

The Pennsylvania State University
The J. Jeffrey and Ann Marie Fox Graduate School

**ARCHITECTURAL OPTIMIZATIONS FOR BOOSTING THE
PERFORMANCE OF RECOMMENDATION SYSTEMS**

A Dissertation in
Computer Science and Engineering
by
Rishabh Jain

© 2025 Rishabh Jain

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2025

The dissertation of Rishabh Jain was reviewed and approved by the following:

Chitaranjan Das
Distinguished Professor, Computer Science and Engineering
Chair of Committee

Mahmut T. Kandemir
Distinguished Professor, Computer Science and Engineering,
Program Head
Dissertation Advisor

Anand Sivasubramaniam
Distinguished Professor, Computer Science and Engineering

Amulya Yadav
Associate Professor, College of Information Sciences and Technology

John Kalamatianos
Fellow Design Engineer,
Research and Advanced Development (RAD), AMD Inc.
Special Member

Onur Kayiran
SMTS Silicon Design Engineer,
Research and Advanced Development (RAD), AMD Inc.
Special Member

Abstract

Personalized recommendation has become one of the most pervasive AI applications on the internet, powering user experiences across e-commerce, social media, and entertainment platforms. Deep Learning Recommendation Models (DLRMs) form the algorithmic backbone of these systems, combining dense and sparse features to deliver accurate personalization at scale. These models are deployed massively across datacenters using CPUs and GPUs, accounting for a major share of all AI inference cycles and driving core revenue for hyperscalers like Meta and Google. However, the rapid growth in model complexity, with trillions of parameters and thousands of embedding tables, has amplified the compute, memory, and latency challenges of inference. Since recommendation is a user-interactive application, inference must meet stringent service-level agreement (SLA) requirements, motivating the need for architectures that can deliver high throughput and low latency.

The objective of this dissertation is to improve the performance of DNN-based recommendation systems using architectural optimizations across CPUs, GPUs, and heterogeneous CPU-GPU platforms. Specifically, the dissertation targets DLRM inference. On CPUs, our in-depth performance breakdown study across multiple DLRM models and datasets, reveals that the embedding stage dominates end-to-end inference time due to irregular memory accesses. To capture the diversity in access patterns, production traces are categorized into datasets of varying hotness, and their behavior is analyzed through cache profiling and reuse distance modeling. Based on these insights, the dissertation proposes application-specific software prefetching to hide long access latencies and application-aware HyperThreading to overlap computation and memory operations, leading to improved core utilization on server-grade CPUs.

As DLRMs continue to scale in table count, pooling operations, and embedding dimension, the work further explores alternative approaches for parallelization of the embedding stage, introducing Table Threading (TT) and Hierarchical Threading (HT) to parallelize individual batches across cores. These techniques result in significant performance benefits. However, even with these approaches, the performance scaling remains imperfect for heterogeneous datasets, where embedding tables vary widely in size, reuse, and pooling operations. To overcome this imbalance, the dissertation develops a load- and memory-level-parallelism (MLP)-aware scheduling framework, which analytically models each table’s workload using reuse-distance-based metrics. This enables intelligent categorization and scheduling of tables across threads. By incorporating

table reordering, core grouping, strategic co-location, and lightweight work stealing, the proposed design achieves lower-latency, improved utilization, and enhanced scalability on modern many-core CPUs.

On GPUs, this dissertation examines the architectural inefficiencies of DLRM inference under growing compute and memory-bandwidth demands. While GPUs offer massive parallelism and high-bandwidth memory, detailed microarchitectural analysis of NVIDIA A100 and H100 shows that the embedding stage continues to be the primary bottleneck due to low occupancy and long memory stalls. To mitigate these issues, the work proposes compiler-guided optimizations, software prefetching, and L2 cache pinning techniques that improve warp utilization and hide latency through purely software-based means, thus boosting overall embedding stage performance. Building on these insights, the dissertation introduces HetT-HetD, a heterogeneous CPU–GPU execution framework that challenges the one-device-fits-all paradigm by leveraging the inherent device and table diversity in modern DLRMs. By mapping high-reuse, large-capacity tables to CPUs and low-reuse, bandwidth-intensive tables to GPUs, and coordinating them via asynchronous data movement and heuristic-driven scheduling, HetT-HetD leverages both CPU and GPU performantly.

Overall, this dissertation advances the architectural understanding and system-level optimization of large-scale DLRM inference, with a central emphasis on achieving low-latency and high-efficiency recommendation serving. All proposed techniques are realized entirely in software and require no modifications to existing hardware or publicly distributed models, making them directly deployable in current datacenter environments. By systematically improving CPU, GPU, and heterogeneous CPU–GPU execution, this work establishes a new performance baseline for future research exploring hardware and software co-designs aimed at further accelerating deep learning–based recommendation systems.

Table of Contents

List of Figures	ix
List of Tables	xvii
Acknowledgments	xix
Chapter 1	
Introduction	1
1.1 Background	2
1.1.1 DLRM Inference	2
1.1.2 Memory Accesses in the Embedding Stage on CPU	3
1.1.3 Memory Accesses in the Embedding Stage on GPU	5
1.1.4 Key Challenges	7
1.1.5 Dissertation Contributions:	7
Chapter 2	
Optimizing CPU Performance for Recommendation Systems At-Scale	10
2.1 Introduction	10
2.2 Background and Motivation	13
2.2.1 DLRM Inference	13
2.2.2 Memory Accesses in the Embedding Stage	14
2.2.3 Related Work	16
2.2.4 Our Novelty	17
2.3 Demystifying the memory access behavior and Multi-core Scaling	18
2.3.1 Theoretical Analysis	19
2.3.1.1 Access count study for different input loads	20
2.3.1.2 Reuse distance study	20
2.3.2 Multi-core Scaling and Bandwidth study	22
2.3.3 Key Takeaways	23
2.4 Proposed Optimizations for Improving the Efficiency of CPU Pipeline	24
2.4.1 Limitations of Existing On-Chip Hardware and Compiler Techniques	25
2.4.2 Application-Specific Software Prefetching	26
2.4.3 Application-aware HyperThreading	28
2.4.4 Synergy of Prefetching and Overlapping	29

2.5	Methodology	30
2.6	Performance Evaluation	31
2.6.1	Improvement in the Embedding stage	32
2.6.2	End-to-end Improvement	32
2.6.2.1	Embedding Heavy Models	32
2.6.2.2	Mixed Models	34
2.6.3	Empirical Characterization	34
2.6.4	Evaluating Across Different CPU Platforms	35
2.6.5	Tail Latency Evaluation	36
2.7	Conclusions	36

Chapter 3

	Parallelization of Embedding Stage on CPUs	38
3.1	Introduction	38
3.2	Background	42
3.2.1	Understanding Embedding Bag Operator	42
3.2.2	Target Architectures	43
3.2.2.1	AMD “Genoa”	43
3.2.2.2	AMD “Genoa-X” – (“Genoa” + V-Cache™)	46
3.2.3	Settings for Embeddings on “Genoa” and “Genoa-X”	46
3.3	Load Imbalance in Embedding Bag	48
3.4	Parallelization Strategies for Embedding Bag Operator	50
3.4.1	Batch Threading (BT)	52
3.4.2	Table Threading (TT)	52
3.4.3	Hierarchical Threading (HT)	53
3.5	Empirical Evaluation	53
3.5.1	Methodology	54
3.5.2	Results	57
3.6	Related Work	63
3.7	Concluding Remarks	65

Chapter 4

	Load and MLP-Aware Thread Orchestration for Recommendation Systems Inference on CPUs	66
4.1	Introduction	66
4.2	Embedding Stage Parallelization Inefficiencies	69
4.2.1	Overview of Embedding Stage Computation and Dataset	70
4.2.2	Limitations of Prior Parallelization Approaches	71
4.2.3	Unboxing the Design of Hierarchical Threading	73
4.2.4	Modeling and Characterizing Table Workloads	74
4.2.5	Per Table Multi-Core Scaling Behavior	78
4.2.6	Microbenchmarking Memory Level Parallelism	79
4.2.7	Key Architectural Insights for CPU Multi-Core Scaling inefficiencies	81
4.3	Load- and MLP-Aware Thread Scheduling	82

4.4	Methodology	84
4.5	Evaluation	87
4.5.1	Main Results	87
4.5.2	Sensitivity Studies	88
4.6	Discussion and Future Work	91
4.7	Related Works	91
4.8	Conclusions	92

Chapter 5

Pushing the Performance Envelope of DNN-based Recommendation Systems Inference on GPUs		93
5.1	Introduction	93
5.2	Background	96
5.2.1	DLRM Inference Using GPUs	96
5.2.2	Key Properties of GPU Microarchitecture	97
5.2.3	Related Works	99
5.3	Dissecting Embedding Bag execution on a GPU	100
5.3.1	Parallel Implementation of the Embedding Bag Operator	100
5.3.2	Quantitative Study of the Memory Access Patterns	102
5.3.3	Architectural Implications of Embedding Bag on a GPU	103
5.3.4	GPU-specific Key Microarchitectural Insights	107
5.4	Optimizations to Improve the Warp Scheduler Issue Slot Utilization	107
5.4.1	Limitations of Off-the-Shelf Solutions	108
5.4.2	Application-Driven Data Prefetching	108
5.4.3	Application-Aware L2 pinning	111
5.4.4	Synergy between Prefetching and Pinning	114
5.5	Methodology	114
5.6	Evaluation	115
5.6.1	Key Results	116
5.6.1.1	Boost in Embedding Stage Performance	116
5.6.1.2	Boost in End-to-End DLRM Performance	116
5.6.1.3	Microarchitectural Justifications	117
5.6.2	Sensitivity Analysis	118
5.6.2.1	Winning Prefetching Scheme	118
5.6.2.2	Improvement over Baseline PyTorch without OptMT	119
5.6.2.3	Heterogeneous Table Mixing	120
5.6.2.4	Evaluation on H100 NVL GPU	121
5.7	Discussion	122
5.8	Conclusion	123

Chapter 6

HetT-HetD: Embracing Heterogeneity in Embedding Tables and Devices for Performant Recommendation Systems		125
6.1	Introduction and Motivation	125

6.2	Preliminary design	127
6.2.1	Key Design Decisions	128
6.3	Optimization Design	128
6.3.1	Mechanism:	129
6.3.2	Policy:	132
6.4	Evaluation	136
6.5	Conclusion	137

Chapter 7

	Conclusions and Future Work	138
7.1	Conclusions	138
7.2	Broader Insights	139
7.3	Future Work	140
7.3.1	DLRM Training	140
7.3.2	Multi-device load balancing	140
7.3.3	Analyzing accuracy vs. performance tradeoffs	140
7.3.4	Going Beyond DLRMs	141

	Bibliography	142
--	---------------------	------------

List of Figures

1.1	Execution time breakdown of different DLRMs.	1
1.2	DLRM Architecture and Inference serving pipeline	2
1.3	Illustration on the working of the embedding_bag operator for one sample (assume BS means batch size, embedding_dim = 128, and lookups per sample as 120). Three levels of indirection are involved.	4
1.4	Parallel implementation of embedding stage by work partitioning across CUDA threads. Here, 1000s of CUDA threads independently work on one output matrix element.	6
2.1	Execution time breakdown of different DLRMs.	11
2.2	A generic DLRM architecture.	14
2.3	Illustration on the working of the embedding_bag operator for one sample (assume BS means batch size, embedding_dim = 128, and lookups per sample as 120). Three levels of indirection are involved.	15
2.4	RM2_1 embedding stage performance comparison across datasets. (a) Batch latency (b) average load latency, and cache hit rates for L1D, L2, L3.	19
2.5	Hot embedding access count (sorted) in 3 datasets.	20
2.6	Our model for capturing the reuse distances, and for indicating cache hit rates and cold misses.	22
2.7	Reuse distance study for different datasets in rm2_1 model for 24 cores & batch size = 64.	23
2.8	Multi-core scalability: (a) Execution time (b) Memory bandwidth used.	24

2.9	Software prefetching timeline: Gather[G] (the demand loads) follow the look-ahead loads[P] at an ideal prefetch distance of four.	27
2.10	(a) Comparing state-of-the-art compiler-inserted prefetching with baseline for the RM2_1 model, (b) Impact of prefetch distance on the performance of 24 cores, (c) Impact of prefetch amount on the L1D\$ hit rate and average load latency.	27
2.11	Overlapping compute and memory access via hyperthreading.	29
2.12	Comparison of performance of embedding-only stage for embedding heavy models for different datasets: (a) single and (b) many cores. . . .	32
2.13	End-to-end performance gain on embedding heavy models for different datasets.: (a) single and (b) many cores.	33
2.14	End-to-end performance gain on mix-model RM_1 for different datasets.	33
2.15	L1D\$ hit rate and average load latency for RM2_1 model (embedding-only) on Low-Hot dataset.	34
2.16	Speedup comparison across different CPUs for (a) single-core (b) multi-core	35
2.17	Tail (P95) latency comparison across designs by varying the arrival time for models: (a) RM2_1 (b) RM1. SLA latency target is marked in red.	36
3.1	DLRM architecture. MLPs are compute-bound and are typically parallelized using data parallelism. On the other hand, embedding tables are memory bandwidth- and memory capacity-bound and are parallelized using model parallelism.	39
3.2	Achieved bandwidth (data processing rate), normalized to peak DDR bandwidth, when using Batch Threading (BT) with different table sizes (x-axis) on a 4th Gen EPYC™ processor. The achieved bandwidth is only a fraction of the theoretical LLC bandwidth for all table sizes, and is lower than the peak DDR bandwidth for some table sizes.	40
3.3	Scalability of BT, with heterogeneous (varying table sizes) datasets, on a 4th Gen EPYC™ processor. Each point on the x-axis corresponds to a different dataset (with a different table size) and each bar corresponds to a core count. Note that performance tends to saturate beyond 16 cores.	41

3.4	Workflow of the embedding bag operator. For each table, a batch of queries is processed. The operator initially performs index lookups for each query item, followed by mapping these indices to the embedding tables and collecting the corresponding embedding vectors. Finally, a pooling operation, such as “sum”, is applied to calculate an output vector for the query item.	43
3.5	A high-level view of “Genoa”. Each group of 8 cores with an L3 cache defines a chiplet (CCD) [1].	44
3.6	A high-level view of “Genoa-X”. V-Cache [™] effectively extends LLC capacity for the cores in each CCD. Note that the LLC with V-Cache [™] remains partitioned across core complexes [1].	45
3.7	Number of tables in various row size bins.	49
3.8	Pooling factors for different tables.	50
3.9	Access pattern behavior for different tables.	50
3.10	Embedding table latencies for different tables.	51
3.11	Batch Threading.	51
3.12	Table Threading.	51
3.13	Hierarchical Threading.	51
3.14	Sorted Histogram of Zipf’s distribution for different values of α . Each distribution randomly selects 10000 values from 100 entries.	53
3.15	AMD Unified AI Software stack.	55
3.16	HT and TT performance relative to BT for different table sizes using Zipf’s distribution on “Genoa”.	56
3.17	HT and TT performance relative to BT for different table sizes for Zipf’s distribution on “Genoa-X”.	57
3.18	Batch size sensitivity for all threading strategies on “Genoa”. Numbers are normalized to the Batch size of 16K.	58

3.19	Performance of various parallelization strategies on “Genoa-X” across different number of tables and table sizes. All numbers are normalized to the 192 number of tables configuration per table size.	58
3.20	HT performance when each thread is scheduled on 4 cores (4C) vs 8 cores (8C) for “Genoa” and “Genoa-X”. All numbers normalized to the 4C case per table size. Number of tables is 192.	59
3.21	Parallelization scheme performance with α ($\alpha=1.15$ vs $\alpha=1.09$) for different table sizes.	60
3.22	Realized bandwidth relative to DDR bandwidth for “Genoa”.	61
3.23	Realized bandwidth relative to DDR bandwidth for “Genoa-X”.	62
3.24	Performance with heterogeneous data set on "Genoa" (129 tables, with table sizes varying from 12.5KB to 610MB). All numbers are normalized to the single core case.	63
3.25	Embedding table latency for different tables.	63
4.1	Performance scaling of batch threading (BT), table threading (TT), and hierarchical threading (HT) over homogeneous random and heterogeneous datasets. Reported speedup is over single core performance. Despite the significant speedup of HT over BT and TT, absolute speedup achieved over heterogeneous tables is only $13\times$ (highlighted) and much lower than the ideal speedup of $96\times$	67
4.2	DLRM architecture consists of four stages: bottom multi-layer perceptron, embedding, feature interaction, and top multi-layer perceptron.	67
4.3	(a) Number of rows in various row size bins, (b) access pattern behavior for different tables, and (c) pooling factors for different tables.	69
4.4	Embedding table latencies for different tables.	70
4.5	State of the art parallelization techniques.	71
4.6	HT is implemented with the PyTorch thread pool. Each CCD is associated with a task queue and tables are distributed across task queues using round robin injection.	73

4.7	Reuse distance based analytical model to calculate the average memory access time of the loads accessed in an embedding table.	75
4.8	(a) AMAT across tables derived using our analytical model. (b) High correlation is observed between TL and measured execution time. (c) When comparing the decreasing trend of TL with pooling factor, we note that pooling factor can incorrectly represent work per table.	76
4.9	Using different TL metric thresholds, the embedding tables can be categorized as heavy, medium, and light-weight.	77
4.10	Deviation study of embedding stage execution time across 120 batches. Minimal deviation is observed.	77
4.11	Characterization of intra-CCD multi-core scaling of embedding tables. (a) Work of each table scales differently with the number of cores, and only a few tables benefit from 8 or 6 cores. (b) Unlike a high reuse table whose works benefits from using up to 8 cores, the work of a low reuse table stops scaling after 4 cores.	78
4.12	Speedup using 2, 4, and 8 cores over tables sorted in descending order of work per table (TL).	79
4.13	Eight thread execution using multiple CCDs with TT over different memory reuse patterns.	80
4.14	Weighted memory-level-parallelism (WMLP) metric models the amount of MLP demand of each table.	81
4.15	Balance is demonstrated for two-core grouping. Tables are injected into the task queues in decreasing WMLP order. Two task queues are used to separately schedule the work of heavy (HWMLP) and light (LWMLP) tables based on a cutoff point. At any point in time, each CCD is assigned only one HWMLP table and one or more LWMLP tables. Work stealing improves load balancing by allowing a CCD to seek work from a secondary queue after a primary queue gets empty.	82
4.16	Speedup of Balance with two and four-core grouping over HT using various core counts and batch size of 1024.	84
4.17	Speedup scaling for Balance with two and four-core grouping over single core Balance and batch size of 1024.	85

4.18	Speedup of Balance with four-core grouping against HT over various core counts and batch sizes (BS).	86
4.19	Speedup of Balance with two-core grouping against HT over various core counts and batch sizes (BS).	86
4.20	Four-core grouping speedup over a single core.	88
4.21	Two-core grouping speedup over a single core.	88
4.22	Impact of cutoff point and work stealing.	89
4.23	Speedup scaling comparison for a large model with 400 tables.	90
4.24	Memory bandwidth consumption of HT and Balance schedulers with 48 cores and 2048 batch size.	90
5.1	Shown is the degradation in inference performance as hotness lowers (working footprint decreases) from left to right. The numbers inside the bars indicate the embedding stage contributions. Here, OptMT provides higher WLP which enhances performance over off-the-shelf PyTorch (base). Yet, a significant gap continues to exist compared to the fastest loads (one item case). We cite this as the research <i>gap</i>	94
5.2	A schematic of a DLRM architecture. The continuous features (e.g., age, location) are processed by Bottom MLP, and categorical features (e.g., movie genre, item ID) by the Embedding Stage. Their outputs are combined in the Feature Interaction Stage, and then fed into the Top MLP, which predicts top-k items with highest Click Through Rate (CTR).	97
5.3	Simplified Nvidia A100 GPU organization.	98
5.4	Parallel implementation of embedding stage by work partitioning across CUDA threads. Here, 1000s of CUDA threads independently work on one output matrix element.	101
5.5	Coverage study for different memory access patterns: it shows the % of total accesses (y axis) that are covered by the % of unique accesses (x axis).	102
5.6	Synthetically varying the number of registers allocated to improve WLP. The primary y-axis is speedup over off-the-shelf PyTorch, and the secondary y-axis is the register spilling penalty based on extra local memory loads (in millions). OptMT refers to the highest speedup at 40 warps.	106

5.7	Buffer locations used for various prefetching schemes. A CUDA thread is shown executing SMPF, a batch of 10 prefetches(P) are launched every 10th iteration, and a reduce(R) operation every iteration.	109
5.8	Prefetching implementations: (a) RPF (b) SMPF	110
5.9	Performance impact of prefetch distance in SMPF.	112
5.10	High-level design of L2P	112
5.11	Detailed study of L2 pinning over various pooling factors. Speedup is reported over off-the-shelf PyTorch.	113
5.12	Embedding-only improvement in latency of the proposed techniques with OptMT over off-the-shelf PyTorch.	116
5.13	End-to-end improvement in latency of the proposed techniques with OptMT over off-the-shelf PyTorch.	117
5.14	Embedding stage contribution in the end-to-end latency.	117
5.15	Comparison of all prefetching techniques with OptMT over off-the-shelf PyTorch.	119
5.16	Comparison of techniques for off-the-shelf Pytorch.	119
5.17	Embedding-only improvement of the proposed techniques with OptMT over off-the-shelf PyTorch and heterogeneous tables.	121
5.18	On H100 NVL GPU, the number of registers allocated is varied to find optimal WLP. The primary y-axis is speedup over off-the-shelf PyTorch and the secondary y-axis is the register spilling penalty based on extra local memory loads (in millions). OptMT on H100 refers to the highest speedup at 32 warps.	122
5.19	Comparison of Embedding-only improvement in latency of the integrated scheme over off-the-shelf PyTorch for H100 NVL and A100 GPU.	122
6.1	GPU speedup over CPU for embedding stage using {H100 GPU, Zen4 64 CPU} and {A100 GPU, Zen3 64 CPU}.	126
6.2	GPU speedup over CPU for non embedding stages using H100 GPU and Zen4 64 core CPU.	126

6.3	Concurrent execution design for embedding stage on CPU and GPU . . .	133
6.4	End-to-end dataflow for one batch of DLRM inference where blue represents tasks mapped onto CPU and orange represents tasks mapped onto GPU. A portion of embedding stage is mapped onto CPU. GPU first executes the bottom-MLP (BMLP) stage, followed by the remaining portion of embedding tables. Feature interaction (FI) stage waits for the output for embedding portions from both CPU and GPU and is followed by the top-MLP (TMLP) stage.	133
6.5	Concurrent execution design for end to end DLRM on CPU and GPU . .	134
6.6	Example policy for achieving an optimal mapping of tables on CPU and GPU	136
6.7	End to end speedup on homogeneous datasets.	137
6.8	End to end speedup on heterogeneous datasets.	137

List of Tables

2.1	Different model characteristics from [2].	14
2.2	Model architecture parameters in our study.	18
2.3	CPU configuration parameters.	30
2.4	Batch execution time (ms) of embedding-only stage in multi-core for different models and datasets.	31
3.1	"Genoa" Configuration [3].	46
3.2	Homogeneous Zipf's distribution based on microbenchmarking coverage set.	54
4.1	TL can predict the range of performance scaling for any given table. . . .	79
4.2	96-core AMD 4th Generation EPYC Processor Configuration [3].	85
5.1	Access Latencies for various levels of the A100 GPU memory hierarchy based on [4].	98
5.2	Cache capacities for server-grade GPUs.	99
5.3	Unique access % in each dataset.	103
5.4	Microarchitectural characterization of Base PyTorch on various datasets. With 74 registers allocated to each CUDA thread, the WLP is limited due to the register pressure.	104

5.5	Microarchitectural characterization of Optimal-Multithreading (OptMT) PyTorch on various datasets. With 42 registers allocated to each CUDA thread, the register pressure lowers and the WLP significantly improves. Still, a performance gap exists between the fastest and slowest loads. . .	106
5.6	System specifications used for evaluation	114
5.7	Heterogeneous Mixture of Model configuration	115
5.8	Microarchitectural details for RPF+OptMT	118
5.9	Microarchitectural details for RPF+L2P+OptMT	118

Acknowledgments

Pursuing a Ph.D. has been one of the most transformative experiences of my life. Starting my Ph.D. during the challenging times of the COVID-19 pandemic brought its own set of obstacles, both academic and personal. Looking back, I feel truly fortunate to have successfully completed this journey, and I am deeply grateful to everyone who has supported and guided me along the way.

I would like to convey my sincere gratitude to a long list of people who have played a significant role in shaping my academic journey throughout this Ph.D. program. First and foremost, my heartfelt thanks goes to my advisor, Prof. Chita Das, for the immense support, numerous opportunities, freedom, guidance, and valuable leanings. In particular, I deeply value the lessons on maintaining a bird's-eye view, following a top-down approach, focusing on the right problem at the right time, and leveraging past efforts instead of starting from scratch. His constant motivation to strive for highest standards, take on challenging problems, continue to persevere and work diligently until the deadline, and aim for top-tier conferences has been truly inspiring. I am also grateful for the connections he helped me build with various experts and alumni. I genuinely cherish the positive lab environment he created and still fondly miss our coffee-time discussions and Indian snacks at HPCL. Thank You, Sir. I would also like to thank my dissertation co-advisor, Prof. Mahmut Kandemir, for brainstorming discussions, guidance in shaping my research ideas, and for helping me connect them with prior compiler research works. I fondly recall our collaborations on various proposal and grant submissions, where his advice on emphasizing vision over implementation details left a lasting impression on me. I am also thankful to him for teaching me CMPEN 431, CSE 521, and CSE 531, which provided a strong foundation for my research. Thank You, Sir.

Next, I would like to thank Prof. Anand Sivasubramaniam for serving on my thesis committee and for collaborating with me on multiple projects. I deeply appreciate his emphasis on understanding why a problem is important, motivating research ideas both fundamentally and intuitively, and thinking from multiple angles to a problem. I have learned from him the art of storytelling—how to create a smooth and logical flow between ideas—and the importance of positioning one's work to highlight unique and interesting contributions. I am grateful for the intellectually stimulating conversations we have had, both about research and broader topics in life. I also thank him for inquisitively teaching CMPSC 473, CSE 511, CSE 532, and CSE 598 which greatly strengthened my foundations. Thank you, Sir.

Next, I would like to thank Prof. Amulya Yadav, John Kalamatianos, and Onur Kayiran for serving on my PhD committee. I am especially thankful to John and Onur for mentoring me during my multiple internships at AMD Research (now AMD Research and Advanced Development). I deeply appreciate the freedom and opportunity they provided to explore my dissertation topic while aligning it with the team’s goals on multithreaded workloads with memory-bound bottlenecks. I am also grateful for their guidance in preparing multiple invention disclosures (IDFs) and research papers, and for the invaluable lessons on the nature and impact of industrial research, which broadened my perspective beyond academia. The opportunities to network within the company were equally enriching and instrumental in my professional growth.

Next, I would like to thank my research collaborators — Prof. Adwait Jog, Prof. Kiwan Maeng, Meena Arunachalam, Krishnakumar Nair, Vishwas Kalagi, Scott Cheng, and Vivek Bhasi — for their valuable discussions and feedback throughout my research journey. Special thanks to Teyuh Chou and Varun Agarwal for their mentorship and for always being my go-to people at AMD Research — your constant support and guidance made my time there truly enjoyable, and I learned a great deal from working with you both. I would also like to express my gratitude to Dr. Gabe Loh for his “intern office hours” sessions and his detailed guidance during my paper submissions within AMD, which were immensely helpful in strengthening the quality and impact of my work. I would also like to thank Prof. Jack Sampson for the many interesting discussions, both related to research and beyond, and Prof. Rui Zhang and Prof. Vijaykrishnan Narayanan for providing access to GPU compute resources. Finally, I would like to thank all the faculty and staff at Penn State whose teaching helped me learn and grow beyond the scope of this dissertation. I also thank the CSE administration and IT staff for all the timely help.

Next, I would like to thank all my labmates and peers at Penn State who have been an integral part of this journey, including Cyan, Anup, Sandeepa, Ziyu, Vishwas Kalagi, Scott, Siddhartha, Vivek, Deeksha, Shruti, Yan, Yintiang, Jun-liang, Lim, Sonali, Aakash, Shulin, Anoosh, Gulsum, Vishwas Kakrannaya, Wanhang, Goktan, Chia-Hao, and Soumya. You have always been there for me and made my time at Penn State truly enjoyable, turning even the busiest days into fun and memorable moments. Among them, I would especially like to thank Cyan, for countless discussions—from research challenges and idea brainstorming to reflections on academia and life. He has been my fire brigade, always stepping in whenever things felt overwhelming. I would also like to thank the HPCL alumni Adwait, Ashutosh, Xulong, Jashwant, Prashanth, Ani, and Asit, for their constant encouragement and guidance. Additionally, I am grateful to my internship friends and colleagues Anshujit, Wantong, Pakize, Kailash, Jagadish, Alex, Vignesh, Heather, and Avinash, for the wonderful experiences.

I am deeply grateful to Mrs. Namita Das (Lucy Aunty) for her love, care, and constant support. Her warmth, encouragement, and amazing food always made me feel at home, even when I was thousands of miles away from my own. I would also like to express my heartfelt gratitude to my friends — Cyan Mishra, Anup Sarma, Koustubh Phalak, Vishwas Kalagi, Elphin Tom-Joe, Abhishek Khare, Nitish Kulshrestha, and Abhinav

Singh — for being a constant source of strength and support. Their encouragement and companionship have helped me through both the highs and lows of my academic and personal journey. I am truly thankful for their friendship and presence in my life. I would also like to thank my Art of Living friends — Kelly, Derek, and Mukundan — for bringing peace, positivity, and joy into my days.

Finally, I would like to express my deepest gratitude to my family — my parents, Ashok Kumar Jain and Sunita Jain, my elder brother Akshay Kumar Jain, my sister-in-law Itisha Jain, my younger brother Akshat Kumar Jain, and my uncle Rajkumar Jain — for being my greatest pillars of love, strength, and inspiration throughout this Ph.D. journey. Your endless encouragement, faith, and blessings have given me the courage to face every challenge and the motivation to keep moving forward. Whatever I have achieved today is because of your unwavering support and belief in me. No words can truly capture how grateful I am for your boundless love and constant presence in my life. I would also like to thank my fiancée, Megha Jain, whose love, joy, and companionship bring warmth and balance to my life every day. Lastly, I wish to remember and thank my school and coaching class teachers, mentors, and well-wishers, who from an early age nurtured my curiosity and instilled in me a lifelong love for learning.

Lastly, I thank the National Science Foundation (NSF). This dissertation has been funded by grants: 2211018, 2116962, 1955815, 1763681, and NSF Chameleon Cloud projects CHI-220948 and CHI-231143. The findings and conclusions are the views of the author and do not necessarily reflect the views of the funding agencies.

Dedication

In loving memory of my late grandfather Shri Shantilal Jain (Baakliwal) and late grandmother Shrimati Manohar Devi Jain.

Chapter 1 |

Introduction

Recommendation systems have transformed how consumers find products among a growing number of choices, specifically in areas such as e-commerce, media and entertainment, and online advertising [5–10]. The market for recommendation systems generates billions of dollars in revenue [11, 12] and is expected to grow at a 36.3% rate from 2024 to 2030 [13]. Deep learning recommendation models (DLRM) [14] enable recommendation systems by working with continuous and categorical features, where large amounts of data for these features are collected from many users and then accessed frequently when processing inference requests. These models have become one of the largest consumers of datacenter inference cycles, accounting for nearly 80% of AI serving workloads [2, 15], and driving the core revenue for technology giants such as Meta [16] and Google [17].

DLRMs consist mainly of four stages: bottom multi-layer perceptron (MLP), embedding lookup, feature interaction, and top MLP. Out of these, the embedding stage is known to be the most memory-intensive stage that dominates the overall execution time [2, 15]. Due to the high computational and memory bandwidth needs of DLRMs [18], both CPUs and GPUs are used for hosting such workloads. Performing a characteriza-

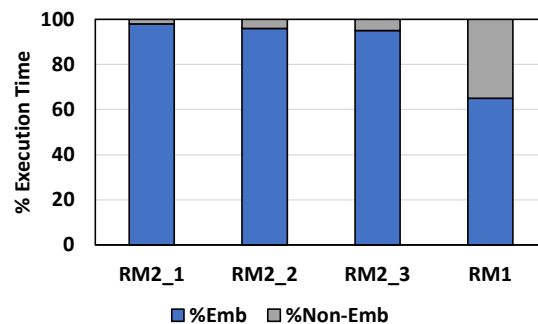


Figure 1.1: Execution time breakdown of different DLRMs.

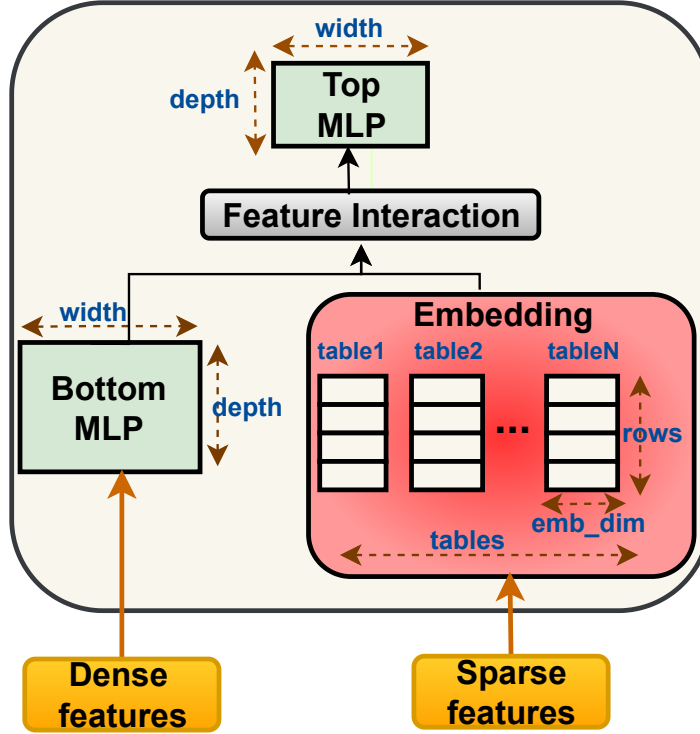


Figure 1.2: DLRM Architecture and Inference serving pipeline

tion study running recent recommendation models/datasets, our study highlights the dominance of embedding stage in the end-to-end inference execution. Figure 1.1 shows that the embedding stage constitutes a significant amount of execution time for different recommendation models.

1.1 Background

1.1.1 DLRM Inference

Multiple steps are involved in an end-to-end DLRM inference. First, the recommendation model, which determines the memory and computation footprints, is loaded into DRAM once. When input queries come, the system chunks queries into batches and prepares dense and sparse feature as inputs to the DLRM. During inference, the computation processes the input batches and produces the prediction. Typically, predictions are user-facing and must be done in near real-time. Consequently, these prediction tasks are administered under Service Level Agreements (SLAs) for guaranteeing QoS requirements for the end-user [19], [20], [21], [22], [23], [24].

Figure 1.2 shows a generic DLRM architecture consisting of four stages: bottom MLP, embedding lookup, feature interaction, and top MLP. These models use two types of input features— *dense features* and *sparse features*. Dense features represent continuous values, such as a user age or a product’s price, whereas sparse features are ones with a categorical, non-continuous values, such as the product id or the genre of the movie. During inference, first, the dense features are fed to the bottom MLP, while the sparse features are converted to a "dense latent representation" through the *Embedding Lookup Stage*. In the embedding lookup stage, the value of each sparse feature is used as an *index* to lookup a large embedding table that holds semantically meaningful embedding vectors for each value in each row. The outputs of the bottom MLP and the embedding table go through the feature interaction layer to calculate the interactions, and goes through the top MLP which predicts the click-through rate (CTR) [2, 25].

In general, embedding stages are "memory-intensive", while others are "compute-intensive" [2, 19, 26]. However, the overall system characteristics depend significantly on the hyperparameters of each stage (e.g., Bottom/Top-MLP’s width and depth, embedding tables’ row size, vector dimension, and the number of tables). For example, [2, 15] showed that real-world production models can be bottlenecked by either the embedding lookup stage or MLP stage, along with widely varying model sizes.

1.1.2 Memory Accesses in the Embedding Stage on CPU

Algorithm 1: Simplified Memory access loop in the embedding stage in CPU.

```

for  $i$  in  $0 \dots \text{num\_batches}$  do
  for  $j$  in  $0 \dots \text{num\_tables}$  do
    for  $k$  in  $0 \dots \text{batch\_size}$  do
      for  $l$  in  $0 \dots \text{lookup\_per\_sample}$  do
        for  $m$  in  $0 \dots \text{embedding\_dim}$  do
          vec.ld accm;
          vec.ld row_block;
          vec.add accm, accm, row_block;
          vec.st accm;

```

As multiple parameters are involved in the embedding stage, it is important to understand the impact of each and its interaction with others, especially as we examine software approaches later on to address the memory bottleneck. Algorithm 1 shows the

Algorithm 2: Main indirect memory access loop in the `embedding_bag` operator.

```

for  $i \leftarrow \text{start}$  to  $\text{end}$  do
  output_ptr  $\leftarrow$  &output[ $i * \text{ddim}$ ]
  zero_ker(output_ptr, ddim);
  for  $s \leftarrow \text{offsets}[i]$  to  $\text{offsets}[i+1]$  do
    row_ptr  $\leftarrow$  &table[indices[ $s$ ] * ddim]
    add_ker(output_ptr, row_ptr, ddim);

```

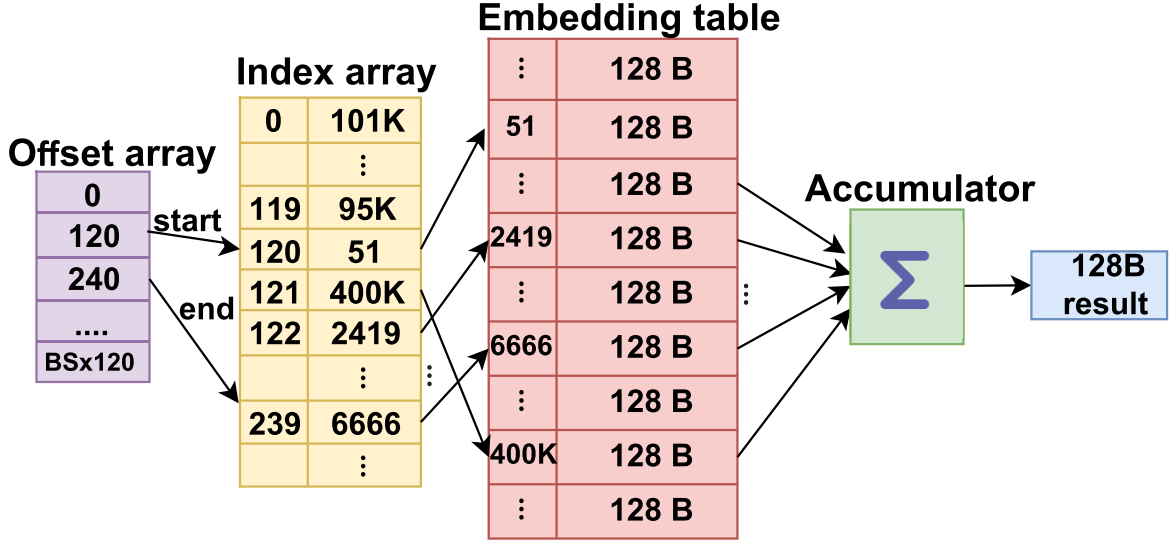


Figure 1.3: Illustration on the working of the `embedding_bag` operator for one sample (assume BS means batch size, `embedding_dim` = 128, and lookups per sample as 120). Three levels of indirection are involved.

simplified memory access loop for an input query. A query generates multiple batches to execute. Each batch is comprised of many samples and each sample is associated with a number of lookups. The model decides the number of tables and embedding dimensions.

A sample may perform multiple lookups to each table. Multiple lookups are aggregated (e.g., summed up and stored in the `accm` row vector). This is accelerated on the CPU by using the SIMD/AVX vector operations in the order: of load, add, and store. As the size of the embedding (`embedding_dim`) is usually not very large (4–8 cachelines), the spatial locality the load exhibits is very low.

We investigate the irregularity associated with the lookup by presenting the working of PyTorch’s `embedding_bag` operator, which is a representative implementation for embedding table lookup used in [25]. Algorithm 2 shows the core memory access loop

in the `embedding_bag` operator. To better understand the memory access behavior, Figure 1.3 gives an example to illustrate the steps and operations involved in Algorithm 2. The `embedding_bag` operator takes an input offset & index arrays for a batch and works on one table. The implementation involves the following four steps: (i) The sample dictates the access to the offset array to find the range defined by start and end values; (ii) In this range, the index array is accessed to obtain an index. These indices correspond to item ids; (iii) The index is used to look up the embedding table to output an embedding row vector; and (iv) The embedding row vectors are accumulated to form the result output embedding. The *three-level indirections (shown in Figure 1.3), which are carried out in loading the embedding row vector, can result in highly irregular memory accesses.*

1.1.3 Memory Accesses in the Embedding Stage on GPU

Algorithm 3 highlights the high-level working of the embedding stage on GPU. As mentioned earlier, arriving queries create “batches”, where each batch is expected to meet the SLA target. For each table, the batch contains a batch size (BS) number of samples, and each sample involves a pooling factor (or lookups per sample) amount of work over the embedding vectors of length equal to the embedding dimension (ED). At the core, each sample does a gather (load) and reduce (accumulation) operation. The amount of data processed in each table can be calculated as $(BS) \times (\text{average lookups per sample}) \times (ED) \times (\text{precision})$. For example, for our chosen configuration, the amount of data processed per table is $2048 \times 150 \times 128 \times 4B = 150 \text{ MB}$. Consequently, the complete embedding stage processes 37.5 GB of data. With the inference running on a GPU, the gather-reduce operations are executed in a Single-Instruction-Multiple-Threads (SIMT) manner, thus exploiting parallelism.

Algorithm 3: Simplified memory access loop for the embedding stage on GPU.

```

for  $v$  in  $0 \dots \text{num\_batches}$  do
  for  $w$  in  $0 \dots \text{num\_tables}$  do
    for  $x$  in  $0 \dots \text{batch\_size}$  do
      for  $y$  in  $0 \dots \text{lookups\_per\_sample}$  do
        SIMT load accm on register;
        for  $z$  in  $0 \dots \text{embedding\_dim}$  do
          SIMT load row_block on register;
          SIMT add accm, row_block;
        SIMT store accm to memory;

```

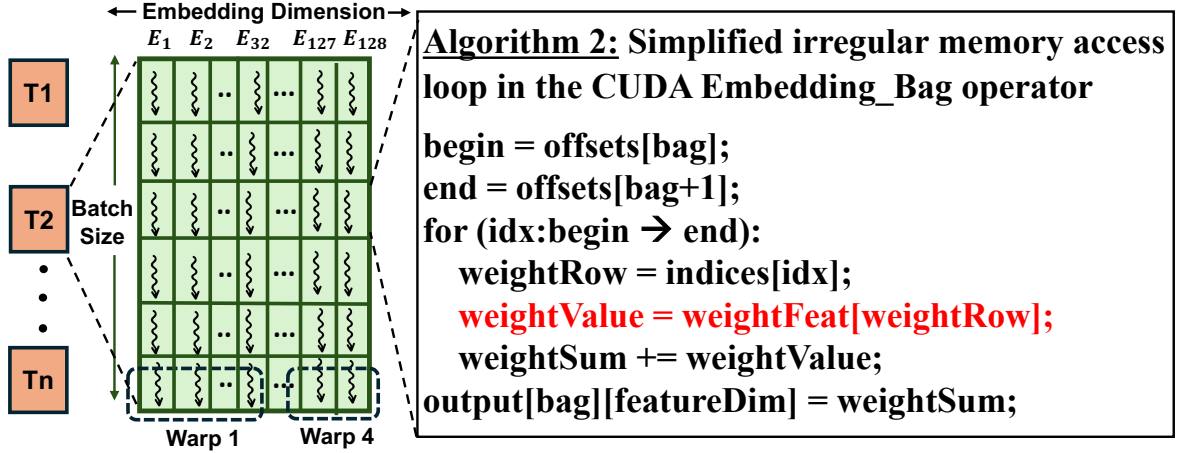


Figure 1.4: Parallel implementation of embedding stage by work partitioning across CUDA threads. Here, 1000s of CUDA threads independently work on one output matrix element.

