UC Davis
UC Davis Electronic Theses and Dissertations

Title
Performance Modeling and Optimization for Machine Learning Workloads

Permalink
https://escholarship.org/uc/item/3w91w6pz

Author
Lin, Zhongyi

Publication Date
2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Performance Modeling and Optimization for Machine Learning Workloads

By

ZHONGYI LIN

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John D. Owens, Chair

Lifeng Lai

Jason Lowe-Power

Committee in Charge

2023

Copyright © 2023 by

Zhongyi Lin

All rights reserved.

To my wife Ruixuan, and my parents Liqun and Dan.

Contents

List of Figures . . . vii
List of Tables . . . ix
Abstract . . . x
Acknowledgments . . . xii

1 Introduction **1**

2 Background **6**

2.1 Single Ops in ML and DL . . . 6
2.1.1 Convolution . . . 6
2.1.2 General Matrix Multiplication (GEMM) . . . 6
2.1.3 Sparse Tensor Operations . . . 6
2.2 Compute Devices and Compute Kernel Libraries . . . 8
2.3 ML Models and Applications . . . 9
2.3.1 Natural Language Processing . . . 10
2.3.2 Computer Vision . . . 10
2.3.3 Recommendation Models . . . 11
2.4 Frameworks for ML and DL . . . 11
2.5 DL Compilers . . . 12

3 Towards Flexible and Compiler-Friendly Layer Fusion for CNNs on Multicore CPUs **13**

3.1 Introduction . . . 13
3.2 Related Works . . . 15
3.2.1 Kernel Optimizations and Auto Tuning . . . 15
3.2.2 Layer Fusion . . . 15
3.3 Principles of Effective Layer Fusion . . . 16
3.3.1 Tiling Principles . . . 18
3.3.2 Scheduling Principles . . . 19

3.4 Implementation 19
3.4.1 Kernel-level optimization 19
3.4.2 AutoTVM implementation of fused kernel 22
3.4.3 Compiler integration of fusion for end-to-end test 23
3.5 Results and Analysis 24
3.5.1 Kernel-level Experiments 24
3.5.2 End-to-End Experiments 27
3.6 Conclusion and Future Works 30

4 Building a Performance Model for Deep Learning Recommendation Model Training on GPUs 32
4.1 Introduction 32
4.2 Related Work 35
4.2.1 Recommendation Models and DLRM 35
4.2.2 GPU operator and kernel performance models 36
4.2.3 Models for GEMM-based kernels 37
4.2.4 Model-level performance modeling 37
4.3 Methodology 38
4.3.1 Per-batch Training Time Breakdown 40
4.3.2 Microbenchmark and Performance Models for Dominating Kernels in DLRM 42
4.3.3 Device Idle Time Analysis 46
4.3.4 E2E GPU Training Performance Model 47
4.4 Results and Analysis 49
4.4.1 Performance Models for Dominating Kernels in DLRM 49
4.4.2 Overheads Analysis 51
4.4.3 E2E GPU Training Performance Model for DLRM and More DL Models 53
4.5 Discussions 55
4.5.1 Performance Modeling for Model-System Co-design 56
4.5.2 Extendibility 57

4.6 Conclusion and Future Work 57

5 An Enhanced Performance Model for Machine Learning Distributed Training on Multi-GPU Platforms **59**

5.1 Introduction 59

5.2 Related Works 61

5.2.1 Performance Modeling for Single-Device and Distributed Training . . . 61

5.2.2 Model Parallelism and Sharding 62

5.3 Backgound 63

5.3.1 Pytorch Benchmark Setups 63

5.4 Methodology 64

5.4.1 Dataset Exploration 65

5.4.2 Multi-GPU Training Benchmark and Analysis 66

5.4.3 Kernel Performance Modeling 68

5.4.4 Multi-GPU Performance Modeling 74

5.5 Evaluation and Analysis 77

5.5.1 Kernel Performance Modeling 79

5.5.2 Multi-GPU E2E Performance Modeling 80

5.6 Application and Discussion 81

5.6.1 Case Study: Fast Sharding Config Selection Using Performance Modeling 81

5.6.2 Discussion 84

5.7 Conclusion and Future Work 85

6 Conclusion and Future Works **86**

6.1 Conclusion 86

6.2 Universal Layer Fusion on Various Platforms 86

6.2.1 Overview 86

6.2.2 Graph-level Automatic Fusion Candidate Search 87

6.2.3 Fusion in Training 88

6.2.4 Tensor Compiler is the Future 88

6.3 Universal Performance Modeling on All Compute Platforms 89
6.3.1 Overview 89
6.3.2 Multi-node Multi-GPU Performance Modeling for Industrial-scale ML workloads 90
6.3.3 Assist Accuracy-Performance Co-optimization with Performance Modeling 91
6.3.4 Sparse Ops Performance Modeling Generalization 91

List of Figures

2.1 Overview of the TVM pipeline. 12

3.1 (a) Roofline model examples for fused kernels. (b) Explanation of recomputation in a fused kernel. 16

3.2 Complex-op fusion algorithm visualization. 22

3.3 TVM pipeline with (top) and without (bottom) complex-op fusion. 23

3.4 Throughput comparison between fused and separate kernels. 25

3.5 Standalone kernel rooflines of all the fused and separate kernels. 27

4.1 GPU utilization of per-batch training time of six DL models. 33

4.2 The high-level model architecture of DLRM. 36

4.3 Performance prediction pipeline overview. 38

4.4 Two cases for dependent ops in profiler traces. 39

4.5 Device time breakdown of three DLRM models (batch size 2048). 41

4.6 Host-side overhead types. 47

4.7 T1 overhead means and standard deviations. 51

4.8 Overheads means and standard deviations of 10 most dominating ops. 52

4.9 E2E per-batch training time prediction of three DLRM models on three GPUs. . 53

4.10 E2E per-batch training time prediction of ResNet50 and Inception-V3. 55

4.11 Separate embedding bag ops (left) and batched embedding op (right). 56

5.1 Enhanced performance prediction pipeline overview. 64

5.2 Per-GPU-stream training execution time breakdown of selected ML workloads. 65

5.3 Histogram of average L values of embedding lookup tables in the dataset. . . . 66

5.4 Generation of reuse factors (RF) from embedding lookup indices. 69

5.5 RF values of top 10 most visited tables of the DLRM open-source dataset. . . . 70

5.6 Typical characteristic curves for data movement. 72

5.7 Inter-rank synchronization (red) and intra-rank synchronization (blue). 75

5.8 Communication topologies of the two multi-GPU platforms. 77

5.9 Fitted curves for *all-to-all* benchmark data on two multi-GPU platforms. 79
5.10 Prediction, baseline, and reference of multi-GPU training performance of DLRMs. 81
5.11 Prediction, baseline, and reference of multi-GPU training performance of NLP models. 82
6.1 Forward and backward passes of unfused and fused convolutions. 89

List of Tables

3.1 End-to-end inference time (in milliseconds) of five models on three CPUs. . . . 28
3.2 Layer table. 30

4.1 Comparison of our work with previous performance models. 37
4.2 MLP performance model search space. 46
4.3 DLRM model configurations. 49
4.4 Statistics of active (kernel) time and E2E time prediction errors across three platforms. 50
4.5 Prediction error of dominating kernel latency. 50

5.1 Prediction error of FBGEMM embedding lookup, all-to-all, and all-reduce kernel performance models. 79
5.2 Statistics of DLRM E2E time prediction errors across two multi-GPU platforms. 80
5.3 Sharders and their indexing or cost functions. 82
5.4 Embedding table sharding config selection experiment results. 83

ABSTRACT

Performance Modeling and Optimization for Machine Learning Workloads

Machine learning (ML) workloads emerge and evolve drastically in a series of aspects in recent years. ML workloads' performance, i.e., training/inference speed on various devices/platforms, stands as one of the top considerations in their development. Performance modeling is a powerful technique that helps ML practitioners understand the performance bottlenecks of ML workloads and optimize them. In this dissertation, we showcase how to use performance models to assist in the optimization of ML performance, and how we design such models that are highly accurate, robust, and versatile with different application configurations, such as training/inference, ML model types, and device types.

We first show how to use the roofline model as a simple operator (op) level performance model to identify kernel/layer fusion candidates in convolution neural networks (CNN). We answer the question of when and why fusing two linearly connected complex ops, i.e., convolution (conv) and depthwise convolution (dw-conv) in an ML model will be beneficial in terms of execution time, and propose a deep learning (DL) compiler friendly solution that enables efficient auto-tuning of fused kernel schedule of two layers on multicore CPUs and beat the separate kernel execution performance of TVM (by 1.09x geomean and 1.29x max) MKLDNN-backed PyTorch (by 2.09x geomean and 3.35x max) and as end-to-end (E2E) baselines.

Next, we present a more complicated application of performance models in predicting and aiding the optimization of ML training performance on GPU platforms. Built on top of a series of kernel-level performance models, either ML-based or analytical, for dominating ops/kernels as well as the overhead analysis for all ops in the deep learning recommendation model (DLRM), we devise a critical-path-based performance model that not only predicts the per-batch training time of DLRM on single GPU with low error rate (geomean: 4.61% for GPU active time, 7.96% for E2E, and 10.15% for E2E with shared overheads) but can also be generalized to other types of ML models such as computer vision (CV) and natural language processing (NLP).

Finally, We further extend this performance model to multi-GPU platforms by adding supports to 1) communication collective performance modeling, 2) GPU stream synchronizations

on the same device and across devices in the E2E time prediction algorithm, and 3) data-distribution-aware and problem size flexible performance modeling of embedding table lookup. On single-node multi-GPU platforms, this enhanced model exhibits robustness on DLRM models with random embedding tables, maintains low training speed prediction error (geomean: 5.21% for E2E with shared overheads on randomly generated DLRMs), and generalizes well to NLP models with 3.00% geomean prediction error. With a use case, we demonstrate its ability to quickly select the embedding table sharding configuration and thus improve the end-to-end training performance of DLRMs.

Acknowledgments

I owe my deepest gratitude to my advisor Prof. John D. Owens. Freedom, trust, and a well-structured thinking process are the three best gifts you give me. With your advice and support, I am always free enough to explore the research topics that interest me the most. "I trust your judgment" is what you keep saying and doing throughout these years, and I do benefit a lot from it by becoming more independent in no matter research or work. I also remember you keep asking why. I used to be weak and ignorant in that part, but I finally see some good changes in myself thanks to you pushing me towards it. What I have learned from you is unique and long-lasting, and it is my great honor to work with you for these six memorable years. Thank you!

I would like to extend my gratitude to Prof. Lifeng Lai, Prof. Jason Lowe-Power, Prof. Venkatesh Akella, and Prof. Ilias Tagkopoulos for dedicating their time to be my dissertation and/or qualifying exam committee members and further giving me valuable guidance and feedback on my research and dissertation. I would also like to thank Prof. Chen-nee Chuah for supervising my master's study and research which built a solid foundation for my Ph.D. career.

I have gained invaluable experience in my four internships during my Ph.D. career. As my managers, I was very lucky to work with Joe Eaton (NVIDIA) and Onur Yilmaz (NVIDIA) on RSVD on GPU, Jingyue Wu (Waymo) on quantization-aware training of ML models on autonomous driving cars, Arun Kejariwal (Meta) and Louis Feng (Meta) on GPU performance modeling of recommendation models, and Yufei Zhu (Meta) and Ning Sun (Meta) on anti-pattern detection and remediation. My appreciation also goes to Alex Fender, Sharon Liu, Paul Donnelly, Ehsan Ardestani, Jaewon Lee, John Lundell, Changkyu Kim, Shubho Sengupta, Pallab Bhattacharya, Xizhou Feng, and Valentin Andrei who have provided me with selfless care and generous support. It is my privilege to work on so many impactful projects with all of you.

I'd like to sincerely thank all of my current and future coauthors, including Matthew Yih, Pınar Muyan-Özçelik, Jeff M. Ota, Xiaoyun Wang, Carl Yang, Evangelos Georganas, Louis Feng, Ehsan Ardestani, Jaewon Lee, John Lundell, Changkyu Kim, Arun Kejariwal, Mingyu Liang, Wenyin Fu, Pavani Panakanti, Shengbao Zheng, Srinivas Sridharan, Christina Delimitrou, Ning Sun, Yufei Zhu, Pallab Bhattacharya, Xizhou Feng, and Valentin Andrei. I really enjoy working

with each of you and I am so happy that our collaboration ends up or is going to end up to be fruitful. In particular, I would like to thank Pınar for leading me to the road of academic research and publication (together with John), Evangelos for supporting me with research insights and visions during my hardest time, and Louis for generously spending so much time caring about my research and internship while your schedule has been so tight all the time.

Owensgroup is unparalleled, and our colleagues are exceptional. It is an amazing experience to work with so many talented and hard-working people: Serban Porumbescu, Matthew Drescher, Matthew Yih, Weitang Liu, Yuechao Pan, Carl Yang, Leyuan Wang, Kerry Seitz, Muhammad Awad, Jason Mak, Muhammad Osama, Afton Geil, Vehbi Eşref Bayrktar, Collin McCarthy, Ahmed H. Mahmoud, Yuxin Chen, Jonathan Wapman, Agnieszka Łupińska, Radoyeh Shojaei, Chuck Rozhon, Toluwanimi Odemuyiwa, Daniel Loran, Cameron Shinn, Marjerie Suresh, Mythreya K., Annie Robison, Chenfei Yao, Eric Yuan, Nima Johari, and Teja Aluru. I will miss the inspiring discussion and every week's group meeting with all of you.

This year (2023) marks one of the most difficult years for job hunters. I am very thankful to Honglei Liu, Jingyue Wu, Joshua Hasten, Muhammad Awad, Muhammad Osama, Nuwan Jayasena, Wenjian Hu, Xiaoyun Wang, Yuduo Wu, and Yuechao Pan for either referring me to a job or having mock interviews with me. I can never reach the goal without your help.

I also want to thank the past and current staff of UC Davis and the ECE department, including Robert Nagel from SISS, Kyle Westbrook, Sacksith Ekkaphanh, Fredrick P. Singh, Jennifer Y. Torres, Carole Bustamante, and Michelle T. Walker from ECE. You make things run smoothly for students and faculty so that we can focus on our research. You are all heroes.

Personally, as a music and piano lover, I am so grateful to encounter the magical piano music composed by Issac Albeniz and Nikolai Medtner during the past six years. Albeniz's creativity and freedom and Medtner's depth and abundance are the best fuel whenever I get stuck at work. You guys are genius. Among so many composers you are the two who can always get me excited. I feel sorry for both of you being significantly underestimated over the years, and I believe you will very soon earn the fame and recognition that you well deserve.

Though being physically far away for years, my family is always the closest to my heart. I want to thank my parents, Liqun and Dan, for their endless love, care, and support since the

first day I came to this world. I feel so lucky to inherit your exploration spirit, father, and your persistence, mother. These are the things that shape me into what I am today: I am part of you, and you are part of me. I want to thank all my family members, including my parents-in-law, for your continuous support all these years. Doing a Ph.D. is never a solo job, and you are all my strongest backup. I also want to thank my late maternal grandmother and paternal grandfather, who introduced me to this world when I was a kid. At this moment, I cannot help to recall the story of the Water Margin you had told me, Grandma, and the coins of ancient China and Caribbean countries you had shown me, Grandpa. May you rest in peace.

Finally, Ruixuan Zhang, my wife, my love, my special one and treasure. If there is any role that is more stressful than a Ph.D. student, a Ph.D. student's partner might be one of them. I have been wondering if I can do as well as you if we switch roles; you just cannot do any better. You made me a better person with your kind heart, your persistence, and most importantly, your love. With all the feelings in my heart that are not able to be expressed by words, I just want to say, to make your life happy is my responsibility and biggest pleasure, and I have the rest of my life ready for it.

Chapter 1

Introduction

Machine learning (ML) has experienced an incredible boom during the past decade. Starting from AlexNet's [49] breakthrough on image recognition in 2012, people's life has been changed by all types of ML-based applications such as online shopping with recommendations, smart security system, AI assistant, autonomous driving cars, etc. As the fundamental of these applications, massive and efficient computation enabled and accelerated by ubiquitous ML software and hardware system is indispensable. Therefore, along with the development of advanced ML algorithms, understanding and optimizing ML system performance are topics in the realm of ML that also inspire numerous research.

The optimization of ML systems aims for one or multiple targets, such as lower latency, higher throughput, higher scalability, and lower memory cost for ML model execution. Recent research has proved that techniques of ML system optimization, including but not limited to quantization, pruning, operator (op) fusion, data/model parallelism, and kernel auto-tuning, can bring significant benefits to one or many of these goals. However, compared to "how to optimize" it turns out "*what to optimize*" and "*why optimize*" are questions that are sometimes even harder to answer. This is mainly because modern ML system problems evolve rapidly and become complex as they stretch and expand across multiple dimensions. In this circumstance, correctly identifying the bottleneck and properly linking it to the optimization might be perplexing. Specifically, there are several factors that contribute to this dilemma:

- **It is difficult to get a holistic view when characterizing ML workloads.** Although

modern profiling tools make the characterization handy and accessible, interpreting and utilizing the results still requires a thorough understanding and expertise of devices, system setup, as well as the tools themselves.

- **Benchmarking and debugging can be costly.** ML models, especially large ones such as natural language processing (NLP) models and recommendation models (RM), are cumbersome in terms of training and debugging. Although big tech companies may own tens of thousands of GPUs in their private cloud, it is not always convenient for an ML practitioner to run a newly developed ML model on 64 or 128 GPUs at any time just to get an idea of if it executes correctly and/or efficiently.

- **Optimization can be complicated and time-consuming and should be proceeded with caution.** Due to the complexity of the application (ML models with hundreds of ops), software systems (ML frameworks and compilers with tens of compiler passes and/or abstraction levels), and hardware systems (various types of heterogeneous devices, including single-GPU and multi-GPU platforms, CPUs, FPGAs, and customized AI chips), as well as the tight link among them, problems tend to span across multiple hierarchies of the optimization stack, e.g., optimizing a single op in the model ends up in making changes to the whole framework/compiler stack. No optimization effort should be made until its effectiveness is justified.

Therefore, the problems of understanding the ML workload performance bottlenecks and optimizing them are equally critical to solving. A fast, low-cost, and generalizable way to obtain insights into an ML workload is required to ensure optimization not only brings efficiency but is also done efficiently, and that is when performance models come to the stage.

A *performance model*, by definition, is one or a series of functions that take in features of a workload such as the problem size and operation dependency, and *quickly* estimates the characteristics of the workload's performance (latency, throughput, etc) on a specific computing platform. The complexity of performance models varies from one single function to a complex software system formed by multiple modules, and because of that, one performance model can serve to predict the performance characteristic of one single operation, such as matrix

multiplication, convolution, and matrix decomposition, up to that of a complicated workload such as an ML model consisting of hundreds of operators. Performance models can quickly capture the performance characteristics of ML workload and generate insights for optimization. With an accurate performance model, one can quickly evaluate the ML model execution efficiency *without actually running it on hardware*, identify possible bottlenecks, and proceed with well-guided optimization. This can result in the benefits of enormous budget and emission savings by significantly reducing engineer work time and compute resource waste.

In this dissertation, we take a closer look at how performance models are constructed and used to assist in the performance optimization of ML workloads. We talk about modeling the performance of ML workloads, and we also talk about how it is linked to optimization and how the optimization is actually done. We first start with using the roofline model [121], a powerful and straightforward performance model to identify kernel/layer fusion candidates in convolution neural networks (CNN). Layer fusion, i.e., combining two convolution layers into a single mathematically-equivalent layer, is one of the most commonly used optimization techniques for ML model execution. We demonstrate the performance benefits and tradeoffs of fusing two convolutional layers on multicore CPUs. We analyze when and why fusion may result in runtime speedups, and study three types of layer fusion: (a) 3-by-3 depthwise convolution with 1-by-1 convolution, (b) 3-by-3 convolution with 1-by-1 convolution, and (c) two 3-by-3 convolutions. We show that whether fusion is beneficial is dependent on numerous factors, including arithmetic intensity, machine balance, memory footprints, memory access pattern, and the way the output tensor is tiled. We devise a schedule for all these fusion types to automatically generate fused kernels for multicore CPUs through auto-tuning. With more than 30 layers extracted from five CNNs, we achieve a 1.04x geomean with 1.44x max speedup against separate kernels from MKLDNN, and a 1.24x geomean with 2.73x max speedup against AutoTVM-tuned separate kernels in standalone kernel benchmarks. We also show a 1.09x geomean with 1.29x max speedup against TVM, and a 2.09x geomean with 3.35x max speedup against MKLDNN-backed PyTorch, in end-to-end inference tests.

Next, we showcase how an end-to-end (E2E) performance modeling pipeline is built to predict the per-batch training time of ML workloads. We devise a performance model for GPU

training of Deep Learning Recommendation Models (DLRM), whose GPU utilization is low compared to other well-optimized computer vision (CV) and natural language processing (NLP) models. We show that both the device active time (the sum of kernel runtimes) but also the device idle time are important components of the overall device time. We therefore tackle them separately by (1) flexibly adopting heuristic-based and ML-based kernel performance models for operators that dominate the device active time, and (2) categorizing operator overheads into five types to determine quantitatively their contribution to the device active time. Combining these two parts, we propose a critical-path-based algorithm to predict the per-batch training time of DLRM by traversing its execution graph. We achieve less than 10% geometric mean average error (GMAE) in all kernel performance modeling, and 4.61% and 7.96% geomean errors for GPU active time and overall E2E per-batch training time prediction with overheads from individual workloads, respectively. A slight increase of 2.19% incurred in E2E prediction error with shared overheads across workloads suggests the feasibility of using shared overheads in large-scale prediction. We show that our general performance model not only achieves low prediction error on DLRM, which has highly customized configurations and is dominated by multiple factors but also yields comparable accuracy on other compute-bound ML models targeted by most previous methods. Using this performance model and graph-level data and task dependency analysis, we show our system can provide more general model-system co-design than previous methods.

Last, we extend this single-GPU performance model to distributed multi-GPU platforms, which incur multiple new problems that do not exist in the single-GPU work. They include 1) complicated synchronization and load-balancing scenarios caused by input data distribution variance, 2) communication networks (e.g., NVLink, PCIe) of different topologies that connect multiple compute devices, and 3) flexible training configuration. On top of the previous works for single-GPU platforms, we address these problems to enable multi-GPU performance modeling by including 1) data-distribution-aware performance models for embedding table lookup and 2) data movement prediction of communication collectives into our upgraded performance modeling pipeline equipped with inter-and intra-rank synchronization for ML workloads trained on multi-GPU platforms. Aside from accurately predicting the per-iteration training time of DLRM models with random configurations with a geomean error of 5.21% on two multi-GPU

platforms, the prediction pipeline also generalizes well to other types of ML workloads such as Transformer-based NLP models with a geomean error of 3.00%. Moreover, it is capable of generating insights such as evaluation of embedding table sharding configuration selection for ML workloads by only consuming the execution graphs of them as input without actually running them on the hardware.

Chapter 2

Background

2.1 Single Ops in ML and DL

2.1.1 Convolution

Convolution is one of the most dominating operations in ML and DL, especially CNN. 2D convolution is commonly used for feature extraction in image classification, detection, and segmentation applications, while 3D convolution applies to tasks like medical imaging and point cloud processing for autonomous vehicles. Convolution can be executed with different tensor input formats, e.g., NCHW, NHWC, NCHW[x]c, etc, while the choice of formats is platform and implementation dependent. The listing below 1 shows the vanilla C++ code of 2D convolution with NCHW format.

2.1.2 General Matrix Multiplication (GEMM)

GEMM is another operation that dominates ML applications like CNNs (as fully-connected layers and the mathematically equivalent 1x1 convolutions in NHWC format) and NLP models such as Transformer [114], etc. Similar to convolution, GEMM also has multiple optional input formats, among which column-major and row-major are the most common ones. The listing below 2 shows the vanilla C++ code of GEMM with the row-major format.

2.1.3 Sparse Tensor Operations

Sparse tensor operations, e.g., sparse GEMM (SpGEMM) and sparse matrix dense vector multiplication (SpMV), are popular operations in NLP and RM models as well as sparse CNNs.

```c
bool out_of_bound(int x, int X) { return x < 0 || x >= X; }
// Input and output tensors
float* Input;   // N * IC * HI * WI
float* Filter; // OC * IC * F * W
float* Output; // N * OC * HO * WO
// <- Pad the input tensor with 0s if needed
for (int n = 0; n < N; n++) {
  for (int oc = 0; oc < OC; oc++) {
    for (int ho = 0; ho < HO; ho++) {
      for (int wo = 0; wo < WO; wo++) {
        // Accumulation
        float temp = 0.0;
        for (int ic = 0; ic < IC; ic++) {
          for (int x = -F/2; x <= F/2; x++) {
            for (int y = -F/2; y <= F/2; y++) {
              int hi = ho * stride + y;
              int wi = wo * stride + x;
              if (!out_of_bound(hi, HI) && !out_of_bound(wi, WI))
                temp += Input[n][ic][hi][wi] * \
                        Filter[F/2+y][F/2+x][ic][oc];
            }
          }
        }
        Output[n][oc][ho][wo] = temp;
      }
    }
  }
}
```

Listing 1: 2D convolution with NCHW format.

In such operations, one or multiple matrices/tensors among the input and output are sparse, and they usually appear in representations such as CSR, CSC, and COO. Compared to the previous two dense operations, sparse tensor operations are more likely to be memory-bound than compute-bound. In NLP and RM models, embedding table lookup, a variation of SpGEMM that incur no multiplications, is commonly used for sparse input data processing. Algorithm 1 shows the embedding table lookup operation with sparse input matrix A, dense input matrix B, and dense output matrix C.

```
float* A; // M * K
float* B; // K * N
float* C; // M * N
for(int i = 0; i < M; i++) {
    for(int j = 0; j < N; j++) {
        // Accumulation
        float sum = 0.0;
        for(int k = 0; K < n; k++) {
            sum += A[i*K + k] * B[k*N + j];
        }
        C[i*N + j] = sum;
    }
}
```

Listing 2: GEMM with row-major.

Algorithm 1 Embedding table lookup operation.

1: **Input:** A, sparse matrix with row indices of B matrix to lookup.
2: **Input:** B, embedding table as a dense matrix of size $K * N$.
3: **Output:** C, dense output matrix of size $M * N$.
4: **for all** $a \in A$ **do**
5: $\quad row \leftarrow row_of(a, A)$
6: $\quad$ **for all** $b \in B[a]$ **do**
7: $\quad\quad col \leftarrow col_of(b, B)$
8: $\quad\quad C[row][col] \leftarrow C[row][col] + b$
9: $\quad$ **end for**
10: **end for**

2.2 Compute Devices and Compute Kernel Libraries

In the past decade, Graphics Processing Units (GPU), especially NVIDIA's, have evolved for several generations, and are unarguably the superstar in the wave of ML. GPUs adopt a highly parallel architecture based on streaming multi-processor (SM) with shared memory and multiple CUDA cores, while these SMs share high-speed L2 and global memory bandwidth and bus that enable the fast access and transaction of data. Such an architecture makes GPUs outstanding in carrying compute-intensive tasks including ML in many use cases. Based on specific task requirements and constraints, ML models are also commonly deployed on different other compute devices such as Intel and AMD CPUs, Google's Tensor Processing Units (TPU) [45] (both for training and inference), FPGAs, IoT devices such as Raspberry Pi and NVIDIA's Jetson Nano,

and mobile devices such as cell phones and smart swatches (all for inference), all come with different architectural designs. As a typical example of ASIC, TPUs revive the architectural features of systolic arrays [50], etc for cost-effective and high-performance computation of matrix. We expect to see various ASICs being commercialized and running AI applications on them in the recent future, thanks to the call of hardware/software co-design in research that results in a booming of AI-chip-making startups, including GraphCore, Groq, and Cambricon, etc.

Optimizing a compute kernel such as GEMM and convolution for a certain device is a highly specialized job that requires expertise in computer architecture, algorithms, mathematics, and engineering. This explains why many DL frameworks call and execute the compute kernels from proprietary and/or open-source high-performance computation libraries instead of providing their own implementations. NVIDIA's cuDNN [15] and cuBLAS [84] are the two most popular ones adopted by most of the frameworks including Tensorflow and PyTorch that execute ML models on the GPU. They are both proprietary in that only the kernel APIs rather than the actual implementations are accessible to users, and they both have heuristics internally to pick the best implementations given the problem size. Besides these two libraries, NVIDIA also provides cuSparse for sparse matrix and tensor operations such as SpGEMM and SpMV, CUTLASS that is open-sourced, templated, and targets both GEMM and convolution and cuML [82] that accelerates traditional ML algorithms by calling cuSolver and cuBLAS in the low level. Widely-used ML compute kernel libraries also include oneDNN (previously MKL-DNN) [39], LIBXSMM [33], and clDNN. Among them, oneDNN and LIBXSMM execute on both Intel and AMD CPUs, while clDNN executes on compute platforms that are OpenCL compatible, including GPUs from NVIDIA, Intel, and AMD, as well as some types of FPGAs.

2.3 ML Models and Applications

The latest surge of artificial intelligence (AI) and machine learning (ML) led by the emergence of deep learning (DL) originates from the success of AlexNet [49] that championed the ILSVRC contest for image classification in 2012. Empowered by the tremendous amount of training data and vast compute power, both academia and the industry witnessed rapid development in the areas

of natural language processing (NLP), computer vision (CV), and large-scale recommendation models (RM).

2.3.1 Natural Language Processing

Recurrent Neural Networks (RNN), particularly Long short-term memory (LSTM), has been the main approach for NLP for years since NNLM [7] and RNNLM [74]. However, RNNs have a series of problems, including being non-parallelizable, requiring the same input and output lengths (for Vanilla RNN), gradient vanishing/exploding in training, and not being able to detect inter-word relationships. Word2Vec [75] is proposed to solve the last problem above the idea of continuous bag-of-word (CBOW), with which a trained model can detect synonyms of words. The rest of the problems are solved by Transformer [114] which is no less than a revolution to this research area. Transformer introduces the encoder-decoder structure, both parts of which can be constructed by stacking general matrix multiply (GEMM) ops and parallelized with flexible length inputs. Transformer's successors, like GPT-2 [89], BERT [19], RoBERTa [62], and GPT-3 [8], are fueled with a massive amount of training data to continue empowering NLP tasks such as machine translation, chatbot, and code writing, etc.

