Ph.D. DISSERTATION

Efficient Execution of Machine Learning Workloads on GPUs

GPU 환경에서 머신러닝 워크로드의 효율적인 실행

FEBRUARY 2023

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Gyeong-In Yu

Efficient Execution of Machine Learning Workloads on GPUs

GPU 환경에서 머신러닝 워크로드의 효율적인 실행

지도교수 전 병 곤

이 논문을 공학박사 학위논문으로 제출함

2022 년 11 월

서울대학교 대학원

컴퓨터 공학부

유 경 인

유경인의 공학박사 학위논문을 인준함

2022 년 12 월

Abstract

Machine learning (ML) workloads are becoming increasingly important in many types of real-world applications. We attribute this trend to the development of software systems for ML, which have facilitated the widespread adoption of heterogeneous accelerators such as GPUs. Today's ML software stack has made great improvements in terms of efficiency, however, not all use cases are well supported. In this dissertation, we study how to improve execution efficiency of ML workloads on GPUs from a software system perspective. We identify workloads where current systems for ML have inefficiencies in utilizing GPUs and devise new system techniques that handle those workloads efficiently.

We first present Nimble, a ML execution engine equipped with carefully optimized GPU scheduling. The proposed scheduling techniques can be used to improve execution efficiency by up to 22.34×. Second, we propose Orca, an inference serving system specialized for Transformer-based generative models. By incorporating new scheduling and batching techniques, Orca significantly outperforms state-of-the-art systems – 36.9× throughput improvement at the same level of latency. The last topic of this dissertation is WindTunnel, a framework that translates classical ML pipelines into neural networks, providing GPU training capabilities for classical ML workloads. WindTunnel also allows joint training of pipeline components via backpropagation, resulting in improved accuracy over the original pipeline and neural network baselines.

Keywords: machine learning, deep learning, scheduling, inference serving, generative models, Transformer, joint training

Student Number: 2017-28658

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the recent decade, Machine Learning (ML) has led advances in many different applications [55, 138, 146, 43, 27, 22, 28, 119, 51, 66]. This success does not come for free; breakthroughs are often achieved by leveraging more computation [27, 37, 71], hence machine learning models are becoming larger and more computationally intensive than ever. Such trend has facilitated rapid adoption of heterogeneous processors, including GPUs, FPGAs, and custom ASICs like TPUs, along with advances in software systems for controlling those processors. Since building an efficient software for the processors like GPUs from scratch requires a high degree of expertise in the hardware, these software systems have been playing a crucial role of providing optimized runtime for machine learning workloads.

Today's ML software stack has made significant strides in terms of execution efficiency, but not all use cases are well supported. This is because ML workloads have highly diverse characteristics, depending on the types of tasks (e.g., vision, language, or advertisement), problem complexity (e.g., ranging from small decision trees to terabyte-scale Transformer models), and whether to train new models or serve already trained models. Each of these workloads also requires different performance optimizations for different hardware architectures. For example, existing systems cannot efficiently handle serving workloads on GPUs that run Transformer-based [146] generative models, even considering specialized systems for Transformer models [7, 5, 3].

1.2 Dissertation Overview

In this dissertation, we introduce ideas to improve execution efficiency of ML workloads on GPUs from a software system perspective. We by no means provide a complete, automated methodology that outperforms state-of-the-art systems for a variety of workloads. Instead, our primary focus is on identifying overlooked cases and designing efficient systems for those cases.

We hypothesize that careful scheduling optimization and graph transformation can make ML workloads on GPUs more efficient. To demonstrate this, we examine three ML workloads where current ML systems cannot utilize GPUs efficiently, devise novel system techniques for optimizing those workloads, and implement systems by incorporating the techniques.

Lightweight and Parallel GPU Scheduling We investigate how general-purpose ML frameworks (e.g., PyTorch [104] and TensorFlow [17]) schedule computations on GPUs. We point out two problems in current scheduling that can significantly inhibit efficient use of GPUs: large scheduling overhead and unnecessary serial execution of GPU computations. While these problems can be hidden when each computation is sufficiently large enough, we argue that this is not true in many workloads such as training of relatively small models or inference serving. To this end, we propose Nimble, a ML execution engine that employs two new techniques – ahead-of-time scheduling and automatic multi-stream execution. Ahead-of-time scheduling removes most of the scheduling overhead during run time by reusing the work done for scheduling. Multi-stream execution analyzes target model's computation graph, finds an optimal operator-to-stream mapping, and embeds the mapping in the graph by graph rewriting. Experiments show that Nimble speeds up inference and training by up to 22.34× and 3.61× compared to PyTorch, respectively. Since Nimble is implemented on top of PyTorch, its shares the same GPU kernel implementation with PyTorch; that is, the speedup comes from the advancements in scheduling mechanisms. Nimble even outperforms state-of-the-art inference systems, TensorRT [9] and TVM [33], by up to 2.81× and 1.70×, respectively.

Serving System for Transformer-Based Generative Models Next, we dive into a more specific, but important workload: inference serving of Transformer-based generative models like GPT-3 [27]. In this workload, to serve a single inference request from a client, one should run the model as many times as the number of tokens to generate. Such multi-iteration characteristic, which does not appear in other types of tasks like image recognition or text classification, calls into question current serving systems' [11, 101] design: they schedule the execution of engine at the granularity of batch of requests. This can greatly limit the execution efficiency because a request finished earlier than others in a batch have to wait for processing of the current batch before returning to its client. Similarly, a request arrived after dispatching a batch also have to wait for the batch before being processed. In this work, we propose iteration-level scheduling where the serving system schedules a single iteration of the model at a time. To maintain the batching functionality – which is crucial for efficiency – while using the proposed scheduling, we suggest to apply batching to a selected set of operators that comprise Transformer. Based on these techniques, we implement a distributed serving system named ORCA, which can scale to models with hundreds of billions of parameters. ORCA outperforms NVIDIA FasterTransformer [7] by a significant margin: 36.9× throughput improvement at the same level of latency.

Neural Translation of Classical ML Pipelines Many real-world ML workloads including click prediction [51] and recommendation [28] often use non-DNN models, which we call *classical* ML models. These workloads compose pipelines of data transformations and models to express the entire logic of feature extraction and learning. We observe that systems for classical ML (e.g., scikit-learn [106] and ML.NET [21]) mostly focus on CPU environment except for very few cases [2]. In this work, we enable GPU execution of classical ML training by translating classical ML pipelines into neural networks. That is, we can leverage existing software stack mainly developed for neural networks to train classical models on GPUs. More importantly, our translation approach unlocks gradient backpropagation capability for classical ML pipelines. To this end, we propose WINDTUNNEL, a framework that jointly optimizes components of classical ML pipelines by neural translation and backpropagation. We suggest translation methods for two most popular non-differentiable operators – gradient boosting trees and categorical feature encoders. Our experiments show that we can achieve better accuracy by joint training, thereby closing the gap between classical ML and neural networks.

1.3 Previous Publications

This dissertation contains material from the following previous publications:

- Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning, NeurIPS 2020. [84]
- Orca: A Distributed Serving System for Transformer-Based Generative Models, OSDI 2022. [156]
- WindTunnel: Towards Differentiable ML Pipelines beyond a Single Model, VLDB 2022. [155]
- Making Classical Machine Learning Pipelines Differentiable: A Neural Translation Approach, Workshop on Systems for ML at NeurIPS 2018. [154]

1.4 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 gives background on ML workloads and system stack for ML. Chapter 3 presents Nimble, a deep learning execution engine that improves efficiency by optimizing GPU scheduling. In Chapter 4, we introduce ORCA, a distributed serving system for Transformer-based generative models. Chapter 5 explains WINDTUNNEL and how it realizes end-to-end training of classical ML pipelines on GPUs. Chapter 6 discuss related work of proposed systems and Chapter 7 concludes this dissertation.

Chapter 2

Background

In this chapter, we provide background on machine learning workloads and how existing ML systems handle the workloads.

2.1 ML Workloads

Machine learning is a paradigm of artificial intelligence that leverages data to automatically improve the performance of computer programs. Broadly speaking, the life cycle of machine learning can be divided into two stages: training and inference. During the training, ML developers feed a training dataset to their model and let the model "learn" to solve a given task, e.g., identify whether a given image is a cat or a dog. In particular, a training process incorporates an iterative procedure that fits the model for a given training dataset. Such iterative procedure is determined by the training objective and algorithm. This process is often very computationally intensive; training models like GPT-3 [27] can take several months even with a thousand of GPUs.

After the training process, we deploy the trained model and use it to solve certain tasks – we call this phase inference (or inference serving). There are some characteristics that distinguish inference from training. First, unlike training, inference does not require complex training algorithms as it does not make changes on the model. Second, inference is often in the critical path of user-facing interface. For example, an email service provider filters spam in real-time by taking inference results from a trained model. Moreover, while we use a finite, carefully curated dataset as input of a training process, inference workloads should handle random client requests arriving in real-time to the deployed environment. In Section 2.4, we provide more details on the system stack for inference serving.

In addition to the training and inference, in practice, machine learning life cycle includes other steps such as data collection, model testing, and monitoring. We do not discuss these steps as they are outside the scope of this dissertation.

2.2 The GPU Execution Model

GPUs provide high throughput for ML computation due to their capability to run thousands of threads in parallel. Such high degree of parallelism comes from tens of parallel computation units called Streaming Multiprocessor (SM), which comprises hundreds of cores.[1] When a program (which runs on a CPU) submits a GPU computation named kernel, which is a grid of multiple blocks, GPU driver dispatches the blocks to SMs that have enough hardware resource (e.g., number of vacant registers) to run the block. A single SM can run multiple blocks concurrently, while a block is the minimum unit of work that can be dispatched to a SM thus cannot be distributed across multiple SMs. A SM

[1]Without losing generality, we use NVIDIA terminology throughout this dissertation for ease of understanding.

subdivides each assigned block into groups of parallel threads called a warp, and selects one (or multiple) warp eligible to issue its next instruction at every cycle. Unlike CPUs, the context (e.g., registers) of all warps on a SM is kept on exclusively dedicated hardware resource – SMs can switch from one warp to another at no cost. Therefore, to fully utilize the computation power of a SM, it requires enough number of assigned warps so that at every cycle there exists at least one warp ready to execute its next instruction.

One way to achieve this is to submit a large GPU kernel that has enough intra-kernel parallelism (i.e., number of threads and blocks). Unfortunately, this is not always possible because the number of threads (or blocks) is often limited by various factors, including the implementation of the kernel and the size of the data (i.e., tensor) being processed. Another way to enhance the GPU utilization is to schedule multiple GPU kernels to run in parallel using multiple GPU streams. A GPU stream is a queue of GPU kernels where the kernels are scheduled sequentially in FIFO order. While kernels on the same stream cannot be executed concurrently, kernels on different streams can run in parallel, occupying different parts of the GPU resources. The execution order between kernels on different streams is not guaranteed unless explicitly specified by stream synchronization primitives [1].

2.3 GPU Scheduling in ML Frameworks

Machine learning systems handle GPU intricacies such as copying neural network weights to GPUs and launching ML operators on GPUs. Operators indicate numerical computations, such as matrix multiplication, convolution, softmax and layer normalization, and consist of one or more *GPU tasks*, including not only GPU kernels but also GPU memory operations (e.g., `memcpy`).

The GPU task scheduling mechanisms of existing general-purpose ML frame-

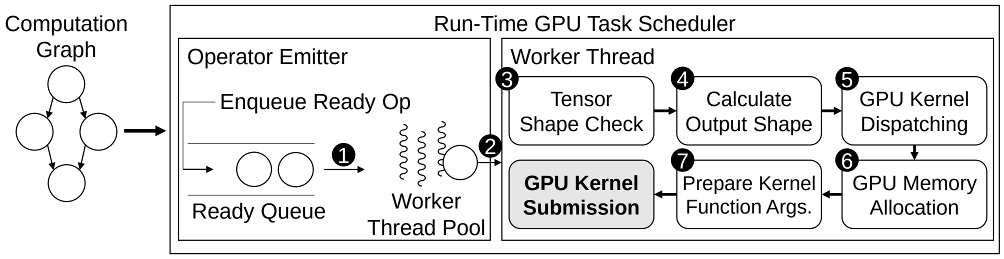


Figure 2.1: GPU task scheduling in ML frameworks that build a computation graph for ML execution.

works are largely divided into two categories.[2] First, ML frameworks including TensorFlow [17], Caffe2 [64] and TorchScript [16] express a ML model as a computation graph where each node represents a ML operator and each edge indicates a dependency between two operators. The runtime stack of such a ML framework consists of two major system components (written in C++): the operator emitter and the workers. The operator emitter maintains a queue of operators whose dependencies are met and emits the operator at the front of the queue to a worker thread. The worker takes the emitted operator and performs a series of preparation steps and finally submits GPU kernels for each operator. As such, ML frameworks in this category *schedule* the GPU tasks at run time through the interplay of the operator emitter and the workers.

Second, ML frameworks including PyTorch [104] and TensorFlow Eager [20] describe a ML model as an imperative Python program. In such ML frameworks, there is no explicit computation graph of the model nor operator emitter in the runtime stack. That is, the operators are emitted by the Python interpreter as the program is executed line by line. The emitted operators are then processed by the worker in a similar manner to the ML frameworks in the first category. As such, ML frameworks in the second category also perform the run-time

[2]Here we do not take frameworks without GPU support into account (e.g., scikit-learn [106].)

scheduling of GPU tasks, through the Python interpreter and the worker.

Figure 2.1 illustrates in detail how ML frameworks such as TensorFlow and Caffe2 carry out run-time scheduling. To submit a GPU task, the run-time scheduler must go through the following process: ❶ select an operator from the ready queue; ❷ emit the operator to a vacant worker thread; ❸ check the types and shapes of input tensors; ❹ calculate the types and shapes of output tensors; ❺ dispatch appropriate GPU kernels for the operator based on tensor types and shapes; ❻ allocate GPU memory for the output tensors and workspace for the kernels, typically by retrieving memory blocks from the cached pool of GPU memory; and ❼ prepare function arguments required for submitting the kernels. While specific steps may differ across ML frameworks, the overall process remains the same.

Regarding the GPU streams, existing ML frameworks are primarily designed and optimized to submit GPU kernels to a single GPU stream. For example, TensorFlow uses a single *compute stream* per GPU for running its kernels.

2.4 Engine Scheduling in Inference Servers

Growing demands for ML-driven applications have made ML inference serving a critical workload in modern datacenters. Users (either the end-user or internal microservices of the application) submit requests to an inference service, and the service gives replies on the requests based on a pre-defined ML model using its provisioned resource, typically equipped with specialized accelerators such as GPUs and TPUs. In particular, the service runs a ML model with input data to generate output for the request. Just like other services operating on datacenters, a well-managed inference service should provide low latency and high throughput within a reasonable amount of cost.

To meet such constraints, service operators often use ML *inference servers* such as Triton Inference Server [11] and TensorFlow Serving [101]. These systems can be seen as an abstraction sitting atop underlying model *execution engines* such as TensorRT [9], TVM [33], TensorFlow [17], and many others [104, 106], being agnostic to various kinds of ML models, execution engines, and computing hardware. While delegating the role of driving the main mathematical operations to the engines, inference servers are in charge of exposing endpoints that receive inference requests, scheduling executions of the engine, and sending responses to the requests. Accordingly, these systems focus on aspects such as batching the executions [38, 11, 101, 130, 85], selecting an appropriate model from multiple model variants [38, 70, 59, 131], deploying multiple models (each for different inference services) on the same device [130, 85, 65, 11], and so on.

Among the features and optimizations provided by ML serving systems, batching is a key to achieve high accelerator utilization when using accelerators like GPUs. When we run the execution engine with batching enabled, the input tensors from multiple requests coalesce into a single, large input tensor before being fed to the first operation of the model. Since the accelerators prefer large input tensors over small ones to better exploit the vast amount of parallel computation units, the engine's throughput is highly dependent on the batch size, i.e., the number of inference requests the engine processes together. Reusing the model parameters loaded from off-chip memory is another merit in batched execution, especially when the model involves memory-intensive operations.

2.5 Inference Procedure of Generative Models

We provide background on the inference procedure of GPT [113, 27], a representative example of Transformer-based generative models that we use throughout this dissertation. GPT is an autoregressive language model based on one of

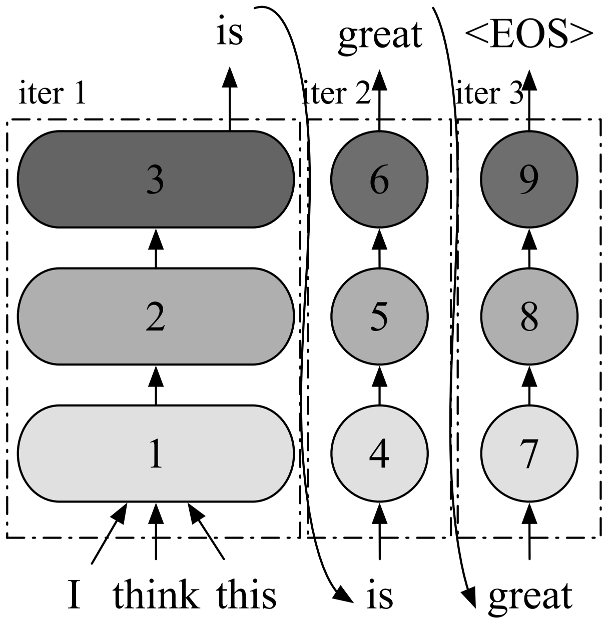


Figure 2.2: A computation graph representing an inference procedure using a GPT model. The graph does not depict layers other than Transformer layers (e.g., embedding) for simplicity.

architectural variants of Transformer [146]. It takes text as input and produces new text as output. In particular, the model receives a sequence of input tokens and then completes the sequence by generating subsequent output tokens. Figure 2.2 illustrates a simplified computation graph that represents this procedure with a three-layer GPT model, where nodes and edges indicate Transformer layers and dependencies between the layers, respectively. The Transformer layers are executed in the order denoted by the numbers on the nodes, and the nodes that use the same set of model parameters (i.e., nodes representing the same layer) are filled with the same color.

The generated output token is fed back into the model to generate the next output token, imposing a sequential, one-by-one inference procedure. This autoregressive procedure of generating a single token is done by running all the

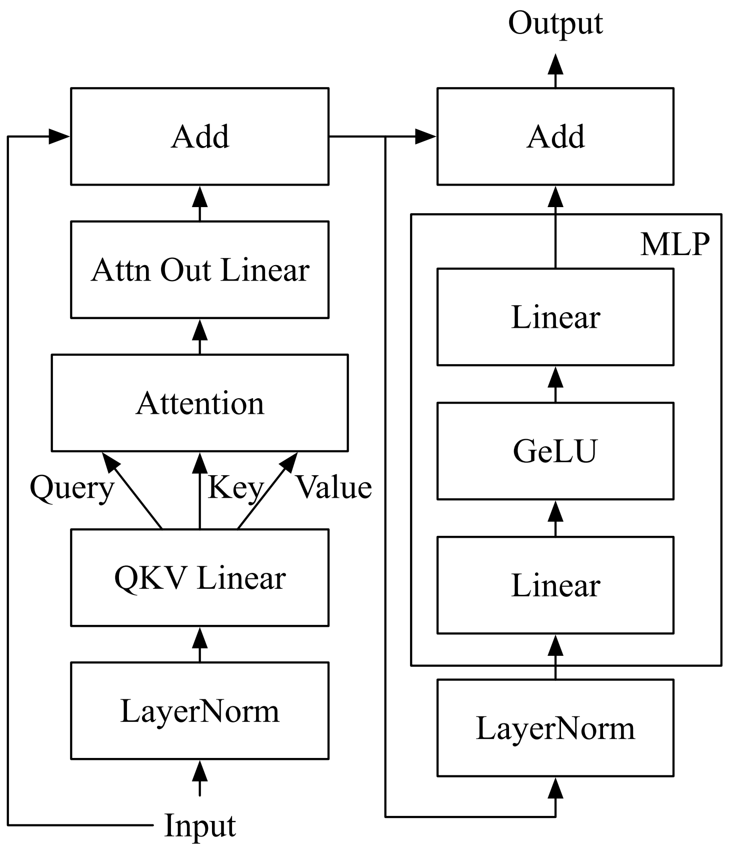


Figure 2.3: A Transformer layer used in GPT.

layers of the model with the input, which is either a sequence of input tokens that came from the client or a previously generated output token. We define the run of all layers as an *iteration* of the model. In the example shown in Figure 2.2, the inference procedure comprises three iterations. The first iteration ("iter 1") takes all the input tokens ("I think this") at once and generates the next token ("is"). This iteration composes an *initiation phase, a procedure responsible for processing the input tokens and generating the first output token.* The next two iterations ("iter 2" and "iter 3"), which compose an *increment phase*, take the output token of the preceding iteration and generate the next token. In this

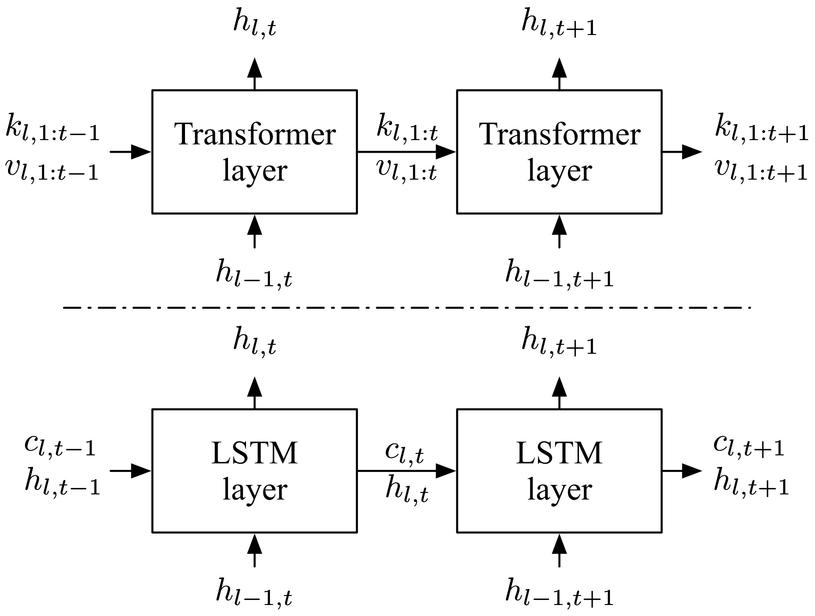


Figure 2.4: Internal state usage of Transformer. h, k, v, and c refer to layer input/output, Attention key, Attention value, and LSTM internal memory, respectively. l denotes layer index and t denotes token index.

case, "iter 3" is the last iteration because it produces "<EOS>", a special end-of-sequence token that terminates output generation. Note that while the increment phase comprises multiple iterations because each iteration is only able to process a single token, the initiation phase is typically implemented as a single iteration by processing all the input tokens in parallel.

The original Transformer [146] employs two stacks of Transformer layers, while GPT's architecture consists of a single layer stack, namely decoder. Figure 2.3 shows a Transformer layer used in GPT. Among the operations that compose the Transformer layer, *Attention* is the essence that distinguishes Transformer from other architectures. At a high level, the Attention operation computes a weighted average of the tokens of interest so that each token in the sequence is aware of the other. It takes three inputs, query, key, and value, computes dot products of the query (for the current token) with all keys (for the tokens of interest), applies Softmax on the dot products to get weights, and conducts weighted average of all values associated with the weights.

Since the Attention requires keys and values of all preceding tokens,[3] we consider the keys and values as internal states that should be maintained across multiple iterations. A naïve, state-less inference procedure would take all tokens in the sequence (including both the client-provided input tokens and the output tokens generated so far) to recompute all the keys and values at every iteration. To avoid such recomputation, fairseq [103] suggests incremental decoding, which saves the keys and values for reuse in successive iterations. Other ML systems specialized for Transformer such as FasterTransformer [7] and Megatron-LM [5] also do the same.

Figure 2.4 illustrates the state usage pattern of Transformer, along with LSTM [57] that also maintains internal states. The main difference is that the size of the states (k for Attention key and v for value) in Transformer increases with iteration, whereas the size of the states (c for LSTM internal memory and h for LSTM layer's input/output) in LSTM remains constant. When processing the token at index t, the Attention operation takes all previous Attention keys $k_{l,1:t-1}$ and values $v_{l,1:t-1}$ along with the current key $k_{l,t}$ and value $v_{l,t}$.[4] Therefore, the Attention operation should perform computation on tensors of different shapes depending on the number of tokens already processed.

Prior to the Attention operation, there are the layer normalization operation (LayerNorm) and the QKV Linear (linear and split operations to get the query, key and value). Operations performed after Attention are, in order, a linear

[3]Language models like GPT use causal masking, which means all preceding tokens are of interest and participate in the Attention operation.

[4]$k_{l,1:t-1}$ represents Attention keys of the l-th layer for tokens at indices 1 to $t-1$ while $k_{l,t}$ is for the Attention key of the l-th layer for the token at index t. Same for $v_{l,1:t-1}$ and $v_{l,t}$.

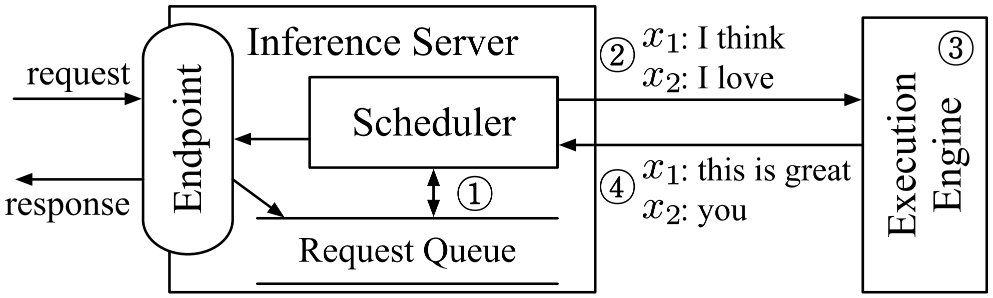


Figure 2.5: Overall workflow of serving a generative language model with existing serving systems.

operation (Attn Out Linear), an add operation for residual connection (Add), layer normalization operation (LayerNorm), the multi-layer perceptron (MLP) operations, and the other residual connection operation (Add).