To better understand the incorporated parallelism, Figure 1.4 breaks down the embedding stage execution into three parts. First, for any number of embedding tables to be completed by one GPU, they are processed sequentially. Second, the embedding bag operator is used to process a table (using PyTorch’s backend CUDA kernel *"EmbeddingBag_updateOutputKernel_sum_mean"* [27]), which generates an output matrix of dimension $(BS) \times (ED)$. Intuitively, we can visualize that, within a batch, each sample and each embedding element is independent of the other. Thus, a CUDA thread works on each embedding element. In this off-the-shelf kernel, we note a static execution launch configuration with a grid size of $(1024, 1, 1)$ and a block size of $(32, 8, 1)$. This results in a large number of CUDA threads, and thus, fully uses all the SMs provided in the latest GPUs (e.g., the 108 SMs in A100). Warps are automatically formed in the GPU by combining adjacent CUDA threads. For example, with an embedding dimension of 128, 4 warps are formed to process a sample. Third, Algorithm in Figure 1.4 highlights the work within a thread which encapsulates a “number of lookups” amount of gather-reduce operations (thus each thread partially completes the two innermost loops in Algorithm 3). Fundamentally, the gather is similar to a pointer-chasing operation as we access a series of arrays to complete it (the offset array, followed by the indices array, followed by the embedding table). Thus, this operation results in the irregular loads which have a significant impact on (and causes variation in) the performance of the embedding stage.

1.1.4 Key Challenges

The pursuit of higher accuracy in modern recommendation systems has led to unprecedented model growth and complexity. The latest models now comprise trillions of parameters, hundreds to thousands of embedding tables of varying sizes, and many pooling operations that capture extensive user interaction histories. This expansion has intensified three fundamental challenges: *model capacity*, as the vast parameter space demands enormous memory and storage; *memory irregularity*, due to presence of varying degree of reuse patterns which can be highly sparse; and *computation demand*, driven by large pooling sizes and increasingly deeper and wider MLP layers. Addressing these challenges requires substantial capital expenditure, as industry leaders like Meta, Google, NVIDIA, and AMD continue to develop new accelerator generations—MTIA [28], TPU [29], Hopper [30], Blackwell [31], and MI300 [32]—to keep pace. These trends naturally raise a central question: *can existing hardware architectures effectively sustain the growing performance, memory, and efficiency demands of next-generation DLRLMs?*

1.1.5 Dissertation Contributions:

Contribution-I: Optimizing CPU Performance for Recommendation Systems At-Scale - We first analyze the performance breakdown for various DLRLMs and show that the embedding stage is the most significant contributor in the end to end execution time. Next, to capture the diversity in the embedding access patterns, we categorize the production traces [33] into different datasets based on their hotness. Over these hotness, we examine the perform implications in terms of execution time and hardware profiling, including cache hit rates and average load latency. Further, we study the locality behavior using an access count distribution study and an analytical model based reuse distance study. To maximize the CPU utilization, we conduct a multi-core scalability study to understand how performance scales and how bandwidth is consumed across various datasets and core counts. Given the presence of large reuse distances and cold misses, we propose to design an application-specific software prefetching to hide the long access latencies, application-aware HyperThreading to overlap computation with memory accesses, and combination of them. Finally, we implement and evaluate our proposals on server-grade CPUs.

Contribution-II: Parallelization of Embedding Stage on CPUs - As the embedding stage is quickly evolving with an increasing number of tables, larger embedding dimensions and dataset sizes, assigning each batch to a separate core may hinder the

ability to meet user-facing latency targets. To address this challenge, we propose parallelizing a single batch across multiple cores. We evaluate the multi-core scaling of state-of-the-art parallelization technique called Batch Threading (BT) and observe limited scaling. Next, we analyze Meta’s DLRM production traces which we label as Heterogeneous Datasets, and observe several non-uniformities across tables including execution latency, row sizes, pooling factors, and memory access patterns. To resolve the limited scaling of BT, we propose to design two new parallelization techniques: Table Threading (TT) and Hierarchical Threading (HT). Finally, we implement and evaluate our proposals on server-grade AMD EPYC™ “Genoa” CPU.

Contribution-III: Balance: Load and MLP-Aware Thread Orchestration for Recommendation Systems Inference on CPUs - Despite Hierarchical Threading (HT) achieving significant performance benefit over Batch Threading (BT) and Table Threading (TT), we note that the multi-core scaling is imperfect over heterogeneous datasets. To understand the limited scaling, we note that HT uniformly maps tables to the work queues which can lead to imbalanced load distribution across threads. We propose to develop a reuse distance based analytical to predict the amount of work (Time Value or TV) of each table, thus able to categorize the tables based on the load contained. Further, we observe that HT over-provisions the cores assigned per table. To resolve the limited scaling of HT, we propose a Load and MLP-aware Thread Scheduling, where we modify the threadpool implementation of HT, and add the features of table reordering, small number of work queues, low core grouping, strategic table co-location, and work stealing. Finally, server-grade AMD EPYC™ “Genoa” CPU is used for evaluation.

Contribution-IV: Pushing the Performance Envelope of DNN-based Recommendation Systems Inference on GPUs - With the ever-increasing compute and memory-bandwidth requirements of DLRMs, it is increasingly being preferred to execute them on GPUs [28, 34] due to their efficient parallel processing capabilities and HBM support. However, with growing model and dataset sizes, efficient utilization of GPUs for improving the performance of inference applications, as will be shown in the dissertation, is insufficiently investigated. We show that the embedding stage continues to be the primary bottleneck even in the GPU inference pipeline. To thoroughly grasp the problem, we conduct a detailed *microarchitecture characterization* and highlight the presence of low occupancy in the standard embedding kernels. By leveraging direct compiler optimizations, we achieve optimal occupancy, pushing the performance significantly. Yet, long memory latency stalls continue to exist. To tackle this challenge, we propose specialized plug-and-play-based *software prefetching and L2 pinning techniques*, which

help in hiding and decreasing the latencies. Further, we propose combining them, as they complement each other. Finally, we implement and evaluate our proposals on a server-grade Nvidia A100 and H100 GPU.

Contribution-V: HetT-HetD: Embracing Heterogeneity in Embedding Tables and Devices for Performant Recommendation Systems - Modern DLRLMs exhibit diverse compute and memory behaviors, challenging the one-device-fits-all paradigm. While non-embedding stages remain compute-bound and favor GPUs, the memory-bound embedding stages vary in capacity and data reuse. CPUs are better suited for high-reuse, large-capacity tables due to their rich cache and memory capacity, whereas GPUs are ideal for low-reuse tables given their high-bandwidth memory. To exploit these complementarities, we propose a concurrent CPU-GPU DLRLM execution that maximally leverages each device’s strengths. Our design aims to (1) efficiently utilize CPU and GPU performance, (2) prevent GPU memory oversubscription, and (3) balance workload across devices. We achieve these goals through asynchronous CPU-GPU data movement and a heuristic-based scheduling policy for table-to-device mapping. Finally, we implement and evaluate our proposals on a server-grade Zen4 CPU and H100 GPU.

Chapter 2 | Optimizing CPU Performance for Recommendation Systems At-Scale

2.1 Introduction

Recommendation systems have made a profound impact in numerous domains such as e-commerce [35–38], entertainment [39–41], and social networks [25], by providing personalized suggestions for decision making to enhance the user experience. At the heart of these systems lie Deep Learning Recommendation Model (DLRM) that incorporates large feature sets gathered not just from one user, but across multiple users [25]. With time, based on continuous usage, the corpus of features and data continue to get added to refine the model to make it more accurate and precise. These models are, thus, very memory/storage intensive, not only having large memory footprints but also requiring frequent, periodic accesses to them when making a recommendation.

DLRMs consist mainly of four stages: bottom multi-layer perceptron (MLP), embedding lookup, feature interaction, and top MLP. Out of these, the embedding stage is known to be the most memory-intensive stage that dominates the overall execution time [2, 15]. Due to the high computational and memory bandwidth needs of DLRMs [18], both CPUs and GPUs are used for hosting such workloads.

Traditionally, GPUs have been a popular choice for DLRM training since their computational demands are much higher than inference [42]. On the other hand, CPU has been a popular choice for inference [2, 15, 19, 43]. Unlike training, inference cannot use a very large batch size due to its tight latency requirement [2, 19], thus, making the GPU less effective and thereby, positioning the CPU as an appealing economical alternative [44–49]. As the growth of the memory requirement has been much higher than the growth of the computational demands of the MLP layer [43], this trend is

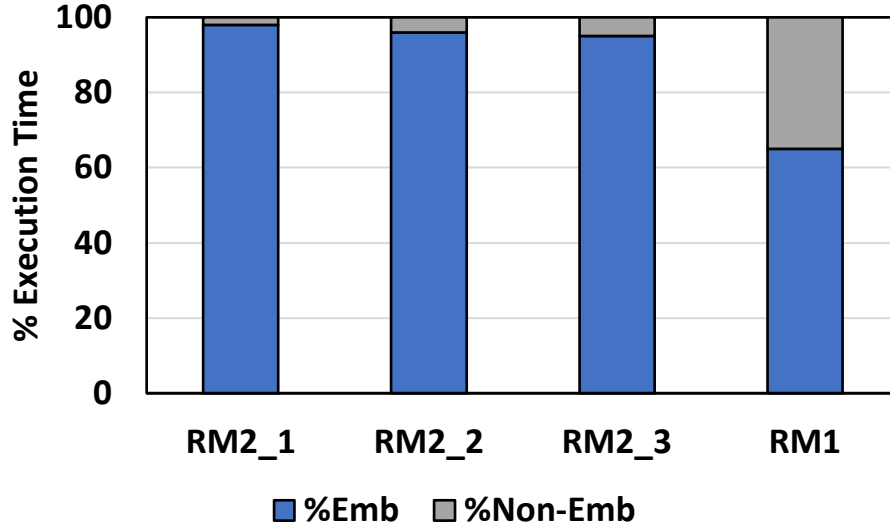


Figure 2.1: Execution time breakdown of different DLRMs.

expected to hold in the new generations of DLRM as well. Using recently released recommendation models/data [50–53], we confirm that CPUs are still a reasonable platform for newer, next-generation DLRMs.

Performing an extensive characterization study running recent recommendation models/datasets on CPUs, our study reaffirms that the memory bottleneck continues to plague the CPU execution. Figure 2.1 shows that the embedding stage constitutes a significant amount of execution time for different recommendation models (detailed later in Section 5.2). We observe that in the embedding lookup stage, accesses to (sampled) rows of several tables have very large (or even infinite – denoting cold misses) reuse distances, making on-chip caches employing LRU or its variants woefully inadequate to store these working sets. To the best of our knowledge, this is *the first study to conduct a detailed memory characterization* of these newer recommendation models and data running on a CPU, to show their reuse behavior and point to the cold/capacity misses arising consequently.

Many prior works have observed similar problems and have tried to address them by boosting memory bandwidth [54, 55] and/or using Processing-In-Memory/Storage mechanisms [56–59]. However, all these approaches require a substantial (r)evolution in the hardware, which may (if at all) take considerable time to commercialize and make the system difficult to adapt to the rapid change of model design. Instead, in this paper, we explore lower cost/design options that can be instantly leveraged by asking the following question: *Can we employ simple software techniques on top of existing hardware to reduce/hide the memory bottleneck, in both an application- and architecture-aware fashion?*

We examine two broad strategies to mitigate the impact of this behavior that can be implemented in software: (1) Prefetching to reduce cache misses, and (ii) Multi (Hyper-)threading to hide the consequences of such misses. While prefetchers can potentially reduce/hide large reuse distance misses and cold misses, traditional hardware- or compiler-based prefetching is inefficient because of the irregular nature of embedding table accesses that show extremely low temporal/spatial locality. Instead, we couple the software doing the sampling with the initiation of corresponding prefetches in the application code to perform the exact prefetch of only the necessary indices. We show that with such an approach we can cut down the miss rate up to 26.8%, which speeds up the embedded lookup stage up to 47% and eventually, the overall inference process up to 46%.

Despite our best efforts, there are bound to be misses even after prefetching. To hide the effects of such misses, we leverage the fact that the bottom MLP and the embedding lookup stage are two independent stages that are compute-bound and memory-bound, respectively. As they can run independently, we run these two threads in each physical core in parallel (i.e., hyperthreading), to better utilize the datapath resources while the other thread is stalling for memory. We find that colocating the bottom MLP and the embedding stage in each core can improve inference performance by 37% independently, and can provide a synergistic 1.59x additional improvement when combined with prefetching. To the best of our knowledge, these two ideas (application-initiated software prefetching and hyperthreading based on application knowledge) have not been previously proposed for DLRM inferences.

Using the recently-developed DLRMs [25, 50, 52, 53] and production traces from Meta [60], and the state-of-the-art server CPU platforms, this paper makes the following contributions:

- We show that off-chip accesses and memory bandwidth continue to be serious bottlenecks in CPU executions of the current and future recommendation models. We also show that the causes are in the large reuse distances (and even cold misses) during the embedding stage which samples and accesses random rows of several tables.
- Without using new technologies and hardware changes, this paper proposes two software-based techniques – prefetching and hyperthreading – on existing hardware to deal with this bottleneck. We tailor both well-known hardware mechanisms in a specific way tailored to DLRM. We initiate software prefetches based on lookup selection, and colocate two independent threads with very different resource

requirements on the same physical core for effective hyperthreading.

- We present a thorough evaluation of the proposed two techniques. Individually, our software prefetching and hyperthreading proposals provide up to 47% and 37% inference time improvements. Further, when combined together, they provide up to a 59% synergistic improvement.

2.2 Background and Motivation

In this section, we provide a brief background on modern recommendation models, analyze the irregularity in the time-consuming embedding stage, and discuss the related works and our novelty.

2.2.1 DLRM Inference

Multiple steps are involved in an end-to-end DLRM inference. First, the recommendation model, which determines the memory and computation footprints, is loaded into DRAM once. When input queries come, the system chunks queries into batches and prepares dense and sparse features as inputs to the DLRM. During inference, the computation processes the input batches and produces the prediction. Typically, predictions are user-facing and must be done in near real-time. Consequently, these prediction tasks are administered under Service Level Agreements (SLAs) for guaranteeing QoS requirements for the end-user [19–24]. Some common SLA targets (as seen in [2]) are shown in Table 2.1.

Figure 2.2 shows a generic DLRM architecture consisting of four stages: bottom MLP, embedding lookup, feature interaction, and top MLP. These models use two types of input features— *dense features* and *sparse features*. Dense features represent continuous values, such as a user’s age or a product’s price, whereas sparse features are ones with a categorical, non-continuous values, such as the product id or the genre of the movie. During inference, first, the dense features are fed to the bottom MLP, while the sparse features are converted to a "dense latent representation" through the *Embedding Lookup Stage*. In the embedding lookup stage, the value of each sparse feature is used as an *index* to lookup a large embedding table that holds semantically meaningful embedding vectors for each value in each row. The outputs of the bottom MLP and the embedding table go through the feature interaction layer to calculate the interactions, and goes through the top MLP which predicts the click-through rate (CTR) [2, 25].

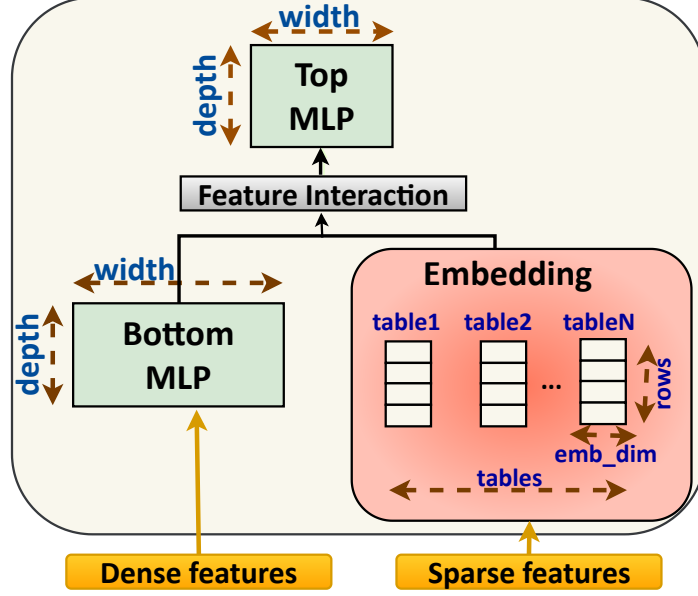


Figure 2.2: A generic DLRM architecture.

Model Class	Execution Bottleneck	Model Size	SLA Target
RMC1	Embedding $\approx 60\%$	Small	100ms
RMC2	Embedding $\approx 90\%$	Large	400ms
RMC3	MLP $\approx 80\%$	Medium	100ms

Table 2.1: Different model characteristics from [2].

In general, embedding stages are "memory-intensive", while others are "compute-intensive" [2, 19, 26]. However, the overall system characteristics depend significantly on the hyperparameters of each stage (e.g., Bottom/Top-MLP's width and depth, embedding tables' row size, vector dimension, and the number of tables). For example, [2, 15] showed that real-world production models can be bottlenecked by either the embedding lookup stage or MLP stage, along with widely varying model sizes (Table 2.1, Table 2.2).

2.2.2 Memory Accesses in the Embedding Stage

As multiple parameters are involved in the embedding stage, it is important to understand the impact of each and its interaction with others, especially as we examine software approaches later on to address the memory bottleneck. Algorithm 8 shows the simplified memory access loop for an input query. A query generates multiple batches to execute. Each batch is comprised of many samples and each sample is associated with a number of lookups. The model decides the number of tables and embedding dimensions. A sample may perform multiple lookups to each table. Multiple lookups are aggregated

Algorithm 4: Simplified Memory access loop in the embedding stage in CPU.

```

for  $i$  in  $0 \dots \text{num\_batches}$  do
  for  $j$  in  $0 \dots \text{num\_tables}$  do
    for  $k$  in  $0 \dots \text{batch\_size}$  do
      for  $l$  in  $0 \dots \text{lookup\_per\_sample}$  do
        for  $m$  in  $0 \dots \text{embedding\_dim}$  do
          vec.ld accm;
          vec.ld row_block;
          vec.add accm, accm, row_block;
          vec.st accm;

```

Algorithm 5: Main indirect memory access loop in the embedding_bag operator.

```

for  $i \leftarrow \text{start}$  to  $\text{end}$  do
  output_ptr  $\leftarrow$  &output[ $i * \text{ddim}$ ]
  zero_ker(output_ptr, ddim);
  for  $s \leftarrow \text{offsets}[i]$  to  $\text{offsets}[i+1]$  do
    row_ptr  $\leftarrow$  &table[indices[ $s$ ] * ddim]
    add_ker(output_ptr, row_ptr, ddim);

```

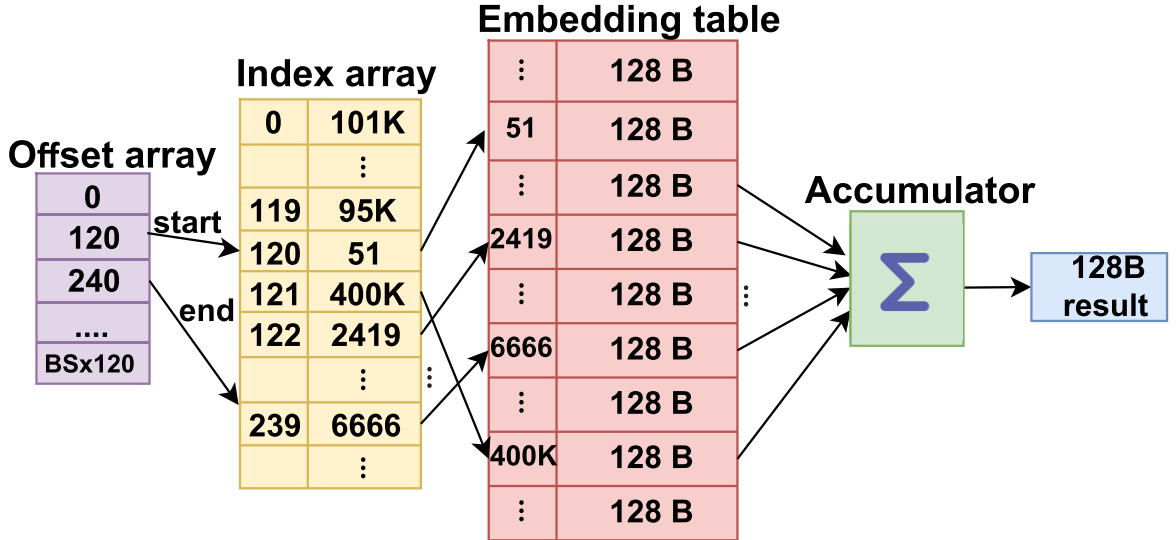


Figure 2.3: Illustration on the working of the embedding_bag operator for one sample (assume BS means batch size, embedding_dim = 128, and lookups per sample as 120). Three levels of indirection are involved.

(e.g., summed up and stored in the `accm` row vector). This is accelerated on the CPU by using the SIMD/AVX vector operations in the order: of load, add, and store. As the size of the embedding (`embedding_dim`) is usually not very large (4–8 cachelines), the spatial locality the load exhibits is very low.

We investigate the irregularity associated with the lookup by presenting the working of PyTorch’s `embedding_bag` operator, which is a representative implementation for embedding table lookup used in [25]. Algorithm 5 shows the core memory access loop in the `embedding_bag` operator. To better understand the memory access behavior, Figure 2.3 gives an example to illustrate the steps and operations involved in Algorithm 5. The `embedding_bag` operator takes an input offset & index arrays for a batch and works on one table. The implementation involves the following four steps: (i) The sample dictates the access to the offset array to find the range defined by start and end values; (ii) In this range, the index array is accessed to obtain an index. These indices correspond to item ids; (iii) The index is used to look up the embedding table to output an embedding row vector; and (iv) The embedding row vectors are accumulated to form the result output embedding. The *three-level indirections (shown in Figure 2.3), which are carried out in loading the embedding row vector, can result in highly irregular memory accesses.*

2.2.3 Related Work

Comparison with Prior Inference Optimizations. [49, 54, 56, 58, 61] have proposed near-memory processing, cache optimizations, and algorithmic approaches to improve recommendation inference. They rely on the fact that a small fraction of embedding entries contribute to a major fraction of accesses, exhibiting temporal reuse. However, we show in Section 2.3.1.1 that temporal locality can be much worse for traces with larger embeddings and low hotness, rendering these prior approaches less effective. Further, many of the previous works modify the underlying hardware, thus, limiting the practicality and generalizability of their approaches. Our techniques can easily be embraced by present applications and CPUs, and can accelerate a wider range of traces regardless of their hotness.

Prefetching Optimizations. [62, 63] provide state-of-the-art compiler pass for automating software prefetching for indirect memory accesses. These works target general workloads and are not tailored to recommendation models, missing specific optimizations that we found to be useful. For example, [63] lacks control over the prefetch amount, which we found to be an important knob for optimization (Section 2.4.2). [64, 65] are state-of-the-art hardware prefetching techniques for irregular accesses. Again, as they

intend to cover a wide variety of workloads, they suffer from limited prefetch accuracy compared to our tailored solution. Off-the-shelf-CPU [66] employ simple hardware prefetchers due to area/power constraints and target only next-line/stride accesses which are insufficient for irregular accesses (Section 2.4.2).

Multithreading and Hyperthreading. [15] showed that naive adoption of hyperthreading for DLRM can hurt the tail latency. Many other previous works [2, 15, 19, 67] dismissed hyperthreading for similar reasons. We show in this work that hyperthreading can actually benefit DLRM workloads when designed in the right way. We also show how it can be synergistically integrated with prefetching for additional performance benefits.

Recommendation Models. Many different models have been proposed by both academia and industry for recommendation [25, 36, 38, 39, 68–70]. These models mostly differ in how they calculate the interactions between different features and select important features; however, they all still commonly use large embedding tables that pressure the memory bandwidth heavily. We expect our proposed design to be applicable to many of these different models, as our design works well for a wide range of table dimensions and access patterns to ease the memory traffic and improve latency.

2.2.4 Our Novelty

We do not claim novelty in running DLRMs on CPUs. As several prior studies note [2, 15, 19, 43–49, 71], CPUs are becoming increasingly competitive to take on these workloads opportunistically in a datacenter with high GPU demand.

Prior published studies with DLRMs on CPUs/GPUs have used datasets/models that are smaller and experienced more skewed access patterns. We explore larger models with a wide range of access patterns, which revealed that *the (temporal) data locality can get much worse than what prior work assumed* (Section 2.3.1.1). This in turn can make some previously proposed techniques ([49, 54–56, 58, 61]) less effective. Instead, what we show is that it is more important to either (i) reduce the latency for off-chip accesses due to LLC misses (using prefetching) or (ii) tolerate the latency of off-chip accesses (using multithreading), or (iii) possibly both, for these newer datasets.

Similarly, prefetching and hyperthreading are themselves not new and we do not claim that we are first to use them for DLRMs – unpublished industrial production systems might be already using them to a certain extent. However, *there is no published research, to our knowledge, that systematically studied how/when to prefetch or do multithreading intelligently for DLRM workloads.*

With respect to prefetching, we show that simply employing a state-of-the-art hard-

Category	Name	Type	Execution time (Emb%)	Emb. Size (GB)	Rows	Emb. dim	Tables	Lookups per sample	Bottom-MLP	Top-MLP	Per table capacity(MB)
RMC2	rm2_1	small	98	28.6	1M	128	60	120	256-128-128	128-64-1	488.3
	rm2_2	medium	96	57.2	1M	128	120	150	1024-512-128-128	384-192-1	488.3
	rm2_3	large	95	81.1	1M	128	170	180	2048-1024-256-128	512-256-1	488.3
RMC1	rm1		65	3.8	500K	64	32	80	2048-2048-256-64	768-384-1	122.0

Table 2.2: Model architecture parameters in our study.

ware or software prefetcher [62–66, 72, 73] does not work very well (Section 2.4.1). Instead, *leveraging application knowledge of when and how the embedding tables are accessed is important to initiate and fine-tune prefetches*. As for hyperthreading, some prior works [15] show this to be detrimental to DLRMs. In contrast, *we show that DLRMs can benefit from hyperthreading, if new threads are spawned and grouped together smartly (based on application knowledge) for maximum effectiveness rather than letting the underlying system do the job*, which we have not seen in any prior study.

Finally, the two proposals *complement each other, producing benefits better than the sum of the parts, which is again a new observation*. This is because prefetching helps in freeing CPU pipeline resources, avoiding issues like full window stalls [74, 75], thereby allowing sibling thread to avail more resources.

2.3 Demystifying the memory access behavior and Multi-core Scaling

Section II highlighted the scope of memory access irregularity originating from the embedding stage. To better understand the architectural behavior of the embedding stage, in this section we perform an in-depth analysis of how the performance of the embedding stage is impacted by the newer datasets [60] and larger models [50–53] as well as the newer CPU platforms used for execution. Though previous works [2, 15, 54, 56, 58, 61] highlighted the presence of memory irregularity problem, they were limited by smaller models and skewed datasets. Specifically, we show that temporal locality worsens in the newer application settings. We motivate why memory access irregularity is an important problem in impacting performance, and develop a theoretical model to study the issue with memory access patterns and memory hierarchy. Further, we do a multi-core

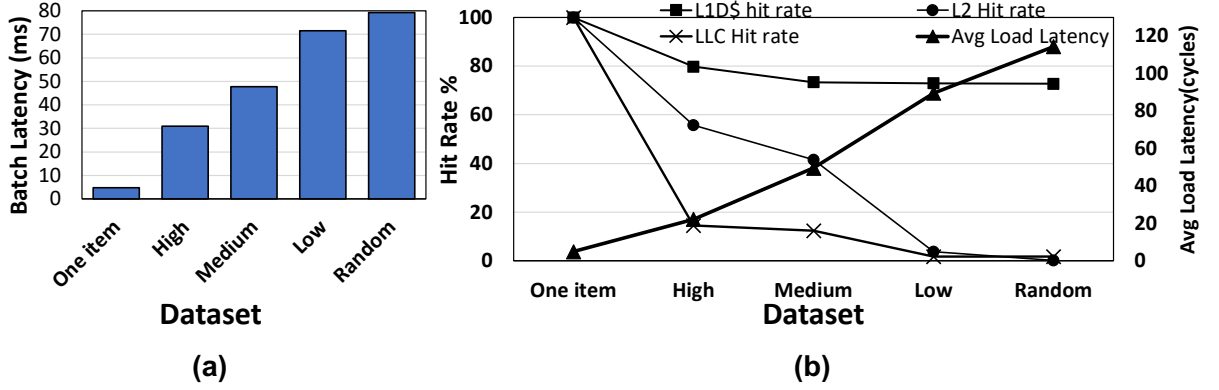


Figure 2.4: RM2_1 embedding stage performance comparison across datasets. (a) Batch latency (b) average load latency, and cache hit rates for L1D, L2, L3.

scalability and bandwidth study and share the key architectural insights.

2.3.1 Theoretical Analysis

To understand the issue of the embedding stage in more depth, we use VTune to perform an empirical characterization of RM2_1 model. Figure 2.4 compares the performance of the RM2_1 model for several metrics under various types of inputs. {High, Medium, Low} Hot corresponds to the three chosen data traces from Meta [60] (more details in 5.5). We consider two synthetic inputs: {one-item, random} to capture the spectrum of execution for the embedding stage. One item refers to the best case where all the accesses only go to one embedding row in a table, and random refers to the worst case where the rows are uniformly accessed. Figure 2.4(a) compares the batch latency across datasets, and Figure 2.4(b) compares the average load latencies and cache hit rates for L1D, L2, L3.

There is a heavy performance skew between one-item and other datasets. Memory accesses of “one item” are very regular as only one embedding row vector is used for a table. This would mean a minimal working set, thus the cache hit rates are maximized, and the average load latency is almost the same as the L1D\$ hit-latency. However, other datasets access many embedding vectors (based on their hotness) in an indirect fashion, resulting in a larger working set. Therefore, they suffer from irregularity which results in a drop in the cache hit rates and an increase in the average load latency. Thus, from Figure 2.4, we can say that the wide gap between one item and other datasets is because of the irregular memory accesses.

2.3.1.1 Access count study for different input loads

Earlier works [42, 54, 56] have highlighted that a power-law distribution is present in the index accesses where a subset of rows are more frequently accessed than others. In other words, some rows are more “hot” than others. The hotness in a given dataset can be quantified by plotting a histogram of access counts. Figure 2.5 shows the hot embedding behavior in the three production datasets. As depicted in Figure 2.5, the access count distribution varies across the three as per the hotness exhibited in them. The hotness of the dataset is important because it governs the working set size within a certain time window. If a dataset is “high hot”, most of the queries lookup the same row and result in a small working set size, while a “low hot” dataset will behave otherwise.

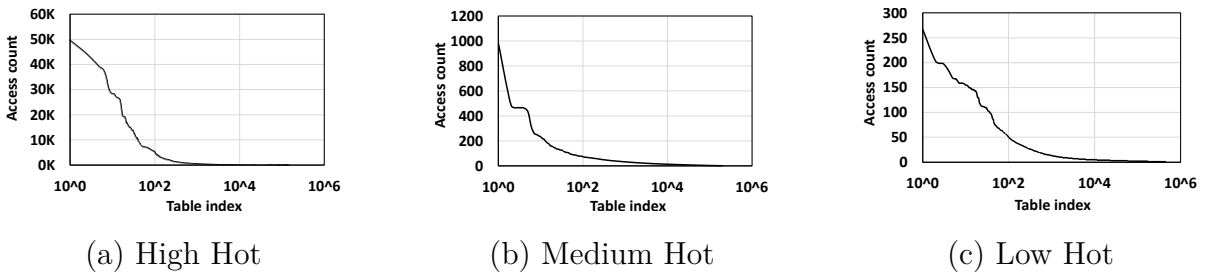


Figure 2.5: Hot embedding access count (sorted) in 3 datasets.

2.3.1.2 Reuse distance study

Given that rows are *reused*, it is still a question of how effective caches are in taking advantage of this row reuse. It would be *wrong* to simply conclude that hot embedding implies good temporal locality because at the time reuse occurs, other rows could already have *evicted* the copy of the needed row. Thus, to quantify how effective caches are in capturing the row reuse we are interested in, the following discussion investigates the reuse distances observed in the execution of the embedding stage.

Qualitative discussion on row reuse: Fundamentally, the dataset implicitly contains the repetition of the indices, which is a major contributor to the row reuse, as earlier shown in Figure 2.5. Connecting the input dataset to the implementation of DLRMs, one can find the reuse opportunities by carefully checking Algorithm I. The maximum working set of a program is based on the memory footprint of the embedding tables. Other reuse scenarios come from the program as it issues multiple loads for one embedding table, which happens in various ways: across lookups within a sample, across samples, and across batches. Also, when multiple cores are running inference, they could share the memory footprint of one table. For example, the cold miss penalty for a load

from one core could potentially help in a cache hit for other cores. To summarize, the loops for `num_batches`, `batch_size`, and `lookups_per_sample` in Algorithm I offer data reuse opportunities.

Quantitative discussion on row reuse: Since various parameters like input dataset, DLRM embedding stage parameters, and execution design impact the data reuse patterns, we propose a "model", which can capture these parameters to reflect on "reuse distance" associated with the memory loads issued in the embedding lookup stage. Figure 2.6 shows the high-level overview of our modeling scheme, which includes the following components: (1) the index access trace is collected based on the input dataset, embedding model parameters, number of cores, batch size, and number of batches. This mimics the index access pattern observed in Algorithm I; (2) reuse distance bins are generated on the input access trace based on the stack distance calculations [76] using the indices accessed; (3) each of the indices corresponds to a memory load request for the full embedding row vector. For a given cache capacity, we can model how many embedding row vectors it can accommodate. For simplicity, we assume a fully-associativity cache. For instance, for a 32KiB D\$ and 128 as the embedding dimension (assume 32-bit floating point precision), the D\$ can store 64 embedding vectors. Thus, the cache capacities are compared to the reuse distances to mark the corresponding hit rates observed. Figure 2.7 shows the reuse distances on the three datasets. The vertical lines mark the observed cache hit rates. The yellow shape on the top right shows the fraction of misses observed as cold misses. The thick red arrow indicates an interesting reuse pattern corresponding to the reuse across batches for the same table. It is observed that L1D\$ hit rates are very bad

Insights on temporal locality: From the above analysis, we can categorize the reuse distances into four classes:

(1) *Intra-table*: The accesses happening during the batched embedding_bag for a table can share the indices and thereby get benefited from the cache hierarchy. However, the temporal locality could only be exploited if the reuse distance fits within the cache capacity.

(2) *Inter-table*: Since two embedding tables have their own memory footprints, no embedding row vectors could be shared. Moreover, the next table would evict the data of the previous table from the caches, thereby destroying any previously created locality. Thus, as the samples switch the tables, cache thrashing can occur.

(3) *Inter-batch*: As each batch accesses all the embedding tables, we can think about the presence of locality across batches. For a table, this would happen when the same

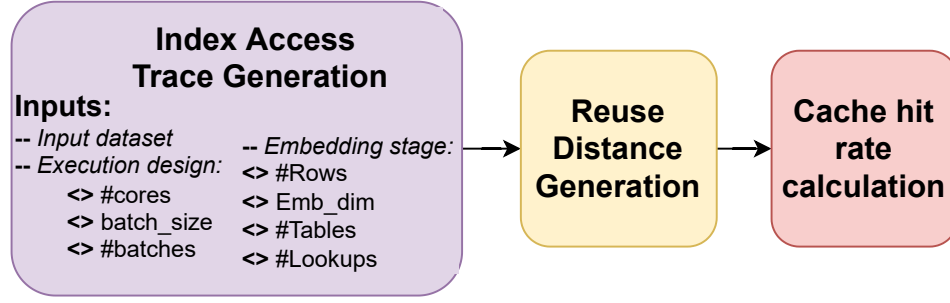


Figure 2.6: Our model for capturing the reuse distances, and for indicating cache hit rates and cold misses.

indices are accessed across batches. However, in this situation, the reuse distance is very large as many unique accesses happen in between corresponding to one less of the total tables. The thick red arrow in Figure 2.7 indicates this situation.

(4) *Inter-core*: During multi-core execution, the LLC capacity, and DRAM bandwidth get shared across cores. This sharing could lead to two situations: (a) Constructive sharing - This would happen when two or more cores are working on the same embedding table. The cache miss corresponding to an embedding row vector could be used by the other core. (b) Destructive sharing - This would generally happen when cores are working on different embedding tables. For all shared buffers (like LLC, row-buffer, etc.), one core could evict/thrash the data needed by another core. Even when two cores are working on the same table, thrashing may still happen as the working set of one table is very large.

Discussion on trace collection process: It is important to highlight how our trace generation compares with collecting traces from a real execution. In a real run, one could either modify the backend codebase to dump the load addresses corresponding to embedding vector accesses or instrument the memory accesses using a tool like PinTool [77]. However, there are two major issues with this setup. First, an actual run would be time-consuming and would limit the experimentation speed. And second, analyzing the role of the number of cores would be limited by the maximum number of cores supported in the setup machine, thus limiting the experimentation scope. Given these challenges, we choose to develop a model, instead of a complex setup in a real machine.

2.3.2 Multi-core Scaling and Bandwidth study

Previous works [2, 15, 54–56] have highlighted limitations of memory bandwidth in CPUs. However, modern CPUs have significantly higher memory bandwidth Cascade Lake [78], Ice Lake [79]. Also, Intel has heavily tuned the PyTorch operators for optimizing the

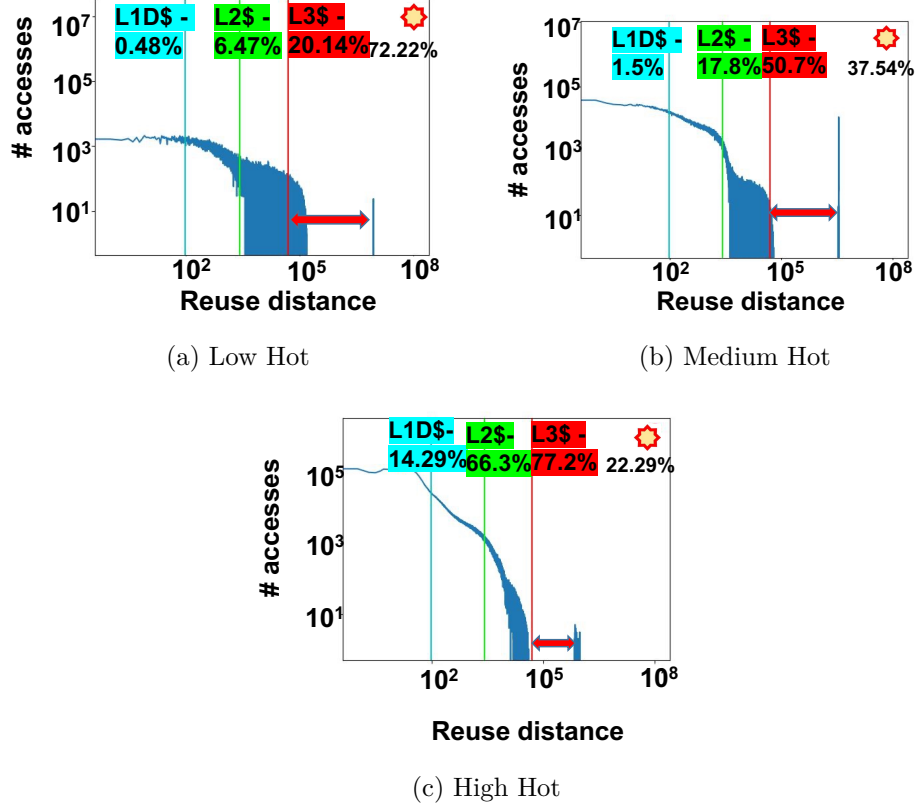


Figure 2.7: Reuse distance study for different datasets in rm2_1 model for 24 cores & batch size = 64.

performance on CPUs [80]. To improve throughput during inference, we propose to utilize all CPU cores, such that a batch is mapped to each core. Figure 2.8 shows the variation in execution time and memory bandwidth of RM2_1 models as the number of cores increases. From a single core to 24 cores, the execution time increases by 14% and the memory bandwidth increases by 15.5 times. While investigating into VTune [81], the execution time does remain memory bandwidth bound by 80% in Low Hot but the bandwidth does not get fully utilized. This gives an opportunity to utilize the remaining memory bandwidth for optimizing the performance.

2.3.3 Key Takeaways

From the above characterization, we make the following key observations:

- (1) Irregularity in the memory accesses is governed by the input dataset (meaning the indices in the query) and the embedding model parameters.
- (2) For a given embedding model, the average load latency could vary significantly based

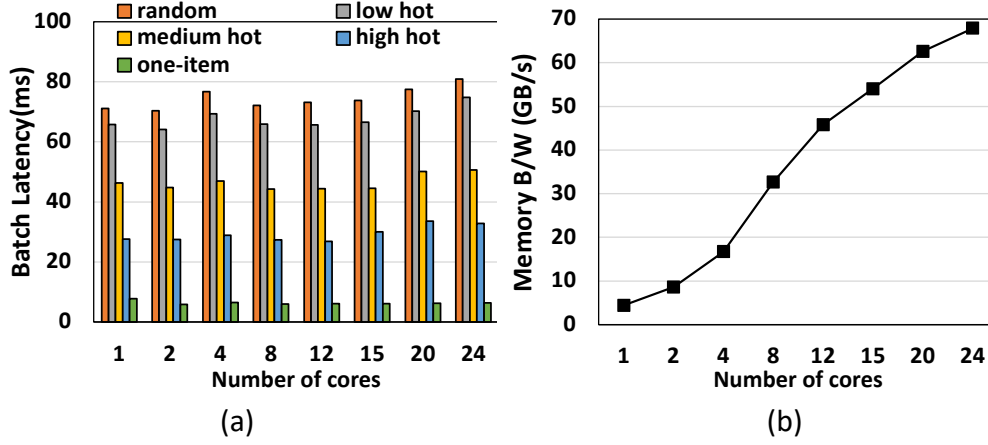


Figure 2.8: Multi-core scalability: (a) Execution time (b) Memory bandwidth used.

on lookup patterns (up to 16 times), as shown in Figure 2.4.

(3) Reuse distance calculations are representative of the memory working sets. For large DLRMs, caches (even as large as 768MB) fail to capture them, particularly in the practical Medium and Low Hot dataset scenarios.

(4) Significant amount of cold misses are observed (as high as 72%). Even, the High Hot dataset experiences about 22% cold misses on average.

(5) The embedding stage contribution in the end-to-end performance varies based on the model properties and input dataset.

(6) Memory-bandwidth and core-count in modern CPUs [79, 82, 83] has significantly increased compared to CPUs evaluated in past works [2, 15]. This poses a need to rethink multi-core scaling and application mapping in DLRMs.

(7) Underutilization opportunities are present in MLP and embedding stages. Caches and memory are underused in MLP while the CPU pipeline is stalled in embedding accesses.

Based on these observations, in the remainder of this paper, we explore two well-known design features – software prefetching and hyperthreading – to boost the performance of the embedding stage.

2.4 Proposed Optimizations for Improving the Efficiency of CPU Pipeline

Since there are three levels of indirection in memory accesses (Figure 2.3), and the reuse distance easily exceeds the cache capacities (Figure 2.7), the long memory access latency becomes a major bottleneck. In this section, we first identify the limitations of existing hardware techniques that hide the memory access latency, and then propose programmer-

inserted software prefetching to reduce the latency on the critical path. Furthermore, given that load instructions still exist, and that the Bottom-MLP and embedding stages are independent tasks, we propose to intelligently exploit hyperthreading to overlap computation and memory accesses.