2.3.2 Computer Vision

After AlextNet, VGG [102], Inception V3 [107], ResNets [31], Yolo [95], SSD [61], and UNet [97] mark several milestones of the development of convolution neural networks (CNN) that push the limit of image classification, recognition, and segmentation on servers and other compute platforms, while SqueezeNet [38] and MobileNet (V1 [37], V2 [98]) enable efficient inference on mobile and IoT devices. Though dominated by convolution, the introduction of operators (op) like batch-normalization (batch-norm/BN) [40], dropout [106], pooling, depthwise-convolution [37] and various activations e.g., ReLu, Sigmoid, etc to CNNs increase their ability to generalize over training data as well as computation efficiency. To further overcome the limitation of manual model design, a trend of searching for the network architecture with the best performance is led by NAS [136] proposed by Google, and followed by MNasNet [109], EfficientNet [108], etc. In recent research such as ViT [20], we also see the adoption of Transformer-like architectures in CV tasks that lead to decent outcomes.

2.3.3 Recommendation Models

RMs can be roughly divided into two categories: content-based filtering and collaborative filtering, based on the interaction type of features of the training data. As they both have obvious drawbacks, e.g., content-based filtering is not able to link the preferences of similar users, and collaborative filter's cold-start problem, it is natural to combine the two ideas and create hybrid RMs. Factorization Machines [96] proposed for this purpose leads to its further combination with the neural network, the idea of which is introduced by NCF [32] in 2017, in DeepFM [29] as an improvement of Google's Wide-and-Deep [14], one of the first neural networks with hybrid recommendation approach. Facebook releases DLRM [78] that further simplifies DeepFM and emphasizes more practical issues such as parallel training, etc [68]. From the compute kernel point of view, modern RMs are mainly dominated by embedding lookup op and GEMM op, for the purposes of processing sparse and dense features, respectively.

2.4 Frameworks for ML and DL

While some of the early frameworks such as Theano [3] that support multi-dimension tensor operation for scientific computing purposes can be dated back to the 2000s, Theano created Caffe [43] created by UC Berkeley in 2014 is the first commonly used DL-specific framework that provides expression and fast execution for DL models on various architectures. In the next few years, PyTorch [86], Tensorflow [1], Keras [16], PaddlePaddle [66], MXNet [9], CNTK [99], and Neon [6], etc, join this feast of DL frameworks, while the first four of them remain active in today's academia and industry. DL frameworks are always built with a granularity of single-op that are bound to compute kernels in the low level as the backends, while these ops are connected to construct a computational graph for a certain task. In terms of the construction of a computational graph, DL frameworks can be divided into two categories: static and dynamic. Frameworks like Tensorflow build the graph statically to avoid rebuilding in runtime and thus are more efficient in execution, while others, with PyTorch as a typical example, dynamically build the graph and are more debug-friendly and flexible, i.e., the graph is rebuilt after each iteration so that it is more suitable to handle tasks such as NLP with variable length inputs [79]. Some of the DL frameworks do not have their own backend, e.g., Keras was initially built as a wrapper

for Tensorflow and later added support for backends from Tensorflow, Theano, and CNTK, while a few others, such as Uber's Horovod [100], wraps an existing framework (Tensorflow) and provides its own enhancement for a certain engineering purpose (e.g., multi-GPU training).

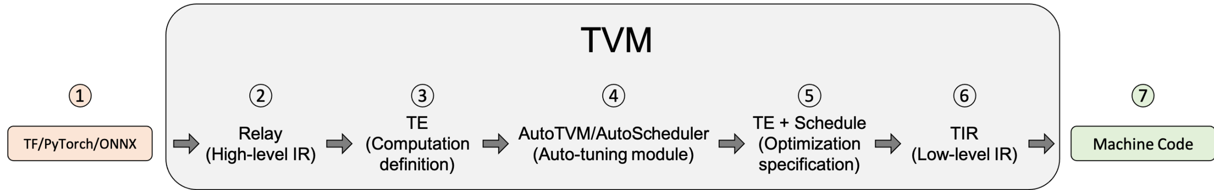


Figure 2.1: Overview of the TVM pipeline.

2.5 DL Compilers

DL compilers are designed to facilitate the optimization of kernels and model workloads for ML and DL. Computational graphs and ops are lowered to multiple levels of intermediate representations (IR) for loop optimization and/or polyhedral analysis so as to generate high-performance machine code. XLA [27] is one of the earliest of them, which uses JIT compilation and op fusion techniques to generate high-performance native machine code for DL workloads created with Tensorflow. Inspired by the idea of decoupling compute and schedule from Halide [90], a C++-based domain-specific language for parallel image and array processing, Chen et al. create TVM [10] (initially with HalideIR) as an LLVM-based DL compiler stack that enables auto-tuning [11], auto-scheduling [133], and quantization [41] of op kernels, and accelerates the deployment of ML workloads (for mainly inference purpose) on different computing platforms. Facebook's historical project TensorComprehension (TC) [113] is also built on Halide IR and incorporates polyhedral JIT compilation for code optimization. TC is also one of the earliest DL compilers that provides kernel auto-tuning functionality. As a DL framework, PyTorch also embraces some of the DL compiler's features like JIT compilation in its submodules such as TorchFX and TorchScript for code generation. The infrastructure of DL and tensor compilers, such as MLIR [51] and TPP [25], is also being rapidly developed to provide multi-level abstractions for efficiency and portability of ML workloads across devices.

Chapter 3

Towards Flexible and Compiler-Friendly Layer Fusion for CNNs on Multicore CPUs

3.1 Introduction

Convolutional neural networks (CNNs) have played an increasingly important role in research and industry for the past decade. CNNs are constructed with a series of layers/operators (*ops*). In the vast majority of CNN implementations, ops have a one-to-one correspondence with compute kernels. For reduced memory requirements and/or better producer-consumer locality, production CNNs perform *fusion* for certain ops, i.e., combining two or more neighboring ops into one and computing it with one single kernel. At its core, the fusion problem is a balancing problem between computation and communication, or between the communication of different memory hierarchies, with the hope that reducing main-memory communication will result in only modest amounts of extra computation or data movement in high-level memory and hence an overall speedup. In particular, today's deep learning (DL) frameworks and compilers (e.g., TVM [10], Tensorflow [1], PyTorch [86], and MXNet [9]), and proprietary kernel libraries like cuDNN [15] and MKLDNN often trivially fuse a convolution layer with the following element-wise layers to save the cost of data movement. Other fusion opportunities such as parallel convolution branch fusion for structures in models like Inception-V3 [107] can be realized by methods like relaxed graph substitution, as proposed by Jia et al. [44].

More complex is the fusion of neighboring *complex ops*, e.g., convolution (conv) and depthwise convolution (dw-conv). As these ops often appear as the hotspot of CNNs, which

are compute and/or memory bandwidth-intensive, fusing them is a potential way to further enhance compute performance. Showing performance improvements for this fusion type is a significant challenge for three reasons. First, unlike the aforementioned fusion types, simple techniques like inlining or data concatenation and split (to reuse existing proprietary libraries) would not work due to the memory access pattern of these ops. Instead, new compute kernels are necessary, and the optimization space of a fused op is so complex, with so many parameters, that a straightforward search would be impractical. Second, the standalone kernels used as a performance baseline are already highly optimized. Finally, integrating a fused kernel into current DL frameworks is also difficult, as modern frameworks are primarily designed at the granularity of one op that maps typically to one kernel.

In this paper, we focus on three opportunities for the fusion of two consecutive complex ops: (1) 3-by-3 dw-conv with 1-by-1 conv, (2) 3-by-3 conv with 1-by-1 conv, and (3) two 3-by-3 convs, all of which are commonly seen in CNNs. The challenge we address in this paper is to show not just *where* we can show performance improvements from complex op fusion but also *why*. We first propose a set of tiling and scheduling principles to intelligently reduce the otherwise intractable parameter space and search for the combination that leads to the best performance. These principles can also be extended to problems with complex loop structures, including other fusion types. Based on these principles, we devise a schedule template for auto-tuning these fused kernels on multicore CPUs and propose an approach to integrating the fused kernels/ops into DL compiler pipelines. Both of these ideas can be adopted by production DL compilers. We implement the fused kernels by incorporating LIBXSMM's [33] batch-reduce GEMM [24] micro-kernels and extending AutoTVM, TVM's auto-tuning tool, to search for the best schedule. We also integrate these kernels into TVM's compiler infrastructure for end-to-end tests by creating a new fused op. The code is open-sourced and can be found at `https://github.com/moderato/LayerFusion`.

We make the following contributions:

- We analyze the fusion of two complex ops and answer the question of when and why such fusion is beneficial.
- We propose a methodology with tiling and scheduling principles of composing fused

kernels for multicore CPUs, and implement kernels for three types of two complex-op fusion.

- We achieve 1.24X and 1.04X geomean speedup in a standalone kernel-level benchmark against TVM and MKLDNN respectively, and 1.09X and 2.09X geomean speedup in end-to-end tests, by testing with more than 30 workloads extracted from five real CNN models on three multicore CPU platforms.

3.2 Related Works

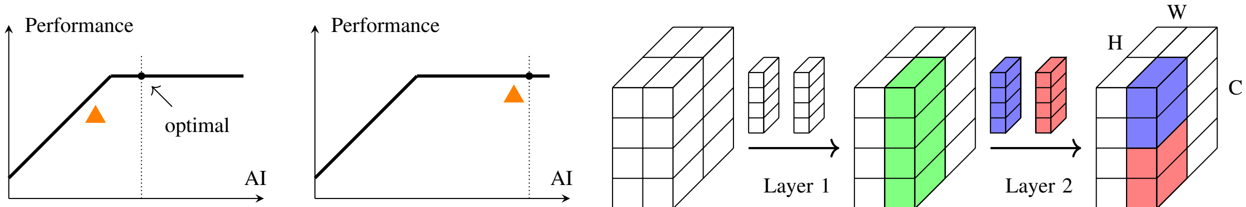
3.2.1 Kernel Optimizations and Auto Tuning

Convolutions are the ops that take the largest runtime in current CNNs. They can be implemented in many styles, e.g., direct convolution, im2col + GEMM [15], FFT-based [67], and Winograd-based [52], etc, on different types of compute devices. For direct convolution on CPUs, Georganas et al. [24, 26] proposed batch-reduce GEMM (BRGEMM) as a basic building block for tensor contractions and convolution and claimed to achieve better runtime performance than MKLDNN (now renamed to oneDNN). In our work, we adopt the BRGEMM micro-kernel implementation provided in the LIBXSMM [33] library as a building block for our fused kernels.

Auto-tuning is a common approach for optimizing kernel implementations and has benefited from continuous development through the years. Auto-tuning can be employed together with the idea of decoupling compute and schedule central to Ragan-Kelley et al. [91, 92] and developed by Mullapudi et al. [77] in Halide as well as Chen et al. [10, 11] in TVM, easing the process of developing high-performance implementations for DL workloads. Recently, the idea of autoscheduling [2, 133] breaks through auto-tuning's limit of relying on a well-composed schedule template by automatically searching for schedules. In this research we adopt AutoTVM as our tool for auto-tuning; in future work, we plan to also port our approach to auto-scheduling tools.

3.2.2 Layer Fusion

The fusion of multiple consecutive convolutions first appears in Alwani et al. [5]. Their kernel implementation is completely hand-tuned on FPGAs and the fusion space is explored with a



(a) The dotted lines are theoretical fused kernel AI. Orange triangles are empirical fused AI (efAI) derived from separate kernel stats. Fusion is likely: **(left)** beneficial, as efAI is not close to optimal; or **(right)** not beneficial as efAI is very close to optimal.

(b) Tiling across the channel axis (C) of the second layer results in recomputation on different cores: the blue and red tensors reside on different cores, while both cores need to compute the green tensor.

Figure 3.1: (a) Roofline model examples for fused kernels. (b) Explanation of recomputation in a fused kernel.

dynamic programming approach. Wang et al. [119] improved inference time on GPUs using a designated subset of fused convolutional layers. However, these works neither explore the large space of code optimization nor show a clear way how the fused kernels can be handily integrated into production end-to-end tests. Our work is the first that focuses on multicore CPUs and addresses the above issues.

3.3 Principles of Effective Layer Fusion

Consider a CNN model that runs on a multicore system, e.g., a multicore CPU. A well-optimized single-op kernel on this system will take advantage of all cores. Typically such a kernel will output a tensor and that tensor is divided into multi-dimensional *tiles*, with an equal number of tiles computed on each core in parallel. This tiling may be a spatial tiling (e.g., across one or more axes among N, H, W in a typical 4D activation tensor) and/or a tiling across channels (e.g., the C axis). At the end of the kernel, all data is written back to memory before the next kernel begins. The next kernel will then read its input from memory assuming the input is large enough and does not fit in any level of cache.

In this paper, we show performance gains from fusing kernels of two complex ops. The important contribution of this paper, however, is not the performance gains but instead why and how we achieve these performance gains. What kind of kernels should be fused, and how should we fuse them?

One of our key tools for analysis and verification is the roofline model [121], with which we

can determine if a kernel's performance is bound by memory or compute. We characterize the combination of the two separate kernels in the roofline model, with the total compute equal to the sum of the compute in the two kernels, and the memory requirements equal to the sum of the reads and writes for both kernels. If the fused-kernel result is memory-bound, the two kernels are *usually* good candidates for fusion, because fusion's primary benefit is saving the memory writes and reads between the two kernels, with the hope to replace slow (main memory) accesses with fast (cache) accesses. However, this is trickier in practice, since many successful fusions we perform tend to be compute-bound. We also see successful fusion for two compute-bound kernels (as we show in a later example) as well as failure for fusion involving extremely memory-bound kernels like 5-by-5 dw-conv. Nevertheless, the results from the roofline model are generally predictive. We can also refer to the empirical results of proprietary separate kernels to select workloads that might benefit from fusion. As shown in Figure 3.1a, fusion is likely beneficial if the derived empirical fused roofline is not too close to the peak throughput at the theoretical fused AI, as fusion moves the roofline towards the *upper right* if it speeds up. In contrast, fusion is likely not beneficial if the derived empirical fused roofline is almost optimal.

We begin by looking at the per-core output of the first kernel and the per-core input of the second kernel. For some pairs of kernels, these are identical, and we can simply concatenate these into one kernel in a straightforward fashion. More often, though, these two do not match. In these cases, writing to memory at the end of the first kernel serves two purposes. The first is to allow a reshuffling of data through the memory system (essentially, a permutation of the intermediate output tensor, distributed across cores). The second is to allow a broadcast so that the output of one core in the first kernel can serve as the input for multiple cores in the second kernel. If we implement a fused kernel, our implementation must either perform a significant amount of intra-core communication for data reuse or perform redundant recomputation. Notice that the memory footprint of fused workloads might fit in any level of cache or none of them. For the extreme case that the footprint fits in L1, there is no data movement cost of the intermediate output to reduce, as it always stays in this fastest cache. We discover that this almost never happens with real CNN workloads, and therefore, from this point of view, fusion is always worth trying.

We employ *tiling* and *scheduling* to find the balance between fusion and speedup. Tiling expresses the subdivision of tensor input/intermediate/output data in a way that allows effective and scalable parallel execution. The computation of each part of the output tensor is typically expressed as a series of nested loops, and we also have the freedom to schedule (i.e., reorder, split, merge, parallelize, etc.) these loops to optimize for locality and execution. Because the two unfused kernels are almost certainly highly optimized for the target architecture when using proprietary libraries, we must make near-optimal decisions for tiling and scheduling to achieve competitive performance with our fused kernel.

3.3.1 Tiling Principles

One of the most important decisions in tiling is choosing along which axis to tile. In our fastest kernels on CPUs, we prefer tiling the second layer along spatial axes to tiling along channel axes, because the latter requires (redundant) recomputation of layer-one entries that are input to multiple different tiles along layer-two's channel axis, as shown in Figure 3.1b. This rules out some compute-heavy kernels like the last few layers of ResNets (i.e., $res_4x/5x$ as examples), which typically tile along a (relatively long) channel axis.

The spatial axes of the second layer also need to offer sufficient parallelism for full tiling; without enough parallelism here, fusion does not make sense. In general, CNN kernel implementations on CPUs tend to pack tensors so that the channel axis is packed as vectors in the last dimension, so CPUs with longer vector length, e.g., AVX-512, are better candidates for fusion, since they exploit the parallelism along the channel axis and compensate for our reluctance to tile/parallelize that axis across cores. Also, more cores make fusion more attractive if batch size goes up and/or the spatial dimension is large since either of these cases exposes more potential parallelism.

Finally, for effective fusion, the tiles for the first and second kernels in separate cases should be comparable in size. Given a fixed-size cache, a significant mismatch in size between the two tiles reduces the opportunity for capturing producer-consumer locality, as it leverages the cache poorly. The $mnb1$ workloads shown later in Table 3.2 are examples of such a failure.

3.3.2 Scheduling Principles

Once we have determined our tiling, we turn to the problem of scheduling. A typical fused kernel in our pipelines of interest has on the order of a dozen loops as well as the option to split these loops. Any sort of exhaustive search over valid reorderings of these loops is virtually intractable. Yet our experience is that some search is necessary; the performance landscape of the many possible implementations is complex enough that auto-tuning is necessary to find the fastest fused kernel.

Our approach is to restrict the search space down to a manageable level by only searching over a subset of the loops. In particular, we do not attempt to change the order of the outermost and innermost loops as they either do not affect data locality or are fixed for optimal register usage within a micro-kernel. In contrast, we do search the remaining loops, whose reordering can have a significant impact on data locality. This will be discussed in the next section.

3.4 Implementation

We mainly focus on three types of 2-layer fusion: (1) 3-by-3 depthwise convolution (dw-conv) followed by 1-by-1 convolution (conv); (2) 3-by-3 conv followed by 1-by-1 conv; (3) two 3-by-3 convs. The first type occurs commonly in computationally lightweight CNN models, e.g., MobileNet-V1 [37], MobileNet-V2 [98], MNasNet-A1 [109], etc. The second and third types occur in computationally heavyweight CNN models like ResNets [31], etc. In this section, we first introduce how we compose schedules to generate kernels for these fusion types, followed by how these kernels are integrated into the TVM inference pipeline.

3.4.1 Kernel-level optimization

Often a part of a domain-specific compiler, modern autotuners usually input the *schedule* that we described above, which describes how the mathematical expression of an op is mapped to the hardware (e.g., loop orderings and manipulations, as well as the search for the split loop lengths and ordering combinations that lead to the best performance) to tune the kernel. This may result in many possible mappings and hence a large search space. For example, two axes of length 4 being split and reordered has a search space size of 3 (split of first axis) $\times$ 3 (split of second axis) $\times$ 2 (reordering) $= 18$. If two layers are naively fused, the search space

size grows exponentially and becomes intractable, even without considering the extra possibility of loop unrolling in the innermost loops. As an aside, though they do not affect the search space, an autotuned fuser must also efficiently integrate element-wise post ops like batch-normalization, ReLU, etc. that follow all complex ops except for the last one.

Algorithm 2 Fused kernel schedule template with BRGEMM micro-kernels.

1: **Inputs:** $input \in \mathbb{R}^{N \times IC_1 \times IH \times IW \times ic_1}$, $weights_1 \in \mathbb{R}^{OC_1 \times IC_1 \times FH_1 \times FW_1 \times ic_1 \times oc_1}$, $weights_2 \in \mathbb{R}^{OC_2 \times IC_2 \times FH_2 \times FW_2 \times ic_2 \times oc_2}$, optional post ops parameters, e.g., $bias_1 \in \mathbb{R}^{OC_1 \times oc_1}$, $bias_2 \in \mathbb{R}^{OC_2 \times oc_2}$
2: **Outputs:** $output \in \mathbb{R}^{N \times OC_2 \times OH \times OW \times oc_2}$
3: **Split** OH into H_t, H_o, and H
4: **Split** OW into W_t, W_o, and W
5: **Split** IC_2 into IC_o and IC_i
6: **for** fused($n = 0 \dots N-1$, $ht = 0 \dots H_t - 1$, $wt = 0 \dots W_t - 1$) **do**
7:     **Exhaustively search** the order of OC_2, IC_o, H_o, and W_o, and mark them as loop 1, 2, 3, 4, and the parallel loop as loop 0 ▷ e.g., IC_o is loop 2.
8:     **Arbitrarily pick** a loop x from loop 0, 1, 2, 3, 4 ▷ e.g., x is 3.
9:     **for** loop 1 **do**
10:         **for** loop 2 **do**
11:             **for** [**do**Sub-tensors compute here.]loop 3
12:                 **for** loop 4 **do**
13:                     BRGEMM micro-kernel for layer 1 sub-tensor
14:                 **end for**
15:                 Compute post-ops of layer 1 if necessary
16:                 **for** loop 4 **do**
17:                     BRGEMM micro-kernel for layer 2 sub-tensor
18:                 **end for**
19:             **end for**
20:         **end for** ▷ IC_o finishes.
21:         Compute post-ops of layer 2 if necessary
22:     **end for**
23: **end for**

Our goal is to achieve high performance on the fused kernel and meanwhile limit the search space to make searching tractable. We accomplish this by classifying loops into three categories: parallel loops, micro-kernel loops, and tunable loops. We determined robust, fixed strategies for the first two categories and thus reduce our search space to only searching for the optimal configuration of the third.

We choose to fix the ordering of the first two categories of loops, because reordering parallel

loops is trivial for batch size 1, while micro-kernel loops are mapped to ready-to-use micro-kernels with fixed loop order. In our schedules, we place the parallel loops at the outermost location, micro-kernel loops at the innermost location, and tunable loops in between. The skeleton of our implementation structure is shown in Algorithm 2 and Fig 3.2. We implement our kernel with the BRGEMM micro-kernels from LIBXSMM. Instead of using the common $NCHW$ or $NHWC$ formats for convolution, we use a packed format, e.g., $NCHW[x]c$ for feature maps, and $(OC)(IC)H_fW_f[x]ic[y]oc$ for weights, where $x, y = 8, 16, 32, 64\ldots$, for better data locality on CPUs [24]. At lines 3 to 5 of the algorithm, OH, OW, and the reduce loop IC_2 of the $output$ tensor are split into 10 loops including the unsplit N. Each loop is split so that the lengths of each sub-loop are factors of its length. We do not split OC_2 as tiling across it brings recomputation as we discuss above. The parallel loops, i.e., N, H_t, and W_t, are fused and parallelized across multiple CPU cores at line 7. Loops H, W, and $oc_{1/2}$ and reduce loops, including FH, FW, IC_i, and $ic_{1/2}$, are all micro-kernel loops expressed within BRGEMM micro-kernels. The remaining four loops, i.e. OC_2, IC_o, H_o, and W_o, are left as the tunable loops to be searched by the auto-tuner.

In our implementation, we exhaustively search for all the orderings of these four loops and at which loop layer we place the computation, i.e., $compute_at$. The sub-tensors of both layers are computed at a loop that is picked among any of these four loops or the fused parallel loop. From a cache point of view, the input feature map sub-tensor of layer one is distributed to different cores on the CPUs, while the weights of each layer are streamed to the cache. The sub-tensor output of layer one is computed with vanilla loop ordering if layer one is a dw-conv, or by calling the BRGEMM micro-kernel if layer one is a conv. Its size is inferred by the compiler given the output size of its consumer, i.e., (H, W, oc_2) is known. This sub-tensor is always complete, i.e., all its reduce loops are fully contracted, before it is consumed by layer two because otherwise, it incurs extra computation for each of the incomplete slices. Therefore, it is safe to insert any post-ops computation directly after it. Subsequently, this sub-tensor always stays in the cache and is consumed by layer two to produce a slice with size (H, W, oc_2). This slice is not necessarily complete as the loop x it computes could be inside the outermost reduce loop of all, IC_o. We insert the post-ops right after IC_o because they can only be computed when IC_o

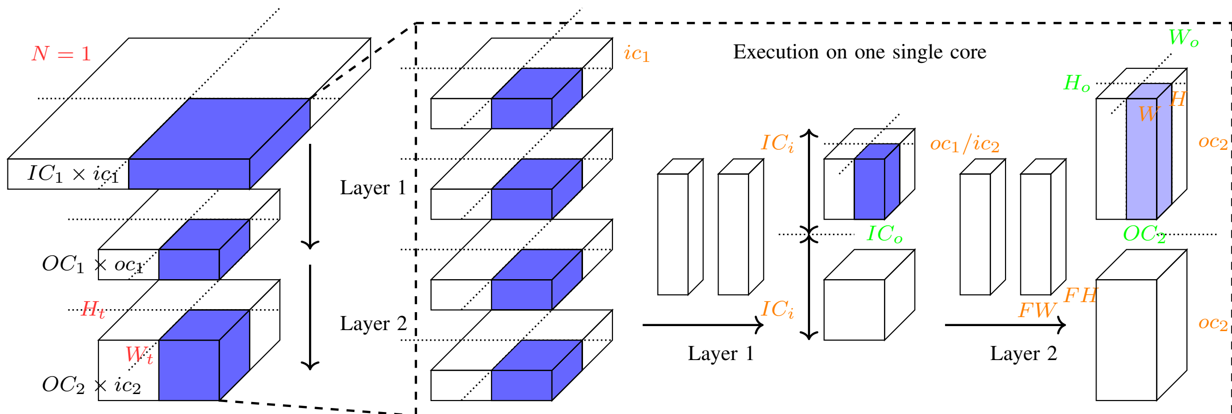


Figure 3.2: Algorithm visualization. (**Left**) Tensors are tiled for parallel multicore execution. (**Right**) A light blue (incomplete, as loop IC_o is not fully contracted) sub-tensor of size (H, W, oc_2) is computed from all blue (complete) sub-tensors. Red/green/orange mark the parallel/tunable/micro-kernel loops, respectively.

is fully contracted. Therefore, the sub-tensor of layer two always stays in the cache for proper tiling factors until it is complete and no longer needs access.

We found that grouping the loops in this way greatly limits the search space, making auto-tuning feasible, but still results in high-performance fusion. It can also be extended more generally to polyhedral problems including fusing other types of layers. In this case, a performant micro-kernel is necessary to serve as the core of the output schedule.

3.4.2 AutoTVM implementation of fused kernel

We implement this algorithm as an AutoTVM schedule. We first define TE compute functions for the fused layer workloads, then we create AutoTVM tuning tasks with these compute functions and schedules so that AutoTVM can auto-tune them using the XGBoost algorithm. For fast convergence, the schedule is tuned without post-ops being added since they do not affect the results, while when the tuning config is produced, we apply it to a new inference schedule where post-ops are added as necessary. The methodology can be handily extended to multiple layers with the (straightforward) addition of compiler support, as we can simply follow the same rule by only blocking the last layer being fused and stacking all previous layers at loop x. In fact, this methodology resembles the (manual) ‘pyramid’ method proposed by Alwani et al. [5], but we generate the fused kernel code automatically via auto-tuning. We also notice that at the cost

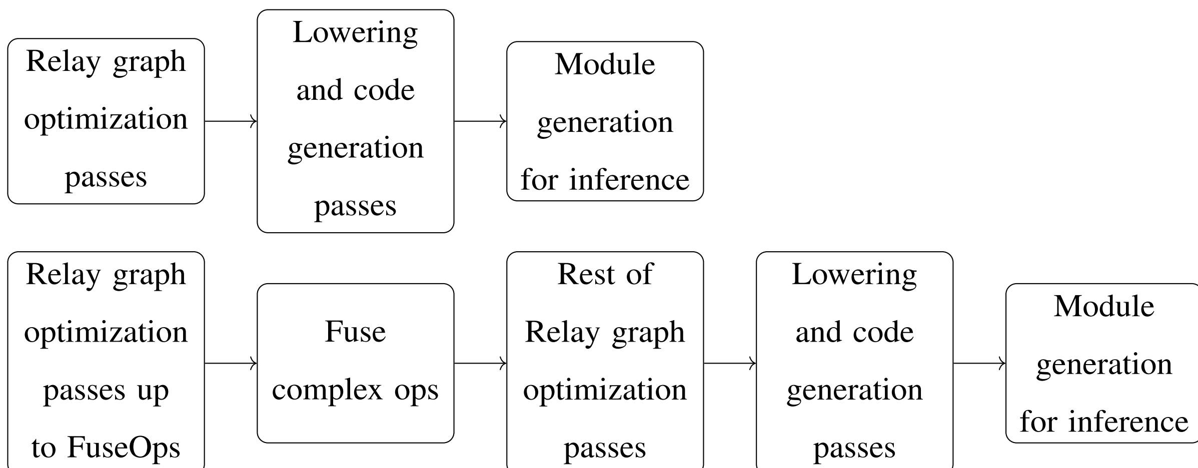


Figure 3.3: **Top:** TVM pipeline without complex-op fusion; **Bottom:** TVM pipeline with complex-op fusion.

of searching a much bigger space by further splitting the second loop group into eight loops and reordering them, higher fused kernel performance could potentially be achieved with better cache-blocking options. However, the advantage of this schedule against ours is marginal, which also requires both smartly designed heuristics and a more powerful autotuner. Hence, we leave the exploration of this idea as future work.

3.4.3 Compiler integration of fusion for end-to-end test

To integrate the fused kernel into end-to-end (full-CNN) tests, we need (1) a new op that defines the compute of the fused conv layers in a compute graph and (2) a compiler pass that is inserted into the pipeline to rewrite the graph for fusion. Both of these are missing in all DL frameworks. Also, post-ops like batch-normalization (BN) of the layers being fused need to be properly handled. Since BN ops are usually simplified to *bias-add*s with parameters such as *mean* and *gamma* folded into *weights* for inference, we do not keep them in the compute function for our *fused-conv2D* op. We simply keep the *bias* tensors of each layer together with the input and weights as the inputs to the *fused-conv2D* op, such that the new op is equivalent to a sub-graph of two fused layers and their post ops, e.g., dw-conv/conv+bias-add+relu+conv+bias-add+relu. As currently most DL frameworks fuse only element-wise ops, we must ensure that the complex-op-fusion pass is inserted *after inference simplification* where ops like BN are simplified, and *before element-wise op fusion* where all element-wise ops are fused.