Figure 2.5 shows an overall workflow of serving a generative language model with existing inference serving systems. As discussed in Section 2.4, an inference serving system comprise an inference server and an execution engine. The main component of the inference server (e.g., Triton [11]) is the scheduler, which is responsible for ① creating a batch of requests by retrieving requests from a queue and ② scheduling the execution engine (e.g., FasterTransformer [7]) to process the batch. The execution engine ③ processes the received batch by running multiple iterations of the model being served and ④ returns the generated text back to the inference server. In Figure 2.5, the server schedules the engine to process two requests (x_1: "I think", x_2: "I love") in a batch and the engine generates "this is great" and "you" for requests x_1 and x_2, respectively.

Chapter 3

Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning

3.1 Introduction

In recent years, growing demands for deep learning (DL) have facilitated the advance of DL frameworks such as Caffe2 [64], MXNet [32], PyTorch [104], and TensorFlow [17]. These frameworks provide implementations of GPU-based neural network computations along with high-level APIs, with which users can express the semantics of neural networks as usual Python programs. Furthermore, such frameworks allow users to describe the training and inference procedure of their networks without the need to control GPUs directly. DL frameworks then automatically handle GPU intricacies such as copying neural network weights to GPUs and launching DL operators on GPUs. Operators indicate numerical computations, like convolution and batch normalization, and consist of one or more *GPU tasks* (i.e., GPU kernels and GPU memory operations).

Before a GPU processes a task, DL frameworks must first go through a series of preparation steps (*GPU task scheduling*), and then submit the task to the GPU (*GPU task submission*). We note that current DL frameworks conduct GPU task scheduling during *run time*. For instance, TensorFlow, Caffe2, and MXNet represent a neural network as a computation graph of DL operators, and schedule the GPU tasks of an operator at run time once the operator's dependencies are met. Meanwhile, for PyTorch and TensorFlow Eager [20], GPU tasks are scheduled at run time as Python code is interpreted line by line.

While under ideal circumstances the running time of neural networks mostly depends on the amount of computation assigned to GPUs, in reality we find otherwise. We point out two important problems in run-time task scheduling that may significantly limit framework performance. First, the time spent on scheduling, which we call *scheduling overhead*, can take a substantial portion of the overall running time. Although the scheduling overhead is negligible when the running time of a GPU task is sufficiently long enough to hide the overhead, we find that this does not hold in many cases, especially when inference and training of a neural network consist of small and short GPU tasks. Modern GPUs [6, 10] have thousands of computation units along with specialized processors like Tensor Core [13], and use high bandwidth memory [36] to avoid bottlenecks from memory bandwidth. While the time spent on running GPU tasks can dramatically be reduced by such GPUs, we observe that the scheduling overhead is constantly imposed by every GPU task, and often dominates the running time of DL inference and training.

Another problem DL frameworks face is that serial execution of GPU tasks misses the opportunity to further improve performance by parallelizing task execution. Recent neural networks exhibit inter-operator level parallelism. For example, topologies of the neural networks obtained by neural architecture search (NAS) [29, 30, 90, 109, 118, 160] are directed acyclic graphs (DAGs) with multiple branches rather than linear chains. In addition, recent works have proposed new types of layers that consist of smaller operators arranged in parallel, such as MixConv [139] and Split-Attention [157] blocks. Leveraging inter-operator parallelism can lead to performance improvements in executing such neural networks, especially in the case of inference. However, existing DL frameworks [17, 32, 104] are designed and optimized to schedule GPU tasks to be executed one at a time, and thus hardly exploit inter-operator parallelism.

To address the above limitations, we present Nimble, a new DL execution engine that schedules GPU tasks to run in parallel with minimal scheduling overhead. The key observation that drives the design of Nimble is that for static neural networks the behavior of a network is predetermined by its architecture. For both inference and training, DL frameworks run the exact same computation graph with the same shapes of inputs over and over again. Thus, we can leverage detailed information about the computation graph and the input shape to optimize the scheduling of GPU tasks.

To avoid the scheduling overhead, Nimble introduces a novel *ahead-of-time (AoT) scheduling* technique. Nimble schedules GPU tasks for a given neural network execution ahead of time; later when Nimble is given an input, Nimble skips scheduling and proceeds immediately to task submission. Since the preparation steps of GPU tasks are invariant to each neural network execution (i.e., independent of the input values), we only need to perform task scheduling once. While Nimble's AoT scheduler performs GPU task scheduling, it records a trace of GPU tasks and GPU memory requests, and generates a *task schedule*. The task schedule contains all information and resources (i.e., result of the scheduling) required for the execution of the neural network, including the submission order between GPU tasks, function arguments for the GPU tasks, and how to run GPU tasks in parallel. At run time, Nimble substitutes the high-overhead scheduling procedure by the raw submission of GPU tasks based on the task schedule, dramatically reducing the scheduling overhead.

To execute multiple GPU tasks in parallel on a GPU, Nimble employs *automatic multi-stream execution*. Although the CUDA programming interface provides Stream API for concurrent kernel execution [1], assigning neural network operators to appropriate streams is a difficult task for users. Nimble automates the stream assignment and synchronization process. Before AoT scheduling, Nimble analyzes dependency relationships between operators and finds an optimal stream assignment that guarantees the smallest number of synchronizations across streams while parallelizing as many operators as possible. Given the operator-to-stream mapping, Nimble rewrites the computation graph of the given neural network to run the GPU tasks of the operators on their corresponding streams with proper synchronizations. The modified graph is then used as an input to the AoT scheduler, which in turn embeds the information about the stream mapping and synchronization in the task schedule.

Nimble is built on top of PyTorch and supports both inference and training of neural networks. Users can seamlessly apply Nimble to their PyTorch programs by wrapping DL model instances in Nimble objects. Our evaluation on a variety of deep neural networks shows that Nimble improves the speed of inference and training by up to 22.34× and 3.61× compared to PyTorch, respectively. Moreover, Nimble outperforms state-of-the-art inference systems, TensorRT [9] and TVM [33], by up to 2.81× and 1.70×, respectively. Nimble is publicly available at `https://github.com/snuspl/nimble`. While we implement and evaluate Nimble only for NVIDIA's server-class GPUs, Nimble's main idea can also be adopted to other hardware such as mobile devices.

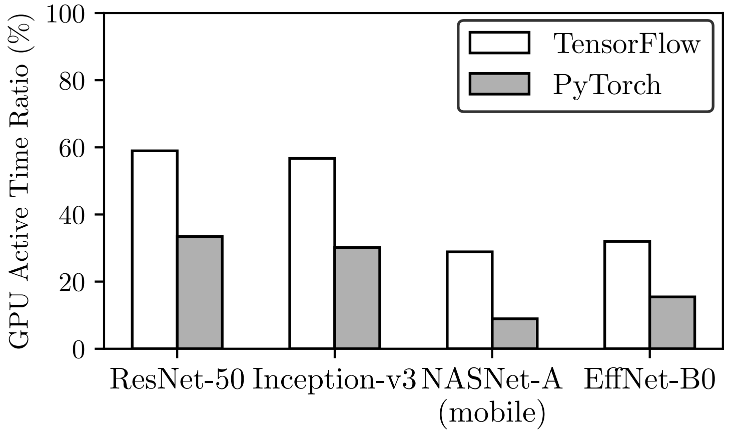


Figure 3.1: Ratio of GPU active time to the overall running time on DL inference.

3.2 Motivation

In this section we present experiments describing the problems in GPU task scheduling of current DL frameworks. The experiments are conducted on TensorFlow [17] and PyTorch [104], the two most popular DL frameworks. The experiment setting is the same as that of the evaluation in Section 3.4.

High Scheduling Overhead Makes GPUs Idle We experimentally demonstrate that the run-time scheduling often incurs prohibitive amount of scheduling overhead such that GPU idle time dominates overall running time of DL execution. Figure 3.1 shows the ratios of the *GPU active time*, sum of the time intervals during which GPU is not idle, to the overall running time spent on the inference of the neural networks [55, 137, 138, 160] with batch size 1. In the result, both TensorFlow and PyTorch leave their GPUs idle for a substantial portion of the running time, up to 71% and 91%, respectively. To be specific, the scheduling overhead of submitting one GPU task in PyTorch ranges from tens of microseconds to one or two hundreds of microseconds, while the time spent for running the submitted GPU task can be less than ten microseconds in inference workloads. While the inefficiency in PyTorch can be partially attributed to the slowness of Python interpreter, the high overhead in TensorFlow implies that the major source of the performance bottleneck lies in the core runtime stack of the framework, and that the overhead remains significant even if the runtime is written in low-overhead language such as C++.

To further support our idea, we measure the performance of a DL framework when its scheduling procedure is minimized. For the experiment, we write a C++ program that can only perform the inference of the specific neural networks [55, 137] with a fixed input shape and uses the same GPU kernels and memory operations as PyTorch. From the assumptions that the given neural network is static and the shape of its input tensor is fixed, we prune away any redundant routines that can be done ahead of the run time. For example, shape check is omitted and the shapes of the output tensors are hardcoded in the program since every shape information can be inferred ahead of time based on the neural network architecture and the predetermined input shape. In this way, the program directly submits the GPU kernels at run time without going through the PyTorch's runtime stack for dispatching them. Likewise, GPU memory allocation is skipped and the tasks reuse fixed, pre-allocated memory regions for every iteration whose addresses are also hardcoded in the program.

Figure 3.2 shows the impact of such optimizations on the scheduling procedure. Despite the fact that exactly the same set of GPU kernels are computed, PyTorch and its scheduling-minimized version present remarkably different inference latencies: 2.37× speedup is obtained in ResNet-50 by the simple minimization of the scheduling procedure. The result confirms that the main source of the GPU idle time is the overhead of the scheduling procedure described in Section 2.3. Greater performance gain is expected in those neural networks

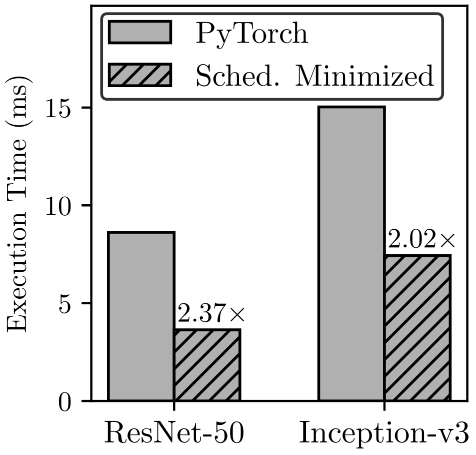


Figure 3.2: Inference latencies of PyTorch and its scheduling-minimized version.

with lower GPU active time ratio (e.g., EfficientNet-B0).

Non-Parallel GPU Task Execution Framework performance can be further improved by parallelizing GPU tasks. Figure 3.3 shows the ratios of *critical path time* to the GPU active time in the inference of the neural networks [90, 118, 137, 160] with batch size 1. The critical path time is sum of the GPU active times spent on the operators in the longest path (in terms of time) of the computation graph. The result implies that inference latency can be reduced by up to 3× when the GPU tasks are fully parallelized and executed on a sufficiently powerful GPU (i.e., a GPU that can compute every concurrent kernel simultaneously).

In spite of the potential performance gain, existing DL frameworks do not effectively support the use of multiple GPU streams. One major obstacle we found is that the high scheduling overhead significantly decreases the chance that GPU tasks on different streams are computed in parallel. For example, Figure 3.4 illustrates the timeline where GPU tasks A and B are scheduled in different streams. Contrary to the expectation that the two tasks are processed

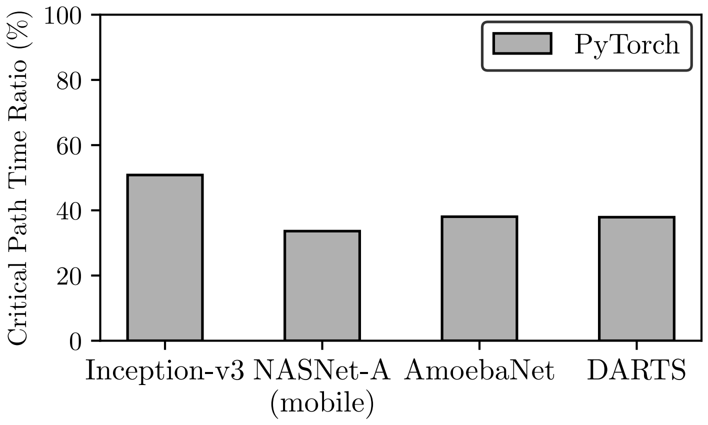


Figure 3.3: Ratio of critical path time to the GPU active time on DL inference.

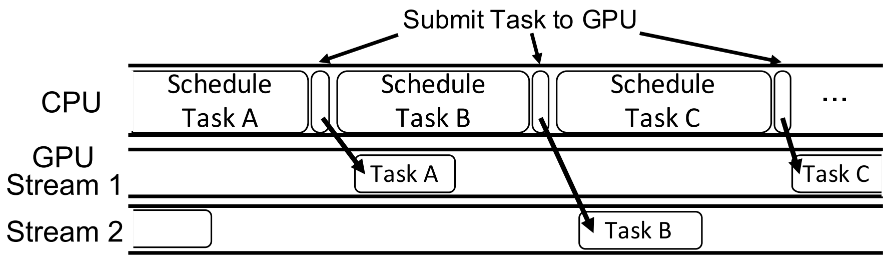


Figure 3.4: High scheduling overhead inhibits efficient use of multiple GPU streams.

at the same time, the scheduling overhead creates a gap between the start time of the two tasks, which is longer than the duration of GPU task A. As a result, the GPU ends up executing the tasks one at a time.

3.3 System Design

Motivated by the observations in Section 3.2, we present Nimble, a DL execution engine to *automatically* avoid the scheduling overhead of DL frameworks and parallelize GPU tasks using multiple GPU streams. Nimble takes a DL framework as its base system, and resolves the inefficiencies in GPU task scheduling without redesigning the framework runtime. In the current implementation,

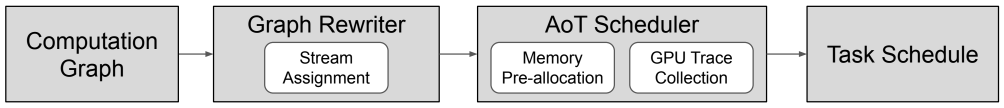


Figure 3.5: System overview of Nimble.

Nimble is built on top of PyTorch, but the system design is applicable to other DL frameworks.

Figure 3.5 summarizes execution steps in Nimble. The system consists of Graph Rewriter and AoT Scheduler. Nimble first takes as input a computation graph of a neural network. The computation graph is represented as a TorchScript [16] graph in PyTorch. The graph rewriter analyzes the computation graph and constructs an operator-to-stream mapping by the algorithm we present in Section 3.3.2. It marks each operator with the stream that the operator will be issued on and embeds synchronization routines to the graph by using custom nodes we add. The AoT scheduler of Nimble then goes through a series of preparation steps for the execution of the GPU tasks ahead of time. During the process, the scheduler collects a *GPU execution trace* and reserves GPU memory used in executing the GPU tasks. Finally, Nimble packs the GPU trace and the reserved memory into a task schedule. At run time, the recorded GPU tasks are replayed on the basis of the task schedule for every DL execution.

3.3.1 Ahead-of-time (AoT) Scheduling

The AoT scheduler aims to generate a task schedule, finishing the scheduling procedure required for submitting GPU tasks ahead of time. Our observation is that we can move the GPU task scheduling outside the run time execution loop without changing the semantics of neural network execution, similar to the loop-invariant code motion in compilers. In other words, while the existing frameworks repeat the scheduling procedure at every neural network execution, Nimble's AoT scheduler finishes the scheduling once ahead of time, providing a significant speedup in executing the neural network. This is possible because Nimble assumes a static neural network that performs the same set of computations for different runs, which means we can reuse the work done for scheduling after it is done once. However, this AoT scheduling raises two challenges: (a) how to distinguish the scheduling procedure that can be safely removed from the run time execution; and (b) how to move the scheduling procedure out of the run time execution.

We solve these challenges by approaching the problem from a direction different from typical performance bottleneck optimization. Instead of differentiating and removing the scheduling procedure from the run time execution, Nimble identifies *non*-scheduling work, i.e., the GPU tasks. That is, Nimble takes advantage of the fact that the computation of a given static neural network is fully described by a fixed set of GPU tasks, and that the scheduling procedure of DL frameworks becomes redundant once the set of the GPU tasks are determined. During the AoT scheduling, Nimble *pre-runs* the given neural network once according to the generated stream mapping, and records all the GPU tasks as an execution trace. The pre-run process is a single iteration of inference/training execution of the given neural network using the base framework of Nimble. During the pre-run process, while the scheduling procedure of the base framework is done as usual, the GPU tasks submitted from the framework are intercepted and recorded. The generated execution trace contains all the essential information resulted from the scheduling: dispatched GPU kernels, function arguments of the kernels, task submission order, task-to-stream assignment, etc. Once the pre-run process is done, Nimble can leverage the execution trace for submitting the tasks to the GPU, skipping the scheduling procedure.

To execute the collected GPU tasks, GPU memory should be allocated for

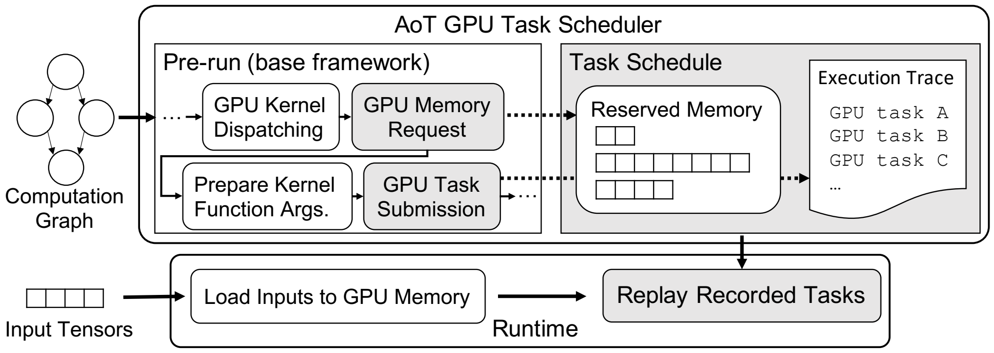


Figure 3.6: AoT GPU task scheduler and Runtime of Nimble. Dashed arrows represent the interception of GPU tasks and memory requests by the AoT scheduler.

inputs and outputs of the tasks. Since a static neural network makes the same sequence of memory requests for different runs, we can pre-allocate the exact amount of GPU memory required for its execution. For this purpose, during the process of pre-run, Nimble also intercepts memory allocate/free requests from the base framework and reserves the GPU memory allocated for the pre-run. The reserved memory is then used for the run time execution of Nimble.

At the end of the AoT scheduling, Nimble packs the execution trace and the reserved memory into a task schedule. At run time, Nimble conducts inference/training of the given neural network by directly submitting the GPU tasks recorded in the task schedule with the addresses of the reserved memory regions. In this manner, the GPU tasks can be executed independently of the base DL framework, without being tied up with the runtime and the memory allocation procedure of the base framework.

Figure 3.6 gives more details about the AoT scheduling technique. According to the stream assignment result, the AoT scheduler pre-runs the neural network once with a dummy input tensor. During the pre-run process, the scheduler intercepts invocations of GPU tasks and allocations of GPU memory, and constructs a task schedule. To be concrete, we use CUDA Stream Capture APIs for capturing information of GPU tasks issued on CUDA Streams, at the beginning and end of the pre-run. Then we instantiate a CUDA Graph [52], a feature introduced in CUDA 10, (i.e., execution trace representation in Nimble) from the captured information. At run time, when there is a request with a new input tensor, Nimble executes the neural network by replaying the recorded GPU tasks on the basis of the task schedule, avoiding the scheduling overhead. We execute the neural network by using CUDA Graph Launch APIs, which submit GPU tasks based on the information in the CUDA Graph.

3.3.2 Stream Assignment Algorithm

Nimble schedules GPU tasks to run in parallel by submitting them to multiple GPU streams in a single GPU. In this section, we describe an efficient algorithm for assigning GPU tasks to streams.

Stream Synchronization Allowing concurrency requires proper synchronizations across streams. For example, assume that two independent GPU tasks A and B are given, and that another GPU task C consumes both outputs of A and B. If the three GPU tasks are submitted to a single stream (with the order of either A→B→C or B→A→C), no synchronization is needed. In contrast, if the three GPU tasks are submitted to three different streams, we should guarantee that GPU task C begins executing only after both GPU tasks A and B are completed. In CUDA, such dependencies across different streams can be enforced by using *events*, a special type of GPU tasks that can act as barriers. In the example, a desirable way to guarantee the execution order is to create and submit an event for each stream where GPU task A or B has been launched. We then call `cudaStreamWaitEvent` for each event to block the stream of GPU task C until both events are processed, which means that the execution of GPU tasks A and B have finished. We refer to issuing an event on the stream of task X and blocking the stream of task Y as a synchronization on the edge (X, Y). We count the number of synchronizations as the number of edges where synchronizations occur.

A few DL frameworks [104, 144] have high-level APIs through which programmers can create, switch, and block the streams where GPU tasks run. Nevertheless, as we pointed out in Section 3.2, leveraging multiple streams on these frameworks rarely yields performance enhancement due to their GPU task scheduling overheads. Additionally, even when the framework users are able to take advantage of the multi-stream execution, it remains as a significant burden for the users to assign and synchronize the streams in a safe and an efficient manner. Nimble resolves these difficulties by *automatically* parallelizing the GPU tasks. The process of parallelization and synchronization is transparent to users but it gives speedup when running neural networks with parallelizable structures.

Goal of the Algorithm Given a computation graph, which is a DAG of DL operators, Nimble finds a *stream assignment*, a mapping from the node set of the computation graph to a stream set of the GPU. Nimble's stream assignment algorithm meets the following two goals:

- **Maximum logical concurrency.** Given a neural network $G = (V, E)$ and a set of streams $S = \{s_1, s_2, ..., s_n\}$, find a mapping $f : V \rightarrow S$ such that if $x, y \in V$ and there is no dependency between x and y (i.e., no established order exists between the two), then $f(x) \neq f(y)$ (i.e., the two nodes are assigned to different streams).

- **Minimum number of synchronizations.** Among such functions, find f

Algorithm 1: Nimble's stream assignment algorithm.

```
Input   A DAG $G = (V, E)$ where $V = \{v_1, v_2, ..., v_n\}$.
Output  A stream assignment $f : V \to S$.

Step 1  Obtain the minimum equivalent graph of $G$.
        We call this graph $G' = (V, E')$.
Step 2  Define a bipartite graph $B = (V_1, V_2, E_B)$ where $V_1 = \{x_1, x_2, ..., x_n\}$,
        $V_2 = \{y_1, y_2, ..., y_n\}$, and $E_B = \{(x_i, y_j) \mid (v_i, v_j) \in E'\}$.
Step 3  Find a maximum matching $M$ of the bipartite graph $B$.
Step 4  Make a collection of sets $\{\{v_1\}, \{v_2\}, ..., \{v_n\}\}$. For each $(x_i, y_j) \in M$,
        combine the two sets that $v_i$ and $v_j$ are in. The result is a partition of $V$.
Step 5  Construct $f : V \to S$ in such a way that $f(v_i) = f(v_j)$
        if and only if $v_i$ and $v_j$ are included in the same set.
```

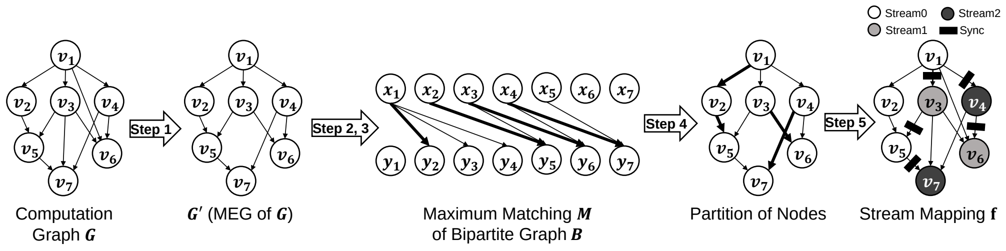


Figure 3.7: Example walk-through of Algorithm 1. Bold lines indicate edges in maximum matching M.

that incurs the smallest number of synchronizations across streams.

Maximum logical concurrency is an optimization strategy that generalizes a common practice. To increase the chance that GPU resources are fully utilized, maximizing the concurrency is desirable. In addition, the algorithm factors in the number of synchronizations needed for safe concurrent execution. Since synchronizations hamper the fast launching of tasks, the algorithm is designed to incur the theoretically smallest number of synchronizations while maintaining maximum concurrency.

Algorithm Description The stream assignment algorithm of Nimble is described in Algorithm 1. Figure 3.7 illustrates how the algorithm is applied to a computation graph G. At Step 1, we compute the *minimum equivalent graph* (MEG) G', which is a subgraph of the computation graph G with the same set of the nodes and the smallest subset of the edges that maintains the same reachability relation as G. Note that the MEG of a finite DAG is unique and can be constructed in polynomial time [60]. At Step 2 and Step 3, we define a bipartite graph B from G' and then find a *maximum matching* of B, a matching that includes the largest number of edges. A maximum matching of a bipartite graph can be computed by Ford-Fulkerson algorithm [49]. At Step 4, we first create a collection of node sets where each node in the graph G' is a separate set. Then for each edge (x_i, y_j) in M, we combine the two node sets that v_i and v_j are in. At Step 5, nodes belonging to the same set are mapped to the same stream, and nodes belonging to different sets are mapped to different streams.

We now demonstrate that the stream assignment constructed from Algorithm 1 meets the two goals by using the following theorems. Detailed proofs on the theorems are presented in Appendix A.1.

