2018. 25

[GDG+17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017. 2, 48

[GKD+21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021. 62

[GPAM+20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020. 18

[HABN+21] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *The Journal of Machine Learning Research*, 22(1):10882–11005, 2021. 62

[HCB+19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019. 2, 3, 28, 43, 48

[Hea] Will Douglas Heaven. Gpt-4 is bigger and better than chatgpt—but openai won't say why. `https://www.technologyreview.com/2023/03/14/1069823`. Accessed: 2023-05-15. 28

[HFA] Hugging face accelerate. `https://huggingface.co/docs/accelerate/index`. Accessed: 2023-06-13. 61

[HJL20] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020. 49, 62

[HKS18] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Can spatiotemporal 3d cnns retrace the history of 2d cnns and imagenet? In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6546–6555, 2018. 18

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 18

[IDBN$^+$21] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021. 62

[JAA$^+$20] Arpan Jain, Ammar Ahmad Awan, Asmaa M. Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G. Anthony, Hari Subramoni, Dhabaleswar K. Panda, Raghu Machiraju, and Anil Parwani. GEMS: gpu-enabled memory-aware model-parallelism system for distributed DNN training. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 45. IEEE/ACM, 2020. 48

[JCM$^+$21] Sebastian Jaszczur, Aakanksha Chowdhery, Afroz Mohiuddin, Lukasz Kaiser, Wojciech Gajewski, Henryk Michalewski, and Jonni Kanerva. Sparse is enough in scaling transformers. *Advances in Neural Information Processing Systems*, 34:9895–9907, 2021. 62

[JDL21] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. Deepcuts: a deep learning optimization framework for versatile gpu workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 190–205, 2021. 24, 25

[JJN$^+$20] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020. 49

[JKC$^+$21] Geonhwa Jeong, Gokcen Kestor, Prasanth Chatarasi, Angshuman Parashar, Po-An Tsai, Sivasankaran Rajamanickam, Roberto Gioiosa, and Tushar Krishna. Union: A unified hw-sw co-design ecosystem in mlir for evaluating tensor operations on spatial accelerators. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 30–44. IEEE, 2021. 24

[JKL$^+$23] Norman P Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. *arXiv preprint arXiv:2304.01433*, 2023. 62

[JLQA18] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *ICML*, pages 2279–2288, 2018. 25

[JPT$^+$19] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019. 13, 25

[JTW$^+$19] Zhihao Jia, James Thomas, Tod Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. *SysML 2019*, 2019. 25

[JYP$^+$17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017. 6, 24

[JYX$^+$21] Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc Le, Yun-Hsuan Sung, Zhen Li, and Tom Duerig. Scaling up visual and vision-language representation learning with noisy text supervision. In *International conference on machine learning*, pages 4904–4916. PMLR, 2021. 30, 42

[JZA19a] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019. 25

[JZA19b] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019. 42, 48, 49

[KFT$^+$19] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, et al. Miopen: An open source library for deep learning primitives. *arXiv preprint arXiv:1910.00078*, 2019. 1, 2, 6, 24

[KHB$^+$21] Sneha Kudugunta, Yanping Huang, Ankur Bapna, Maxim Krikun, Dmitry Lepikhin, Minh-Thang Luong, and Orhan Firat. Beyond distillation: Task-level mixture-of-experts for efficient inference. *arXiv preprint arXiv:2110.03742*, 2021. 62

[KKC$^+$17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017. 6, 24

[KKL20] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020. 62

[KLZ$^+$23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023. 59, 60

[KPB19] Samuel Kaufman, Phitchaya Mangpo Phothilimthana, and Mike Burrows. Learned tpu cost model for xla tensor programs. In *Proc. Workshop ML Syst. NeurIPS*, pages 1–6, 2019. 24

[Kri14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014. 2, 48

[KTW16] Harlan M Krumholz, Sharon F Terry, and Joanne Waldstreicher. Data acquisition, curation, and use for a continuously learning health system. *Jama*, 316(16):1669–1670, 2016. 28

[LAB$^+$21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021. 24

[LAP$^+$14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*, pages 583–598, 2014. 2, 48

[LCC$^+$21] Guei-Yuan Lueh, Kaiyu Chen, Gang Chen, Joel Fuentes, Wei-Yu Chen, Fangwen Fu, Hong Jiang, Hongzheng Li, and Daniel Rhee. C-for-metal: high performance simd programming on intel gpus. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 289–300. IEEE, 2021. 6, 24

[LCZY23] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. *CoRR*, abs/2308.15762, 2023. 48

[LH21] Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. 48

[LLX⁺20] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020. 2, 28, 62

[LP22] Joel Lamy-Poirier. Breadth-first pipeline parallelism. *arXiv preprint arXiv:2211.05953*, 2022. 48

[LPZ⁺24] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024. 57

[LZPL22] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for gpu kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–27. IEEE, 2022. 25

[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

[MHH⁺21] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, et al. High-performance, distributed training of large-scale deep learning recommendation models. *arXiv preprint arXiv:2104.05158*, 2021. 2, 48

[MPL⁺17] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017. 25, 49

[MXY⁺20] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020. 2, 6, 25

[MYS20] Linjian Ma, Jiayu Ye, and Edgar Solomonik. Autohoot: Automatic high-order optimization for tensors. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 125–137, 2020. 6, 24

[NGW+21] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021. 2, 6, 10, 12, 25

[NHP+19] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019. 2, 3, 25, 28, 43, 48, 49

[NMS+19] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019. xii, 28, 30, 42, 43

[NPS+21] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021. xii, 2, 3, 28, 30, 43, 48

[NSC+21] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021. 2, 28, 48

[NVD] Nvidia deep learning accelerator (nvdla). `http://nvdla.org/`. Accessed: 2021-08-25. 6

[One] Intel onednn. `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onednn.html`. Accessed: 2021-09-27. 1, 6, 18, 24

[opea] Ai and compute. `https://openai.com/research/ai-and-compute`. Accessed: 2023-05-15. 28

[Opeb] Intel openvino. `https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html`. Accessed: 2021-09-27. 6, 24

[Ope23] OpenAI. Gpt-4 technical report, 2023. 28, 30, 42, 62

[PDC+23] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean.