Figure 3.3 presents how we leverage TVM's existing compiler pipeline to integrate our

design into it. Following the above principles, we create a new Relay op for the fused conv layers, as well as a new compiler pass to detect the fusible patterns and rewrite them with the *fused-conv2D* ops. In the TVM pipeline, a Relay compute graph is passed through passes that optimize the graph structure and then passes for code generation, and finally module generation. We insert the new *fuse-conv* pass right before the *fuse-ops* pass such that post-ops like *bias-add* ops are properly handled for the *fused-conv2D*, while post-ops are still normally handled for other unfused complex ops in the following *fuse-ops* pass. Our *fused-conv2D* op is also compatible with TVM's graph tuning [63] for layout optimization for CPU inference.

3.5 Results and Analysis

We extract 33 eligible layers with batch size 1 from five CNN models, including MobileNet-V1, MobileNet-V2, MNasNet-A1, ResNet-18, and ResNet-50. The full list of layers can be found in Table 3.2. We auto-tune each kernel for 4000 iterations using AutoTVM with XGBoost as the searching algorithm. We conduct both throughput and roofline analysis on these workloads and integrate them into the end-to-end tests of the five models. The experiments are conducted on one Intel Core(TM) i7_7700K CPU @ 4.2 GHz, one Intel Xeon quad-core Google Cloud Platform (GCP) server (unknown model with Cascade Lake microarchitecture) @ 3.1 GHz, and one AMD EPYC 7B12 quad-core GCP server @ 2.25GHz, all with Linux Ubuntu 18.04, Python 3.8, and PyTorch 1.8.1. We run our kernel-level experiments for fused kernels against separate kernels shipped by MKLDNN and those generated by AutoTVM as baselines, and model-level experiments against the baselines of AutoTVM + graph tuner as well as MKLDNN-backed PyTorch. For roofline analysis, we measure the DRAM bytes with PCM and measure the cache bytes and FLOP counts of the kernels with SDE.

3.5.1 Kernel-level Experiments

In Figure 3.4, we show the throughput of all the kernels on three platforms normalized with respect to MKLDNN in the standalone kernel benchmark. Overall, we achieve {geomean, maximum, and minimum} speed-ups of fused-kernel against MKLDNN-separate-kernels of {1.04X, 1.44X, and 0.53X}, and against TVM-separate-kernels of {1.24X, 2.73X, and 0.63X}. As expected, we see that workloads with big activations in their first layer such as $mv1_1$, $mv2_1$,

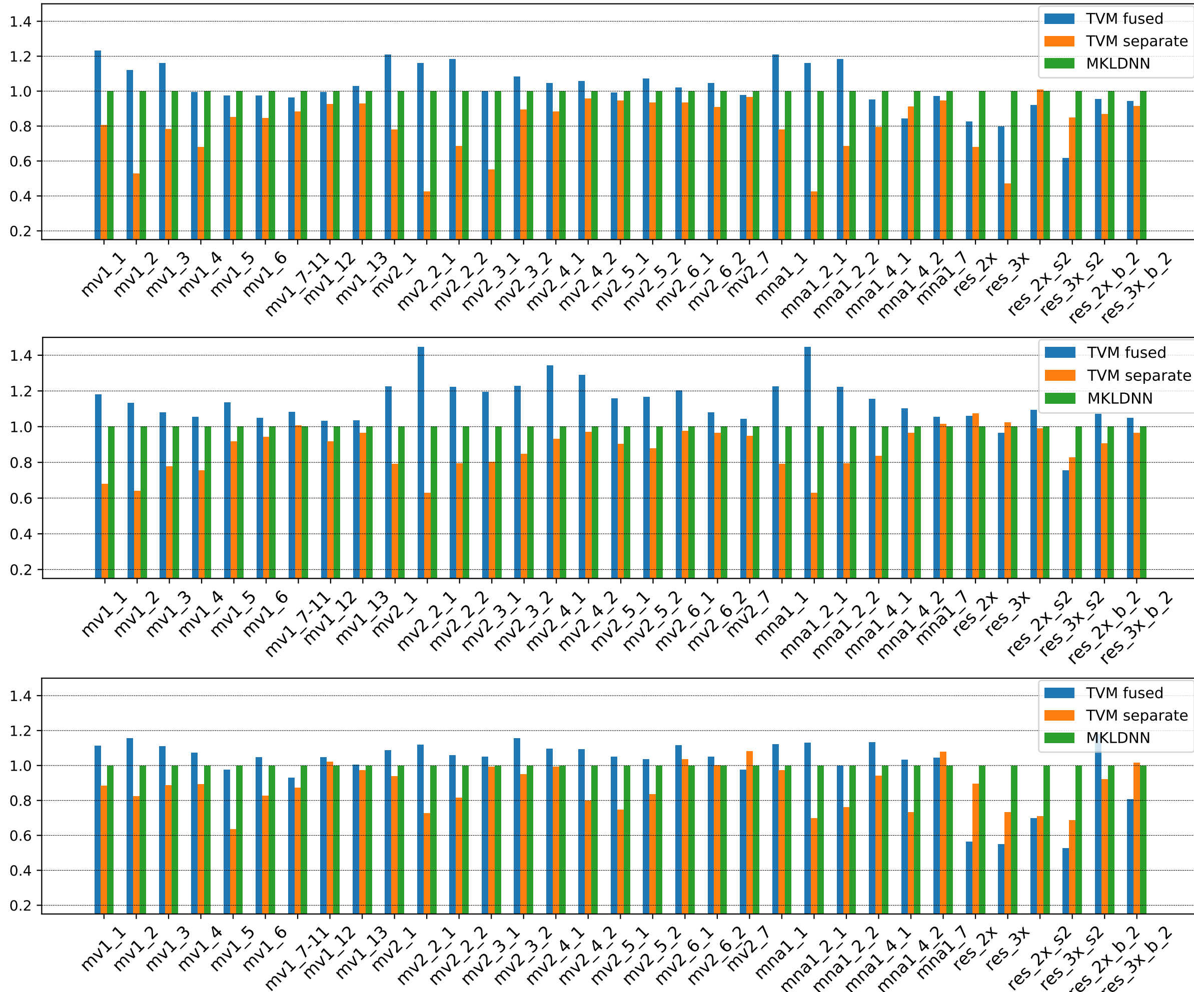


Figure 3.4: Throughput comparison between fused and separate kernels without post-ops, normalized with respect to MKLDNN. A "TVM fused" result of greater than one indicates our fusion delivers better performance than MKLDNN. **Top:** Intel i7_7700K (511 GFLOPS); **middle:** GCP Intel (711 GFLOPS); **bottom:** GCP AMD (413 GFLOPS).

$mv2_2_1$, etc. have higher speed-ups. Comparing all the MobileNet-V1 and MobileNet-V2 workloads, i.e., $mv1_x$ and $mv2_x$, we can see that although the spatial size of the activation changes almost the same way throughout the model, e.g., $112 \rightarrow 56 \rightarrow 28 \rightarrow 14 \rightarrow 7$, we see a higher overall fusion speed-up on $mv2_x$ than on $mv1_x$. This is because $mv2_x$ tends to scale *down* the channel axis between its fused layers, e.g., $mv2_1$ has 32 channels for its 3-by-3 dw-conv and only 16 channels for its 1-by-1 conv, while $mv1_x$ tends to scale *up*. Therefore, $mv2_x$ is generally less compute-bound than $mv1_x$ and in these cases, closer to the machine balance, as we will show later. We can thus suggest to model designers that without sacrificing accuracy, models can benefit more from fusion if the sizes of adjacent layers are designed to have more balanced AI. In addition, we still see a few examples where considerably compute-bound

ResNet workloads are sped up by fusion, which suggests that even compute-bound kernels could also benefit from our methods.

Across platforms, we observe that GCP Intel, the only AVX-512 CPU with the highest peak throughput in our evaluation, achieves the highest geomean speed-up (1.13X) on fused kernels against MKLDNN, versus (1.01X) and (0.99X) on the other two platforms. This implies the fact that fusion works better on platforms with higher peak throughput (making workloads "less compute-bound").

We also plot the roofline model of all the fused and separate kernels we test on the i7_7700K CPU in Figure 3.5. We treat each pair of TVM separate kernels and MKLDNN kernels as one standalone kernel and derive its AI as $\text{AI}_s = (\text{flop}_1 + \text{flop}_2)/(\text{bytes}_1 + \text{bytes}_2)$, where bytes_1 and bytes_2 are measured memory traffic for either DRAM or cache. We observe a trend that the roofline of the fused kernel gets closer and closer to that of the separate kernels for both DRAM and L2, especially in workloads from $mv1_1$ to $mv1_13$ in MobileNet-V1. This again verifies that fusion tends to get less benefit at later layers than earlier ones. Typically, for workloads in MobileNet-V2 and MNasNet-A1 such as $mv2_2_1$, $mv2_3_1$, etc, the fused kernels have a smaller advantage on AI compared to their predecessor or successor workloads that have either similar input or output feature sizes. This is because the bottleneck of these workloads is the stride-2 access of their input feature maps that reduces the data reuse; this bottleneck is not relieved by fusion. We also verify that overall the theoretical peaks DRAM AI for $mv2_x$ tend to be closer to the machine balance than $mv1_x$ so that they benefit more from fusion. We see that most of the fused kernels still do not reach the theoretical peak DRAM AI and incur extra DRAM accesses, which means there is still room for optimization by extending the schedule's search space. Notice that in all cases the L2 AI moves to the *right*, indicating that fusion increases data reuse in the L2 cache. For workloads being sped up, the L2 AI moves towards the *upper-right*, which matches our expectations. Among those not sped up, the rooflines of ResNet workloads are very close to the peak throughput and theoretical peak fused AI intersection, matching our previous synthetic examples that project such workloads are less likely to benefit from fusion.

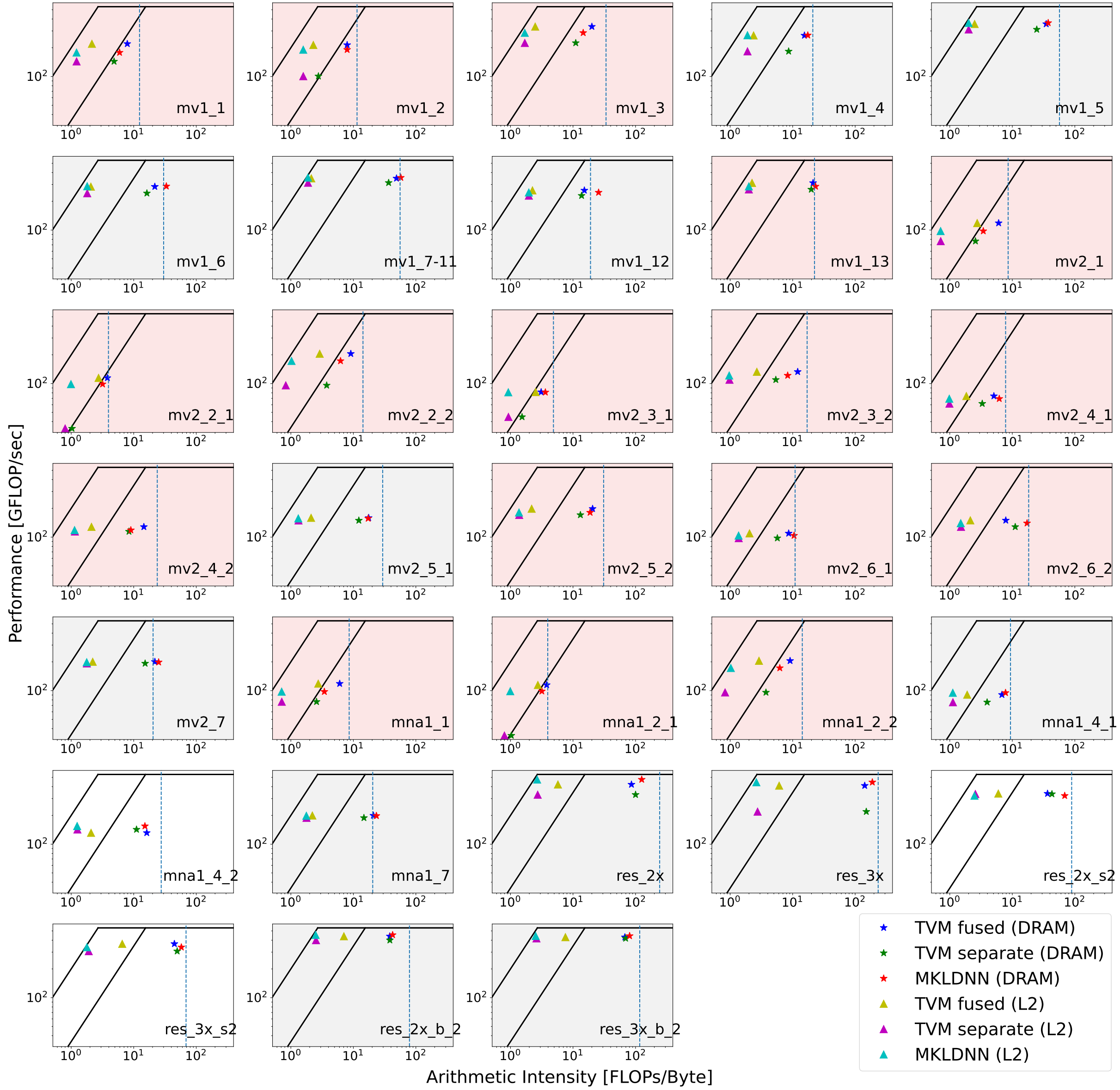


Figure 3.5: Standalone kernel rooflines of all the fused and separate kernels without post ops on the Intel i7_7700K CPU. In most cases, both DRAM and L2 AI of the fused kernel move towards the *upper-right* as fusion delivers better performance. The left and right slopes represent the L2 bandwidth (198.9 GB/s) and DRAM bandwidth (34.7 GB/s). The vertical blue dotted line represents the theoretical peak DRAM AI for the fused kernel. The red and grey background mark layers in which our fused kernel beats both and at least one, respectively.

3.5.2 End-to-End Experiments

We present the results of the end-to-end inference tests for the five models on three CPU platforms in Table 3.1. For each model, we tune variants of the compute graph that use a fused form of

Table 3.1: End-to-end inference time (in milliseconds) of five models on three CPUs. The shortest inference time across tools is shown in bold. Fusion doesn't apply to ResNet-18 on GCP AMD, since the fusion performance of two 3-by-3 convs is inferior to that of both TVM separate and MKLDNN.

CPU types	Models	TVM fused	TVM separate	PyTorch
Intel i7_7700K	MobileNet-V1	**3.38**	3.58	6.11
	MobileNet-V2	**2.44**	2.92	7.28
	MNasNet-A1	**3.31**	3.73	6.53
	ResNet-18	9.69	**9.35**	10.58
	ResNet-50	**20.42**	20.66	26.80
GCP Intel	MobileNet-V1	**2.77**	3.00	5.97
	MobileNet-V2	**2.35**	2.89	7.88
	MNasNet-A1	**3.42**	3.66	7.62
	ResNet-18	**7.17**	7.24	9.79
	ResNet-50	**16.18**	16.19	24.96
GCP AMD	MobileNet-V1	**7.72**	8.42	8.83
	MobileNet-V2	**4.65**	6.01	10.23
	MNasNet-A1	**6.79**	7.35	9.83
	ResNet-18	-	**15.38**	15.47
	ResNet-50	**38.86**	39.82	39.60

kernel pairs where we have seen a kernel-level advantage, and for layers that fusion does not have an advantage, e.g., *mv2_7*, *mna1_7*, etc., both fused and not fused, as the fused version might still perform better if layout transformations are needed for the unfused versions. Then we select the instance with the shortest inference time. In practice, we see only a few variants per model and this tuning step is short.

In all test cases except for ResNet-18 on i7_7700K and GCP AMD, fusion speeds up inference, with up to 3.35X against PyTorch for MobileNet-V2 on GCP Intel and 1.29X against TVM-separate for MobileNet-V2 on GCP AMD. We observe that the end-to-end speed-up is not exactly aligned with the aggregation of individual kernel speed-ups. In the standalone kernel benchmark, the cache is flushed for each iteration and the input data of layer one is always read from DRAM. But in end-to-end tests, the output data of a layer stays in the cache and may allow the next layer to benefit from the producer-consumer locality. Also, the framework itself might also affect the end-to-end results.

We see that the speedups of ResNets are marginal (up to 1.02X) on all three CPU platforms,

matching our expectation that fusion for compute-bound layers has limited benefit. We only fuse res_2x for ResNet-18, and res_2x and/or res_3x for ResNet-50, while it is the number of res_4x that primarily drives the depth of ResNets; thus we expect to see marginal benefit from kernel fusion on ResNets that have more than 50 layers.

Models	name	input	layer 1	layer 2
MobileNet-V1	mv1_1	(1, 112, 112, 32)	(3, 1, 1, dw-conv, relu)	(1, 64, 1, conv, relu)
	mv1_2	(1, 112, 112, 64)	(3, 1, 2, dw-conv, relu)	(1, 128, 1, conv, relu)
	mv1_3	(1, 56, 56, 128)	(3, 1, 1, dw-conv, relu)	(1, 128, 1, conv, relu)
	mv1_4	(1, 56, 56, 128)	(3, 1, 2, dw-conv, relu)	(1, 256, 1, conv, relu)
	mv1_5	(1, 28, 28, 256)	(3, 1, 1, dw-conv, relu)	(1, 256, 1, conv, relu)
	mv1_6	(1, 28, 28, 256)	(3, 1, 2, dw-conv, relu)	(1, 512, 1, conv, relu)
	mv1_7-11	(1, 14, 14, 512)	(3, 1, 1, dw-conv, relu)	(1, 512, 1, conv, relu)
	mv1_12	(1, 14, 14, 512)	(3, 1, 2, dw-conv, relu)	(1, 1024, 1, conv, relu)
	mv1_13	(1, 7, 7, 1024)	(3, 1, 1, dw-conv, relu)	(1, 1024, 1, conv, relu)
MobileNet-V2	mv1_1	(1, 112, 112, 32)	(3, 1, 1, dw-conv, relu6)	(1, 16, 1, conv, bias)
	mv2_2_1	(1, 112, 112, 96)	(3, 1, 2, dw-conv, relu6)	(1, 24, 1, conv, bias)
	mv2_2_2	(1, 56, 56, 144)	(3, 1, 1, dw-conv, relu6)	(1, 24, 1, conv, bias)
	mv2_3_1	(1, 56, 56, 144)	(3, 1, 2, dw-conv, relu6)	(1, 32, 1, conv, bias)
	mv2_3_2	(1, 28, 28, 192)	(3, 1, 1, dw-conv, relu6)	(1, 32, 1, conv, bias)
	mv2_4_1	(1, 28, 28, 192)	(3, 1, 2, dw-conv, relu6)	(1, 64, 1, conv, bias)
	mv2_4_2	(1, 14, 14, 384)	(3, 1, 1, dw-conv, relu6)	(1, 64, 1, conv, bias)
	mv2_5_1	(1, 14, 14, 384)	(3, 1, 1, dw-conv, relu6)	(1, 96, 1, conv, bias)
	mv2_5_2	(1, 14, 14, 576)	(3, 1, 1, dw-conv, relu6)	(1, 96, 1, conv, bias)
	mv2_6_1	(1, 14, 14, 576)	(3, 1, 2, dw-conv, relu6)	(1, 160, 1, conv, bias)
	mv2_6_2	(1, 7, 7, 960)	(3, 1, 1, dw-conv, relu6)	(1, 160, 1, conv, bias)
	mv2_7	(1, 7, 7, 960)	(3, 1, 1, dw-conv, relu6)	(1, 320, 1, conv, bias)
MNasNet-A1	mna1_1	(1, 112, 112, 32)	(3, 1, 1, dw-conv, relu)	(1, 16, 1, conv, bias)
	mna1_2_1	(1, 112, 112, 96)	(3, 1, 2, dw-conv, relu)	(1, 24, 1, conv, bias)
	mna1_2_2	(1, 56, 56, 144)	(3, 1, 1, dw-conv, relu)	(1, 24, 1, conv, bias)
	mna1_4_1	(1, 28, 28, 240)	(3, 1, 2, dw-conv, relu)	(1, 80, 1, conv, bias)
	mna1_4_2	(1, 14, 14, 480)	(3, 1, 1, dw-conv, relu)	(1, 80, 1, conv, bias)
	mna1_7	(1, 7, 7, 960)	(3, 1, 1, dw-conv, relu)	(1, 320, 1, conv, bias)
~~MNasNet-B1~~	~~mnb1_3_1~~	(1, 56, 56, 72)	(5, 1, 2, conv, relu)	(1, 40, 1, conv, bias)

	~~mnb1_3_2~~	(1, 28, 28, 240)	(5, 1, 1, conv, relu)	(1, 40, 1, conv, bias)
	~~mnb1_5_1~~	(1, 14, 14, 480)	(3, 1, 1, conv, relu)	(1, 112, 1, conv, bias)
~~MNasNet-B1~~	~~mnb1_5_2~~	(1, 14, 14, 672)	(3, 1, 1, conv, relu)	(1, 112, 1, conv, bias)
	~~mnb1_6_1~~	(1, 14, 14, 672)	(5, 1, 2, conv, relu)	(1, 160, 1, conv, bias)
	~~mnb1_6_2~~	(1, 7, 7, 960)	(5, 1, 1, conv, relu)	(1, 160, 1, conv, bias)
ResNet-18	res_2x	(1, 56, 56, 64)	(3, 64, 1, conv, relu)	(3, 64, 1, conv, bias)
	res_3x	(1, 28, 28, 128)	(3, 128, 1, conv, relu)	(3, 128, 1, conv, bias)
	~~res_4x~~	(1, 14, 14, 256)	(3, 256, 1, conv, relu)	(3, 256, 1, conv, bias)
	~~res_5x~~	(1, 7, 7, 512)	(3, 512, 1, conv, relu)	(3, 512, 1, conv, bias)
	res_2x_s2	(1, 56, 56, 64)	(3, 64, 2, conv, relu)	(3, 64, 1, conv, bias)
	res_3x_s2	(1, 28, 28, 128)	(3, 128, 2, conv, relu)	(3, 128, 1, conv, bias)
	~~res_4x_s2~~	(1, 14, 14, 256)	(3, 256, 2, conv, relu)	(3, 256, 1, conv, bias)
	~~res_5x_s2~~	(1, 7, 7, 512)	(3, 512, 2, conv, relu)	(3, 512, 1, conv, bias)
ResNet-50	res_2x_b_2	(1, 56, 56, 64)	(3, 64, 1, conv, relu)	(1, 256, 1, conv, relu)
	res_3x_b_2	(1, 28, 28, 128)	(3, 128, 1, conv, relu)	(1, 512, 1, conv, relu)
	~~res_4x_b_2~~	(1, 14, 14, 256)	(3, 256, 1, conv, relu)	(1, 1024, 1, conv, relu)
	~~res_5x_b_2~~	(1, 7, 7, 512)	(3, 512, 1, conv, relu)	(1, 2048, 1, conv, relu)

Table 3.2: Layer table. Input sizes are in (N,H,W,C) format, while layer configs are (filter HW, output channel or multiplier, stride HW, layer type, and post-op). Layers that do not benefit from fusion are crossed out and not shown in the result section.

3.6 Conclusion and Future Works

Individual kernels, such as those inside MKLDNN or NVIDIA's cuDNN, are highly optimized. They set a high bar for the implementation of fused kernels. One conclusion we draw is that the benefits of fusion are dependent on both the characteristics of the workloads and the CPU on which they are run, and the benefits of fusion are difficult to predict and, at this point, require actually implementing and running the kernels.

In future work, we hope to integrate the idea of fusion with TVM's autoscheduler [133] so as to leverage its power to search for high-performance schedules for fused layers. Next, we plan to target the compiler level to enable intermediate padding so that fusion can be extended to more layer types. Finally, we would also like to study fusion for the cases when the batch size is greater than 1.

In this chapter, so far we witness the use of the roofline model as a straightforward yet powerful tool to assist ML system optimization. Although not covered in this chapter, this methodology can be applied to the same problem (layer fusion) on GPUs and many other compute devices. For the specific problem of layer fusion, we only focus on inference as training requires some extra work (discussed in Section 6.2.3). However, training is still a broader and more impactful research topic than inference in general. In the next two chapters, we will discuss how to build a performance modeling pipeline at scale for training iteration time prediction and more tasks.

Chapter 4

Building a Performance Model for Deep Learning Recommendation Model Training on GPUs

4.1 Introduction

Recommendation models (RMs) have been widely deployed across various industries to improve user experiences and engagements in products and services. Examples include search [110], shopping [103], media consumption [18, 21], and social networking [105]. Driven by ever-increasing demands, training these models for better prediction rates has become both data- and computationally intensive by involving training data with hundreds of billions of samples, model sizes of up to multiple TBs [131], and multiple (often hundreds of) hosts and devices [132] for distributed training. This situation incurs high resource demands for development, debugging, and optimization, which significantly affects the productivity of ML engineers and the operation cost of data centers. Therefore, a performance model that accurately predicts an RM's training performance (e.g., speed, memory usage, etc.) based on its configurations (e.g., batch size, data sharding, number of layers, etc.) is very useful. It removes dependencies on hardware for some tasks and relieves these resource burdens. The flexibility to get performance metrics for varying inputs and configurations helps researchers answer what-if questions, identify bottlenecks, and better meet design constraints. Example questions that performance models can help to answer include but are not limited to 1) how does the change of batch size and/or the number of

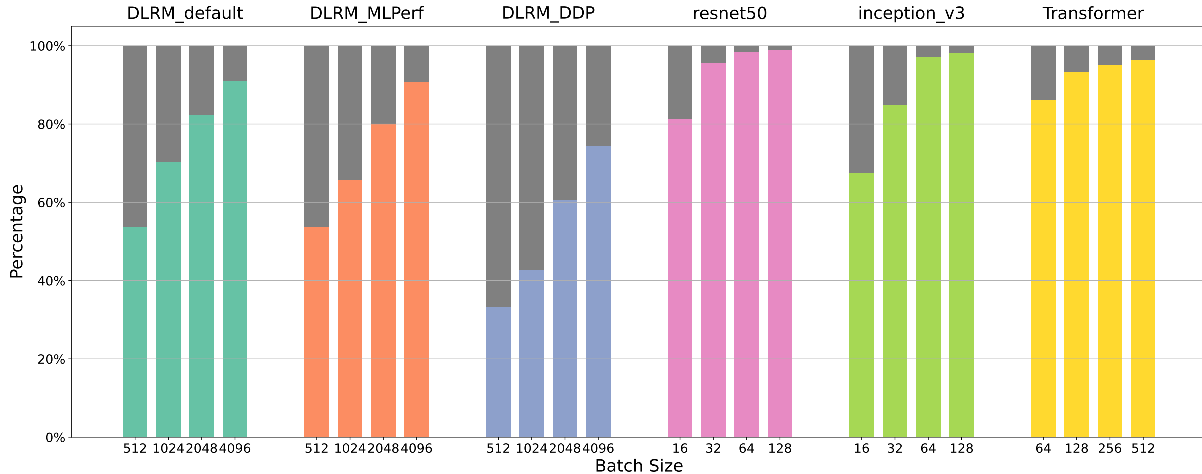


Figure 4.1: GPU utilization of per-batch training time of six DL models on an NVIDIA Tesla V100 GPU. The batch sizes shown here are those commonly used in training. RMs such as *DLRM** have substantially more device idle time than other models. Whereas other models can be adequately modeled by summing kernel time, modeling RMs is a more complex problem.

parameters impact performance and memory constraints; 2) how much performance can be gained with new GPUs; 3) can optimizations such as operator (op) fusion improve performance; 4) how to improve embedding table sharding load balance, etc. However, building such a performance model faces three major challenges:

- Models with lower GPU utilization are difficult to model. We quantify "GPU utilization" as the ratio of *GPU active time* (i.e., when kernels for compute or data transfer are running on the device) over *total training time* per batch.[1] Figure 4.1 shows that the GPU utilization of some computer vision (CV) and natural language processing (NLP) models like ResNet [31] and Transformer [114] are close to 100%, whereas that of RMs (with DLRM [78] as an example) are much lower. While end-to-end (E2E) runtime of workloads with high GPU utilization can be accurately modeled by simply adding their constituent kernel runtimes, the same method fails for workloads with lower GPU utilization like RMs.

- The combination of GPU asynchronous execution with task/data dependencies makes it difficult to estimate the contribution of each operator's device kernel time and host-side

[1]Slightly different from *nvidia-smi*'s definition of GPU utilization (measured over a sample period between 1 and 1/6 second). Notice that "GPU utilization" here is a temporal metric and should be distinguished from hardware utilization.

overheads to the per-batch training time on the device. Previous approaches focused on op-level execution times did not account for these complexities, missing opportunities for a more general and accurate approach.

- Finally, an RM comprises a broader range of operators than convolution-dominated CNNs and matrix-multiply-dominated Transformers. While simple models with one kernel performance model may suffice for these simpler cases, RMs require more kernel performance models to characterize their behavior.

In summary, previous performance models for DL workloads were not accurate enough to model DLRM and by extension other complex workloads because they did not address low-GPU-utilization, asynchronous, or many-complex-operator workloads.