Theorem 1. *A stream assignment f satisfies maximum logical concurrency on G if and only if f satisfies maximum logical concurrency on G'. Also, for any stream assignment f that satisfies maximum logical concurrency, the minimum number of synchronizations required for f on G is equal to the minimum number of synchronizations required for f on G'.*

Theorem 2. *There exists one-to-one correspondence Φ from the set of the matchings of the bipartite graph B to the set of the stream assignments that satisfy maximum logical concurrency on G'. In fact, Φ is constructed by Step 4 and Step 5 of Algorithm 1.*

Theorem 3. *For any matching m of the bipartite graph B, the minimum number of synchronizations required for the corresponding stream assignment $\Phi(m)$ is $|E'| - |m|$.*

Theorem 4. *For a maximum matching M of the bipartite graph B, $\Phi(M)$ is a stream assignment that satisfies maximum logical concurrency and requires minimum number of synchronizations among the stream assignments satisfying maximum logical concurrency.*

Proof of Theorem 4. Based on Theorem 1, the algorithm derives the desired stream assignment from G' instead of G. From Theorem 2, it follows that $\Phi(M)$ is a stream assignment with maximum logical concurrency. Now, suppose that there exists a stream assignment $g : V \to S$ that satisfies maximum logical concurrency with strictly less number of synchronizations than that of $\Phi(M)$. By Theorem 2 and Theorem 3, g corresponds to some matching $\Phi^{-1}(g)$ of B such that $|M| < |\Phi^{-1}(g)|$. The inequality, however, is contradictory to the definition of M since M is a maximum matching of the bipartite graph B. Thus, Theorem 4 follows. $\square$

3.4 Evaluation

Experimental Setup We implement Nimble on PyTorch v1.4 with CUDA 10.2 and cuDNN 8.0.2. For evaluation, we use an NVIDIA V100 GPU along with 2.10GHz Intel Xeon CPU E5-2695 v4.

To evaluate DL inference, we compare Nimble with popular DL frameworks, PyTorch, TorchScript and Caffe2, as well as state-of-the-art inference systems, TensorRT (v7.1) [9] and TVM (v0.6.1) [33]. To evaluate DL training, Nimble is compared with PyTorch and TorchScript. Note that TensorRT and TVM employ graph optimizations (e.g., aggressive operator fusion) and kernel selection/tuning, which are orthogonol to our idea. In Nimble, we also implement the operator fusion (a subset of TensorRT's) and basic kernel selection, which chooses the faster implementation of convolution operators between cuDNN [35] and PyTorch's native implementation.

We use the implementations of the neural networks from various open-source repositories. We summarize the information below.

- torchvision repository[1]
  - ResNet-50, ResNet-101, Inception-v3, MobileNetV2
- Pretrained models for PyTorch repository[2]
  - NASNet-A (mobile), NASNet-A (large)
- PyTorch Image Models repository[3]
  - EfficientNet-B0, EfficientNet-B5
- Differentiable Architecture Search repository[4]
  - AmoebaNet, DARTS
- NVIDIA Deep Learning Examples repository[5]
  - BERT

Throughout the evaluation, TorchScript modules are created through PyTorch's tracing API. For Caffe2, TensorRT and TVM, PyTorch models are first converted into ONNX [12] models and then parsed by the respective parsers of the systems. For the evaluation on inference latency, we use synthetic 224×224 RGB images as inputs, except for Inception-v3, NASNet-A (large), and EfficientNet-B5. For these neural networks, the inputs are larger size images - 299×299 for Inception-v3, 331×331 for NASNet-A (large), and 456×456 for EfficientNet-B5 - following the description in the original literature [137, 160, 138]. For the evaluation on training, we use 224×224 RGB

[1]`https://github.com/pytorch/vision`
[2]`https://github.com/Cadene/pretrained-models.pytorch`
[3]`https://github.com/rwightman/pytorch-image-models`
[4]`https://github.com/quark0/darts`
[5]`https://github.com/NVIDIA/DeepLearningExamples`

images for the ImageNet dataset, and 32×32 RGB images for the CIFAR-10 dataset. We use a sequence length of 128 in the experiments with BERT, following the setting used for pretraining in the original literature [43].

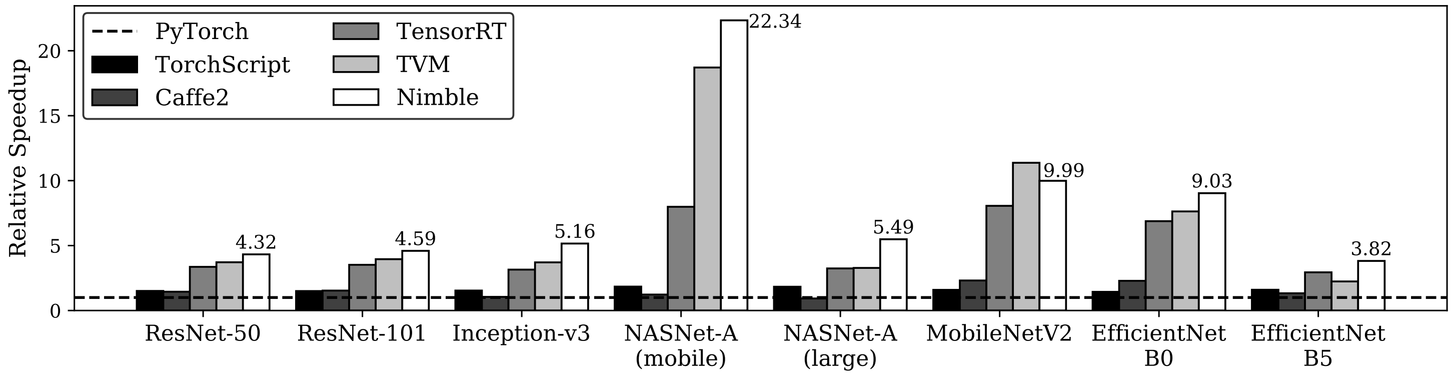


Figure 3.8: Relative inference speedup of Nimble and other systems (batch size 1). We use various neural networks [55, 125, 137, 138, 160], all trained on ImageNet [124].

Table 3.1: Impact of the multi-stream execution of Nimble on DL inference, compared to its single-stream counterpart. Deg. stands for maximum degree of logical concurrency of each architecture.

Architecture	**Speedup**	**Deg.**	**#MACs**
Inception-v3	1.09×	6	5.7B
DARTS	1.37×	7	0.5B
AmoebaNet	1.45×	11	0.5B
NASNet-A (M)	1.88×	12	0.6B
NASNet-A (L)	1.31×	15	23.9B

3.4.1 Inference Latency

Figure 3.8 presents the relative inference speed of Nimble and the other systems. We set PyTorch as the baseline. The result shows that Nimble outperforms PyTorch, TorchScript and Caffe2 significantly. The primary reason for this performance gap is the substantial scheduling overhead, which makes GPU idle for most of the time. In addition, since the DL frameworks hardly utilize parallelism among operators in a neural network, the performance gap widens in the neural networks with parallelizable structures like NASNet-A (mobile) (up to 22.34×). Nimble also shows higher performance than TensorRT on all of the tested neural networks, by up to 2.81× (NASNet-A (mobile)). Moreover, Nimble surpasses performance of TVM in most cases, by up to 1.70× (EfficientNet-B5). The only exception is MobileNetV2 [125]. TVM spends two days in kernel tuning (1500 trials for each convolution), and finds much faster GPU kernels for MobileNetV2 than those of cuDNN and PyTorch. Results on different GPUs are provided in Appendix A.2.

3.4.2 Impact of Multi-stream Execution

We select a set of deep neural networks with parallelizable structures and investigate the impact of the multi-stream execution of Nimble on the inference latency of such neural networks. Table 3.1 shows the relative speedup of the multi-stream execution compared to the single-stream execution of Nimble. The result indicates that multi-stream execution of Nimble can accelerate DL inference by up to 1.88× compared to the single-stream execution, and that Nimble exploits logical concurrency to the degree (15) that programmers cannot effectively assign and synchronize the streams manually.

In addition, we observe that the acceleration rates considerably differ across the neural networks. Neural networks with a higher degree of logical concurrency tend to benefit more from the multi-stream execution. For example, the neural network with the lowest degree of logical concurrency (Inception-v3) gains the smallest speedups. Also, we can see the trend that neural networks with less amount of computation are more likely to be accelerated by the multi-stream execution. For instance, although NASNet-A (large) exhibits higher degree of logical concurrency than NASNet-A (mobile), the former gets limited speedup compared to the latter because the former consists of kernels with a large number of multiply-and-accumulate (MAC) operations, each of which occupies most of the GPU resources. The comparison between Inception-v3 and DARTS reports the same tendency.

3.4.3 Training Throughput

Figure 3.9 shows the performance of Nimble on neural network training. Since training of a neural network is commonly conducted with large batch sizes, GPU scheduling overhead imposed during training is less pronounced, and the impact of multi-stream execution is also limited. Accordingly, in the results of ResNet on the ImageNet dataset and BERT [43], Nimble shows marginal performance improvement. However, Nimble still brings up substantial speedup when neural networks are trained with small-size inputs (e.g., low-resolution images). For

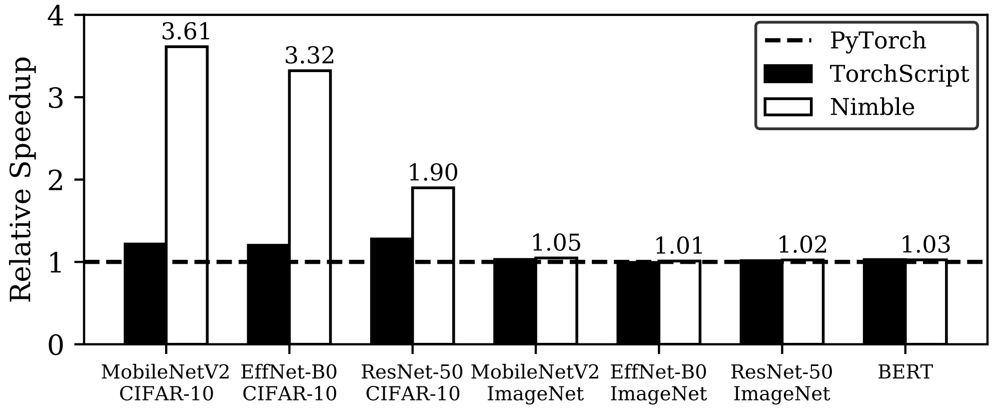


Figure 3.9: Relative training speedup of Nimble and TorchScript. All neural networks [43, 55, 125, 138] are trained with batch size 32.

example, in the field of computer vision, the CIFAR-10 [82] dataset is widely used among researchers and many neural networks are trained on the dataset. Figure 3.9 shows Nimble's performance when neural networks [55, 125, 138] are trained on CIFAR-10. The result implies that the scheduling overhead can still be a major performance bottleneck even in training. Nimble eliminates such inefficiency and increases training throughputs by up to 3.61×. Results on different batch sizes are presented in Appendix A.3.

3.5 Summary

We introduce Nimble, a high-speed DL execution engine for static neural networks. We first show two problems of the run-time scheduling of GPU tasks: scheduling overhead and serial execution. Nimble minimizes the scheduling overhead by finishing the scheduling procedure ahead of time before executing the GPU tasks at run time. Moreover, Nimble schedules independent GPU tasks to be executed in parallel, further boosting its performance. Our evaluation on various neural networks shows that Nimble outperforms popular DL frameworks and state-of-the-art inference systems. Nimble is publicly available at `https://github.com/snuspl/nimble`.

Chapter 4

Orca: A Distributed Serving System for Transformer-Based Generative Models

4.1 Introduction

Language generation tasks are becoming increasingly paramount to many types of applications, such as chatbot [122, 18], summarization [105, 127, 98], code generation [31], and caption generation [151, 153]. Moreover, recent works published by AI21 Labs [87], DeepMind [114, 58], Google [46, 149, 37], Meta Platforms [23, 158], Microsoft [116], Microsoft & NVIDIA [133], and OpenAI [27] have reported that every language processing task, including translation [26, 42], classification [126, 45], question-answering [83, 95, 74] and more, can be cast as a language generation problem and have shown great improvements along this direction. The rise of generative models is not limited to the language domain; the AI community has also given growing interest to generation problems in other domains such as image, video, speech, or a mixture of multiple domains [117, 148, 44, 89]. At the heart of generative models lies the Transformer architecture [146] and its variants [113, 115, 114, 37]. By relying on the attention mechanism [146], Transformer models can learn better representations where each element of the sequence may have a direct connection with every other element, which was not possible in recurrent models [57].

To use generative models in real-world applications, we often delegate the inference procedure to a separate service responsible for ML inference serving. The growing demands for this service, which should provide inference results for client requests at low latency and high throughput, have facilitated the development of *inference servers* such as Triton Inference Server [11] and TensorFlow Serving [101]. As discussed in Section 2.4, these systems can use a separately-developed ML *execution engine* to perform the actual tensor operations. For example, we can deploy a service for language generation tasks by using a combination of Triton and FasterTransformer [7], an execution engine optimized for the inference of Transformer-based models. In this case, Triton is mainly responsible for grouping multiple client requests into a batch, while FasterTransformer receives the batch from Triton and conducts the inference procedure in the batched manner.

Unfortunately, we notice that the existing serving systems, including both the inference server layer and the execution engine layer, have limitations in handling requests for Transformer-based generative models. Since these models are trained to generate a next token in an autoregressive manner, one should run the model as many times as the number of tokens to generate, while for other models like ResNet [55] and BERT [43] a request can be processed by running the model once. That is, in order to process a request to the generative model, we have to run multiple *iterations* of the model; each iteration generates a single output token, which is used as an input in the following iteration. Such multi-iteration characteristic calls into question the current design of serving systems, where the inference server schedules the execution of the engine at the granularity of request. Under this design, when the server dispatches a batch of requests to the engine, the engine returns inference results for the entire batch at once after processing all requests within the batch. As different client requests may require different numbers of iterations for processing, requests that have finished earlier than others in the batch cannot return to the client, resulting in an increased latency. Requests arrived after dispatching the batch also should wait for processing the batch, which can significantly increase the requests' queueing time.

In this work, we propose to schedule the execution of the engine *at the granularity of iteration* instead of request. In particular, the server invokes the engine to run only a single iteration of the model on the batch. As a result, a newly arrived request can be considered for processing after waiting for only a single iteration of the model. The inference server checks whether a request has finished processing after every return from the engine – hence the finished requests can also be returned to the clients immediately.

Nevertheless, a noticeable challenge arises when we attempt to apply batching and the iteration-level scheduling at the same time. Unlike the canonical request-level scheduling, the proposed scheduling can issue a batch of requests where each request has so far processed a different number of tokens. In such a case, the requests to the Transformer model cannot be processed in the batched manner because the attention mechanism calls for non-batchable tensor operations whose input tensors have variable shapes depending on the number of processed tokens.

To address this challenge, we suggest to apply batching only to a selected set of operations, which we call *selective batching*. By taking different characteristics of operations into account, selective batching splits the batch and processes each request individually for the Attention[1] operation while applying batching to other operations of the Transformer model. We observe that the decision not to batch the executions of the Attention operation has only a small impact on efficiency. Since the Attention operation is not associated with any model parameters, applying batching to Attention has no benefit of reducing the amount of GPU memory reads by reusing the loaded parameters across multiple requests.

Based on these techniques, we design and implement Orca, a distributed serving system for Transformer-based generative models. In order to handle large-scale models, Orca adopts parallelization strategies including intra-layer and inter-layer model parallelism, which were originally developed by training systems [132, 129] for Transformer models. We also devise a new scheduling algorithm for the proposed iteration-level scheduling, with additional considerations for memory management and pipelined execution across workers.

We evaluate Orca using OpenAI GPT-3 [27] models with various configurations, scaling up to 341B of parameters. The results show that Orca significantly outperforms FasterTransformer [7], showing 36.9× throughput improvement at the same level of latency. While we use a language model as a driving example and conduct experiments only on language models, generative models in other domains can benefit from our approach as long as the models are based on the Transformer architecture and use the autoregressive generation procedure [117, 148, 44, 89].

[1]In some literature the Attention operation has an extended definition that includes linear layers (QKV Linear and Attn Out Linear; Figure 2.3). On the other hand, we use a narrow definition as described in Figure 2.3.

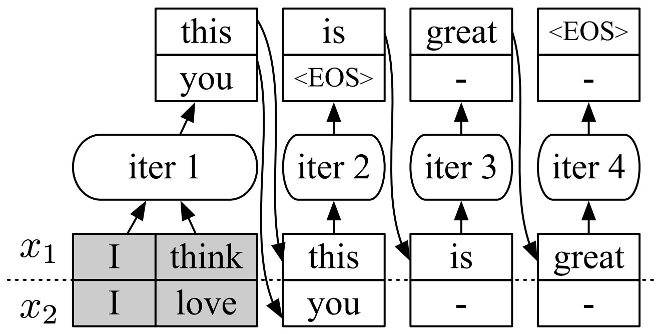


Figure 4.1: An illustration for a case where the requests have the same input length but some requests finish earlier than others. Shaded tokens represent input tokens. "-" denotes inputs and outputs of extra computation imposed by the scheduling.

4.2 Challenges and Proposed Solutions

In this section, we describe challenges in serving Transformer-based generative models and propose two techniques: iteration-level scheduling and selective batching.

C1: Early-finished and late-joining requests. One major limitation of existing systems is that the inference server and the execution engine interact with each other only when (1) the server schedules the next batch on an idle engine; or (2) the engine finishes processing the current batch. In other words, these systems are designed to schedule executions at *request* granularity; the engine maintains a batch of requests fixed until all requests in the batch finish. This can be problematic in the serving of generative models, since each request in a batch may require different number of iterations, resulting in certain requests finishing earlier than the others. In the example shown in Figure 4.1, although request x_2 finishes earlier than request x_1, the engine performs computation for both "active" and "inactive" requests throughout all iterations.

Such extra computation for inactive requests (x_2 at iter 3 and 4) limits the efficiency of batched execution.

What makes it even worse is that this behavior prevents an early return of the finished request to the client, imposing a substantial amount of extra latency. This is because the engine only returns the execution results to the server when it finishes processing all requests in the batch. Similarly, when a new request arrives in the middle of the current batch's execution, the aforementioned scheduling mechanism makes the newly arrived request wait until all requests in the current batch have finished. We argue that the current request-level scheduling mechanism cannot efficiently handle workloads with multi-iteration characteristic. Note that this problem of early-finished and late-joining requests does not occur in the training of language models; the training procedure finishes processing the whole batch in a single iteration by using the teacher forcing technique [150].

S1: Iteration-level scheduling. To address the above limitations, we propose to schedule executions at the granularity of *iteration*. At high level, the scheduler repeats the following procedure: (1) selects requests to run next; (2) invokes the engine to execute *one iteration* for the selected requests; and (3) receives execution results for the scheduled iteration. Since the scheduler receives a return on every iteration, it can detect the completion of a request and immediately return its generated tokens to the client. For a newly arrived request, the request gets a chance to start processing (i.e., the scheduler may select the new request to run next) after execution of the currently scheduled iteration, significantly reducing the queueing delay. With iteration-level scheduling, the scheduler has a full control on how many and which requests are processed in each iteration.

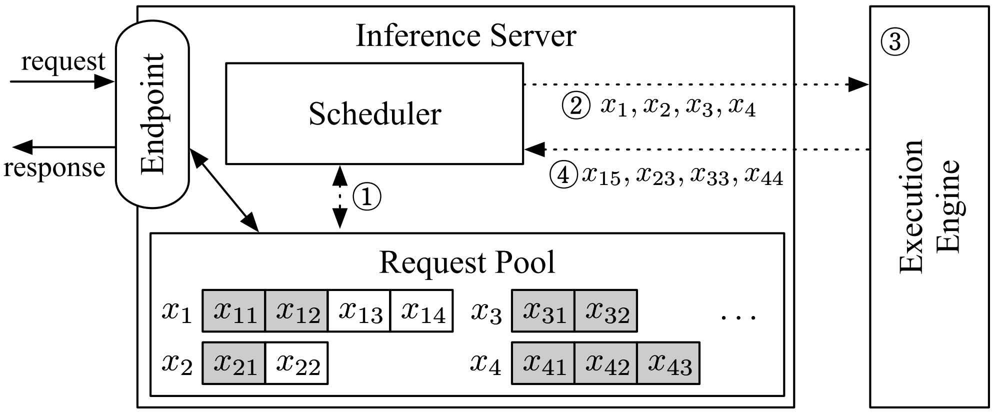


Figure 4.2: System overview of ORCA. Interactions between components represented as dotted lines indicate that the interaction takes place at every iteration of the execution engine. x_{ij} is the j-th token of the i-th request. Shaded tokens represent input tokens received from the clients, while unshaded tokens are generated by ORCA. For example, request x_1 initially arrived with two input tokens (x_{11}, x_{12}) and have run two iterations so far, where the first and second iterations generated x_{13} and x_{14}, respectively. On the other hand, request x_3 only contains input tokens (x_{31}, x_{32}) because it has not run any iterations yet.

Figure 4.2 depicts the system architecture and the overall workflow of ORCA using the iteration-level scheduling. ORCA exposes an *endpoint* (e.g., HTTPS or gRPC) where inference requests arrive at the server and responses to the requests are sent out. The endpoint puts newly arrived requests in the *request pool*, a component that manages all requests in the server during their lifetime. The pool is monitored by the *scheduler*, which is responsible for: selecting a set of requests from the pool, scheduling the *execution engine* to run an iteration of the model on the set, receiving execution results (i.e., output tokens) from the engine, and updating the pool by appending each output token to the corresponding request. The engine is an abstraction for executing the actual tensor operations, which can be parallelized across multiple GPUs spread across multiple machines. In the example shown in Figure 4.2, the scheduler ① interacts with the request pool to decide which requests to run next and ② invokes the engine to run four selected requests: (x_1, x_2, x_3, x_4). The scheduler provides the engine with input tokens of the requests scheduled for the first time. In this case, x_3 and x_4 have not run any iterations yet, so the scheduler hands over (x_{31}, x_{32}) for x_3 and (x_{41}, x_{42}, x_{43}) for x_4. The engine ③ runs an iteration of the model on the four requests and ④ returns generated output tokens $(x_{15}, x_{23}, x_{33}, x_{44})$, one for each scheduled request. Once a request has finished processing, the request pool removes the finished request and notifies the endpoint to send a response. Unlike the method shown in Figure 2.5 that should run multiple iterations on a scheduled batch until finish of all requests within the batch, ORCA's scheduler can change which requests are going to be processed at every iteration. We describe the detailed algorithm about how to select the requests at every iteration in Section 4.3.2.

C2: Batching an arbitrary set of requests. When we try to use the iteration-level scheduling in practice, one major challenge that we are going to face is batching. To achieve high efficiency, the execution engine should be able to process any selected set of requests in the batched manner. Without batching, one would have to process each selected request one by one, losing out on the massively parallel computation capabilities of GPUs.

Unfortunately, there is no guarantee that even for a pair of requests (x_i, x_j), for the next iteration, their executions can be merged and replaced with a batched version. There are three cases for a pair of requests where the next iteration cannot be batched together: (1) both requests are in the initiation phase and each has different number of input tokens (e.g., x_3 and x_4 in Figure 4.2); (2) both are in the increment phase and each is processing a token at different index from each other (x_1 and x_2); or (3) each request is in the different phase:

initiation or increment (x_1 and x_3). Recall that in order to batch the execution of multiple requests, the execution of each request must consist of identical operations, each consuming identically-shaped input tensors. In the first case, the two requests cannot be processed in a batch because the "length" dimension of their input tensors, which is the number of input tokens, are not equal. The requests in the second case have difference in the tensor shape of Attention keys and values because each processes token at different index, as shown in Figure 2.4. For the third case, we cannot batch the iterations of different phases because they take different number of tokens as input; an iteration of the initiation phase processes all input tokens in parallel for efficiency, while in the increment phase each iteration takes a single token as its input (we assume the use of fairseq-style incremental decoding [103]).

Batching is only applicable when the two selected requests are in the same phase, with the same number of input tokens (in case of the initiation phase) or with the same token index (in case of the increment phase). This restriction significantly reduces the likelihood of batching in real-world workloads, because the scheduler should *make a wish* for the presence of two requests eligible for batching at the same time. The likelihood further decreases exponentially as the batch size increases, making it impractical to use a large batch size that can pull out better throughput without compromising latency.

S2: Selective batching. We propose *selective batching*, a technique for batched execution that allows high flexibility in composing requests as a batch. Instead of processing a batch of requests by "batchifying" all tensor operations composing the model, this technique selectively apply batching only to a handful of operations.

The main problem regarding batching described above is that the three

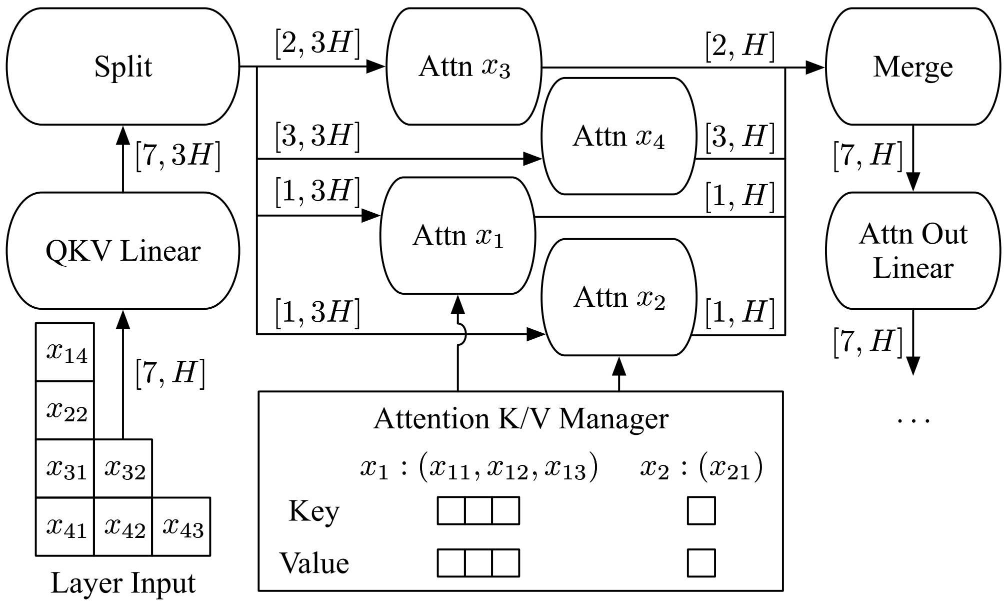


Figure 4.3: An illustration of ORCA execution engine running a Transformer layer on a batch of requests with selective batching. We only depict the QKV Linear, Attention, and Attention Out Linear operations for simplicity.

aforementioned cases[2] correspond to irregularly shaped input (or state) tensors, which cannot be coalesced into a single large tensor and fed into a batch operation. In the canonical batching mechanism, at each iteration, a Transformer layer takes a 3-dimensional input tensor of shape $[B, L, H]$ generated by concatenating multiple $[L, H]$ input tensors of requests in a batch, where B is the batch size, L is the number of tokens processed together, and H is the hidden size of the model. For example, in Figure 4.1, "iter 1" (initiation phase) takes an input tensor of shape $[2, 2, H]$ and "iter 2" (increment phase) takes a tensor of shape $[2, 1, H]$. However, when the scheduler decides to run an iteration on batch (x_1, x_2, x_3, x_4) in Figure 4.2, the inputs for requests in the initiation phase ($x_3 : [2, H]$ and $x_4 : [3, H]$) cannot coalesce into a single tensor

[2]We use the first case as a driving example, but the argument can be similarly applied to the other two cases.

of shape $[B, L, H]$ because x_3 and x_4 have different number of input tokens, 2 and 3.