2.4.1 Limitations of Existing On-Chip Hardware and Compiler Techniques

Modern CPUs are equipped with various techniques to deal with long memory latencies. OoO (Out-of-Order) execution is one such technique that is commonly used to tolerate long latency but it is limited by the instruction window size [84]. The cache hierarchy and prefetching also help tolerate memory latency. As the memory accesses offer limited temporal locality, we see that even in the presence of a cache hierarchy, average load latency is still quite high (Figure 2.4).

Intel CPUs are typically equipped with a total of four prefetchers [66] dedicated to each core where two work with L1D\$ and the other two work with L2\$.¹ Since there are indirect accesses, it is hard for the prefetcher to detect the patterns effectively, and load latency remains high. Our evaluation found that turning the hardware prefetcher on or off have a small performance impact on DLRMs, sometimes even giving slight speedup (Section 2.6).

Compiler-inserted prefetching is another alternative to capture the indirect memory accesses automatically using compilers. While optimizing compilers are able to capture "regular" stride patterns easily [72, 73], they still suffer for DLRM workloads as the access pattern is highly random and input-dependent. We evaluated the benefits of compiler-inserted prefetching and hardware prefetching on a 24-core configuration for the RM2_1 model and the execution time results are reported in Figure 2.10(a) with respect to the baseline results (hardware prefetcher on). As the results indicate, these off-the-self techniques² show limited benefits, or even marginally degrade the performance. The result motivates us to design application-specific prefetching techniques to address the long memory access problem.

¹By default, all the hardware prefetchers are enabled.

²gcc is used with "-fprefetch-loop-arrays", and icc is used with "-qopt-prefetch=5"

2.4.2 Application-Specific Software Prefetching

Plainly employing state-of-the-art software or hardware prefetchers [62–66, 72, 73] could be naïve and costly [85]. Thus, smartly tailoring the available software prefetching intrinsics for DLRM and associated hardware is necessary. In this section, we discuss our proposed programmer-inserted prefetching support which is both application- and architecture-aware.

Specifically, to do a design space exploration, we set to answer the following questions in the context of prefetching for embedding table lookups: i) *What to prefetch?* ii) *When to prefetch?* iii) *How to prefetch?* and iv) *Where to prefetch?*

What to prefetch? We leverage the insight that the software can look ahead in the indices array (Figure 2.3) and easily calculate the addresses for future accesses. Thus, while the demand load fetches an embedding row vector, we anticipate a load miss in the L1D\$ for a later sample, for which a corresponding software prefetch request is issued. That is, we put a software prefetch instruction for each data request we predict to result in a miss.

When to prefetch? Earlier, Algorithm 8 describes the memory access loop in the embedding lookup stage. This suggests multiple options to prefetch where one can consider prefetching the embedding vectors for the next batch or table or sample or lookup. Since the embedding vector is comprised of multiple cache lines, one can even consider prefetching a part or all of them. This presents a conundrum of choice(s). One can find the right option by playing with the prefetch distance. Intuitively, prefetching for the next cache lines of the same embedding vector seems to be too early, and prefetching for the next sample seems to be too late. So, we consider finding the prefetch distance at the granularity of lookups to look-ahead, which can meet the timeliness requirement. Figure 2.10(b) shows the execution time vs prefetch distance. We find that a prefetch distance of 4 gives the best performance. It corresponds to about 200 instructions between look-ahead prefetch and demand load.

How to prefetch? We use the intrinsic “`_mm_prefetch`” [86] supported by gcc to manually insert prefetches. It fetches a cache line for the input address. This allows bringing the data of embedding row vectors into the caches.

Where to prefetch? The intrinsic provides multiple options to suggest where to load the prefetched data. It can be put into either L1D\$, L2, or LLC based on the strategy macro passed. We choose to pick L1D\$ as it brings the data closest to the processor. However, since the L1D\$ is only 32KiB, one must be cautious about potential cache pollution. A 32KiB capacity can hold up to 64 embedding vectors (assuming an

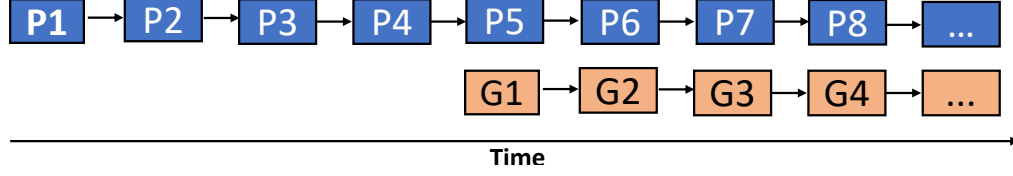


Figure 2.9: Software prefetching timeline: Gather[G] (the demand loads) follow the look-ahead loads[P] at an ideal prefetch distance of four.

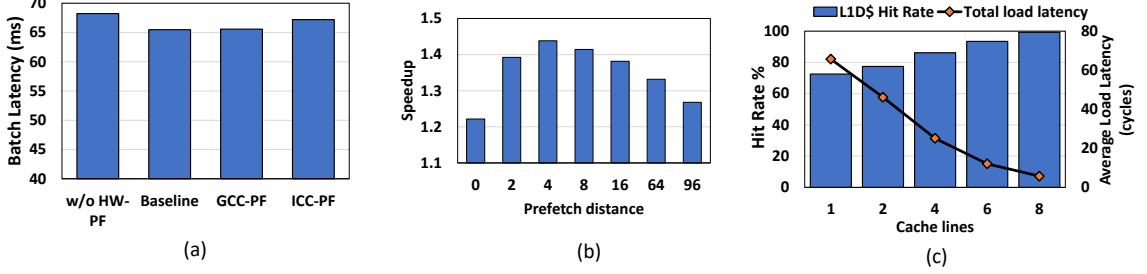


Figure 2.10: (a) Comparing state-of-the-art compiler-inserted prefetching with baseline for the RM2_1 model, (b) Impact of prefetch distance on the performance of 24 cores, (c) Impact of prefetch amount on the L1D\$ hit rate and average load latency.

embedding dimension of 128, and 32-bit floating point precision). Since the prefetch distance decides how many embedding vectors are fetched between the prefetch and the demand load, a distance of four means $4 \times 512\text{B} = 2\text{KB}$ amount of prefetch injections, which is reasonably low compared to the L1D\$ cache capacity. Figure 2.10(c) shows the impact of the prefetch amount on the cache hit rate and average load latency. Since an embedding vector occupies 8 cache lines, we find that prefetching the complete embedding vector gives the highest cache hit rate and minimizes the load latency. Thus, we choose a prefetch amount of 8. Algorithm 6 captures the above points of discussion and illustrates the prefetch implementation.

To better visualize software prefetching, we can look into the timeline of look-ahead loads and demand loads, shown in Figure 2.9. It can be observed that the look-ahead loads match, one-to-one, to the demand loads, and consequently, the total number of loads requested by the CPU core becomes doubled. However, the look-ahead load pays the memory access latency and brings the data into L1D\$. Therefore, the critical path for demand load is highly reduced to only a few cycles based on the L1D\$ latency.

As the number of loads increase due to prefetching, the cache and memory bandwidth gets more stressed. Also, when multiple cores are working, the situation could worsen. To alleviate the stress, we additionally leverage hyperthreading to overlap compute- and memory-intensive stages.

Algorithm 6: Prefetch Insertion in `embedding_bag` operator.

```
pf_dist ← 4
pf_blocks ← ddim/16
for i ← start to end do
    output_ptr ← &output[i * ddim]
    zero_ker(output_ptr, ddim);
    for s ← offsets[i] to offsets[i+1] do
        row_ptr ← &table[indices[s] * ddim]
        pf_ptr ← &table[indices[s + pf_dist] * ddim];
        for cb ← 0 to pf_blocks do
            _mm_prefetch(pf_ptr + cb * 16, _MM_HINT_T0);
            add_ker(output_ptr, row_ptr, ddim);
```

2.4.3 Application-aware HyperThreading

Multithreading is a common technique to improve application performance. But, it could hurt performance if not carefully applied. A previous work [15] used hyperthreading to run two DLRM inference instances on the same core and showed it to be detrimental to performance. Many other previous works [2,15,19,67] dismissed hyperthreading for similar reasons. In contrast to these works, we realize the additional knowledge of the embedding stage and Bottom-MLP as independent tasks is a fortunate opportunity to deploy hyperthreading smartly. It is well-known that modern CPUs employ a Simultaneous Multithreading design [87] like HyperThreading [88] in Intel CPUs.

Figure 4.6 shows three possible designs: sequential (baseline), naive hyperthreading referred to as Data-Parallel HT (DP-HT), and Model-Parallel HT (MP-HT). Sequential refers to the commonly adopted designs in DLRM [2,25,89] where Embedding is followed by Bottom-MLP. DP-HT implements mapping two inference instances on the hyperthreads. These threads are executed sequentially and are colocated on the same core. This could lead to memory-memory or compute-compute overlap in the CPU pipeline, and thereby may worsen the performance of each thread. Luckily, we have embedding and Bottom-MLP as independent stages, and they can be executed in parallel. MP-HT partitions the work of one batch such that it maps the embedding stage on one thread, and Bottom-MLP on the other. This fine-grained control is ideal for hyperthreading because of a favorable memory-compute overlap scenario and minimal cache/bandwidth contention across threads due to smaller Bottom-MLP working sets. One could even think to deploy the independent Bottom-MLP and embedding threads on separate CPU

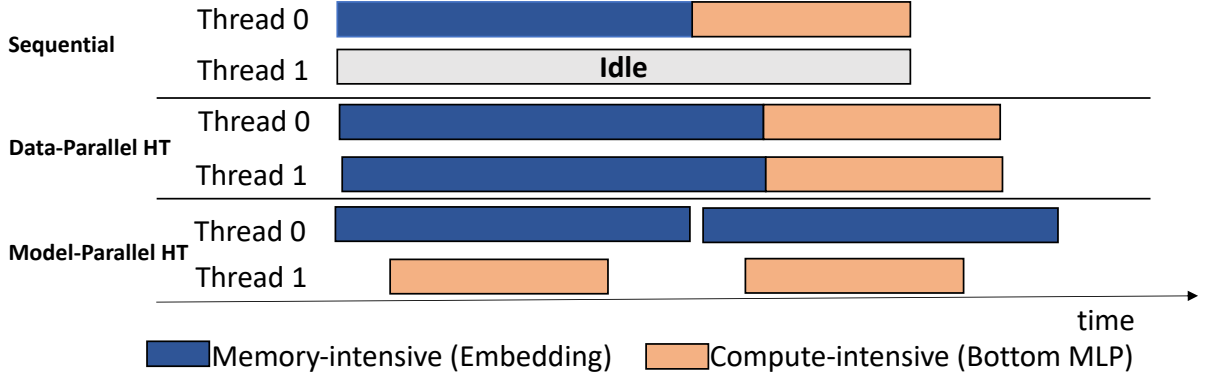


Figure 2.11: Overlapping compute and memory access via hyperthreading.

cores. However, it would cost double the CPU cores, and synchronization overheads. Among possible choices, we argue that MP-HT is the best scheme for performance gains.

To achieve this goal, we modify the thread pool implementation in PyTorch. PyTorch supports multi-threading inference, where each thread from the thread pool pulls an inference operator from the task queue. Our hyperthreading design revises the PyTorch thread pool implementation to be hyperthreading-aware. Specifically, two threads on the same physical core share one task queue instead of sharing with the entire thread pool. Thus, one inference instance will always run on the same physical core, and other threads on other physical cores cannot steal the inference task.

2.4.4 Synergy of Prefetching and Overlapping

Despite prefetching, there could still be miss related stalls. Further, sometimes prefetching itself adds to the memory traffic to worse performance [90]. Hyperthreading, on the other hand, is orthogonal to prefetching and can more effectively use on-chip compute resources when they are under-utilized by one of the threads stalling for memory [74, 87, 91]. To further improve the CPU pipeline utilization, we propose combining our proposed schemes. This is interesting for two reasons: (1) Since the prefetching would improve the performance of the embedding thread, the memory instructions would earlier free the pipeline resources like instruction widow entries, thus providing more resources for the Bottom-MLP thread. (2) As the Bottom MLP stages are compute-heavy and only require a few MBs of memory footprint [61], prefetching could better utilize the cache and memory resources. Thus, we argue that hyperthreading can work symbiotically with prefetching and improve overall performance by higher CPU pipeline and cache/memory utilization.

Machines	Cascade Lake
Model	6240R
Frequency	2.4GHz
Sockets	2
SIMD	AVX-512
L1D Cache latency	5 cycles
L1D Cache size	32KB
L2 Cache size	1MB
L3 Cache size	35.75 MB
DRAM Capacity	96 GB
DDR Type	DDR4-2933
DDR B/W per socket	140 GB/s

Table 2.3: CPU configuration parameters.

2.5 Methodology

Datasets: We use the recently released production traces from Meta [60], which provides the input data for embedding table lookups (indices and offsets) and has similar memory access patterns to Meta’s DLRM traces. The hot embedding behavior differs across tables within a dataset, and we categorize them into three groups based on the hotness: low, medium, and high. Unique accesses represent % of unique item ids present in a given dataset, which governs the memory working set for an embedding table. We found that the unique accesses in Low, Medium, & High are 60%, 24%, & 3% respectively, which matches Meta’s input traces [15, 57].

Models: As highlighted in Section 5.2, [15] and [2] have categorized the DLRMs into three classes. RMC1 and RMC2 are embedding-dominated, whereas RMC1 has a reasonable contribution from the MLP stages. As our work is focused on embedding heavy models, we consider models from these two classes. One issue is that limited model configurations are available in public where [2] is the only work that shares the exact details of three models (one in each class). Since recent models have drastically grown in various configurations, the model configurations provided in [2] may not be accurate. Therefore, we use the guidelines from [15] and [2] on model configurations to project practical models adhering to the upper bounds mentioned in the recent works [50–53]. We vary the model configurations to project larger RMC1 and RMC2 models. To do this, we increase the embedding dimension, rows, tables, lookups, and make the Bottom/Top-MLP larger. Bottom-MLP is kept larger than Top-MLP as suggested in [67, 92, 93]. The adopted model configuration details are given in Table 2.2. The batch size is chosen as

	Low				Medium				High			
	RM2_1	RM2_2	RM2_3	RM1	RM2_1	RM2_2	RM2_3	RM1	RM2_1	RM2_2	RM2_3	RM1
HW-PF OFF	72.59	180.42	306.77	11.23	48.94	115.76	196.93	7.33	32.92	83.18	126.54	5.85
Baseline	74.36	180.88	303.56	10.95	49.65	120.48	201.87	6.62	29.89	70.28	124.84	4.68
SW-PF	51.91	129.61	232.79	9.14	36.74	90.88	146.39	5.31	24.43	60.65	99.26	3.95

Table 2.4: Batch execution time (ms) of embedding-only stage in multi-core for different models and datasets.

64 to maximize throughput while meeting the SLA requirements associated with each class as mentioned in [2] (Table 2.1). In our evaluation, we refer to RM2_{1,2,3} as embedding heavy models and RM_1 as mixed-model.

Hardware: Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz (Cascade Lake) [78] is used in the evaluation. Table 2.3 summarizes the properties of the CPU. We collect results on a single-core and multi-core(using all cores) configuration. Turbo-mode is disabled and frequency is set to 2.4GHz for all cores.

Software: PyTorch(v1.12.0) is used combined with Intel’s extension for PyTorch (IPEX) [80], which provides optimized operators and is used in MLPerf benchmarking [89, 94]. Our schemes are built on top of IPEX. The batch size used is 64 which meets the SLA latency target shared in [2].

2.6 Performance Evaluation

In this section, we analyze the performance benefits of our optimization designs. Our experimentation configuration is described in Section 5.5. We report the inference performance for single and multi-core, in terms of batch latency (averaged over 120 batches) and speedup. Similar to [2, 26], one batch is treated as a quantum of work, and each batch is mapped onto a core. Multiple cores are used in inference for maximizing inference throughput and CPU utilization. Note that we use all the physical cores present in one socket. We assess various design points: (1) w/o HW-PF: hardware prefetching turned OFF; (2) baseline: hardware prefetching turned ON; (3) SW-PF: proposed software prefetching turned ON; (4) DP-HT: naive hyperthreading; (5) MP-HT: proposed smarter hyperthreading (6) Integrated: the combination of SW-PF and MP-HT. As described in Section 5.5, we consider three datasets {High, Medium, Low} and three models RM2_1, RM2_2, RM2_3 for our evaluation.

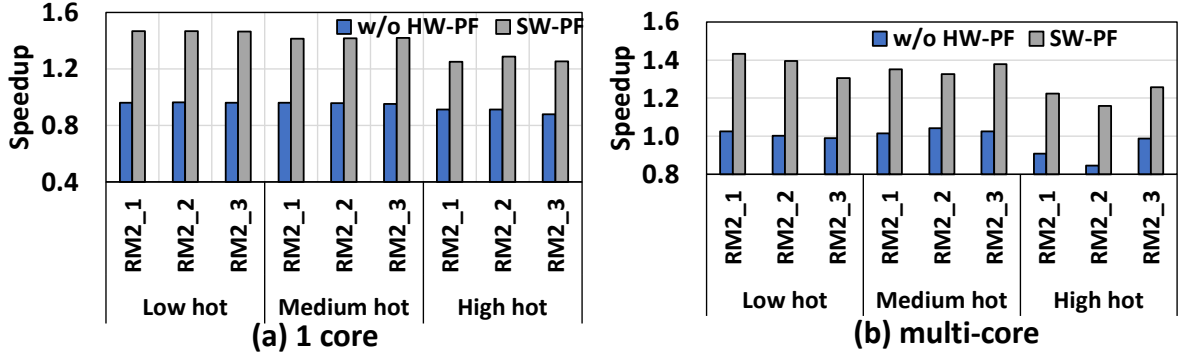


Figure 2.12: Comparison of performance of **embedding-only** stage for **embedding heavy models** for different datasets: (a) single and (b) many cores.

2.6.1 Improvement in the Embedding stage

We first evaluate the performance benefits of the embedding stage since the software prefetching design optimizes only the memory accesses in embedding lookups. Figure 2.12 shows the performance of the design points for a single and multi-core execution.

Compared to the baseline, disabling the hardware prefetcher generally yields slightly reduced performance except in the High Hot case, where performance improves. Our proposed software prefetching is seen to outperform the baseline. Specifically, in the single-core comparison, it gives a speedup ranging from 1.25x (RM2_1 in High Hot) to 1.47x (RM2_3 in Low Hot), while for the multi-core comparison(Figure 2.12(b)), it gives a speedup in the range of 1.16x (RM2_2 in High Hot) to 1.43x (RM2_1 in Low Hot). This shows that software prefetching is scalable to multi-core systems. As the degree of cold misses and cache performance worsens from high hot to low hot (Section 2.3.1), it is observed that software prefetching performs best in the Low Hot dataset as it offers more irregularity, while still proving to be beneficial in High Hot situations. We observe similar performance gains across models owing to them having the same embedding dimension.

2.6.2 End-to-end Improvement

Here, we consider full DLRM and further evaluate the MP-HT and Integrated schemes. We test both embedding-heavy(RM2_{1,2,3}) and mixed models(RM_1).

2.6.2.1 Embedding Heavy Models

Similar to Figure 2.12, Figure 2.13 compares the performance of all design points for different datasets, models, and cores. First, we observe that turning off hardware

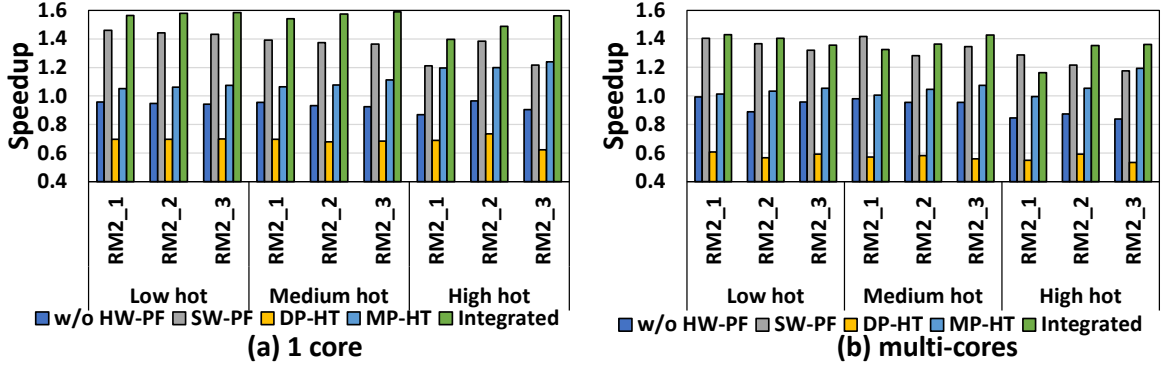


Figure 2.13: End-to-end performance gain on **embedding heavy models** for different datasets.: (a) single and (b) many cores.

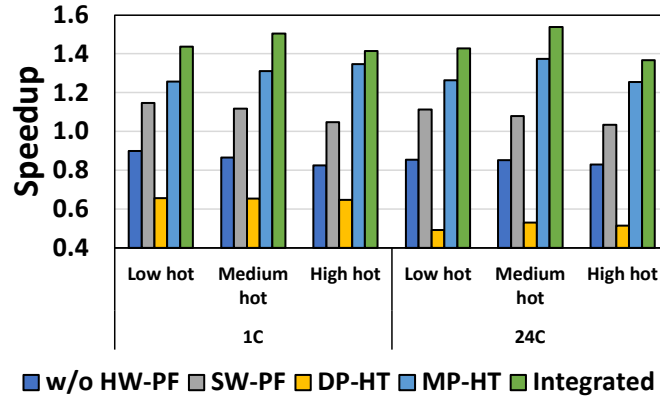


Figure 2.14: End-to-end performance gain on **mix-model** RM_1 for different datasets. prefetching hurts performance in all cases when compared to the baseline. This is because hardware prefetching is useful in the compute-intensive stages as they bring regular access patterns. Next, we see that DP-HT underperforms compared to our baseline (up to 0.62x) and it exceeds the SLA requirement by 152ms in RM2_3 for the low hot dataset.

SW-PF outperforms the baseline by a considerable 1.21x-1.46x in single-core and 1.18x-1.42x in the multi-core scenario. MP-HT yields a speedup of up to 1.24x (in RM2_3 in High Hot) in single core, and up to 1.19x (in RM2_3 in High Hot). Even though all models are heavy in embedding, RM2_3 provides a relatively higher opportunity for overlap. Moreover, as the hotness increases, the embedding stage executes faster, thereby improving the scope for overlap further. Thus, MP-HT is seen to perform relatively better for High Hot and models with relatively lighter embedding. Finally, we see that our integrated design outperforms the baseline by 1.40x-1.59x in single-core and 1.29x-1.43x in multi-core. Table 2.4 details the actual execution time for embedding-only multi-core case.

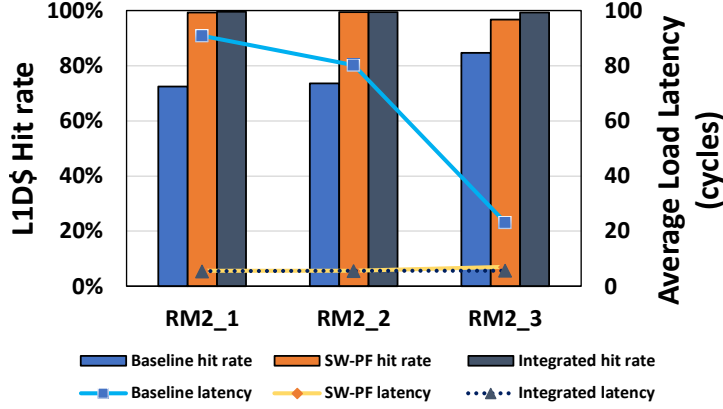


Figure 2.15: L1D\$ hit rate and average load latency for RM2_1 model (embedding-only) on Low-Hot dataset.

2.6.2.2 Mixed Models

Figure 2.14 shows the performance of all design points for the RM_1 model. On average, HP-OFF, and DP-HT exhibit performance degradation of 0.85x, 0.50x, and 0.60x, respectively. Software prefetching yields an average speedup of 1.1x. This is relatively low compared to that of embedding heavy models due to reduced irregularity and the lower contribution of embedding. MP-HT produces a speedup of 1.25x-1.37x. This is higher than that of embedding heavy as RM_1 models provide a better overlapping opportunity of embedding and Bottom-MLP stages. Interestingly, the integrated design gives a considerable non-linear speedup ranging from 1.37x to 1.54x due to the synergy as described in Section 2.4.4.

2.6.3 Empirical Characterization

Using VTune, we profile the embedding stage to obtain low-level CPU metrics. Figure 2.15 shows the L1D\$ hit rate and average load latency for RM2_1 model. The baseline observes an L1D\$ hit rate between 72%-84%, and an average load latency between 23 to 90 cycles. Across models, the hit rate increases, and thereby the load latency decreases. This is because larger models (RM_2, RM_3) offer more accesses per table due to higher lookups per sample. This situation is similar to Intra-table discussion in Section 2.3.1. Software prefetching improves significantly over baseline by achieving a cache hit rate of 96.7% to 99.4%, and an average load latency of 5.6-7.1 cycles. Integrated improves over SW-PF by achieving a cache hit rate of 99.3% to 99.5%, and an average load latency of 5.5-5.7 cycles. We observe similar results for other datasets, but for brevity, we analyze

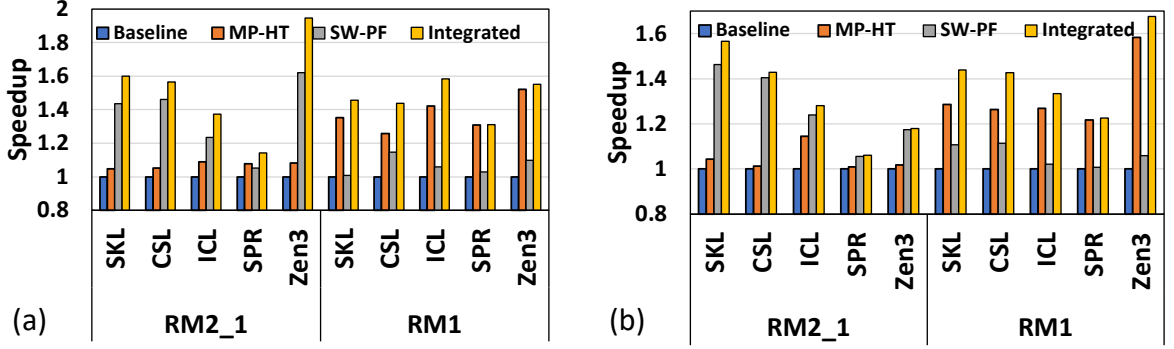


Figure 2.16: Speedup comparison across different CPUs for (a) single-core (b) multi-core only the Low Hot dataset. This shows that our optimization designs are able to improve the long memory latency bottleneck, thereby improving the CPI and thus the execution time (as discussed in the previous subsection).

2.6.4 Evaluating Across Different CPU Platforms

Data centers contain a range of CPU platforms [15] [2] [19] [47] [95]. To understand how our optimizations perform across different CPU architectures, we consider 5 different CPUs: Intel’s SkyLake(SKL, 24 cores) [96], Cascade Lake(CSL, 48 cores) [78], IceLake(ICL, 32 cores) [97], Sapphire Rapids (SPR, 112 cores) [82], and AMD’s EPYC 7763(Zen3, 128 cores) [83]. To evaluate both embedding heavy and mixed-model situations, RM2_1 and RM1 are tested on the low hot dataset. Figure 2.16 shows the performance of our proposed optimizations across different CPU platforms, for single-core (Fig. 2.16(a)) and for multi-core (using all CPU cores, Fig. 2.16(b)) setups. Our optimizations consistently improve the performance over the baseline across a wide range of CPUs.

It is observed that multi-core speedups are lower than their respective single-core performance. This is because of interference across cores for the shared resources. For example, Zen3 observes high speedups except for RM2_1 in (a) due to severe contention in memory bandwidth with 128 cores running in parallel. Overall, compared to CSL, the lower performance of ICL and SPR is due to microarchitecture differences. In particular, ICL & SPR have a higher instruction window width by 58% & 129% respectively which implicitly improves the memory-level-parallelism(MLP). Note that we tuned the prefetcher to extract optimal performance for ICL, SPR, & Zen3, and the optimal prefetch amount of 2, 2, & 4 is found. It is interesting to note that though hardware implicitly improves MLP using various techniques (OoO, MSHR, etc), software prefetching remains

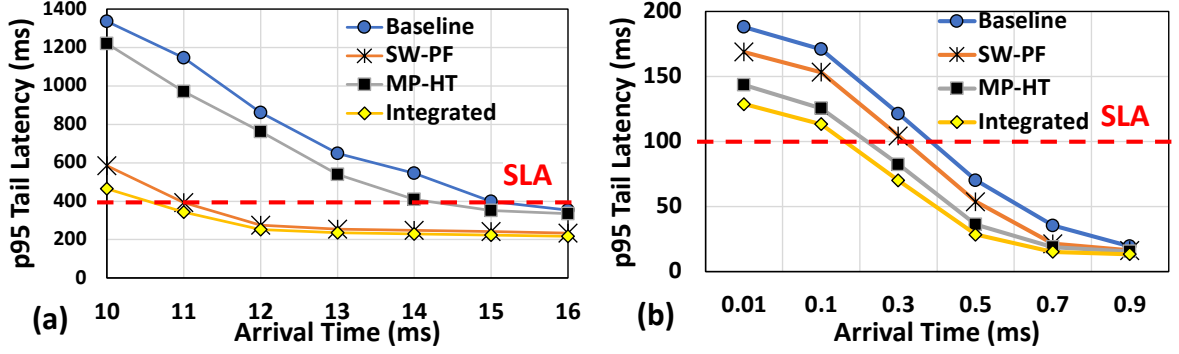


Figure 2.17: Tail (P95) latency comparison across designs by varying the arrival time for models: (a) RM2_1 (b) RM1. SLA latency target is marked in red.

effective, and the integrated scheme reaps the best performance.

2.6.5 Tail Latency Evaluation

As recommendation systems are deployed as part of interactive applications, it is critical to evaluate the SLA and tail latency behavior of such systems [2, 19, 98]. Similar to [2], we model a load generator that generates requests with a Poisson distribution. The end-to-end inferences for mixed-model RM1 and a heavy-embedding model RM2_3 on the low hot dataset are evaluated. Figure 2.17 plots the p95 tail latency for various schemes. Arrival times are carefully chosen to consider both SLA-compliant and saturation regions. The SLA-compliant region refers to the range of arrival times for which the p95 latency meets the SLA target, while the saturation region refers to the non-compliant range. The baseline design meets the SLA targets for an average arrival time greater than 15ms for RM2_1, and 0.5ms for RM1. For these arrival times, our optimization designs considerably improve the tail latency by up to 1.8x for RM2_1, and 2.5x for RM1. Further, the integrated design brings significant improvements by tolerating faster arrival rates upto 1.4x for RM2_1, and 2.3x for RM1 while meeting the 400ms and 100ms respective SLA targets. These improvements can be attributed to the fact that our designs improve the inference computation time and thereby, less suffer from queueing delays.

2.7 Conclusions

As CPUs continue to be a preferred platform for running DLRM inference workloads, improving the performance of these emerging workloads on single and multicore CPUs

becomes critical. In this context, optimizing the embedding stage, which is the major bottleneck in the DLRM pipeline, is especially important. While prior efforts have proposed several techniques to enhance DLRM performance primarily through heterogeneous CPU+GPU platforms and NMP accelerators, this paper focuses on improving the performance on state-of-the-art CPUs by utilizing well-known software techniques. We propose customized software prefetching and computation overlapping via hyper-threading to minimize the embedding stage overhead and overall execution latency. Our experimental results show that the proposed techniques improve the performance of embedding heavy & mixed models for various hotness datasets by up to 1.59x. We believe that our techniques can be implemented straightforwardly in CPUs for making them a competitive platform for DLRM inference. Finally, with these two enhancements, requiring no change in hardware or in the models publicly distributed, we are getting on an average 40% improvement. This sets a new baseline for any future research proposing hardware and software enhancements to improve upon.

Chapter 3 |

Parallelization of Embedding Stage on CPUs

3.1 Introduction

Building on the CPU performance optimizations discussed in the previous chapter, we now shift our focus toward improving the parallel efficiency of the embedding stage itself. While Chapter 2 addressed memory bottlenecks through prefetching and hyperthreading which improve single core performance, this chapter investigates how to better distribute embedding workloads across multiple CPU cores to enhance scalability and end-to-end performance. Recall the DLRM model published by Meta [14] as depicted in Figure 3.1. This model has MLPs (Multi-layer perceptron), Embeddings, and an Interaction Layer, each having unique characteristics. Specifically, the top and bottom MLPs are compute-dominated. In contrast, the embedding stage is memory bandwidth- and memory capacity-dominated. DLRM parallelization is carried out using both “data parallelism” (MLP layers) and “model parallelism” (embedding stage). Even though efforts of tolerating the overhead of the embedding stage while parallelizing the MLPs have been successful [99] [100], a significant portion of DLRM inference cycles is still spent in the embedding stage. Hence, innovations in improving the performance of embedding stage can have a non-trivial impact on the overall DLRM inference latency. This in turn can lead to improvements in Queries-per-Second (QPS) while meeting the Quality-of-Service (QoS) requirements. Another benefit of improving the performance of the embedding stage is to enable larger recommendation models that meet the QoS requirements. Doing so would lead to higher accuracy of prediction and thus higher user satisfaction.

State-of-the-Art: One of the current strategies frequently used by hyperscale companies is to employ GPU-based systems with HBM (high bandwidth memory) for

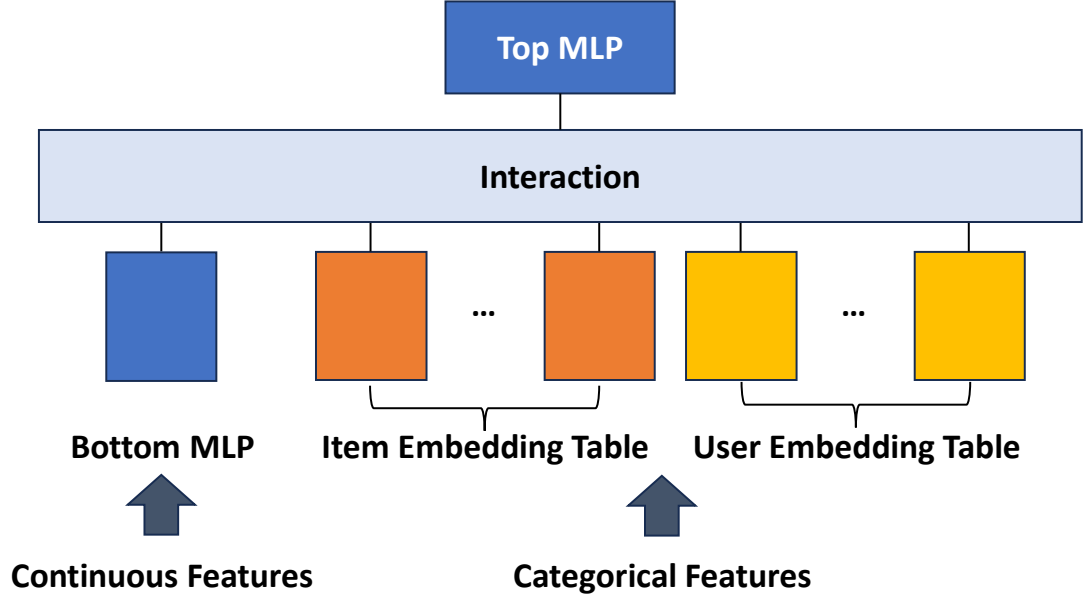


Figure 3.1: DLRM architecture. MLPs are compute-bound and are typically parallelized using data parallelism. On the other hand, embedding tables are memory bandwidth- and memory capacity-bound and are parallelized using model parallelism.

DLRM inference. In particular, GPUs provide higher matrix-multiply TOPS (trillions of operations per second) that benefit the compute-bound layers of DLRM. Furthermore, HBM memory provides higher bandwidth for processing the embedding stage, which, as mentioned above, is memory bandwidth-bound. However, HBM configurations provide lesser memory capacity compared to DDR-based ones and at a higher cost. Since embedding tables in DLRM can be quite large, a recommendation model must be *split* across a number of GPUs to accommodate the embedding tables in the available HBM capacity.

Motivation: Our goal is to enable scalable inference with deep learning recommendation models that utilize very large, sparse embedding tables. Our proposed approach is motivated by two observations:

- Firstly, while DLRM can be accelerated by GPUs or custom accelerators, CPUs can be a more “cost-effective” platform and easier to optimize DLRM compared to other target architectures. For example, popular Hyperscalers such as Google Cloud Platform [101], Amazon Web Services [102], Azure Cloud [103], etc., offer CPU processors with competitive performance per \$ that can host recommendation systems. So, from a cost-effectiveness angle, there is a motivation to consider and significantly improve performance of CPU-based implementation of DLRM (in particular, the embedding bag operator).

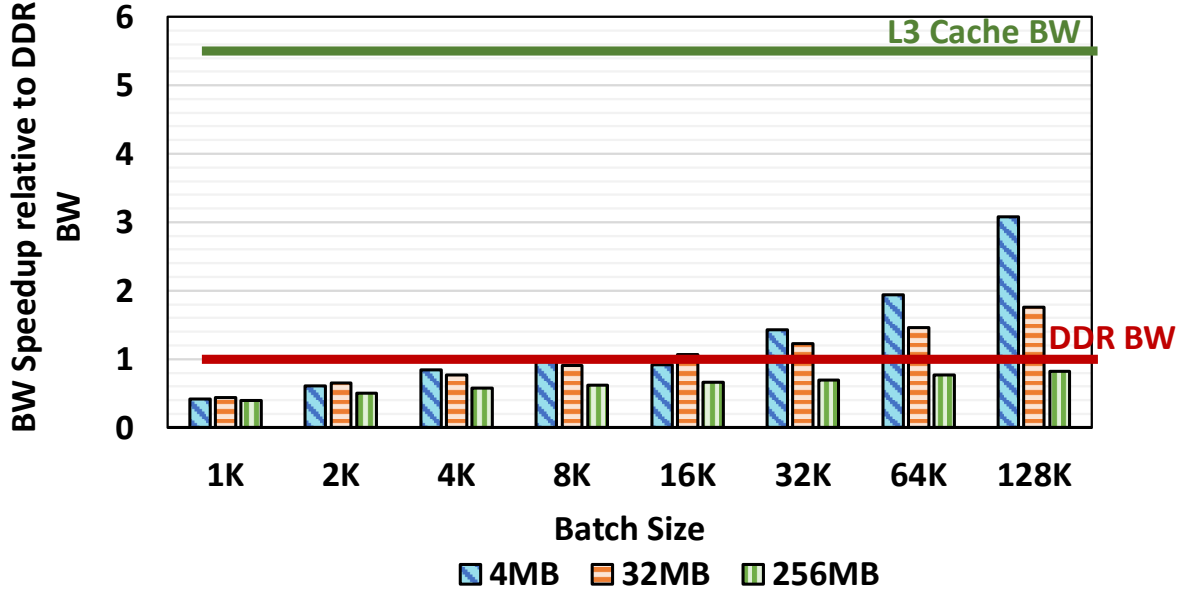


Figure 3.2: Achieved bandwidth (data processing rate), normalized to peak DDR bandwidth, when using Batch Threading (BT) with different table sizes (x-axis) on a 4th Gen EPYC™ processor. The achieved bandwidth is only a fraction of the theoretical LLC bandwidth for all table sizes, and is lower than the peak DDR bandwidth for some table sizes.

- Secondly, it is known that DLRM exhibits a large degree of data reuse when accessing the embedding tables and the majority of the table indices follow a Zipf’s law distribution [104]. Preliminary data for leveraging this data reuse for select hot tables has been shown to provide good performance for embedding operations [99]. However, the embedding stage remains a memory-bound kernel as shown in Figure 3.2 which plots the achieved data processing rate (called “bandwidth”) for the embedding stage employing different table sizes (bars) and batch size (x-axis). The results in this graph reveal that the achieved bandwidth is at least 43.6% lower than the peak LLC bandwidth and in some cases 40% lower than the peak DDR bandwidth, indicating a large amount of remaining memory traffic. Thus, it is important to consider CPU-based optimizations for the embedding stages. Note also that the state-of-the-art embedding bag parallelization method on CPU multicores, called “batch threading” (BT), which parallelizes accesses to each embedding table across all available threads (more on this later), suffers from lack of scaling for heterogeneous workloads. This is illustrated in Figure 3.3) on a server-grade CPU multicore system for embedding stage running with varying table properties (heterogeneous). It can be seen that batch threading does not scale beyond 16

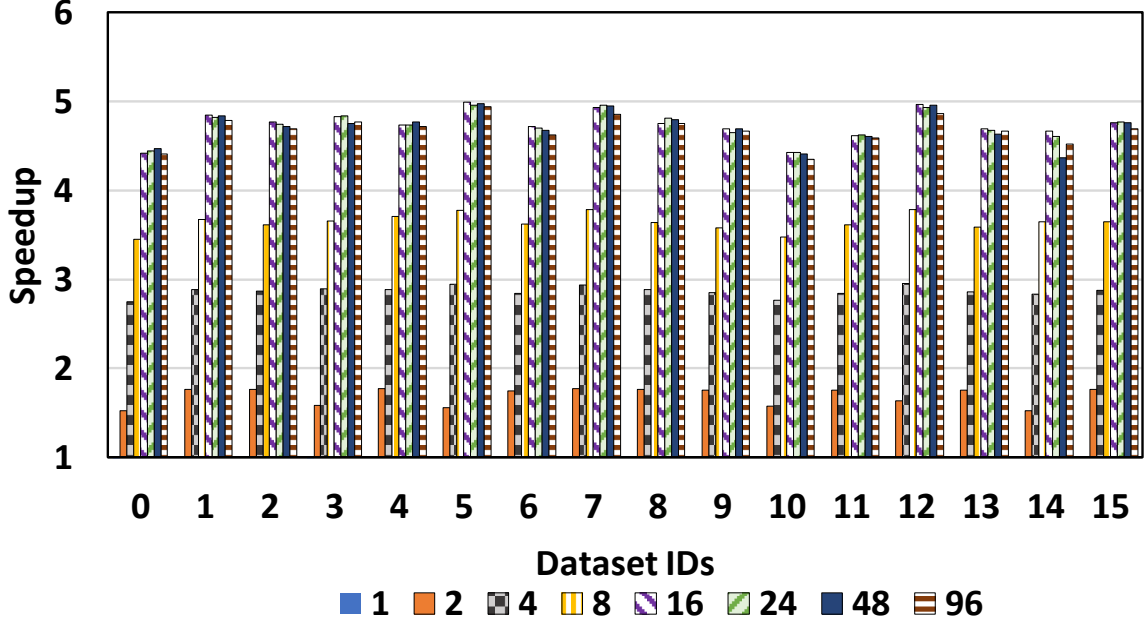


Figure 3.3: Scalability of BT, with heterogeneous (varying table sizes) datasets, on a 4th Gen EPYC™ processor. Each point on the x-axis corresponds to a different dataset (with a different table size) and each bar corresponds to a core count. Note that performance tends to saturate beyond 16 cores.

cores. This poor scalability results clearly motivate for exploring alternate parallelization strategies that will enable scaling across all available compute (CPU cores).

Contributions: Motivated by the observations above, this paper makes the following novel contributions:

- We leverage the cache hierarchy and chiplet-based architecture found in state-of-the-art server-grade AMD CPU multicores to propose different parallelization strategies for the embedding bag operator, that enable scaling and improve data processing rate. Our defended parallelization strategies, referred to as “Table Threading” (TT) and “Hierarchical Threading” (HT), leverage the high data reuse in embedding tables to improve the utilization of L1, L2 and L3 (LLC) cache capacities, rather than relying on the available DRAM bandwidth.
- We discuss the role of various hardware and software parameters in maximizing the performance of our proposed HT, and also provide detailed reasoning on how these settings help.
- We present comprehensive performance metrics based on the statistical distribution properties of the Zipf’s distribution.
- We provide a comparison of our proposed parallelization strategies on state-of-the-

art CPU multicores, and discuss the scenarios under which the different parallelization strategies perform better than the others. Specifically, the experimental evaluations indicate that our defended embedding bag parallelization scheme achieves, on 4th Gen EPYC™ processors, up to 9.9x improvement in the embedding performance over the state-of-the-art.