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

[PGM$^+$19a] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019. 2, 6, 18, 24, 48

[PGM$^+$19b] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 59

[PGN$^+$20] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. In *International Conference on Learning Representations*, 2020. 25

[PSD$^+$20] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020. 49

[PSS$^+$21] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, et al. A flexible approach to autotuning multi-pass machine learning compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–16. IEEE, 2021. 6, 24

[QCS$^+$21] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021. 2

[RFA$^+$18] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018. 2, 6, 24

[RGC$^+$16] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016. 49

[RKBA$^+$13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013. 2, 6, 24

[RKH$^+$21] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021. xii, 30, 42, 43

[RLW$^+$18] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 58–68, 2018. 10

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015. 18

[RPG$^+$21] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR, 2021. 30, 42

[RRA$^+$21] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021. 49, 62

[RRR$^+$21] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021. 62

[RRRH20] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020. 48

[RSG19] Ari Rasch, Richard Schulze, and Sergei Gorlatch. Generating portable high-performance code via multi-dimensional homomorphisms. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 354–369. IEEE, 2019. 6, 24

[RSVG21] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021. 62

[SCP$^+$18] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018. 2, 28, 48

[SCSZ19] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019. 25

[SGBJ19] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. *arXiv preprint arXiv:1905.07799*, 2019. 62

[Sha19] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019. 62

[SMM$^+$17] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017. 62

[SPP$^+$19] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. 2, 28, 49, 56, 62

[SRSA21] Omais Shafi, Chinmay Rai, Rijurekha Sen, and Gayathri Ananthanarayanan. Demystifying tensorrt: Characterizing neural network inference engine on nvidia edge devices. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 226–237, 2021. 6

[sum] Summit supercomputer. `https://www.olcf.ornl.gov/summit/`. Accessed: 2023-09-06. 42

[SUV20] Annamalai Suresh, R Udendhran, and S Vimal. *Deep neural networks for multimodal imaging and biomedical applications*. IGI Global, 2020. 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, et al. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023. 62

[TBT+16] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 209–223, 2016. 24

[Tena] Nvidia tensorrt. https://developer.nvidia.com/tensorrt. Accessed: 2021-08-05. 1, 2, 6, 10, 12, 18, 24, 25

[Tenb] Nvidia tensorrt deserialization. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html. Accessed: 2022-06-09. 18

[TLI+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. 59

[TMS+23] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. 59

[TNP21] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34:24829–24840, 2021. xii, 2, 3, 28, 30, 43, 44, 48, 49

[TNS82] Kazuhiko Takamizawa, Takao Nishizeki, and Nobuji Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the ACM (JACM)*, 29(3):623–641, 1982. 37

[TPD+20] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators. *Advances in Neural Information Processing Systems*, 33:15451–15463, 2020. 25

[TTP+20] Wei Tan, Prayag Tiwari, Hari Mohan Pandey, Catarina Moreira, and Amit Kumar Jaiswal. Multimodal medical image fusion algorithm in the era of big data. *Neural Computing and Applications*, pages 1–21, 2020. 28

[UJW+22] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating dnn training through joint optimization of algebraic transformations and parallelization.

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

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. 2, 48

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 30, 42

[VZT+18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018. 24

[WCQ+23] Xiao Wang, Guangyao Chen, Guangwu Qian, Pengcheng Gao, Xiao-Yong Wei, Yaowei Wang, Yonghong Tian, and Wen Gao. Large-scale multi-modal pre-trained models: A comprehensive survey. *Machine Intelligence Research*, pages 1–36, 2023. 30, 42

[WCS+23] Siyu Wang, Zongyan Cao, Chang Si, Lansong Diao, Jiamang Wang, and Wei Lin. Ada-grouper: Accelerating pipeline parallelism in preempted network by adaptive group-scheduling for micro-batches. *arXiv preprint arXiv:2303.01675*, 2023. 40

[WHL19] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019. 49

[WXW+21] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. Light-Seq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers (NAACL-HLT)*, pages 113–120. Association for Computational Linguistics, June 2021. 62

[WYZ+18] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018. 49, 62

[WZS+14] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014. 1, 2, 6, 10, 18

[XGD$^+$17] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017. 18

[XLA] Tensorflow xla. `https://www.tensorflow.org/xla`. Accessed: 2021-09-27. 2, 6, 18, 24, 25

[YJK$^+$22] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022. 56, 61

[YPW$^+$21] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization, 2021. 25

[YZZ$^+$22] 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

[ZBIW19] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE, 2019. 62

[ZJS$^+$20] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. 6, 24

[ZLW$^+$20] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020. 24

[ZLZ$^+$22] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022. 2, 25, 28, 43, 48, 49

[ZRA$^+$19] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint arXiv:1910.01578*, 2019. 25

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

[ZYS+19] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *arXiv preprint arXiv:1903.06631*, 2019. 49

[ZYYH18] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018. 62

[ZZL+20] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. Fusionstitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924*, 2020. 2, 6, 25