Interestingly, not all operations are incompatible with such irregularly shaped tensors. Operations such as non-Attention matrix multiplication and layer normalization can be made to work with irregularly shaped tensors by flattening the tensors. For instance, the aforementioned input tensors for x_3 and x_4 can compose a 2-dimensional tensor of shape $[\sum L, H] = [5, H]$ without an explicit batch dimension. This tensor can be fed into all non-Attention operations including Linear, LayerNorm, Add, and GeLU operations because they do not need to distinguish tensor elements of different requests. On the other hand, the Attention operation requires a notion of requests (i.e., requires the batch dimension) to compute attention only between the tokens of the same request, typically done by applying cuBLAS routines for batch matrix multiplication.

Selective batching is aware of the different characteristics of each operation; it splits the batch and processes each request individually for the Attention operation while applying token-wise (instead of request-wise) batching to other operations without the notion of requests. Figure 4.3 presents the selective batching mechanism processing a batch of requests (x_1, x_2, x_3, x_4) described in Figure 4.2. This batch has 7 input tokens to process, so we make the input tensor have a shape of $[7, H]$ and apply the non-Attention operations. Before the Attention operation, we insert a `Split` operation and run the Attention operation separately on the split tensor for each request. The outputs of Attention operations are merged back into a tensor of shape $[7, H]$ by a `Merge` operation, bringing back the batching functionality to the rest of operations.

To make the requests in the increment phase can use the Attention keys and values for the tokens processed in previous iterations, ORCA maintains the generated keys and values in the *Attention K/V manager*. The manager maintains these keys and values separately for each request until the scheduler explicitly asks to remove certain request's keys and values, i.e., when the request has finished processing. The Attention operation for request in the increment phase (x_1 and x_2) takes keys and values of previous tokens (x_{11}, x_{12}, x_{13} for x_1; x_{21} for x_2) from the manager, along with the current token's query, key, and value from the Split operation to compute attention between the current token and the previous ones.

4.3 Orca System Design

Based on the above techniques, we design and implement ORCA: a distributed serving system for Transformer-based generative models. We have already discussed the system components and the overall execution model of ORCA while describing Figure 4.2. In this section, we answer the remaining issues about how to build an efficient system that can scale to large-scale models with hundreds of billions of parameters. We also describe the scheduling algorithm for iteration-level scheduling, i.e., how to select a batch of requests from the request pool at every iteration.

4.3.1 Distributed Architecture

Recent works [71, 27] have shown that scaling language models can dramatically improve the quality of models. Hence, system support for serving such large language models is getting more importance, especially when the model does not fit in a single GPU. In such a case, one should split the model parameters along with the corresponding computation and distribute them across multiple GPUs and machines.

ORCA composes known parallelization techniques for Transformer models: intra-layer parallelism and inter-layer parallelism. These two model parallelism

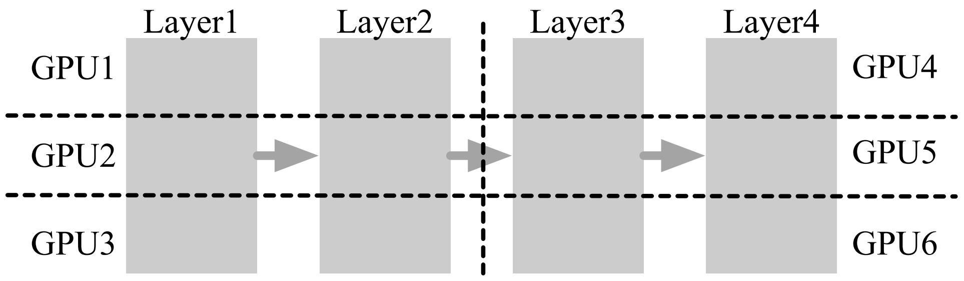


Figure 4.4: An example of intra- and inter- layer parallelism. A vertical dotted line indicates partitioning between layers and a horizontal line indicates partitioning within a layer.

strategies, which are also used by FasterTransformer [7], have been originally developed for distributed training. Intra-layer parallelism [129, 132] splits matrix multiplications (i.e., Linear and Attention operations) and their associated parameters over multiple GPUs. We omit the detail about how this strategy partitions each matrix multiplication. On the other hand, inter-layer parallelism splits Transformer layers over multiple GPUs. ORCA assigns the same number of Transformer layers to each GPU. Figure 4.4 illustrates an example application of intra- and inter- layer parallelism to a 4-layer GPT model. The 4 layers are split into 2 inter-layer partitions, and the layers in the partition are subdivided into 3 intra-layer partitions. We assign each partition to a GPU, using a total of 6 GPUs.

The ORCA execution engine supports distributed execution using the techniques described above. Figure 4.5 depicts the architecture of an ORCA engine. Each *worker process* is responsible for an inter-layer partition of the model and can be placed on a different machine from each other. In particular, each worker manages one or more CPU threads each dedicated for controlling a GPU, the number of which depends on the degree of intra-layer parallelism.

The execution procedure of the ORCA execution engine is as follows. Once

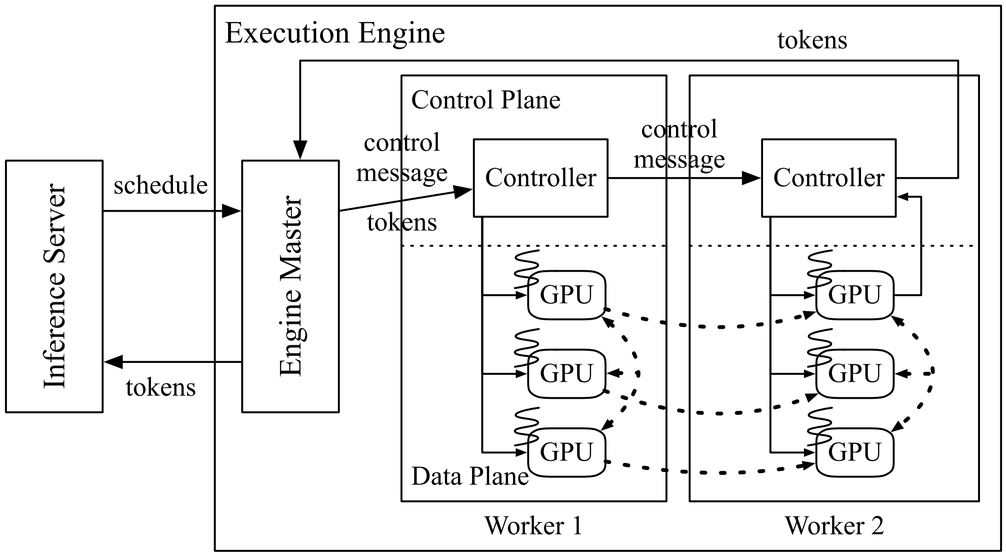


Figure 4.5: An illustration of the distributed architecture of ORCA's execution engine using the parallelization configuration shown in Figure 4.4. For example, the first inter-layer partition (Layer1 and Layer2) in Figure 4.4 is assigned to Worker1, while the second partition is assigned to Worker2.

the engine is scheduled to run an iteration of the model for a batch of requests, the *engine master* forwards the received information about the scheduled batch to the first *worker process* (Worker1). The information includes tokens for the current iteration and a control message, which is composed of ids of requests within the batch, current token index (for requests in the increment phase), and number of input tokens (for requests in the initiation phase). The *controller* of Worker1 hands over the information received from the engine master to the GPU-controlling threads, where each thread parses the information and issues proper GPU kernels to its associated GPU. For example, the kernel for the Attention operation uses the request id and the current token index to get the GPU memory address of previous keys and values kept by the Attention K/V manager. In the meantime, the controller also forwards the control message to the controller of the next worker (Worker2), without waiting for the completion of the kernels issued on the GPUs of Worker1. Unlike Worker1, the controller of the last worker (Worker2) waits for (i.e., synchronize with) the completion of the issued GPU kernels, in order to fetch the output token for each request and send the tokens back to the engine master.

To keep GPUs busy as much as possible, we design the Orca engine to minimize synchronization between the CPU and GPUs. We observe that current systems for distributed inference (e.g., FasterTransformer [7] and Megatron-LM [5]) have CPU-GPU synchronization whenever each process receives control messages[3] because they exchange the messages through a GPU-to-GPU communication channel – NCCL [8]. The exchange of these control messages occurs at every iteration, imposing a non-negligible performance overhead. On the other hand, Orca separates the communication channels for control messages (plus tokens) and tensor data transfer, avoiding the use of NCCL for data used by CPUs. Figure 4.5 shows that the Orca engine uses NCCL exclusively for exchanging intermediate tensor data (represented by dashed arrows) as this data is produced and consumed by GPUs. Control messages, which is used by the CPU threads for issuing GPU kernels, sent between the engine master and worker controllers by a separate communication channel that does not involve GPU such as gRPC [4].

4.3.2 Scheduling Algorithm

The Orca scheduler makes decisions on which requests should be selected and processed at every iteration. The scheduler has high flexibility in selecting a set of requests to compose a batch, because of the selective batching technique that allows the engine to run any set of requests in the batched manner. Now

[3]This includes various metadata such as batch size, sequence length, and whether a request within the batch has finished processing.

the main question left is how to select the requests at every iteration.

We design the Orca scheduler to use a simple algorithm that does not change the processing order of client requests; early-arrived requests are processed earlier. That is, we ensure iteration-level first-come-first-served (FCFS) property. We define the iteration-level FCFS property for workloads with multi-iteration characteristics as follows: for any pair of requests (x_i, x_j) in the request pool, if x_i has arrived earlier than x_j, x_i should have run the same or more iterations than x_j. Note that some late-arrived requests may return earlier to clients if the late request requires a smaller number of iterations to finish.

Still, the scheduler needs to take into account additional factors: diminishing returns to increasing the batch size and GPU memory constraint. Increasing the batch size trades off increased throughput for increased latency, but as the batch size becomes larger, the amount of return (i.e., increase in throughput) diminishes. Therefore, just like other serving systems [38, 11], Orca also has a notion of a max batch size: the largest possible number of requests within a batch. The Orca system operator can tune this knob to maximize throughput while satisfying one's latency budget. We will discuss this in more details with experiment results in Section 4.5.2.

Another factor is the GPU memory constraint. Optimizing memory usage by reusing buffers for intermediate results across multiple operations is a well-known technique used by various systems [9, 7], and Orca also adopts this technique. However, unlike the buffers for intermediate results that can be reused immediately, buffers used by the Attention K/V manager for storing the keys and values cannot be reclaimed until the Orca scheduler notifies that the corresponding request has finished processing. A naïve implementation can make the scheduler fall into a deadlock when the scheduler cannot issue an iteration for any requests in the pool because there is no space left for storing a

Algorithm 2: ORCA scheduling algorithm

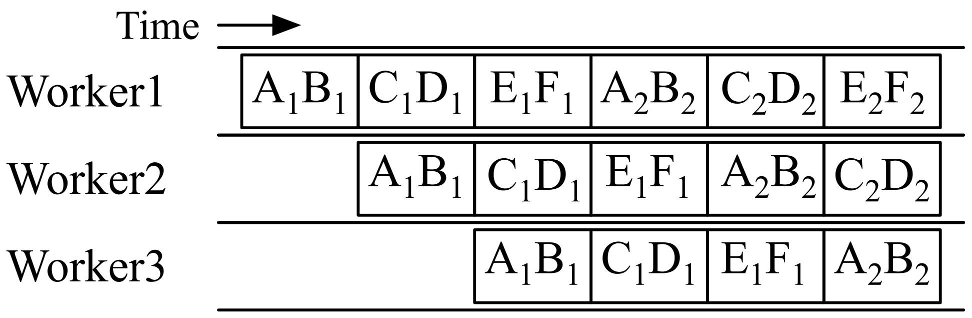
```
Params: n_workers: number of workers, max_bs: max batch size,
        n_slots: number of K/V slots
n_scheduled ← 0
n_rsrv ← 0
while true do
    batch, n_rsrv ← Select(request_pool, n_rsrv)
    schedule engine to run one iteration of the model for the batch
    foreach req in batch do
        req.state ← RUNNING
    n_scheduled ← n_scheduled + 1
    if n_scheduled = n_workers then
        wait for return of a scheduled batch
        foreach req in the returned batch do
            req.state ← INCREMENT
            if finished(req) then
                n_rsrv ← n_rsrv − req.max_tokens
        n_scheduled ← n_scheduled − 1

def Select(pool, n_rsrv):
    batch ← {}
    pool ← {req ∈ pool | req.state ≠ RUNNING}
    SortByArrivalTime(pool)
    foreach req in pool do
        if batch.size() = max_bs then break
        if req.state = INITIATION then
            new_n_rsrv ← n_rsrv + req.max_tokens
            if new_n_rsrv ¿ n_slots then break
            n_rsrv ← new_n_rsrv
        batch ← batch ⋃ {req}
    return batch, n_rsrv
```

new Attention key and value for the next token. This requires the ORCA scheduler to be aware of the remaining size of pre-allocated memory regions for the manager.

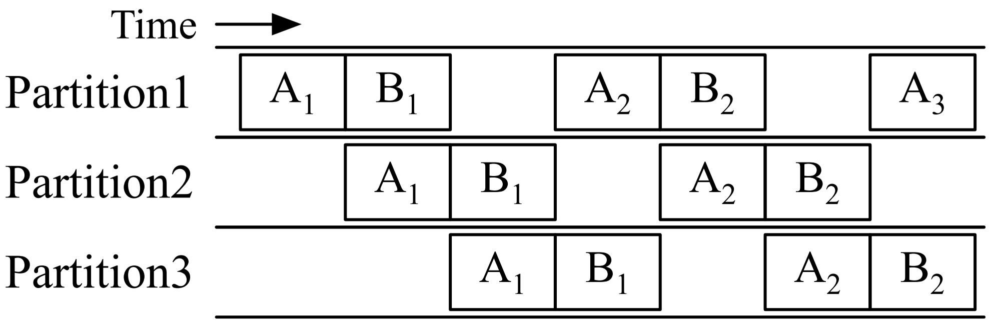
The ORCA scheduler takes all these factors into account: it selects at most "max batch size" requests based on the arrival time, while reserving enough space for storing keys and values to a request when the request is scheduled for the first time. We describe the scheduling process in Algorithm 2. The algorithm selects a batch of requests from the request pool (line 4) and schedules the batch (line 5). The *Select* function (line 17) selects at most *max_bs* requests from the pool based on the arrival time of the request (lines 20-22). Algorithm 2 does not depict the procedure of request arrival and return; one may think of it as there exist concurrent threads inserting newly arrived requests into *request_pool* and removing finished requests from *request_pool*.

When the scheduler considers a request in the initiation phase, meaning that the request has never been scheduled yet, the scheduler uses the request's *max_tokens*[4] attribute to reserve *max_tokens* slots of GPU memory for storing the keys and values in advance (lines 23-26). The scheduler determines whether the reservation is possible (line 25) based on *n_rsrv*, the number of currently reserved slots, where a slot is defined by the amount of memory required for storing an Attention key and value for a single token. Here, *n_slots* is a parameter tuned by the ORCA system operator indicating the size of memory region (in terms of slots) allocated to the Attention K/V manager. Since the number of tokens in a request cannot exceed *max_tokens*, if the reservation is possible, it is guaranteed that the manager can allocate buffers for the newly generated keys and values until the request finishes.

[4]The *max_tokens* attribute is a per-request option, meaning the maximum number of tokens that a request can have after processing.


Time
Worker1
A_1B_1 C_1D_1 E_1F_1 A_2B_2 C_2D_2 E_2F_2
Worker2
A_1B_1 C_1D_1 E_1F_1 A_2B_2 C_2D_2
Worker3
A_1B_1 C_1D_1 E_1F_1 A_2B_2

(a) Orca execution pipeline.



(b) FasterTransformer execution pipeline.

Figure 4.6: Comparison of the use of pipeline parallelism in Orca and Faster-Transformer where X_i is the i-th iteration of request X.

Unlike the tuning of *max_bs* that requires quantifying the trade-off between latency and throughput, the Orca system operator can easily configure *n_slots* without any experiments. Given a model specification (e.g., hidden size, number of layers, etc.) and degrees of intra- and inter- layer parallelism, Orca's GPU memory usage mostly depends on *n_slots*. That is, the operator can simply use the largest possible *n_slots* under the memory constraint.

Pipeline parallelism. Orca's scheduler makes the execution of workers in the engine to be pipelined across multiple batches. The scheduler does not wait for the return of a scheduled batch until *n_scheduled*, the number of currently scheduled batches, reaches *n_workers* (line 9-10 of Algorithm 2). By doing so, the scheduler keeps the number of concurrently running batches in the engine to be *n_workers*, which means that every worker in the engine is processing one of the batches without being idle.

Figure 4.6a depicts the execution pipeline of 3 ORCA workers, using a max batch size of 2. We assume that the request A arrives before B, which arrives before C, and so on. At first, the scheduler selects requests A and B based on the arrival time and schedules the engine to process a batch of requests A and B (we call this batch AB), where Worker1, Worker2, and Worker3 process the batch in turn. The scheduler waits for the return of the batch AB only after the scheduler injects two more batches: CD and EF. Once the batch AB returns, requests A and B get selected and scheduled once again, because they are the earliest arrived requests among the requests in the pool.

In contrast, the interface between current inference servers and execution engines (e.g., a combination of Triton [11] and FasterTransformer [7]) does not allow injecting another batch before the finish of the current running batch, due to the request-level scheduling. That is, Triton cannot inject the next request C to FasterTransformer until the current batch AB finishes. To enable pipelined execution of multiple inter-layer partitions under such constraint, FasterTransformer splits a batch of requests into multiple *microbatches* [61] and pipelines the executions of partitions across the microbatches. In Figure 4.6b, FasterTransformer splits the batch AB into two microbatches, A and B. Since each partition processes a microbatch (which is smaller than the original batch) in the batched manner, the performance gain from batching can become smaller. Moreover, this method may insert *bubbles* into the pipeline when the microbatch size is too large, making the number of microbatches smaller than the number of partitions. While FasterTransformer needs to trade batching efficiency (larger microbatch size) for pipelining efficiency (fewer pipeline bubbles), ORCA is free of such a tradeoff – thanks to iteration-level scheduling – and can easily pipeline requests without dividing a batch into microbatches.

4.4 Implementation

We have implemented ORCA with 13K lines of C++, based on the CUDA ecosystem. We use gRPC [4] for the communication in the control plane of the ORCA engine, while NCCL [8] is used in the data plane, for both inter-layer and intra-layer communication. Since we design ORCA to focus on Transformer-based generative models, ORCA provides popular Transformer layers as a building block of models including the original encoder-decoder Transformer [146], GPT [113], and other variants discussed in Raffel et al. [115]. Supporting more complex neural network architecture like Mixture of Experts [48] is future work.

We have also implemented fused kernels for LayerNorm, Attention, and GeLU operators, just like other systems for training or inference of Transformer models [132, 3, 7]. For example, the procedure of computing dot products between Attention query and keys, Softmax on the dot products, and weighted average of Attention values are fused into a single CUDA kernel for the Attention operator. In addition, we go one step further and fuse the kernels of the split Attention operators by simply concatenating all thread blocks of the kernels for different requests. Although this fusion makes the thread blocks within a kernel have different characteristics and lifetimes (which is often discouraged by CUDA programming practice) because they process tensors of different shapes, we find this fusion to be beneficial by improving GPU utilization and reducing the kernel launch overhead [91, 84].

# Params	# Layers	Hidden size	# Inter-partitions	# Intra-partitions
13B	40	5120	1	1
101B	80	10240	1	8
175B	96	12288	2	8
341B	120	15360	4	8

Table 4.1: Configurations of models used in the experiments.

Scheduling overhead and prefetching As we make the ORCA scheduler interact with the ORCA engine at every iteration, ORCA inserts an extra scheduling overhead compared to the baseline systems. Per our experiment, we observe that such extra overhead leads to a small gap between the two consecutive iterations where the GPUs are left idle. While in most cases this does not incur any negative performance impact, it can be problematic for relatively small models with less than 100M parameters. In such a case, one can consider changing the condition in line 9 of Algorithm 2 to $n\_scheduled = n\_workers + 1$. By doing so, the ORCA engine prefetches one more control message for the next iteration, hiding the extra scheduling overhead inserted by the iteration-level scheduling.

4.5 Evaluation

In this section, we present evaluation results to show the efficiency of ORCA.

Environment. We run our evaluation on Azure ND96asr A100 v4 VMs, each equipped with 8 NVIDIA 40-GB A100 GPUs connected over NVLink. We use at most four VMs depending on the size of the model being tested. Each VM has 8 Mellanox 200Gbps HDR Infiniband adapters, providing an 1.6Tb/s of interconnect bandwidth between VMs.

Models. Throughout the experiments, we use GPT [27] as a representative example of Transformer-based generative models. We use GPT models with various configurations, which is listed in Table 4.1. The configurations for 13B and 175B models come from the GPT-3 paper [27]. Based on these two models, we change the number of layers and hidden size to make configurations for 101B and 341B models. All models have a maximum sequence length of 2048, following the setting of the original literature [27]. We use fp16-formatted model parameters and intermediate activations for the experiments. We also apply inter- and intra- layer parallelism strategies described in Section 4.3.1, except for the 13B model that can fit in a GPU. For example, the 175B model is partitioned over a total of 16 GPUs by using 2 inter-layer partitions subdivided into 8 intra-layer partitions, where the 8 GPUs in the same VM belongs to the same inter-layer partition.

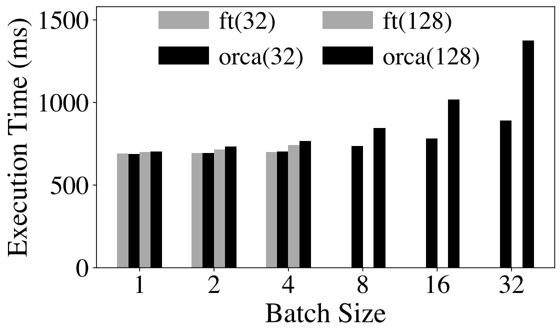
Baseline system. We compare with FasterTransformer [7], an inference engine that supports large scale Transformer models via distributed execution. While there exist other systems with the support for distributed execution such as Megatron-LM [5] and DeepSpeed [3], these systems are primarily designed and optimized for training workloads, which makes them show relatively lower performance compared to the inference-optimized systems.

Scenarios. We use two different scenarios to drive our evaluation. First, we design a microbenchmark to solely assess the performance of the Orca engine without being affected by the iteration-level scheduling. In particular, we do not run the Orca server in this scenario. Instead, given a batch of requests, the testing script repeats injecting the same batch into the Orca engine until all requests in the batch finishes, mimicking the behavior of the canonical request-level scheduling. We also assume that all requests in the batch have the same number of input tokens and generate the same number of output tokens. We report the time taken for processing the batch (not individual requests) and compare the result with FasterTransformer [7].

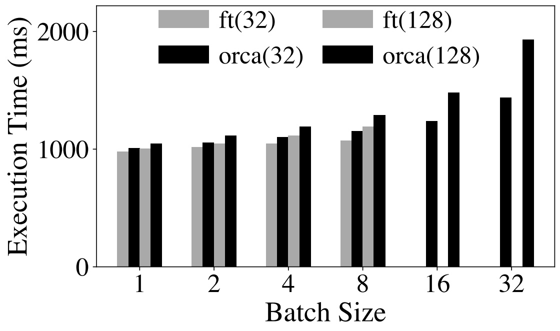
The second scenario tests the end-to-end performance of ORCA by emulating a workload. We synthesize a trace of client requests because there is no publicly-available request trace for generative language models. Each request in the synthesized trace is randomly generated by sampling the number of input tokens and a *max_gen_tokens* attribute, where the number of input tokens plus *max_gen_tokens* equals to the *max_tokens* attribute described in Section 4.3.2. We assume that all requests continue generation until the number of generated tokens reaches *max_gen_tokens*. In other words, we make the model never emit the "<EOS>" token. This is because we have neither the actual model checkpoint nor the actual input text so we do not have any information to guess the right timing of the "<EOS>" token generation. Once the requests are generated, we synthesize the trace by setting the request arrival time based on the Poisson process. To assess ORCA's behavior under varying load, we change the Poisson parameter (i.e., arrival rate) and adjust the request arrival time accordingly. We report latency and throughput using multiple traces generated from different distributions for better comparison and understanding of the behavior of ORCA and FasterTransformer.

4.5.1 Engine Microbenchmark

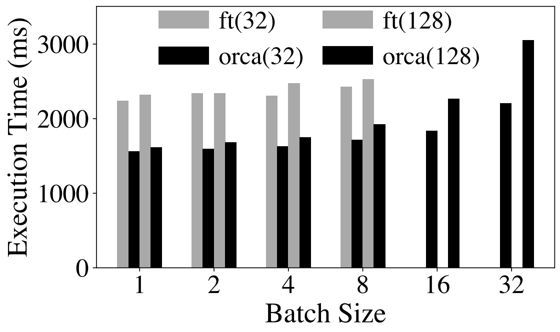
We first compare the performance of FasterTransformer and the ORCA engine using the first scenario. We set all requests in the batch to have the same number of input tokens (32 or 128) and generate 32 tokens. That is, in this set of experiments, all requests within the batch start and finish processing at the



(a) 13B model, 1 GPU.