Novelty: Prior work on accelerating DLRM Inference on CPU multicores [99] proposed parallelization at coarse levels of granularity: (a) executing bottom MLP in parallel to the embedding stage and (b) executing an entire batch per CPU core for the embedding stage. In this paper, we focus on finer granularity parallelization strategies that better exploit the underlying architecture: (i) we propose novel parallelization schemes that exploit a chiplet-based multicore architecture consisting of core complexes, each with a shared LLC, by pinning the per-table or per table-group memory traffic onto CPU cores or Core complexes; (ii) we study the scaling of the embedding stage on production datasets, which combines the heterogeneous aspects in the embedding stage such as non-uniform table sizes, varying number of lookups per user, and different row access distribution across tables; (iii) we identify embedding bag configurations (based on batch size, table size, embedding dimension, and row reuse distribution), scaling when running on state-of-the-art CPU multicores; and finally, (iv) we identify the performance scaling impact of 3D stacked LLC on the embedding bag operator.

3.2 Background

This section describes our target architectures (4th Gen EPYC™ processors technology (formerly known as “Genoa”) and 4th Gen EPYC™ processors with 3D V-Cache™ technology (formerly known as “Genoa-X”)), and the system parameters that play a crucial role in the performance of the embedding bag operator.

3.2.1 Understanding Embedding Bag Operator

A typical Embedding Bag is illustrated in Figure 3.4. Each embedding operator does a series of vector reads into a table. The width of each of the vector is called “Embedding Dimension”. A “Pooling Factor” number of vectors are then *pooled* together to form a single "output" vector. Multiple such embedding operations are batched together for each table (note that the size of the pooling factor can vary across batches). This operation is then repeated for multiple tables. In general, the table size can also vary [33]. Current

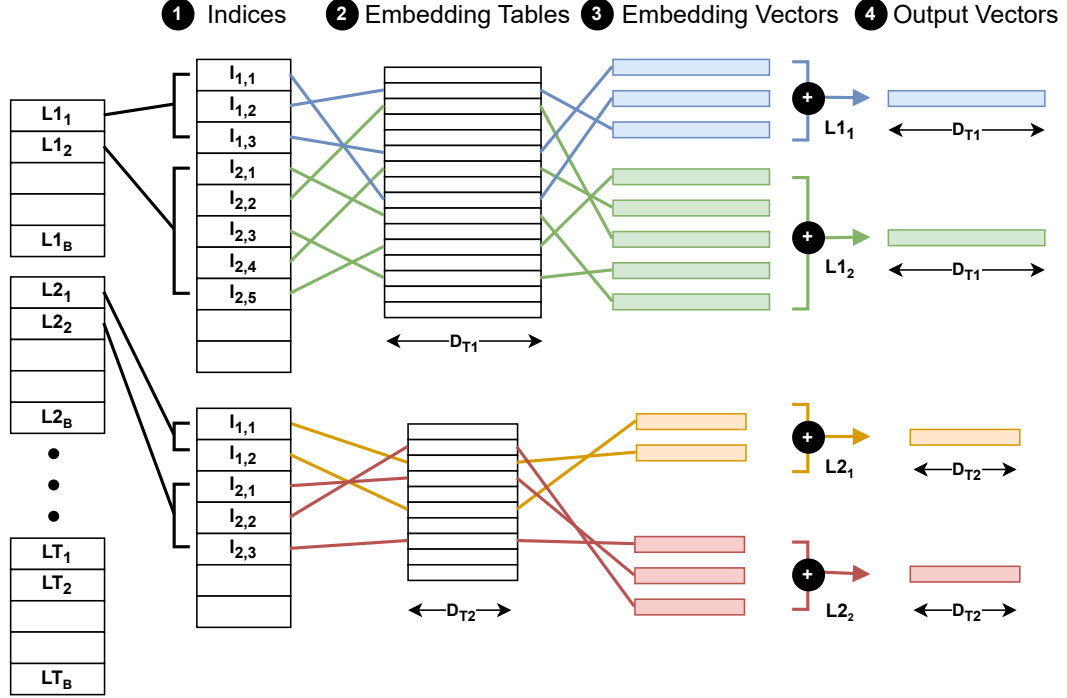


Figure 3.4: Workflow of the embedding bag operator. For each table, a batch of queries is processed. The operator initially performs index lookups for each query item, followed by mapping these indices to the embedding tables and collecting the corresponding embedding vectors. Finally, a pooling operation, such as “sum”, is applied to calculate an output vector for the query item.

implementations of the embedding bag operator process the work per table in parallel, but invoke the operator across the tables *sequentially*. As a result, the time required to complete the embedding bag kernel is the “sum” of the processing time for each table.

3.2.2 Target Architectures

3.2.2.1 AMD “Genoa”

“Genoa” is based on the successful multi-chip module (MCM) based Chiplet architecture SoC (System-on-a-Chip). “Genoa” leverages the fourth generation of AMD “Zen 4” compute cores and incorporates up to 96 of these cores in a single SoC. A high-level view of “Genoa” is illustrated in Figure 3.5. The cores have been updated to support compute on BFloat16 and Int8 data types as well as the VNNI and AVX512 ISA extensions for efficient deep learning performance. The Zen4 core builds upon the Zen 3 core and incorporates improved branch prediction and larger micro-op caches (small dedicated

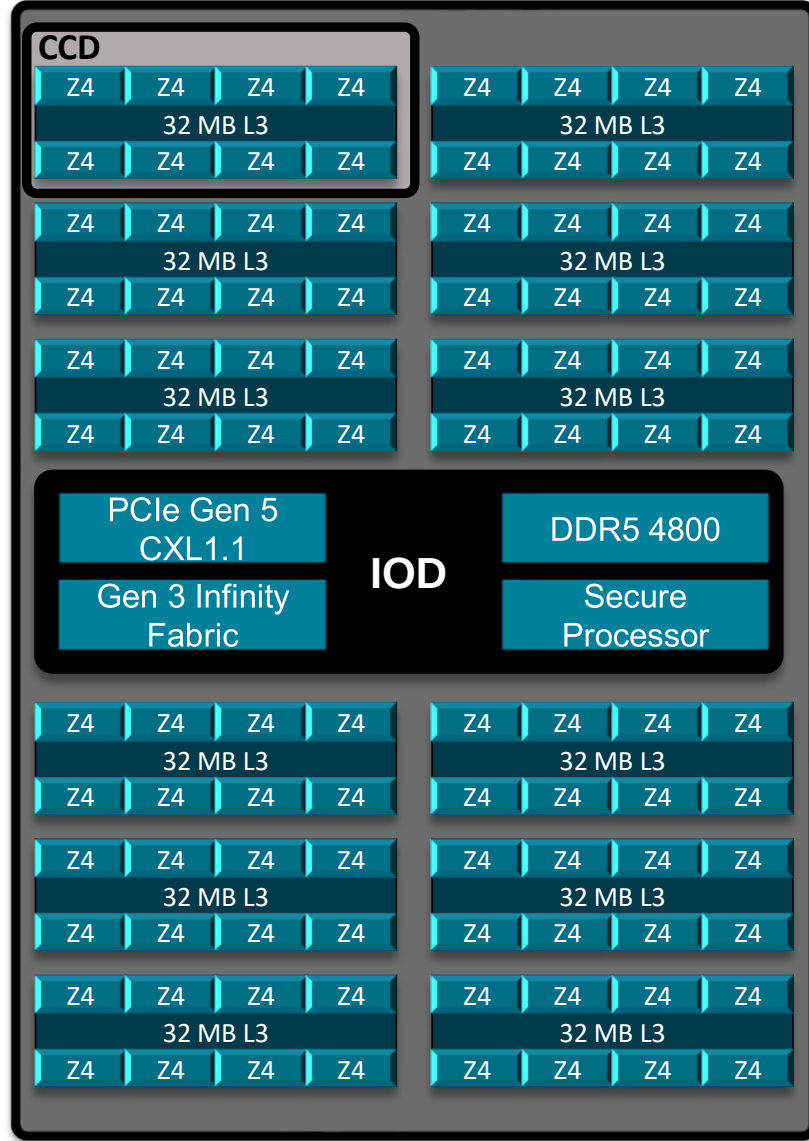


Figure 3.5: A high-level view of “Genoa”. Each group of 8 cores with an L3 cache defines a chiplet (CCD) [1].

caches for micro-ops, which are RISC-like instructions obtained by breaking up complex x86 instructions), register files, and instruction retire queues. Zen4 also has an improved load store unit with a 32KB L1 cache and a 1M L2 cache per core as well as larger TLBs. Table 3.1 gives the details of the target system used in this study.

“Genoa” builds on the next-generation Infinity Fabric to connect the cores composed into Core Complex Dies (CCDs) with the IO Die. The IO Die incorporates class-leading 12 channels of state-of-the-art DDR5 memory, providing up to 460GBps of peak theoretical

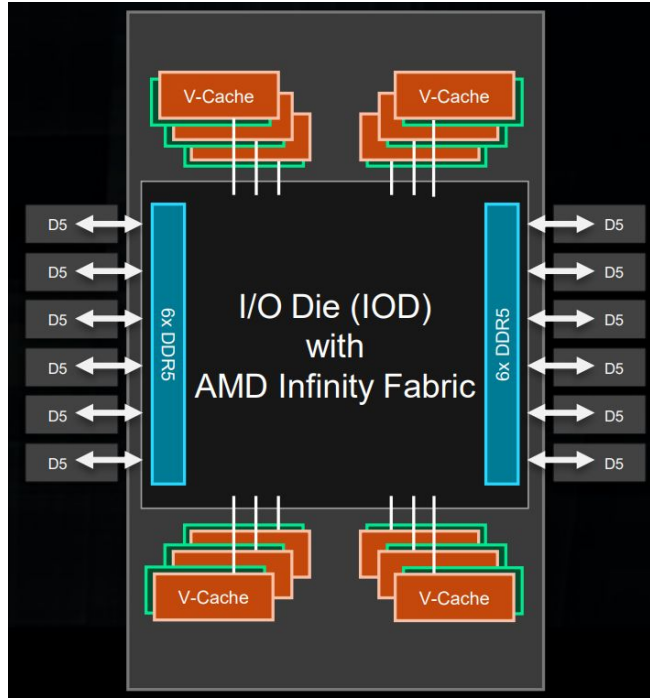


Figure 3.6: A high-level view of “Genoa-X”. V-Cache™ effectively extends LLC capacity for the cores in each CCD. Note that the LLC with V-Cache™ remains partitioned across core complexes [1].

memory bandwidth per socket. Additionally, with up to 2 DIMMs per channel, 2 socket configurations can have up to 12TB of memory by leveraging 3DS memory. “Genoa” enables up to 128 lanes of Gen5 PCIe. 64 of these lanes are also CXL 1.1 capable. TDPs for “Genoa” SoC extend up to 400w making for an extremely powerful server.

Each “Genoa” CCD has 8 cores with up to 12 CCDs in an SoC. Each of these CCDs has up to 32 MB L3, bringing the total L3 capacity in a 96-core 12 CCD SoC to 384 MB. The L3 in each of these CCDs are designed to provide extremely high bandwidth and low latency access for all the cores in the CCD and can cache the entire memory address space. Cache consistency is automatically maintained across all CCDs in the system. The Infinity Fabric allows coherent scaling to multiple sockets with flexible topology options.

For NUMA-optimized workloads, the “Genoa” SoC can be configured to operate as multiple NUMA Nodes Per Socket (NPS). Instead of distributing the memory owned by applications running on all 96 cores equally across all 12 memory channels, configuring NPS to be greater than 1 makes the OS *prefer* memory that is closer to the CCD/core on which the requesting application is running. Specifically, NPS can be configured to 1, 2, or 4. In the 4 NPS config and 2P config, the platform appears like an 8S config to

System	Sockets PCIe	1 128 lanes Gen5
SoC	Cores per SoC CCDs per SoC Memory BW Memory Capacity TDP	96 12 460GBps across 12 DDR5 channels 768GB per socket 360W
CCD	Cores per CCD L3 per CCD	8 32MB
Core	L1 Cache (instr/data) L2 Cache DTLB entries (L1/L2) Frequency	32KB/32KB 1MB 2048/3072 2.4GHz

Table 3.1: “Genoa” Configuration [3].

the OS, each with 3 of the 12 memory channels. The NUMA distances between NPS nodes on the same socket are lower than the distances between 2 NPS nodes on different sockets.

3.2.2.2 AMD “Genoa-X” – (“Genoa” + V-Cache™)

“Genoa-X” is available with a state-of-the-art 3D stacked extension of L3 (called “V-Cache”) that provides an additional 64MB capacity per core complex, for a total of 96MB L3 per core complex and 1.12GB of L3 per socket [105]. A high-level view of “Genoa-X” is illustrated in Figure 3.6. The 3D-stacked layers of L3 cache sit right on top of the CPU cores and logically extend the L3 cache. V-Cache™ is particularly effective for applications that access large datasets that possess large enough data reuse (e.g., EDA and select HPC applications).

3.2.3 Settings for Embeddings on “Genoa” and “Genoa-X”

There are multiple settings such as configuration of huge pages, numactl, and OpenMP that impact the system performance. Getting the correct values for these settings is critical for optimizing performance of a workload on a CPU-based system. Along with the threading strategies (to be discussed shortly), we carried out experiments to identify the optimal knobs for these settings.

Transparent Huge Pages (THP) is a Linux kernel feature for memory management to improve performance of the application via large memory pages. Thus, it can, in principle, reduce the overheads of the Translation Lookaside Buffer (TLB). It operates mainly in two modes, namely, *always* and *advise*. In the *always* mode, the system kernel tries to assign huge pages anytime new page is requested and enough free memory is available. In the *advise* mode, kernel only assigns huge pages when new page is requested with `madvise` system call. In our experiments, we used the *always* mode for better performance.

Among the available OpenMP settings, “thread affinity” plays a key role in extracting the best performance since it has direct impact on the *proximity* of the data to the processor on which the executing thread is created. Since we have used the OpenMP constructs for parallelization, the optimal values for the OpenMP environment variables are essential for getting good performance. Most of the settings are common for all the three threading strategies discussed later. However, there are few settings which are specific to different threading strategies. `OMP_PROC_BIND` variable controls how the threads are bound and distributed to the places. Available thread affinity policies are *master*, *close*, and *spread*, whose impacts are more pronounced when using the table threading and hierarchical threading policies.

On the other hand, `OMP_WAIT_POLICY` describes the expected behavior of the waiting threads during program execution. It can take two abstract values: *active* or *passive*. The default value is *passive*. When set to *active*, the waiting threads are mostly active, meaning that the threads consume CPU cycles while waiting (e.g., waiting threads can spin while waiting). The *active* wait policy gives better performance for the table and batch threading schemes, whereas the *passive* wait policy has an edge for hierarchical threading.

“numactl” addresses the allocation of memory by Linux kernels across the designated non-uniform memory access (NUMA) nodes. Also, it restricts the command or process execution to predetermined NUMA nodes. Employing interleaved memory policy and strategic confinement of processes across selected nodes using `-interleave` and `-cpunodebind`, respectively, for various threading strategies generates better performance, as compared to other numactl settings.

Finally, Nodes per Socket (NPS) enhances the capabilities of NUMA. Depending on the workload, NPS can be configured to achieve optimal performance. With default setting being NPS1, highly parallel workloads, such as the embedding bag operator, can benefit significantly by changing the NPS value to NPS4. This partitions the CPU

into four “NUMA domains” and memory is interleaved across memory channels of every NUMA domain.

3.3 Load Imbalance in Embedding Bag

As mentioned in Section 3.2.1, the Embedding Bag operator is a combination of table lookups performed on all embedding tables within the model. For a given model, the time to process a table depends on two factors: the number of lookups performed per input and the latency per lookup. Thus, the time per table could vary. Furthermore, each input can perform a different number of lookups per table, thus even if one assigns all lookups per input to a thread, there will be varying work per thread. Depending on the variability of lookups per thread, straggler cores may arise during execution. The latency per lookup *need not* be similar, because the data can be found anywhere in the local cache hierarchy, remote caches (in different CCDs) or DRAM. The table row locality will influence the cache hit rates and therefore the lookup latency.

In real data sets published by Meta [33], we observe significant variations in model properties like the number of lookups per input or table sizes. The time to process these tables on a single core can vary by as much as 57x, as can be seen in Figure 3.10. As mentioned above, the current implementations of embedding bag kernel depend on the properties of different tables because each user is processed independently. Thus, *the optimizations proposed using different parallelization strategies (discussed in the next section) attempt to mitigate the performance impact coming from the “non-uniformity” in table properties.*

Next, we analyze the input data set properties to understand how they impact the embedding bag behavior. Specifically, we study the recently-released Meta’s DLRM production traces [33], which we believe are representative of typical memory access patterns. Previous research efforts [100, 106–109] targeting inference have focused on input data sets that are limited in various dimensions, involving a small number of tables, a low pooling factor (number of lookups per user), and a high concentration of hot table indices. While a recent paper [99] considers more appropriate parameters compared to past works, it is limited by its homogeneous nature, which assigns every table and user with the same amount of work and similar memory access patterns.

Meta’s DLRM production traces show significant “non-uniformity” across parameters such as row sizes, pooling factors, memory access patterns, and time taken per table. We evaluate a 110 GB sized model generated using the production traces containing 129

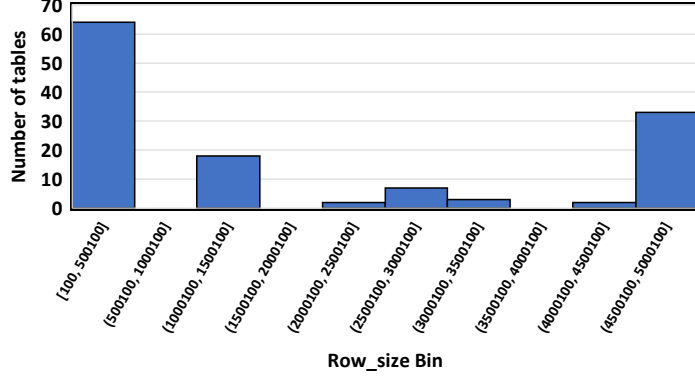


Figure 3.7: Number of tables in various row size bins.

tables (more details in Section 3.5). Figure 3.7 shows the number of tables binned by row_size and sorted by the number of tables in each bin. It is clear that the model skews heavily towards tables with row_sizes between 100 and 5 million, which translates to varying memory capacity requirements per table. For instance, there may be many cases of a small table with under 4K rows (2MB size) that fit within a 2MB page size.

Table size is one parameter that can influence the work per table. Other factors can contribute more, such as the number of lookups per user and table row locality of access. Figure 3.8 shows the behavior of average and maximum “pooling factor” per table. Average pooling factor is the average number of lookups per user within each table, and maximum pooling factor is the maximum number of lookups per user within each table. We observe that users have varying numbers of lookups, resulting in variability between the average and maximum number of lookups in many tables. Memory access locality also influences the execution time. Figure 3.9 highlights two important metrics to characterize memory accesses within a table: unique indices and 10% coverage. Unique indices show the fraction of table rows touched by the data set, while 10% coverage represents the percentage of lookups captured by the top 10% of the unique indices. These two metrics together define the Zipf’s distribution (also called as power law distribution in prior work) within the data set. We note that both metrics vary between 0% to 100%. A data set can be classified based on the degree of hotness as “high hot”, “medium hot”, and “low hot”, as defined in the previous work [99]. A high hot classification means a small number of rows cover a substantial fraction of the indices, while a low hot situation means the opposite. It is evident that the production traces exhibit a wide range of hotness levels.

Finally, we study the execution time per table in the embedding bag operator. Figure 3.10 shows the time taken by each table in descending order. The range is between 40us to 2292us, with an average time of 301us. This wide range is justified by variation of

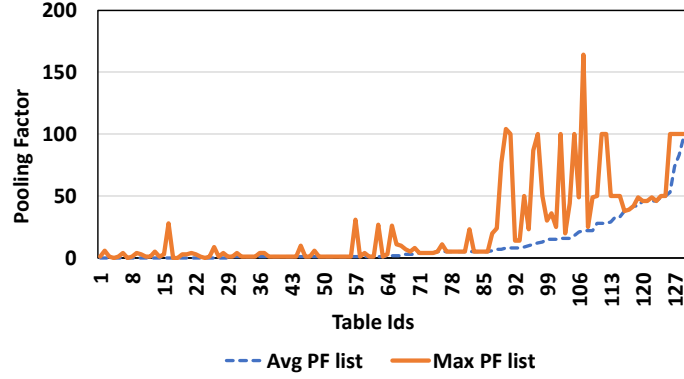


Figure 3.8: Pooling factors for different tables.

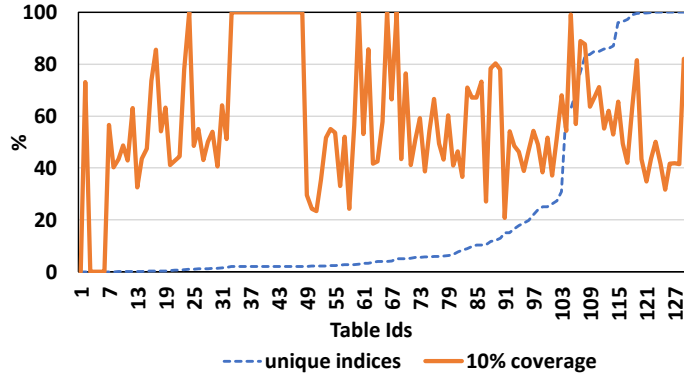


Figure 3.9: Access pattern behavior for different tables.

table size, number of lookups per table and memory access locality. The execution time variation per table can result in “load imbalance”, which is mitigated with our proposed parallelization schemes.

3.4 Parallelization Strategies for Embedding Bag Operator

The embedding bag operator is critical for DLRM Inference performance. In large-scale deployments, embedding bag performance can be the primary factor determining if SLAs are met [99, 100]. Motivated by this observation, we introduce two novel parallelization strategies (called “Table Threading” and “Hierarchical Threading”) for the embedding bag operator. The default embedding bag parallelization strategy for CPU-based DLRM systems is referred to as “Batch Threading” [80, 104, 110]. We also present a qualitative analysis of the three strategies (illustrated in Figures 3.11, 3.12, and 3.13), and explain

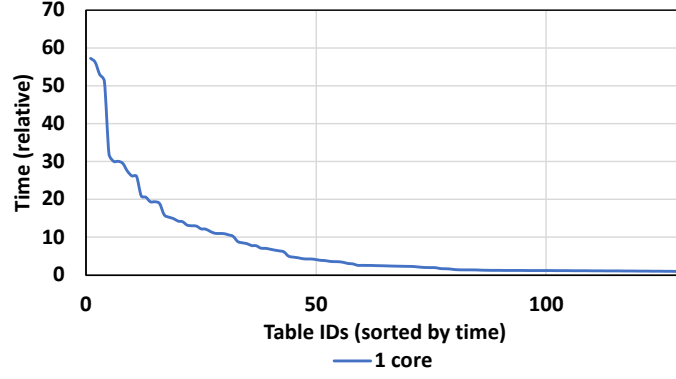


Figure 3.10: Embedding table latencies for different tables.

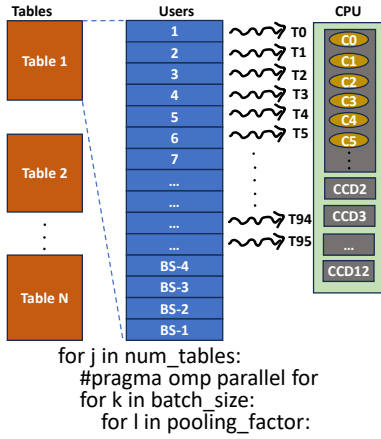


Figure 3.11: Batch Threading.

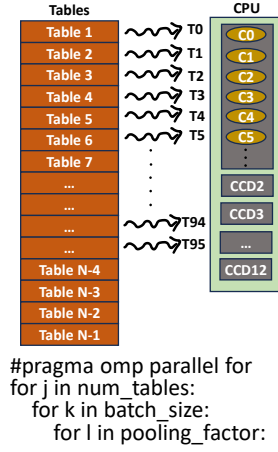


Figure 3.12: Table Threading.

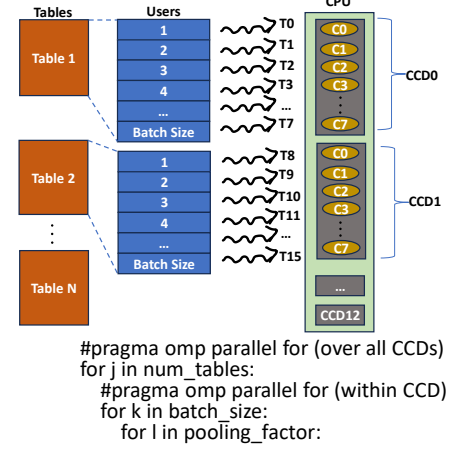


Figure 3.13: Hierarchical Threading.

the scenarios under which each strategy is expected to outperform the others. An experimental evaluation/comparison of these strategies is presented in Section 3.5.

We start by observing that the table size influences the number of rows that need to be stored in the on-chip caches. Each table exhibits different degrees of “hotness”, as demonstrated by prior works such as [111] and [99]. Furthermore, parameters like the pooling factor per batch as well as the embedding dimension per table can also differ, leading to a variable number of rows fetched per batch/table, thus varying the work assigned per SW thread. Below, we study the new parallelization strategies on state of the art, chiplet-based CPU multicore systems.

3.4.1 Batch Threading (BT)

Batch Threading is shown in Figure 3.11, both pictorially as well as in a pseudo-code format. As stated earlier, this is the default strategy employed in embedding bag parallelization on current CPU multicores [80, 104, 110]. BT maps batches within a table to all the hardware contexts (threads). Subsequent tables are scheduled after all batches in the current table complete. Under BT, tables are processed sequentially as is also evident from the pseudo-code in Figure 3.11.

Since, under BT (which is the scheme employed in ZenDNN and vanilla PyTorch), the work per table is spread across all cores, shared hot table rows accessed by different offsets (and thus threads) can be shared via the shared L3 caches. In the target chiplet-based CPU multicore architecture, this may lead to high cache-to-cache traffic both within the same and across Core Complexes and may not utilize well the on-die cache capacity. In reality, table sizes vary, thus when processing small tables, caches (as well as other on-chip resources) may not be well utilized. Furthermore, since the work per offset varies, there can be load imbalance across the threads. Since a batch size number of threads are created where each offset is a very small amount of work, the threads would observe significant runtime overheads like fork/join system calls. Motivated from these limitations, we next discuss new parallelization schemes which attempt to overcome these problems.

3.4.2 Table Threading (TT)

TT takes a different approach to the parallelization of the embedding bag operator by assigning the work of each table to a single thread. The work of a table is assigned to a core only after the work of the previous table is complete. Hot table rows accessed by different offsets are now accessed via the private core caches (L1 and L2). However, shared caches (L3) store table rows that do not fit in private caches and can experience contention among the threads depending on table working set size. Moreover, with TT the overhead of thread fork/join is amortized over more work, and there is no duplication of data across private caches.

However the amount of work per table and the table size varies (see Section 3.3) which can translate to poor load balancing across threads with TT, further differentiating thread processing times, leading to lower core utilization. In one case where the number of tables is the same as the number of threads, the overall execution time is determined by the processing time of the slowest table. Clearly, when the number of tables is larger than

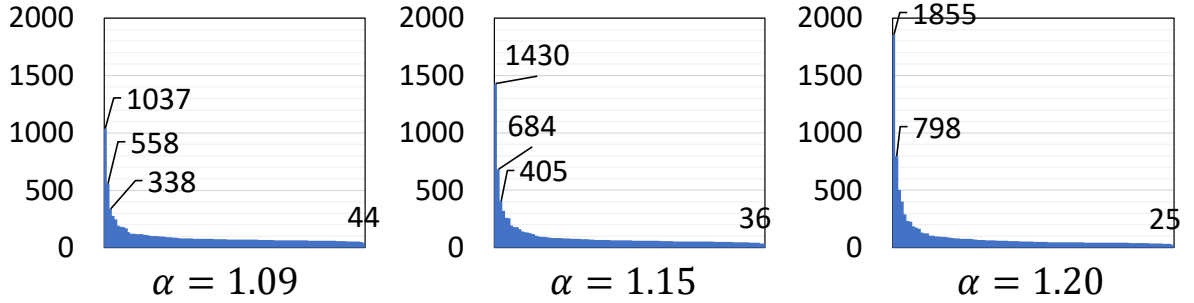


Figure 3.14: Sorted Histogram of Zipf’s distribution for different values of α . Each distribution randomly selects 10000 values from 100 entries.

the number of threads, a careful table-to-thread assignment can improve TT performance. For instance, note in Figure 3.10 that the maximum over minimum time per table ratio is 57x.

3.4.3 Hierarchical Threading (HT)

BT is expected to have good load balance; but can suffer from cache to cache traffic and low cache capacity utilization. In contrast, TT is expected to have better cache performance but has potential load imbalance. HT strikes a balance between these two parallelization strategies. Under HT, the tables are distributed across chiplets (called CCDs¹) and the threads (cores) in a CCD operate – in parallel – on the tables assigned to that CCD, one table at a time. In other words, HT shares table hot rows both via private caches (as in TT) and shared (L3) cache (as in BT). Unlike BT though, it does not require cross-CCD traffic for shared hot rows because all threads for the same table are mapped to the same CCD. If necessary, the table assignment granularity can be less than a CCD, e.g., 2 cores or 4 cores.

HT can enable good load balancing and good cache performance by carefully *mapping* the amount of work per table to the appropriate number of cores (threads).

3.5 Empirical Evaluation

In this section, we present an experimental evaluation of three embedding bag parallelization strategies discussed in this paper.

¹Recall that, in AMD architectures, a CCD is a cluster of 8 CPU cores that share a L3 cache. Depending on the particular system, an AMD CPU can have one or more active CCDs per physical processor.

Experimental Knob	Coverage Values
Number of Tables	192, 384, 576
Size of Each Table	4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB
α value	1.09, 1.15
Batch Size	1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K
Threads per CCD for HT	4, 8
Threading Schemes	BT, HT, TT
Pooling Factor	25
Embedding Dimension	64

Table 3.2: Homogeneous Zipf’s distribution based on microbenchmarking coverage set.

3.5.1 Methodology

Datasets: Our experimental evaluation includes *both* synthetic traces and real-life traces which are grouped in two categories: “homogeneous” and “heterogeneous”. The homogeneous datasets follow uniform properties across tables such as the table size, batch size, pooling factor, α (the exponent parameter), and embedding dimension for a particular performance run. Since embedding lookup indices are known to follow Zipf’s distribution [33, 104, 111], we follow the methodology used in FBGEMM [104] by injecting indices generated from Zipf’s distribution for measurement.² Figure 3.14 shows a sorted histogram of 10000 randomly picked values from a 100 entries using Zipf’s distribution for three different settings of α . As α increases, frequency of low rank value increases and frequency of high rank decreases (as shown by the top three values and the last value). FBGEMM [104] from Meta uses a value of $\alpha = 1.09$ for inference and $\alpha = 1.15$ for training. We have shown the histogram in Figure 3.14 for $\alpha = 1.2$ in addition to $\alpha = 1.09$ and $\alpha = 1.15$, to highlight how the distribution varies with increasing values of α . However, we have only used $\alpha = 1.09$ and $\alpha = 1.15$ in our experiments. This synthetic dataset is termed as “homogeneous Zipf” in the remainder of this section. Table 3.2 captures all the performance test parameters that that were covered as a part of the performance study.

In addition to these datasets, we also used “representative” datasets provided by Meta in their DLRM dataset repository [33]. This dataset is referred to as “heterogeneous” because of non-uniformity in table properties such as the number of rows, lookups per user and memory access patterns. Please check Section 3.3 for quantitative details about

²Zipf’s distribution is found in the linguistic corpus where frequencies of words are inversely proportional to their ranks.

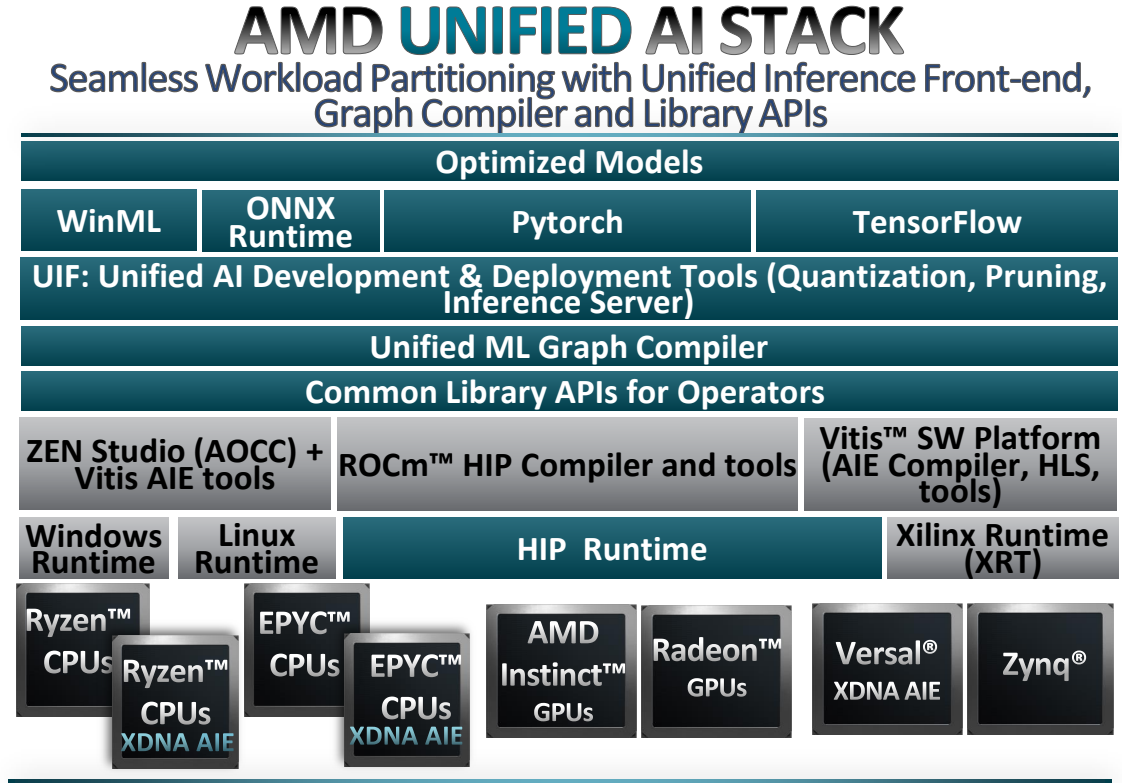


Figure 3.15: AMD Unified AI Software stack.

non-uniform characteristics present in the heterogeneous dataset. Note that there are 16 different datasets present in [33], and we study all of them in Figure 3.3. However, for the sake of brevity, remaining heterogeneous dataset discussion uses dataset id 0 unless mentioned otherwise. Further, in the dataset 0, we have reduced the number of tables from 788 to 129 to fit in the DRAM on the system. Despite using fewer tables, we ensured that the dataset’s heterogeneity is maintained as can be seen in varying latency for different tables in Figure 3.10.

Metrics: Recall from Section 3.3 that embedding stage is memory intensive. Therefore, we believe “Bandwidth” is a representative and comparable metric for performance. With every batch, the total work with the memory accesses can be thought as the amount of data to be executed. It can be calculated as a product of model parameters: batch size, number of tables, pooling factor, embedding dimension and precision of embedding values. For example, in the homogeneous dataset, it would be $(1024) \times (96) \times (150) \times (128) \times (4) = 7.03$ GB. Batch latency means the time it takes to finish 7.03GB size of work by a given execution scheme. It is to be noted that any change in model parameters and hardware configuration would impact the batch latency. Thus, comparing batch

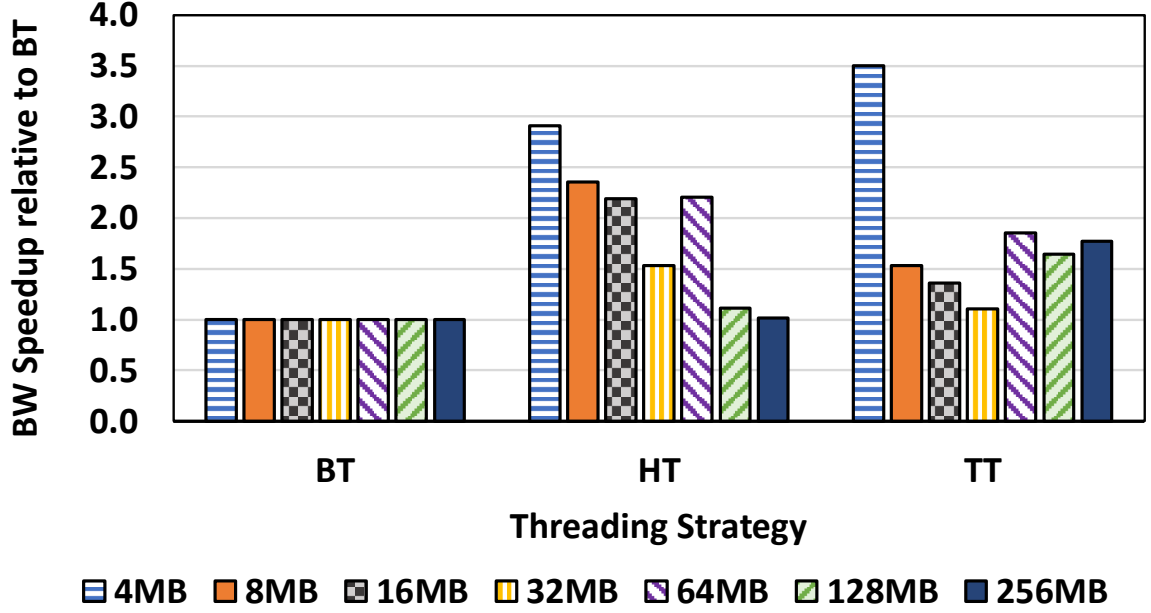


Figure 3.16: HT and TT performance relative to BT for different table sizes using Zipf’s distribution on “Genoa”.

latency across two different settings may *not* directly reveal the impact on underlying performance. Consequently, taking inspiration from [104], we use realized bandwidth as a new metric in our evaluations along with batch latency. Note that the realized bandwidth (or bandwidth, for short) can be calculated using the formula (amount of data read)/(batch latency) which is interpreted as the “rate” at which “amount of data” is consumed, thus having the unit GBps. As a result, this metric allows one to compare the rate of work done across different model and execution configurations.

Hardware: Our experiments are performed on the “Genoa” and “Genoa-X” architectures. The detailed descriptions of these architectures as well as system settings for THP, thread affinity, wait policy numactl and interleave policy have been given earlier in Section 3.2.

Software Stack: ZenDNN (Zen Deep Neural network) is the primary deep learning software stack for AMD CPUs. ZenDNN contains optimized implementations of deep-learning operators including fused operators. ZenDNN augments the capabilities provided by popular machine-learning frameworks such as Pytorch and Tensorflow and works seamlessly with them. Developers can use ZenDNN to develop neural network models and solutions with high performance and deploy at scale on EPYC™ CPUs. ZenDNN uses AMD AOCL-BLIS (which is an extension of BLAS) for linear algebra functions and

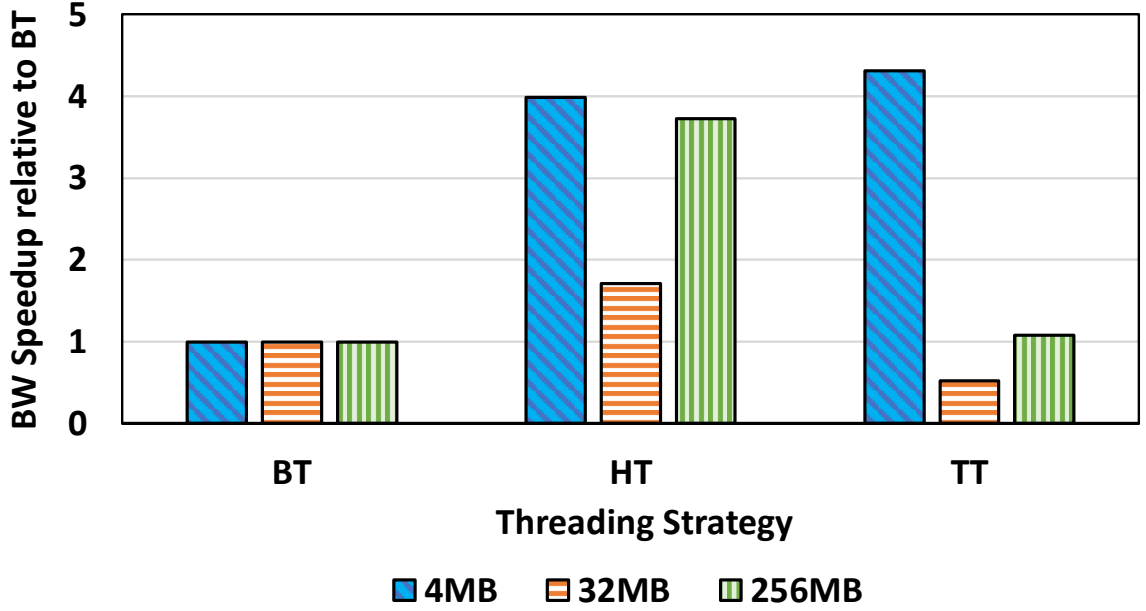


Figure 3.17: HT and TT performance relative to BT for different table sizes for Zipf’s distribution on “Genoa-X”.

AMD Math library, LibM, for core mathematical functions. ZenDNN is released as part of Zen Studio software suite for EPYC™ CPUs, and it will be integrated into the AMD Unified AI software stack (UAI), as shown in Figure 3.15. The optimized embedding bag implementation will be part of an upcoming ZenDNN release.

3.5.2 Results

We start by presenting, in Figure 3.16, the performance of the parallelization schemes assuming a Zipf based table access distribution. All results are *normalized* to BT. We observe that, for table sizes that fit in the L3 (Table size of 4MB), TT and HT outperform BT (4.5x and 3x, respectively), because both TT and HT process more work (tables) in parallel than BT while memory traffic is not high (due to table row locality) enough to stall execution. TT outperforms HT for the same reason. We expect similar trend for smaller table sizes that fit in L3. For table sizes between 8M and 32M, HT outperforms BT and TT (2.3x over BT for 8MB table size). Overall performance is lower than that of 4MB tables because of more capacity/conflict L3 misses due to the increased data working set per thread. HT has better performance than TT because of the lower data working set size per CCD, which leads to less memory traffic. Finally, for large enough tables (e.g., table sizes of 128MB and 256MB), TT outperforms both BT and HT. Note

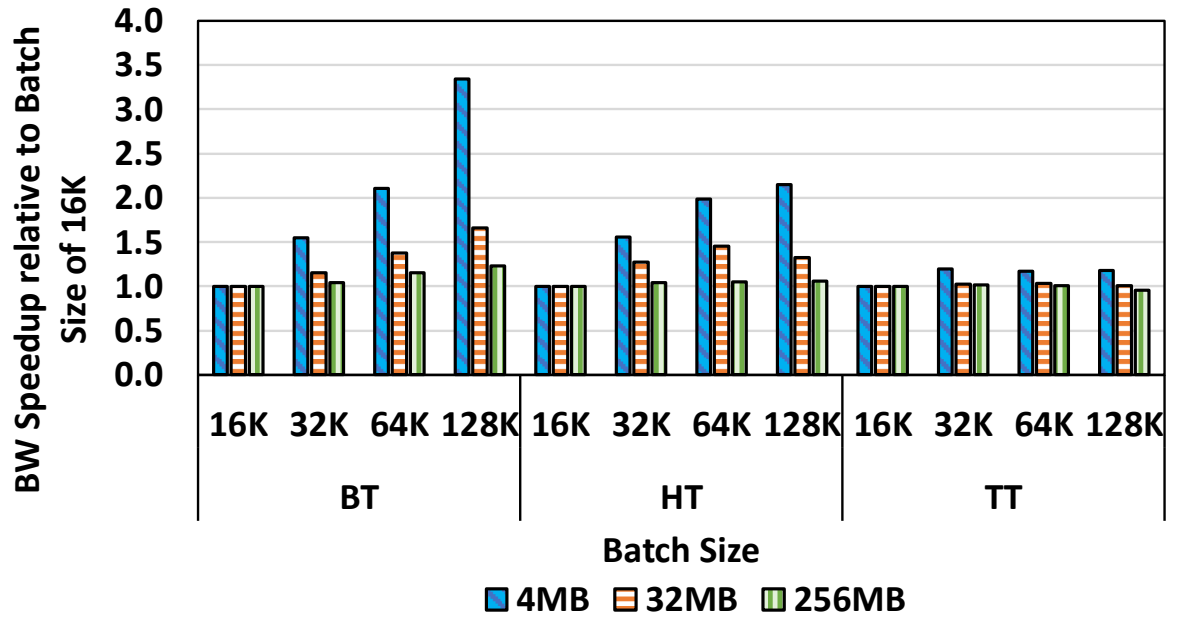


Figure 3.18: Batch size sensitivity for all threading strategies on “Genoa”. Numbers are normalized to the Batch size of 16K.