Our research[2] addresses these complexities by proposing a new performance model for the GPU training of DLRM. Once built, this performance model can provide high-confidence metrics to answer questions proposed above and beyond without the need to profile *new* workloads on GPUs. Here, we focus on a single-GPU configuration in the context of the above challenges, leaving multi-GPU for future work. We begin by analyzing the device execution time of DLRM to identify dominating operators and kernels. Then, using heuristic or ML approaches, we build performance models for these kernels for a wide range of input configurations and achieve less than 10% geometric mean average error (GMAE) for each of them in predicting kernel execution time. Beyond accurate kernel models, we also must incorporate host-side overheads into our model. This analysis is a key insight of our work. We categorize host-side overheads into five types and experimentally show that these overheads are consistent across different ops. Using our runtime observer inside PyTorch, we record DLRM's execution graph for its inputs, outputs, and data dependencies. Combining the above components and the ML model execution graph, we construct a critical-path-based E2E performance model for DLRM training on GPUs. This method achieved 4.61% and 7.96% geomean errors for per-batch GPU active time and training latency, respectively, compared to the actual measured time collected by running the DLRM benchmark. We demonstrate that using shared overheads across workloads only incurs a slight 2.19% prediction error increase compared to using individual workloads' overheads. This means

[2]Code is open-sourced at `https://github.com/owensgroup/ml_perf_model`.

a user can maintain a shared database for large-scale predictions for numerous workloads. We compared our performance model with several existing performance models on representative CV and NLP models beyond DLRM. The results show that our method is general and works well across a variety of workloads on different generations of GPUs. We also discuss potential use cases of our performance model at the end of the paper, where we demonstrate the model's ability to provide insights into the RM workload characterization and assist practical model-system co-design with the support of the execution graph. Our contributions to this research include:

- For predicting the GPU training time of DL models, we show our critical-path-based E2E performance model is a more generalized solution than previous methods that only focus on the device active time, especially those with low GPU utilization such as DLRM.
- We separately predict kernel time and GPU idle time and show that compared to op-based methods, this separation facilitates performance modeling by *sharing kernel performance models across ops* that call the same type of kernels and thus reducing the cost of collecting metrics from microbenchmarks. The principles and techniques we used to model kernels can model other kernels that are not included in DLRM as well.
- With our specialized model execution graph observer that captures data dependencies among ops, we provide more flexible simulation and performance modeling options that together assist model-system co-design than previous methods do. Without actually running the computation on GPUs, users can model performance impacts optimization of DL models, such as changing batch size, hardware, operator fusion, reordering, and parallelization, by simply transforming and changing the model execution graph.

4.2 Related Work

4.2.1 Recommendation Models and DLRM

RMs have evolved from simple regression-based predictive models [116], collaborative filtering [34], and neighborhood methods [80] to deep-learning-based RMs [14, 29, 58, 78, 118]. Some deep learning models, such as DIEN [134], also consider sequences of users' actions. The key characteristics that differentiate RMs from CNNs and NLPs are a mixture of sparse and

dense computations, large training data volumes, and large, potentially unbounded model sizes.

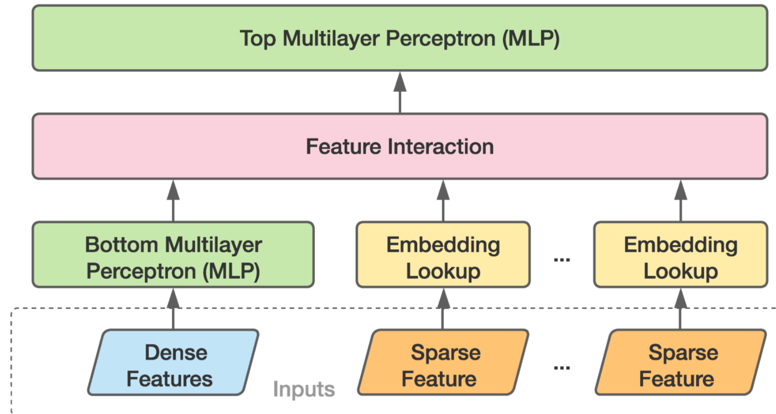


Figure 4.2: The high-level model architecture of DLRM. The inputs (usually user and product data in practice) can be dense and sparse (categorical) features. Each embedding table contains up to millions of embedding vectors and hundreds of values per vector, and because of which they are often sharded across multiple devices in the distributed training.

We choose to use DLRM implemented in PyTorch as a modern representative workload in our analysis. The reasons are: 1) DLRM is a typical example of ML workloads that are highly customizable and at the risk of having low GPU utilization; 2) DLRM forms a common and effective paradigm of using embedding lookup and MLP to process sparse and dense features respectively that generalize to RM design. Figure 4.2 depicts DLRM's high-level model architecture. In contrast to embedding table lookups, which are memory-intensive, the multilayer perceptron (MLP) operations are compute-intensive, while any or both of them can dominate the execution time. Besides, the feature interaction is bounded by communication if the model is trained on a multi-GPU platform, and the inputs might be memory-capacity-bound if the training data size is large. Compared to other kinds of models, including CNNs and NLPs, DLRM is potentially bounded by these multiple factors, and as a result building a performance model for it is technically more challenging.

4.2.2 GPU operator and kernel performance models

Op-level and kernel-level performance models usually fall into two categories. *Heuristic models* (e.g., the roofline model [121]) estimate the kernel execution time by estimating memory traffic,

Table 4.1: Comparison of our work with previous performance models. E2E prediction of Zhu et al. is marked as 'Limited' as it only estimates the optimization efficacy on certain kernels instead of making predictions for every single kernel.

Work	Kernel Pred.	Idle Time Pred.	E2E Pred.	Target Model Types
Justus et al. [46]	✓	✗	✓	CNNs
Pei et al. [87]	✓	✗	✓	CNNs
Liao et al. [59]	✓	✗	✓	CNNs
Zhu et al. [135]	✗	✗	Limited	Multiple
Yu et al. [127]	✓	✗	✓	Multiple
Rajagopal et al. [93]	✗	✗	✓	CNNs
Ours	✓	✓	✓	Multiple+RMs

floating point operations, etc. *ML-based models* are trained with benchmark data of kernel execution to predict kernel time for any input size.

4.2.3 Models for GEMM-based kernels

With the current PyTorch release, MLP layers (intrinsically matrix multiplication) rely on cuBLAS and its GEMM-based kernels as the low-level implementation on NVIDIA GPUs. Either using the roofline model or designing a heuristic performance model for these kernels turns out to be infeasible because of not only the lack of source code but also the special tile quantization and wave quantization effects of cuBLAS [81]. In existing research (e.g., Lym et al. [65]) on heuristic performance model design for proprietary libraries like cuDNN, many parameters are still opaque or extremely difficult to measure. Therefore, rather than heuristic ones, an ML-based performance model is more suitable in this case. Previous work [59, 127] shows that either a CNN or MLP model is sufficient to capture the performance features of the GEMM operation. In our work, we use MLP to construct the performance model for cuBLAS kernels called by PyTorch ops like *addmm*, *bmm*, *linear*, etc., which are all GEMM-based.

4.2.4 Model-level performance modeling

Previous work [46, 57, 59, 87, 93, 127] mainly focuses on CNNs and/or NLP models, which are primarily dominated by compute-bound convolution or GEMM ops and have high GPU utilization. In contrast, our work targets a more complex model (DLRM) that can be highly customized with multiple dominating factors, and handles DLRM's substantial device idle time in our E2E training time prediction. *Daydream* [135] predicts model runtime after certain optimizations by simulating execution based on the kernel-task dependency graph. This work

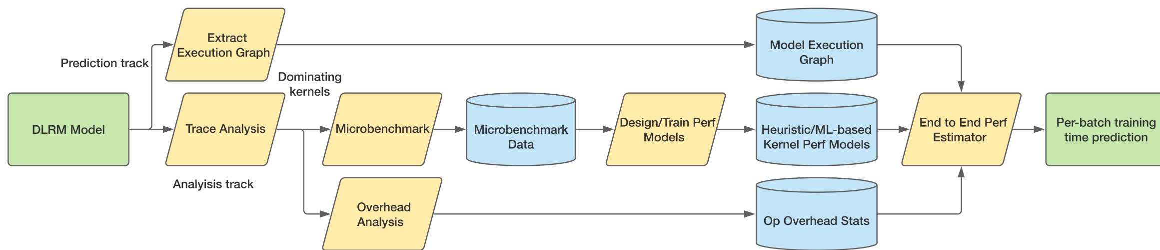


Figure 4.3: An overview of our prediction pipeline. We begin with DLRM models taken as inputs. These are sent through the *Analysis Track* for trace analysis, microbenchmark data collection, kernel performance model design/training, and op overhead analysis. Armed with these analyses, subsequent DLRM models simply go through the *Prediction Track*, where their execution graphs are extracted and their performance is predicted. This prediction pipeline is designed to be modular so that building blocks of the pipeline marked with blue cylinders can be reused and enriched for modeling tasks for workloads beyond DLRM.

has a similar approach to ours in addressing the timing of both CPU and GPU threads; however, it lacks the ability to directly predict individual kernel runtime. This limits its capability in predictions for varying input and configuration changes without recollecting performance data using hardware. Separately, *Habitat* [127] presented a performance predictor using MLP models trained with kernel metrics. It showed that combining *Habitat* and *Daydream* resulted in a higher average error of 16.1% than *Daydream* alone. We reduce prediction error compared to this previous work by actually predicting the kernel runtime and overheads based on a finer granularity of instrumentation. In addition, *Daydream*'s kernel dependency graph does not capture data dependencies and thus is limited in discovering and predicting the efficacy of other optimizations such as concurrent kernel execution. In our work, data dependencies are well-captured by the execution graph and therefore we can accurately model a wider variety of optimizations, such as performance-model co-design. Table 4.1 summarizes different features implemented in previous work and ours. To the best of our knowledge, our work is the first that can successfully target the performance modeling complexities characteristic of complex models like DLRM.

4.3 Methodology

Typically, the per-batch training time is estimated by summing the execution time of each *op* in a certain way. Op execution time can be either measured at the host or the device as the

sum of kernel execution time. Since GPU kernels are scheduled asynchronously, it is hard to accurately predict an op's *host time* from the computation it conducts, and thus the op's execution on the device is usually the time to be measured. For example, CNNs usually resemble the right-hand-side case in Figure 4.4: ops are mostly convolution and GPU compute-bound, and therefore they usually have high GPU utilization. Previous studies that have primarily targeted CNNs can safely make the prediction by summing the individual kernel time and the effects of omitting CPU overhead are minimal. However, this method is sometimes not sufficient to accurately model the E2E execution time, if the model's GPU utilization is low. As noted in Section 4.1, DLRM, with its varying sizes and composition of ops, could possibly resemble either the left or right cases in Figure 4.4 and have as low as 40% GPU utilization. This means the per-batch training time prediction error will be 60% by following the same method, even if the kernel prediction accuracy is 100%. In practice, execution inefficiencies and inherent model design could both be the cause of low GPU utilization. These complexities necessitate a better methodology of building the performance model for DLRM as well as other models with low GPU utilization.

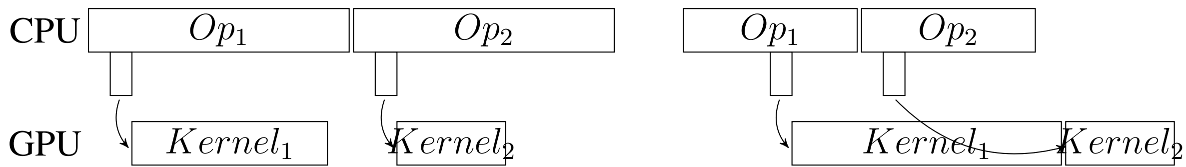


Figure 4.4: Two cases for dependent ops. The small rectangles below the (CPU) ops indicate the launch of their GPU kernels. The left trace is CPU-bound and the right one is GPU-bound. In either case, summing the device active time of the two ops does not properly represent the total execution time, in part because host-side overhead are not considered.

To address this challenge, we devise a performance modeling pipeline that separates the prediction of device active time and idle time and integrates both parts with a critical-path-based algorithm that tracks the execution time on both the CPU and GPU. Such a separation brings two major advantages in building kernel performance models:

- Ops (e.g., *addmm*/*bmm* vs. {*Addmm*/*Bmm*}*Backward*) that have the same type of kernel calls (i.e., cuBLAS GEMM kernels) can *share the same performance model*. This saves us a large amount of time for microbenchmarks and training of ML-based kernel performance models.

- Although ML-based performance models can predict kernel time and op overhead as a whole for each op, heuristic models solely based on an op's mathematical expression are not able to address its overhead. Separating them allows us to flexibly choose between these two approaches, while the overhead is handled separately.

Figure 4.3 depicts an overview of the prediction pipeline. Although we focus on modeling DLRM's performance in this section, it should be noted that this performance model can be handily extended to model ML workloads beyond DLRM by adding new kernel performance models and operator overhead information to the pipeline as assets. Typically, our performance model runs fast and usually finishes a single E2E prediction in a few seconds. The remainder of this section explains how each building block of the pipeline works in detail.

4.3.1 Per-batch Training Time Breakdown

To understand the device active time and identify dominating ops and kernels, we perform a breakdown of per-batch training time by analyzing PyTorch profiler trace files, in which the metadata of all events, i.e., calls to operators, is flattened. We construct an event tree to represent the calling stack of each op so that the device execution time of each kernel is attributed to the corresponding op, and thus we know the dominating kernels by knowing the dominating ops. The device time breakdown of three DLRM models (configurations shown later) is presented in Figure 4.5. We observe that:

- Just as we noted in Section 4.1, the device-side idle time forms a non-negligible proportion of the total device time because the host-side op overhead and data dependencies implicitly contribute to it by blocking the scheduling of GPU kernels. This demonstrates the necessity of analyzing kernel execution time and overhead separately.

- There is no single op that dominates the device active time of the model. Ops that jointly dominate include compute-bound ops *addmm* and *bmm*, the memory-bound op *embedding lookup*, ops *concat* and *to* (memory copy), and their counterparts in the backward pass.

- Trivial/element-wise ops such as *relu* and *MseLoss* sum to around 5% of the E2E time. This means they should not be omitted in order to achieve high prediction accuracy.

Furthermore, we perform an in-depth analysis of the kernel composition of the dominating ops. The analysis reveals that most of them are composed of or dominated by one single kernel. Exceptions include *AddmmBackward* and *BmmBackward0* that are dominated by two GEMM kernels, and *Optimizer*'s forward and backward ops that are both dominated by a series of element-wise kernels. Ops in the last category are handled by predicting their sum of kernel time as a whole, possibly ignoring minor kernels that do not appreciably impact the run time. We conclude that there are six major kernels that dominate the per-batch device active time for DLRM training: sparse embedding lookup kernels (both forward/backward) for embedding table lookup, GEMM kernels for the bottom and top MLP, and four memory kernels including concatenation, data copy, tensor permutation, and *IndexBackward* (low triangular matrix extraction and flatten in feature interaction).

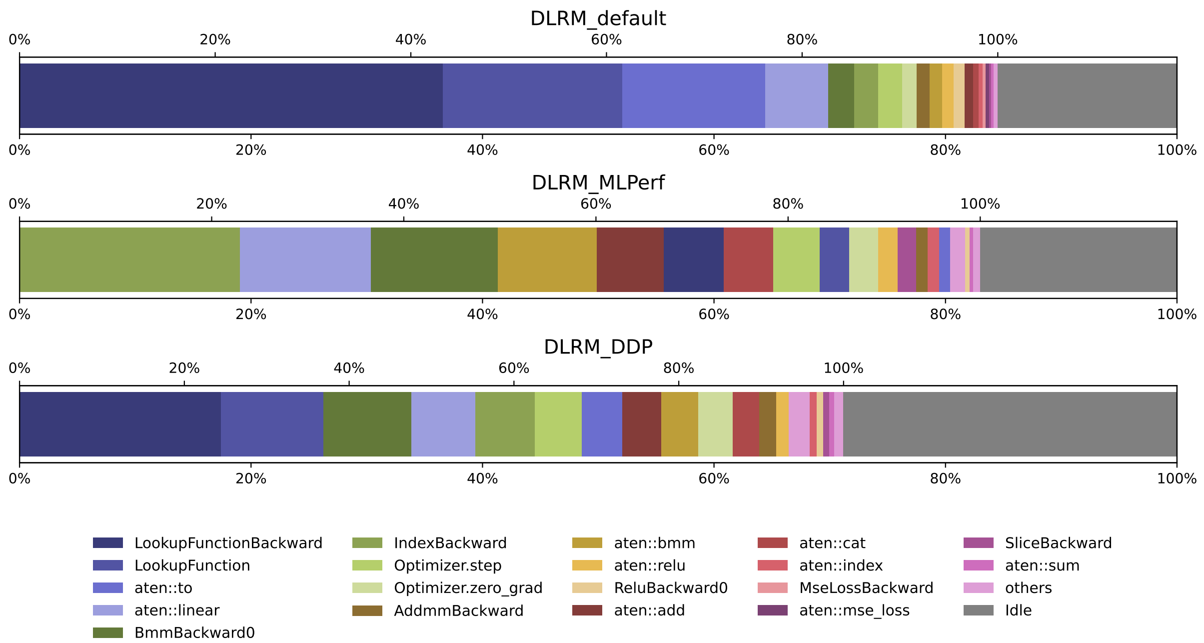


Figure 4.5: Device time breakdown of three DLRM models with a batch-size of 2048 on a V100 GPU, with profiler overhead *excluded*. Notice that with different configurations, DLRM is dominated by different kernels, e.g., embedding lookup forward and backward dominates the first and third cases, whereas in the second case, it appears to be less important, giving the domination in to *IndexBackward* and FC.

4.3.2 Microbenchmark and Performance Models for Dominating Kernels in DLRM

We create microbenchmarks for seven kernels in total based on the results we get from the breakdown: the six mentioned above plus the trivial *IndexForward* that partners with *IndexBackward*. We run the microbenchmarks that sweep through a wide range of (up to 30k) tensor shapes and arguments for each target kernel and take days to run. Specifically, since all GEMM-related ops are dominated by one or two GEMM kernel calls, we skip benchmarking all these ops and share the GEMM kernel benchmark data for their performance modeling. We also discover that the only type of tensor permutation that occurs in DLRM is the batched matrix transpose, i.e., permutation of the second and third axes of a 3D tensor, and thus it becomes the only type of permutation we benchmark. We first execute the corresponding PyTorch operators on one single GPU for 5 iterations as a warm-up, then use NVIDIA's *nvprof* profiler to extract the name of the dominating kernels, and then solely benchmark these kernels for 30 iterations to extract their execution time. Default GPU application clocks are applied, and the CPUs' turbo boost is turned off to guarantee both the accuracy and stability of the benchmark.

With this data, we are able to develop kernel performance models for each of the dominating kernels, as it is impossible to apply one single such model to accurately predict the kernel execution time for all dominating kernels we identify. These performance models are designed in two different ways:

1. For kernels without *source code access*, such as cuBLAS, PyTorch JIT generated kernels, etc., we predict their execution time with ML-based performance models trained and verified with microbenchmark data.

2. For kernels that are either *accessible or trivial*, i.e., element-wise, we predict their execution time by either using the roofline model or designing heuristic performance models with memory and throughput estimation through code analysis. As such, the microbenchmark data is solely used to verify the prediction accuracy.

The following subsections elaborate on how these kernel performance models are developed. Our performance models are highly extensible, as the principles and techniques we introduce

(code analysis, ML-based kernel performance model training, etc) also apply to any new ops not covered by this work.

4.3.2.1 Heuristic Performance Models

4.3.2.1.1 Characterizing the Embedding Lookup Kernels

The embedding lookup layers are intrinsically SpMM operations that map categorical features to dense representations. Therefore, the procedure that we describe here for modeling embedding lookup kernels also applies to other kernels of a similar type with irregular memory access patterns and/or is possibly bound by GPU global memory bandwidth. Given a matrix of the vector of weights $A \in \mathbb{R}^{m \times t}$ that contains t multi-hot vectors of length m and an embedding table (weight matrix) $W \in \mathbb{R}^{E \times d}$, the embedding lookup operation can be written as $S = A^T W$. Since a real industrial-scale DLRM model usually contains multiple embedding tables, we can simply concatenate these embedding tables, and pack and batch the input indices into new input tensors, such that the embedding lookup operation over multiple embedding tables can be done in one pass. We integrate the implementation of this batched embedding table lookup algorithm (with SGD for the backward case) from Tulloch [112] into DLRM. The following analysis is based on the code of this implementation. Important parameters of the implementation include B as the batch size, E as the number of embeddings per table, T as the number of tables, L as the number of lookup operations to produce one dense vector and D as the embedding vector length. Note that we extend the definition of "warp" for simplicity and refer to a group of threads that all have the same *blockIdx*$.x/y/z$ and *threadIdx*$.y/z$ as a WARP. In practice, this typically refers to groups of threads of sizes 32, 64, or 128.

We spot that the bounding factor of this op is the memory traffic caused by looking up embedding vectors from the weight tensor. In practice, the value of E can range from a few hundred to thousands of millions, while L is much smaller, i.e., up to one hundred. We can expect that embedding vectors are more frequently fetched from DRAM than from the L2 cache. Therefore, we approximate the execution time of the forward kernel by its DRAM access time,

which is given by

$$\begin{aligned}
\mathit{tr_table_offsets}_w &= 32 \text{ bytes} \\
\mathit{tr_offsets}_w &= 64 \text{ bytes} \\
\mathit{tr_indices}_w &= \lceil 4 \times L/32 \rceil \times 32 \text{ bytes} \\
\mathit{tr_weights}_w = \mathit{tr_outputs}_w &= \lceil 4 \times D/32 \rceil \times 32 \text{ bytes} \\
t &= \frac{\mathit{DRAM_traffic}}{\mathit{peak_DRAM_BW}} \\
&= \frac{B \times T \times (\mathit{sum\ of\ all\ above})}{\mathit{peak_DRAM_BW}}.
\end{aligned}$$

The subscript w denotes that these are per-WARP DRAM traffic; $B \times T$ is the total number of WARPs. For the backward kernel, we simply replace the per-WARP weights traffic by

$$\mathit{tr_weights}_w = \lceil 2 \times 4 \times L \times D/32 \rceil \times 32 \text{ bytes},$$

and follow exactly the same other equations.

This method can be further enhanced by estimating the L2 cache hit rate of accessing the embedding lookup table and separating the total memory traffic into DRAM traffic and L2 traffic. As one thread WARP is responsible for computing one vector in the output tensor, assuming only one CTA resides on each streaming-multiprocessor (SM) on the GPU at a time, the number of embedding lookup tables whose (at least part of) data simultaneously reside in L2 cache is given by

$$\mathit{num_tables} = \mathit{rows_per_block} \times (\#\mathit{SM})/B,$$

where *rows_per_block* is a kernel argument specifying how many output vectors are computed per CTA. With the L2 cache size of the GPU known to us, we can calculate the number of rows per table that resides in the L2 cache as

$$\mathit{avg_cached_rows_per_table} = \min\left(\frac{\mathit{L2_cache_size}}{(\mathit{num_tables}) \times D}, E\right),$$

where the second term covers the case when an embedding lookup table with E rows is small enough to reside in the L2 cache. Therefore, the hit rate of the L2 cache, i.e., the probability that the accesses to a total of L embedding lookup table row vectors among all E vectors can be

estimated by

$$p = \frac{\binom{avg_cached_rows_per_table}{L}}{\binom{E}{L}}.$$

Notice that the *table_offsets* and *offsets* tensors are relatively very small and frequently accessed, and thus we assume they always stay in L2. Therefore, we construct the enhanced performance model as:

$$\begin{aligned}
tr_{L2} &= tr_table_offsets_w + tr_offsets_w + p \times tr_weights_w \\
tr_{DRAM} &= tr_indices_w + tr_outputs_w + (1-p) \times tr_weights_w \\
t &= \frac{DRAM_traffic}{peak_DRAM_BW} + \frac{L2_traffic}{peak_L2_BW} \\
&= B \times T \times \left(\frac{tr_{DRAM}}{peak_DRAM_BW} + \frac{tr_{L2}}{peak_L2_BW} \right).
\end{aligned}$$

4.3.2.1.2 Characterizing Element-wise Kernels

For memory kernels of ops including *concat*, *memcpy*, etc. that involve intra-GPU or CPU-GPU data transfer, as well as element-wise kernels of ops like *ReLU*, *sigmoid*, etc., it is straightforward to estimate their execution time by applying the roofline model [121]:

$$\begin{aligned}
t &= \max(t_compute, t_memory) \\
&= \max\left(\frac{FLOP}{peak_throughput}, \frac{bytes_{read} + bytes_{write}}{peak_BW} \right).
\end{aligned}$$

We use the maximum measured bandwidth of the benchmark as the corrected peak bandwidth in the calculation.

4.3.2.2 ML-based Performance Models

Dominating kernels of DLRM that require ML-based performance models include GEMM, transpose, and the forward and backward kernels of tril, for their source code being either non-accessible or too complex to model heuristically. Specifically, we find that it is non-trivial to model the performance of transpose ops like *T* or *permute*, because technically the underlying implementations of tensor transpose might differ significantly [30, 115], yet these implementations are opaque to users in PyTorch since the kernel is JIT-generated. Therefore, we adopt the ML performance modeling approach for transpose kernels.

For each kernel in this category, we train an MLP model that takes the kernel's input dimensions as the input features and predicts the kernel execution time as the output. We conduct a grid search over a universal search space defined in Table 4.2 for the best configuration by training a series of MLP models over the microbenchmark data and keeping the one with the lowest prediction error. The loss function for training is Mean Square Error (MSE). As the input sizes of the benchmark are chosen in an almost exponential scale, e.g., 32, 64, 128, etc., we preprocess the dataset by taking logarithm values of both the sizes and the results. We also scale the learning rate by 10 if *SGD* is chosen as the optimizer. Typically, obtaining such an MLP model for one kernel through grid search takes a few hours of training on one single GPU.

Table 4.2: MLP performance model search space.

Hyperparameter	Range
num_layers	[3,4,5,6,7]
num_neurons_per_layer	[128,256,512,1024]
optimizer	[Adam, SGD]
learning_rate	[1e-4, 2e-4, 5e-4, 1e-3, 2e-3, 5e-3, 1e-2]

4.3.3 Device Idle Time Analysis

Device idle time, as we show in Figure 4.5, is an important part of the total device execution time. We predict device idle time based on overhead obtained by analyzing the trace files generated by profilers. In a single-GPU context, the main source of device idle time is the *host overhead that is not hidden*. There are two assumptions we make for this overhead:

- Model-independence: Same types of overhead of the same op have the same stats on the same machine.
- Size-independence: overhead does not depend on input/output tensor sizes of ops.

That means overhead is supposed to only depend on the training platform (i.e., CPUs) configurations. Based on these two assumptions, we analyze the host-side overhead and categorize them into five types as shown in Figure 4.6, including:

- Type 1: Overhead between two top-level PyTorch op calls.
- Type 2: Overhead before an op's first kernel launch begins.

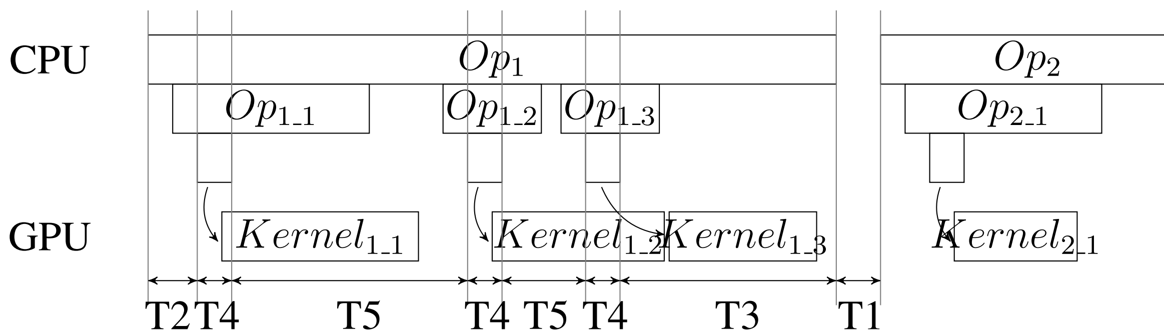


Figure 4.6: Host-side overhead types. The labels T1–T5 indicate the five overhead types introduced in Section 4.3.3. Each op has one T2 and one T3 overhead, and at least one T4 overhead if it has device kernel calls.

- Type 3: Overhead after an op's last kernel launch ends.
- Type 4: Execution time of CUDA runtime functions, e.g., *cudaLaunchKernel*, *cudaMemcpyAsync*, etc.
- Type 5: overhead between two kernel launches.

Some of the overhead, namely T2, T3, and T5, should be independent of the input parameters of the op, as we assume all the parameter-defined operations, mainly the computation and data movements, are offloaded to the device. By analyzing 100-iteration trace files of the models we choose, we characterize each type of overhead and store their mean values in a JSON file to be used in the E2E performance model. To guarantee accuracy, the profiler overheads of CPU and GPU events are excluded by subtracting them from the execution time of each event. In practice, we use 4 μs as indicated in the PyTorch source code to model the profiler overhead of GPU events, while that of CPU events varies from platform to platform, and we find that an empirical value of 2 μs is a good choice.

4.3.4 E2E GPU Training Performance Model

One challenge of building an end-to-end performance model of an ML training workload is to have sufficient information about its run-time execution. Early implementations of ML frameworks such as Caffe [43] define an ML model as a static graph in the protobuf format. In recent years, ML frameworks such as TensorFlow [1] and PyTorch [86] have closely integrated

programming language bindings to support dynamic ML model graphs characterized by conditionals and loops. Furthermore, they support eager mode execution. With the flexibility of these frameworks, the ML model definition is essentially a program and requires execution to fully capture the run-time characteristics. We implemented an execution graph observer inside PyTorch that allows us to extract both the operators executed and their inputs and outputs data dependencies during the model training process. Once the ML model's run-time execution is captured, the execution graph can be reconfigured to use different data inputs or hardware devices. For example, we may collect the execution graph while running on the CPU and apply our performance models to the execution graph to predict the workload's performance on the GPU or other types of hardware.