(b) 101B model, 8 GPUs.



(c) 175B model, 16 GPUs.

Figure 4.7: Execution time of a batch of requests using FasterTransformer and the ORCA engine without the scheduling component. Label "ft(n)" represents results from FasterTransformer processing requests with n input tokens. Configurations that incurs out of memory error are represented as missing entries (e.g., ft(32) for the 101B model with a batch size of 16).

same time. We conduct experiments using three different models: 13B, 101B, and 175B. For each model, we use the corresponding parallelization strategy shown in Table 4.1.

Figure 4.7 shows the performance of FasterTransformer and the Orca engine for processing a batch composed of the same requests. In Figure 4.7a, the Orca engine shows a similar (or slightly worse) performance compared to FasterTransformer across all configurations. This is because Orca does not apply batching to the Attention operations, while FasterTransformer apply batching to all operations. Still, the performance difference is relatively small. Despite not batching the Attention operation, the absence of model parameters in Attention makes this decision has little impact on efficiency as there is no benefit of reusing model parameters across multiple requests.

Figure 4.7b presents similar results for the 101B model that uses all of the 8 GPUs in a single VM. From these results, we can say that the Orca engine and FasterTransformer have comparable efficiencies in the implementations of CUDA kernels and the communication between intra-layer partitions. Note that FasterTransformer cannot use a batch size of 8 or larger with the 13B model (16 or larger with the 101B model) because of the fixed amount of memory pre-allocation for each request's Attention keys and values, which grows in proportion to the max sequence length of the model (2048 for this case). In contrast, Orca avoids redundant memory allocation by setting the size of buffers for the keys and values separately for each request based on the *max_tokens* attribute.

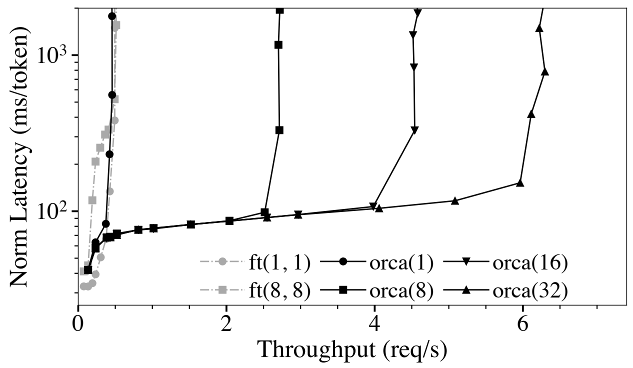
Next, we go one step further and experiment with the 175B model, which splits the layers into two inter-layer partitions. In this case, for better comparison, we disable pipelined execution of the inter-layer partitions for both systems. For FasterTransformer, we set the size of a microbatch to be equal to the batch size to disable pipelining. As shown in Figure 4.7c, the Orca engine outperforms FasterTransformer by up to 47%. We attribute this performance improvement to the control-data plane separation described in Section 4.3.1. We omit the 341B model as it has similar results compared to the 175B model.

4.5.2 End-to-end Performance

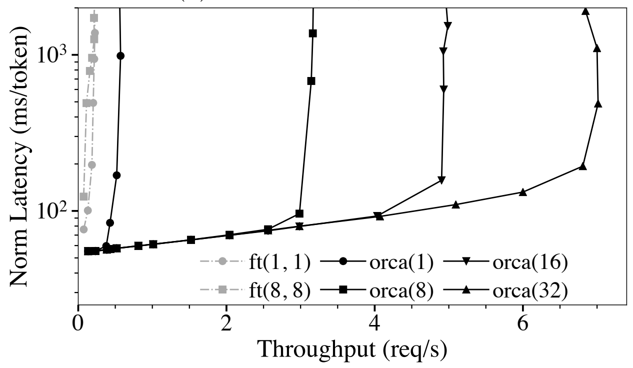
Now we assess the end-to-end performance of Orca by measuring the latency and throughput with the synthesized request trace under varying load. When synthesizing the trace, we sample each request's number of input tokens from $U(32, 512)$, a uniform distribution ranging from 32 to 512 (inclusive). The *max_gen_tokens* attributed is sampled from $U(1, 128)$, which means that the least and the most time-consuming requests require 1 and 128 iterations of the model for processing, respectively.

Unlike the microbenchmark shown in Section 4.5.1, to measure the end-to-end performance, we test the entire Orca software stack including the Orca server. Client requests arrive to the Orca server following the synthesized trace described above. We report results from various max batch size configurations. For FasterTransformer that does not have its own inference server, we implement a custom server that receives client requests, creates batches, and injects the batches to an instance of FasterTransformer. We make the custom server create batches dynamically by taking at most max batch size requests from the request queue, which is the most common scheduling algorithm used by existing inference servers like Triton [11] and TensorFlow Serving [101]. Again, we report results from various max batch size configurations, along with varying microbatch sizes, an additional knob in FasterTransformer that governs the pipelining behavior (see Section 4.3.2).

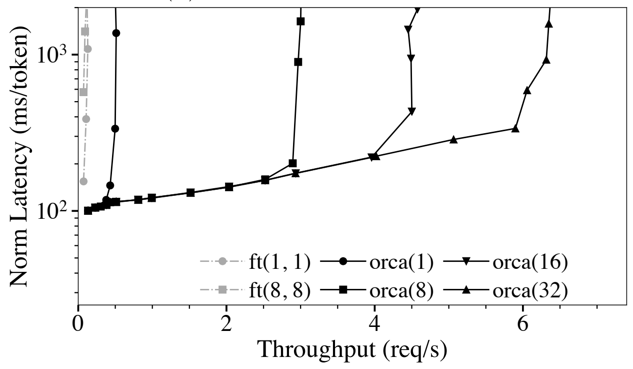
Figure 4.8 shows median end-to-end latency and throughput. Since each



(a) 101B model, 8 GPU.



(b) 175B model, 16 GPUs.



(c) 341B model, 32 GPUs.

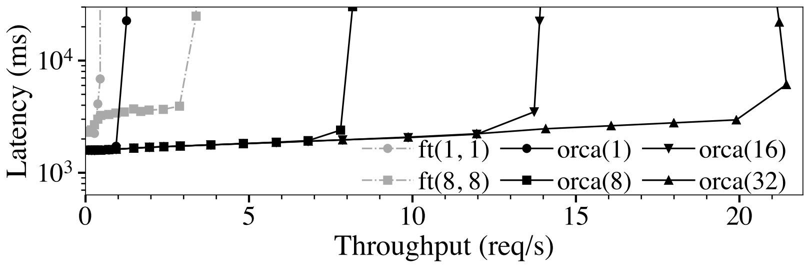
Figure 4.8: Median end-to-end latency normalized by the number of generated tokens and throughput. Label "orca(max_bs)" represents results from ORCA with a max batch size of max_bs. Label "ft(max_bs, mbs)" represents results from FasterTransformer with a max batch size of max_bs and a microbatch size of mbs.

request in the trace requires different processing time, which is (roughly) in proportion to the number of generated tokens, we report median latency normalized by the number of generated tokens of each request. From the figure, we can see that Orca provides significantly higher throughput and lower latency than FasterTransformer. The only exception is the 101B model under low load (Figure 4.8a). In this case, both Orca and FasterTransformer do not have enough number of requests to process in a batch. That is, the latency will mostly depend on the engine's performance, which is shown in Figure 4.7b. As the load becomes heavier, Orca provides higher throughput with a relatively small increase in latency, because the Orca scheduler makes late-arrived requests hitch a ride with the current ongoing batch. In contrast, FasterTransformer fails to efficiently handle multiple requests that (1) arrive at different times; (2) require different number of iterations to finish; or (3) start with different number of input tokens, resulting in a peak throughput of 0.49 req/s and much higher latency. If we use the 175B or 341B model (Figures 4.8b and 4.8c) that employs more than one inter-layer partitions, Orca outperforms FasterTransformer under every level of load in terms of both latency and throughput, resulting in an order of magnitude higher throughput when we compare results at a similar level of latency. For example, to match a median normalized latency of 190ms for the 175B model, which is a double of the normalized execution time (by the number of generated tokens) of "orca(128)" shown in Figure 4.7c, FasterTransformer provides a throughput of 0.185 req/s whereas Orca provides a throughput of 6.81 req/s, which is a 36.9× speedup.

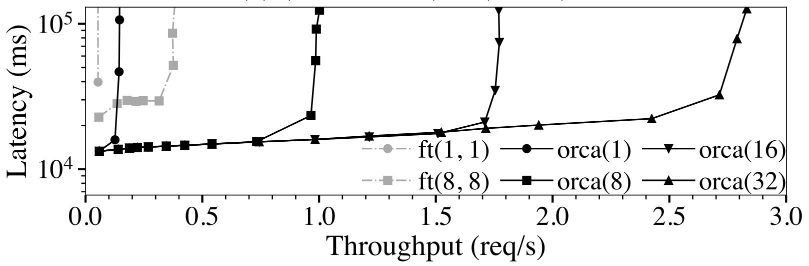
Varying batch size configurations. Figure 4.8 shows that the increase of the max batch size of Orca results in a higher throughput without affecting the latency. This is because the iteration-level scheduling of Orca resolves the problem of early-finished and late-joining requests. Nevertheless, there is no guarantee that increasing the batch size will not negatively affect the latency, for arbitrary hardware settings, models, and workloads. As mentioned in Section 4.3.2, the max batch size must be set carefully by considering both the required latency and throughput requirements.

Interestingly, larger max batch size in FasterTransformer does not necessarily help improving throughput. By testing all possible combinations of max batch size (*max_bs*) and microbatch size (*mbs*) on all models under varying load, we find that (*max_bs*, *mbs*) = (1, 1) or (8, 8) are the best options. Per our discussion in Section 4.3.1, FasterTransformer's microbatch-based pipelining can be less efficient because the engine is going to process at most *mbs* number of requests in the batched manner, which explains why the configurations with the maximum possible *mbs* (which is the same as *max_bs*) have better performance than others. In addition, while increasing *max_bs* can improve performance due to the increased batch size, at the same time, this also increases the likelihood of batching requests with large difference in the number of input tokens or the number of generated tokens. In such cases, FasterTransformer cannot efficiently handle the batch because (1) for the first iteration of the batch, FasterTransformer processes requests as if they all had the same input length as the shortest one; and (2) early-finished requests cannot immediately return to the clients.

Trace of homogeneous requests. We test the behavior of ORCA and FasterTransformer when using a trace of homogeneous requests, i.e., all requests in a trace have the same number of input tokens and the same *max_gen_tokens* attribute. Since all requests require the same number of iterations to finish processing, the problem of early-leaving requests does not occur for this trace. As a result, now the increase of the *max_bs* has a noticeable positive impact



(a) (# in, # gen) = (32, 32)



(b) (# in, # gen) = (256, 256)

Figure 4.9: Median end-to-end latency and throughput, using the 175B model with traces composed of homogeneous requests. We do not normalize the latency since all requests have the same characteristic.

on the performance of FasterTransformer, as shown in Figure 4.9. Still, Orca outperforms FasterTransformer (*max_bs*=8) except for the case using a max batch size of 1, where Orca degenerates into a simple pipeline of the Orca workers that does not perform batching.

4.6 Summary

We present iteration-level scheduling with selective batching, a novel approach that achieves low latency and high throughput for serving Transformer-based generative models. Iteration-level scheduling makes the scheduler interact with the execution engine at the granularity of iteration instead of request, while selective batching enables batching arbitrary requests processing tokens at different positions, which is crucial for applying batching with iteration-level scheduling. Based on these techniques, we have designed and implemented a distributed serving system named Orca. Experiments show the effectiveness of our approach: Orca provides an order of magnitude higher throughput than current state-of-the-art systems at the same level of latency.

Chapter 5

WindTunnel: Towards Differentiable ML Pipelines Beyond a Single Model

5.1 Introduction

The recent decade has witnessed two distinct trends in Machine Learning (ML). On one hand, the success of Deep Neural Networks (DNNs) has been the driving force of many recent advances in ML, pushing the limits of various tasks that use unstructured data such as image recognition [55, 138, 124], machine translation [43, 146, 54], and speech recognition [22, 56]. One of the key factors of this success was the power of backpropagation, which allows the DNNs to learn and extract important higher-level features for the given task. DNNs comprise multiple layers, which can be seen as multiple cascaded operators. These layers are trained simultaneously using backpropagation by which parameters can be globally estimated end-to-end to reach better minima.

On the other hand, many real-world ML applications including recommen-

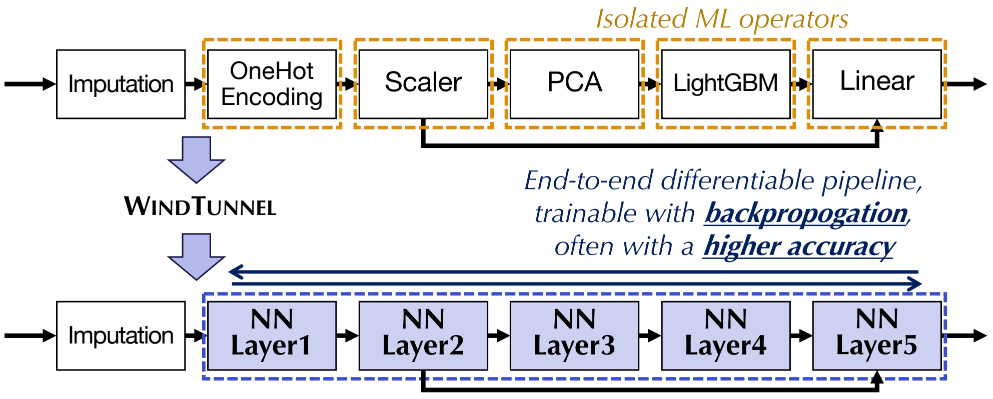


Figure 5.1: An illustration of WINDTUNNEL. The input to WINDTUNNEL is a ML pipeline and the output is its (partially) differentiable counterpart.

dation [19, 28, 119, 120], click prediction [67, 51, 68], and malware prediction [78, 100, 66] use structured data, which is often represented in a tabular form within RDBMSs. These applications often use *classical* [1] *machine learning pipelines* composed of multiple data transformations and ML models [21, 107, 140, 134] rather than a single model. Such pipelines are Directed Acyclic Graphs (DAGs) of operators and are enriched by domain knowledge from practitioners and domain experts via feature engineering and model selection. However, these pipelines are trained sequentially by following the topological order specified in the DAG. They are not end-to-end differentiable, thus cannot take advantage of backpropagation in jointly optimizing the whole pipeline beyond a single model.

Inspired by these observations, in this work we ask: *Can we combine the strength of backpropagation and ML pipelines?* To answer this question, we propose WINDTUNNEL, a framework that translates operators of a given ML pipeline into differentiable Neural Network (NN) modules. The translated NN

[1]The term "classical" is generally used to diversify this type of ML from DNN-based approaches.

modules are wired together to form a WindTunnel pipeline (Figure 5.1), hence enabling end-to-end training via backpropagation. This allows us to bypass the greedy one-operator-at-a-time training scheme and boost the accuracy of the pipeline. During the translation phase, we can retain the information already acquired by training the original ML pipeline and provide a useful parameter initialization for the translated NN modules, making further training of WindTunnel pipeline more accurate and faster. Neural translation also enables GPU-acceleration over ML pipelines without reinventing the wheel (i.e., support for hardware acceleration) for classical ML frameworks.

To demonstrate the benefits brought by WindTunnel, we conduct experiments on three large-scale real-world datasets with three ML pipelines made up of multiple operators. The results show that we can arrive at better accuracy by jointly training these operators. Furthermore, we find that WindTunnel provides informative knowledge transfer from pre-trained pipelines, along with efficient neural architecture that performs better than previous work [73]. WindTunnel currently supports several among data transforms and ML models (the full list is contained in Table 5.1).

Comparison with the state of the art approaches. WindTunnel has the following benefits compared to other approaches.

1. Compared with the original ML pipeline that cannot optimize multiple operators in an end-to-end fashion, a WindTunnel pipeline has higher accuracy because of its ability to jointly fine-tune the pipeline with backpropagation. While jointly optimizing multiple operators, WindTunnel is also able to maintain the knowledge encoded in the structure of the original pipeline by experts, such as how input features are wired to operators and hyperparameters of the operators (how many trees in a Gradient Boosting Decision

Tree (GBDT) model, the number of principal components for PCA, etc.).

2. Compared with DNNs, WindTunnel leads to a higher accuracy because ML pipelines are often better than DNNs for handling tabular data, and WindTunnel successfully leverages such advantage in the translation. For example, Ke et al. [73] compared the performance of ML pipelines with various DNNs developed for tabular data including Wide&Deep [34], DeepFM [53], and PNN [112], and showed that the ML pipeline with LightGBM [72] outperforms all the DNNs for every dataset. Rendle et al. [121] also showed that Matrix Factorization [80] can outperform recent DNN-based approaches. One can try to manually design a DNN that matches the neural architecture of the WindTunnel pipeline, however, the DNN should be trained from scratch while WindTunnel provides an informative initialization point by transferring weights from the original ML pipeline.

Multi-operator pipeline vs. Single model. At this point, the readers might wonder: *What's the difference between composing a ML pipeline with multiple operators and a single model? Can't we just replace the multi-operator pipeline with a single model?* We have evidence that this is not the current trend in data science. For instance, in [111] we crawled 6 million python notebooks on GitHub and joined this information with telemetry data on the internal usage within Microsoft of ML.NET [21]. The analysis suggested that the majority of Scikit-learn [107] pipelines used in public notebooks contain 2 or more operators (with a max length of 43), whereas in ML.NET telemetry the distribution is even more tail-skewed, with few pipelines having even up to hundreds of operators. This evidence suggests that multi-operator pipelines are widely used in practice both in the open-source domain and in industry. We attribute such trend to the additional information encoded by experts in the structure of the pipeline, including how to featurize the input data and how to wire the connection between operators.

Challenges of translating non-differentiable operators. Nevertheless, noticeable challenges arise when the pipeline involves operators that are intrinsically non-differentiable, such as decision trees or word tokenization. This requires us to develop new methods in translating non-differentiable operators into differentiable NN modules. To address this challenge, we develop translation methods for a selected set of non-differentiable operators. First, we propose a translation method that translates tree ensemble (e.g., GBDT) into a batch of Multi-Layer Perceptrons (MLPs), where each MLP corresponds to a tree in the ensemble. The translated NN module (i.e., batch of MLPs) directly inherits the decision procedure of the original tree ensemble, thus the learning capacity of the NN module varies according to hyperparameters of the ensemble like number of trees. Leveraging this additional knowledge infused by the ML experts relieves the burden of laborious neural architecture tuning. We also suggest multiple parametrization levels when optimizing the translated module to balance good fit and inductive bias.

For categorical features, we translate categorical feature encoders (e.g., one-hot encoding) into embedding lookup modules. By doing so, WindTunnel learns the dense representations of sparse categorical features by exploiting the information propagated from the final loss function. The translated embedding module inherits the data transformation procedure of the original encoder, following the same principle as GBDT translation. Conversely, the original ML pipeline uses a fixed encoding logic *regardless of the final prediction result.* To the best of our knowledge, this is the first work that proposes joint optimization of categorical encoders and downstream operators (e.g., GBDT) in ML pipelines. Although the embedding technique itself is well-recognized in ML community especially in the context of deep learning [108], combining classical ML models with the embedding technique were not possible without explicit use of models with latent parameters [119]. This is because the ML models did not allow backpropagating gradients to upstream operators, while the neural translation unlocks this capability.

Practical impact. WindTunnel will be open sourced as part of Hummingbird [97, 81]: a tool recently released [141] by Microsoft enabling inference of classical ML pipelines over hardware accelerators (e.g., GPUs). Hummingbird converts ML pipelines into non-differentiable tensor computations and thus can directly leverage the capabilities of DNN runtimes [104, 33]. Hummingbird is part of the PyTorch ecosystem [143], and is integrated with ONNXML-Tools [142]. WindTunnel extends Hummingbird by enabling conversion of pipelines into differentiable modules, and therefore allowing the fine-tuning of pipelines along with fast inference. Enabling training of ML pipelines over DNN runtimes and hardware accelerators has been suggested as one of the important extension to Hummingbird both from the open-source community[2] and internal conversations with product partners within Microsoft.

Limitations. As one of the first systems that focus on the differentiable translation of ML pipelines, WindTunnel by no means provides a complete solution to this challenging problem. One major limitation is that there are some operators that we cannot translate into a differentiable format yet. Word tokenization, data cleansing, and imputation are such examples. These operators require sophisticated algorithms that are too difficult to parametrize.

[2]`https://github.com/microsoft/hummingbird/issues/165`

Table 5.1: WindTunnel's currently supported ML operators.

Supported Operators			
linear models	normalizers	categorical encoders	
SVM	PCA	LDA	KMeans
naive bayes	random forest	gradient boosting trees	
matrix factorization		factorization machine	

Since we currently do not handle these operators, WindTunnel does not translate them and keep them as they are. Nevertheless, in all the cases we studied, these non-translatable operators are placed at the beginning of the pipeline and do not affect backpropagation through the rest of the translated pipeline. Hence, we can still compute gradients and jointly optimize the downstream operators, which are the more essential parts of the ML pipeline.

Another notable limitation is that WindTunnel requires more training time than classical ML pipelines. In particular, the fine-tuning stage is more than an order of magnitude slower than the training of the original ML pipeline. This is due to the large amount of computation required for optimizing the NN modules. One can consider using recent hardware that enables faster GEMM [136] or applying distributed training techniques for sparse parameters [76] to mitigate this problem. We leave these directions as a future work.

5.2 Pipeline Translation

A classical machine learning pipeline is defined as a DAG of data-processing operators, and these operators are mainly divided into two categories: (1) the *arithmetic* operators and (2) the *algorithmic* operators. Arithmetic operators are typically described by a single mathematical formula. These operators are, in turn, divided into two sub-categories of *parametric* and *non-parametric* operators. Non-parametric operators define a fixed arithmetic operation on their inputs; for example, the logistic sigmoid function can be seen as a non-parametric arithmetic operator. In contrast, parametric operators involve numerical parameters on the top of their inputs in calculating the operators' outputs. For example, an affine transform is a parametric arithmetic operator where the parameters consist of the affine weights and biases. The parameters of these operators can be potentially tuned via some training procedure.

The algorithmic operators, on the other hand, are those whose operation is not described by a single mathematical formula but rather by an algorithm. For instance, the one-hot encoder that converts categorical features into one-hot vectors is an algorithmic operator that mainly implements the look-up operation. Given a DAG of arithmetic and algorithmic operators, we propose the following procedure for translating it into a differentiable format:

1. For an arithmetic operator, translate the mathematical formula into a neural network (NN) module. In the case of parametric operator, copy the values of the operator's parameters into the resulting NN module.

2. For an algorithmic operator, translate the operator by rewriting the algorithm as a differentiable operation.

3. Compose all the resulting modules into a WindTunnel pipeline by following the dependencies in the original pipeline.

The final output of the above translation process is a pipeline of NNs that provides the same prediction results (unless the translation includes approximation described in Section 5.2.2) as the original pipeline. Note that Step 1 and 2 in the above procedure are where the actual translation happen, and will be described in details next.

5.2.1 Translating Arithmetic Operators

Arithmetic operators comprise non-parametric and parametric operators, and it is straightforward to translate the former into a NN module: the mathematical function of the operator can in fact be directly rewritten using the math API provided by a DL framework like PyTorch [104]. On the other hand, parametric operators are often implicitly derived from ML models[3], which are not straightforward to translate. ML models typically consist of three key components: (1) the prediction function, (2) the loss function, and (3) the learning algorithm. While the prediction function defines the functional form of the model, the learning algorithm and the loss function define how it is trained toward what objective, respectively. Take the popular linear Support Vector Machine (SVM) as an example: the prediction function is a linear combination of input features; the loss function is the Hinge loss, and the learning algorithm is gradient descent in the dual space.

A crucial observation is that once the training is complete, the data-processing logic of any ML model can be completely defined by the prediction function regardless of the loss function and the learning algorithm. Hence, we can translate a parametric operator derived from a ML model by applying the translation method for non-parametric operators to the model's prediction function and properly initializing the parameters. For example, a linear SVM can be translated into a linear NN module of one output unit having the weights transferred from the trained SVM. It is worth noting that the translation of a ML pipeline into a pipeline of NN modules is uniquely done starting from the data-processing logic (i.e., prediction function in case of parametric operator derived

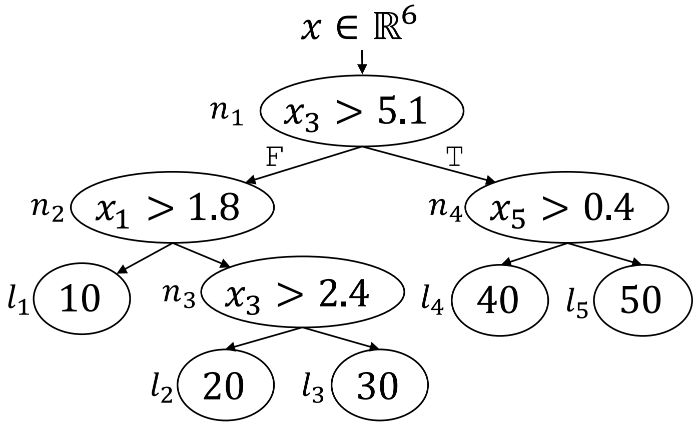
[3]Some parametric operators are not derived from ML models (e.g., normalizer). Still, these operators can be translated using the same mechanism for parametric operators derived from ML models.

from ML model), independently on how different parts of the ML pipeline have been trained. This enables us to translate different operators of a pipeline using the same formalism even though they might have been obtained via different learning algorithms or objectives.