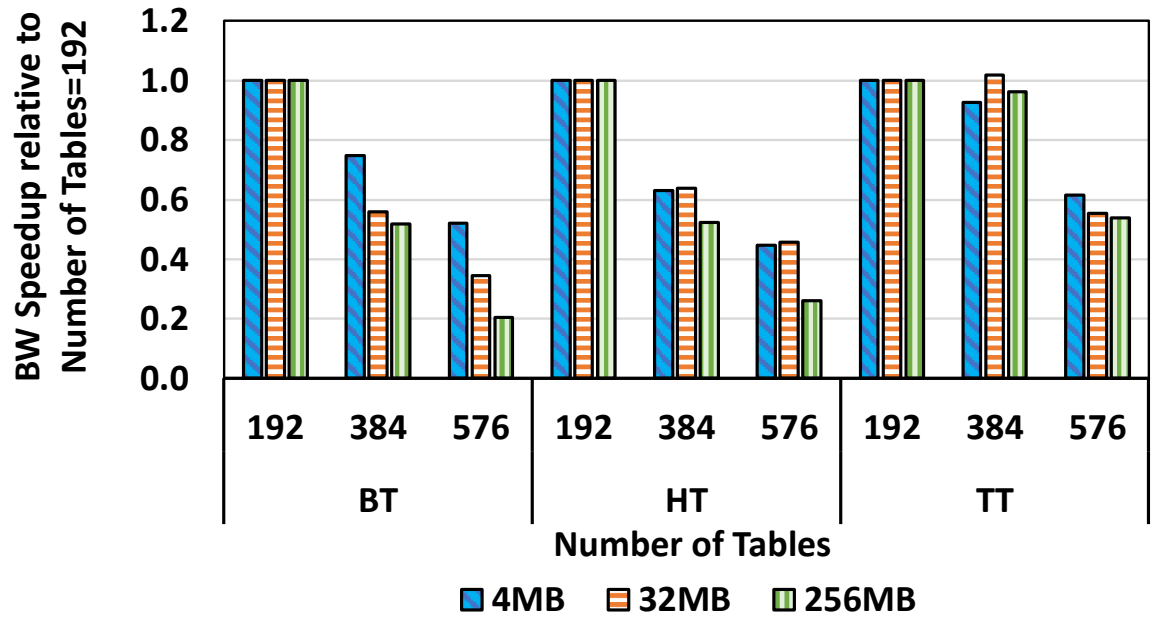


Figure 3.19: Performance of various parallelization strategies on “Genoa-X” across different number of tables and table sizes. All numbers are normalized to the 192 number of tables configuration per table size.

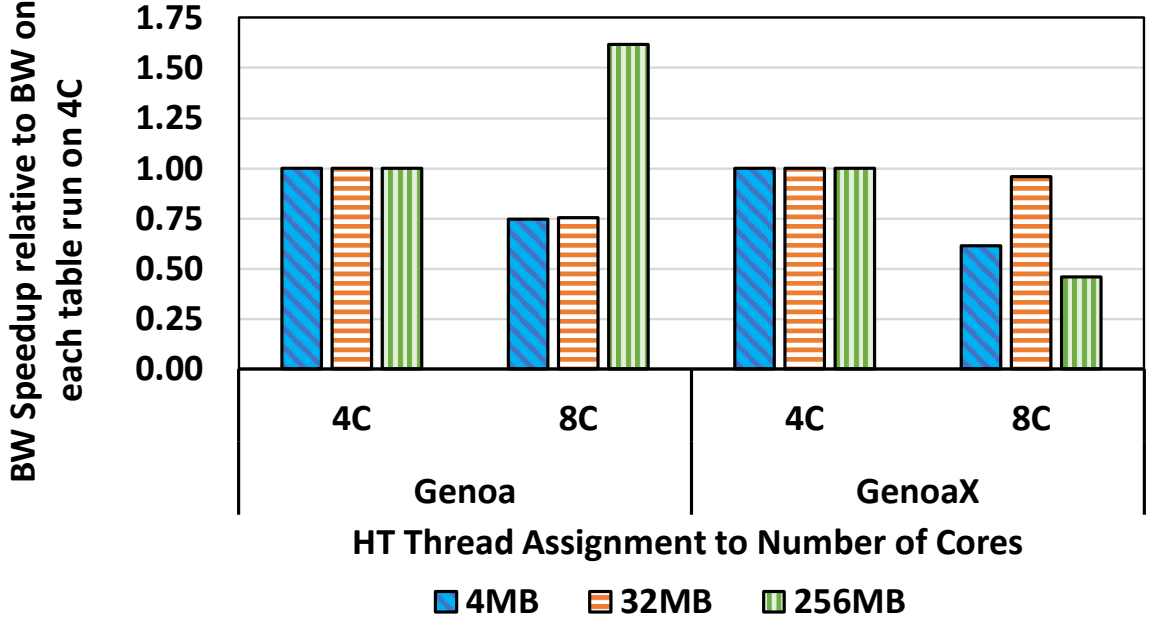


Figure 3.20: HT performance when each thread is scheduled on 4 cores (4C) vs 8 cores (8C) for “Genoa” and “Genoa-X”. All numbers normalized to the 4C case per table size. Number of tables is 192.

that, the supported Memory Level Parallelism (MLP) per CCD is defined by the number of L3 MSHRs (Miss Status Hold Registers), and remains independent of the table size or number of tables assigned per CCD. Therefore, if we assign sufficient table capacity or number of tables per CCD that does not fit in the L3 but saturates MLP, the fastest parallelization scheme is the one that processes the most tables simultaneously. We observe from Figure 3.17 that HT outperforms both TT and BT even for large tables (256MB). This is because, “Genoa-X”’s much larger cache capacity significantly reduces L3 capacity/conflict misses and does not saturate the CCD’s MLP, allowing HT to perform better due to lower overall memory traffic.

Another way to study the impact of capacity is from the perspective of batch size. Increasing the batch size increases the working set size as we pick up more table entries from the Zipf’s probability distribution. In Figure 3.18, the BT’s performance benefits from larger working set size across table sizes since it provides the largest cache capacity. The HT’s performance benefits from increases in batch size for small table sizes. For medium sized tables, the HT performance improves until a batch size of 64K and drops for a batch size of 128K. Also, TT saturates as batch size increases for both medium and large tables. For small tables, the performance of TT improves until 32K and saturates for higher batch sizes. On the other hand, for low batch sizes such as 1K (Refer to figure

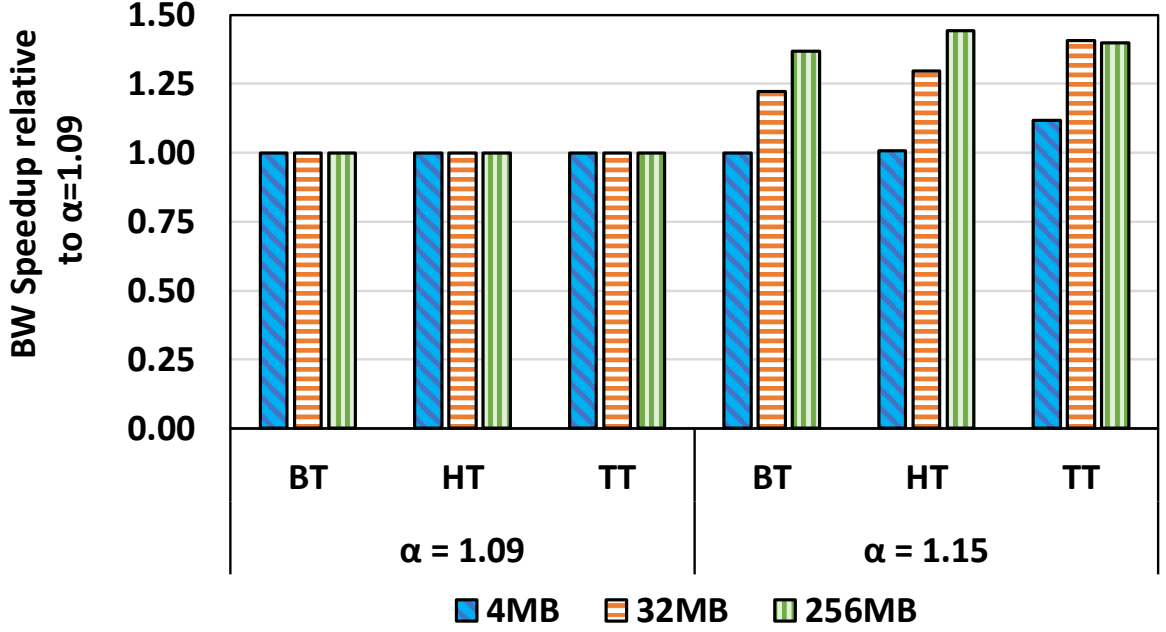


Figure 3.21: Parallelization scheme performance with α ($\alpha=1.15$ vs $\alpha=1.09$) for different table sizes.

3.22), TT outperforms both HT and BT. For 4MB, 32MB and 256MB size tables, we see 9.9x, 4.3x and 4x gain over BT, respectively.

The number of tables also impacts execution time. Figure 3.19 shows the performance scaling of all three schemes with 192, 384 and 576 tables on “Genoa-X”. Within each parallelization scheme, performance has been *normalized* to that of 192 tables (data is shown for different tables sizes). All schemes show lower performance as we increase the number of tables (note that performance metric is the rate at which we process table data). This is because the amount of fetches required for the pooling operations that generate the output vectors increases with the number of tables. And, those operations are expensive because they require cache-to-cache transfers across CCDs (BT) or within a CCD (HT) or memory accesses. TT maximizes the overlap across pooling operations and shows less sensitivity to the number of tables.

We also studied the sensitivity of HT to the number of cores per CCD assigned per thread. Figure 3.20 shows results when assigning a thread to either half (4C) or all cores (8C) per CCD in both “Genoa” and “Genoa-X” for 192 tables. In the 4C configuration, 2 tables are scheduled on the same CCD, increasing the work processing rate of HT compared to 8C. Therefore, if the table size fits into L3 (e.g., table sizes of 4MB and 32MB), then the embedding bag is mostly compute-bound and its performance scales better with 4C in both “Genoa” and “Genoa-X”. When the memory traffic dominates

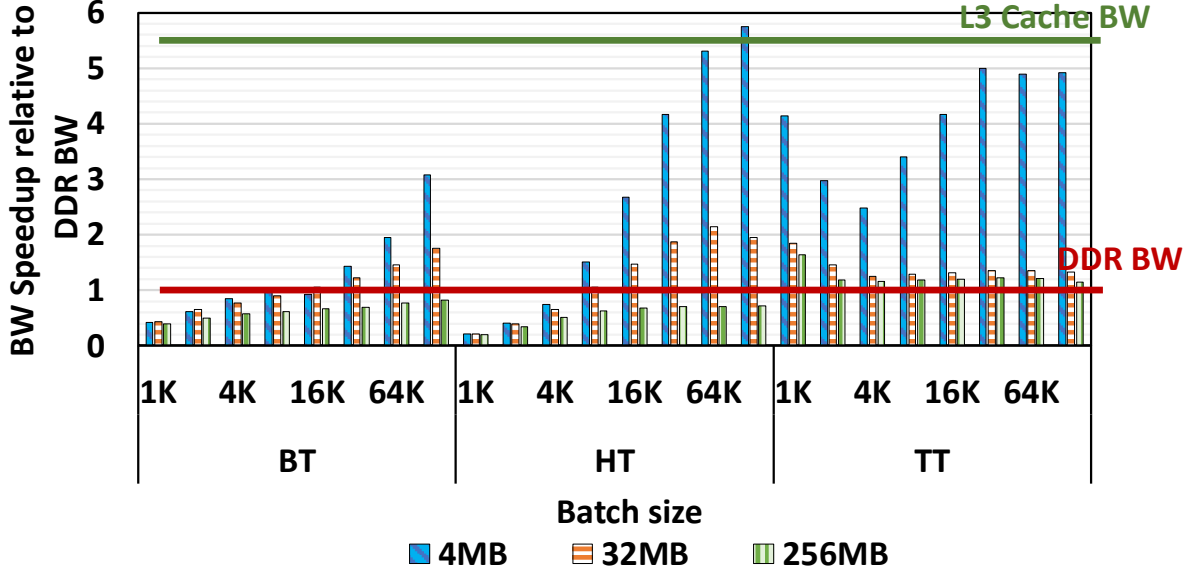


Figure 3.22: Realized bandwidth relative to DDR bandwidth for “Genoa”.

(e.g., 256MB table size on “Genoa”), 8C performs better because of the lack of inter-thread conflict L3 misses (observed in 4C). Note that the inter-thread conflict L3 misses are much less of an issue in “Genoa-X” due to its much larger L3 capacity (number of cache sets), and as a result, 4C performs better.

All the graphs discussed so far have been generated using Zipf’s distribution having an $\alpha=1.09$. We also evaluated sensitivity of the parallelization schemes to higher table row reuse by using a value of $\alpha=1.15$. In Figure 3.21, we evaluate the performance change relative to $\alpha=1.09$ on “Genoa”. All values are *normalized* to $\alpha=1.09$ for each scheme separately. When the working set per CCD fits into L3 (e.g., table sizes of 4MB for all three schemes) there is no or small performance change (TT which suffers from inter-table interference in L3 sees small performance gain). On the other hand, when the working set per CCD does not fit into L3, the performance improves due to better table row reuse across the caches. TT does not benefit for really large tables (e.g., 256MB) because of very high inter-table interference.

Recall in Figure 3.2 we highlighted the need for improvement in realized bandwidth. In Figure 3.22 and Figure 3.23 we show the gains of our proposed threading schemes relative to DDR bandwidth vs Batch threading on “Genoa” and “Genoa-X”. On “Genoa”, we have been able to realize 5.7x bandwidth relative to DDR using HT on table size of 4MB for batch size of 64K. For smaller batch size of 1K and 2K for a 4MB table size, TT generates 3x and 2.5x bandwidth over DDR. HT performs well on “Genoa-X” with 32MB and 256MB tables. We see a realized bandwidth of 3.8x and 1.5x relative to DDR

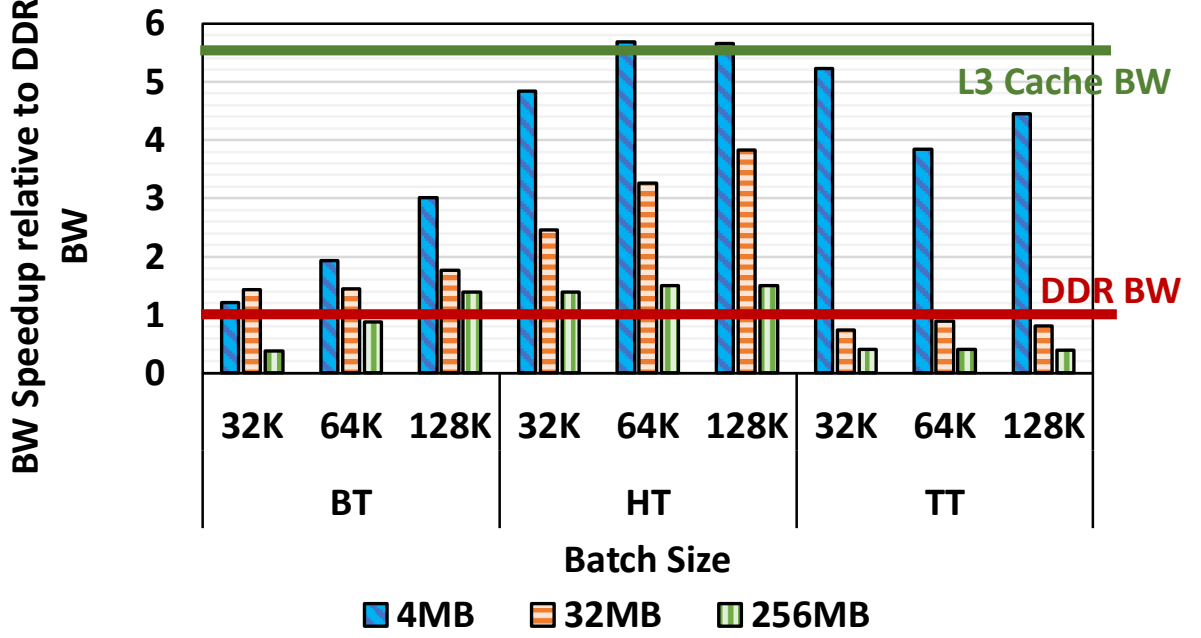


Figure 3.23: Realized bandwidth relative to DDR bandwidth for “Genoa-X”.

bandwidth for 32MB and 256MB tables, respectively, for a batch size of 128K.

We also studied performance scaling across core counts with heterogeneous data sets (Meta traces) in Figure 3.24. While BT continues to not scale beyond 16 cores (for the same reasons as described earlier), TT scales up to 24 cores, achieving up to 1.9x speedup while HT scales all the way up to 96 cores and achieves up to 2.95x speedup. As discussed earlier in Section 3.4, HT scales better because it utilizes resources better than TT. More specifically, with HT, there is no inter-table traffic interference at the L3 cache (which is significant because of the large table size) and less L3 miss traffic due to lower working-set per core complex. HT can dedicate more resources (cores + L3 capacity) to the actively processed tables with 48 and 96 cores. With a total of 129 tables to process and 48/96 cores, TT will underutilize the available cores and L3 cache for the last 33 tables whereas HT for the last 9 tables only, leading in turn to lower load imbalance. Also, recall from Section 3.3 that, the work per table heavily varies which leads to large variation in execution times per core. This is also highlighted in Figure 3.25 which shows the distribution of time per table in the heterogeneous dataset with HT. The execution time ratio of the largest over smallest sized table is 4.7x lower, when embedding bag is executed with 8 cores.

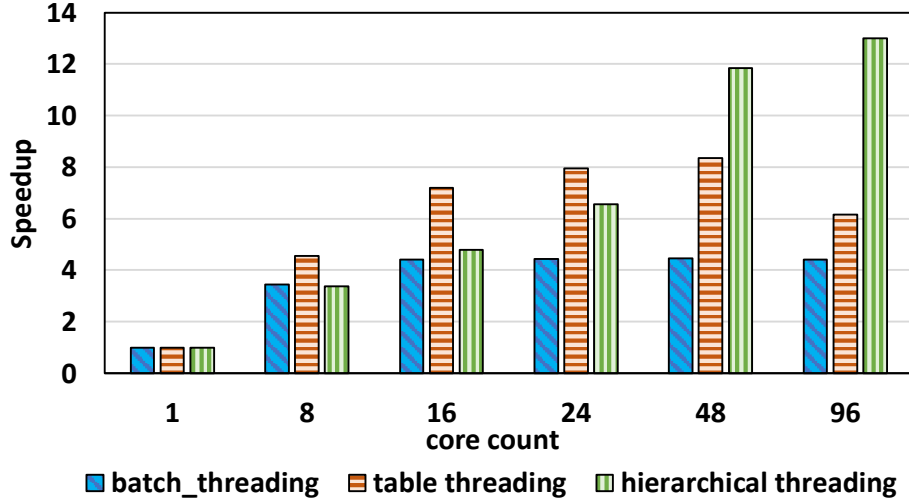


Figure 3.24: Performance with heterogeneous data set on "Genoa" (129 tables, with table sizes varying from 12.5KB to 610MB). All numbers are normalized to the single core case.

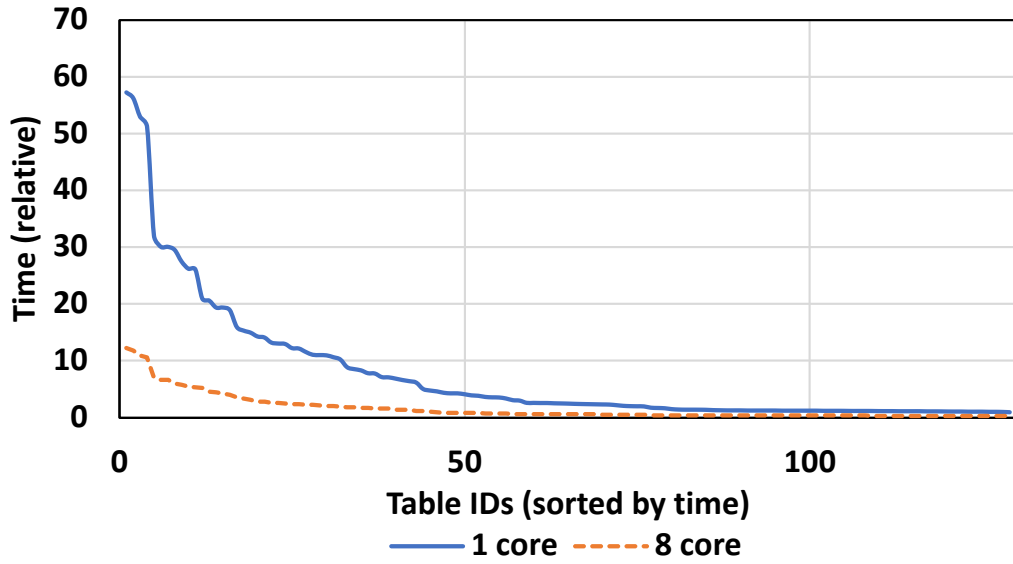


Figure 3.25: Embedding table latency for different tables.

3.6 Related Work

DLRM [14] has gained a lot of attention in industry since it was first proposed by Facebook in 2019. Several works have since been proposed to accelerate DLRM training or inference [18, 49, 50, 99, 100, 106–109, 112–122].

To improve DLRM workloads at scale, prior work identified bottlenecks via profiling and optimizing end-to-end performance. For example, [113, 123, 124] focused on DLRM

training on CPU clusters. They all considered “hybrid parallelism”, which utilized data parallelism and model parallelism at the same time. Moreover, they tried to reduce communication overheads for better training throughput.

In addition to CPU-based studies, several studies have also focused on optimizing DLRM training on GPU clusters. [114, 125–128]. Specifically, [114] optimized the embedding cache using the information of future cache accesses, ensuring that data always reside in GPU memory. [127] and [126] developed lookahead algorithms to improve caching. In comparison, [128] improved the collective communication through GPU-initiated networking and efficiently overlapped communication with computation.

In [129] and [130], authors explored various optimizations for hybrid CPU+GPU systems, and [18] and [50] optimized training on different hardware platforms, enabling efficient training of DLRM models. Additionally, [112] proposed and evaluated a system to reduce the overhead of checkpointing DLRM models during training.

Other research focused on optimizing DLRM inference [99, 106, 107, 111, 115, 131, 132]. For example, [99] explored prefetching, hyperthreading, and overlapping of computation with memory accesses, to speed up DLRM inference on CPUs. In comparison, [111] sped up inference by extending the memory hierarchy to Storage Class Memory (SCM) through Software Defined Memory (SDM). [131] proposed a hierarchical parameter server that can achieve low-latency inference via the GPU embedding cache and hierarchical storage architecture. [132] proposed a framework that can find and cache the popular items using the Item Co-occurrence Graph (ICG) to reduce the memory traffic during inference.

Due to the extremely large size of embedding tables, many prior works concentrated on optimizing memory-intensive operators. For instance, [118] optimized data storage to reduce data stalls during DLRM training. [133] proposed a new data structure for DLRM training on heterogeneous memory systems, and [134] proposed an optimized caching mechanism. Additionally, [117] and [109] explored in-storage and near-storage computing, while [122], [120], and [121] investigated near-memory computing to accelerate DLRM inference. Further efforts to enhance memory efficiency, including [116], [119] and [49], proposed embedding- or feature-based optimizations. Finally, [108] improved cache utilization through various data placement strategies, and [135] proposed a software cache approach to improve efficiency.

3.7 Concluding Remarks

General-purpose CPUs are ubiquitous in datacenters and can be cost-effective compared to GPUs. DLRM inference cycles consume a large fraction of AI cycles in data centers. Though embedding stage in DLRM is limited by memory bandwidth and capacity, table indices have been shown to follow Zipf’s distribution which has high potential for reuse. AMD 4th Generation EPYC[™] processors have large caches as well as high DDR capacity that can benefit the embedding stage. While prior DLRM optimization efforts on CPUs have used software prefetching and computation overlapping, this paper focuses on exploiting chiplet-aware threading schemes that better utilize the cache hierarchy. We propose two novel parallelization schemes (table threading and hierarchical threading) that scale DLRM Inference on state-of-the-art CPU multicores across the full core count. We demonstrate up to 9.9x speedup on homogeneous datasets based on Zipf’s distribution, and 2.95x speedup on heterogeneous datasets. We have been able to realize an effective bandwidth of up to 5.7x of DDR Bandwidth. Our techniques have been implemented in the CPU-based ZenDNN library, making them a competitive platform for DLRM inference. We believe our work sets a new baseline for any future research targeting DLRM Inference.

Chapter 4 |

Load and MLP-Aware Thread Orchestration for Recommendation Systems Inference on CPUs

4.1 Introduction

As mentioned in Chapter 1, a DLRM inference task passes through four stages: embedding, bottom multi-layer perceptron, feature interaction, and top multi-layer perceptron, as shown in Figure 4.2. The embedding stage is known to be the most memory-intensive stage [100, 107, 136–138]. The other three non-embedding stages are compute bound. During inference, the entire recommendation model is loaded once into main memory and queries by multiple users are batched. Each inference request generates top-K recommended items to the users serviced by each batch.

As discussed in Chapter 2 and Chapter 3, the embedding stage accounts for the majority of execution time [136, 138–140]. Furthermore, it is a latency-critical stage because of tight user-facing latency requirements [100, 115, 141]. The embedding stage consists of a nested loop that loads data in vector registers from various rows of large arrays/tables and performs additions. Given the irregular memory traffic required by the embedding stage, performance benefits are limited by increasing the compute resources of CPU processors. Chapter 2, Chapter 3, and prior works [136, 140, 142] investigated parallelization and thread scheduling techniques to optimize the embedding stage in the context of CPUs. The state-of-the-art CPU parallelization technique for DLRM inference [140], called hierarchical threading (HT), aims to optimize the embedding stage

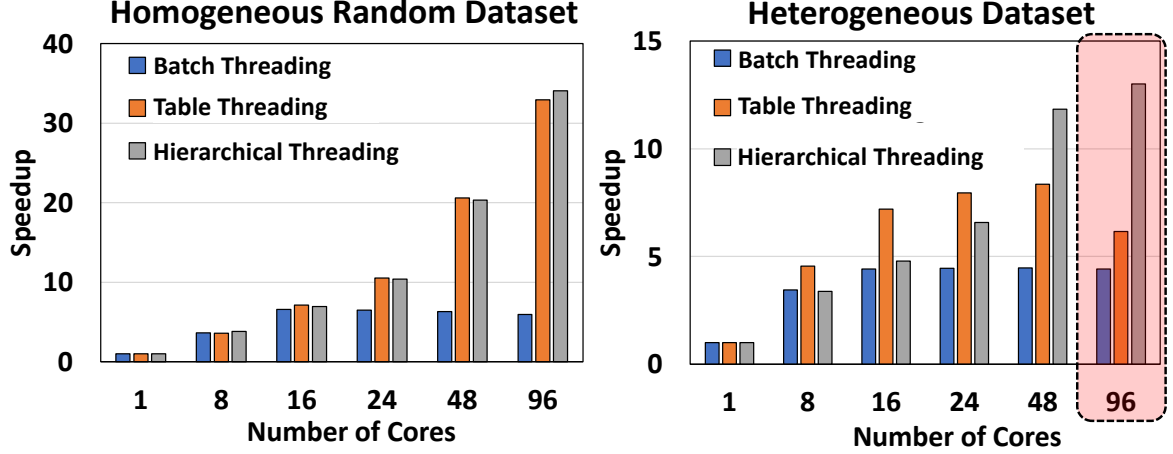


Figure 4.1: Performance scaling of batch threading (BT), table threading (TT), and hierarchical threading (HT) over homogeneous random and heterogeneous datasets. Reported speedup is over single core performance. Despite the significant speedup of HT over BT and TT, absolute speedup achieved over heterogeneous tables is only $13\times$ (highlighted) and much lower than the ideal speedup of $96\times$.

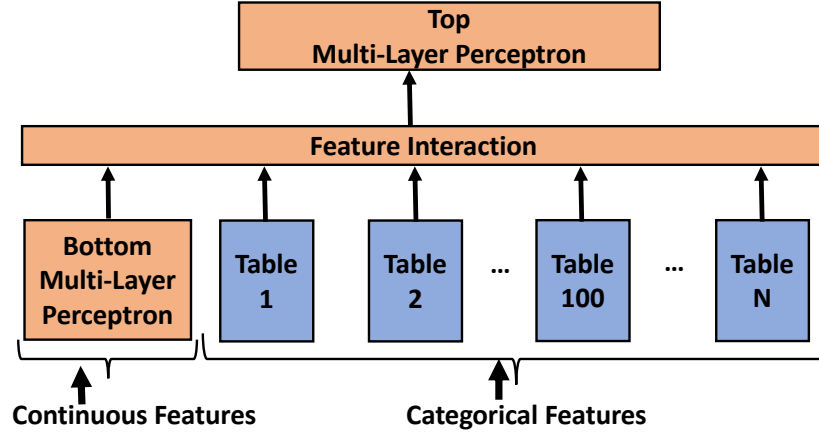


Figure 4.2: DLRM architecture consists of four stages: bottom multi-layer perceptron, embedding, feature interaction, and top multi-layer perceptron.

by mapping the work of each table on a separate CPU chiplet, also called core complex dies (CCD), for some architectures. The performance benefits of this approach over prior techniques are mostly due to lower embedding table data replication across private caches, absence of data replication across per-CCD shared L3 caches, lower capacity contention at the per-CCD shared L3 caches, and reduced load imbalance across threads. However, we find that HT does not effectively utilize all of the cores in the system.

Figure 4.1 shows the performance scaling of batch threading [140, 143, 144] (BT), table

threading [140] (TT), and HT as the number of cores increases, for two different datasets (i.e., homogeneous random tables and heterogeneous tables, which are representative of production tables). As we can observe, HT can provide a $13\times$ speedup with the heterogeneous dataset using 96 cores. There remains a significant performance gap between the achieved $13\times$ speedup compared to the ideal speedup of $96\times$. A detailed discussion is presented in Section 4.2.2.

We analyze HT and identify three key opportunities. First, due to the heterogeneous embedding table structure, some CCDs are assigned more work compared to others. This *load imbalance*¹ across CCDs can cause under-utilization of compute and cache resources. Second, core utilization can be low because many of the tables do not have enough associated work to keep all the cores in a CCD busy. Third, tables are not only heterogeneous in terms of their total work, but also in terms of their data reuse characteristics. This means that some CCDs utilize their cache capacity better while they under-utilize their memory-level parallelism (MLP) resources such as miss status handling registers (MSHRs), while other CCDs saturate their MLP and leave cache capacity underutilized. As a result, execution achieves *lower MLP utilization* and thus lower memory bandwidth, which in turn limits the performance of the memory-bound embedding stage. To our knowledge, ours is the first work to study the architectural implications when executing HT with heterogeneous embedding tables and to point out the inefficiencies leading to multi-core underutilization.

To quantify the impact of table heterogeneity on *core allocation*, we introduce a heuristic called *table load (TL)* that approximates the execution time of a table. We use TL to categorize tables as heavy-weight, medium-weight, or light-weight. Experimental analysis reveals that light-weight tables experience higher thread-scheduling overheads [145] and benefit less from more cores compared to heavy-weight tables. To quantify the impact of table heterogeneity on *MLP*, we perform a study with more MLP resources (i.e., L3 MSHRs) while other compute resources stay the same and observe significant speedup. Next, we introduce a new metric called *weighted MLP (WMLP)* that approximates the MLP demand of each table.

Based on these analyses, we develop three mechanisms to exploit the identified opportunities. First, to address *low core utilization*, instead of always executing the work associated with a table using all cores of a CCD, we propose to statically allocate a subset of the cores for each table. Second, to improve *MLP* utilization, we use two work queues (one for high- and one for low-WMLP tables) to co-locate tables from different

¹HT has low load imbalance among threads in the same CCD.

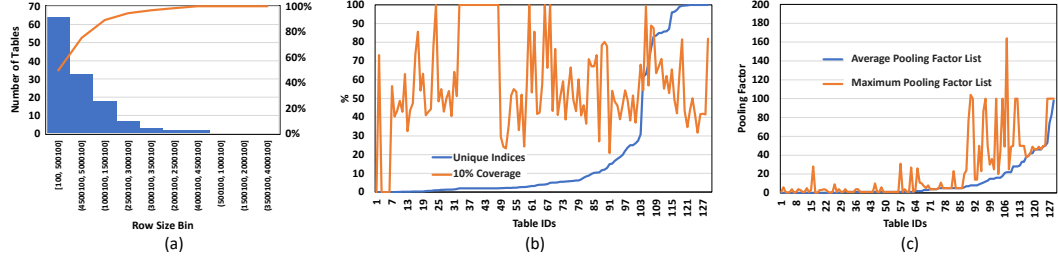


Figure 4.3: (a) Number of rows in various row size bins, (b) access pattern behavior for different tables, and (c) pooling factors for different tables.

queues on the same CCD. Finally, we employ a combination of work stealing and table reordering mechanisms to reduce *load imbalance*. To our knowledge, the three ideas (low-core grouping, MLP-aware scheduling, work stealing and table reordering) have not been proposed by prior works for DLRM inference.

The combination of these three mechanisms results in a $1.67\times$ performance improvement over HT. Our analyses and evaluation demonstrate that the performance of DLRM inference on CPUs can be significantly improved with better utilization of core, cache, and MLP resources in the SoC.

Algorithm 7: Embedding stage kernel consists of five loops. The two outer loop nests are implemented in Python [146], while the three inner loop nests are implemented inside the C++ embedding bag operator [147].

```

1 [1]  $i$  in  $num\_batches$   $j$  in  $num\_tables$   $k$  in  $batch\_size$   $l$  in  $pooling\_factor$ 
    $vec.ld$   $accm$ ;  $m$  in  $embedding\_dim$   $vec.ld$   $row\_block$ ;
    $vec.add$   $accm$ ,  $accm$ ,  $row\_block$ ;  $vec.st$   $accm$ ;

```

4.2 Embedding Stage Parallelization Inefficiencies

This section first discusses the DLRM software pipeline and the quantitative properties of the DLRM inference production traces. Then we discuss prior DLRM parallelization approaches and quantify their limitations in multi-core scaling. To better understand the scaling issues, this section analyzes limitations of HT by exploring the following questions: (1) How prevalent is load imbalance across CCDs? (2) How well does the work of each embedding table scale with respect to the number of cores? (3) How effectively are a CCD’s MLP capabilities being utilized? The answers motivate the solutions adopted by Balance.

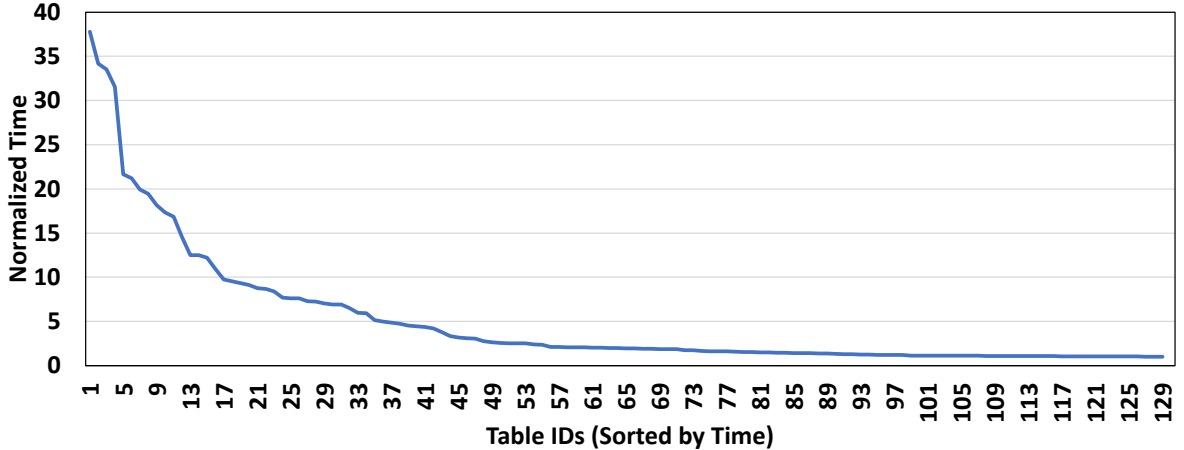


Figure 4.4: Embedding table latencies for different tables.

4.2.1 Overview of Embedding Stage Computation and Dataset

Algorithm 7 shows the embedding stage kernel in pseudo code. Batches are formed from arriving queries where each batch executes work associated with each table and contains a batch size number of samples. Each sample executes some pooling operations where a number of embedding vectors associated with the input indices are first looked up (SIMD load operation) in an embedding table and then aggregated (SIMD add operation). This process creates a load-to-add dependency chain. The volume of data loaded per batch is a function of the model and dataset parameters. For example, with 100 tables, a batch size of 1024, and a static pooling factor of 150, a total of 7.5GB of data will be loaded per batch. This substantial amount of data loading has to be completed within tens of milliseconds, making this stage very memory-intensive, and thus demanding efficient usage of the memory hierarchy.

Next, we discuss the properties of the DLRM production traces. Prior research efforts targeting inference [49, 100, 107, 115, 136, 137, 140] have focused on input datasets that are limited in various dimensions, involving a small number of embedding tables, a low pooling factor (number of lookups per user), homogeneous (in accessing table rows) tables, or a high concentration of hot table rows. In contrast, the production traces from Meta [33] contain hundreds of embedding tables and observe significant variations in table properties such as the number of lookups per input, row sizes, and memory access patterns. We refer to this industrial dataset as the heterogeneous dataset.

Figure 4.3 describes various non-uniformities for a 110 GB sized model containing 129 tables. We observe the following: (1) Table sizes range from 100 to 5 million rows,

which implies diverse memory capacity per table. For instance, tables with memory capacity under 1MB (2K rows) or 24MB (48K rows) can fully reside in the L2 or L3 caches, respectively. (2) Memory access patterns differ across tables. To quantify this, we use two metrics: unique indices and 10% coverage. Unique indices show the fraction of table rows touched by the dataset, while 10% coverage represents the percentage of lookups captured by the top 10% of the unique indices. It is evident that the production traces exhibit a wide range of hotness levels that imply varying cache utilizations across tables. (3) The maximum pooling factor across the tables ranges from zero to 164, which implies diversity in the raw amount of pooling operations across tables. For instance, a pooling factor of zero implies that there is no work associated with the table. Finally, the variations in table size, pooling factor, and memory access patterns influence the execution time per table. Figure 4.4 shows the sorted normalized execution time of each table. We observe that the time per table can drastically vary, by as much as $37.8\times$.

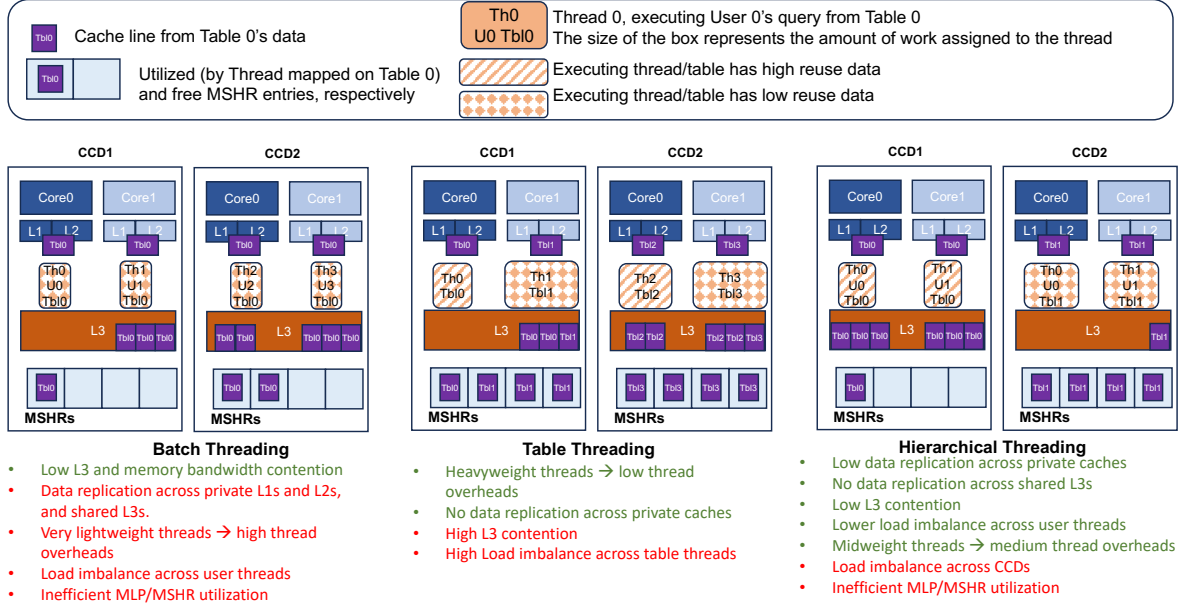


Figure 4.5: State of the art parallelization techniques.

4.2.2 Limitations of Prior Parallelization Approaches

To accelerate the DLRM embedding stage, shown in Algorithm 7, Jain et al. [136] proposed parallelizing the embedding stage execution at a coarser granularity by mapping each batch to a different CPU core. However, given the trend of growing model and dataset sizes [50, 148, 149], Nair et al. [140] proposed novel ways to parallelize the work

assigned to embedding tables as described in Figure 4.5.

BT is the default strategy used by popular frameworks such as PyTorch [144] and IPEX [143]. BT maps the work of a single embedding table (within a batch) across all cores in a SoC. Thus, only one table is processed at any given time, allowing the table to utilize the entire on-chip cache capacity. However, BT can encounter three challenges: (1) inefficient use of on-chip cache resources due to embedding table data replication in private and shared caches and due to the presence of tables with a small number of accessed rows; (2) runtime overheads due to frequent fine-grain thread synchronization to perform reductions and thread join/fork activity; and (3) load imbalance due to straggler threads (some users accessing more embedding table data than others).

TT, proposed by Nair et al. [140], assigns the work of each table to one CPU core. In this case, for an N -core CPU, N tables are processed in parallel. Thus, TT spawns fewer threads, which reduces runtime overheads and more efficiently uses private cache capacity. However, TT can suffer from (1) cache thrashing at the L3 cache and (2) load imbalance because threads process varying amounts of work per table. TT’s load imbalance is generally much higher than BT, especially when the work of the tables with many accessed rows is mapped to one CPU core or to the cores of the same CCD simultaneously.