Algorithm 3 E2E GPU Training Performance Model.

1: **Input:** Execution graph G of a DLRM model; Kernel performance models $\{M\}$; overhead Ov.
2: **Output:** Predicted per-batch training time T.
3: Initialize $cpu_time = 0, gpu_time = 0$
4: **for** each op in G **do**
5:     Look up $T1, T2, T3, T4, T5$ from Ov for op
6:     $cpu_time += T1$
7:     **if** op has kernel calls **then**
8:         $cpu_time += T2$
9:         **for** each kernel k op calls **do**
10:             Predict kernel time T_k with the corresponding performance model picked from $\{M\}$
11:             $gpu_time = \max(gpu_time + 1, cpu_time + T4/2) + T_k$
12:             $cpu_time += T4$
13:             **if** k is not the last kernel **then**
14:                 $cpu_time += T5$
15:             **end if**
16:         **end for**
17:         $cpu_time += T3$
18:     **else**
19:         $cpu_time += T5$
20:     **end if**
21: **end for**
22: $T = \max(gpu_time, cpu_time)$

We devise a critical-path-based Algorithm 3 that integrates the predicted kernel time and

overhead to predict the E2E training time of DLRM. We identify the critical path of execution by keeping track of both the execution time on CPU and GPU. For each operator, we first add T1 and T2 to the CPU time as a prerequisite. If the op has kernel calls, we set the start time of each kernel based on whether the CPU or GPU time is the critical path (line 11), so that the device idle time caused by the host overhead is counted. Each kernel time is then added to the GPU time, while T4 and T5 are added to the CPU time. T3 is added after all kernels are processed. Eventually, we take the maximum of CPU and GPU time as the critical path and thus the final E2E predicted time.

4.4 Results and Analysis

Table 4.3: DLRM model configurations.

	DLRM_default	DLRM_MLPerf	DLRM_DDP
Bot MLP	512-512-64	13-512-256-128	128-128-128-128
EL Tables	8	26	8
Rows	1000000	Up to 14M	80000
EL Dim	64	128	128
Top MLP	1024-1024-1024-1	1024-1024-512-256-1	512-512-512-256-1

We evaluate our benchmark and performance models on three different NVIDIA GPUs—Tesla V100, Tesla P100, and GeForce GTX TITAN Xp—with CUDA 11.3 and Python 3.9. We conduct the E2E tests on three open-sourced DLRM models that can be accessed in Meta's DLRM repo on Github [22]. We name them DLRM_default, DLRM_MLPerf, and DLRM_DDP, and show their configurations in Table 4.3. To launch the training of the DLRM_MLPerf model, we use the Kaggle Criteo dataset as the training dataset, and change the embedding table sparse feature size of *DLRM_MLPerf* from 128 to 32 to allow it to fit into the memory of our TITAN Xp and P100. We also use the code repository of Konstantinidis et al. [48] to benchmark the GPU hardware parameters, e.g., FLOPS, DRAM bandwidth, etc., that are needed by the heuristic performance models.

4.4.1 Performance Models for Dominating Kernels in DLRM

In Table 4.5 we can see that on all types of GPU, our plain performance model for batched embedding table lookup achieves a varying yet low error rate for all table sizes and a stable

Table 4.4: Statistics of active (kernel) time and E2E time prediction errors across three platforms. Abbreviations: **SE2E**: shared E2E, **g.m.**: geomean.

	Overall			V100			TITAN Xp			P100		
	g.m.	min	max	g.m.	min	max	g.m.	min	max	g.m.	min	max
Active	**4.61%**	0.41%	15.25%	2.69%	0.41%	7.82%	5.73%	1.18%	11.04%	6.37%	1.99%	15.25%
E2E	**7.96%**	0.09%	24.92%	7.56%	0.73%	21.96%	6.97%	0.09%	24.92%	9.59%	2.04%	22.76%
SE2E	**10.15%**	0.75%	28.38%	6.92%	0.75%	20.79%	12.52%	1.13%	26.17%	12.09%	1.06%	28.38%

Table 4.5: Prediction error of dominating kernel latency. Abbreviation examples: **EL** (embedding lookup), **GEMM** (fully connected and interaction layers), **memcpy** (memory copy from host to device), **concat** (concatenation), **tril** (lower triangular extraction and flatten), **F** (forward), **B** (backward), **H** (with hit rate estimation for EL), **L** (large size, average embedding table size greater than 100000).

Approach	GPU Kernel	V100 GMAE	V100 mean	V100 std	TITAN Xp GMAE	TITAN Xp mean	TITAN Xp std	P100 GMAE	P100 mean	P100 std
Heuristic	EL-F	11.46%	35.92%	56.81%	12.81%	34.05%	38.92%	8.63%	33.19%	54.72%
	EL-FL	6.93%	11.22%	8.96%	7.54%	16.76%	16.01%	2.89%	5.52%	6.26%
	EL-FH	9.27%	16.73%	16.39%	11.88%	25.44%	26.04%	6.42%	13.06%	14.81%
	EL-FHL	7.85%	12.68%	10.02%	8.84%	18.20%	16.68%	3.84%	7.02%	7.08%
	EL-B	9.53%	34.39%	60.91%	8.31%	38.62%	65.77%	12.49%	35.26%	62.70%
	EL-BL	5.27%	5.94%	2.29%	2.38%	2.95%	1.61%	9.88%	10.13%	2.37%
	EL-BH	7.39%	13.37%	15.01%	5.57%	15.16%	23.99%	8.42%	12.59%	12.12%
	EL-BHL	5.69%	6.24%	2.28%	2.55%	3.21%	1.68%	10.19%	10.42%	2.33%
	concat	5.34%	11.45%	14.76%	8.17%	11.48%	9.08%	3.30%	6.54%	12.63%
	memcpy	0.57%	0.96%	2.46%	7.05%	13.87%	17.45%	5.10%	7.95%	8.28%
ML-based	GEMM	5.80%	10.00%	10.33%	8.92%	14.24%	11.83%	7.59%	12.30%	10.39%
	transpose	2.95%	5.47%	6.71%	5.75%	10.13%	9.67%	3.35%	5.92%	6.84%
	tril-F	2.13%	3.67%	3.81%	3.23%	6.54%	8.17%	3.71%	6.74%	8.31%
	tril-B	3.67%	7.35%	9.40%	3.08%	6.69%	9.30%	2.71%	4.76%	4.51%

and lower error rate for big table sizes ($E > 100$k). This is because when the lookup tables are small, the L2 cache can capture substantial locality, and thus our assumption that lookup traffic comes from DRAM is no longer valid. However, with our enhanced performance model, we successfully reduce and stabilize the error rate for all table sizes while still maintaining a lower error rate for big table sizes. Thus we adopt the enhanced model in our E2E analysis. Except for embedding lookup, we also achieve decent (i.e., less than 10%) GMAE errors on both ML-based models and other heuristic models for all other kernels. The errors of our kernel performance models correlate across all three different GPU devices.

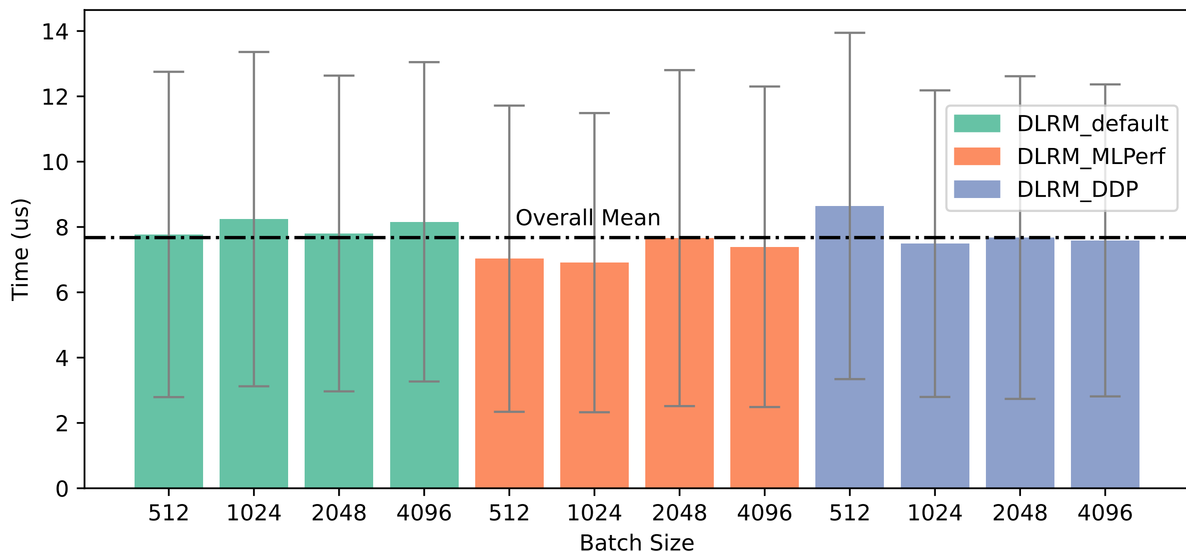


Figure 4.7: T1 overheads means and standard deviations of all models and batch sizes on V100.

4.4.2 Overheads Analysis

We perform analysis on overheads extracted from collected traces of models' E2E execution. We remove per-type outliers outside whiskers ($Q1 - 1.5IQR, Q3 + 1.5IQR$) for each individual workload. The reason we do not conduct an op-level microbenchmark for overheads is that it hardly simulates the overhead behaviors in actual E2E execution. Fig. 4.7 and 4.8 show the statistics of T1 and T2/3/5 overheads respectively. We omit T4 here as we use a value of 10 μs to approximate all the CUDA runtime functions. We see that the means of T1 of different models and batch sizes are close to each other around 8 μs. With different overall mean values, the same conclusion holds for all comparisons shown in Fig. 4.8. From these two figures, no trends of model types or tensor sizes (represented by the batch size while treating all ops per E2E run as an ensemble) being able to affect the overhead statistics are observed. Although this is not a strict mathematical proof of the model/size independence, as we only need a simple estimation for the overheads to fill the gap between device active time and per-iteration time, we argue that it is safe to use the mean values of overheads per type per workload in E2E prediction.

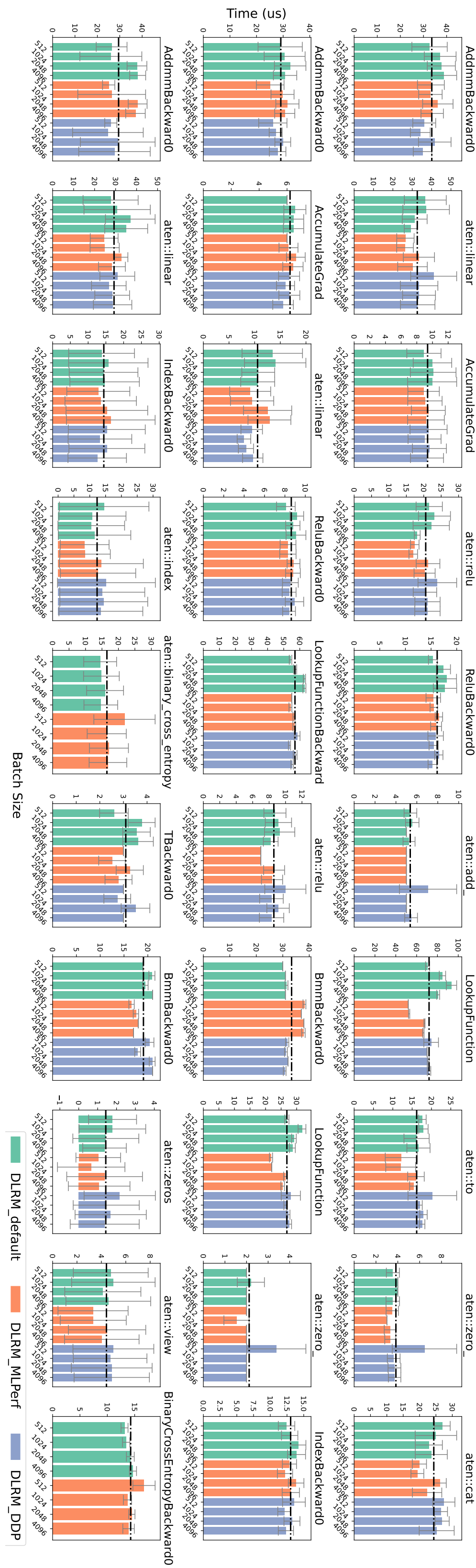


Figure 4.8: Overheads means and standard deviations of 10 most dominating ops per overhead type for each model and batch size on V100. Each row represents T2, T3, and T5, respectively in top-down order. Dash lines in each subplot are the overall means of each overhead type per op.

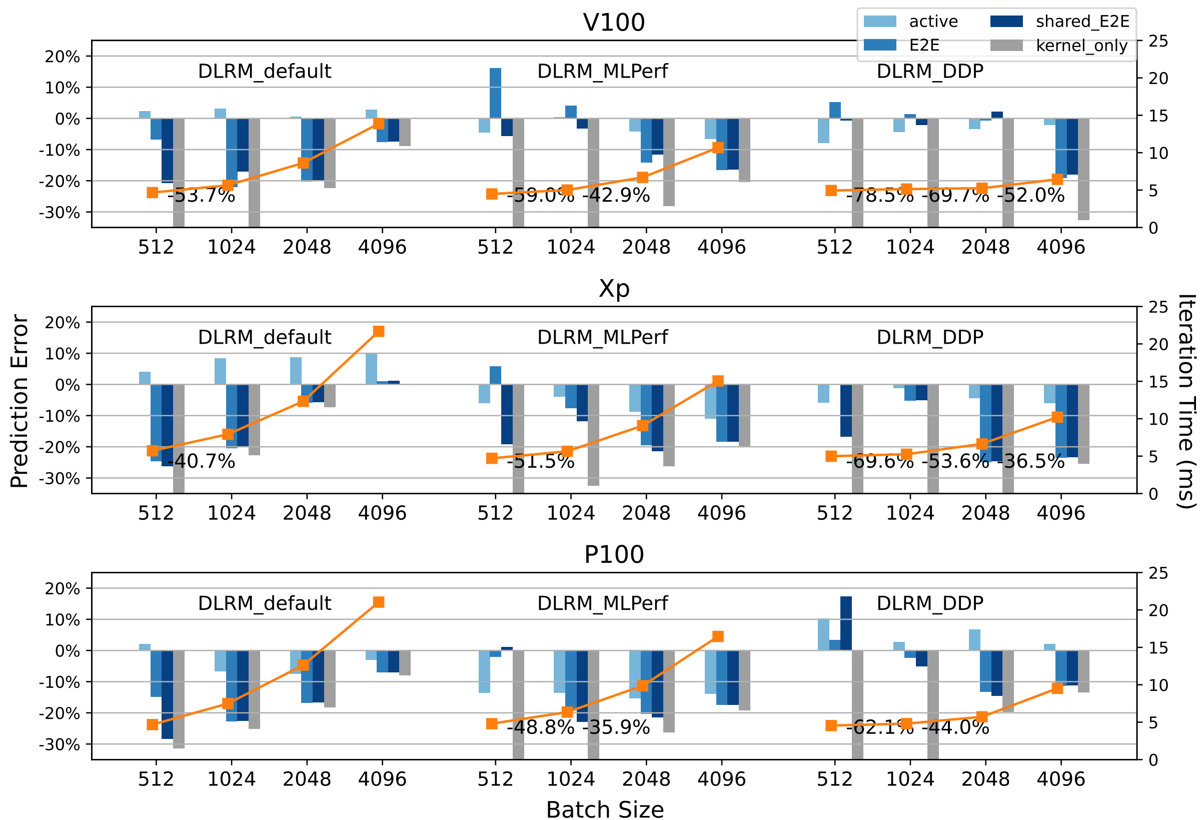


Figure 4.9: E2E per-batch training time prediction of three DLRM models on three GPUs. *active*, *total*, *kernel_only* are respectively the prediction errors of GPU active time, E2E per-batch time, and solely using GPU active time without modeled idle time as the E2E time. The measured iteration time is plotted in orange color for reference.

4.4.3 E2E GPU Training Performance Model for DLRM and More DL Models

We evaluate our E2E prediction on the three DLRM models on three GPUs and show the results in Table 4.4 and Figure 4.9. The baseline we use ("kernel_only" in Figure 4.9) is the E2E training time prediction error by summing up solely the predicted kernel execution time *without* the modeled overheads i.e. GPU active time. We predict the E2E training time with our proposed algorithm. Specifically, "E2E" means to predict E2E time with overheads from individual workloads, while "shared_E2E" means to predict with *shared overheads aggregated across the workloads.*, i.e., averaging the samples across the workloads collected in overhead analysis. We see that the geomean values of active and E2E time prediction error are 4.61% and 7.96% respectively. The E2E prediction error with shared overheads is 10.15%, only 2.19% higher

than that with individual overheads; this indicates the feasibility of maintaining an overhead database for large-scale ML workload predictions in an industrial environment. We notice a trend of gaps between *E2E* and *kernel_only* shrinking as batch size increases. This is because GPU utilization increases with batch size and therefore our performance model degenerates towards "kernel_only". The fact that *kernel_only* prediction errors are much worse than *E2E* when GPU utilization is low justifies the necessity and success of including the modeling of device idle time in our prediction algorithm. Device-wise, the GPU active time error on V100 is the lowest among the three, while the E2E error is the lowest on the platform with TITAN Xp. The prediction error of the device active time comes from the kernel execution time prediction error. For example, the MLPerf model has non-constant table sizes and thus we have to use the average table size in the performance model, which affects its accuracy. Overall, the device active time error rate lies within the range of our expectation, proving the success of the kernel performance model. The E2E time predictions have a clear trend of underestimation, which can be explained by the underestimation of device idle time. We suspect that it is because some of the overheads, e.g., T1, or T4 of *cudaMemcpyAsync*, etc., have long-tail distributions with high variation, while we remove many upper outliers and use their mean values in the predictions. Since these are usually common overheads (T1 is the most common as it occurs for every single op), the error might accumulate quickly and thus result in an underestimation of device idle time and E2E time. In addition, we observe no systematic or correlated errors in either active or idle time and are confident that the E2E device active time and total time are appropriately predicted.

As Figure 4.10 shows, we also compare our performance model on two CV models (ResNet50 and Inception-V3) with two previous works, *Habitat* [127] and *MLPredict* [46], neither of which supports DLRM mainly because of their limited coverage of ops. We do not compare with *Daydream* as it is not open-source and does not make E2E predictions. To enable the prediction of these two models, we extend our microbenchmark to cover the convolution and batch-normalization ops. We can see that our work achieves comparable or better prediction errors against the two previous works. The reason *MLPredict* fails to produce accurate results on some tests might be that the pretrained predictor does not cover certain batch sizes (possibly due to GPU memory limits) and/or convolution input sizes (such as Inception-V3's 1×7 and 7×1

convolution filters).

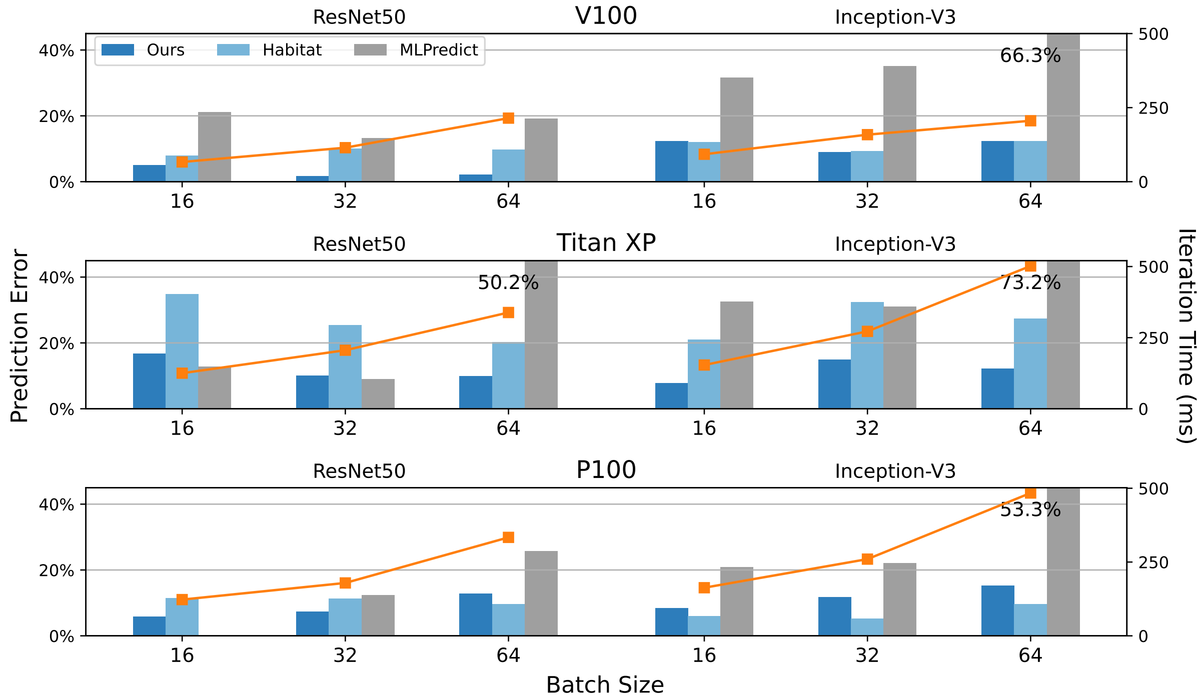


Figure 4.10: E2E per-batch training time prediction of ResNet50 and Inception-V3 as representatives of non-DLRM DL models on three different GPUs. We used Habitat open-source project to collect the prediction result on TITAN Xp since it was not reported in the paper. The actual iteration time is also plotted in orange color for reference.

4.5 Discussions

The advantages of our performance model against previous works include: (1) accurate prediction of individual kernel performance and op overhead and (2) op data dependencies capturing with our execution graph. Therefore, we are able answer the questions we ask in Section 4.1 with more comprehensive and flexible performance modeling and simulation options than both previous works and trace file inspection. Typical use cases of our work include iterative model tuning and op optimization such as fusion. Beyond the models and devices used in this paper, our system as shown in Figure 4.3 is highly extendible for performance modeling of other types of ML workloads on heterogeneous platforms with types of devices from other vendors such as Intel and AMD.

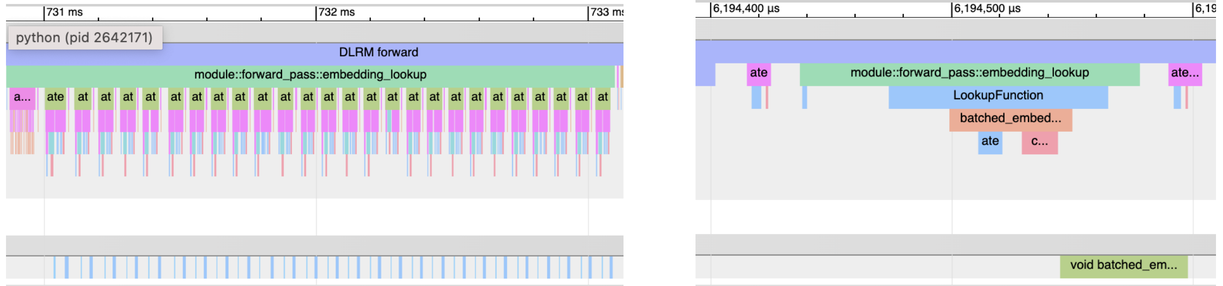


Figure 4.11: Separate embedding bag ops (left) and batched embedding op (right).

4.5.1 Performance Modeling for Model-System Co-design

4.5.1.1 Iterative Model Tuning

To ensure both high precision/recall and fast training speed, the iterative tuning of configurations of ML models (e.g., number and size of layers) is necessary yet difficult, especially when frequent training job launches in an industrial environment are costly and not always practical. With our performance model, users can handily make transformations like *insert*, *remove*, *replace*, *resize*, and *parallelize* on our easily mutable execution graph and predict the outcome of their optimization without actually running the code. Specifically, it is straightforward to change metadata of tensor shapes of selected ops and their parent and child nodes in the graph for *resize*, and to assign ops in parallel branches with no data dependency to different GPU streams for *parallel*. This can only be performed with our support of data dependencies between ops and individual kernel runtime prediction. In fact, our performance model could be integrated as a module into network architecture search (NAS) and significantly improve automatic search for the best ML model configuration. We see this as exciting future work.

4.5.1.2 Op Fusion

Op fusion is a common optimization technique that brings speedup by replacing multiple ops with a mathematically equivalent one to reduce both the compute time and overheads. When users implement a new op, it is good to know how it improves the performance in an ML model generally (i.e., with arbitrary input tensor shapes). Figure 4.11 shows an example of optimization that we have done with the performance model. On the left side, it shows a series of embedding bag ops as a good target (i.e., causing too much device overhead) to be fused into a batched embedding op, as shown on the right side. Our prediction pipeline captures optimization

opportunities like this during trace analysis. We then can easily modify the execution graph and replace the subgraph of all embedding bag ops with arbitrary input shapes with one single batched embedding op, whose performance is then predicted by our kernel performance model. This is extremely efficient when there are a large number of ML models to be optimized and evaluated since we never need to launch jobs and benchmark them.

4.5.1.3 Load Balancing

In the cases of multi-GPU training, subgraphs that are too expensive to be computed on one single device are distributed to several through data- or model-parallelism. This is also a common practice for DLRM, especially the enormous embedding tables. Our performance model enables the evaluation of each device's performance upon any schemes of splitting embedding tables that results in different combinations of embedding table sizes on these devices. Again, this greatly accelerates the development and debugging of DLRM training on multi-GPU platforms.

4.5.2 Extendibility

Our performance model is designed to be highly extendible for both workloads and devices. To extend, users only need to design/train new kernel performance models and collect op overhead information for the new devices, which is a straightforward and relatively simple effort. To run on different devices, one should also make sure PyTorch's Kineto tracing is able to capture events of kernels running on these new devices in order to support dominating kernels identification and overhead extraction. Besides, the extension of this work to (distributed) multi-GPU platforms also requires kernel performance models of communication collectives (e.g., *all_to_all*, *all_reduce*). This is one of our works in progress.

4.6 Conclusion and Future Work

We devise a performance model for GPU training of DLRM as well as other ML models. We find that some ML workloads, with DLRM as a typical example, consist of a broad range of ops and have GPU utilization. Therefore, we propose to use different approaches for constructing kernel performance models for these ops; compared to simply predicting the E2E time as the sum of kernel time, our work is a more general methodology that covers the case of model configurations with low GPU utilization. Our final end-to-end performance model is proved to

have low error and high extendibility and is able to assist model-system co-design.

We look ahead to problems in a broader scope given the single-GPU performance modeling problem has been addressed. First, we would like to investigate communication collective performance and consequently model the E2E performance of ML workload training on (distributed) multi-GPU platforms. Another of our goals is to model the performance of embedding lookups with a non-constant number of embeddings and the number of lookups per table, which should improve our overall model accuracy. These problems are addressed in the next chapter which features the performance modeling of multi-GPU training and its use cases.

Chapter 5

An Enhanced Performance Model for Machine Learning Distributed Training on Multi-GPU Platforms

5.1 Introduction

Modern machine learning (ML) workloads tend to grow in both size and computation usually beyond the capability of one single GPU to host and train. Platforms with multiple compute devices such as GPUs, therefore, play a critical role in undertaking training jobs for these workloads. Modern industrial recommendation models, such as DLRM [78], consist of embedding tables of hundreds of Gigabytes and are trained distributedly on 16 nodes $\times$ 8 GPUs with advanced software-hardware co-designed system [76]. In the area of large language models (LLM), the incredibly growing number of model parameters is also pushing the number of compute devices used to train them to grow outrageously. Examples include GPT3 [8] (2020, 175B, 1024 NVIDIA V100 GPUs (estimated)) Megatron-Turing NLG [104] (2022, 530B, 2240 80-GB NVIDIA A100 GPUs), PaLM [17] (2022, 540B, 6144 Google TPU v4), OPT-175B [130] (2022, 175B, 992 80-GB NVIDIA A100 GPUs), and LLaMA [111] (2023, 65B, 2048 80-GB NVIDIA A100 GPUs). Understanding the performance behavior of multi-GPU training jobs is the key to performance bottleneck identification and optimization. This helps ML practitioners save development time and budget, improves ML service quality provided to users, and avoids excessive emissions of CO_2 to the environment. Users not only want to know how fast the

training is (prediction of iteration time / query-per-sec (QPS) / FLOPS etc) but also want to know how to make training fast and how to use the hardware efficiently. However, it is also not an easy goal to achieve. The difficulties mainly come from the following two aspects:

- Communication collectives, such as *all-to-all* and *all-reduce* across various communication networks (e.g., NVLink, PCIe, Network Cards) of different topologies that connect multiple compute devices, are essential operations in multi-GPU training and commonly are the performance hotspot. The performance modeling of these operations is missing in previous single-GPU works.
- More importantly, the synchronization scenarios of multiple GPU streams on the same device or across multiple devices are complicated. Previous works only considered CPU-GPU synchronization at most so they are not sufficient to account for all the GPU idle time caused by data dependency and multi-GPU synchronization, and thus not able to accurately model the workload execution time.

Except for multi-GPU coverage, the performance modeling of certain operators (ops) in previous works is rigid and only able to cover very limited problem sizes. For example, in reality, embedding lookup in DLRM has input data with unknown distribution and loads, resulting in unpredictable access patterns to the memory and making it hard to predict its performance.

To summarize, the performance modeling problems of communication collective, embedding lookup with randomly distributed input data, and multi-GPU stream synchronization remained unsolved until we address them in this paper. We first benchmark the training of deep learning recommendation model (DLRM) and natural language processing (NLP) workloads on multi-GPU platforms to understand the performance detail. Then, we extend a previous work [60] by Lin et al. to enable multi-GPU performance modeling for ML workloads by adding the supports of:

- Performance modeling with embedding table lookup with flexible lookup numbers and patterns (i.e., input data distribution) through an ML-based approach.
- Performance modeling of communication operations (*all-to-all* and *all-reduce*) through improving a basic heuristic model with straightforward and efficient sigmoid curve fitting.