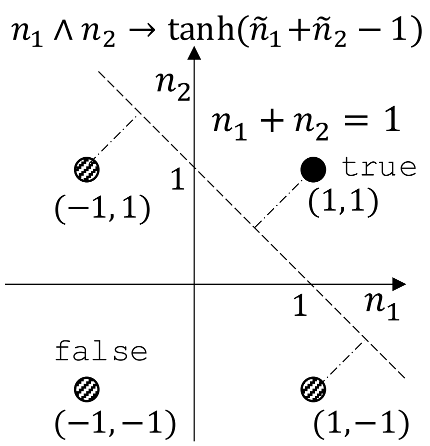
5.2.2 Translating Algorithmic Operators: GBDT

While most ML models correspond to arithmetic operators that can be directly translated, some do not. One prominent example is GBDT whose prediction function is not differentiable. Instead, each prediction of a GBDT model is made by executing a sequence of `if-else` statements for each tree and computing the mean over the trees. In that respect, GBDT's prediction function is an algorithmic operator rather that an arithmetic one, which means we cannot use backpropagation. In order to jointly optimize GBDT with other operators, we should rewrite its prediction function as a differentiable function of tunable parameters. We use GBDT as a running example here because they are widely used in practical data science [111]. Naturally, the same approach applies over any tree-based model (e.g., decision trees, random forests, etc.).

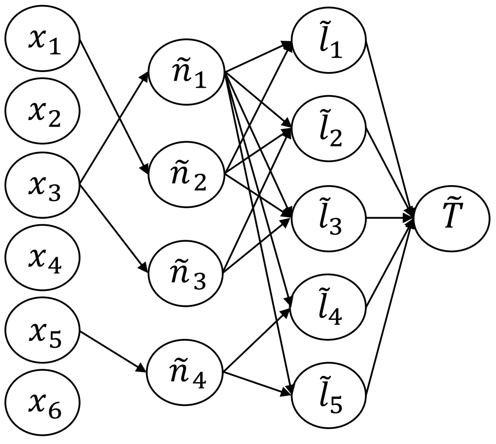
We introduce parameters that fully determine GBDT's prediction function, and smooth the non-differentiable points of the function so that it can be differentiated. At a given internal node n of a binary decision tree of GBDT, the prediction function evaluates a boolean-valued function $n(x) = x_{i(n)} > \theta_n$, where x is a vector representing the input of the tree, $i(n)$ is the index of the feature examined at node n, and θ_n is the decision threshold at node n. We smooth this non-differentiable function by making the function output a real number, $\tilde{n}(x) = tanh(\frac{1}{\tau}(x^T e_{i(n)} - \theta_n))$, where $e_{i(n)}$ is the canonical basis vector along the $i(n)$-th dimension of the feature space and τ is a temperature parameter. If $n(x)$ is `true`, $\tilde{n}(x)$ is close to 1, otherwise $\tilde{n}(x)$ is close to -1. We



(a) An example decision tree.



(b) Logical cunjunction.



(c) A MLP translated from the decision tree of Figure 5.2a.

Figure 5.2: Translating a decision tree into a multi-layer perceptron.

set τ as 1 throughout this work. As we employ smaller τ, the differentiable approximation becomes steeper and degenerates into the original boolean-valued function.

Next, we note that the value of a leaf node is outputted as the final value of a tree if and only if the path from the root node to that leaf node is traversed. For example, in Figure 5.2a, the tree will output 30 (i.e. the value of leaf l_3) iff $n_1(x)$ is `false`, $n_2(x)$ is `true`, and $n_3(x)$ is `true`. As such, we denote the leaf activation function of l_3 as a conjunction of l_3's ancestors: $l_3(x) = \neg n_1(x) \wedge n_2(x) \wedge n_3(x)$. To get a differentiable approximation of the logical conjunction, we write $\tilde{l}_3(x) = tanh\left(\frac{1}{\tau}\big(- \tilde{n}_1(x) + \tilde{n}_2(x) + \tilde{n}_3(x) - C_{l_3} + 1\big)\right)$, where C_l is the total number of literals in the conjunction (the path length from the root to the leaf node l; e.g., $C_{l_3} = 3$). Figure 5.2b visualizes this approximation for 2 inputs. The equation $n_1 + n_2 = 1$ is a maximum-margin hyperplane between `true` and `false` evaluations of $n_1 \wedge n_2$. In the case of no approximation (i.e., $\tau \to 0$), one and only one of the leaf activation functions $\tilde{l}(x)$ evaluates to 1 for any given input x, while the rest are -1.

Having translated the function of internal and leaf nodes into the smooth functions described above, any decision tree $T(x)$ can be translated into a MLP $\tilde{T}(x)$ with two hidden layers. Figure 5.2c shows an example of this translation procedure. The first hidden layer implements a hidden unit $n(x)$ per each internal node. The second hidden layer allocates a hidden unit $l(x)$ for each leaf node. Finally, the output layer is defined as a linear layer with one unit, $\tilde{T}(x) = \sum_{l_i \in L} \frac{\nu_i}{2}(1+\tilde{l}_i(x))$, where L is the set of all leaf nodes and ν_i is the value of the leaf node l_i. Translation of GBDT or Random Forest follows directly by computing the average of $\tilde{T}(x)$ over the trees. We batch the computation of multiple MLPs using variants of `gemm` such as `baddbmm` and `addbmm`. Since each MLP only has tens (or one to two hundred) of hidden units, we cannot fully utilize the computation power of modern GPUs without batching them.

Note that when smoothing boolean-valued functions, we map `false` evaluation of $n(x)$ and $l(x)$ to a real number close to -1, not 0. Suppose we map `false` to a value close to 0 and use dropout [135] when training the translated module. In this case, dropping a `true` neuron by zeroing its output can be seen as flipping the evaluation from `true` to `false`, introducing unintended bias. Instead of using the logistic sigmoid function to map `false` to a number close to 0, we use *tanh* for smoothing the boolean functions to make the dropped neurons unbiased, meaning neither `true` nor `false`.

Once the translation is complete, the question is which of the parameters should be declared as trainable. We suggest four levels of parametrization to balance good fit and inductive bias:

L1: The weights and biases for computing the output layer $\tilde{T}$, initialized using leaf node values ν, are declared as trainable.

L2: In addition to L1, the biases for computing the first hidden layer $\tilde{n}$, initialized using the decision threshold values θ's at the internal nodes, are declared as trainable.

L3: In addition to L2, the weights for computing the first hidden layer $\tilde{n}$, initialized using the canonical basis vectors $e_{i(n)}$ in the equation of $\tilde{n}(x)$, are declared as trainable.

L4: In addition to L3, the weights (including the non-existing zero weights) and biases for computing the second hidden layer $\tilde{l}$ are declared as trainable.

As level number increases, we declare more parameters as trainable and as such increase the capacity of the MLP to fit to data better. While L1 and L2 can only change the leaf and the decision threshold values in the tree, L3 can additionally lead to examining a linear combination of features at each

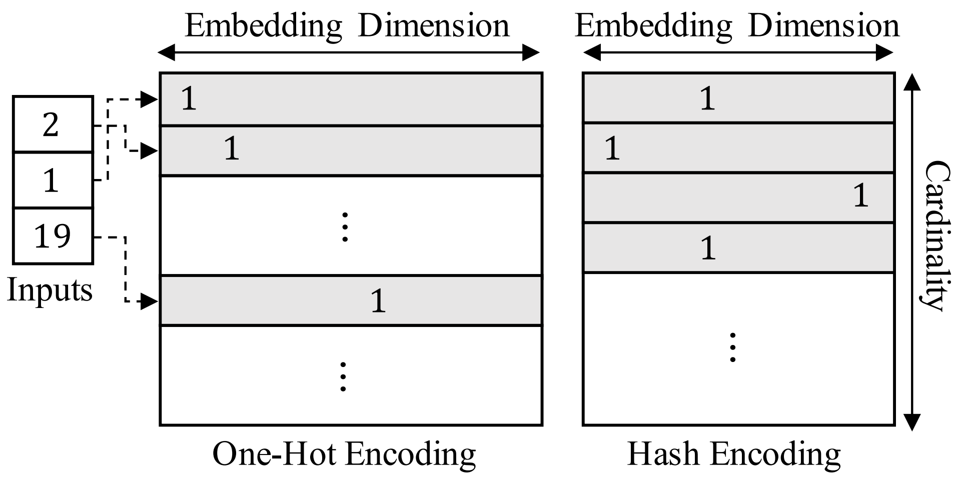


Figure 5.3: Translating one-hot encoder and hash encoder into embedding lookup modules.

internal node rather than a single feature. Up to L4, the decision structure that determines whether or not to activate a leaf node by examining internal nodes on the path from the root to the leaf is preserved; whereas, at L4, we let this decision structure change. That is, L4 gives us a fully-connected and fully-trainable MLP initialized by a decision tree. This level can be disabled if, for example due to some governance constraints, the decision structure must be maintained for explainability.

5.2.3 Translating Algorithmic Operators for Categorical Features

Classical ML pipelines often convert categorical features into numerical values using non-differentiable, algorithmic operators. The simplest yet most popular technique is one-hot encoding [94], which generates sparse one-hot vectors out of categorical inputs. This operator is intrinsically non-differentiable since its inputs lie on discrete spaces such as integer or string.

Our observation is that one-hot encoding consumes raw categorical inputs, which means that we do not have to backpropagate further through its discrete inputs due to the absence of upstream operators. We adopt the embedding technique that has been studied intensively by the machine learning community. As shown in Figure 5.3, one-hot encoding can be seen as an embedding vector lookup operation with the embedding dimension matching the cardinality of categories. We can declare this embedding matrix as trainable in order to replace sparse one-hot vectors with dense representations and learn relationship between different categorical features, which is not possible with one-hot encoding. The same statement holds for hash encoding [93], except that data scientists can control the size of embedding dimension explicitly. Although in the original hash encoding we may have collisions between different categories, we have one row in the embedding matrix for each input category after translation. This means that even if the resulting vectors are equivalent, the "collision information" is actually stored into the embedding matrix. For example, the hash encoder in Figure 5.3 initially maps the first and fourth category to equivalent one-hot vectors, but they are in two different rows of the embedding matrix, and therefore they will be eventually trained differently.

By translating categorical encoders into embedding modules, we can use the well-recognized embedding technique along with arbitrary ML operators, which was not possible before. There indeed exist ML algorithms such as Matrix Factorization [80] and Factorization Machine [119, 68] that learn latent factors, which are conceptually equivalent to embedding parameters. Yet, the adoption of latent factors using these algorithms requires the ML operator to use a specific form of prediction function that explicitly models two-way interaction between features [119], which may not be desirable for certain type of tasks. In contrast, WindTunnel can combine embedding modules with arbitrary ML models, allowing data scientists to use any prediction function they want without restriction.

5.2.4 Fine-Tuning

After translating the operators into NN modules, one can jointly optimize the trainable parameters of the translated pipeline via backpropagation. We refer to this training process as *fine-tuning*. There are many scenarios for which this fine-tuning step can be useful. First, by fine-tuning the resulting pipeline on the original training data, we can potentially improve the generalization of the model since we are now jointly optimizing all the operators of the pipeline toward the final loss function. We empirically demonstrate this in Section 5.4. Second, as we discussed in Section 5.2.1, the translation process does not depend on the loss functions that different operators have been trained toward before. This means that once the translation is complete, the resulting pipeline can be fine-tuned toward a completely different objective that is more suitable for a given application. Third, fine-tuning can be used to adapt the model to new data that were not available before, which is not straightforward without re-training the original ML pipeline with the old and new data [73]. It is worth noting that other methods for fine-tuning such as boosting may increase the model size and complexity, while WindTunnel does not. Also, the ensemble model obtained by boosting can be seen as a pipeline containing multiple models that were not jointly optimized, so it can also benefit from our translation approach.

5.3 Implementation

Based on the translation mechanism described in Section 5.2, we have implemented prototypes of WindTunnel on different classical ML libraries (i.e., scikit-learn and ml.net). WindTunnel design is in fact simple, flexible, and easy to extend. The main component of WindTunnel is a *pre-defined mapping table* between the supported operators (listed in Table 5.1) and neural network modules implemented in PyTorch [104]. On top of the mapping table, WindTunnel provides a set of *converters* for extracting information from the trained ML operator and materialize the information into parameters of the corresponding NN module. We have different converters based on the ML framework the input pipeline was authored in. For the experiments in Section 5.4, we use scikit-learn [107] and PyTorch [104] for implementing ML pipelines.

During the translation process, WindTunnel refers to the proper converters and mapping table entries, and replaces the operator in the original pipeline with its differentiable counterpart. For the operators that we do not support, we either (1) cache the operators' outputs and reuse them in the fine-tuning stage; or (2) if streaming execution is provided by the ML framework, we stream data into the untouched operators and redirect their outputs to the WindTunnel pipeline. Caching is in general possible because the unsupported operators are often placed at the beginning of the pipeline thus their outputs do not change. We can consider this as a separate data pre-processing step, which is typically done before actual training.

We are currently working on adding the pre-defined mapping table containing the PyTorch implementations to Hummingbird[4].

5.4 Experiments

In this section, we empirically evaluate the performance of WindTunnel. The main goal of the experiments is to show that we can improve the performance of ML pipelines by joint optimization instead of training each operator individually. We carry our experiments on binary classification tasks for three tabular

[4]`https://github.com/microsoft/hummingbird/tree/mainterl/fine-tune-trees`.

Table 5.2: Statistics of datasets used in experiments. #Rec is the number of data records, #Num is the number of numerical features, #Cat is the number of categorical features, #Unq is the number of unique categories that appear in the training split (i.e. the sum of cardinalities of categorical features), and Positive ratio is the percentage of records with positive label.

Dataset	#Rec	#Num	#Cat	#Unq	Positive ratio
Flight	21.6M	2	6	694	20.4%
Avazu	40.4M	0	23	8.93M	17.0%
Criteo	45.8M	13	26	30.8M	25.6%

datasets. We start with details about experimental setup, such as dataset description, pipeline composition, and training configurations.

5.4.1 Experimental Setup

Datasets. We conduct experiments on real-world datasets listed in Table 5.2. The Flight [102] dataset is used for predicting whether a scheduled flight will be delayed more than 15 minutes inclusive. We use records from the year of 2006 and 2007 as training set (about 14M records), while the records from the year of 2008 are divided into two splits and used as validation set (Jan to Jun) and test set (Jul to Dec). The Avazu [24] and Criteo [39] datasets are from Kaggle competitions that call for click-through rate prediction models. We use the first 90% of the Criteo dataset as training set, while the next 5% and the last 5% is used as validation and test set, respectively. For the Avazu dataset, we use the same ratio for splitting the dataset after a random shuffling step, to ensure that the distribution of "day of week" feature is consistent between train-validation-test splits.

Data pre-processing. We experiment with three different data pre-processing schemes to show both the general applicability of WindTunnel, as well as the impact of different pre-processing operations over the embedding dimensions and final accuracy. The first pre-processing scheme (Pre1) drops categories that appear less than 25 times in the training set to reduce noise, followed by a binary encoder for handling categorical features. In the second pre-processing scheme (Pre2), we replace the binary encoder with a two-hot encoder, while the rest are left the same as the first scheme (Pre1). Two-hot encoder is similar to the well-known one-hot encoder [94], except that there are two "hot" elements (value of 1) in the resulting vector. Switching the pre-processing step from Pre1 to Pre2 allows us to test how accuracy changes when increasing the embedding dimensions. The last, most complex scheme (Pre3) is composed as follows: (1) drop the lower 1% categories by frequency; (2) drop categories that appear less than 10 times in the training set; (3) add additional categorical features by bucketizing numeric features using 32 bins; (4) apply two-hot encoding and target encoding [110]. We take inspiration from previous literature [73] for designing Pre3.

ML pipelines. We evaluate the performance improvements using three ML pipelines. Each pipeline not only serves as the source pipeline for WindTunnel translation, but also serves as a baseline for comparison. In the first pipeline (Pipe1), we use a logistic regression model following the pre-processing scheme. After translation, WindTunnel jointly optimizes the embedding modules translated from categorical encoders and the logistic regression module. The second pipeline (Pipe2) employs a GBDT model trained by LightGBM [72] after the pre-processing scheme. The third pipeline (Pipe3) is composed as follows: (1) process the data using the described pre-processing scheme; (2) train a LightGBM model with the processed data and label; (3) for each tree in the trained LightGBM model, create a one-hot vector that marks the index of the activated leaf as 1 and keeps others 0 using the leaf activation function $l(x)$ (see Section 5.2.2); (4) train a Factorization Machine [119] model with the output from (3) and label. We take inspiration from the winning solution of Kaggle competition [67] for designing Pipe3. These pipelines cover the most common ML algorithms (linear models and decision-tree variants) used in practice [69].

DNN baseline. In addition to the three ML pipelines described above, we compare DeepGBM [73], a state-of-the-art neural network that takes advantage of classical ML by distilling knowledge from gradient boosting machine. We do not compare with other DNN-based models because Ke et al. [73] already demonstrated that DeepGBM is consistently better than Wide&Deep [34], DeepFM [53] and PNN [112]. We also do not report results from ML pipelines using a single operator because they fall far behind other models.

Configurations. We set the LightGBM to create 64 leaves for each tree, and we construct 100 trees for all experiments that use LightGBM. The Factorization Machine (FM) model uses a latent dimension of 20 for all experiments. We set learning rate to 0.25 and 10^{-3} for training LightGBM and FM, respectively. Regarding the training of WindTunnel pipelines, we use the parametrization level L4 for GBDT-translated modules unless otherwise noted. Dropout [135] is applied to each NN layer of WindTunnel pipeline, with a zeroing probability of 0.1. We use the Adam [77] optimizer with a batch size of 4096 and weight decay of 10^{-6} for all experiments. Learning rate is set to 10^{-4} for the Flight and Avazu dataset, and 10^{-5} for the Criteo dataset. We select these rates by sweeping a grid of $\{10^{-2}, 10^{-3}, \cdots, 10^{-6}\}$ for learning rate and $\{10^{-5}, 10^{-6}, 10^{-7}\}$ for weight decay. We let the training process run until convergence. Regarding the experiments using DeepGBM, we use an open-source implementation. For the

Flight and Criteo dataset, we use the hyperparameter setting described in the original literature. Since the literature did not use the Avazu dataset, we set the hyperparameters same as Criteo's.

Table 5.3: Overall performance comparision. We report AUC on test split following the previous work [73]. ML is the original ML pipeline, while W.T. is for WindTunnel. PreX means different preprocessing schemes, and PipeX denotes different ML pipelines. The best result is marked bold.

Model	Flight			Avazu			Criteo		
	Pre1	Pre2	Pre3	Pre1	Pre2	Pre3	Pre1	Pre2	Pre3
ML (Pipe1)	0.6783	0.6847	0.7126	0.6896	0.7264	0.7553	0.7167	0.7442	0.7769
ML (Pipe2)	0.7358	0.7427	0.7507	0.7521	0.7550	0.7718	0.7739	0.7781	0.7925
ML (Pipe3)	0.7519	0.7547	0.7467	0.7597	0.7616	0.7728	0.7838	0.7884	0.7954
DeepGBM	0.7793	0.7695	0.7726	0.7682	0.7680	0.7760	0.7965	0.7918	0.7972
W.T. (Pipe1)	0.6913	0.6910	0.7244	0.7588	0.7589	0.7637	0.7750	0.7804	0.7903
W.T. (Pipe2)	0.7790	0.7897	0.7960	**0.7753**	**0.7742**	**0.7763**	0.8006	0.8053	0.8041
W.T. (Pipe3)	**0.7829**	**0.7906**	**0.7989**	0.7746	0.7718	0.7753	**0.8014**	**0.8058**	**0.8048**

5.4.2 Overall Performance

We first evaluate the overall performance of WindTunnel. The comparison results can be found in Table 5.3. As we can see, WindTunnel greatly improves AUC of the original pipeline by jointly optimizing the ML operators which were trained separately, providing up to 10.0% higher AUC. This demonstrates the power of end-to-end training and WindTunnel's ability to leverage such advantage. WindTunnel also surpasses DeepGBM by a significant margin (up to 3.4%) for all cases, except the Avazu dataset using Pre3 pre-processing where the margin is small (0.7760 vs. 0.7763). We credit the AUC gap to joint training of embedding modules (translated from categorical encoders) and downstream modules (translated from GBDT and FM), which is not possible in DeepGBM.

Impact of data pre-processing. We study the impact of using different pre-processing schemes, and find that it can largely affect the AUC of both the original pipeline and WindTunnel pipeline. First of all, the use of one-hot encoder [94] is not suitable for large-scale dataset because it requires too much host and GPU memory and computation power. The number of embedding dimension grows linearly with the number of unique categories (denoted as C), which makes training of both the original pipeline and WindTunnel pipeline extremely difficult. On the other hand, for the binary encoder [92] and two-hot encoder, the minimum required size of embedding dimension is roughly log_2C and $\sqrt{C}$, respectively.

Second, Pre2 shows better AUC on all cases using ML pipelines compared to Pre1. Similarly, WindTunnel pipelines tends to work better with Pre2 than Pre1. This means that both ML and WindTunnel pipelines prefer two-hot encoding to binary encoding. We attribute this trend to the high separability of two-hot vectors compared to binary vectors at the cost of larger embedding dimension. Note that DeepGBM prefers Pre1 to Pre2 because it explicitly selects top-N elements of the input vector based on the amount of information computed by LightGBM and use only them for training the GBDT-distilled neural network. Due to this feature selection policy, when we use two-hot encoder that produces larger encoded vector, some elements of the vector with meaningful information are dropped, thus resulting in worse AUC compared to binary encoder.

If we go one step further and compare Pre2 and Pre3, Pre3 shows better AUC in most cases. This means that if we carefully design the pre-processing scheme and adopt more sophisticated feature engineering, we can achieve better results compared to using simple, less-optimized pre-processing scheme.

Discussion. From this experiment, we notice the importance of getting a proper pre-processing scheme. We deem developing neural translation of the pre-processing operators a promising direction able to improve the performance. This also aligns with recent trends in computer vision domain that learns to augment input images by parametrizing and tuning the pre-processing scheme [40, 88, 75]. Developing new types of pre-processing operators that are naturally tunable (e.g., embedding) could also be an alternative solution.

5.4.3 Ablation Study

Next, we evaluate the performance of WindTunnel with varying configurations. As a representative for the datasets we study, we select the largest one (i.e, Criteo) and conduct experiments on it. We also focus on the settings with Pre2 & Pre3 schemes and Pipe2 & Pipe3 pipelines, because we produce top results with these settings.

Table 5.4: AUC of the Criteo dataset using different translation scopes.

Model	Pre2	Pre3
ML (Pipe2)	0.7781	0.7925
GBDT2NN (Pipe2)	0.7962	0.7998
W.T. (Pipe2)	0.8053	0.8041

Table 5.5: AUC of the Criteo dataset using different parameter initialization regimes.

Model	Pre2		Pre3	
	Cold	Warm	Cold	Warm
W.T. (Pipe2)	0.7983	0.8053	0.7990	0.8041
W.T. (Pipe3)	0.7966	0.8058	0.7975	0.8048

Joint optimization. We study the impact of joint optimization in more details. Table 5.4 reports additional results by translating only the GBDT model of Pipe2, not categorical encoders (denoted by "GBDT2NN"). Note that a GBDT model is already an ensemble of multiple trees, so translating only the GBDT model (GBDT2NN) also employs joint optimization of multiple MLP modules. From the results, we can observe a clear pattern that as the scope of translation gets wider, the AUC increases (ML < GBDT2NN < WindTunnel). This further supports the claim that joint optimization is a promising technique able to improve the performance. In particular, the gap between GBDT2NN and WindTunnel suggests that the neural translation of pre-processing operators (categorical encoder) is indeed effective in improving the accuracy.

Parameter initialization and architecture. In this set of experiments, we show that the translation of trained ML operators provides informative initialization of WindTunnel pipelines. We experiment with two regimes of initialization for the parameters of WindTunnel pipeline: (1) in the *cold start*

Table 5.6: AUC of the Criteo dataset using different GBDT parametrization levels and dataset sizes (1%, 10%, 100%).

Model	100%	10%	1%
ML (Pipe2)	0.7781	0.7777	0.7657
W.T. (Pipe2, L2)	0.7924	0.7874	0.7685
W.T. (Pipe2, L3)	0.8051	0.7912	0.7699
W.T. (Pipe2, L4)	0.8053	0.7906	0.7691

regime the parameters are randomly initialized (denoted by "Cold"); and (2) in the *warm start* regime the parameters are carried over from the original ML pipeline (denoted by "Warm", used as default setting for other WindTunnel experiments).

Table 5.5 shows that the warm start outperforms the cold start, which means that the parameters extracted from the original ML pipeline provide an informative initialization for WindTunnel. Interestingly, the cold start regime performs better than DeepGBM. This shows that WindTunnel not only delivers meaningful information by parameter initialization, but also provides a good neural architecture that can achieve better results than the baseline.

GBDT parametrization level and dataset size. As described in Section 5.2.2, the proposed translation of GBDT increases the capacity of the model if we adopt higher parametrization levels (L3 or L4). We evaluate the effect of the parametrization level in Table 5.6, using the combination of Pre2 and Pipe2. From the results, we can observe the significant gap between L2 and L3, verifying that the extra capacity helps improving the performance. Yet, L2 still outperforms the original ML pipeline, due to WindTunnel's ability to jointly optimize the categorical encoders and GBDT. We also evaluate trade-offs between flexibility and inductive bias come from different levels. For this, we use subsets of the training split with various sampling ratios (1%, 10%, 100%). As the subset size decreases, the gap between L2 and the other two level closes. This trend shows overfitting of L3 and L4 in small data experiments and how it is avoided by L2 that has much smaller capacity. In other words, lower levels provide a natural regularization mechanism in small data experiments. The results also show that the gap between the original pipeline (ML) and WindTunnel gets wider as the subset size increases. This suggests that WindTunnel has better scalability (in terms of accuracy) than the original pipeline by exploiting the extra model capacity that comes from the joint optimization.