Finally, HT capitalizes on the advantages and mitigates the weaknesses of BT and TT. It maps the work of each table to one CCD.² In this case, for a D -CCD CPU, D tables execute in parallel. However, HT could still suffer from various issues, as we discuss next.

We study the multi-core scaling of prior schemes as shown in Figure 4.1. “Homogeneous random” refers to the random dataset used in recent works [136, 150–153]. We observe that HT scales up to 96 cores on both datasets and performs better than BT and TT by up to $5.2\times$ and $1.56\times$, respectively. Thus, we consider HT as the most performant parallelization scheme. However, HT’s performance speedup with the heterogeneous dataset is $2.6\times$ lower than the speedup over the homogeneous dataset. In addition, HT’s speedup of $13\times$ is much lower than the ideal speedup of $96\times$ with all 96 cores. Therefore, there exists a substantial gap (and opportunity) between the achieved and the ideal performance scaling.

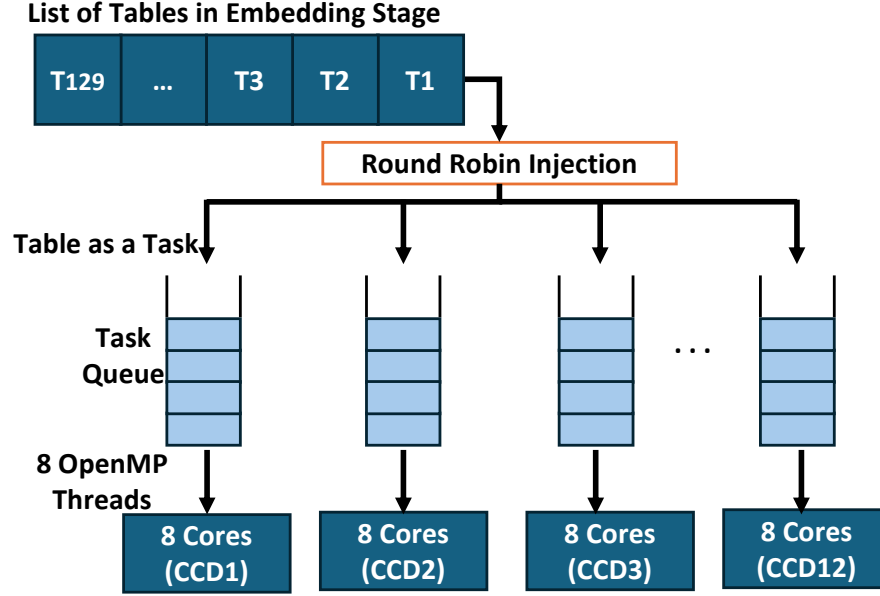


Figure 4.6: HT is implemented with the PyTorch thread pool. Each CCD is associated with a task queue and tables are distributed across task queues using round robin injection.

4.2.3 Unboxing the Design of Hierarchical Threading

To implement HT, the processing of a table is multi-threaded across CCDs using the “pytorch.jit” library [154]. Specifically, in the top-level python code (Algorithm 7), “torch.jit._fork” is used to spawn the tasks that process the work of a table, and “torch.jit._wait” is used to collect the embedding output matrix from each task. These spawned tasks are managed using PyTorch’s backend C++ thread pool implementation [147]. A task queue is coupled with a CCD to buffer the tasks to be completed by the CCD. Once a CCD receives a task, the batch amount of work associated with the table is multi-threaded across the CCD’s eight cores using the OpenMP library. The OpenMP parallelization is implemented in the embedding bag operator.

Figure 4.6 provides an overview of HT. First, a list of tables present in the embedding stage are assigned to tasks as they enter the thread pool. Second, these tasks are enqueued into different task queues using a round-robin injector that aims to balance the distribution of tables across the queues. Third, the scheduler dequeues a task from its corresponding queue one at a time and executes the work using all of the cores of its assigned CCD.

²In 96-core AMD 4th Generation EPYC™ Processors, a CCD is a group of eight CPU cores that share a L3 cache. Different AMD CPUs can have one or more CCDs per socket.

We believe three key issues are present with the HT scheduler:

1. *Load imbalance*: Because the queues are balanced in terms of the number of tables and not the execution time for processing the tables, several heavy tables could be mapped to one task queue. Furthermore, when the task queue associated with a CCD gets empty, the CPU cores become idle despite tasks pending in other queues. This issue can lead to straggler threads.
2. *Suboptimal core allocation*: Despite the non-uniformity of work across tables shown in Figure 4.3, HT assigns the same number of eight cores for the work of each table. This leads to a wide variation of per-thread execution time and lower core utilization. Moreover, tables with less work are more vulnerable to runtime overheads.
3. *Memory level parallelism (MLP) imbalance*: When the work of a low-reuse table executes on a CCD, L3 cache MLP resources (such as MSHRs) could get over-subscribed and create backpressure on the CPU cores. Similarly, when multiple low-reuse tables execute together on different CCDs, memory bandwidth could be saturated. On the other hand, when high-reuse tables execute on one or more CCDs, L3 cache MLP and memory bandwidth resources could be underutilized.

4.2.4 Modeling and Characterizing Table Workloads

Because HT maps the work of each table across CCDs without considering the per-table work, it is possible that many heavy tables are mapped to one or a few CCDs, creating load imbalance issues. For example, in the worst-case workload distribution scenario, we observed that the CCD with the highest workload performs $37.8\times$ more work than the CCD with the least amount of work, causing severe underutilization in the remaining eleven CCDs. Thus, we need to make more efficient work per table-to-CCD assignments.

For a given model and data set, the load or work per table is impacted by the embedding dimension, batch size, pooling factor, and memory access patterns. To effectively differentiate the work across tables, work can be modeled as a function of pooling factor and memory access patterns. Pooling factor defines the gather-reduce operations per table (Algorithm 7). However, pooling operation time depends on the latencies of the underlying loads, which could be significantly high with low-reuse table row access patterns [136]. To quantify the work associated with a given table, temporal locality can be used to estimate the load instruction latency.

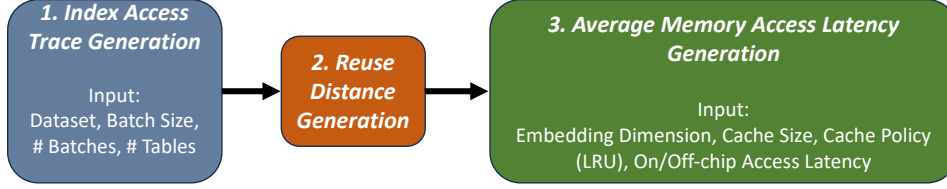


Figure 4.7: Reuse distance based analytical model to calculate the average memory access time of the loads accessed in an embedding table.

To approximate the work per table, we take inspiration from Jain et al. [136] and introduce a heuristic called the table load (TL) that is the product of average pooling factor (APF) and average memory access time (AMAT). For a given batch size, APF is calculated by averaging the total table row indices accessed in the batch. Figure 4.7 describes how to calculate AMAT: (1) track table row indices as a proxy for unique memory addresses visited and collect a trace of all touched table row indices in the first batch; (2) calculate reuse distance over this trace by tracking the number of unique row indices touched in between two instances of the same row index; (3) calculate AMAT by marking each row index access as a cache hit or miss, and attaching an average cost expressed in time units. We assume an embedding dimension of 128 elements (512B) and an average cost of 30 time units for a L3 hit and 450 time units for a L3 miss based on memory latency measurements on actual silicon [155], thus giving a ratio of 1:15 for on-chip versus off-chip access. Further, we identify row index accesses as hits or misses by simulating a fully-associative, 32MB L3 cache with LRU replacement. This effectively defines the maximum recorded reuse distance to be $32\text{MB}/512\text{B} = 64\text{K}$ row index accesses.

To illustrate the above steps, consider a sample table with an APF of 81.3. The model outputs the on-chip cache hit rate as 67.4%. Thus, AMAT is $(67.4 \times 1 + (100 - 67.4) \times 15)/100 = 5.57$ and TL can be calculated as $81.3 \times 5.57 = 452.7$. Figure 4.8(a) depicts the AMAT values for a list of sample tables, and confirms the presence of heavy variation across the reuse behavior within tables. Also, Figure 4.8(b) compares the TL metric with the measured execution time and the high correlation provides high confidence in its applicability. Notably, prior studies [50, 116] have estimated the work per table using only APF and have missed the impact of temporal locality. In contrast, we account for both and that enables us to more accurately compare the work across tables. Specifically, Figure 4.8(c) illustrates that TL and APF values vary across tables. For example, while the execution time for Table ID 3 is only 56% higher than Table ID 6, the APF of the former is $3.6\times$ larger than the latter, resulting in significant inaccuracies when comparing

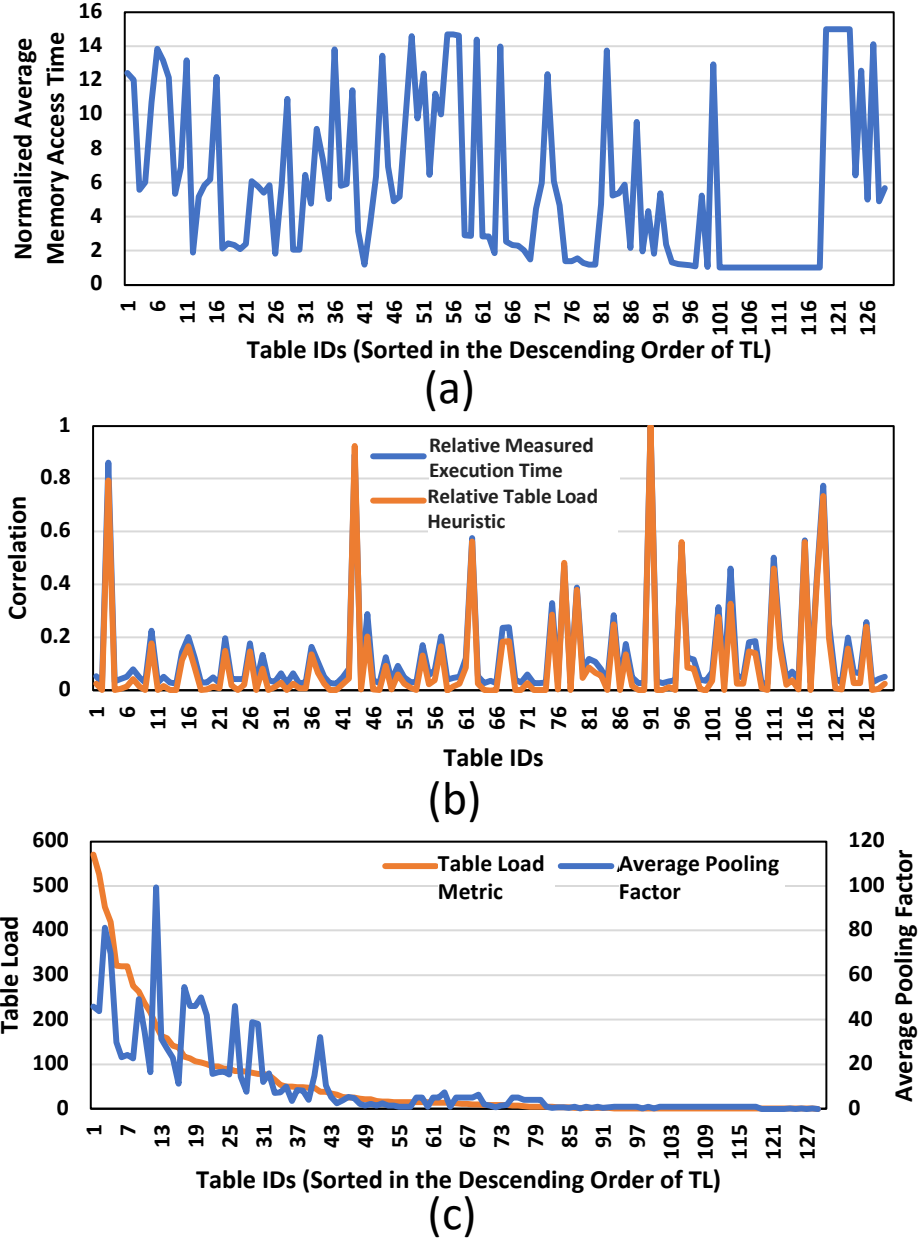


Figure 4.8: (a) AMAT across tables derived using our analytical model. (b) High correlation is observed between TL and measured execution time. (c) When comparing the decreasing trend of TL with pooling factor, we note that pooling factor can incorrectly represent work per table.

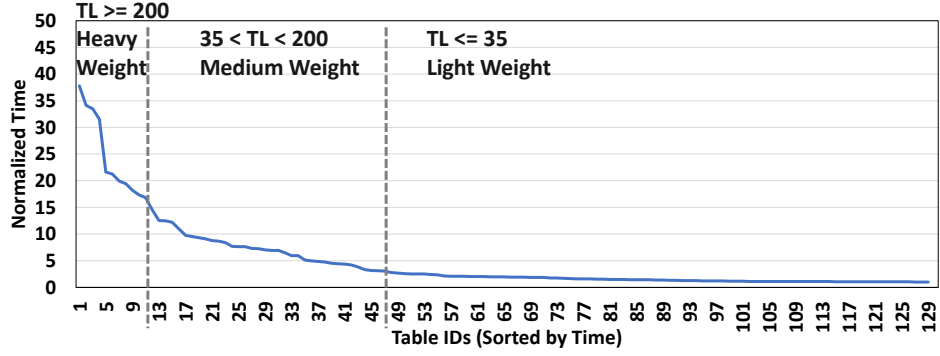


Figure 4.9: Using different TL metric thresholds, the embedding tables can be categorized as heavy, medium, and light-weight.

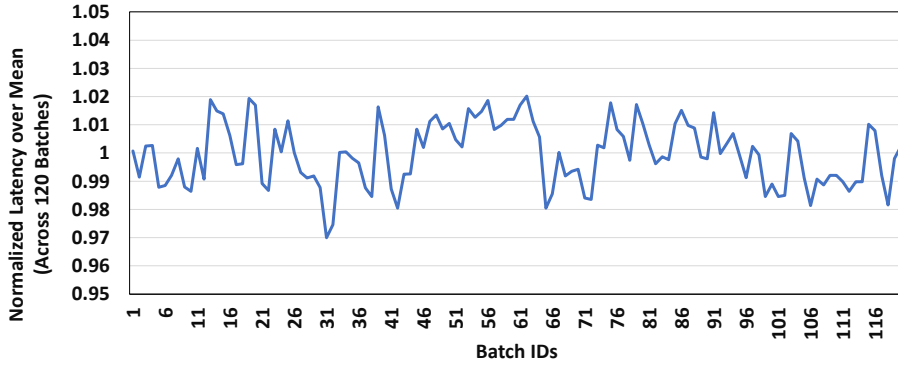


Figure 4.10: Deviation study of embedding stage execution time across 120 batches. Minimal deviation is observed.

the work across tables. AMAT more closely approximates the work per table.

Finally, we cluster tables based on their TL. Figure 4.9 sorts the tables in descending order of their execution time and categorizes them into heavy, medium, and light-weight classes using TL. This categorization helps reduce load balancing across CPU cores by avoiding straggler threads assigned the work of heavy tables as described later in Section 4.3.

Discussion on applicability of TL: We believe that it is sufficient to derive the TL value from profiling a few batches because we note that table properties, like execution time, APF, and AMAT, exhibit minimal variation across batches. Figure 4.10 confirms the robustness of TL across batches. If the DLRM embedding tables are updated, which may happen every few hours after serving hundreds of thousands of batches [49, 149], the TL metric can be very quickly regenerated over the table row access patterns of new batches. Furthermore, the TL metric is more accurate compared to other profiling techniques [156, 157], which depend on silicon-specific profiles such as actual cache hit

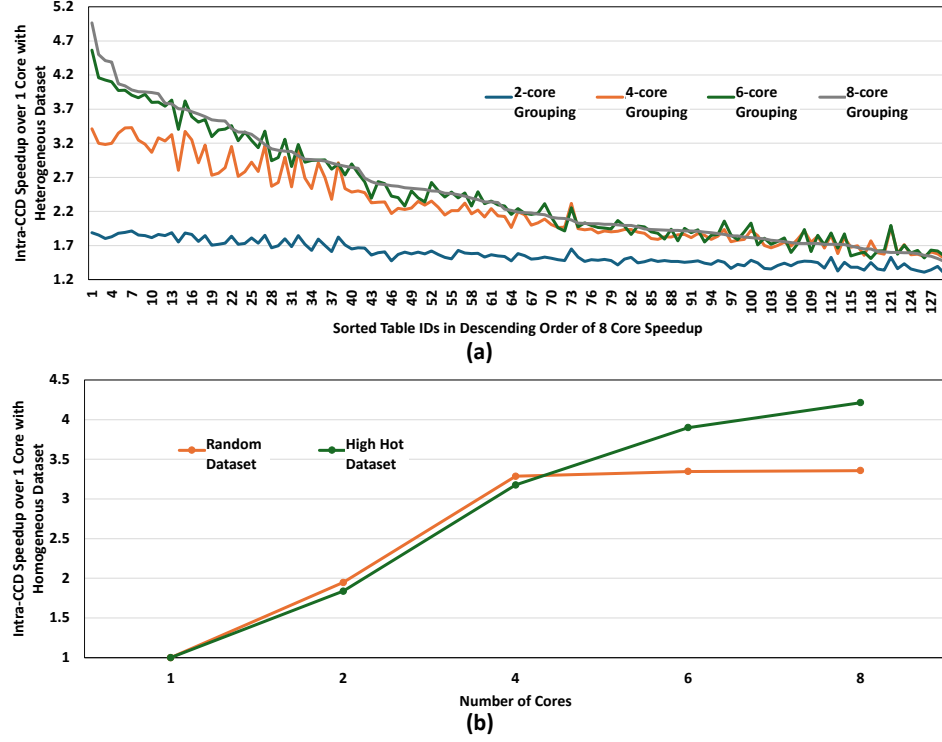


Figure 4.11: Characterization of intra-CCD multi-core scaling of embedding tables. (a) Work of each table scales differently with the number of cores, and only a few tables benefit from 8 or 6 cores. (b) Unlike a high reuse table whose works benefits from using up to 8 cores, the work of a low reuse table stops scaling after 4 cores.

rates or incur additional runtime overheads. The TL metric approximates work per table based on properties of the application and high-level table row-level reuse profiles.

4.2.5 Per Table Multi-Core Scaling Behavior

The previous section studied the per-table heterogeneous behavior, especially in terms of execution time and reuse of per table row accesses and found wide variations across tables. We now study how the work per table scales with the number of cores per CCD because HT maps the work of each table to all cores per CCD (8 cores in our target SoC). Figure 4.11 shows the scaling of embedding tables over various core counts. We note that five tables benefit from using all eight cores, 35 tables benefit from six cores, and the remaining long tail of 89 tables benefit from two or four cores. Thus, only a small fraction of tables can fully utilize all eight cores, confirming that prior schemes tend to over-provision the number of cores for the majority of the tables.

To identify the minimum number of cores needed for optimal scaling of a table,

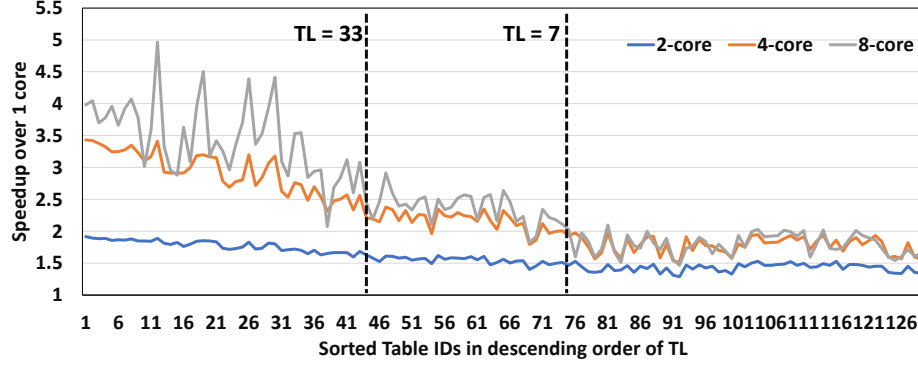


Figure 4.12: Speedup using 2, 4, and 8 cores over tables sorted in descending order of work per table (TL).

Figure 4.12 plots the realized speedup of per-table work using 2, 4, and 8 cores. The tables, in the x-axis, are sorted in descending order of their TL. We note that tables with larger TL benefit from more cores. In other words, there is a correlation between speedup and TL. Thus, we can use the TL metric to cluster tables based on the optimal number of cores, as shown in Table 4.1.

Core Count	TL > 33	33 ≤ TL < 7	TL ≤ 7
2	1.6 – 1.9x	1.5 – 1.6x	1.3 – 1.5x
4	2.5 – 3.5x	2 – 2.5x	1.5 – 2x
8	3 – 5x	2 – 3x	1.5 – 2x

Table 4.1: TL can predict the range of performance scaling for any given table.

4.2.6 Microbenchmarking Memory Level Parallelism

To complete the gather-reduce tasks as part of the pooling operation, the CPU cores are essentially dependent on the rate of data supplied by the memory hierarchy. This highlights the importance of understanding how memory access patterns coupled with a given memory hierarchy organization impacts multi-core scaling. Figure 4.11(b) evaluates the intra-CCD multi-core scaling for two extreme table row access patterns. We observe that the tables with high reuse behavior (high hot data set) scale up to eight cores. However, tables with low reuse behavior (random data set) do not scale after four cores. This suggests a resource limitation in the L3 cache’s ability to exploit the MLP needed to service the off-chip requests generated after more cores are tasked to execute the work of the tables.

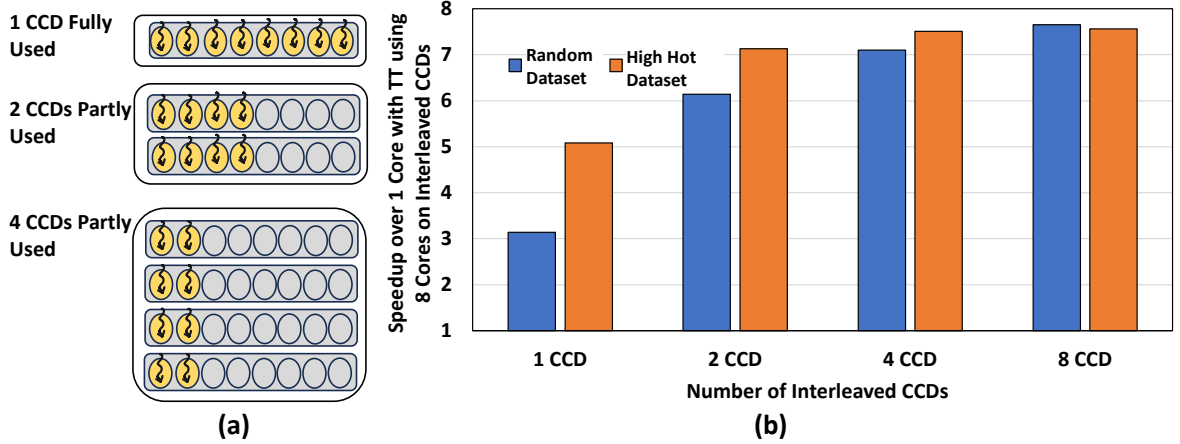


Figure 4.13: Eight thread execution using multiple CCDs with TT over different memory reuse patterns.

To highlight the limited L3 cache MLP per CCD, we conduct a systematic experiment that isolates the number of cores from the memory hierarchy. We do this by fixing the number of threads and cores to eight and vary the number of CCDs to increase the available memory hierarchy for the eight threads. Specifically, we utilize the eight cores by distributing the threads across CCDs. Figure 4.13(a) highlights how eight threads can be distributed across one or more CCDs. It is important to note that even though the number of cores remains the same, the memory hierarchy observed by the threads highly differs. For example, when eight threads are using a single CCD, they share the L3 cache capacity and MSHRs. However, when eight threads are mapped to one core per each of eight different CCDs, each thread uses the entire L3 cache capacity and MSHRs which can minimize the number of misses and maximize the number of simultaneous miss requests per thread.

Given that any two embedding tables do not share data, we use TT for the 8-core scaling experiment. This prevents any sharing of data among threads of the same table. Figure 4.13(b) shows the multi-core scaling of TT for low-reuse and high-reuse tables. Speedups up to $7.7\times$ are achieved over the low-reuse case, confirming that higher L3 MLP helps in increasing the data supply rate to the cores. This is mainly due to an increase in outstanding L3 cache misses because multi-core scaling for the high-reuse tables, which have lower memory footprint and better cache hit rates, saturates with two cores.

Because MLP requirements vary per table, the MLP resources per CCD could get oversubscribed. Thus, it becomes useful to predict the MLP demand per table and

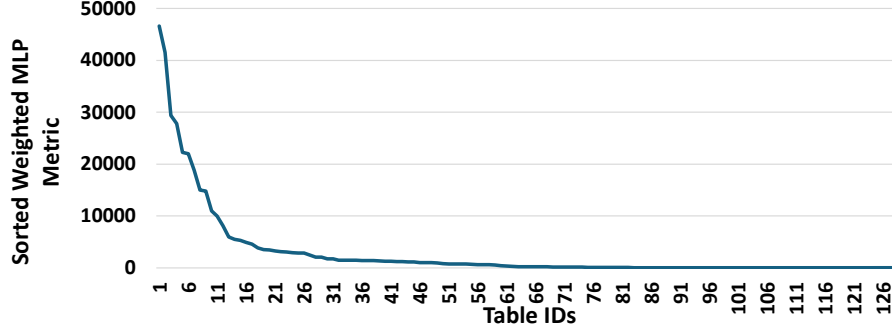


Figure 4.14: Weighted memory-level-parallelism (WMLP) metric models the amount of MLP demand of each table.

leverage this information by the thread scheduler. We develop a new metric called Weighted-MLP (WMLP) that captures the MLP required for a table. It is defined as the product of TL and L3 cache hit ratio (%). Figure 4.14 shows the WMLP variation across tables, which ranges from 0 to 46600. Tables with high pooling factors and high cache miss rates generate a large WMLP. On the other hand, tables with low pooling factors and low cache miss rates have a small WMLP. A few tables, which have zero pooling operations, exhibit a WMLP value of zero. Overall, significant variations in the WMLP are observed.

4.2.7 Key Architectural Insights for CPU Multi-Core Scaling inefficiencies

Based on the above characterization study, the following architectural insights can be drawn:

- Drastic differences in the work per table are possible, which can lead to load imbalance across CCDs. The work per table can be accurately modeled using the TL heuristic, which could be used to better map the tables onto cores.
- Allocating eight cores per table is excessive for the majority of tables. Instead, an appropriate amount of cores can be predicted based on the TL of each table. Specifically, two or four cores per table could help better utilize the cores.
- Available MLP in a given CPU impacts the multi-core scaling. The WMLP metric predicts the MLP demand of each table and can be used to distribute low-reuse tables across CCDs to improve scaling.

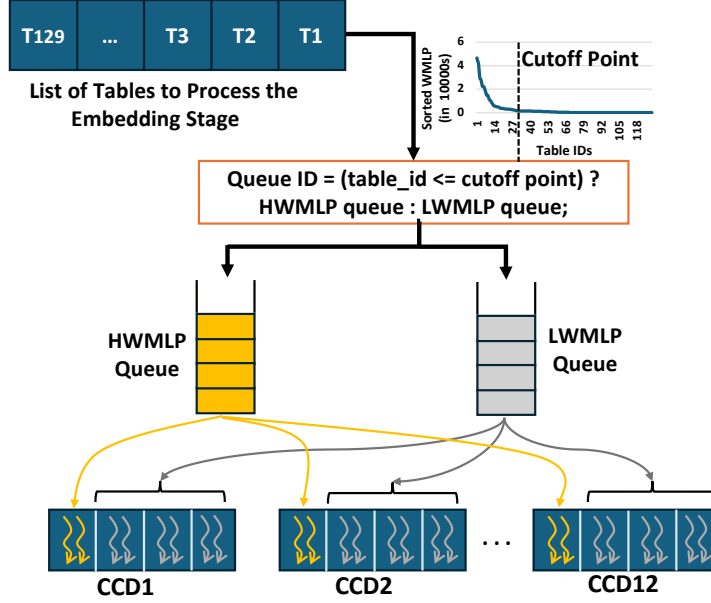


Figure 4.15: Balance is demonstrated for two-core grouping. Tables are injected into the task queues in decreasing WMLP order. Two task queues are used to separately schedule the work of heavy (HWMLP) and light (LWMLP) tables based on a cutoff point. At any point in time, each CCD is assigned only one HWMLP table and one or more LWMLP tables. Work stealing improves load balancing by allowing a CCD to seek work from a secondary queue after a primary queue gets empty.

4.3 Load- and MLP-Aware Thread Scheduling

In this section we describe our novel thread scheduler called “Balance.” We modify the original HT-based thread pool implementation (Figure 4.6) by adding WMLP-based table ordering, MLP-conscious task scheduling using two task queues across all CCDs per SoC, low core grouping per table and work stealing. Figure 4.15 shows an overview of our proposed thread scheduler. Note that the embedding stage execution is bulk synchronous and therefore its performance is dictated by the total completion time of all tables and not by the completion time of individual tables.

Low-core grouping. Figure 4.11 showed that the number of cores, beyond which performance of per-table work does not scale, varies across tables. If the work per table does not scale up to all cores in a CCD, the unused cores can be assigned to work on other tables, enabling higher resource utilization per CCD. Low-core grouping allows per-table work to be mapped to the minimum number of cores needed to scale performance. In order to keep the HT scheduler complexity down, we propose a simplified static allocation of either two or four cores per table. Figure 4.15 shows a two-core allocation for all the

tables. Once a new task is received, the work associated with the table is mapped onto the designated number of cores (two or four) by leveraging OpenMP threads. Compared to eight-core grouping in HT, since the work per thread is two and four times higher in the four-core and two-core grouping, respectively, we expect lower runtime overheads compared to HT.

MLP-aware scheduler. Embedding tables exhibit different MLP requirements as noted in Section 4.2.6; therefore, for each inference task, we separate the tables into two groups: HWMLP and LWMLP. Each group is sorted in descending WMLP order. The scheduler employs two queues. Tables from the HWMLP (LWMLP) group are injected into the HWMLP (LWMLP) queue. The work for each table is mapped to two or four cores based on its predefined core grouping. The scheduler first starts mapping the HWMLP tables across the CCDs in a round robin fashion, limiting a maximum of one HWMLP table per CCD. This limit typically leaves enough MLP capacity for LWMLP tables to be co-located on the same CCD. Figure 4.15 shows an example with one HWMLP table co-located with three LWMLP tables in every CCD. This approach provides a good tradeoff between maximizing the number of tables being scheduled simultaneously and utilizing MLP resources per CCD. Note that if we fully utilize MLP resources per CCD, we also maximize memory bandwidth usage across the SoC because memory bandwidth depends on the in-flight requests sent by all the CCDs to main memory.

The MLP-aware scheduler has several advantages over the baseline HT scheduler. First, it uses only two queues instead of a number of queues equal to the number of CCDs in the SoC. This reduces scheduling overheads, which is now independent of the number of cores per CCD. Second, low-core grouping maximizes core utilization. Third, MLP-conscious scheduling allows for higher utilization of MLP and on-chip cache resources per CCD. Fourth, the combination of MLP-conscious table scheduling and low-core grouping improves load balancing compared to the baseline HT because it allows for a more balanced distribution of work across the CCDs. In contrast, the HT scheduler maps each table to a CCD independent of its MLP or compute requirements. Fifth, the Balance scheduler avoids scheduling multiple HWMLP tables onto the same CCD, therefore reducing the probability of straggler threads compared to the HT scheduler.

Work Stealing. Even though MLP-based table scheduling balances the work assignment across the two task queues, there can still exist cases where load imbalance may arise if, in one inference task, a HWMLP group contains far fewer tables than the LWMLP group. This scenario can lead to cases where one of the task queues is empty

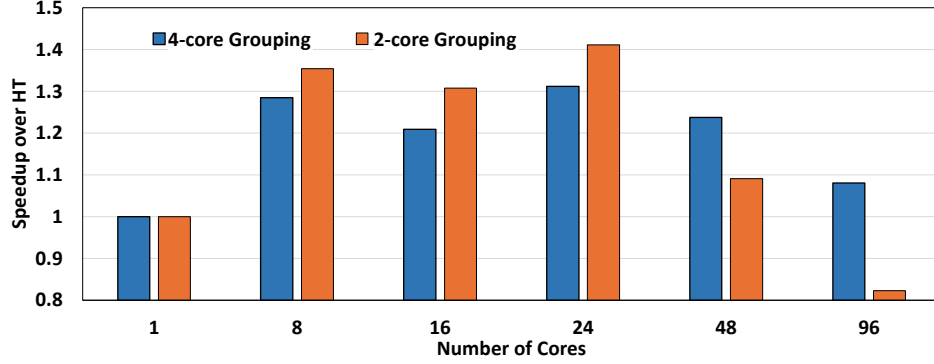


Figure 4.16: Speedup of Balance with two and four-core grouping over HT using various core counts and batch size of 1024.

while the other still has more work to schedule. In order to address this, we employ work stealing. When one of the queues (HWMLP or LWMLP) is empty, we steal the work of a table from the other queue. Thus, the scheduler will continue scheduling work from both queues.

Synergy between Balance scheduler features. We believe each of the above proposed features can complement each other. To understand this, we consider three pathological scenarios: (1) if a DLRM inference task includes a skewed distribution of embedding tables (e.g., HWMLP versus LWMLP), MLP-conscious table reordering and work stealing ensures that Balance efficiently distributes tables across cores, lowering the chances of straggler threads; (2) if a DLRM inference task must process embedding tables with very small TL values (similar to the tail in Figure 4.12), low-core grouping ensures that cores remain utilized; (3) if an inference task processes embedding tables with either very high-reuse or very low-reuse memory accesses, MLP-aware scheduling ensures that caches and MLP resources remain highly utilized. Thus, we expect Balance to perform better in all of the above challenging scenarios compared to a HT scheduler.

4.4 Methodology

Model and Datasets: We use an embedding dimension of 128 and a batch size of 1024 (unless mentioned otherwise). Our experimental evaluation includes two sets of traces classified as “homogeneous” and “heterogeneous.” The homogeneous dataset exhibits uniform properties across tables such as table size, batch size, pooling factor, and embedding dimension for each inference task. More specifically, we consider an inference task with 96 embedding tables, 500K rows each, and 150 lookups per user (pooling

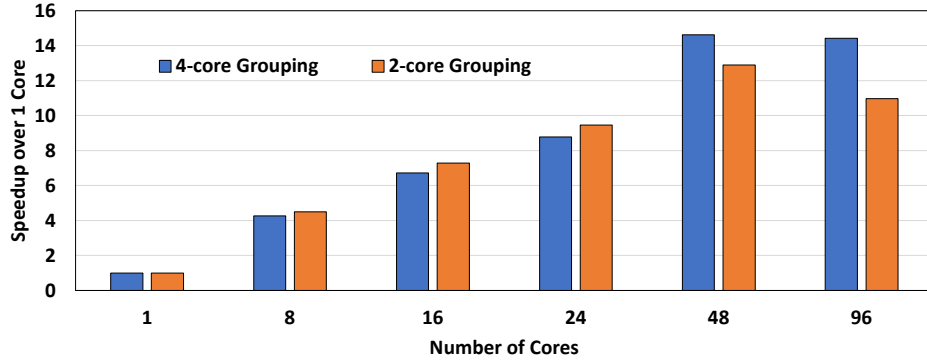


Figure 4.17: Speedup scaling for Balance with two and four-core grouping over single core Balance and batch size of 1024.

System	Sockets	1
	PCIe	128 lanes Gen5
SoC	Cores per SoC	96
	CCDs per SoC	12
	Memory BW	460GBps across 12 DDR5 channels
	Memory Capacity	768GB per socket
	TDP	360W
CCD	Cores per CCD	8
	L3 per CCD	32MB
Core	L1 Cache (instr/data)	32KB/32KB
	L2 Cache	1MB
	L3 Cache	4MB
	DTLB entries (L1/L2)	2048/3072
	Frequency	3.5GHz

Table 4.2: 96-core AMD 4th Generation EPYC Processor Configuration [3].

factor). The homogeneous datasets are obtained from [136] and are categorized based on reuse as high hot, medium hot, low hot, and random. The heterogeneous datasets are production traces from Meta [33] and possess non-uniform properties across tables such as table size, pooling factor, and memory access patterns, as described in Section 4.2.1. For a batch size of 1024, a total of 128 batches are run, where the first eight batches are marked as cold execution due to initial function calls and performance is averaged over the later 120 warm batches.

Hardware: Table 4.2 describes the hardware configuration of our system setup. In

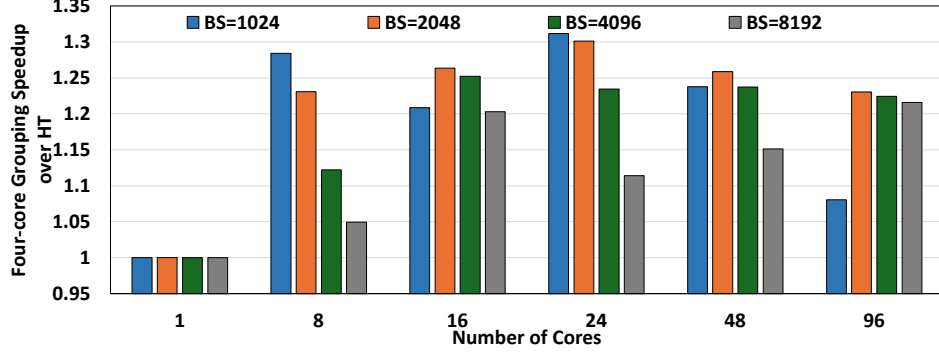


Figure 4.18: Speedup of Balance with four-core grouping against HT over various core counts and batch sizes (BS).

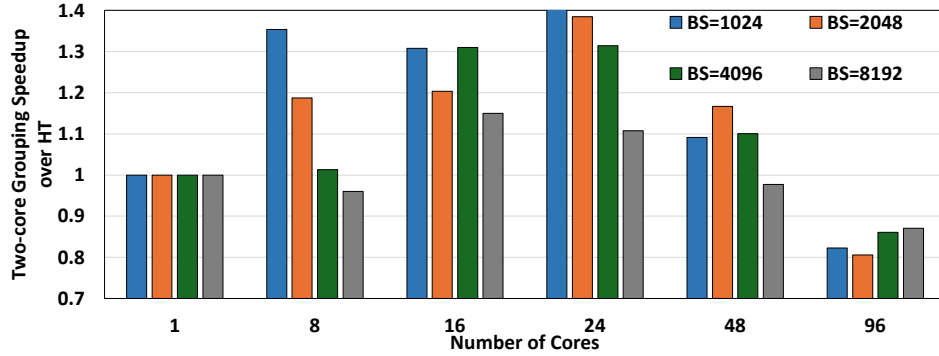


Figure 4.19: Speedup of Balance with two-core grouping against HT over various core counts and batch sizes (BS).

all of our evaluations, performance is studied for a one-socket processor. A core complex die (CCD) refers to a group of eight cores sharing a L3 cache in our target CPU.

Software: PyTorch (v1.13.1) is source-compiled along with the ZenDNN library featuring an AVX512-based embedding bag operator implementation. GCC 11.4.0 is used for code compilation. Our setup runs under Ubuntu®/Linux® (Ubuntu 22.04.2 LTS) with the 5.19.0-32-generic kernel.

System: We enable Transparent Huge Pages (THP) and frequency boost. SMT and stream hardware prefetchers (L1 and L2) are disabled. THP is enabled to take the address translation overhead out of the critical path of the embedding bag (EB) kernel. This increases DTLB hit rates and, as a result, batch latency improves by 20-30%. Stream prefetchers do not help with the irregular memory traffic associated with the EB kernel. SMT does not help the EB kernel performance due to limited MLP per thread [136].

4.5 Evaluation

In this section, we evaluate the performance of the Balance scheduler. Given that HT is the most performant parallelization scheme for the embedding kernel of DLRM, we compare HT against Balance using two-core and four-core groupings. The two (four) core grouping allows for a maximum of four (two) tables to be processed per 8-core CCD. All evaluations are conducted using the heterogeneous datasets.

4.5.1 Main Results

Here we evaluate the threading schemes over a model with 129 tables. Figure 4.16 compares Balance and HT for a batch size of 1024. Balance with either four or two-core grouping achieves up to $1.3\times$ and $1.41\times$ speedup over HT for a 24-core configuration while scaling stagnates with more cores mainly due to the higher memory traffic in the data fabric (DF, network on chip) as we schedule work on more CCDs. However, the four-core grouping improves performance over HT even with 96 cores. Performance scaling over a single core with Balance (and stagnation at 48 cores) is also shown in Figure 4.17.

To increase the work per thread, we evaluate larger batch sizes (2048, 4096, and 8192) as shown in Figure 4.18 and Figure 4.19. Four-core grouping achieves consistent speedup over HT across all batch sizes and core counts, while two-core grouping does not always, especially with larger batch sizes. This is because two-core grouping allows a larger number of tables to be processed in parallel generating more L3 misses and DF memory traffic at a given core count than four-core grouping. Figure 4.20 and Figure 4.21 show similar comparisons but over a single core implementation. A higher batch size implies more work per table, and thus better amortizes thread creation/deletion overheads allowing Balance to better scale performance. Performance improvements over single core peaks at $21.65\times$ with a 8192 batch size. Thus, Balance is able to achieve $1.67\times$ better scaling over 96 cores compared to HT with batch size of 1024. Moreover, the absolute embedding bag kernel latency under Balance is 2.5msec, which can better meet real-time latency requirements.

In Figure 4.19, Balance with two-core grouping does not achieve consistent performance improvements with higher core count and batch size. A similar observation is made with table threading and 96 cores (Figure 4.1). This performance anomaly is because a few high-work tables are mapped to the same CCD, taking longer to complete the task. Our initial studies found that varying the cutoff point impacts performance. Thus, we

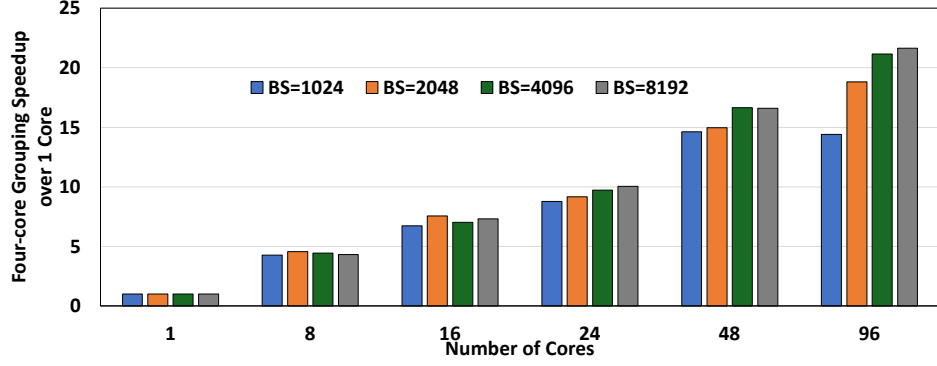


Figure 4.20: Four-core grouping speedup over a single core.

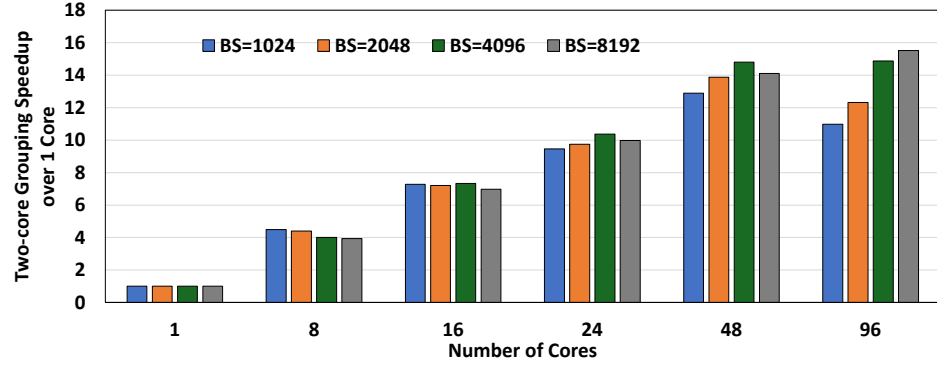


Figure 4.21: Two-core grouping speedup over a single core.

expect modifying the cutoff point can prevent this behavior via better distribution of the tables over CCDs and two-core worker groups.