- Enhanced critical-path-based end-to-end (E2E) performance modeling algorithm.

We claim that *both inter-rank and intra-rank synchronizations are the keys to accurately model ML workloads training performance on multi-GPU, or even more broadly, different types of ML workloads, each running on groups of systems, each group of such systems with distinct compute and I/O capabilities.* Therefore, the third item above is the most critical contribution of this paper. We demonstrate the performance model's capability by obtaining 5.21% and 3.00% geomean prediction errors on predicting the per-iteration of randomly generated industrial-scale DLRM workloads and Transformer-based NLP models such as BERT [19], GPT2 [89], and XLNet [126] respectively on two different multi-GPU platforms. In addition, in a use case of embedding table sharding config selection without running the (DLRM) workloads, the performance model achieves an 85% success rate to meet the criterion, showcasing the performance model's ability to generate insights for multi-GPU training optimization.

5.2 Related Works

5.2.1 Performance Modeling for Single-Device and Distributed Training

There is abundant prior art in predicting the execution time of per-batch training and/or inference of ML workloads on a single GPU. A majority of them [46, 57, 59, 60, 87, 93, 127] focus on convolutional neural networks (CNNs), while some also cover NLP [127] and recommendation models like DLRM [60]. Although these works have not marched into the area of multi-GPU or distributed training, they compile a series of methodologies for accurately modeling performance on operator and kernel levels.

Early performance models for distributed training are mostly analytical and consider computation and communication respectively. Yan et al. [124] studied data-parallelism and model-parallelism of CNNs on CPU clusters and model their performance and scalability of them. Oyama et al. [85] also proposed a performance model that targets GPU-equipped supercomputers and not only predicts per-batch execution time but also statistics of mini-batch size and staleness for asynchronous SGD algorithm. As a use case, both approaches claimed to be able to search for the best system configurations for distributed training of CNNs. Qi et al. [88] introduced a per-layer modeling technique PALEO for CNN execution time estimation, and demonstrated its

ability to accurately model CNN training performance at scale on cloud clusters.

However, nowadays the environment and setup of training clusters have evolved significantly and outdated many settings such as sub-batch size (less than 16 [85] vs. 2048 to 65536 nowadays), model workload size and diversity, and hardware settings. As decentralized distributed training with multi-GPU is preferred against parameter servers, the problem of modeling the communication among GPUs should be addressed well.

Wang et al. [117] built a framework for the characterization of various types of DL workloads, which focused on coarse-grain (workload-level) estimation on the parameter server architecture. It also implied a possible speed-of-light (SOL) model that the total execution time is the maximum among the time of data, computation, and communication, assuming these operations are perfectly overlapped.

5.2.2 Model Parallelism and Sharding

Model parallelism becomes critical in scaling up ML workloads as they grow outrageously bigger and bigger in recent years. Lepikhin et al. [53] summarizes the main challenges of model parallelism that include: 1) device under-utilization due to the sequential dependency of the network and gradient-based optimization; 2) superlinear compute cost vs. model size; 3) poor infra scalability; 4) non-trivial implementation of partition strategy. As an example of model parallelism, modern recommendation models such as industrial DLRMs often have hundreds of embedding lookup tables with up to tens of millions of rows, which are impossible to be held on a single GPU and require to be sharded and dispatched to multiple GPUs. Per the consideration of load balancing, there are many strategies proposed for sharding embedding tables wisely so that the per-device cost (i.e., compute latency, memory storage, etc.) is balanced as much as possible. Lui et al. [64] first applied baseline (e.g., dimension-based, row-based, and size-based) heuristics to shard embedding tables in DLRMs. Sethi et al. proposed RecShard [101] that uses integer linear programming (ILP) to optimize the sharding problem. Zha et al. further improved the sharding efficiency and balance by addressing the problem with reinforcement learning in AutoShard [128] and Dreamshard [129]. In our paper, we demonstrate in a case study that our performance model is able to evaluate the performance of multiple sharding algorithms and quickly select the one that leads to the best E2E execution time of DLRM.

5.3 Backgound

5.3.1 Pytorch Benchmark Setups

5.3.1.1 DistributedDataParallel (DDP) vs. DataParallel (DP)

DDP and DP are two popular paradigms for distributed training with Pytorch. DDP's multi-process decentralized execution makes it a preferred choice for distributed training of ML models over DP. In addition, Pytorch's profiler always fails to capture some GPU events with DP, which makes the profiler traces unusable. This is most likely due to Python's global interpreter lock (GIL) that hinders DP's multi-thread scalability. Therefore, we focus on DDP (though with model parallelism for DLRM's embedding lookup module) throughout this work.

5.3.1.2 Early Barrier

Conventionally there is no explicit barrier at the beginning of each batch in distributed training. Each rank reads its own mini-batch data independently, then kicks off the training of the batch, and only synchronizes at the first communication collective. However, it is common that for each batch the data ingestion latency is different across ranks, so the training on each rank might not start simultaneously and thus may incur a waiting time for other ranks when a communication collective is launched on some ranks. Since the profiler takes this time into account when it measures the collective time, it creates confusion in measurement and evaluation, especially when the loads such as embedding table shards in DLRMs on different ranks are imbalanced. Therefore, for evaluation and prediction purposes we manually insert a barrier at the beginning of each batch to avoid this confusion and keep the latency of the first communication collective roughly equal across all ranks. This might introduce a random but tiny performance overhead compared to actual executions. However, ranks are synchronized at the end of training of the last batch by an *all-reduce*, and data ingestion is usually well-optimized (i.e., asynchronous and overlapped well by the training of the last batch) and not a bottleneck. Therefore, we can still expect our setup to be representative and accurately simulate the execution on a well-optimized training system with an almost simultaneous start on each rank.

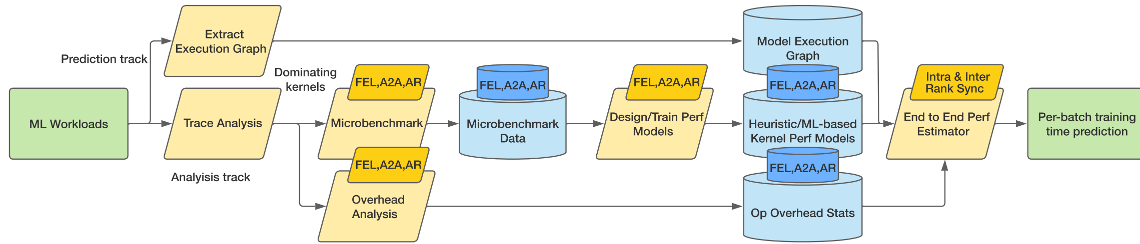


Figure 5.1: An overview of the prediction pipeline based on the one proposed by Lin et al. [60], with new components marked with small shapes and italic texts, such as microbenchmark and kernel-level performance models of FBGEMM embedding lookup (FEL), all-to-all (A2A), and all-reduce (AR), as well as inter- and intra-rank synchronization mechanism in the critical-path algorithm to handle multi-GPU end-to-end performance prediction.

5.3.1.3 Gradient Bucketing

Gradient bucketing is proposed by Li et al. [56] to resolve the problem of low bandwidth utilization and data-dependency-caused execution blocking due to frequent *all-reduce* launches and achieve higher throughput and lower latency. We adopt the default setting gradient bucket size (i.e., 25 MB) in our experiments, as altering it only brings marginal benefits to the training performance for a very limited number of workloads. Notice that the execution graph of the first training iteration does *not* reflects gradient bucketing because gradient exchange is completely unbucketed during the first iteration. We thus extract the execution graphs of the second and onward iterations for prediction purposes.

5.4 Methodology

The focus and contributions of this work are specifically on the performance modeling on single-node multi-GPU platforms. Therefore, we leverage and extend a modular and highly extendable previous work on single-GPU performance modeling by Lin et al. [60] as the starting point. The reason why this makes sense is that kernel performance models are reused and shared across different ML workloads as inputs, and the E2E execution time prediction is done with a traversal of ops in the execution graph of the target workload, which is dictated by the same algorithm. This means that the E2E performance model is *model-architecture-agnostic*. To extend the support of new ML workloads and multi-GPU platforms, what we need is just adding kernel

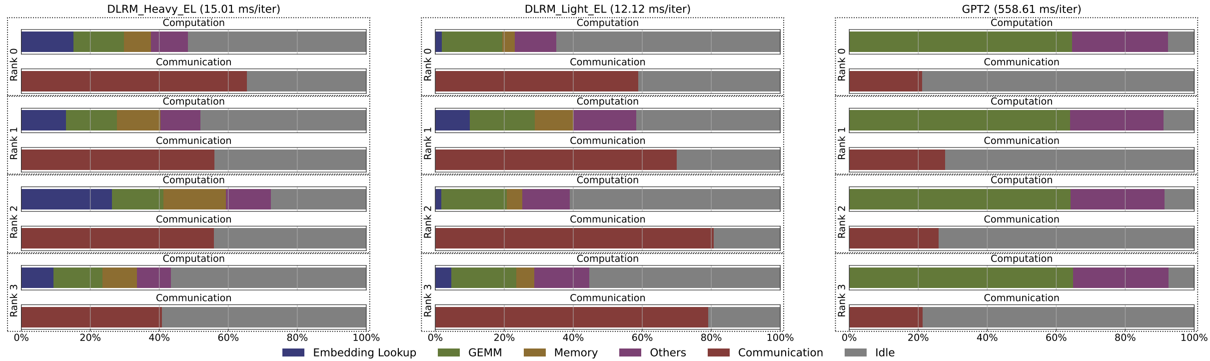


Figure 5.2: Per-GPU-stream training execution time breakdown of selected ML workloads on a 4-GPU platform. The per-iteration time of each workload is provided for reference. We discuss these results in Section 5.4.2.

performance models for dominating ops/kernels of these new workloads, and modifying the algorithm designed for single-GPU so that it can handle the simulation of multi-GPU execution.

Figure 5.1 depicts an overview of our prediction pipeline. It inherits the main modules and procedures and reuses all kernel performance models (for *GEMM*, *memcpy*, *transpose*, etc) and the overhead estimator from the original pipeline Lin et al. introduced. The logic of the prediction pipeline remains unchanged. To accumulate system assets ((blue cylinders in Figure 5.1)), ML workloads written with Pytorch are profiled with execution graphs extracted and trace analyzed for dominating ops. Then, we collect microbenchmark data for target kernels/ops, design/train and verify performance models using this data, and extract overhead statistics for these kernels/ops as well. In the prediction phase, the pipeline takes the execution graphs of a workload as the input, runs with the assets, and predicts the per-iteration training time of the workload in seconds. In this work, we focus on DLRM and Transformer-based workloads, although it is not limited to being used on many other types of ML workloads.

5.4.1 Dataset Exploration

The open-source DLRM dataset [73] contains synthetic embedding lookup data that resemble the memory access reuse pattern of Meta's production data. The 2021 data includes a batch of 65536 samples embedding lookup data of 856 tables, while the 2022 data includes a batch of 131072 samples of 788 tables. Due to the unpredictable lookup pattern in production data, the pooling factors (average number of lookups per data sample, denoted with L) vary significantly

across all tables. This is very different from what has been studied in previous work [60, 112], where L is a fixed value across data batches and even tables in the same workload. Exploratory analysis (Figure 5.3) shows that average L values of tables in the dataset tend to concentrate in the range of $[0, 10)$, especially $[0, 1]$, while there are still a number of tables with average L value spreading to a few hundred. We define tables with an average L equal to greater than 20 as "heavy tables". As shown in Figure 5.3, there are 103 and 166 heavy tables in the 2021 and 2022 data respectively.

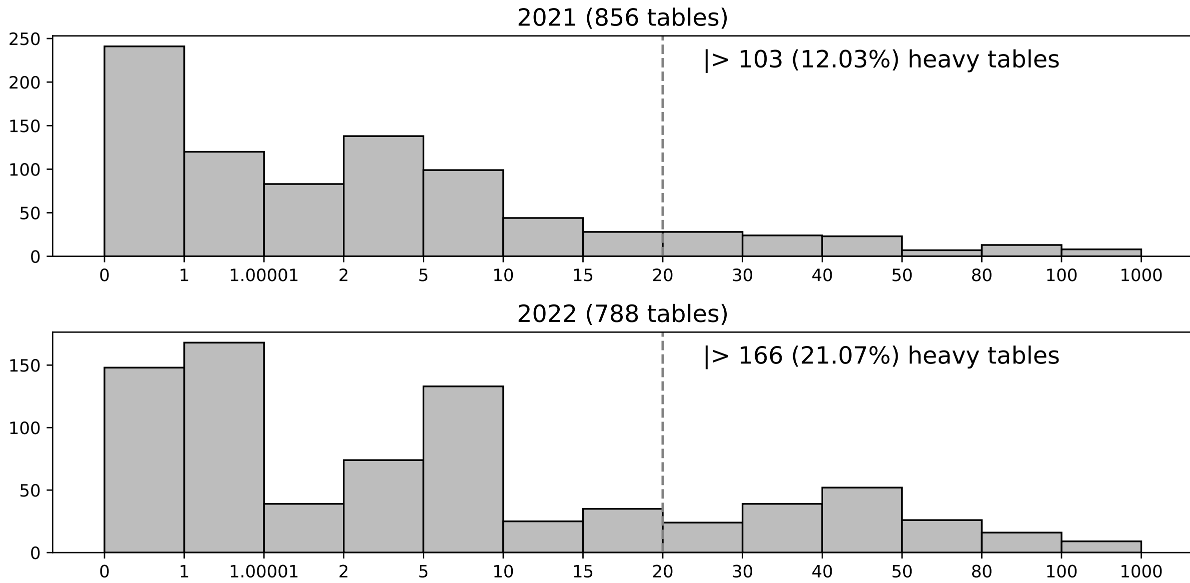


Figure 5.3: Histogram of average L values of embedding lookup tables in the dataset. All bins but the last one are right-open.

5.4.2 Multi-GPU Training Benchmark and Analysis

We first benchmark several ML workloads that will be further analyzed in later sections to understand the behavior of multi-GPU training performance. Figure 5.2 shows the per-GPU-stream execution time breakdown of three workloads: *DLRM_Heavy_EL* with the batch size 4096 (Task_14 in Section 5.5.2.1, involving heavy embedding tables only), *DLRM_Light_EL* with the batch size 4096 (Task_5 in Section 5.5.2.1, involving random embedding tables), and *GPT2* with batch size 64 on each rank of the 4xGV100 platform (details shown later). The breakdown is attributed to a few groups of kernels and types: embedding lookup (forward and backward), GEMM (forward and backward), memory operations (*concat*, *copy*, *transpose*, etc),

others (normalizations, elementwise, etc), communication (*all-to-all* and *all-reduce*), and idle time. A few insights can be summarized from the results:

- **Communication operations can be but is not always a performance bottleneck.** It is obvious that communication operations dominate the two DLRM models. This is mainly because in addition to the normal *all-reduce* caused by accumulating gradients in the backward pass, training the embedding lookup module in DLRM with model-parallelism introduces the *all-to-all* operations for merging data with that from the bottom and top multilayer perceptron (MLP) modules trained with data-parallelism. In most of the stream pairs shown here, except for *DLRM_Heavy_EL*'s rank 2, the communication stream has a longer GPU active time than that of the computation stream, which means it can never be fully overlapped by the computation and thus similar workloads tend to thrash the communication network of the interconnected multi-GPU platform. On the contrary, GPT2 (and most of the Transformer-based language models) is quite compute-dominated; *all-reduce* time is minor and can theoretically be fully overlapped.

- **Load balance substantially varies across the workloads.** DLRM models experience load imbalance across ranks due to the existence of embedding lookup tables with various compute and communication loads. Notice that instead of the forward and backward computation of embedding lookup itself, it is in fact the *all-to-all* it introduces that dominates, *especially when the embedding lookup is light*. Examples such as *DLRM_Light_EL*'s rank 3 and 4 show that communication can occupy up to 80% of the execution time, in which case GPU computation resources are wasted by sitting idle. On the contrary, we see an almost-perfect load balance for GPT2 as it is purely DDP-trained and the load is evenly distributed.

- **The significance of GPU stream idle time is way more than "being non-negligible."** What we see in single-GPU training is that idle time is contributed mostly by CPU op calls and overheads that block the scheduling of GPU kernels; data dependencies play a less important role given there is only one stream. However, the case of multi-GPU training is reversed, meaning that data dependencies of ops lead to stream waiting time between each

other. This time cannot be predicted in the same way (i.e., statistically) as the previous work such as Lin et al. [60] did, and it raises the problem of synchronizing each GPU stream in order to predict the per-iteration time of training.

- **"Others" ops do not mean "minor" ops when they are aggregated on (Transformer-based) NLP models.** We see that "others" ops (*layer-norm*, *dropout*, *gelu*, *tanh*, etc) contribute to more than 20% of execution time on GPT2, although each one of them only contributes a tiny amount of time every time it is executed. This might not be a surprise, given the performance of Transformer-based models has been profoundly studied these years. There are a bunch of ways such as advanced layer-fusion that can effectively optimize and reduce their execution time, but that is beyond the scope of this paper. Under our experiment settings and from the performance modeling point of view, this observation basically means more ops from this category should be supported than the previous work did.

The above two subsections guide the direction of our research which includes the following components:

- (Section 5.4.3.1) Embedding table lookup with flexible lookup numbers and patterns (input data distribution);
- (Section 5.4.3.2) Communication operations like *all-to-all* and *all-reduce*;
- (Section 5.4.3.3) Additional ops in (Transformer-based) NLP models, such as *layer-norm*, *dropout*, *gelu*, and *tanh*, etc;
- (Section 5.4.4) Advanced E2E performance modeling for multi-GPU training.

5.4.3 Kernel Performance Modeling

5.4.3.1 Embedding Lookup Kernel Modeling

The performance modeling of *Embedding lookup (EL)*, an indispensable and commonly dominating [60] operator in many recommendation models including DLRM, is challenging in a few aspects. First, the dimensions and parameters of every single embedding table, specifically

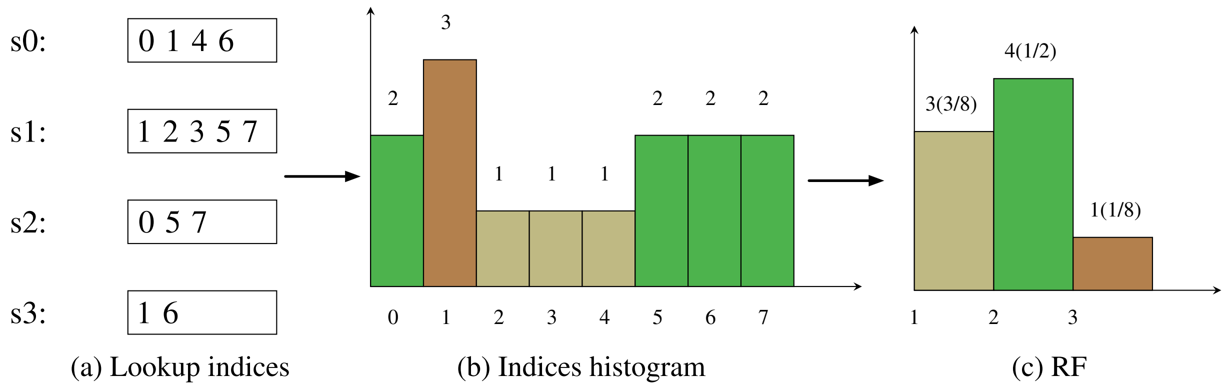


Figure 5.4: Generation of reuse factors (RF) from embedding lookup indices. (a) is an example of lookup indices of a batch of 4 samples on one embedding table. In (b), the x-axis is the indices and the y-axis is the count of these indices. In (c), the x-axis is the count of accesses and the y-axis is the number of such indices.

number of embeddings (E), *embedding dimension* (D), and *pooling factor* (L), can all be different in real workloads and data. Second, the distribution of input data to each table varies considerably across each batch. Both these two factors significantly increase the difficulty of applying a heuristic-based performance model to the *EL* op, since it is almost impossible to accurately estimate the L2 cache hit rate and data movement and thus predict the latency of the op. Previous work such as Lin et al. [60] only considers modeling the performance of the case when D, E, and L are all fixed for each table and data distribution is uniform. Therefore their solution is not sufficient to deal with real-world workloads and data.

To address this issue, we adopt a straightforward and clear expression of describing *EL*'s input data distribution using *reuse factors* (RF), as introduced by Meta's open-source DLRM dataset [73]. Figure 5.4 shows an example of generating reuse factors (RF) from embedding lookup indices. First, the histogram of per-table index counts in a batch of input data is calculated ((a) to (b)). Then, the "histogram of histogram," i.e., how many indices are accessed once, twice, thrice, etc, is calculated, which is thus the RF of this batch on a certain table ((b) to (c)). In practice, the bins for this step have sizes of exponents of 2, e.g., $[0, 1)$, $[1, 2)$, $[2, 4)$, $[4, 8)$, etc. Finally, the counts of every bin are normalized into the range of $[0, 1]$. In this way, the distribution of lookup indices is described as by what probability any row of a table is hit by a certain range of times, such as 2^m to 2^n times where m and n are both integers and $m < n$. Meta's DLRM dataset provides the RF of 17 bins for each data file, and we follow this convention in our experiments.

The advantages of RF are its low overhead in terms of both processing time and parameter size. The calculation is relatively simple, and thus it is trivial to obtain RF values of each batch of input data on the fly before the training of a batch is actually started. Because each table has 17 RF values and the number of tables residing on each rank is usually limited, e.g., around 10, the total number of RF values for each batch of input data remains acceptable.

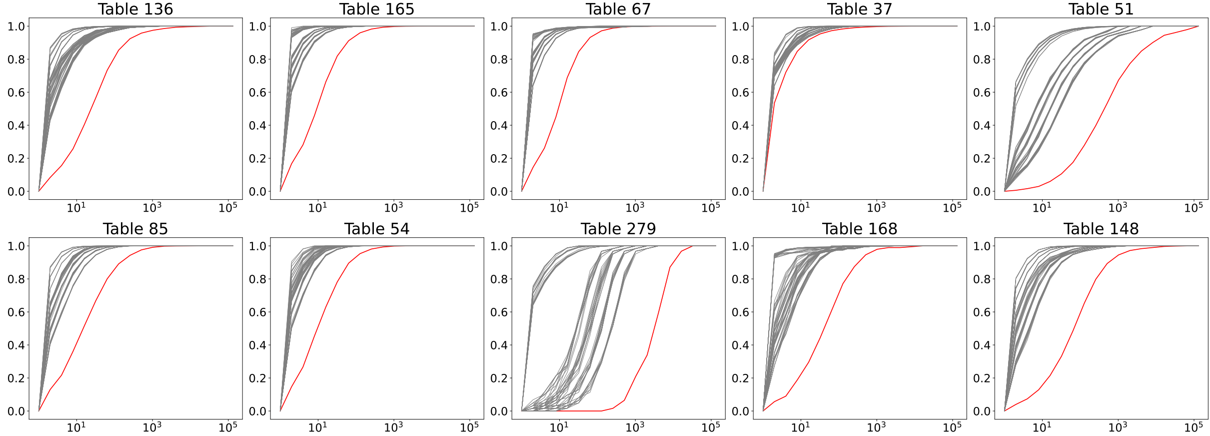


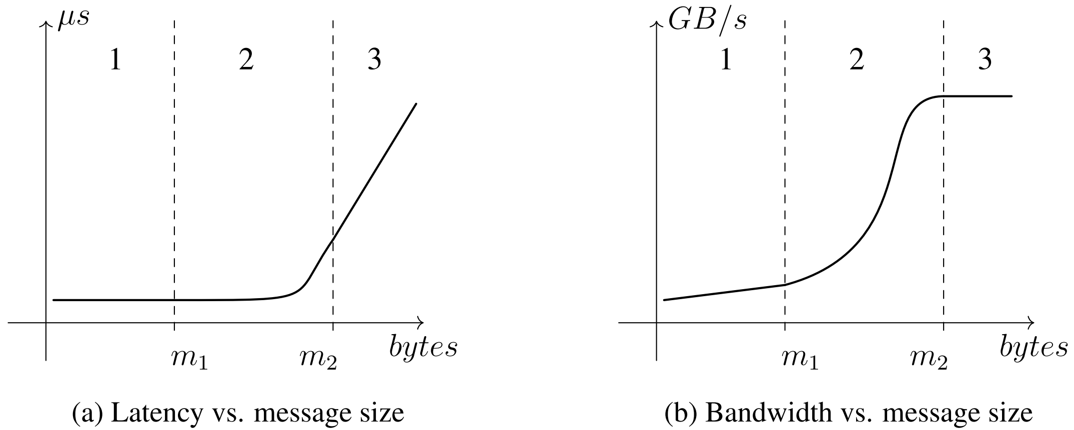
Figure 5.5: RF values of top 10 most visited tables among 6000 random samples from the DLRM open-source dataset for EL microbenchmark. Red lines represent the cumulative distribution function (CDF) of overall RF values of each table in the whole dataset, while each thin gray line represents the CDF of RF values of a sample that visits a certain table. It can be clearly observed that the data distribution (represented by RF values) of each data batch is different from that of the whole dataset. Compared to overall RF values, the RF values of a data batch tend to concentrate towards 0 along the X axis.

We adapt the cost model proposed by Zha et al. [129] for our use of real-time performance modeling of *EL*. Zha et al. create a multi-head MLP cost model to simultaneously predict the cost of the forward time, backward time, and communication time of a sharding scheme of *EL* as a whole on multiple devices. Our method is different from it in two aspects. 1) We handle these three constituent times separately in the granularity of ops to accommodate our robust E2E algorithm shown in Section 5.4.4. This better simulates and models the actual execution since considering these three times as a whole misses the opportunity to explain the computation-communication overlaps (also elaborated in Section 5.4.4). 2) Instead of using RF values of each table, we use RF values of each *batch* for better prediction based on the reason stated in the last paragraph. We construct our MLP performance model resembling one of the heads in Zha et

al.'s cost model and match those used for other kernels for convenient training and inference. To enable *EL*'s execution incorporating flexible D, E, and L, in this work, we use a popular, efficient, and flexible batched-EL implementation provided by FBGEMM [47], an open-source high-performance kernel library for training and inference on both CPUs and GPUs. We first create a microbenchmark dataset by randomly sampling tables and batches (e.g., 10 tables out of 856, and a batch of 1024 lookups out of 65536) from the DLRM open-source dataset, and make sure that the size of the sampled tables in each sample will not exceed the DRAM memory size of the selected GPU. Note that while in the previous work, the cost model is trained with RF values of *the whole dataset*, it is improper to do so for real-time performance modeling because RF values tend to spread in a larger range. This fact is easy to understand intuitively. Suppose the batch size is small, the number of times each row in a table is hit tends to concentrate close to 0 because there are few indices in the batch. If we consider the whole dataset as a huge batch (e.g., size 65536 for the DLRM dataset we use), it is very likely that there are some rows in a table to be hit many times, such as 500 times, which could never appear with a small batch size such as 256. In a word, these two distributions are different (as shown in Figure 5.5), and thus the distribution of the whole dataset is not representative of a batch of data sampled from it. Therefore, we calculate the RF values of *each batch* and store them together with the execution time as part of the training data. Finally, we train and validate the performance models with the microbenchmark results as the input dataset that is split into a training set and a test set with a split factor of 0.8.

5.4.3.2 Communication Collective Performance Modeling

All-to-all and *all-reduce* are two communication collectives commonly seen in multi-GPU training of recommendation workloads: *all-to-all* is usually called to redistribute intermediate results from multiple devices during the transition from model-parallelism to data-parallelism; *all-reduce* is indispensable in the backward pass as it aggregates gradients across all devices. The latency of these two ops contributes significantly to the per-iteration time of multi-GPU training. The difficulty of modeling the performance of these two ops is that their performance highly depends on how multiple devices are connected (e.g., network connection pattern and medium such as NVLink and PCIe), while this configuration differs from platform to platform.



(a) Latency vs. message size

(b) Bandwidth vs. message size

Figure 5.6: Typical characteristic curves for data movement. X-axes are in log scale. m_1 and m_2 are boundary message sizes that separate the three sections.

We devise a simple way to model the performance of *all-to-all* and *all-reduce*. Grama et al. [28] define the cost of transferring a message between two nodes on a network as

$$t_{comm} = t_s + t_w m, \tag{5.1}$$

where t_s is the startup latency for small messages, t_w is the per-word transfer time, and m is the message size in words. Although simple enough, this basic model might miss the accuracy at the transition region where the second term starts to dominate. Later, Li et al. [54] present a comprehensive study on multi-GPU communication on platforms with different settings. From this study, we distill a critical observation on the relationship of message size versus bandwidth and data transfer latency:

Regardless of the ops, network connection pattern, and network medium, as shown in Figure 5.6 the curve of message size versus bandwidth can always be divided into three sections as the message size increases: 1) linear bandwidth with constant latency; 2) S-shape bandwidth curve with non-linear latency; 3) constant (saturated) bandwidth with linear latency.