5.5 Summary

Inspired by the existing gap between classical ML pipelines and neural networks, we propose WindTunnel, a framework for translating pipelines of ML operators into neural networks and further *jointly* fine-tuning them. As part of the translation procedure, we also propose techniques for translating popular non-differentiable operators including GBDT and categorical encoders. The experimental results show that the translation with knowledge transfer followed by the fine-tuning leads to significant accuracy improvements over the original pipeline and state-of-the-art NNs. Furthermore, we see that our translation mechanism can be seen as an approach for designing neural network architectures for a given task that is inspired by the classical ML pipeline structure for that task. We deem this work as a step towards filling the gap between classical ML pipelines and neural networks over tabular data.

Chapter 6

Related Work

Optimizations for GPU task scheduling. The core ideas of Nimble can be compared with some previous works. First, in an attempt to reduce the scheduling overhead, TensorFlow recently introduced a new runtime [15] that has a thin operator dispatch routine. While redesigning a runtime stack costs tremendous engineering efforts, the AoT scheduling of Nimble provides an automated way to avoid the scheduling overhead. Second, although the pre-run process of Nimble is similar to the tracing of TorchScript, they differ in the purpose and the target of tracing process. In the tracing of TorchScript, DL operator calls are recorded to construct a computation graph, which is used for serialization and graph-level optimization. Meanwhile, Nimble records GPU tasks during the pre-run process to perform the scheduling procedure once. Lastly, in comparison to HiveMind [99] that has a parallel runtime for multi-model workloads, the multi-stream execution of Nimble parallelizes operators in a single model, using a more sophisticated algorithm.

Fine-grained batching for recurrent models. We would like to highlight BatchMaker [50] as one of the most relevant previous works of Orca. BatchMaker is a serving system for RNNs that performs scheduling and batching at the granularity of RNN cells, motivated by the unique RNN characteristic of repeating the same computation. Once a request arrives, BatchMaker breaks the dataflow graph for processing the request into RNN cells, schedules execution at the granularity of cells (instead of the entire graph), and batches the execution of identical cells (if any). Since each RNN cell always performs the exact same computation, BatchMaker can execute multiple RNN cells in a batched manner regardless of the position (i.e., token index) of the cell. By doing so, BatchMaker allows a newly arrived request for RNN to join (or a finished request to leave) the current executing batch without waiting for the batch to completely finish.

However, BatchMaker cannot make batches of cells for Transformer models because there are too many distinct cells (a subgraph that encapsulates the computation for processing a token; Figure 2.4) in the graph. Each cell at a different token index t must use a different set of Attention Keys/Values. As the cell for each t is different, the graph comprises L different cells (L denotes the number of input and generated tokens), significantly lowering the likelihood of cells of the same computation being present at a given moment (e.g., in Figure 4.8, L ranges from $33 = 32 + 1$ to $640 = 512 + 128$). Thus execution of the cells will be mostly serialized, making BatchMaker fall back to non-batched execution. BatchMaker also lacks support for large models that require model and pipeline parallelism.

While BatchMaker is geared towards detecting and aligning batch-able RNN cells, our key principle in designing Orca is to perform as much computation as possible per each round of model parameter read. This is based on the insight that reading parameters from GPU global memory is a major bottleneck in terms of end-to-end execution time, for large-scale models. Adhering to this principle, we apply iteration-level scheduling and selective batching to process all "ready" tokens in a single round of parameter read, regardless of whether the processing of tokens can be batched (non-Attention ops) or not (Attention ops).

DL compilers. There have been a body of works on the system-level optimization of DL inference and training. For example, DL compilers [14, 33, 41, 123, 145, 159] have been proposed to generate optimized codes for target hardware. These works take different approach from Nimble in that they aim to reduce the time spent on GPU tasks whereas Nimble tackles the inefficiencies in the scheduling of GPU tasks. DL compilers are also orthogonal to the contributions of ORCA. In fact, Nimble and ORCA also employ kernel fusion and kernel selection techinques, which are commonly used by DL compilers, as discussed in Section 3.4 and Section 4.4.

Specialized execution engines for Transformer models. The outstanding performance of Transformer-based models encourages the development of inference systems specialized for them. FasterTransformer [7], LightSeq [147], TurboTransformers [47] and EET [86] are such examples. Each of these systems behave as an backend execution engine of existing inference server systems like Triton Inference Server [11] and TensorFlow Serving [101]. That is, these systems delegate the role of scheduling to the inference server layer, adhering to the canonical request-level scheduling. Instead, ORCA suggests to schedule executions at a finer granularity, which is not possible in current systems without changing the mechanism for coordination between the server and the execution engine. Note that among these systems, FasterTransformer is the only one with the support for distributed execution. While systems like Megatron-LM [5] and DeepSpeed [3] can also be used for distributed execution, these systems are primarily optimized for large-scale training rather than inference serving.

Interface between inference servers and execution engines. Current general-purpose inference servers such as Triton Inference Server [11] and Clipper [38] serve as an abstraction for handling client requests and scheduling executions of the underlying execution engines. This approach is found to be beneficial by separating the design and implementation of the server layer and the engine layer. However, we find that the prevalent interface between the two layers is too restricted for handling models like GPT [27], which has the multi-iteration characteristic. Instead, we design ORCA to tightly integrate the server and the engine, simplifying the application of the two proposed techniques: iteration-level scheduling and selective batching. While in this work we do not study a general interface design that supports the two techniques without losing the separation of abstractions, it can be an interesting topic to explore such possibility; we leave this issue to future work.

End-to-end training of ML pipelines. Milutinovic et al. [96] proposes the end-to-end training of ML pipelines via propagating gradients across multiple differentiable operators. This work however has no discussion about non-differentiable operators, while in WINDTUNNEL we attempt to backpropagate through non-differentiable (e.g., GBDT) and non-trainable operators (e.g., categorical encoding). Additionally, this work requires users to manually write "backward" code for operators from non-NN libraries (e.g., scikit-learn [106]), while WINDTUNNEL exploits the automatic differentiation capabilities of DL libraries by neural translation.

Tree-based models and neural networks. There have been early works [128, 25, 63] that initialize parameters of a MLP by using a trained decision tree. However, these works have several limitations: (1) the resulting MLP cannot back-propagate gradients to upstream (or downstream) operators because the input and output layers are not designed with end-to-end training in mind; and (2) the MLP is not well-suited for adopting dropout [135] due to the use of logistic sigmoid activation that introduces bias. They also did not demonstrate generalizability on tree ensemble models like GBDT or Random Forest, and only experimented with a single decision tree that is unlikely used in practice.

DJINN [62] initializes a MLP in a different way, where the depth of the decision tree is used to decide the number of layers. Weights are randomly initialized, while the information on the tree is retained only for sparsely connecting the neurons. WindTunnel instead extracts more information from a GBDT model including tree structure and decision thresholds to initialize the parameters. DNDT [152] suggests to build tree-like neural networks for interpretability. This is different from our approach of handling trees because: (1) it builds a tree-like neural network using random weight initialization, while in WindTunnel we retain the behavior of trained trees by neural translation; and (2) WindTunnel's translated pipeline learns to use all features for making decision at each internal node (L3 and L4), while this work uses a single feature at each neuron and requires a wider network whose number of neurons grows exponentially as the number of features grows. dNDF [79] combines neural networks (CNNs) and decision tree classifiers by enabling backpropagation, with a focus on computer vision tasks. Similar to DNDT, dNDF also starts with a random initialization of tree parameters.

Finally, DeepGBM [73] *distills* trees into neural networks by transferring the knowledge of tree outputs and feature importance learned by GBDT. Given this distilled neural network, DeepGBM incorporates an additional embedding-based neural network called *CatNN* for handling categorical features only. For this, DeepGBM requires several hyperparameters such as the number of layers and hidden units (for both the distilled NN and CatNN), weights for controlling the strength between knowledge distillation and final loss, the number of features used for training the distilled NN, how to group trees for distillation, and so on. Instead, WindTunnel directly translates a ML pipeline so the structure of resulting pipeline solely depends on the structure of the original one.

Neural translation for fast inference. While in WindTunnel we focus on translating ML operators into differentiable modules for fine-tuning, in the Hummingbird project [97] we translate ML operators (including unsupported ones in WindTunnel) into tensor operations without requiring the operations to be differentiable. This allow us to run inference of end-to-end pipelines (1) completely on DL frameworks without any additional data conversion overhead; and (2) on hardware accelerators specialized for tensor operations.

Chapter 7

Conclusion

In this dissertation, we introduce recipes for efficient execution of ML workloads on GPU environment.

We first present Nimble, a high-speed DL execution engine for static neural networks. We show two problems of the run-time scheduling of GPU tasks: scheduling overhead and serial execution. Nimble minimizes the scheduling overhead by finishing the scheduling procedure ahead of time before executing the GPU tasks at run time. Moreover, Nimble schedules independent GPU tasks to be executed in parallel, further boosting its performance. Our evaluation on various neural networks shows that Nimble outperforms popular DL frameworks (e.g., PyTorch [104]) and state-of-the-art inference systems (e.g., TensorRT [9] and TVM [33]).

Next, we propose Orca, a distributed serving system that achieves low latency and high throughput for serving Transformer-based generative models. Orca incorporates two techniques: iteration-level scheduling and selective batching. Iteration-level scheduling makes the inference server interact with the execution engine at the granularity of iteration instead of request, while selective batching enables batching arbitrary requests processing tokens at different positions, which is crucial for applying batching with iteration-level scheduling. Experiments show the effectiveness of our approach: ORCA provides an order of magnitude higher throughput than current state-of-the-art systems (e.g., FasterTransformer [7]) at the same level of latency.

Lastly, we present WINDTUNNEL, a framework for translating pipelines of classical ML operators into neural networks and further *jointly* fine-tuning them. As part of the translation procedure, we also propose techniques for translating popular non-differentiable operators including GBDT and categorical encoders. The experimental results show that the translation with knowledge transfer followed by the fine-tuning leads to significant accuracy improvements over the original pipeline and state-of-the-art NNs. Furthermore, by translating classical ML pipelines into neural networks, we can leverage existing systems (e.g., PyTorch [104]) primarily developed for DL workloads and train classical ML pipelines on GPUs. We deem this work as a step towards filling the gap between classical ML pipelines and neural networks.

Bibliography

[1] CUDA C++ programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[2] cuML - GPU Machine Learning Algorithms. Retrieved Nov 10, 2022 from `https://github.com/rapidsai/cuml`.

[3] DeepSpeed. Retrieved Dec 13, 2021 from `https://github.com/microsoft/DeepSpeed`.

[4] gRPC. Retrieved Dec 13, 2021 from `https://grpc.io`.

[5] Megatron-LM. Retrieved Dec 13, 2021 from `https://github.com/NVIDIA/Megatron-LM`.

[6] NVIDIA A100 Tensor Core GPU architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`.

[7] NVIDIA FasterTransformer. Retrieved Dec 13, 2021 from `https://github.com/NVIDIA/FasterTransformer`.

[8] NVIDIA NCCL. Retrieved Dec 13, 2021 from `https://github.com/NVIDIA/nccl`.

[9] NVIDIA TensorRT. Retrieved Dec 13, 2021 from https://developer.nvidia.com/tensorrt.

[10] NVIDIA Tesla V100 GPU architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[11] NVIDIA Triton Inference Server. Retrieved Dec 13, 2021 from https://developer.nvidia.com/nvidia-triton-inference-server.

[12] ONNX: Open neural network exchange. https://github.com/onnx/onnx.

[13] Tensor Cores. https://developer.nvidia.com/tensor-cores.

[14] TensorFlow XLA. Retrieved Dec 13, 2021 from www.tensorflow.org/xla.

[15] TFRT: A new TensorFlow runtime. https://github.com/tensorflow/runtime.

[16] TorchScript. https://pytorch.org/docs/stable/jit.html.

[17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

[18] D. Adiwardana, M.-T. Luong, D. R. So, J. Hall, N. Fiedel, R. Thoppilan, Z. Yang, A. Kulshreshtha, G. Nemade, Y. Lu, et al. Towards a Human-like Open-Domain Chatbot. *arXiv preprint arXiv:2001.09977*, 2020.

[19] E. Agichtein, E. Brill, and S. Dumais. Improving web search ranking by incorporating user behavior information. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 19–26, 2006.

[20] A. Agrawal, A. N. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, I. Ganichev, J. Levenberg, M. Hong, R. Monga, and S. Cai. TensorFlow Eager: A multi-stage, python-embedded DSL for machine learning. In *MLSys*, 2019.

[21] Z. Ahmed, S. Amizadeh, M. Bilenko, R. Carr, W. Chin, Y. Dekel, X. Dupré, V. Eksarevskiy, S. Filipi, T. Finley, A. Goswami, M. Hoover, S. Inglis, M. Interlandi, N. Kazmi, G. Krivosheev, P. Luferenko, I. Matantsev, S. Matusevych, S. Moradi, G. Nazirov, J. Ormont, G. Oshri, A. Pagnoni, J. Parmar, P. Roy, M. Z. Siddiqui, M. Weimer, S. Zahirazami, and Y. Zhu. Machine learning at microsoft with ml. net. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2448–2458, 2019.

[22] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. Deep speech 2 : End-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 173–182, 2016.

[23] M. Artetxe, S. Bhosale, N. Goyal, T. Mihaylov, M. Ott, S. Shleifer, X. V. Lin, J. Du, S. Iyer, R. Pasunuru, G. Anantharaman, X. Li, S. Chen, H. Akin, M. Baines, L. Martin, X. Zhou, P. S. Koura, B. O'Horo, J. Wang, L. Zettlemoyer, M. Diab, Z. Kozareva, and V. Stoyanov. Efficient Large

Scale Language Modeling with Mixtures of Experts. *arXiv preprint arXiv:2112.10684*, 2021.

[24] Avazu. Avazu click-through rate prediction dataset, 2021.

[25] A. Banerjee. Initializing neural networks using decision trees. *Computational Learning Theory and Natural Learning Systems*, 4:3–15, 1997.

[26] P. F. Brown, J. Cocke, S. A. D. Pietra, V. J. D. Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin. A Statistical Approach to Machine Translation. *Computational Linguistics*, 16(2):79–85, 1990.

[27] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 2020.

[28] C. J. Burges. From ranknet to lambdarank to lambdamart: An overview. Technical Report MSR-TR-2010-82, June 2010.

[29] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu. Path-level network transformation for efficient architecture search. In *ICML*, 2018.

[30] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019.

[31] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan,

S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

[32] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[33] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pages 579–594, 2018.

[34] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pages 7–10, 2016.

[35] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[36] J.-H. Cho, J. Kim, W. Lee, D.-U. Lee, T. K. Kim, H. B. Park, C. Jeong, M.-J. Park, S. G. Baek, S. Choi, B. K. Yoon, Y. J. Choi, K. Y. Lee, D. Shim, J. Oh, J. Kim, and S.-H. Lee. A 1.2V 64Gb 341GB/S HBM2 stacked DRAM with spiral point-to-point TSV structure and improved bank group data control. In *ISSCC*, 2018.

[37] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. PaLM: Scaling Language Modeling with Pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[38] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, pages 613–627, 2017.

[39] CriteoLabs. Kaggle display advertising challenge dataset, 2021.

[40] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of*

the 2019 IEEE Conference on Computer Vision and Pattern Recognition, pages 113–123, 2019.

[41] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[42] R. Dabre, C. Chu, and A. Kunchukuttan. A Survey of Multilingual Neural Machine Translation. *ACM Computing Surveys*, 53(5), 2020.

[43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[44] M. Ding, Z. Yang, W. Hong, W. Zheng, C. Zhou, D. Yin, J. Lin, X. Zou, Z. Shao, H. Yang, and J. Tang. CogView: Mastering Text-to-Image Generation via Transformers. *Advances in Neural Information Processing Systems*, 2021.

[45] L. Dixon, J. Li, J. Sorensen, N. Thain, and L. Vasserman. Measuring and Mitigating Unintended Bias in Text Classification. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 67–73, 2018.

[46] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. Bosma, Z. Zhou,

T. Wang, Y. E. Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. V. Le, Y. Wu, Z. Chen, and C. Cui. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. *arXiv preprint arXiv:2112.06905*, 2021.

[47] J. Fang, Y. Yu, C. Zhao, and J. Zhou. TurboTransformers: An Efficient GPU Serving System for Transformer Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[48] W. Fedus, B. Zoph, and N. Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[49] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[50] P. Gao, L. Yu, Y. Wu, and J. Li. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[51] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoft's bing search engine. In *Proceedings of the 27th International Conference on Machine Learning*, pages 13–20, 2010.

[52] A. Gray. Getting started with CUDA Graphs. NVIDIA Developer Blog, 2019. `https://devblogs.nvidia.com/cuda-graphs/`.

[53] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. Deepfm: A factorization-machine based neural network for ctr prediction. In *Proceedings of the*

26th International Joint Conference on Artificial Intelligence, pages 1725–1731, 2017.

[54] H. Hassan, A. Aue, C. Chen, V. Chowdhary, J. Clark, C. Federmann, X. Huang, M. Junczys-Dowmunt, W. Lewis, M. Li, S. Liu, T.-Y. Liu, R. Luo, A. Menezes, T. Qin, F. Seide, X. Tan, F. Tian, L. Wu, S. Wu, Y. Xia, D. Zhang, Z. Zhang, and M. Zhou. Achieving human parity on automatic chinese to english news translation, 2018.

[55] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[56] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[57] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[58] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. v. d. Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre. Training Compute-Optimal Large Language Models. *arXiv preprint arXiv:2203.15556*, 2022.

[59] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying Large Video

Datasets with Low Latency and Low Cost. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pages 269–286, 2018.

[60] H. T. Hsu. An algorithm for finding a minimal equivalent graph of a digraph. *Journal of the ACM*, 22(1):11–16, 1975.

[61] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. *Advances in Neural Information Processing Systems*, 2019.

[62] K. D. Humbird, J. L. Peterson, and R. G. McClarren. Deep neural network initialization with decision trees. *IEEE Transactions on Neural Networks and Learning Systems*, 30(5):1286–1295, 2018.

[63] I. Ivanova and M. Kubat. Initialization of neural networks by means of decision trees. *Knowledge-Based Systems*, 8(6):333–344, 1995.

[64] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[65] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D. G. Andersen, and G. R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant Video Processing. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 29–42, 2018.

[66] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting concept drift in malware classifi-

cation models. In *Proceedings of the 26th USENIX Security Symposium*, pages 625–642, 2017.

[67] Y. Juan, W.-S. Chin, and Y. Zhuang. 3 idiots' approach for display advertising challenge, 2014.

[68] Y. Juan, Y. Zhuang, W.-S. Chin, and C.-J. Lin. Field-aware factorization machines for ctr prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 43–50, 2016.

[69] Kaggle. State of data science and machine learning 2020, 2020.

[70] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[71] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*, 2020.

[72] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 3149–3157, 2017.

[73] G. Ke, Z. Xu, J. Zhang, J. Bian, and T.-Y. Liu. Deepgbm: A deep learning framework distilled by gbdt for online prediction tasks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 384–394, 2019.

[74] D. Khashabi, S. Min, T. Khot, A. Sabharwal, O. Tafjord, P. Clark, and H. Hajishirzi. UNIFIEDQA: Crossing Format Boundaries with a Single

QA System. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1896–1907, 2020.

[75] J.-H. Kim, W. Choo, and H. O. Song. Puzzle mix: Exploiting saliency and local statistics for optimal mixup. In *Proceedings of the 37th International Conference on Machine Learning*, pages 5275–5285, 2020.

[76] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the 14th EuroSys Conference*, pages 1–15, 2019.

[77] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference for Learning Representations*, 2015.

[78] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, 2006.

[79] P. Kontschieder, M. Fiterau, A. Criminisi, and S. R. Bulò. Deep neural decision forests. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 4190–4194, 2016.

[80] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[81] D. Koutsoukos, S. Nakandala, K. Karanasos, K. Saur, G. Alonso, and M. Interlandi. Tensors: An abstraction for general data processing. *Proceedings of the VLDB Endowment*, 14(10):1797–1804, 2021.

[82] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[83] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee, et al. Natural Questions: a Benchmark for Question Answering Research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019.

[84] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun. Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning. *Advances in Neural Information Processing Systems*, 2020.

[85] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pages 611–626, 2018.

[86] G. Li, Y. Xi, J. Ding, D. Wang, B. Liu, C. Fan, X. Mao, and Z. Zhao. Easy and Efficient Transformer: Scalable Inference Solution For large NLP model. *arXiv preprint arXiv:2104.12470*, 2021.

[87] O. Lieber, O. Sharir, B. Lenz, and Y. Shoham. Jurassic-1: Technical details and evaluation. 2021.

[88] S. Lim, I. Kim, T. Kim, C. Kim, and S. Kim. Fast autoaugment. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 6665–6675, 2019.

[89] X. Lin, G. Bertasius, J. Wang, S.-F. Chang, D. Parikh, and L. Torresani. Vx2text: End-to-end learning of video-based text generation from multi-

modal inputs. In *Proceedings of the 2021 IEEE Conference on Computer Vision and Pattern Recognition*, pages 7005–7015, 2021.

[90] H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *ICLR*, 2019.

[91] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou. *Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks*, pages 881–897. 2020.

[92] W. McGinnis. Binary encoder for categorical variables, 2016.

[93] W. McGinnis. Hashing encoder for categorical variables, 2016.

[94] W. McGinnis. One-hot encoder for categorical variables, 2016.

[95] T. Mihaylov, P. Clark, T. Khot, and A. Sabharwal. Can a Suit of Armor Conduct Electricity? A New Dataset for Open Book Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2381–2391, 2018.

[96] M. Milutinovic, A. G. Baydin, R. Zinkov, W. Harvey, D. Song, F. Wood, and W. Shen. End-to-end training of differentiable pipelines across machine learning frameworks. *NIPS AutoDiff workshop*, 2017.

[97] S. Nakandala, K. Saur, G.-I. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi. A tensor compiler for unified machine learning prediction serving. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 899–917, 2020.

[98] R. Nallapati, B. Zhou, C. N. dos Santos, Ç. Gülçehre, and B. Xiang. Abstractive Text Summarization using Sequence-to-sequence RNNs and

Beyond. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, pages 280–290, 2016.

[99] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*, 2018.

[100] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: Visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, pages 1–7, 2011.

[101] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. TensorFlow-Serving: Flexible, High-Performance ML Serving. *Workshop on Machine Learning Systems at NIPS 2017*, 2017.

[102] A. S. on Statistical Computing. Data expo 2009 - airline on-time performance, 2021.

[103] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, 2019.

[104] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-

Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 2019.

[105] R. Paulus, C. Xiong, and R. Socher. A Deep Reinforced Model for Abstractive Summarization. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

[106] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[107] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[108] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 16th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, 2018.

[109] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *ICML*, 2018.

[110] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. Catboost: unbiased boosting with categorical features. In *Proceedings*

of the 32nd International Conference on Neural Information Processing Systems, pages 6639–6649, 2018.

[111] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer. Data science through the looking glass and what we found there, 2019.

[112] Y. Qu, H. Cai, K. Ren, W. Zhang, Y. Yu, Y. Wen, and J. Wang. Product-based neural networks for user response prediction. In *Proceedings of the 16th IEEE International Conference on Data Mining*, pages 1149–1154, 2016.

[113] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[114] J. W. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young, E. Rutherford, T. Hennigan, J. Menick, A. Cassirer, R. Powell, G. v. d. Driessche, L. A. Hendricks, M. Rauh, P.-S. Huang, A. Glaese, J. Welbl, S. Dathathri, S. Huang, J. Uesato, J. Mellor, I. Higgins, A. Creswell, N. McAleese, A. Wu, E. Elsen, S. Jayakumar, E. Buchatskaya, D. Budden, E. Sutherland, K. Simonyan, M. Paganini, L. Sifre, L. Martens, X. L. Li, A. Kuncoro, A. Nematzadeh, E. Gribovskaya, D. Donato, A. Lazaridou, A. Mensch, J.-B. Lespiau, M. Tsimpoukelli, N. Grigorev, D. Fritz, T. Sottiaux, M. Pajarskas, T. Pohlen, Z. Gong, D. Toyama, C. d. M. d'Autume, Y. Li, T. Terzi, V. Mikulik, I. Babuschkin, A. Clark, D. d. L. Casas, A. Guy, C. Jones, J. Bradbury, M. Johnson, B. Hechtman, L. Weidinger, I. Gabriel, W. Isaac, E. Lockhart, S. Osindero, L. Rimell, C. Dyer, O. Vinyals, K. Ayoub, J. Stanway, L. Bennett, D. Hassabis, K. Kavukcuoglu, and G. Irving.

Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *arXiv preprint arXiv:2112.11446*, 2021.

[115] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[116] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. *arXiv preprint arXiv:2201.05596*, 2022.

[117] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. Zero-Shot Text-to-Image Generation. In *Proceedings of the 38th International Conference on Machine Learning*, pages 8821–8831, 2021.

[118] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.

[119] S. Rendle. Factorization machines. In *Proceedings of the 10th IEEE International Conference on Data Mining*, pages 995–1000, 2010.

[120] S. Rendle. Scaling factorization machines to relational data. *Proceedings of the VLDB Endowment*, 6(5):337–348, 2013.

[121] S. Rendle, W. Krichene, L. Zhang, and J. Anderson. Neural collaborative filtering vs. matrix factorization revisited. In *Proceedings of the 14th ACM Conference on Recommender Systems*, pages 240–248, 2020.

[122] S. Roller, E. Dinan, N. Goyal, D. Ju, M. Williamson, Y. Liu, J. Xu, M. Ott, E. M. Smith, Y.-L. Boureau, and J. Weston. Recipes for Building an Open-Domain Chatbot. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 300–325, 2021.

[123] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2019.

[124] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[125] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[126] T. Schick and H. Schütze. Exploiting Cloze-Questions for Few-Shot Text Classification and Natural Language Inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 255–269, 2021.

[127] A. See, P. J. Liu, and C. D. Manning. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, 2017.

[128] I. K. Sethi. Entropy nets: From decision trees to neural networks. *Proceedings of the IEEE*, 78(10):1605–1613, 1990.

[129] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. *Advances in Neural Information Processing Systems*, 2018.

[130] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[131] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast Video Classification via Adaptive Cascading of Deep Models. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3646–3654, 2017.

[132] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[133] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990*, 2022.