4.5.2 Sensitivity Studies

In this section, we evaluate threading schedulers over a DLRM model with 400 tables.

1) *Impact of cutoff point and work stealing:* Figure 4.22 shows the performance of work stealing in the Balance scheduler with a batch size of 2048 and 48 cores under various cutoff points. Without work stealing, we observe wide performance variations across the cutoff points. The deviation is higher for two-core grouping because the latter is more sensitive to uneven table distribution.

Performance deviation is largely reduced under work stealing because it dynamically balances the load across the workers. In addition, work stealing at the optimal cutoff point improves performance by up to $1.5\times$ and $2.06\times$ for four- and two-core grouping schedulers, respectively. We also note that work stealing was invoked 35.4 and 71.2 times

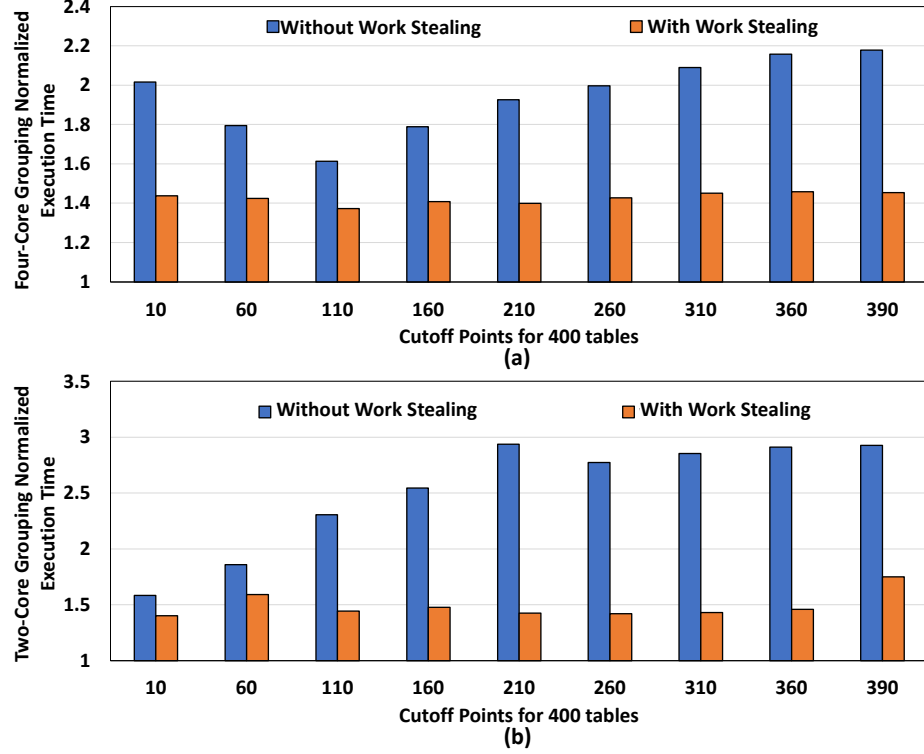


Figure 4.22: Impact of cutoff point and work stealing.

per batch by the four- and two-core grouping Balance schedulers, respectively. To keep the load balanced, the two-core grouping Balance scheduler coupled with a 1:3 worker ratio invokes work stealing more often than the four-core grouping.

2) *Boost in embedding stage performance:* Figure 4.23 compares the performance of Balance over HT for batch sizes of 1024 and 2048. Similar to Figure 4.18 and Figure 4.19, Balance continues to outperforms HT, especially with the four-core grouping scheduler, especially at the highest core counts.

3) *Memory bandwidth utilization:* Figure 4.24 compares the memory bandwidth of HT and Balance schedulers in a 48-core and 2048 batch size configuration. Balance schedulers achieve higher memory bandwidth utilization than HT by improving cache utilization (lower traffic to DRAM and faster execution time). However, two-core grouping sees a higher memory bandwidth utilization of $1.39\times$ even though the performance improves only by $1.2\times$. Because two-core grouping schedules up to four tables per CCD, it frequently suffers from higher shared L3 cache miss rates compared to four-core grouping, resulting in higher memory bandwidth due to higher traffic to DRAM.

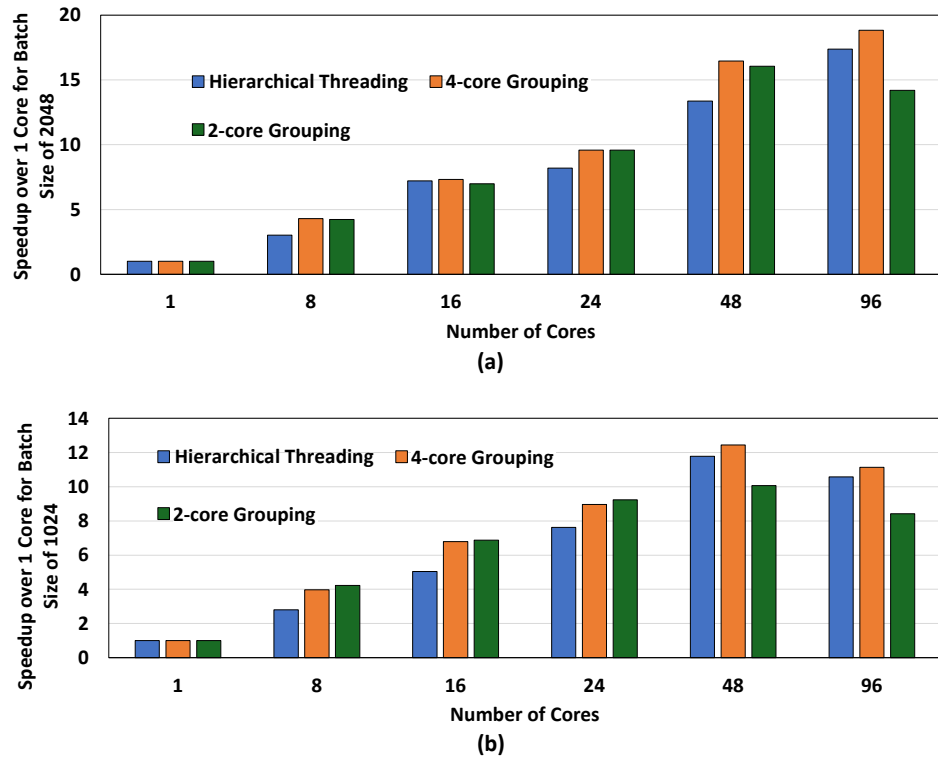


Figure 4.23: Speedup scaling comparison for a large model with 400 tables.

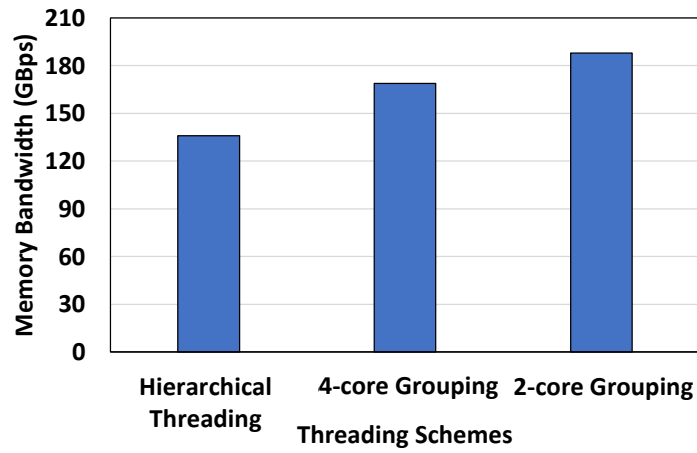


Figure 4.24: Memory bandwidth consumption of HT and Balance schedulers with 48 cores and 2048 batch size.

4.6 Discussion and Future Work

Applicability of Balance on non-AMD CPU-based SoCs: We believe that Balance is applicable to CPU-based SoC architectures that do not group cores in CCDs. The idea of core grouping is independent of the CCD architecture and primarily depends on embedding table properties. The WMLP ordering metric depends on both table properties and L3 miss rates. The L3 miss rates can be estimated using either a logically shared or distributed L3 cache. The ordering may be different, depending on the L3 architecture, but the WMLP-based table ordering metric is still applicable. HWMLP versus LWMLP table partitioning is also independent of the CCD architecture because it maps all tables into 2 queues (as opposed to a number of queues matching the number of CCDs in HT). Work stealing is independent of the SoC architecture. Finally, the allocation of threads to cores in Balance is aware of the CCD core count and the isolation offered across L3 partitions. However, it can be modified to better suit a shared L3 architecture. In one example, a Balance scheduler can virtually partition a shared L3 into fixed size clusters of N cores and schedule the tables to groups of consecutive, available cores to minimize inter-core communication latency and cross L3 NoC bandwidth.

Training: In DLRM training with GPUs, as shown in prior work [50, 53, 116, 158], communication overhead has been found to be the biggest performance bottleneck due to load imbalance. To resolve this, prior work has approximated the work per table with the pooling factor only. In contrast, as shown in Section 4.2.4, our TL heuristic more accurately captures the workload by accounting for off-chip memory accesses. Thus, for DLRM training, we expect our TL heuristics to balance table placement over a large pool of GPUs.

4.7 Related Works

Several prior works [28, 49, 100, 106, 107, 115, 121, 122, 132, 136–138, 140, 159–162] have studied and optimized DLRM inference, including DLRM inference on CPUs [100, 107, 115, 136, 140]. Jain et al. [136] proposed mapping each batch to a CPU core. However, the work per batch increases with larger models (containing more embedding tables and higher embedding dimensions) and data set sizes, thus making it difficult to meet real-time latency targets. Nair et al. [140] proposed parallelization of a batch using table threading (TT) and hierarchical threading (HT) and achieved significant performance improvements. However, as mentioned in Section 4.2.4, HT can suffer from scalability

issues, which can be resolved using our proposed Balance scheduler.

Various prior works have developed accelerators [28, 106, 109, 121, 122, 160–164]. However, these proposals require custom silicon and demand considerable time and effort to bring to market, which makes it challenging to adopt them given the rapidly changing model parameters. Instead, Balance requires only software changes and runs on CPUs, thus making it much easier to deploy.

4.8 Conclusions

Optimizing DLRM inference for CPUs can provide significant value to data center efficiency as well as improve user experience. Therefore, understanding the inefficiencies in existing DLRM inference tasks is important. The heuristics and performance metrics developed in this work (TL heuristic and WMLP metric) have been successful in mitigating the inefficiencies. In addition, we demonstrate how load imbalance across threads and CCDs, suboptimal core allocation for embedding tables, and inefficient utilization of MLP resources can be addressed by our proposed mechanisms and techniques. We believe our analyses and proposed solutions can open up further opportunities not only in optimizing DLRM inference, but also workloads with similar irregular memory access patterns and MLP characteristics.

Chapter 5 |

Pushing the Performance Envelope of DNN-based Recommendation Systems Inference on GPUs

5.1 Introduction

As highlighted in Chapter 1, modern recommendation systems provide personalized suggestions to enhance user experience through Deep Learning Recommendation Models (DLRM) [14]. The growing importance of DLRMs is evident in their widespread deployment by hyperscalers for both training and inference. This translates to a significant portion of AI inference cycles being dedicated to DLRMs [100], while being deployed on a variety of platforms including CPUs([34, 100, 136, 140, 159]), GPUs([28, 34, 107, 115, 137]), and accelerators([28, 106, 121, 122, 160–162]). While Chapter 2, 3, 4, and prior works [99, 140, 165] optimizes the single- and multi-core thread execution on CPUs, the compute parallelism and memory bandwidth of CPUs remain inherently limited. With the ever-increasing compute and memory requirements of DLRMs, it is increasingly being preferred to execute them on GPUs [28, 34] due to their efficient parallel processing capabilities. However, with growing model and dataset sizes, efficient utilization of GPUs for improving the performance of inference applications, as will be shown in this paper, is insufficiently investigated.

DLRMs primarily comprise four stages: embedding, bottom multi-layer perceptron (MLP), feature interaction, and top MLP. The latter three stages are marked as non-embedding stages. Prior works [100, 107, 136, 137, 140] have shown that the embedding stage is memory intensive (due to frequent and irregular memory accesses) and the non-embedding stages are compute intensive.

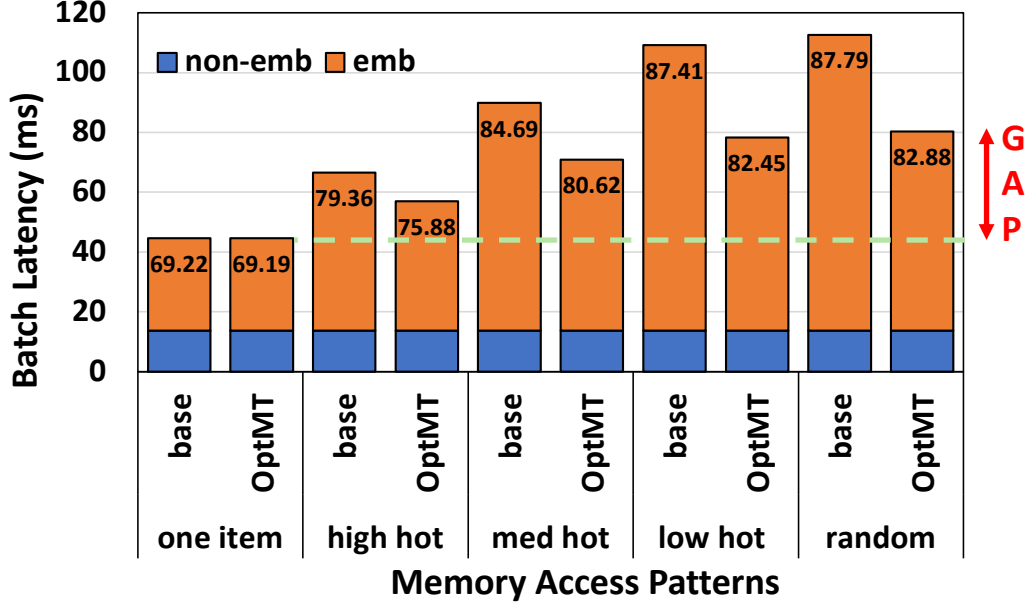


Figure 5.1: Shown is the degradation in inference performance as hotness lowers (working footprint decreases) from left to right. The numbers inside the bars indicate the embedding stage contributions. Here, OptMT provides higher WLP which enhances performance over off-the-shelf PyTorch (base). Yet, a significant gap continues to exist compared to the fastest loads (one item case). We cite this as the research *gap*.

Using the latest off-the-shelf PyTorch-based embedding bag CUDA kernels over an A100 GPU, we conduct an extensive characterization study over the latest DLRMs with production datasets. These experiments reiterate that the embedding stage continues to bottleneck the inference performance as shown in Figure 5.1. GPUs are well-known for their Multi-Threading (MT) or Warp-Level-Parallelism (WLP)¹ support to hide memory latencies with computation [166]. Unfortunately, the available parallelism is not enough. We observe up to $3.2\times$ embedding-only performance slowdown when comparing ‘random’ with the ‘one item’ case (referring to the fastest case where all embedding accesses point to one row in a table, leading to $\sim 100\%$ cache hits). This sub-optimal MT is observed in both ready-made packaged and source-compiled PyTorch implementations as they suffer from register pressure (details mentioned in Section 5.3.3).

To test if increasing the parallelism can address this issue, we synthetically increased the parallelism level to an optimal amount (Optimal MT (OptMT)).² We did so by lowering the register allocation per warp using available compiler optimizations. Although

¹We use MT and WLP interchangeably in the paper.

²Note that due to register spilling optimal MT may not be the maximum WLP supported by the GPU. We quantify this effect in Figure 5.6.

it did improve the performance (reduction in batch latency) by 53%, as seen in Figure 5.1, OptMT is insufficient as a significant performance gap continues to remain between the ‘one item’ and ‘random’ cases. Looking under the hood with detailed profiling (described in Table 5.4), we observe that both off-the-shelf PyTorch and OptMT implementations underutilize the “warp issue slots” and “average HBM read bandwidth”, demonstrating that the kernel is memory latency bound.

Previous DLRM-based works have cited the memory-bound issue arising due to embeddings, and have developed heterogeneous platform-based scheduling frameworks [107, 115, 137, 167], distributed inference strategies [43, 149], accelerator designs [28, 106, 121, 122, 160–162], and algorithmic-system designs [?, ?, 53, 114]. However, they are limited to either using out-of-the-box kernels or highly skewed (exhibiting high temporal reuse) datasets, and *none of the prior solutions address the long latency load stalls arising on a GPU platform*. Towards this, we explore easy-to-adopt design solutions by asking the following question: *Given the high adoption of GPUs by hyperscalars [168] even while being expensive, can we develop cost-friendly software techniques that are both application- and architecture-aware to alleviate the memory bottleneck?*

Building on our understanding of the unique characteristics of DLRMs, we leverage the features of modern GPUs to make the following contributions:

- To the best of our knowledge, ours is the *first work to study the architectural implications of DLRM inference on GPUs* and to point out the *microarchitectural inefficiencies leading to memory latency bottleneck (our research gap)*. Our in-depth *characterization* shows that out-of-the-box PyTorch DLRM implementation has several performance-related inefficiencies. First, out-of-the-box kernel is plagued with long latency load stalls (later described as long scoreboard stalls), leading to a significant performance gap across the spectrum of memory access patterns. Second, the kernel suffers from register pressure leading to limited WLP. The number of hardware registers is simply not enough to support the maximum number of warps allowable. Even if we allow an optimal number of warps by reducing register allocation per warp, there is plenty of scope for reducing latency further. That is, even optimal number of warps is simply *not* enough to hide the long memory latencies.
- We show that memory latency (not bandwidth) is a major performance-limiting issue for DLRM performance (embedding bag CUDA kernel). On top of optimal WLP, we present two plug-and-play hardware-software co-design based optimizations: (i)

by leveraging the bandwidth headroom of modern GPUs, their hardware-supported scoreboarding, and multiple memory resources as buffer stations, we perform **Prefetching** to hide memory latency stalls, and (ii) by taking advantage of known apriori power-law distribution in embedding accesses, we perform **L2 Cache Pinning** to pin the most frequently accessed entries by exploiting the new L2 cache residency control feature on GPUs. Further, we show these two designs can complement each other.

- We evaluate and compare the benefits of the proposed techniques. In isolation, prefetching and L2 pinning improve embedding lookups by up to 97% and 62%, and end-to-end inference by up to 73%, and 48%, respectively. Moreover, pinning and prefetching *complement* each other, and when combined, improve embedding lookups by up to 103%, and end-to-end inference by 77%. Finally, with their synergy, the worst-case performance gap significantly lowers by 163% over base PyTorch, and 53% over OptMT. Also, we believe our proposed designs can be generally applied to a wide range of memory-bound kernels.

5.2 Background

In this section, we discuss (1) the architecture of a modern recommendation system, (2) the key microarchitectural features of the latest GPUs, and (3) the related works on improving the DLRM and memory-bound kernel for GPUs.

5.2.1 DLRM Inference Using GPUs

Many industries use GPUs to execute DLRM inference [28, 34, 107, 115]. The primary steps in inference involve (1) a one-time loading of the complete model onto the GPU memory, (2) feeding the input batches (each batch is large and comprises a group of samples) to the GPU, and (3) executing the inference to predict the top-k items for each sample within a batch. For large models exceeding the memory capacity of one GPU, multiple GPUs/nodes are used with model and data parallelism [50, 158]. Regardless of the number of GPUs used, each GPU executes one or more embedding tables serially [146, 169].

Figure 5.2 shows a simplified diagram of a typical DLRM [14, 100]. Each sample comes with continuous (e.g., age, location) and categorical features (e.g., movie genre, item id). Former are fed to the Bottom MLP stage while latter are fed to the Embedding

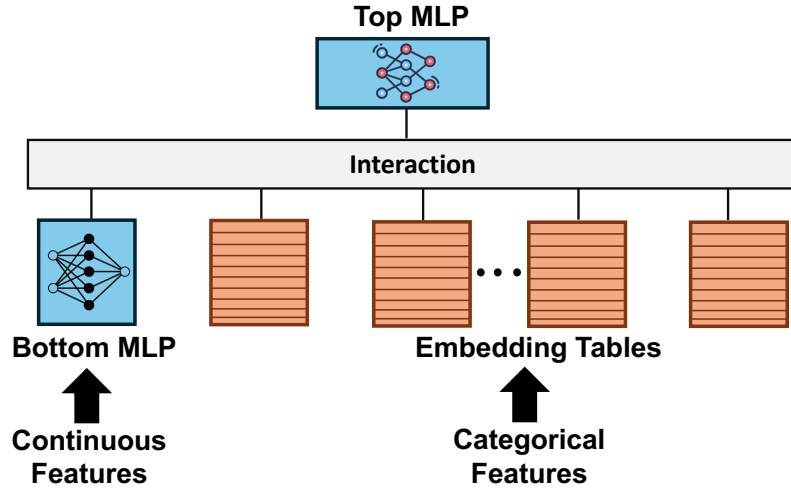


Figure 5.2: A schematic of a DLRM architecture. The continuous features (e.g., age, location) are processed by Bottom MLP, and categorical features (e.g., movie genre, item ID) by the Embedding Stage. Their outputs are combined in the Feature Interaction Stage, and then fed into the Top MLP, which predicts top-k items with highest Click Through Rate (CTR).

Stage. The feature interaction stage merges (concatenation/dot product) the outputs of the previous two stages, and feeds it to the Top MLP stage, generating the top-k items with the highest predicted CTR (Click-Through Rate). Several past works highlight the embedding stage to be memory intensive ([?, 121, 122, 136, 140]) and non-embedding stages to be compute intensive [107].

5.2.2 Key Properties of GPU Microarchitecture

GPUs, also known as throughput processing engines, contain a hierarchical array of compute cores (CUDA cores). Figure 5.3 shows a simplified diagram of the GPU organization. Modern GPUs (Nvidia based) contain 100s of SMs (Streaming Multiprocessors) [30, 170], and each SM offers 4 SMSP (Streaming Multiprocessor Sub-Partition). Each SMSP is associated with a warp scheduler and provides the capability to issue one eligible warp every cycle while maintaining a queue of resident warps, thus facilitating WLP or MT. Further, a scoreboarding mechanism [171] is adopted in the core pipeline to promote instruction level parallelism (ILP). The memory organization of each SM consists of: (1) a large register file storing the context of all resident warps, allowing zero overhead warp switching and (2) a private cache shared among all residing warps. All SMs share an L2 cache and an off-chip memory. Additionally, Nvidia GPUs (Ampere generation and onwards) provide a unique programmer-controlled L2 access management for setting-aside

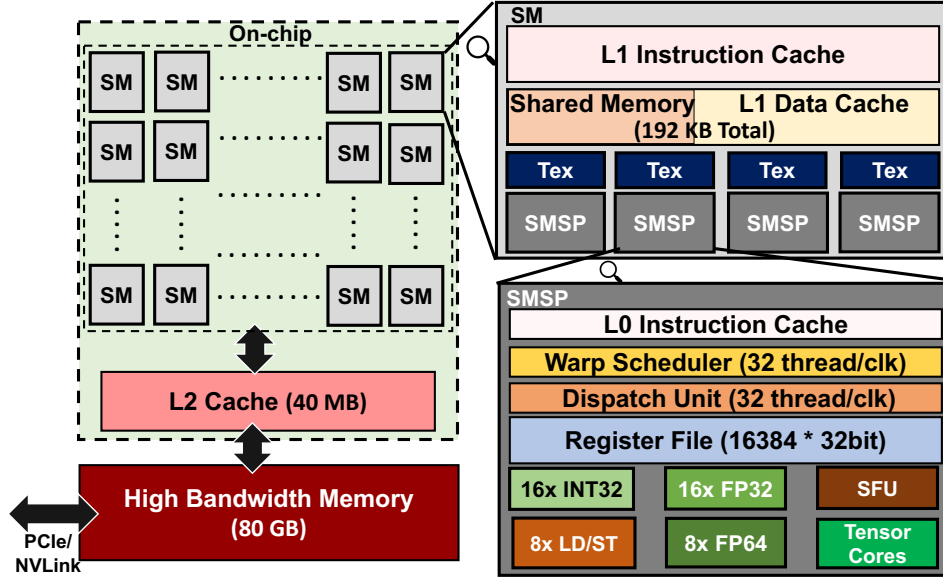


Figure 5.3: Simplified Nvidia A100 GPU organization.

a region for persisting accesses [172]. Section 5.4.3 guides on how we take advantage of this feature for a performant embedding stage execution.

The efficiency of DLRM execution is heavily influenced by the unique properties of GPUs. All the stages have parallel implementations to reap benefits of the massive number of CUDA cores. Also, the high bandwidth memory (HBM) helps in meeting the heavy off-chip access requirements for the embedding stage. Additionally, a modest on-chip cache hierarchy helps in capturing the locality in memory accesses. Table 5.1 shows the access latencies [4] for different access locations. Note that fetching data is more costlier compared to CPUs. Table 5.2 shows the L1 and L2 cache capacities in server-grade and powerful GPUs. Note that the cache sizes are much larger in the latest GPUs. For example, (1) A100 offers 1.5x and $\sim 7x$ large sizes over V100 for L1 and L2, respectively, and (2) RTX 4090 offers $\sim 12x$ large L2 cache over RTX 3090 Ti.

Table 5.1: Access Latencies for various levels of the A100 GPU memory hierarchy based on [4].

Access Location	Access Latency (cycles)
Register	1
Shared Memory	29
L1D cache	37.9
L2 cache	261.5
Global Memory	466.3

Table 5.2: Cache capacities for server-grade GPUs.

Device	LLC cache size (MB)	L1 cache size (KB)
A100	40	192
H100	50	256
L40	96	128
RTX4090	72	128

5.2.3 Related Works

DLRM Optimizations: Prior works [107, 115, 137, 167] have looked into scheduling frameworks and heterogeneous platforms for inference serving and [43] discusses the system design for effective distributed inference. [?, ?, 114] considers highly skewed (exhibiting high temporal reuse) dataset cases and proposes algorithmic and system designs for effectively using GPU’s main memory. However, none of the prior works have studied the microarchitectural implications of DLRM inference on a GPU. Our work improves the embedding table performance via on-chip optimizations for a diverse set of access patterns (Figure 5.5), making it orthogonal, and thus it can be combined the prior works.

Accelerator designs: [28, 106, 121, 122, 160–162] have proposed targeted custom solutions for MLP and embedding stages in DLRMs. However, these proposals require substantial time and effort to commercialize, making it difficult to adopt with the fast changing model parameters. With GPUs being widely adopted [168], our plug-and-play solutions can be instantly leveraged (Section 5.4).

Scheduling and Virtualization: Several warp and CTA scheduling works have been proposed in the past [173–175] to improve GPU performance by hiding memory latencies effectively. Most of these works focused on improving cache and memory contention or finding optimal thread-level parallelism. We show that even on the recent A100 GPUs, latency hiding capability is limited due to the limited register file (Section 5.3.3). Register-file virtualization techniques [176–178] have been proposed in the past to address the issues related to limited register file. However, they are implemented in GPU hardware and often have non-trivial overheads. In contrast, we provide a complementary software-only solution (prefetching and pinning) that is aware of both GPU application and underlying hardware (Section 5.4).

Prefetching on GPUs: Given that GPU memory bandwidth is limited, data prefetching needs to be done carefully to result into any performance benefits. Prior works on GPU prefetching [179–182] consider this issue and show performance improvements. However, to the best of our knowledge, there is no prior work that considers software prefetching in GPUs that is particularly tailored for emerging applications such as DLRM

(Section 5.4.2).

L2 Cache Management: Recently, with Nvidia’s Ampere architecture and onwards [30, 170], the GPUs feature a CUDA/PTX-based programmer control for L2 cache management [172]. [183–185] uses the L2 cache control for improving GEMM, LSTM, fully-connected and convolution-based kernels. In contrast, our paper proposes to apply this feature for embedding stage by pinning the most frequently accessed embeddings (Section 5.4.3).

5.3 Dissecting Embedding Bag execution on a GPU

For better understanding the inference behavior on GPUs, this section discusses: (1) the parallel implementation, work partitioning, and mapping of the embedding stage on CUDA threads; (2) a quantitative study of memory access patterns used in production deployments; and (3) the architectural implications of off-the-shelf and optimal-MT PyTorch-based DLRM inference on GPUs, catering to a variety of memory accesses. Finally, we conclude that memory latency continues to remain a challenge, and motivate towards optimizations addressing this issue to achieve better performance.

5.3.1 Parallel Implementation of the Embedding Bag Operator

The embedding stage of DLRM involves numerous parameters, and understanding how each one affects the performance is crucial. To illustrate this, Algorithm 8 highlights the high-level working of the embedding stage. Arriving queries create “batches”, where each batch is expected to meet the SLA target. For each table, the batch contains a batch size (BS) number of samples, and each sample involves a pooling factor (or lookups per sample) amount of work over the embedding vectors of length equal to the embedding dimension (ED). At the core, each sample does a gather (load) and reduce (accumulation) operation. The amount of data processed in each table can be calculated as $(BS) \times (\text{average lookups per sample}) \times (ED) \times (\text{precision})$. For example, for our chosen configuration (described in Section 5.5), the amount of data processed per table is $2048 \times 150 \times 128 \times 4B = 150 \text{ MB}$. Consequently, the complete embedding stage processes 37.5 GB of data. With the inference running on a GPU, the gather-reduce operations are executed in a Single-Instruction-Multiple-Threads (SIMT) manner, thus exploiting parallelism.

To better understand the incorporated parallelism, Figure 5.4 breaks down the em-

Algorithm 8: Simplified memory access loop for the embedding stage on GPU.

```

for v in 0 ... num_batches do
  for w in 0 ... num_tables do
    for x in 0 ... batch_size do
      for y in 0 ... lookups_per_sample do
        SIMT load accm on register;
        for z in 0 ... embedding_dim do
          SIMT load row_block on register;
          SIMT add accm, row_block;
        SIMT store accm to memory;
      end for
    end for
  end for
end for

```

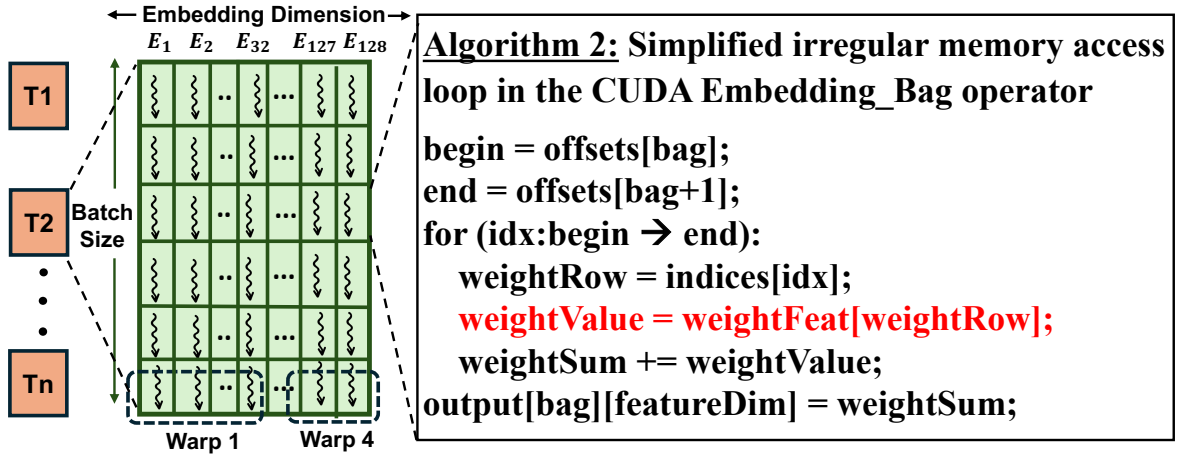


Figure 5.4: Parallel implementation of embedding stage by work partitioning across CUDA threads. Here, 1000s of CUDA threads independently work on one output matrix element.

embedding stage execution into three parts. First, for any number of embedding tables to be completed by one GPU, they are processed sequentially. Second, the embedding bag operator is used to process a table (using PyTorch’s backend CUDA kernel *"EmbeddingBag_updateOutputKernel_sum_mean"* [27]), which generates an output matrix of dimension (BS) $\times$ (ED). Intuitively, we can visualize that, within a batch, each sample and each embedding element is independent of the other. Thus, a CUDA thread works on each embedding element. In this off-the-shelf kernel, we note a static execution launch configuration with a grid size of (1024,1,1) and a block size of (32,8,1). This results in a large number of CUDA threads, and thus, fully uses all the SMs provided in the latest GPUs (e.g., the 108 SMs in A100). Warps are automatically formed in the GPU by combining adjacent CUDA threads. For example, with an embedding dimension of 128, 4 warps are formed to process a sample. Third, Algorithm 2 (Figure 5.4) highlights the

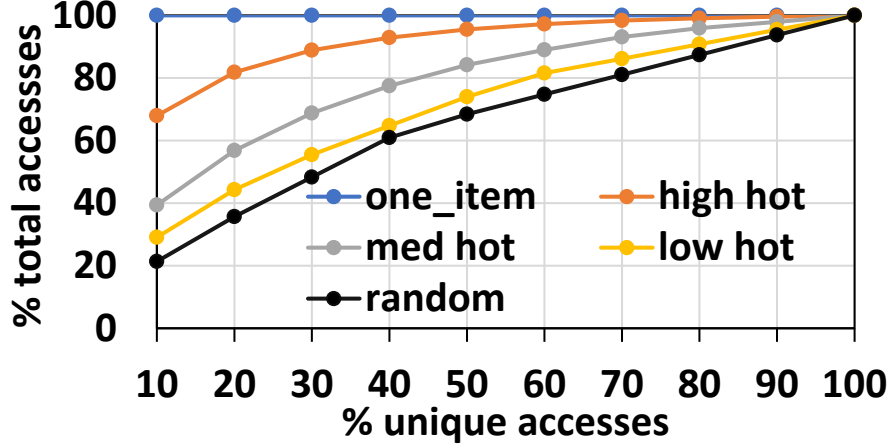


Figure 5.5: Coverage study for different memory access patterns: it shows the % of total accesses (y axis) that are covered by the % of unique accesses (x axis).

work within a thread which encapsulates a “number of lookups” amount of gather-reduce operations (thus each thread partially completes the two innermost loops in Algorithm 8). Fundamentally, the gather is similar to a pointer-chasing operation as we access a series of arrays to complete it (the offset array, followed by the indices array, followed by the embedding table). Thus, this operation results in the irregular loads which have a significant impact on (and causes variation in) the performance of the embedding stage (Figure 5.1).

5.3.2 Quantitative Study of the Memory Access Patterns

Embedding accesses in DLRMs follow a “power-law” distribution where a small portion of embedding table entries services a large fraction of accesses [100, 107, 136]. In our study, we use the datasets from a recent work [136], which extracts homogeneous datasets using Meta’s production traces [33].

Building on prior works [100, 107, 116, 136], we investigate various memory access patterns encountered in real-world industrial settings and categorize them based on their degree of “hotness”. To understand hotness, we define two metrics that classify memory access patterns within datasets: unique access % and coverage study. For a given table, unique access % represents the proportion of distinct accesses compared to the total number of accesses. Essentially, it measures the variety across memory locations accessed within the table. Thus, considering a total of R accesses (number of rows in a table) and U unique accesses, the unique access % is calculated as $U \times 100 / R$. Table 5.3 shows each dataset’s unique access %. Note that one item and random are “synthetic” datasets; the former corresponding to the case where all indices match and point to the same entry in

a table, whereas the latter means all indices are uniformly distributed within a range of $[0, R)$. Thus, unique accesses range between 0 to 100%, being lowest for one item and highest for random.

Table 5.3: Unique access % in each dataset.

Datasets	one item	high hot	med hot	low hot	random
unique access%	0.0002	4.05	20.50	46.21	63.21

Further, the distribution of the unique accesses influences the actual memory access pattern. Figure 5.5 shows the coverage study by noting how much total accesses get covered by a given number of % unique access. For example, in the one item case, one embedding covers all 100% accesses (making the trend across x-axis uniform), whereas, in the high hot case, 10% of the total unique items are sufficient to capture 68% of the total accesses. Regardless of the hotness, it is important to note that the total memory access count remains the same in each of these datasets. Thus, it is fair to compare the performance of different datasets while ensuring the same amount of observed loads. Using these two metrics, it can be noted that, for a given table, the hotness decreases from 'one item' to 'random', causing an increase in the working set size and total number of irregular loads.

5.3.3 Architectural Implications of Embedding Bag on a GPU

Previous subsections have highlighted the amount of parallelism offered in the embedding bag operator and how it leverages the GPU for execution. Modern GPUs [186,187] provide larger caches and HBM capacity, which directly help embeddings data reuse behavior and bandwidth needs. In this spirit, various previous works [107,115,116,137,158,188] have used GPUs for DLRM inference and training. However, to our knowledge, no prior work has conducted a detailed profiling to study the 'microarchitecture behavior' of DLRM execution on GPUs. Specifically, given that DLRMs are generally memory bound [100,107,136], it is important to thoroughly verify whether the primary application kernel is effectively utilizing the GPU's resources. With this motivation, we carefully investigate the embedding bag kernel Figure 5.4 [27] using Nvidia's Nsight Compute Tool (NCU) on an A100 80 GB GPU. Since various memory access patterns affect the memory-bound behavior, we evaluate multiple datasets (Table 5.3).

Table 5.4 describes the off-the-shelf PyTorch characterization using various NCU metrics. Recall that Figure 5.1 highlighted that random performs $3.2\times$ slower than the fastest one item case, even though both observe the same number of loads. This is

Table 5.4: Microarchitectural characterization of Base PyTorch on various datasets. With 74 registers allocated to each CUDA thread, the WLP is limited due to the register pressure.

NCU metrics/datasets	one item	high hot	med hot	low hot	random
Kernel time (us)	138	237	341	428	442
#load insts (M)	2.47	2.47	2.47	2.47	2.47
SM Throughput %	71.45	41.27	26.65	21.23	20.42
warp cycles per executed inst	7.06	11.7	17.56	21.94	22.86
long scoreboard stall (cycles)	1	7.2	13.1	17.7	18.6
issued warp per scheduler per cycle	0.77	0.47	0.31	0.25	0.24
Global L1\$ hit rate %	98.7	42.74	30.11	20.36	19
L2\$ hit rate %	99.46	93.96	59.5	18.71	7.7
Device Memory size read(MB)	0	4.87	45.96	122	144.57
Avg HBM Read BW (GBps)	~0	20.8	135	286.5	329.5
Avg HBM Read BW Utilization (%)	~0	1.04	6.75	14.33	16.5

because the SM or compute throughput is heavily impacted by random accesses. With the decrease in hotness (one item to random), the data reuse gets reduced [136], causing an increase in the warp cycles per executed instruction. As each CUDA thread performs a pooling factor amount of gather-reduce operations, a load-use dependency arises. We look into the breakdown of warp cycles and inspect the long scoreboard stall cycles to exactly capture these dependency stalls. For all datasets except one item, the warp cycles are mainly constituted from the long scoreboard stalls. The absolute stall cycles are impacted by the amount of data captured by the caches. Note that both warp cycles per executed instruction and long scoreboard stalls are *averaged* over all executed instructions, and so they cannot be directly compared to the memory latency values in Table 5.1. As one item dataset has a minimal working set (512B), it experiences much lower stalls due to best cache locality. However, both L1 and L2 cache hit rates significantly drop as the hotness lowers, increasing the amount of data read from the device memory. Therefore, the average bandwidth demand is significantly higher towards the random dataset, reaching up to 329.5GBps. Further, the peak read bandwidth (measured using Nvidia Nsight Systems [189]) achieved is 510GBps for the random case. However, this observed bandwidth is small compared to the theoretical peak bandwidth of HBM (2TBps). This significant disparity makes us suggest that the Embedding Bag operator is a memory latency-bound kernel.

The Nvidia A100 GPU is based on compute capability 8 [186], meaning that one SM houses up to a maximum of 64 resident warps. These resident warps enable WLP

primarily helping in hiding any kinds of stalls. We observe that the PyTorch CUDA kernel [27] uses a high number of registers (74), and thus suffers from the register pressure, leading to a low theoretical occupancy of 37.5% (or 24 resident warps per SM). Figure 5.3 indicated that one SM contains a total of 4 warp schedulers, meaning that each scheduler gets to work with only 6 warps, even though the hardware supports a maximum of 16 warps. The metric “issued warp per scheduler per cycle” (also called as “issue slot utilization”) captures the number of issued warps every cycle, which is a function of both WLP and warp cycles per issued instruction. Similar to the SM throughput, it decreases as the hotness lowers. Thus, even though the CUDA kernel encounters significant long scoreboard stalls, the application lacks in providing effective WLP, limiting the hardware’s capability to hide these stalls.

Since higher WLP could potentially better mitigate the memory latency, we force the compiler to strategically *limit* the allocated registers during compilation, resulting in more warps to be resident in one SM, eventually improving the WLP. To achieve this, we compile PyTorch with “-maxrregcount maxreg” [190] flag, where the “maxreg” amount of allocated registers is enforced by the compiler. However, with lower registers in use, now register spilling occurs. The compiler spills the registers to local memory which in turn penalizes the performance. By varying the number of registers, we can sweep through different WLP configurations as seen in Figure 5.6. Here, we capture the performance improvement over different datasets. Higher WLP helps in gaining performance with maximum gain at 40 resident warps (denoted as OptMT). Also, higher improvements are seen for low hot and random cases as they require more latency hiding. Further, even though 48 and 64 resident warps provide better WLP, the performance drops due to an increase in register spilling. The impact of register spilling is measured in terms of local memory loads. In the baseline PyTorch (24 warps per SM), all loads/store accesses go to the global memory and none to the local memory, meaning that all the embedding accesses are served from the global memory. However, with an increase in WLP, the register spilling increases, causing an increase in the local memory loads which hurts performance. Thus, there is a clear tradeoff between WLP and the spilling penalty. For instance, in the high hot case, 64 resident warps per SM underperform (compared to the baseline) as the spilling penalty overshadows the potential benefits from multi-threading.

Table 5.5 describes the microarchitectural characterization for OptMT. For one item, the performance matches the off-the-shelf PyTorch. For the remaining datasets, the performance significantly improves with the rise in SM throughput. The metric “warp cycles per executed inst” is slightly higher compared to the baseline for two reasons: (1)

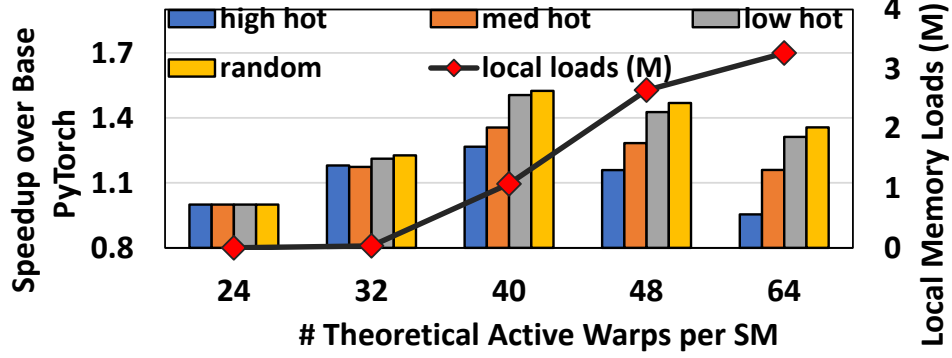


Figure 5.6: Synthetically varying the number of registers allocated to improve WLP. The primary y-axis is speedup over off-the-shelf PyTorch, and the secondary y-axis is the register spilling penalty based on extra local memory loads (in millions). OptMT refers to the highest speedup at 40 warps.