The same type of characteristic functions is also mentioned in the study of the relationship between granularity (bytes) and the accelerator's speedup in the work of LogCA performance model [4]. It inspires us to treat the size-latency characteristic as a piece-wise function and fit each piece respectively. The latency of section 1 can be modeled by the measured constant data

movement overhead; the latency of sections 2 and 3 can be modeled by $t = m/BW$, where BW of section 2 is fitted with a sigmoid function while that of section 3 is simply the measured maximum *algorithm* bandwidth specific to the platform and op. Overall, the data movement latency can be expressed as a piece-wise function of message size m:

$$t_{comm} = \begin{cases} t_s & \text{if } m \leq m_1 \\ \text{f}(m, param) & \text{if } m_1 \leq m \leq m_2 \\ t_s + \frac{m}{BW_{max}} & \text{if } m \geq m_2 \end{cases} \tag{5.2}$$

where

$$f(m, param) = \frac{\log_2^m}{10^{sigmoid(m,param)}} \tag{5.3}$$

and $param$ include 4 parameters L, x_0, k, and b that define the function

$$sigmoid(x) = \frac{L}{1 + e^{-k*(x-x_0)}} + b. \tag{5.4}$$

Equation 5.2 is in fact an improvement of Equation 5.1. This method only involves 8 parameters for each op (4 for sigmoid, 2 for section boundaries, 1 for startup latency, and 1 for maximum bandwidth). It is fast and simple for both curve fitting and prediction and is topology-agnostic that can be generalized to any communication patterns. For both ops, we utilize the PARAM benchmark [72] and conduct a simple microbenchmark with equal message size starting from 4 bytes on each device and doubling it until exceeding the device memory. This data is used for curve fitting. The section boundaries m_1 and m_2 are simply determined when the symmetric differences of the data points are greater than 1.1 and less than 0.9, respectively. To test the accuracy of the performance model, because the message size might be different across all devices, we modify the communication microbenchmark in PARAM and sample 20000 data points with random message sizes on each device for *all-to-all* and each multi-GPU platform. We calculate the input message size to the performance model as "the maximum of (the maximum of sent/received message size per device) across all devices". For example, suppose the configuration of all-to-all is "256,1-2-4-7,256", meaning the batch size is 256, the embedding dimension is 256, the device count is 4 and each device has 1, 2, 4, and 7 tables respectively. The input message size is $256 \times \max(\max(1, 13), \max(2, 12), \max(4, 10), \max(7, 7)) \times 256 \times 4$ bytes/float $= 3407872$ bytes. Since the input message sizes for *all-reduce* across different devices are always the same,

we randomly sample 1000 data points with *equal* message sizes on each device for accuracy testing. The ranges of message sizes for both collectives match practical problem sizes that occur in real DLRM workloads, i.e., batch size ranging in $[256, 512, 1024, 2048, 4096]$, table number ranging in 1 to 20, and embedding dimension ranging in $[32, 64, 128, 256]$. In addition, as we notice that *all-reduce* only occurs in the backward pass, there is no need to create a performance model for its backward latency prediction. Also, the backward operation of *all-to-all* is exactly the reverse operation of the forward *all-to-all*, which means the message sizes on each device are exactly the same with just the transmission direction reversed. It means its latency is thus theoretically equal to its forward counterpart, given bidirectional links are almost always symmetric.

5.4.3.3 Additional Ops Performance Modeling Support

We add kernel performance models for additional ops including *layer_norm*, *dropout*, and element-wise ops such as *gelu* and *tanh*. Using the same paradigm as the base model[60], we predict the latency of *layer_norm* and *dropout* with ML-based models trained with microbenchmark data of Pytorch ops, while that of element-wise ops is predicted using the roofline model.

5.4.4 Multi-GPU Performance Modeling

As we mentioned, the main difference between single-GPU and multi-GPU performance modeling is that in multi-GPU scenarios, there are communication collectives like *all-to-all* and *all-reduce* that simultaneously execute on multiple devices and trigger synchronizations and waiting across ranks and streams. Therefore, the simulation of such synchronizations and waiting during the execution analysis is required, and this section elaborates on how this problem is handled.

Before we dive deep into the multi-GPU E2E performance modeling algorithm, we need to first understand two types of synchronization in distributed training. They are:

- *inter-rank synchronization* that occurs at the termination of a communication collective kernel, and
- *intra-rank* or *inter-stream synchronization* that happens at the launch of a compute/memory kernel that depends on the last communication kernel.

These two types of synchronization are depicted in Figure 5.7. It is expected that communication kernels' launch time is different across ranks because of the different latency each rank takes to reach these ops due to unbalanced prior loads, such as data movement and computation. However, since communication kernels are synchronous, theoretically they must terminate at exactly the same moment, while in reality they also terminate almost at the same time with negligible variance. This is the *inter-rank synchronization*, marked by the red dashed line in the plot. When it occurs, the time front of the communication stream on different ranks should be set to the same value during analysis. On the other hand, since the output data of the communication ops are used as inputs to certain successive ops, successive communication kernels, e.g., the last communication kernels on S_{cm} on both ranks in Figure 5.7, should be launched at least after the *first dependent kernel* of the previous communication kernel (marked in blue dashed lines) rather than right after the previous communication kernel. This is called *intra-rank/inter-stream synchronization*. To reflect it, we set the communication stream time front to the same value as the launch time of the first dependent kernel.

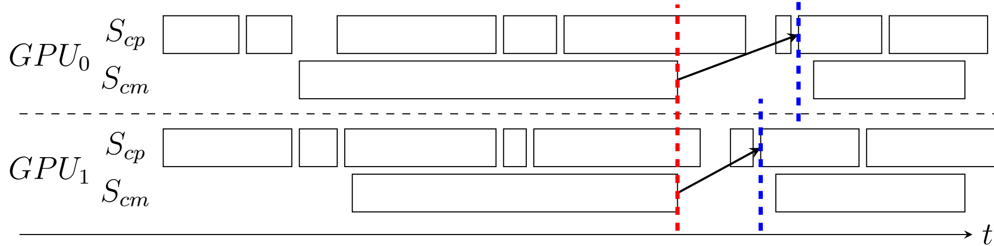


Figure 5.7: Two types of synchronization: inter-rank synchronization (red) and intra-rank/inter-stream synchronization (blue). For simplicity, we assume two GPUs and two streams (S_{cp} and S_{cm}, for compute and communication respectively) per GPU, while CPU op calls are omitted in the plot. GPU kernels are represented by rectangles, and arrows indicate the data dependency between compute and communication kernels.

We claim that *both inter-rank and intra-rank synchronizations are the keys to accurately modeling ML workloads training performance on multi-GPU, or even more broadly, different types of ML workloads, each running on groups of systems, each group of such systems with distinct compute and I/O capabilities.* This is because the best modeling method without them is to sum the kernel time per GPU stream and take the maximum as the E2E prediction result. It can possibly work for a few workloads that are completely dominated by one certain type of

stream, e.g., GPT2 being dominated by the compute stream as shown above. However, in other cases, this will miss a large amount of GPU idle time (e.g., the gap between the red and blue lines in 5.7) created by data dependency and other possible waits happen during the execution, which will result in huge prediction error.

To take these synchronizations into consideration, the information on data dependency between ops in the ML model/workload is needed, which is thankfully provided by the Pytorch execution graph (EG). We extend the critical-path-based algorithm proposed by Lin et al. [60] for single-GPU performance modeling to adapt to multi-GPU scenarios by incorporating inter-rank and intra-rank synchronizations, as shown in Algorithm 4. The new algorithm has a similar basic structure as that of the single-GPU version. For simplicity, we only assume two streams on each rank: one for compute and memory kernels and one for communication kernels. Instead of running one single process for training time prediction on single-GPU, the new algorithm runs N processes in parallel, processing N EGs and predicting execution time for N ranks simultaneously (line 4). In addition to CPU time and GPU active time, communication stream time T_{cm} and compute/memory stream time T_{cp} are also tracked per process (line 6). At line 7, we also initialize a variable $last_comm_op$ to keep track of the dependency between compute/memory ops and the last communication op. During the traversal of EGs, intra-rank synchronization happens when a dependent op of the last communication op or a new communication op is encountered (lines 12-15); when the current op is a communication op, we do inter-rank synchronization after processing all kernels of it (lines 29-31) and update $last_comm_op$ for the next iteration (lines 36-38). At the end (lines 40-41), we calculate the local total time and GPU active time and do an *all_gather* operation across all ranks to collect their values, the maximums of which are returned as the predicted total time and GPU active time.

In the experiments to predict the multi-GPU training performance of DLRM models, we first generate 20 tasks with embedding tables randomly sampled from the DLRM open-source dataset. Among these 20 tasks, each of the 2021 and 2022 parts of the dataset contributes 5 heavy tasks (i.e., all sampled tables are heavy) and 5 normal tasks (i.e., tables can either be heavy or light). The total number of embedding tables per task is in the range of $(0.7 \sim 1.3) \times \#$GPU $\times$ 13. To prevent out-of-memory error, for each task the sum memory footprint of all embedding tables

on each rank is guaranteed not to exceed 80% of the GPU's DRAM size. The default sharder is size_lookup_greedy based, i.e., the cost of each table is estimated as $L \times D \times \log(E)$. For each training iteration in the E2E test of each task, a mini-batch of data (with batch size set to 512/1024/2048/4096) corresponding to each table selected in the task is randomly sampled from the dataset and distributed to each rank based on the sharding scheme. The overhead statistics (mean latency, etc) of Pytorch ops are aggregated from all collected traces and shared by *all* tested workloads. The actual/predicted time is measured/calculated by averaging over 30 iterations.

5.5 Evaluation and Analysis

(a) 4xGV100 (b) 4xA100

Figure 5.8: Communication topologies of the two multi-GPU platforms used in our experiments. Thin lines: 4 NVLinks (NV4); thick lines: 12 NVLinks (NV12); dashed lines: PCIe.

We evaluate our kernel and E2E performance models on various platforms with Pytorch v2.0 along with FBGEMM v0.4.1, CUDA 11.7, and Python 3.9. The performance model of FBGEMM's embedding lookup kernel is evaluated on single NVIDIA GV100 and A100 (40 GB) GPUs, while that of both communication kernels and multi-GPU E2E are evaluated on two multi-GPU platforms, including 4xGV100 equipped with 48-core Intel(R) Xeon(R) Gold 6146 CPU @ 3.20 GHz, and 4xA100 equipped with GCP's a2-highgpu-4g with 48-vCPU. The GPU communication topologies of these two platforms are shown in Figure 5.8. We use data sampled from the DLRM open-source dataset [73] as both microbenchmark data for FBGEMM embedding lookup kernel performance model training and verification, and input data for embedding lookup of the DLRM models in E2E tests. Multiple pieces (.pt files) of the dataset are merged for later use. The multi-GPU E2E evaluation in this work covers DLRM models training (code adapted from `https://github.com/facebookresearch/dlrm`) and

Algorithm 4 E2E Multi-GPU Training Performance Model.

1: **Input:** $EG_{0,1,...N-1}$ of an ML workload trained on a single-node N-GPU platform, one EG per rank; Kernel performance models $\{M\}$; Overheads Ov.
2: **Output:** Predicted per-batch training time T.
3: Spawn N processes $P_{0,1,...,N-1}$.
4: **parfor** $i \leftarrow 0, N-1$ **do**
5: Initialize $cpu_time = 0$ and $gpu_time = 0$ for P_i.
6: Initialize communication stream time T_{cm} and compute stream time T_{cp} for process P_i.
7: Initialize $last_comm_op$ as none.
8: **for** each *op* in EG_i **do**
9: Look up $T1, T2, T3, T4, T5$ from Ov for *op*.
10: Identify current stream s (as *cm* or *cp*).
11: ▷ Intra-rank synchronization
12: **if** *op* depends on $last_comm_op$, or *op* is an communication op **then**
13: Set $last_comm_op$ to none.
14: Synchronize T_{cm} and T_{cp} for process P_i.
15: **end if**
16: cpu_time += $T1$
17: **if** *op* has kernel calls **then**
18: cpu_time += $T2$
19: **for** each kernel call k under *op* **do**
20: Predict kernel time T_k with M
21: $T_s = \max(T_s + 1, cpu_time + T4/2) + T_k$
22: Update gpu_time with T_s and T_k.
23: cpu_time += $T4$
24: **if** k is not the last kernel **then**
25: cpu_time += $T5$
26: **end if**
27: **end for**
28: ▷ Inter-rank synchronization
29: **if** *op* is a communication op **then**
30: Synchronize T_{cm} across all processes.
31: **end if**
32: cpu_time += $T3$
33: **else**
34: cpu_time += $T5$
35: **end if**
36: **if** *op* is a communication op **then**
37: $last_comm_op = op$
38: **end if**
39: **end for**
40: $T = \max(T_{cm}, T_{cp}, cpu_time)$
41: Synchronize T and gpu_time across all processes and take their maximums.
42: **end parfor**

finetuning of natural language processing (NLP) models such as BERT [19], GPT2 [89], and XLNet [126] (all code adapted from HuggingFace's Transformers library [122]).

Table 5.1: Prediction error of FBGEMM embedding lookup, all-to-all, and all-reduce kernel performance models. **ELF** (embedding lookup forward), **ELF** (embedding lookup forward), **A2A** (all-to-all), **AR** (all-reduce).

Kernel	4xGV100			4xA100		
	GMAE	MAPE	std	GMAE	MAPE	std
ELF	4.37%	7.11%	7.44%	5.64%	9.17%	9.52%
ELB	3.08%	4.42%	3.39%	3.63%	5.44%	4.33%
A2A	6.28%	9.42%	7.76%	5.25%	7.14%	4.72%
AR	6.35%	9.17%	8.23%	4.98%	6.77%	4.11%

5.5.1 Kernel Performance Modeling

We obtain less than 10% GMAE and MAPE prediction errors for all kernel performance models shown in Table 5.1. Particularly, the adjustment of *all-to-all* message size yields a low prediction error of latency, implying that the operation is bounded by the biggest per-device data bulk sent from or received by one certain device. Specifically, we present the fitted curves for *all-to-all* benchmark data on both platforms in Figure 5.9. We observe that the practical problem size for *all-to-all* in DLRM workloads lies in section 2 (the transitional section), which justifies our improvement on Equation 5.1.

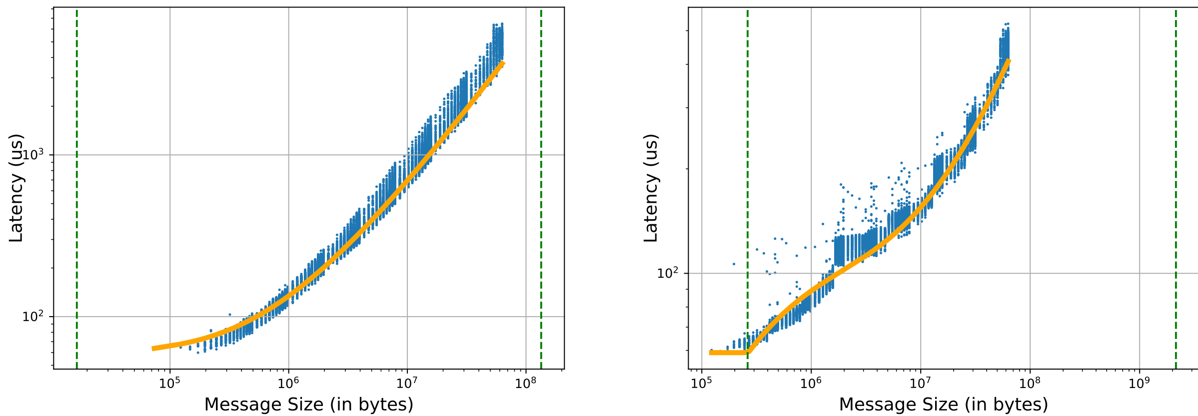


Figure 5.9: Fitted curves for *all-to-all* benchmark data on 4xGV100 (**left**) and 4xA100 (**right**). Message size section boundaries are plotted as green vertical dashed lines.

5.5.2 Multi-GPU E2E Performance Modeling

5.5.2.1 DLRM on Multi-GPU Platforms

We present the statistics of all E2E performance modeling tests on the two multi-GPU platforms in Table 5.2, and the prediction error and the reference time of each task in Figure 5.10. The baseline result to be compared with our prediction in Figure 5.10 is given by the maximum sum of kernel active time of each GPU stream. We can see that this baseline prediction, yielding error values higher than 60%, is not sufficient to be used as the predicted E2E time per iteration because no idle time nor waiting time caused by data dependency between the streams is considered. Instead, our enhanced algorithm can accurately predict both normal and heavy tasks with high accuracy, with an overall geomean prediction error of 5.21%. The majority of prediction results on both devices underestimate the actual time. One possible reason is that with big batch sizes, the communication time dominates the per-iteration time so the syncing and waiting time among all ranks also increases and contributes to the per-iteration time. The remaining tests that overestimate might be explained by the overestimation of GPU idle time caused by CPU overheads when the workload is latency-bound. Device-wise, the behavior of prediction errors, such as geomean/minimum/maximum error and trends when the batch size changes, does not deviate much, which justifies the consistency and stability of our prediction algorithm across platforms.

Table 5.2: Statistics of DLRM E2E time prediction errors across two multi-GPU platforms.

Overall			4xGV100			4xA100		
g.m.	min	max	g.m.	min	max	g.m.	min	max
5.21%	0.05%	19.38%	5.60%	0.27%	19.38%	4.85%	0.05%	17.87%

5.5.2.2 NLP Models Performance Modeling

We also test our E2E performance model on Transformer-based NLP models including BERT, GPT2, and XLNet. From Figure 5.11 we see that the absolute prediction errors are less than 10% in all tests but in two where they slightly exceed. The geomean prediction error of all the presented tests is 3.00%. Also, the variance of the prediction errors is obviously lower than that of DLRM workloads. The reason is that: 1) these NLP models are compute (GEMM) dominated with the communication stream being well overlapped by the compute stream; 2) the loads

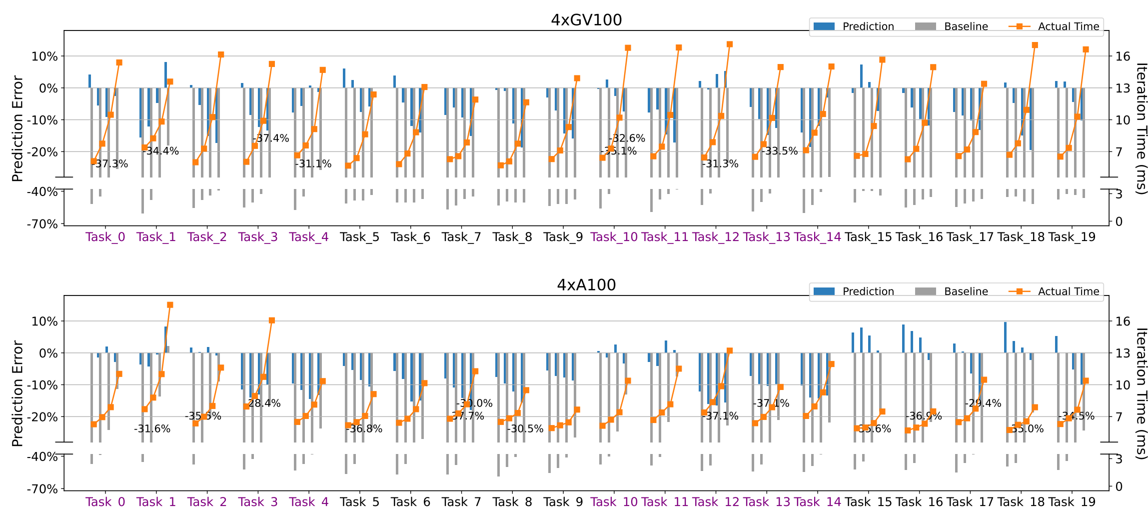


Figure 5.10: Prediction, baseline, and reference of multi-GPU training performance of DLRMs on the two multi-GPU platforms. Notice that the "Task_x"s on different platforms do not have the same embedding table configuration because the random task generation is platform-specific. Tasks 0-9 and 10-19 on each platform are from the 2021 and 2022 parts of the dataset, respectively. Heavy tasks are marked with purple color. Percentage figures are the error of bars ending in the broken area.

across devices are more balanced than the DLRM workloads, thus intra-rank and inter-rank synchronizations are rare and have little interference in the prediction. With our highly-accurate kernel performance models, we are able to precisely predict the aggregation of compute kernel time and subsequently the E2E time of these models.

5.6 Application and Discussion

5.6.1 Case Study: Fast Sharding Config Selection Using Performance Modeling

Industrial DLRM models can take days to train. Therefore, selecting a sharding configuration (i.e., the way to distribute embedding tables to multiple GPUs) that balances the loads on GPUs and speeds up the E2E per-iteration time is critical to reducing their training costs. Industrial sharding configs might consist of a sharding algorithm (greedy, multi-cost greedy, etc), cost functions (multi-cost, memory-based, compute-based, etc), table partition (column-based or row-based), and memory placement of tables (HBM/UVM). Previously, the best config is selected by benchmarking and grid-searching over a big search space formed by these factors, which can take as long as one day per workload. We consider using our performance model for this task so

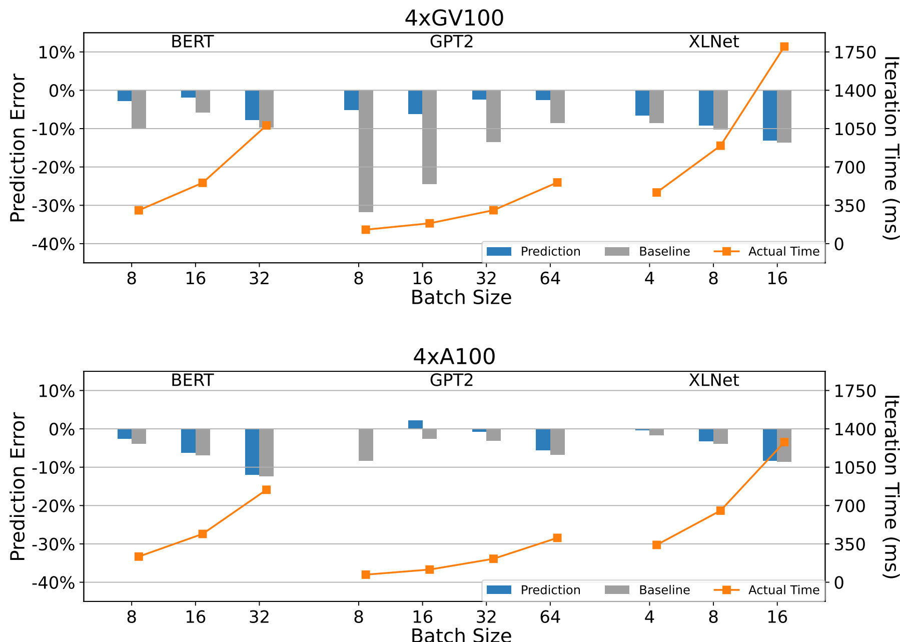


Figure 5.11: Prediction, baseline, and reference of multi-GPU training performance of BERT, GPT2, and XLNet on the two multi-GPU platforms. Batch sizes greater than 32 on BERT and 16 on XLNet result in out-of-memory errors and these tests are thus skipped.

that *without running the workload*, the selection time can be shortened to around 1 minute, with E2E per-iteration time predicted in seconds for each config. Notice that we are *not* proposing a smart sharding algorithm here like some previous works did [101, 128, 129]. Instead, our goal is to quickly evaluate these sharding algorithms or configs and pick the fastest one for a specific problem size *without benchmarking the model*.

Table 5.3: Sharders and their indexing (i) or cost (c) functions. An indexing function assigns a table directly to a certain rank, while a cost function estimates the cost of a table for the greedy algorithm.

Sharders	Functions
naive	(i) lambda x : $x.idx\%ngpus$
random	(i) lambda x : $random(ngpus)$
size_greedy	(c) lambda x : $x.E$
lookup_greedy	(c) lambda x : $x.L * x.D$
norm_lookup_greedy	(c) lambda x : $x.L/x.E$
size_lookup_greedy	(c) lambda x : $x.L * x.D * np.log10(x.E)$

We conduct an experiment to demonstrate how our performance model can quickly select the best sharding config for DLRM training on multi-GPU. To demonstrate the idea with simplicity, we only consider sharding algorithms as the config and omit all other factors mentioned above.

Table 5.4: Embedding table sharding config selection experiment results. Time is measured in microseconds (us). Abbreviations: **P**: predicted; **A**: actual; **AP**: actual (time) of predicted (fastest config); **S**: size; **L**: lookup; **SL**: size_lookup; **NL**: norm_lookup. Abs error is given by $(Time_{AP} - Time_A)/Time_A \times 100\%$, which is 0 when the selected fastest config is exactly the actual fastest config. Notice again that Task_x on different platforms are different workloads.

Platforms	Tasks	Fastest (P)	Time (P)	Time (AP)	Fastest (A)	Time (A)	Abs Error	Meets Criterion?
4xGV100	Task 0	naive	13.40	15.80	SL_greedy	15.39	2.66%	✓
	Task 1	naive	12.13	15.46	SL_greedy	13.59	13.76%	✗
	Task 2	naive	12.40	15.81	random	14.74	7.26%	✓
	Task 3	SL_greedy	13.13	15.25	random	14.47	5.39%	✓
	Task 4	NL_greedy	12.26	15.54	L_greedy	14.00	11.00%	✗
	Task 10	naive	11.77	15.13	naive	15.13	0.00%	✓
	Task 11	naive	11.75	14.22	naive	14.22	0.00%	✓
	Task 12	naive	12.39	15.07	naive	15.07	0.00%	✓
	Task 13	NL_greedy	12.61	16.04	SL_greedy	14.96	7.22%	✓
	Task 14	naive	11.87	14.46	L_greedy	13.13	10.13%	✗
4xA100	Task 0	naive	8.68	10.62	naive	10.62	0.00%	✓
	Task 1	L_greedy	13.15	15.05	random	14.52	3.65%	✓
	Task 2	naive	9.81	11.43	naive	11.43	0.00%	✓
	Task 3	naive	11.31	14.04	random	13.85	1.37%	✓
	Task 4	naive	8.64	10.10	naive	10.10	0.00%	✓
	Task 10	random	8.69	10.85	SL_greedy	10.38	4.53%	✓
	Task 11	naive	9.18	11.40	naive	11.40	0.00%	✓
	Task 12	naive	9.26	11.76	S_greedy	11.54	1.91%	✓
	Task 13	naive	8.08	10.32	SL_greedy	9.79	5.41%	✓
	Task 14	naive	8.13	9.67	NL_greedy	9.18	5.34%	✓

We also do not include recently sophisticated sharders ([101, 128, 129]), although it is straightforward to integrate and test them in the future from an engineering point of view. We use six sharders listed in Table 5.3 for the experiment. In addition, we pick the 10 heavy tasks from the 20 tasks on each device generated in Section 5.5.2.1 and run them again with the batch size 4096 and all sharders except for size_lookup_greedy. This is to guarantee that embedding lookup latency dominates the E2E time and thus the sharding config is likely to make a difference.

Table 5.4 shows the actual and prediction time of using different sharding configs in selected tasks trained with platforms 4xGV100 and 4xA100. We set the success criterion to be either the performance model accurately selects the fastest config, *or* the absolute error between the *actual* time of the predicted fastest config and the time of the actual fastest config is less than 10%. The reason for this criterion is that we not only care about if the fastest config is selected, but also how close *the actual time of the predicted fastest config* is from the actual fastest time when the fastest config is *not* selected. This is because in practice it is tolerable to fail to select the fastest config as long as the actual time of the selected config is close enough to the actual fastest time.

We can see in the rightmost column that the prediction result of our performance model meets the criterion in 17 out of 20 (85%) tasks. In the rest 3 tasks, the absolute errors are only 13.76% at most. This proves that our performance model can aid the multi-GPU training optimization at the low time and compute costs. Since we have demonstrated that our performance model's prediction error is low both generally and in individual cases, we are confident that it can also perform well in unseen future cases.

5.6.2 Discussion

One of the biggest advantages of our performance model is its strong adaptability to new ML models. It is essentially an *execution simulator* built in the granularity of kernels and ops. To support any new ML models is incredibly straightforward: one only needs to add the performance models and overhead statistics of any new kernels and ops in these workloads that are yet missing. The library of supported kernels and ops grows as the system evolves, making it even easier to support new models later. More importantly, it is the execution of an ML model rather than its architecture that is important to the performance model. When the same ML model is trained with different strategies, our performance model can always capture its performance characteristic from the execution graph which is easy to obtain. This means our performance model has unlimited potential to be used on unseen future workloads and execution paradigms.

So far this work has a few limitations, including:

- Currently the prediction pipeline only supports ML workloads in FP32 precision. However, it can be seamlessly adapted for workloads in other precision types (FP16, INT8, etc) by preparing kernel performance models for all ops in these types.

- This work now only covers single-node multi-GPU performance modeling. To extend it for multi-node multi-GPU platforms, kernel performance models of *all-to-all* and *all-reduce* for the multi-node communication network must be prepared. Algorithm 4 should also be slightly modified to track multiple processes from all nodes. This will further adapt the performance prediction pipeline to the industrial environment for enormous ML workloads such as large language models (LLM).

- Some additional infrastructure features, such as the support of capturing the dynamic

tensor size and fused ops information in the execution graph, can increase the robustness of this work on various types of ML workloads (such as training NLP models with variable-input-length and no padding) and cooperate with modern ML compilation and optimization techniques.

We plan to extend our current code base to support these features.

5.7 Conclusion and Future Work

We extend a previous work of single-GPU performance model with the support for input-data-distribution-aware performance modeling of embedding lookup as a general enhancement, and performance modeling of communication collectives and E2E training time prediction with inter-rank and intra-rank synchronization to enable multi-GPU training performance modeling of ML workloads. We achieve high prediction accuracy on various types of ML workloads such as randomly generated DLRM and NLP models, and demonstrate the performance model's ability to speed up DLRM training through the use case of embedding table sharding config selection.

There are a series of future works to be derived from this work. Except for what has been mentioned in Section 5.6.2, the data-distribution-aware method we use to model the performance of FBGEMM embedding lookup can be generalized to other sparse ops such as SpMM and SpGEMM to handle the training and inference performance prediction of future sparsified neural networks.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

This dissertation discusses performance modeling and optimization for ML workloads. Through the example of convolutional layer fusion on multi-core CPUs, we first answer when and why fusion can be beneficial to CV model inference speedup, and demonstrate how to use the roofline model to identify fusion candidates as well as how to actually auto-tune and generate efficient code for the fused kernel that can beat vendor libraries and by-then state-of-the-art separate kernel generation by auto-tuning. Then, we propose a performance modeling pipeline as a complete system that can accurately predict the per-iteration training time for various ML models including CV, NLP, and RM on single- and multi-GPU platforms. This pipeline is highly usable and able to generate insights for future performance optimizations such as op fusion and sharding config improvement. Based on the existing work, we propose the following future research directions.