[134] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In

Proceedings of the 33rd IEEE International Conference on Data Engineering, pages 535–546, 2017.

[135] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[136] D. Stosic. Training neural networks with tensor core, 2020.

[137] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[138] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, pages 6105–6114, 2019.

[139] M. Tan and Q. V. Le. MixConv: Mixed depthwise convolutional kernels. In *BMVC*, 2019.

[140] H. team. H2o, 2021.

[141] H. team. Hummingbird, 2021.

[142] O. team. Onnxmltools, 2021.

[143] P. team. Pytorch ecosystem, 2021.

[144] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *KDD*, 2019.

[145] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor Comprehen-

sions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[146] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is All you Need. *Advances in Neural Information Processing Systems*, 2017.

[147] X. Wang, Y. Xiong, Y. Wei, M. Wang, and L. Li. LightSeq: 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*, pages 113–120, 2021.

[148] Z. Wang, W. Liu, Q. He, X. Wu, and Z. Yi. Clip-gen: Language-free training of a text-to-image generator with clip. *arXiv preprint arXiv:2203.00386*, 2022.

[149] J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le. Finetuned Language Models are Zero-Shot Learners. In *Proceedings of the 10th International Conference on Learning Representations*, 2022.

[150] R. J. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1(2):270–280, 1989.

[151] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 2048–2057, 2015.

[152] Y. Yang, I. G. Morillo, and T. M. Hospedales. Deep neural decision trees, 2018.

[153] Z. Yang, Y. Yuan, Y. Wu, W. W. Cohen, and R. R. Salakhutdinov. Review Networks for Caption Generation. *Advances in Neural Information Processing Systems*, 2016.

[154] G.-I. Yu, S. Amizadeh, B.-G. Chun, M. Weimer, and M. Interlandi. Making Classical Machine Learning Pipelines Differentiable: A Neural Translation Approach. In *Proceedings of the Workshop on Systems for ML at NeurIPS*, 2018.

[155] G.-I. Yu, S. Amizadeh, S. Kim, A. Pagnoni, C. Zhang, B.-G. Chun, M. Weimer, and M. Interlandi. WindTunnel: Towards Differentiable ML Pipelines beyond a Single Model. *Proceedings of the VLDB Endowment*, 15(1):11–20, 2022.

[156] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, pages 521–538, 2022.

[157] H. Zhang, C. Wu, Z. Zhang, Y. Zhu, Z. Zhang, H. Lin, Y. Sun, T. He, J. Mueller, R. Manmatha, M. Li, and A. Smola. ResNeSt: Split-attention networks. *arXiv preprint arXiv:2004.08955*, 2020.

[158] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068*, 2022.

[159] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica. Ansor: Generating High-Performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 863–879, 2020.

[160] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.

Appendix A

Appendix: Nimble

A.1 Proofs on the Stream Assignment Algorithm of Nimble

In this chapter, we provide detailed proofs on the theorems presented in Section 3.3.2.

Problem Setting We assume that the computation graph of a neural network is given. The computation graph is represented as a finite DAG $G = (V, E)$. Also, we are given a set of GPU streams $S = \{s_1, s_2, \cdots, s_{|V|}\}$. Algorithm 1 must find a stream assignment $f : V \to S$, which satisfies the following conditions:

- **Maximum logical concurrency.** If $u, v \in V$ and there exists no path between u and v in G, then $f(u) \neq f(v)$.
- **Minimum number of synchronizations.** Among such functions, f incurs the smallest number of synchronizations across streams.

Here we define important concepts and terminologies used in the following proofs.

Definition 1. For a graph $G = (V, E)$, a **synchronization plan** $\Lambda \subseteq E$ is a set of edges on which synchronizations are planned to be performed (regardless of stream assignments).

Definition 2. For a stream assignment f on $G = (V, E)$, a synchronization plan $\Lambda \subseteq E$ is **safe** if it satisfies the following condition.

For any $(u, v) \in E$, $f(u) = f(v)$ or there exists a path $P \subseteq E$ from u to v such that $P \cap \Lambda \neq \emptyset$.

In other words, the plan Λ is safe when the execution order between every pair of adjacent nodes u and v is guaranteed: either by assigning them to the same streams or by performing a synchronization somewhere after u and before v.

Notation. We denote by $min_{sync}(G, f)$ the minimum number of synchronizations required when applying f to the graph G. That is,

$$min_{sync}(G, f) = min\{|\Lambda| \in \mathbb{Z}_{\geq 0} \mid \Lambda \subseteq E \text{ is safe for } f \text{ on } G\}$$

A.1.1 Proof of Theorem 1

Theorem 1 includes two statements, which are presented here as Theorem 1-1 and Theorem 1-2, respectively.

Theorem 1-1. *A stream assignment f satisfies maximum logical concurrency on a computation graph G if and only if f satisfies maximum logical concurrency on the minimum equivalent graph G'.*

Proof of Theorem 1-1. By definition of MEG, G' has the same reachability relation as G. Thus, if no path exists between a pair of nodes in G, then there is no path between the same pair of nodes in G', and vice versa. □

Prior to the proof of Theorem 1-2, we describe and prove Lemma 1 and Lemma 2.

Lemma 1. *For a minimum equivalent graph $G' = (V, E')$ of G, if $(u, v) \in E'$, then $\{(u, v)\}$ is the only path in G from u to v.*

Proof of Lemma 1. We will prove by contradiction. Suppose there is another path $P \subseteq E$ from u to v that goes through $w \in V$. By the definition of MEG, G' must preserve reachability from u to w and w to v. Consequently, removing the edge (u, v) from E' does not change the reachability relation. This is contradictory to the definition of MEG, because we can construct another subgraph $G^* = (V, E' \setminus \{(u, v)\})$, where the number of edges of G^* is smaller than that of G' while preserving the reachability relation. □

Lemma 2. *A synchronization plan $\Lambda \subseteq E$ is safe for a stream assignment f on G if and only if Λ is safe for f on G'.*

Proof of Lemma 2. We first show that if Λ is safe for f on G, then Λ is safe for f on G'. We will prove by contradiction. Suppose Λ is safe for f on G but not safe for f on G'. Then there is an edge $(u, v) \in E'$ such that $f(u) \neq f(v)$ and $(u, v) \notin \Lambda$. Since G' is the MEG of G and $(u, v) \in E'$, $\{(u, v)\}$ is the only path in G from u to v by Lemma 1. Consequently, $(u, v) \in E$ is an edge that $f(u) \neq f(v)$ and every path in G from u to v does not include any edge in Λ, which is contradictory to the assumption that Λ is safe for f on G.

Next, we show that if Λ is safe for f on G', then Λ is safe for f on G. We will prove by contradiction. Suppose Λ is safe for f on G' but not safe for f on G. Then there is an edge $(u, v) \in E \setminus E'$ such that $f(u) \neq f(v)$ and every path from u to v in G does not include any edge in Λ. Since $(u, v) \notin E'$ and G' preserves the same reachability relation as G, there must exist a node $w_1 \in V$ such that $(u, w_1) \in E'$ and a path from w_1 to v exists in G'. As every path from u to v in G does not include any edge in Λ, $f(u) = f(w_1)$ must hold to meet the assumption that Λ is safe for f on G'. Then, we have two vertices w_1 and v such that $f(w_1) \neq f(v)$ and every path from w_1 to v in G does not include any edge in Λ. Since G is a finite DAG, if we repeat this process, we end up with two vertices w_n and v with the following conditions: $(w_n, v) \in E'$, $f(w_n) \neq f(v)$, and $(w_n, v) \notin \Lambda$, which contradicts the assumption that Λ is safe for f on G'. □

Theorem 1-2. *For any stream assignment f that satisfies maximum logical concurrency on G, the following equation holds.*

$$min_{sync}(G, f) = min_{sync}(G', f).$$

That is, the minimum number of synchronizations required for f on G is equal to the minimum number of synchronizations required for f on G'.

Proof of Theorem 1-2. This directly follows from Lemma 2. □

A.1.2 Proof of Theorem 2

Prior to the proof of Theorem 2, we clarify the meaning of *the set of the stream assignments*. Let $F = \{f \mid f : V \to S\}$. We can define an equivalence relation $\sim$ on F as follows.

> *For stream assignments $g, h \in F$, $g \sim h$ if and only if $g = \sigma \circ h$ for some permutation σ over S.*

Note that any permutation on S does not affect the degree of logical concurrency and the number of synchronizations of a stream assignment. In other words, for stream assignments $g, h \in F$ such that $g \sim h$, it directly follows that 1) g meets maximum logical concurrency if and only if h meets maximum logical concurrency, and 2) $min_{sync}(G', g) = min_{sync}(G', h)$. Therefore, if two stream assignments can be converted to one another by some permutation on S, we do not differentiate the two stream assignments. Furthermore, we do not differentiate a stream assignment $f \in F$ from its equivalence class $[f]$, because we only consider which nodes are mapped to the same streams, but do not consider the exact value of f. From now on, we *identify* $[f]$, the equivalence class of f, as f.

Remark. The set of the stream assignments $\mathbb{F}$ is as follows.

$$\mathbb{F} = \{[f] \mid f : V \to S\}$$

Now we can reinterpret Theorem 2 using the definition of the set of the stream assignments.

Theorem 2. *Let $\mathbb{M}$ be the set of the matchings of the bipartite graph B obtained from G', and $\mathbb{F}_{max}$ be the set of the stream assignments that satisfy maximum logical concurrency on G'. Then one-to-one correspondence $\Phi : \mathbb{M} \to \mathbb{F}_{max}$ exists.*

Proof of Theorem 2. We construct Φ according to Step 4 and Step 5 of Algorithm 1.

First, we show that $\Phi(m) \in \mathbb{F}_{max}$, i.e., $\Phi(m)$ meets maximum logical concurrency, for any matching $m \in \mathbb{M}$. We prove this by contradiction. Choose an arbitrary matching $m \in \mathbb{M}$ and suppose that $\Phi(m)$ does not satisfy maximum logical concurrency. In other words, suppose that a pair of nodes $v_i, v_j \in V$ exists such that there is no path from v_i to v_j in G' but $\Phi(m)(v_i) = \Phi(m)(v_j)$. Since v_i and v_j are mapped to the same stream, it follows from Step 4 that there exists a sequence of edges $\{(x_i, y_{k_1}), (x_{k_1}, y_{k_2}), \cdots, (x_{k_l}, y_j)\} \subseteq m$. This, in turn, means that there exists a path $\{(v_i, v_{k_1}), (v_{k_1}, v_{k_2}), \cdots, (v_{k_l}, v_j)\} \subseteq E'$, which is contradictory to the assumption. Therefore, for any $m \in \mathbb{M}$, $\Phi(m)$ meets maximum logical concurrency.

Secondly, we show that Φ is injective. Again, we will prove by contradiction. Suppose that $\Phi(m_1) = \Phi(m_2)$ for some matchings $m_1 \neq m_2$. Since $m_1 \neq m_2$, there exists an edge $(x_i, y_j) \in E_B$ that is included in either of the two matchings. Without loss of generality, assume $(x_i, y_j) \in m_1$. Then the equation $\Phi(m_1)(v_i) = \Phi(m_1)(v_j)$ holds, and so does the equation $\Phi(m_2)(v_i) = \Phi(m_2)(v_j)$. The latter equation implies that there exists a sequence of edges $\{(x_i, y_{k_1}), (x_{k_1}, y_{k_2}), \cdots, (x_{k_l}, y_j)\} \subseteq m_2$. This, in turn, means that a path from v_i to v_j other than than edge (v_i, v_j) exists in E', which is contradictory to the assumption that G' is the MEG of the graph G by Lemma 1.

Lastly, we demonstrate that Φ is surjective. Assume that an arbitrary stream assignment $f \in \mathbb{F}_{max}$ is given. We construct $m_f \subseteq E_B$ in such a way that $(x_i, y_j) \in m_f$ if and only if $f(v_i) = f(v_j)$ and $(v_i, v_j) \in E'$. Then $\Phi(m_f) = f$ follows by definition of Φ. □

A.1.3 Proof of Theorem 3

Definition 3. For a stream assignment f on G', we define $Q(f) \subseteq V$ as follows.

$$Q(f) = \{v \in V \mid \exists p \in V \ s.t. \ (p, v) \in E' \text{ and } f(p) = f(v)\}$$

That is, a node $v \in V$ is included in $Q(f)$ if and only if it has at least one parent node which is mapped to the same stream as v by f.

Definition 4. For a stream assignment f that satisfies maximum logical concurrency on G', we define a function $R_f(v) : Q(f) \to V$ as follows.

$$R_f : v \mapsto p \ s.t. (p, v) \in E' \text{ and } f(p) = f(v)$$

Lemma 3. *The function R_f is well-defined.*

Proof of Lemma 3. By definition of $Q(f)$, $R_f(v)$ exists for any $v \in Q(f)$. What we have to show is the uniqueness of such p for each v. Suppose $\exists p_1, p_2 \in$

V such that $(p_1, v), (p_2, v) \in E'$ and $f(p_1) = f(p_2)$. Since f satisfies maximum logical concurrency, there is a path between p_1 and p_2. Without loss of generality, assume that there is a path from p_1 to p_2. Then $(p_1, v) \in E'$ can be removed from the MEG of G, which contradicts the assumption that G' is MEG of G. □

Lemma 4. *For a stream assignment f that satisfies maximum logical concurrency on G',*

$$min_{sync}(G', f) = |E'| - |Q(f)|.$$

Proof of Lemma 4. We first show that $min_{sync}(G', f) \leq |E'| - |Q(f)|$. For any node $v \in Q(f)$, there exists an edge $(R_f(v), v) \in E'$. Observe that synchronization on the edge $(R_f(v), v)$ is redundant because $f(R_f(v)) = f(v)$. Thus, among all of the edges in E', we can guarantee that at least $|Q(f)|$ edges do not require synchronizations.

Conversely, we show that $min_{sync}(G', f) \geq |E'| - |Q(f)|$. Let $\Lambda \in E'$ be a safe synchronization plan for f on G' such that $|\Lambda| = min_{sync}(G', f)$. Select an arbitrary node $v \in V$ and let $I_v \subseteq E'$ be the set of the incoming edges to v in G'. If $v \notin Q(f)$, for any edge $e = (p, v) \in I_v$, $e \in \Lambda$. This is because, by Lemma 1, $\{e\}$ is the only path between p and v, and, therefore, any safe synchronization plan must include the edge e. If $v \in Q(f)$, any edge $e \in I_v$ other than $(R_f(v), v)$ must be included in Λ. Thus, the following inequality holds.

$$min_{sync} \geq \sum_{v \notin Q(f)} |I_v| + \sum_{v \in Q(f)} (|I_v| - 1)$$

Clearly, the righthand side is equal to $|E'| - |Q(f)|$. □

Theorem 3. *For any matching $m \in \mathbb{M}$, the following equation holds.*

$$min_{sync}(G', \Phi(m)) = |E'| - |m|.$$

Proof of Theorem 3. Let $m \in \mathbb{M}$ be a matching of the bipartite graph B. By Theorem 2 and Lemma 4, it suffices to show $|Q(\Phi(m))| = |m|$. For this purpose, we define a function $\Psi_m : Q(\Phi(m)) \to m$ and demonstrate that Ψ_m is a bijection.

We first define a function $H : E' \to E_B$ as $H : (v_i, v_j) \mapsto (x_i, y_j)$. Since we construct the bipartite graph B in the same manner as H, it is trivial that the function H is bijective. Now we define Ψ_m as

$$\Psi_m(v) = H(R_{\Phi(m)}(v), v), \ \ \forall v \in Q(\Phi(m))$$

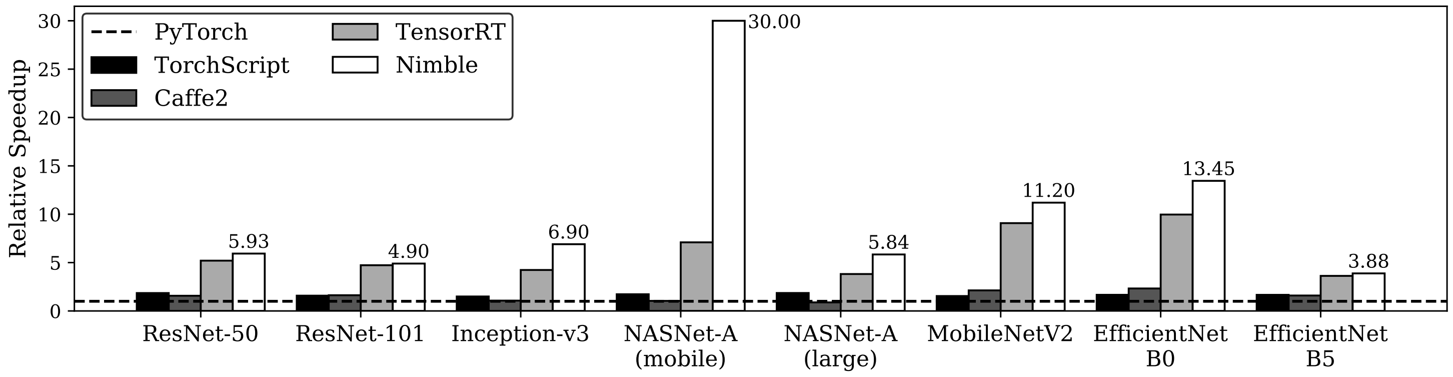
We can easily confirm that Ψ_m is injective. Since H is bijective, if $\Psi_m(u) = \Psi_m(v)$ then $(R_{\Phi(m)}(u), u) = (R_{\Phi(m)}(v), v)$. Thus, $u = v$ follows.

Next, we show that Ψ_m is surjective. Select an arbitrary edge $(x_i, y_j) \in m$. Since $(x_i, y_j) \in E_B$, $(v_i, v_j) \in E'$. Also, by definition of Φ, $\Phi(m)(v_i) = \Phi(m)(v_j)$. Thus, it follows that $v_j \in Q(\Phi(m))$ and $R_{\Phi(m)}(v_j) = v_i$. That is, the first coordinate of $\Psi_m(v_j)$ is x_i. In addition, from the definition of Ψ_m and H, it is clear that the second coordinate of $\Psi_m(v_j)$ is y_j. To sum up, it follows that $\Psi_m(v_j) = (x_i, y_j)$, i.e., Ψ_m is surjective.

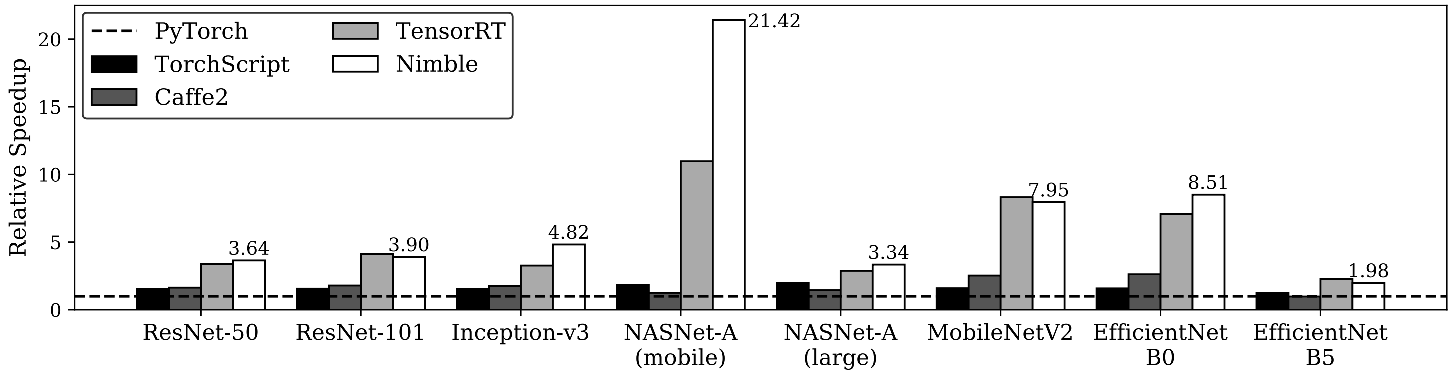
Since Ψ_m is a bijection between $Q(\Phi(m))$ and m, cardinality of the two sets are equal. □

A.1.4 Time Complexity Analysis

Since the computation graph $G = (V, E)$ is a finite DAG, its minimum equivalent graph can be obtained in $O(V^3)$ time [60]. To convert G' into the bipartite graph B, Nimble computes the transitive closure of G', which again takes $O(V^3)$ time. Additionally, in calculating a maximum matching of the bipartite graph B, Nimble uses Ford-Fulkerson method [49] which costs $O(VE)$ time. To sum up, the stream assignment algorithm of Nimble takes $O(V^3)$ time in total. Note that Nimble computes the stream assignment once before the AoT scheduling, so the time spent on Algorithm 1 is amortized over iterations. Therefore, the time spent on the stream assignment algorithm can be considered negligible.



(a) Results on an NVIDIA Titan RTX GPU.



(b) Results on an NVIDIA Titan Xp GPU.

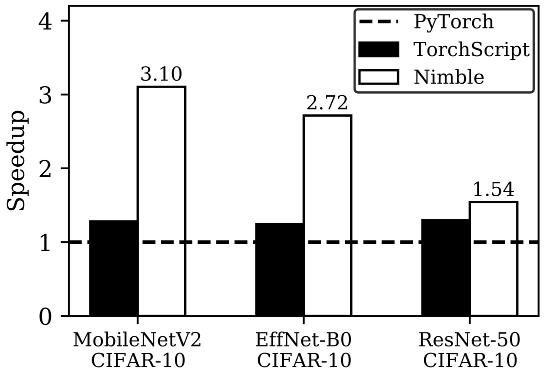
Figure A.1: Relative inference speedup of Nimble and other systems (batch size 1).

A.2 Evaluation Results on Various GPUs

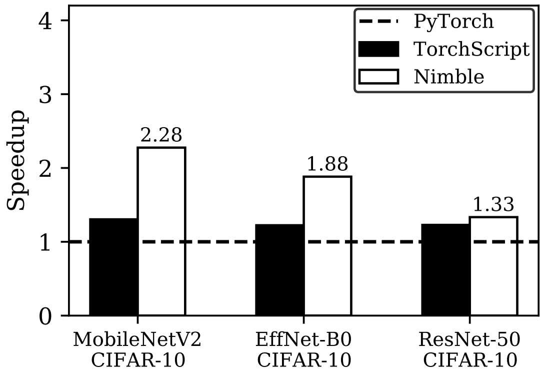
In addition to the evaluation results described in Section 5, we attach results on the different types of GPUs: NVIDIA Titan RTX and NVIDIA Titan Xp. We keep the other experimental settings the same. Note that we exclude TVM from this set of experiments because TVM needs to tune the kernels separately for each type of GPU for a long time. Figure A.1 shows that Nimble achieves significant speedup across various GPU architectures ranging from Pascal to Turing.

A.3 Evaluation Results on Different Training Batch Sizes

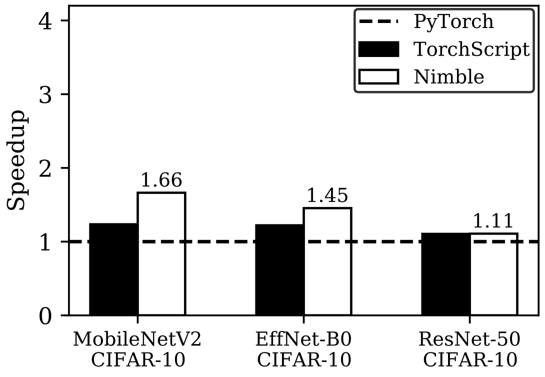
We also present results on the performance of Nimble when training the neural networks with varying batch sizes. We use an NVIDIA V100 GPU, following the setting described in Section 5. Figure A.2 shows that Nimble can achieve performance improvement in the training of the neural networks on the CIFAR-10 dataset even when the batch size is sufficiently large.



(a) Training with batch size 64.



(b) Training with batch size 128.



(c) Training with batch size 256.

Figure A.2: Relative training speedup of Nimble and TorchScript.

초록

최근 경향을 보면 다양한 종류의 애플리케이션에서 머신 러닝(ML) 워크로드가 점점 더 중요하게 활용되고 있다. 이는 ML용 시스템 소프트웨어의 개발을 통해 GPU와 같은 이기종 가속기의 광범위한 활용이 가능해졌기 때문이다. 많은 연구자들의 관심 덕에 ML용 시스템 소프트웨어 스택은 분명 하루가 다르게 개선되고 있지만, 여전히 모든 사례에서 높은 효율성을 보여주지는 못한다. 이 학위논문에서는 시스템 소프트웨어 관점에서 GPU 환경에서 ML 워크로드의 실행 효율성을 개선하는 방법을 연구한다. 구체적으로는 오늘날의 ML용 시스템이 GPU를 효율적으로 사용하지 못하는 워크로드를 규명하고 더 나아가서 해당 워크로드를 효율적으로 처리할 수 있는 시스템 기술을 고안하는 것을 목표로 한다.

본 논문에서는 먼저 최적화된 GPU 스케줄링을 갖춘 ML 실행 엔진인 Nimble을 소개한다. 새 스케줄링 기법을 통해 Nimble은 기존 대비 GPU 실행 효율성을 최대 22.34배까지 향상시킬 수 있다. 둘째로 Transformer 기반의 생성 모델에 특화된 추론 서비스 시스템 Orca를 제안한다. 새로운 스케줄링 및 batching 기술에 힘입어, Orca는 동일한 수준의 지연 시간을 기준으로 했을 때 기존 시스템 대비 36.9배 향상된 처리량을 보인다. 마지막으로 신경망을 사용하지 않는 고전 ML 파이프라인을 신경망으로 변환하는 프레임워크 WindTunnel을 소개한다. 이를 통해 고전 ML 파이프라인 학습을 GPU를 사용해 진행할 수 있게 된다. 또한 WindTunnel은 gradient backpropagation을 통해 파이프라인의 여러 요소를 한 번에 공동으로 학습 할 수 있으며, 이를 통해 파이프라인의 정확도를 더 향상시킬 수 있음을 확인하였다.

주요어: 머신러닝, 딥러닝, 스케줄링, 추론 서빙, 생성 모델, 트랜스포머, 공동 학습
학번: 2017-28658