6.2 Universal Layer Fusion on Various Platforms

6.2.1 Overview

We propose to explore automatic candidate searches for multi-layer fusion. On the compute graph level, fusion on different devices still faces the same set of operators. Therefore, a similar design of the auto-tuning search space for the kernel schedule as we presented in Chapter 3 can be used *across various devices*. The difference in the search space on these devices, depending

on their architectures, is the number of splits per axis and a few axes reorder if necessary. To emphasize, the key is to split each final output axes (i.e., H, W, and C (assuming $N = 1$) for conv and dw-conv, M and N for GEMM), and share the same split of the channel axes (i.e., C for conv and dw-conv, N for GEMM) of the previous layer with the (long) reduce axis (i.e., RC for conv, K for GEMM, etc) of the current layer. In this way, together with other techniques like kernel auto-tuning configuration sharing, we can narrow the size of the optimization space and prevent it from growing exponentially as the number of layers increases. The rest of this section elaborates on a few of the important problems that we are interested in solving. One lower level than computational graphs, except for binding vendor library (micro)kernels to fused ops as we did in Chapter 3, the idea of Bring-Your-Own-Codegen [13] enables interfacing and binding custom proprietary kernels into the DL compiler. This facilitates layer fusion by leveraging the human effort of kernel optimization in a flexible way.

6.2.2 Graph-level Automatic Fusion Candidate Search

Currently, fusion candidates in end-to-end tests are hard-coded, which means we first select these candidates by comparing the fused kernel speed with that of separate kernels, and then we program it such that corresponding layers are fused in the inference pipeline. This is obviously not the optimal solution, and therefore, a graph-level automatic fusion search algorithm is needed.

We see previous work has proposed solutions to this problem. For example, Jangda et al. [42] recommend using a dynamic-programming (DP) based algorithm with cost function involving locality, tiling, prefetching, etc, to group the nodes to fuse in an image processing pipeline. Xiao et al. [123] also use DP with a depth-first-search branch-and-bound algorithm to generate the best fusion strategy for CNN inference on FPGAs, although here convolution workloads are not actually 'fused' but only pipelined. DP turns out to be suitable for this problem, yet it is usually time-consuming when searching for the optimal. In a non-fusion problem with a similar context, Liu et al. [63] successfully reduce search time by replacing DP CNN graph tuning for layout transformation with partitioned boolean quadratic programming (PBQP). These are all potential solutions to the problem we propose. The difference between it and the previous work is that in our case, each evaluation that decides if the current optimal should be updated in DP/PBQP and similar algorithms not only involves benchmarking the fusion candidates but also auto-tuning

them, which is much more time-consuming. This issue can be addressed by configuration sharing and/or performance modeling with roofline just like we present in Chapter 3.

6.2.3 Fusion in Training

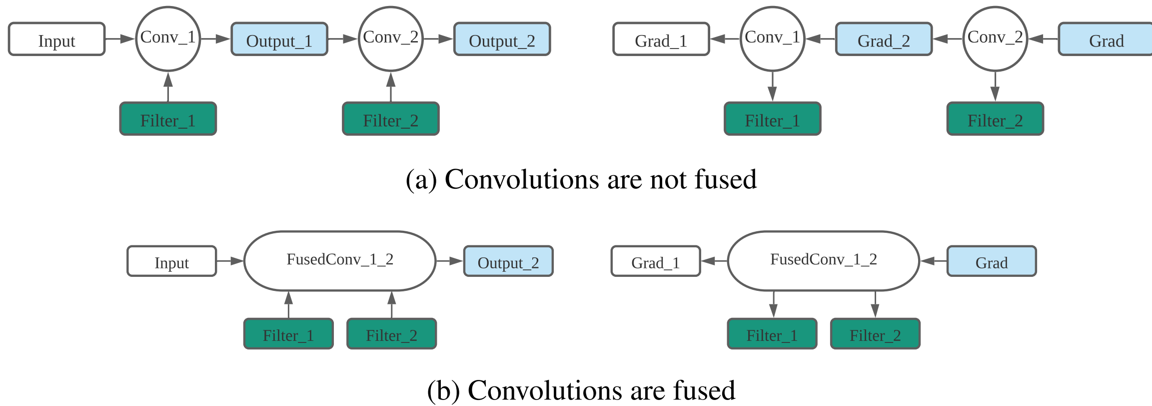
To our best knowledge, all layer-fusion-related previous work including ours is focused on inference rather than training. At the graph level, fusion in training is completely different than in inference. Figure 6.1 denotes the difference between forward and backward compute graphs with unfused and fused convolutions in training. To incorporate fusion with ML model training, as we foresee, the following work is required:

- Derivation of the mathematical expression for the fused backward op
- Implementation and auto-tuning of fused backward op
- Evaluation of the performance of this new backward op in the end-to-end test, and possible redesign of the op scheduling scheme of the framework

The difficulties of this problem are mainly on the engineering side. Among all the popular DL frameworks and compilers, TVM is inference-oriented and provides very little support on model training optimization; PyTorch and Tensorflow support optimization for training yet lacks a module for kernel auto-tuning and requires non-trivial work to improve the op scheduling scheme. However, a recent work by Rausch et al. [94] supports ML model training optimization with graph operations and optimizations based on a *data-centric* IR. It is possible that instead of the so far popular *op-centric* paradigm, fusion in training can be realized with data-centric optimization.

6.2.4 Tensor Compiler is the Future

So far our discussion is based on the fact that ML models are expressed as a set of connected ops in the lowest programming abstraction level that is exposed to users, and the low-level intermediate representation (IR) is Halide-like, i.e., separating *compute* and *schedule*, such as that in TVM. This has some limitations when it comes to deeply nested loops, e.g., fused ops, and the schedule becomes much more complicated for optimization and generalization across devices and compute platforms. Instead, in the cost of integration with auto-tuning, polyhedral-based



(a) Convolutions are not fused

(b) Convolutions are fused

Figure 6.1: Forward pass (left) and backward pass (right) of compute graphs of unfused and fused convolutions. Tensors in green are loaded from hard disks and tensors in blue are computed and stored in RAM, e.g., global memory. In an unfused graph, the intermediate output $Output_1$ is cached or checkpointed for the gradient computation of $Conv_1$. However, in a fused graph, $Output_1$ is computed on the fly and never materialized as a complete tensor. Therefore the backpropagation for op $FusedConv_1_2$ is completely different from that of unfused convolutions, the kernel implementation of which should be re-evaluated.

IR would greatly facilitate such optimization and generalization. For cases like fusion and tiling, various polyhedral transformations can be easily applied [55]. Examples include Tensor Processing Primitives (TPP) [25] which is built on top of the polyhedral-based compiler PlaidML. Other types of low-level IR also include MLIR [51] where hardware-specific optimizations are applied to them before being lowered to LLVM IR. We expect fusion will work better with compilers that adopt polyhedral IR, although the implementation detail is way beyond the scope of our discussion here. However, regardless of the type and category of compilers, we believe that the tensor compiler is the future of machine learning and HPC.

6.3 Universal Performance Modeling on All Compute Platforms

6.3.1 Overview

Performance models are powerful tools as we have demonstrated in Chapter 4 and 5. We envision the following potential trends of development in this area:

- The missing piece of multi-node multi-GPU performance modeling will be filled soon. This might trigger infrastructural work on the engineering side, and will significantly

benefit performance modeling universally (i.e., on all devices and all types of platforms, for all ML applications).

- Performance models can become a cost function in training to enable accuracy-performance co-optimization of ML models.

- We will see more performance modeling of sparse kernels/ops.

The following subsections will discuss these three bullet points respectively.

6.3.2 Multi-node Multi-GPU Performance Modeling for Industrial-scale ML workloads

The immediate next step of Chapter 5 is to extend the performance model to support multi-node multi-GPU performance modeling. We wish to address the problem of modeling the performance of communication collective running on a multi-node multi-GPU platform with *random* communication network topology. This would be an important problem to solve given GPUs are always allocated on-demand to train large ML models so that the machines and the ways they are connected might differ from time to time.

The building of a performance modeling pipeline for ML workloads on multi-node multi-GPU platforms might trigger a revolution of ML infrastructure on the engineering side. Currently, profilers such as Pytorch's Kineto [71] rely on vendor libraries such as CUPTI [83] to capture the low-level (e.g., kernel) performance behavior during model execution. The development and integration of such libraries are usually much slower compared to the emergence of ML models and their performance optimization, especially on new devices like customized AI chips. This means the early-stage optimization on these devices might be done somewhat blindly without sufficient guidance information. As people realize how important and powerful performance modeling is in ML optimization and research, the building of ML infrastructure, including developing these libraries and integrating them into existing frameworks, will be accelerated and significantly benefit performance modeling universally.

6.3.3 Assist Accuracy-Performance Co-optimization with Performance Modeling

Network architecture search (NAS) [136] is a methodology for searching for the best neural network architecture that fulfills both the accuracy and performance requirements. The main constraint of NAS is the incredibly high computational cost of model variant training. Therefore, performance predictors are widely used to speed up the NAS algorithms. As pointed out by White et al. [120], since these predictors usually predict performance in a coarse (i.e., model) granularity using ML approaches, they either require the full training of thousands of neural networks or are overwhelmed by the huge search space of NAS. NAS benchmark suites such as NAS-Benchmark-Suite [70] were developed to make the making of such performance predictors easier. However, this does not fundamentally solve the problem since a large-scale benchmark is still required.

We consider the execution-graph-based performance modeling framework we proposed to be a more appropriate solution to current and future NAS problems. Our framework has a finer granularity than previous works of NAS performance predictors (i.e., op/kernel vs. model). In this new NAS paradigm, only the execution graph of the model variants (i.e., op tensor size and dependency information), which is easy to obtain during training, is needed to predict the performance of model variants. Therefore, we completely bypass all kinds of prerequisite benchmarks and we are able to use the performance model as a cost function to guide the search for the best network architecture. This will be extremely helpful when it comes to NAS of large models such as DLRM or LLM, where a large-scale benchmark of thousands of variants is almost impossible.

6.3.4 Sparse Ops Performance Modeling Generalization

The advancement of the performance optimization of sparse computation (e.g., SpMM [23, 35, 36, 69, 125], SpConv [12, 23], and embedding lookup for recommendation models) coincides with the effort to make neural networks lighter-weight and more efficient by pruning and sparsification. Sparse inference and probably training of neural networks are becoming common practices. Unlike dense ops such as convolution and GEMM, the performance of sparse ops is input data distribution-dependent and format-dependent, which increases the difficulty of making accurate

predictions of it. We believe the performance modeling method we propose for embedding lookup in 5.4.3.1 is also the solution for other sparse ops. The key of our method is that with a low cost to extract input data distribution feature, it is able to make accurate *per-batch* latency prediction regardless of data and model sparsity as well as the configuration of sparse format. We look forward to seeing its application on sparse ML workloads training and inference.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 16, pages 265–283, 2016. doi: 10.5555/3026877.3026899.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and et al. Learning to optimize Halide with tree search and random programs. *ACM Transactions on Graphics*, 38(4):1–12, July 2019. ISSN 1557-7368. doi: 10.1145/3306346.3322967.

[3] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016. doi: 10.48550/arXiv.1605.02688.

[4] Muhammad Shoaib Bin Altaf and David A. Wood. LogCA: A high-level performance model for hardware accelerators. In *2017 ACM/IEEE 44th Annual Inter-*

national Symposium on Computer Architecture (ISCA), pages 375–388, 2017. doi: 10.1145/3079856.3080216.

[5] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, October 2016. ISBN 9781509035083. doi: 10.1109/micro.2016.7783725.

[6] Arjun Bansal, Will Constable, Scott Cyphers, Zach Dwiel, Scott Gray, Stewart Hall, Aravind Kalaiah, Urs Koster, Scott Leishman, Yinyin Liu, Jennifer Myers, Anthony Ndirago, Alex Park, Hanlin Tang, Anil Thomas, and Evren Tumer. Nervana Neon 1.5, June 2016.

[7] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, March 2003. ISSN 1532-4435. doi: 10.5555/944919.944966.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. doi: 10.5555/3495724.3495883.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. doi: 10.48550/arXiv.1512.01274.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association. ISBN 9781931971478. doi: 10.5555/3291168.3291211.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc. doi: 10.5555/3327144.3327258.

[12] Xuhao Chen. Escort: Efficient sparse convolutional neural networks on gpus. *CoRR*, abs/1802.10280, 2018. doi: 10.48550/arXiv.1802.10280.

[13] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. Bring your own codegen to deep learning compiler. *CoRR*, abs/2105.03215, 2021. doi: 10.48550/arXiv.2105.03215.

[14] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, pages 7–10, 2016. ISBN 9781450347952. doi: 10.1145/2988450.2988454.

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. doi: 10.48550/arXiv.1410.0759.

[16] François Chollet et al. Keras. `https://github.com/keras-team/keras`, 2015. Accessed: 06.15.2023.

[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022. doi: 10.48550/arXiv.2204.02311.

[18] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for YouTube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 191–198, September 2016. ISBN 9781450340359. doi: 10.1145/2959100.2959190.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019. doi: 10.18653/v1/n19-1423.

[20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=YicbFdNTTy`.

[21] Ehtsham Elahi and Ashok Chandrashekar. Learning representations of hierarchical slates in collaborative filtering. In *Fourteenth ACM Conference on Recommender Systems*, RecSys '20, pages 703–707, September 2020. ISBN 9781450375832. doi: 10.1145/3383313.3418484.

[22] Facebook. DLRM Github repo, June 2019. URL `https://github.com/facebookresearch/dlrm`.

[23] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, November 2020. ISBN 9781728199986. doi: 10.1109/SC41405.2020.00021.

[24] Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Sasikanth Avancha, Anand Venkat, Michael Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. Harnessing deep learning via a single building block. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 222–233, 2020. doi: 10.1109/IPDPS47924.2020.00032.

[25] Evangelos Georganas, Dhiraj Kalamkar, Sasikanth Avancha, Menachem Adelman, Cristina Anderson, Alexander Breuer, Jeremy Bruestle, Narendra Chaudhary, Abhisek Kundu, Denise Kutnick, Frank Laub, Vasimuddin Md, Sanchit Misra, Ramanarayan Mohanty, Hans Pabst, Barukh Ziv, and Alexander Heinecke. Tensor processing primitives: A programming abstraction for efficiency and portability in deep learning workloads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476206.

[26] Evangelos GeorgaZoph:2017:LTA, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov 2018. ISBN 9781538683842. doi: 10.1109/sc.2018.00069.

[27] Google. XLA Tech Blog. `https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html`, 2017. Accessed: 06.15.2023.

[28] Ananth Grama and Anshul Gupta. *Introduction to parallel computing*. Pearson, Noida, 2nd ed. edition, 2003.

[29] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: A factorization-machine based neural network for CTR prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 1725–1731. AAAI Press, 2017. ISBN 9780999241103. URL `https://www.ijcai.org/proceedings/2017/0239.pdf`.

[30] Juan Gómez-Luna, I-Jui Sung, Li-Wen Chang, José María González-Linares, Nicolás Guil, and Wen-Mei W. Hwu. In-place matrix transposition on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):776–788, 2016. doi: 10.1109/TPDS.2015.2412549.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[32] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 173–182, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. ISBN 9781450349130. doi: 10.1145/3038912.3052569.

[33] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 981–991, 2016. doi: 10.1109/SC.2016.83.

[34] Jonathan L. Herlocker, Joseph A. Konstan, and John Riedl. Explaining collaborative filtering recommendations. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 241–250, 2000. ISBN 1581132220. doi: 10.1145/358916.358995.

[35] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, page 66–79, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357852. doi: 10.1145/3208040.3208062.

[36] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 300–314, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295712.

[37] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolu-

tional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. doi: 10.48550/arXiv.1704.04861.

[38] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, 2017. URL `https://openreview.net/forum?id=S1xh5sYgx`.

[39] Intel. oneDNN Documentations. `https://github.com/oneapi-src/oneDNN`, 2016. Accessed: 06.15.2023.

[40] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. doi: 10.5555/3045118.3045167.

[41] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. Efficient execution of quantized deep learning models: A compiler approach. *CoRR*, abs/2006.10226, 2020. doi: 10.48550/arXiv.2006.10226.

[42] Abhinav Jangda and Uday Bondhugula. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 261–275, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349826. doi: 10.1145/3178487.3178507.

[43] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, pages 675–678, 2014. doi: 10.1145/2647868.2654889.

[44] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN computation with relaxed graph substitutions. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 27–39, 2019. URL `https://proceedings.mlsys.org/paper_files/paper/2019/file/4dd1a7279a8cfeea2660fbc34f02a2bc-Paper.pdf`.

[45] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris

Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017. ISSN 0163-5964. doi: 10.1145/3140659.3080246.

[46] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. Predicting the computational cost of deep learning models. In *2018 IEEE International Conference on Big Data*, BigData 2018, pages 3873–3882, December 2018. doi: 10.1109/BigData.2018.8622396.

[47] Daya Shanker Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. FBGEMM: enabling high-performance low-precision deep learning inference. *CoRR*, abs/2101.05615, 2021. doi: 10.48550/arXiv.2101.05615.

[48] Elias Konstantinidis and Yiannis Cotronis. A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing*, 107:37–56, 2017. ISSN 0743-7315. doi: 10.1016/j.jpdc.2017.04.002.

[49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017. ISSN 1557-7317. doi: 10.1145/3065386.

[50] H. T. Kung. Why systolic architectures? *Computer*, 15:37–46, 1982. doi: 10.1109/MC.1982.1653825.

[51] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020. doi: 10.48550/arXiv.2002.11054.

[52] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016. doi: 10.1109/CVPR.2016.435.

[53] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=qrwe7XHTmYb`.

[54] A. Li, S. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(01):94–110, Jan 2020. ISSN 1558-2183. doi: 10.1109/TPDS.2019.2928289.

[55] Mingzhe Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32:708–727, 2021. doi: 10.1109/TPDS.2020.3030548.

[56] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13 (12):3005–3018, aug 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415530.

[57] Shijian Li, Robert J. Walls, and Tian Guo. Characterizing and modeling distributed training with transient cloud GPU servers. In *2020 IEEE 40th International Conference on Distributed Computing Systems*, ICDCS 2020, pages 943–953, November 2020. doi: 10.1109/ICDCS47774.2020.00097.

[58] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. XDeepFM: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 1754–1763, 2018. ISBN 9781450355520. doi: 10.1145/3219819.3220023.

[59] Ying-Chiao Liao, Chuan-Chi Wang, Chia-Heng Tu, Ming-Chang Kao, Wen-Yew Liang, and Shih-Hao Hung. PerfNetRT: Platform-aware performance modeling for optimized deep neural networks. In *2020 International Computer Symposium*, ICS 2020, pages 153–158, December 2020. doi: 10.1109/ICS51289.2020.00039.

[60] Zhongyi Lin, Louis Feng, Ehsan K. Ardestani, Jaewon Lee, John Lundell, Changkyu Kim, Arun Kejariwal, and John D. Owens. Building a performance model for deep learning recommendation model training on GPUs. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics*, HiPC 2022, pages 48–58. IEEE, December 2022. doi: 10.1109/hipc56025.2022.00019.

[61] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single shot multibox detector. In *European Conference on Computer Vision*, pages 21–37. Springer, 2016. doi: 10.1007/978-3-319-46448-0_2.

[62] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach, 2020. URL `https://openreview.net/forum?id=SyxS0T4tvS`.

[63] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/liu-yizhi.

[64] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding capacity-driven scale-out neural recommendation inference. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 162–171, 2021. doi: 10.1109/ISPASS51385.2021.00033.

[65] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. DeLTA: GPU performance model for deep learning applications with in-depth memory system traffic analysis. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 293–303, March 2019. doi: 10.1109/ISPASS.2019.00041.

[66] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1): 105, 2019. doi: 10.11871/jfdc.issn.2096.742X.2019.01.011.

[67] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through FFTs. In *International Conference on Learning Representations*, April 2014. 2nd International Conference on Learning Representations, ICLR 2014 ; Conference date: 14-04-2014 Through 16-04-2014.

[68] Maximilian Beckers. Modern recommender systems. https://towardsdatascience.com/modern-recommender-systems-a0c727609aa8, 2021. Accessed: 06.15.2023.

[69] Atefeh Mehrabi, Donghyuk Lee, Niladrish Chatterjee, Daniel J. Sorin, Benjamin C. Lee, and Mike O'Connor. Learning sparse matrix row permutations for efficient SpMM on GPU architectures. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS 2021, pages 48–58. IEEE, March 2021. doi: 10.1109/ISPASS51385.2021.00016.

[70] Yash Mehta, Colin White, Arber Zela, Arjun Krishnakumar, Guri Zabergja, Shakiba Moradian, Mahmoud Safari, Kaicheng Yu, and Frank Hutter. NAS-bench-suite: NAS evaluation is (now) surprisingly easy. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=0DLwqQLmqV.

[71] Meta. Kineto profiler. https://github.com/pytorch/kineto, 2020. Accessed: 06.15.2023.

[72] Meta. PARAM benchmark, September 2020. URL https://github.com/facebookresearch/param.

[73] Meta. DLRM open-source datasets, December 2021. URL https://github.com/facebookresearch/dlrm_datasets.

[74] Tomás Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048. ISCA, 2010. doi: 10.21437/Interspeech.2010-343.

[75] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013. doi: 10.48550/arXiv.1301.3781.

[76] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 993–1011, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3533727.

[77] Ravi Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling Halide image processing pipelines. *ACM Transactions on Graphics*, 35:1–11, July 2016. doi: 10.1145/2897824.2925952.

[78] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019. doi: 10.48550/arXiv.1906.00091.

[79] Andrew Ng and Kian Katanforoosh. Stanford CS230 Section 5. https://cs230.stanford.edu/section/5/, 2021. Accessed: 06.15.2023.

[80] Xia Ning, Christian Desrosiers, and George Karypis. A comprehensive survey of neighborhood-based recommendation methods. In Francesco Ricci, Lior Rokach, and Bracha Shapira, editors, *Recommender Systems Handbook*, pages 37–76. Springer, 2015. doi: 10.1007/978-1-4899-7637-6_2.

[81] NVIDIA. cuBLAS deep learning performance matrix multiplication, July 2020. URL `https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html`.

[82] NVIDIA. cuML Documentations. `https://docs.rapids.ai/api/cuml/stable/`, 2020. Accessed: 06.15.2023.

[83] NVIDIA. CUPTI Documentations. `https://docs.nvidia.com/cuda/cupti/index.html`, 2022. Accessed: 06.15.2023.

[84] NVIDIA Corporation. CUDA CUBLAS library, 2011. `http://developer.nvidia.com/`.

[85] Yosuke Oyama, Akihiro Nomura, Ikuro Sato, Hiroki Nishimura, Yukimasa Tamatsu, and Satoshi Matsuoka. Predicting statistics of asynchronous sgd parameters for a large-scale distributed deep learning system on GPU supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 66–75, 2016. doi: 10.1109/BigData.2016.7840590.

[86] Adam Paszke, S. Gross, Francisco Massa, A. Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Z. Lin, N. Gimelshein, L. Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, NeurIPS 2019, 2019. doi: 10.5555/3454287.3455008.

[87] Ziqian Pei, Chensheng Li, Xiaowei Qin, Xiaohui Chen, and Guo Wei. Iteration time prediction for CNN in multi-GPU platform: Modeling and analysis. *IEEE Access*, 7: 64788–64797, 14 May 2019. doi: 10.1109/access.2019.2916550.

[88] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL `https://openreview.net/forum?id=SyVVJ85lg`.

[89] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 14 February 2019. URL `https://openai.com/blog/better-language-models/`.

[90] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, July 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185528.

[91] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, July 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185528.

[92] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530. ACM Press, June 2013. ISBN 9781450320146. doi: 10.1145/2491956.2462176.

[93] A. Rajagopal and C. Bouganis. perf4sight: A toolflow to model CNN training performance on edge GPUs. In *2021 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, pages 963–971, Los Alamitos, CA, USA, oct 2021. IEEE Computer Society. doi: 10.1109/ICCVW54120.2021.00112.

[94] Oliver Rausch, Tal Ben-Nun, Nikoli Dryden, Andrei Ivanov, Shigang Li, and Torsten Hoefler. A data-centric optimization framework for machine learning. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392815. doi: 10.1145/3524059.3532364.

[95] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016. doi: 10.1109/CVPR.2016.91.

[96] Steffen Rendle. Factorization machines. In *2010 IEEE International Conference on Data Mining*, pages 995–1000, 2010. doi: 10.1109/ICDM.2010.127.

[97] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing. ISBN 978-3-319-24574-4. doi: 10.1007/978-3-319-24574-4_28.

[98] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. doi: 10.1109/CVPR.2018.00474.

[99] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 2135, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2945397.

[100] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR*, abs/1802.05799, 2018. doi: 10.48550/arXiv.1802.05799.

[101] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. RecShard: Statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 344–358, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507777.

[102] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. doi: 10.48550/arXiv.1409.1556.

[103] Brent Smith and Greg Linden. Two decades of recommender systems at Amazon.com. *IEEE Internet Computing*, 21(3):12–18, May/June 2017. ISSN 1941-0131. doi: 10.1109/MIC.2017.72.

[104] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zheng, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing NLG 530b, A large-scale generative language model. *CoRR*, abs/2201.11990, 2022. doi: 10.48550/arXiv.2201.11990.

[105] Qingquan Song, Dehua Cheng, Hanning Zhou, Jiyan Yang, Yuandong Tian, and Xia Hu. Towards automated neural interaction discovery for click-through rate prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 945–955, 2020. ISBN 9781450379984. doi: 10.1145/3394486.3403137.

[106] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435. doi: 10.5555/2627435.2670313.

[107] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2016. doi: 10.1109/cvpr.2016.308.

[108] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019. doi: 10.48550/arXiv.1905.11946.

[109] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. MnasNet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018.

[110] Moshe Tennenholtz and Oren Kurland. Rethinking search engines and recommendation systems: A game theoretic perspective. *Commun. ACM*, 62(12):66–75, November 2019. ISSN 0001-0782. doi: 10.1145/3340922.

[111] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023.

[112] Andrew Tulloch. Batch embedding lookup GPU kernel and more, May 2020. URL `https://github.com/ajtulloch/sparse-ads-baselines`.

[113] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018. doi: 10.48550/arXiv.1802.04730.

[114] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., December 2017. URL `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[115] Jyothi Vedurada, Arjun Suresh, Aravind Sukumaran Rajam, Jinsung Kim, Changwan Hong, Ajay Panyala, Sriram Krishnamoorthy, V. Krishna Nandivada, Rohit Kumar Srivastava, and P. Sadayappan. TTLG - an efficient tensor transposition library for GPUs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 578–588, 2018. doi: 10.1109/IPDPS.2018.00067.

[116] Strother H. Walker and David B. Duncan. Estimation of the probability of an event as a function of several independent variables. *Biometrika*, 54(1–2):167–179, June 1967. ISSN 0006-3444. doi: 10.1093/biomet/54.1-2.167. URL `https://doi.org/10.1093/biomet/54.1-2.167`.

[117] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia. Characterizing deep learning training workloads on alibaba-pai. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 189–202, Los Alamitos, CA, USA, nov 2019. IEEE Computer Society. doi: 10.1109/IISWC47752.2019.9042047.

[118] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*, ADKDD'17, 2017. ISBN 9781450351942. doi: 10.1145/3124749.3124754.

[119] Xueying Wang, Guangli Li, Xiao Dong, Jiansong Li, Lei Liu, and Xiaobing Feng. Accelerating deep learning inference with cross-layer data reuse on GPUs. In *Euro-Par*

2020: Parallel Processing, pages 219–233. Springer International Publishing, 2020. ISBN 9783030576752. doi: 10.1007/978-3-030-57675-2_14.

[120] Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 28454–28469, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/ef575e8837d065a1683c022d2077d342-Abstract.html.

[121] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785.

[122] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, Oct 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6.

[123] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349277. doi: 10.1145/3061639.3062244.

[124] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1355–1364, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336642. doi: 10.1145/2783258.2783270.

[125] Carl Yang, Aydın Buluç, and John D. Owens. Design principles for sparse matrix multiplication on the GPU. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Proceedings of the 24th International European Conference on Parallel and Distributed Computing*, pages 672–687, August 2018. doi: 10.1007/978-3-319-96983-1_48.

[126] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. XLNet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. doi: 10.5555/3454287.3454804.

[127] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 503–521. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL `https://www.usenix.org/conference/atc21/presentation/yu`.

[128] Daochen Zha, Louis Feng, Bhargav Bhushanam, Dhruv Choudhary, Jade Nie, Yuandong Tian, Jay Chae, Yinbin Ma, Arun Kejariwal, and Xia Hu. AutoShard: Automated embedding table sharding for recommender systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 4461–4471, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393850. doi: 10.1145/3534678.3539034.

[129] Daochen Zha, Louis Feng, Qiaoyu Tan, Zirui Liu, Kwei-Herng Lai, Bhargav Bhushanam, Yuandong Tian, Arun Kejariwal, and Xia Hu. DreamShard: Generalizable embedding table placement for recommender systems. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL `https://openreview.net/forum?id=_atSgd9Np52`.

[130] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models, 2022.

[131] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. AIBox: CTR prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, pages 319–328, 2019. ISBN 9781450369763. doi: 10.1145/3357384.3358045.

[132] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical GPU parameter server for massive scale deep learning ads systems. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020*, MLSys 2020. mlsys.org, March 2020. URL `https://proceedings.mlsys.org/book/315.pdf`.

[133] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 2020, pages 863–879. USENIX Association, November 2020.

[134] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 1059–1068, 2018. ISBN 9781450355520. doi: 10.1145/3219819.3219823.

[135] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 337–352. USENIX Association, 2020. URL `https://www.usenix.org/system/files/atc20-zhu-hongyu.pdf`.

[136] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016. doi: 10.48550/arXiv.1611.01578.