Table 5.5: Microarchitectural characterization of Optimal-Multithreading (OptMT) PyTorch on various datasets. With 42 registers allocated to each CUDA thread, the register pressure lowers and the WLP significantly improves. Still, a performance gap exists between the fastest and slowest loads.

NCU metrics/datasets	one item	high hot	med hot	low hot	random
Kernel time (us)	135	189	250	282	290
#load insts (M)	3.54	3.54	3.54	3.54	3.54
SM Throughput %	71.89	54.93	39.3	34.72	33.84
warp cycles per executed inst	10.61	15.2	20.93	24.74	25.44
long scoreboard stall (cycles)	1.33	8.6	15.3	19.6	20.4
issued warp per scheduler per cycle	0.79	0.59	0.42	0.36	0.35
Global L1\$ hit rate%	98.7	37	27.2	19.85	19
L2\$ hit rate %	85.36	92.3	56.51	16.48	7.1
Device Memory size read(MB)	0.3	7.5	54.1	131.9	151
Avg HBM Read BW(GBps)	2.57	43	226.5	485.4	547.5
Avg HBM Read BW Utilization (%)	~0	2.2	11.3	24.3	27.4

with the increase in more resident warps, there exist times when multiple warps are ready and not selected, thus causing an increase in the “Stall Not Selected” stalls; (2) with the increase in warp switching and local memory accesses, more cache thrashing occurs for global accesses as visible with a slight decrease in cache hit rates, leading to a slight increase in the “long scoreboard stalls”. Based on the latter reason, the total reads from device memory also slightly increases. Finally, the HBM read bandwidth increases to meet the higher demand from WLP. While our approach using limited register allocation demonstrates performance gains compared to the baseline, the “issue slot utilization” for

the lower hotness cases remains significantly less than the one-item case. This observation suggests that *even with the enhanced warp-level parallelism (WLP), memory latency continues to be a bottleneck.*

5.3.4 GPU-specific Key Microarchitectural Insights

The following key insights emerge from the preceding microarchitectural characterization study on an A100 GPU:

- Due to the variations in memory access patterns, a significant performance gap could exist across the spectrum of memory access patterns. In the baseline (or off-the-shelf) PyTorch, this gap is visible and arises from the long scoreboard stalls due to high cache misses.
- Although GPUs are equipped with high multi-threading support, the off-the-shelf CUDA kernel suffers from register pressure and fails to provide enough WLP.
- The WLP can be increased by forcing the compiler to lower the allocated registers. However, it comes at the cost of a performance penalty from register spilling. Peak performance gain is seen with 40 resident warps (marked as OptMT).
- OptMT lowers batch latency by up to 53% over the baseline with visible benefits in the SM throughput. Yet, the “issue slot utilization” continues to see a high gap between the fastest and slowest loads.
- The average read memory bandwidth increases with OptMT, yet it remains small compared to the peak of HBM. Thus, we can conclude that *the embedding bag operator is memory latency bound on the GPU.*

Thus, the best WLP contained within an application is insufficient in fully hiding the long latency loads. In the following section, we discuss two complementary techniques to minimize the memory latency further.

5.4 Optimizations to Improve the Warp Scheduler Issue Slot Utilization

For bridging the gap in the issue slot utilization across the ends of the memory access spectrum, in this section, we first describe the limitations of existing hardware and

software approaches. We then propose an application-driven software prefetching strategy and discuss how it can take advantage of various memory resources as buffering stations. Following that, we propose an L2 pinning strategy to overcome the potential shortcomings of prefetching. Finally, we describe how prefetching and pinning can work synergistically.

5.4.1 Limitations of Off-the-Shelf Solutions

To deal with memory bound kernels, GPUs employ: (1) WLP and zero-overhead context switching to keep executing useful work even when a warp stalls; (2) large cache line sizes (128 bytes) which help in exploiting spatial locality; (3) a scoreboarding mechanism which helps to issue and execute consecutive instructions without stalling the pipeline until a dependent instruction is reached; and (4) scratch-pad memory and L2 cache with explicit programmer control for intelligent data placement and reuse. While the off-the-shelf CUDA kernel takes advantage of (1), (2) and (3), it still lacks in performance, as previously highlighted in Section 5.3.4.

To take advantage of (3) and (4), an optimizing compiler or application-specific software development can help. An optimizing compiler would try to come up with a valid reordering of instructions to improve performance. For instance, compiler could hoist independent instructions between a load and use dependency and thus promote scoreboarding to minimize pipeline stalls. In this direction, *loop unrolling* could be useful as it broadens the scope of finding independent instructions using later iterations. However, when testing the optimal compiler (in O3 level), we note that inserting “`#pragma unroll`” does not have any positive impact on performance because of the runtime-dependent loop bounds (Figure 5.4). Furthermore, the compiler cannot directly manage the scratch pad or L2. We believe these challenges open the door for application-specific software optimizations which can take advantage of specific hardware features of GPUs.

5.4.2 Application-Driven Data Prefetching

Data prefetching is a classic technique to improve performance for memory latency-bound kernels and has been well-adopted in both software [191] and hardware [85,192]. However, on-chip prefetching in GPUs is *uncommon*, due to GPUs extensive reliance on WLP for latency hiding [166]. Also, unlike CPUs, GPUs do not employ dedicated on-chip hardware-based prefetching engines. Thus, the CUDA programmer can develop tailor-made prefetching solutions for their application while minimizing its overhead challenges.

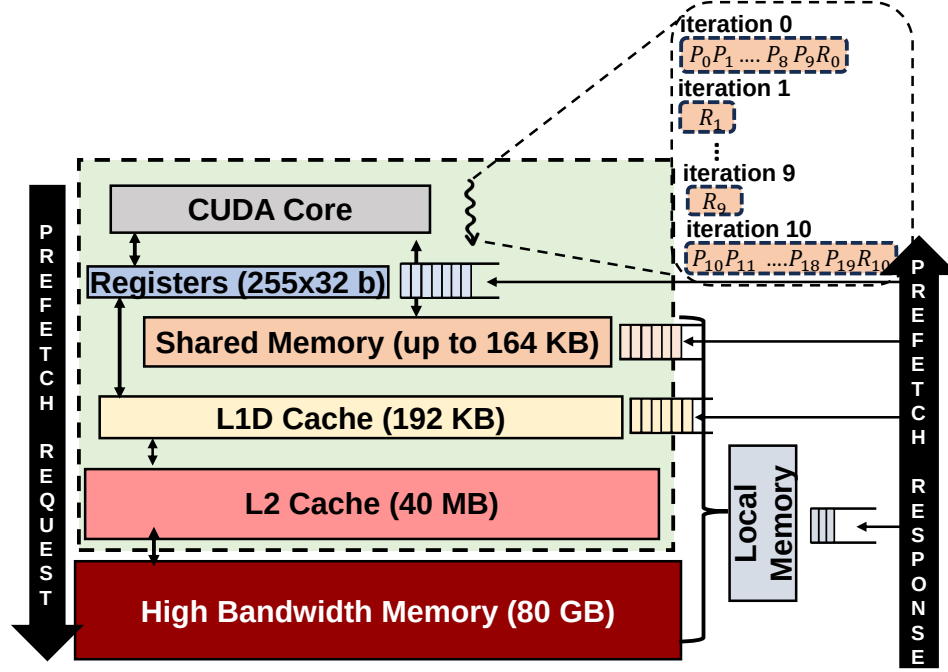


Figure 5.7: Buffer locations used for various prefetching schemes. A CUDA thread is shown executing SMPF, a batch of 10 prefetches(P) are launched every 10th iteration, and a reduce(R) operation every iteration.

To comprehensively explore the design space of prefetching, we formulate a series of key questions as discussed next:

What to Prefetch? Recall that Section 5.3.1 highlighted the presence of the gather-reduce operations (forming a load-use dependency chain) as part of pooling operations performed by each CUDA thread. Here, the gather operation is an indirect load (pointer-based load) and spans a variety of memory access patterns (Table 5.3), which leads to long scoreboard stalls (Tables 5.4 and 5.5). Thus, this gather operation is our prefetch target. During the kernel launch of the embedding bag operator, each CUDA thread receives an offset and indices array. These two arrays essentially provide each thread with the complete set of addresses it will need to load data from. Consequently, by leveraging this knowledge of future access patterns before the actual loads are required, we can insert 100% accurate prefetches.

Where to Prefetch? Ideally, for the goal of latency hiding, we would like to prefetch the data as close to the CUDA core pipeline as possible. However, the hardware could be limited in resources, and thus it is wise to consider a variety of locations for storing the prefetches (buffer stations). Figure 5.7 shows a total of 4 buffer stations – register, shared memory, L1 D\$, and local memory. We do not pick L2 since its access latency is quite high (261.5 cycles [4]). Note that each buffer station has pros and cons. Specifically, in

```

INITIALIZE: bfr0, bfr1, pf_cnt = 0,
begin = offsets[bag], end = offsets[bag+1];
START:
for (idx:begin → end):
    int trigger_prefetch = pf_cnt%2;
    if (trigger_prefetch == 0){
        weightRow = indices[idx];
        bfr0 = weightFeat[weightRow];
        weightRow = indices[idx+1];
        bfr1 = weightFeat[weightRow];
    }
    pf_cnt = pf_cnt + 1;
    switch(trigger_prefetch){
        case 0: weightValue = bfr0;
        case 1: weightValue = bfr1; }
    weightSum += weightValue;
    output[bag][featureDim] = weightSum;

```

```

INITIALIZE: prefetch_bfr[256][10],
begin = offsets[bag], end = offsets[bag+1],
pf_cnr = 0;
START:
for (idx:begin → end):
    int trigger_prefetch = pf_cnr%10;
    if(trigger_prefetch==0){
        #pragma unroll
        for (int zz = 0; zz < 10; zz++){
            weightRow = input[emb + zz];
            prefetch_bfr[thread_id][zz] =
                weightFeat[weightRow];
        }
    }
    pf_cnr = pf_cnr + 1;
    weightValue = prefetch_bfr[thread_id]
        [trigger_prefetch];
    weightSum += weightValue;
    output[bag][featureDim] = weightSum;

```

a) Register-based Prefetching

b) Shared Memory-based Prefetching

Figure 5.8: Prefetching implementations: (a) RPF (b) SMPF

terms of access latency, the register is optimal, whereas, in terms of size, L1 D\$ and local memory are optimal locations. Note also that local memory is the scope of a variable, and the data can reside in L1/L2/HBM [193]. We design and implement prefetching for all 4 buffer locations, and use the following abbreviations: RPF for Register-based Prefetching, SMPF for Shared memory-based Prefetching, LMPF for Local memory-based Prefetching, and L1DPF for L1D\$ Prefetching.

How and When to Prefetch? Fundamentally, we want to issue a prefetch much ahead of its demand to hide the worst-case latency (timeliness property). Also, as mentioned at the beginning of the section, GPU employs a scoreboard mechanism for ILP. In the CUDA program, one can take advantage of it by manually reordering the instruction stream to pack a batch of needed load instructions. Towards this, for RPF, SMPF, & LMPF, (1) we mimic prefetching with the ahead-of-time issue of the demand loads and storing the data into a buffer station, thus the prefetch becomes the producer of the data, and (2) when this data is consumed by the reduce operation, it fetches it from same buffer station. Based on this understanding, Figure 5.8 shows a simplified implementation of the RPF and SMPF schemes on top of Algorithm 2. LMPF can be similarly implemented.

For L1DPF, we use a PTX-based intrinsic “`prefetch.global.L1`” [194] to prefetch the data into L1D\$, similar to commonly used CPU intrinsics [191]. To achieve timeliness (and ultimately optimal performance), we vary the prefetch distance to find the optimal value. Figure 5.9 evaluates the prefetch distance for the SMPF scheme over various datasets and finds the optimal prefetch distance as 10.

Countering the Potential Overheads? As discussed above, by systematically navigating the search space, prefetching can find the optimal design points and improve performance. However, various overheads could arise: (1) With the addition of software prefetch support, the total instructions executed could increase, leading to increased computation. For example, a 37.2% overhead is observed in SMPF. (2) Many prefetch distance choices could degrade the performance. For example, in Figure 5.9, a distance of 1 hurts the performance for all datasets. Also, similar to [175], a large distance could create LSU stalls. Furthermore, injected prefetches could hurt the locality of other parts of the code. (3) With the optimal prefetch distance of 10 in Figure 5.9, for random case, while the long scoreboard stalls significantly reduce from 18.6 cycles (Table 5.4) to 4.6 cycles, the remaining stalls suggest suboptimal timeliness. (4) For a given buffer station, we are limited by either the size or latency it offers. For example, while the register access is fastest (Table 5.1), their limited count can become a bottleneck, as noted with limited WLP and register spilling (Section 5.3.3). (5) Though not in our case, any scope of inaccurate prefetching or unavailability in off-chip bandwidth headroom could hurt performance by causing cache pollution [182] or throttling of demand memory requests [195].

Therefore, through a combination of profiling-based study and empirical tuning, we establish a high-performance prefetching strategy that delivers 100% coverage and accuracy. Yet, various overheads (like extra instructions and increase in bandwidth demand) and sub-optimal timeliness could hold. To resolve these, in the next subsection, we propose L2 pinning which can complement prefetching.

5.4.3 Application-Aware L2 pinning

Traditionally, GPUs have prioritized compute cores over cache capacities while relying on WLP for *tolerating* long latency stalls. However, with the rise of new applications (e.g., those based on deep learning) and larger chip areas, modern GPUs are witnessing an increase in cache capacities (Table 5.2). Further, Nvidia GPUs (from Ampere architecture onwards) have recently released support for CUDA programmer-based L2 cache access management (residency control [172], which enables a portion of the L2 cache to be

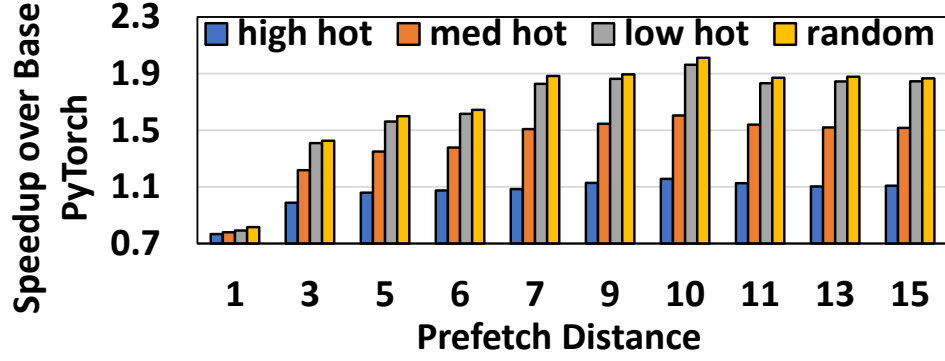


Figure 5.9: Performance impact of prefetch distance in SMPF.

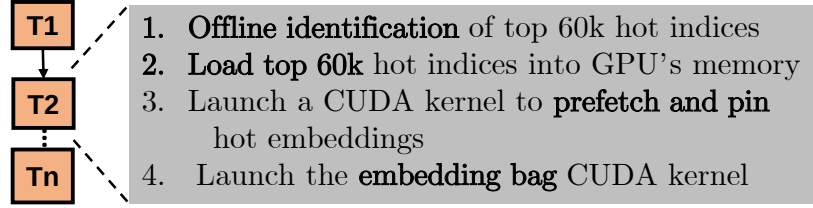


Figure 5.10: High-level design of L2P

used for persistent data access). Given that off-chip memory accesses are very costly (Table 5.1), a programmer with the knowledge of the underlying memory access pattern behavior can *mark* the persistence of high-reuse regions that otherwise may suffer from thrashing. Given that the memory accesses in the embedding stage follow a Power Law distribution (Figure 5.5), we propose an L2 pinning (L2P) design that can benefit from L2 residency control for the embedding accesses. Towards this, we first discuss the design and implementation, and then discuss the performance expectations and associated overheads.

What, When, and How to Pin? In an A100 GPU, a maximum of 30MB (75% of L2) can be set-aside for residency control, while the remaining (at least 10MB) is completely hardware managed. To promote the highest locality, we propose using the complete 30MB for storing the most frequent embedding vectors. Since each embedding vector is of size 512KB, thus a maximum of 60K embedding vectors can be pinned in L2. Figure 5.10 shows the high-level design for L2P. We conduct an offline profiling to identify the top 60K hot indices present for each dataset which are used as candidate embedding entries for pinning. In the beginning of the inference server, these indices can be loaded into the GPU's main memory. The embedding tables are processed sequentially, and each table follows two steps: (1) launch a CUDA kernel that prefetches and pins embedding vectors corresponding to the hot indices, and (2) launch the default embedding bag

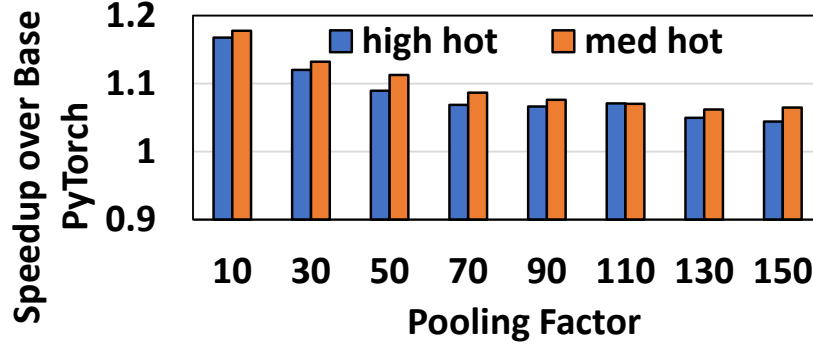


Figure 5.11: Detailed study of L2 pinning over various pooling factors. Speedup is reported over off-the-shelf PyTorch.

CUDA kernel. We use the inline PTX instruction "prefetch.global.L2::evict_last" [194], which takes an address as input, loads the associated cache line into L2, and marks the eviction policy as "evict_last". Setting this eviction policy allows the marked/chosen data to persist in L2 over others. For instance, during an eviction event in L2 for a set, the marked cache lines are less likely to be kicked out. Thus, the embedding stage is expected to observe lower access latency for a majority of the memory accesses (261 cycles over 466 cycles from Table 5.1).

Performance Expectations and Overheads? Although hardware caches capture hotness to an extent (Table 5.4), we expect L2 pinning to further enhance locality by avoiding: (1) L2 cold misses for hot indices, and (2) thrashing of highly reused embeddings. Moreover, the effectiveness of hardware caches is impacted by the batch and pooling sizes (based on the model and scheduling policy) where smaller sizes lead to lower reuse situations. As expected, L2P improves over the baseline, yielding more benefits in lower pooling cases, and performs slightly better for the med hot case (Figure 5.11).

Clearly, embedding access patterns can change over time, potentially reducing the effectiveness of L2 pinning. To address this challenge, similar to prior research [49], we can update the pinned data periodically. This ensures that the L2 cache always stores the most frequently accessed elements, maximizing the benefit of pinning. The overhead of storing the top 60K indices for every table on a GPU is minimal. For example, for 250 tables, it would be $250 \times 60K \times 8B = \sim 120MB$. Finally, the overhead of the L2P kernel is small and can be hidden by overlapping it with the CPU pre-processing required before the embedding bag kernel launch.

5.4.4 Synergy between Prefetching and Pinning

In this subsection, we discuss how prefetching and pinning can *symbiotically* work with each other, and thus further improve upon embedding bag’s memory-latency bound regime. While prefetching hides the long load latency by bringing the demand loads ahead of time near the CUDA core pipeline (registers, shared memory, or L1D\$), it still suffers from suboptimal timeliness and puts pressure on the off-chip bandwidth. Similarly, even though pinning lowers the load latency by bringing and holding the frequently-accessed embeddings in the L2 cache, it still lacks because (1) 30MB of L2 set-aside cannot fully cover the working set required by the datasets, especially in the low hot and random cases, and (2) access latency with L2 is significantly high (261.5 cycles; see Table 5.1), compared to registers, shared memory, or L1D\$. When combined, prefetching strengthens pinning by providing 100% coverage and faster access to the CUDA core pipeline, and pinning bolsters prefetching by improving the timeliness while cutting down on HBM requests.

5.5 Methodology

Hardware: Table 5.6 captures the hardware properties of our real-system evaluation setup.

Table 5.6: System specifications used for evaluation

CPU	AMD EPYC 7763
RAM	1 TB
GPU	Nvidia A100-SXM4-80GB
# SMs	108
Register File per SM	64K $\times$ 32 bit
L1D Cache size	192KB
Shared Memory size	up to 164KB
L2 Cache size	40MB
Device Memory	80GB, HBM2e
HBM Bandwidth	1.94TB/s

Software: PyTorch (v2.1.0) [196] is source-compiled on a Linux/Ubuntu machine with CUDA Driver Version: 535.129.03, and nvcc version 12.2. Here, source-compiled PyTorch matches off-the-shelf packaged PyTorch in performance. “nvcc” compilation is highly optimized with O3 flag [197].

Model: Taking inspiration from [50, 100, 107, 136], we pick model configurations which are representative of industrial inference settings. Following are used: (1) Bottom MLP dimensions are "1024-512-128-128" (2) Embedding stage has 250 tables, each table

having 500,000 rows and 128 embedding dimensions (3) Top MLP dimensions are "128-64-1". 4-byte precision is used which makes each embedding vector of size 512KB. The total model weight is of size ~ 60 GB which can completely fit within one GPU's memory while the remaining memory is used for computations across intermediate layers. Unless mentioned otherwise, all tables are homogeneous in hotness. Table 5.7 describes a mixture of tables for heterogeneous evaluation.

Table 5.7: Heterogeneous Mixture of Model configuration

Mixture/Datasets	high hot	med hot	low hot	random
Mix1 (#tables)	100	75	50	25
Mix2 (#tables)	62	63	63	62
Mix3 (#tables)	25	50	75	100

Datasets: Following an earlier work [136], we use the publicly released homogenized production traces from Meta [33, 198]. Thus, we consider a variety of hotness (memory access patterns): one item, high hot, medium hot, low hot, and random. Section 5.3.2 quantitatively compares these datasets. To represent a large access pool similar to a real inference, we calculate the unique access % averaged over 100 measurements (Table 5.3). Inspired by [33, 107, 116, 136], a large batch size of 2048 and a large "lookups per sample" (or pooling factor) of 150 is used, .

Nomenclature for combined schemes: Any combined scheme is denoted using a '+' symbol. For example, RPF+L2P+OptMT is a combination of 3 schemes, namely, RPF, L2P, and OptMT.

5.6 Evaluation

Based on real system measurements, we evaluate the benefits of our proposed latency hiding schemes: OptMT (Section 5.3.3), Software Prefetching [RPF, SMPF, LMPF, L1DPF], L2 Pinning (L2P), as well as their combined versions (Section 5.4). We first study the key improvements in embedding stage and end-to-end DLRM with micro-architectural justifications, followed by sensitivity studies. In all our evaluations below, off-the-shelf PyTorch [27, 146] (with the property of 24 theoretical active warps per SM) is used as the "baseline", and all the stages of DLRM inference run on a GPU. Performance is measured as "batch latency", and all improvements are reported over off-the-shelf/base PyTorch.

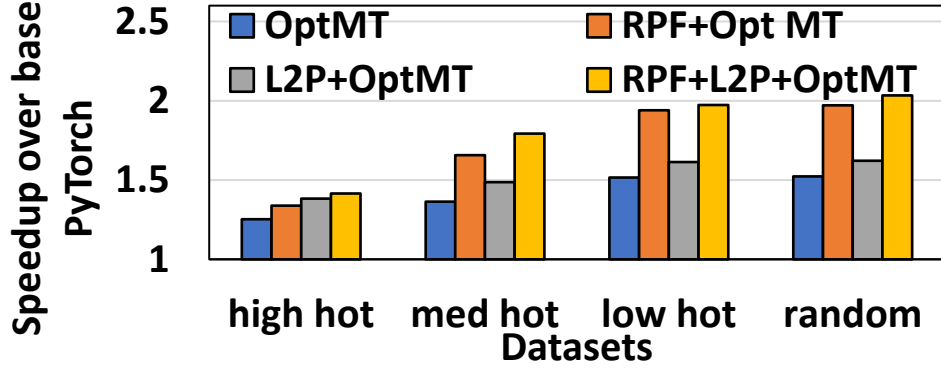


Figure 5.12: Embedding-only improvement in latency of the proposed techniques with OptMT over off-the-shelf PyTorch.

5.6.1 Key Results

5.6.1.1 Boost in Embedding Stage Performance

Given our proposed schemes target the primary bottleneck (Embedding Stage) in DLRM, we first highlight the embedding-only benefits. Figure 5.12 evaluates the performance of various design points across different datasets. As highlighted earlier in Section 5.3.3, OptMT exploits the GPU’s WLP better, thus, better hiding the long latency loads. RPF and L2P complement OptMT by further lowering the latency (note that we have picked RPF as the winning prefetching scheme due to it performing slightly better over other schemes; see Section 5.6.2.1). As L2P is more useful for smaller working set situations, more improvement is seen with the high/med hot cases, and as RPF is more suitable for long latency load situations, more improvement is seen with low hot/random cases. Finally, we note that RPF and L2P combined further improves the performance, achieving up to a $2.03\times$ speedup (random case). Interestingly, the highest benefit is 13.5% in the med hot case over the previous optimal (RPF+OptMT), thus maximally complementing each other.

5.6.1.2 Boost in End-to-End DLRM Performance

Given that DLRM is composed of 4 stages (Section 5.2.1) which collectively influence the batch latency, we also evaluate the benefit of our proposed schemes with respect to the end-to-end latency (Figure 5.13). Note that the trends in speedup remain similar to Figure 5.12, with a minor degradation in the final speedups (since embedding is the bottleneck). It can also be noted that the combined scheme (RPF+L2P+OptMT) achieves a significant speedup of up to 77% (random case). Recall Figure 5.1 highlighted

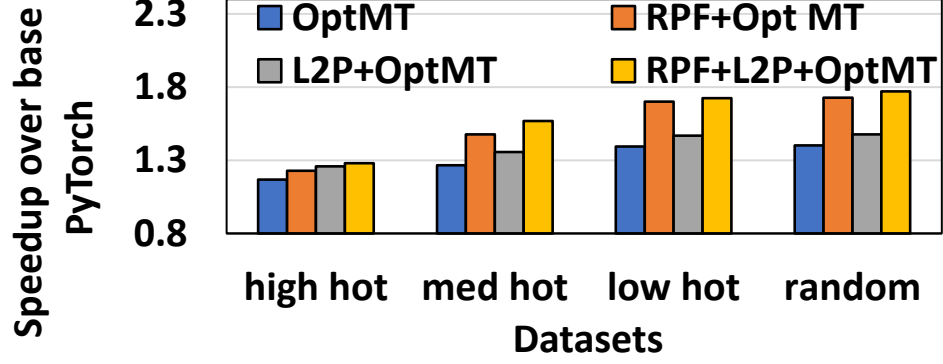


Figure 5.13: End-to-end improvement in latency of the proposed techniques with OptMT over off-the-shelf PyTorch.

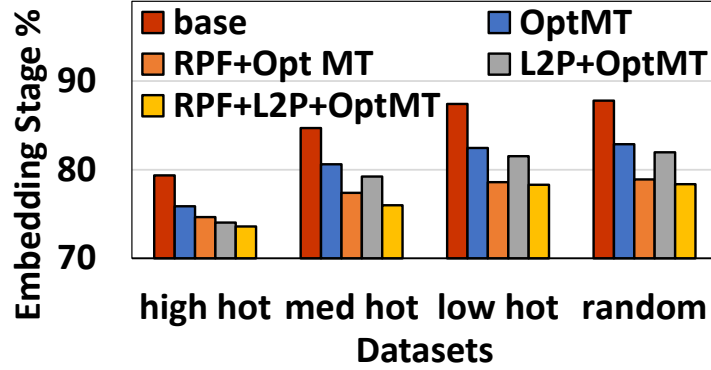


Figure 5.14: Embedding stage contribution in the end-to-end latency.

the performance gap between the fastest (one item dataset) and slowest (random dataset) loads being $3.2\times$ and $2.1\times$ for off-the-shelf and OptMT Pytorch, respectively. With the synergistic integration of our proposed schemes in play, we are able to substantially lower the performance gap (highlighted in Figure 5.14) to only $1.57\times$, thus decreasing it by 163% and 53% over off-the-shelf and OptMT PyTorch, respectively. Therefore, with the Embedding Stage running optimally, its contribution in the end-to-end execution reduces by up to 10% (random).

5.6.1.3 Microarchitectural Justifications

To better understand the above gains, we profile the proposed schemes using NCU [199]. Table 5.8 and Table 5.9 show the microarchitectural measurements for RPF+OptMT and RPF+L2P+OptMT designs, respectively. Due to limitations in NCU profiling [200], we could not measure all the metrics for the integrated scheme. For the random case, RPF+L2P+OptMT achieves an issue slot utilization of 44%, thereby improving by 83% over baseline, and 26% over OptMT. This is because RPF+OptMT better utilizes the memory bandwidth, reaching up to 700 GBps, significantly higher than the baseline

(329.5 GBps). With L2P combined, for the high and med hot cases, it lowers the total amount of data read from the device memory by 71% and 16.2%, thus lowering memory access latencies and saving memory bandwidth.

Table 5.8: Microarchitectural details for RPF+OptMT

NCU metrics/datasets	high hot	med hot	low hot	rand
Kernel time (us)	177	205	220	224
#load insts (M)	4.43	4.43	4.43	4.43
SM Throughput %	59.3	49.7	44.4	43.3
issued slot utilization (%)	59.17	49.63	44.32	43.5
Device Memory size read(MB)	8.4	53	133	151.8
Avg HBM Read BW(GBps)	51.4	277.7	629.1	699.4
Avg HBM Read BW Util- ization (%)	2.6	13.9	31.5	35

Table 5.9: Microarchitectural details for RPF+L2P+OptMT

NCU metrics/datasets	high hot	med hot	low hot	rand
Kernel time (us)	167	190	216	217
#load insts (M)	4.43	4.43	4.43	4.43
SM Throughput %	60	49.9	44.5	43.3
issued slot utilization (%)	60.12	50.21	44.64	43.61
Device Memory size read(MB)	4.9	45.6	128	150
Avg HBM Read BW(GBps)	30	240.6	613.2	698
Avg HBM Read BW Util- ization (%)	1.5	12.3	30.7	34.9

5.6.2 Sensitivity Analysis

5.6.2.1 Winning Prefetching Scheme

Earlier, Section 5.4 proposed 4 different data prefetching schemes based on the buffering location of the prefetches – register, shared memory, local memory, or L1 data cache. Because of this, the implementation of each scheme differs. For each prefetching scheme (on top of OptMT), we empirically find the optimal “prefetch distance” by doing a sweep similar to that in Figure 5.9. Interestingly, we find that all schemes perform best at a prefetch distance of 2. Figure 5.15 compares all prefetching schemes in conjunction with OptMT over the baseline PyTorch. It can be noted that all schemes improve on top of the baseline, where L1DPF improves the least and RPF improves the most. However, to implement prefetching, we modify the CUDA kernel which results in extra instructions, thus adding overhead by increasing the total raw instructions to be processed. Among all

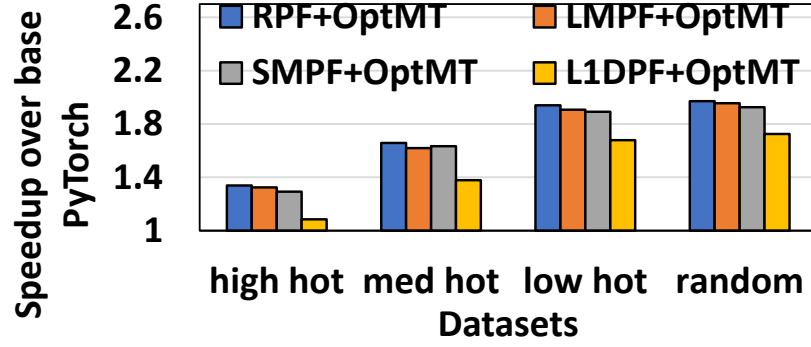


Figure 5.15: Comparison of all prefetching techniques with OptMT over off-the-shelf PyTorch.

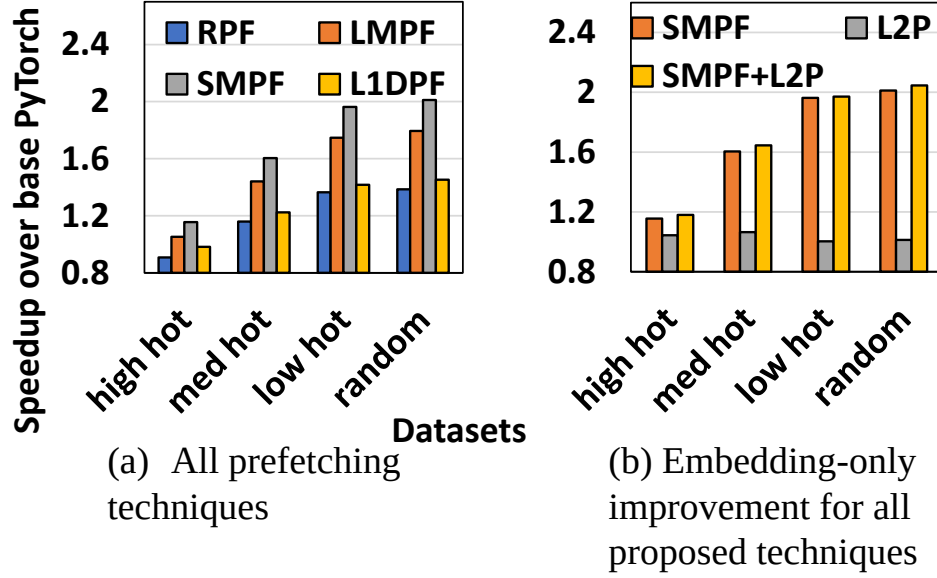


Figure 5.16: Comparison of techniques for off-the-shelf PyTorch.

the schemes tested, we note that L1DPF suffers the most from this overhead. We further note this overhead becomes more critical in the high hot cases where the prefetching finds less opportunity to benefit, resulting in a 15% drop in speedup when compared to the only OptMT.

RPF, SMPF, and LMPF perform very close in all the datasets, with RPF marginally winning. This is because the register file is closest to the execution pipeline when compared to other buffer locations (Table 5.1). Thus, prefetching achieves {34%, 66%, 94%, 97%} speedups for the {high, med, low} hot and random, respectively.

5.6.2.2 Improvement over Baseline PyTorch without OptMT

Earlier, Figure 5.6 highlighted that OptMT improves the performance over baseline PyTorch, and our proposed techniques can work in conjunction (Figure 5.12) with

OptMT. However, we also evaluate our proposed schemes directly over baseline (no OptMT) to validate their effectiveness in the original situation. Here, we compare all prefetching schemes with their optimal prefetch distance (i.e., $\{4,10,10,5\}$ for $\{RPF, SMPF, LMPF, \text{ and } L1DPF\}$, respectively) (Figure 5.16.a). Compared to Figure 5.15, the winning scheme is not RPF but SMPF. Further, LMPF performs second, L1DPF third, and RPF fourth. For the high hot case, RPF and L1DPF underperforms due to higher instruction overheads. Also, SMPF enhances the performance of all the datasets. When comparing SMPF to RPF+OptMT, it matches the performance in the low hot and random datasets, and slightly underperforms in the high and med hot datasets. We noticed that nvcc compiles the SMPF implementation with 32 warps per SM, instead of 24 (as used in off-the-shelf PyTorch). Using the higher benefit of multi-threading (Figure 5.6), this makes SMPF perform better than LMPF, thus making it winning scheme. In contrast, for RPF, nvcc allocates more registers per kernel as the prefetch distance increases, leading to very few (16) warps per SM (for distances ≥ 5), thus severely hurting performance.

Considering the winning scheme to be SMPF, Figure 5.16.b, highlights the embedding-only improvement of L2P and SMPF+L2P over the base PyTorch. L2P improves the high hot and med hot cases by 4.5% and 6.4%, respectively, while marginally improving for the low hot and random cases. Further, L2P combines with SMPF and further enhances the performance over SMPF only. When comparing to the benefit coming with OptMT (Figure 5.12), it can be seen that SMPF + L2P matches the performance for the low hot and random datasets while slightly underperforming for the high hot and med hot cases. This is because OptMT helps in hiding the latency without adding any instruction overhead. It is also interesting to note that L2P+OptMT performs quite well over OptMT, since L2P better holds the data which otherwise could get evicted due to thrashing across warps.

5.6.2.3 Heterogeneous Table Mixing

Given that DLRMs are executed in both homogeneous [107, 136] and heterogeneous settings [116, 121], we evaluate a synthetic case where non-uniform tables (Table 5.7) are present within the embedding stage by having a mixture of hotness. Figure 5.17 compares the performance of all proposed schemes in association with OptMT. In general, all schemes perform better on a higher mix due to more contributions from the low hot and random datasets. Further, within any mix, the combined scheme performs the best by improving over any individual scheme.

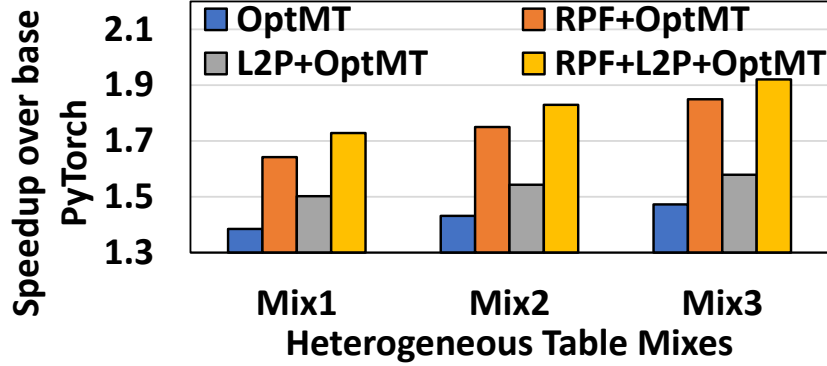


Figure 5.17: Embedding-only improvement of the proposed techniques with OptMT over off-the-shelf PyTorch and heterogeneous tables.

5.6.2.4 Evaluation on H100 NVL GPU

We also evaluate the applicability of our proposed schemes on an H100-NVL GPU [201], which is increasingly being embraced by the datacenters [202, 203]. H100 NVL has 132 SMs (with a total of 16896 CUDA cores), 192KB L1, 50MB L2, 3.84 TBps HBM3 (at 2.7 GHz DDR). The measured base-PyTorch latency values (in us) are {174, 228, 282, 295} for the {high, medium, low, random} datasets, respectively. Thus, H100 gives an average 47% uplift in performance (comparing with (Table 5.4)). Notably, our optimization designs on A100 perform 23% faster than the H100 base performance, thus making it a more cost-effective solution than simply adopting more expensive GPUs.

Figure 5.18 sweeps through possible WLP configurations and finds maximum gain at 32 resident warps (which is different from 40 warps for A100). Similar to Figure 5.6, higher gains of MT are visible for low hot and random cases. Finally, Figure 5.19 shows the performance benefit of RPF+L2P+OptMT for H100 and compares it with A100. For both OptMT and integrated schemes, H100 observes a little lower speedup compared to A100 which is due to the microarchitectural differences, particularly with H100 having 27% faster SM clock, 33% larger L1D\$, 25% larger L2\$, 20% wider HBM width, and 64% faster HBM clock. Yet, we continue to see significant speedups for all datasets (up to 84%).

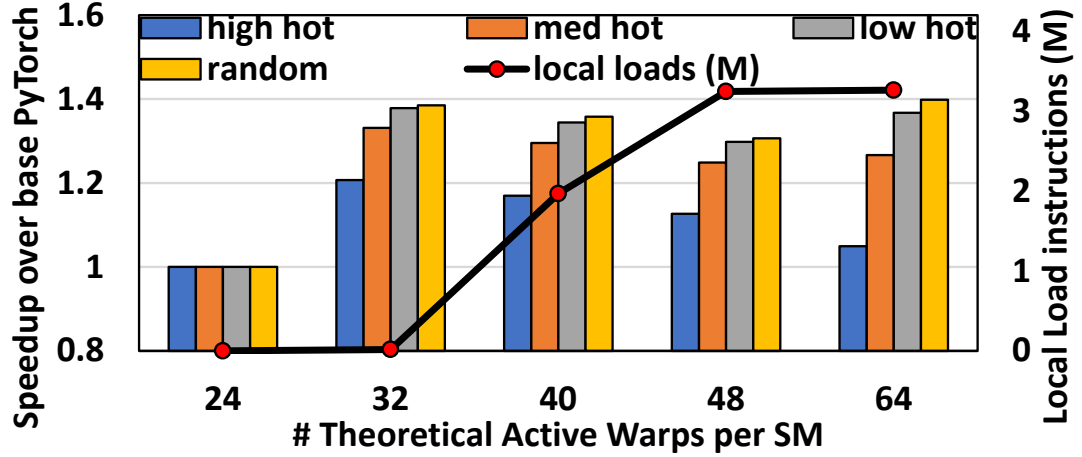


Figure 5.18: On H100 NVL GPU, the number of registers allocated is varied to find optimal WLP. The primary y-axis is speedup over off-the-shelf PyTorch and the secondary y-axis is the register spilling penalty based on extra local memory loads (in millions). OptMT on H100 refers to the highest speedup at 32 warps.

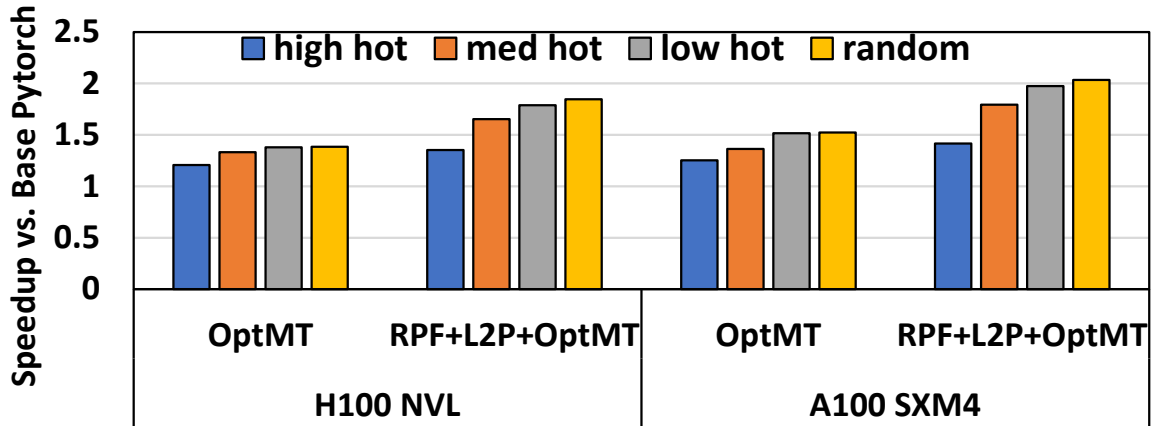


Figure 5.19: Comparison of Embedding-only improvement in latency of the integrated scheme over off-the-shelf PyTorch for H100 NVL and A100 GPU.

5.7 Discussion

A Static Profiling Framework for adoption of our proposed designs: Achieving the ideal performance (like Figure 5.12) requires finding the optimal design points for multi-threading and prefetching (where and when to prefetch). We propose a static profiling framework instead of developing an analytical model or heuristics. As the default hardware involves various in-place optimizations like memory-level-parallelism in conjunction with multi-threading, it makes it difficult to holistically capture the complexities with a heuristic. Additional challenges arise with proprietary nvcc and

limited public details on GPU microarchitecture, making the heuristics susceptible to errors.

Thus, we introduce a static profiling framework aimed at conducting design space exploration to identify the most effective design points. The steps are as follows: (i) Assess if the kernel is memory latency bound by checking the memory access patterns, cache misses, and long latency scoreboard stalls. (ii) Assess if the kernel occupancy is maximum. If not, check the usage of register, shared memory, and kernel launch configurations. (iii) If register usage is high, OptMT can be found by varying the allocated registers. Use nvcc compiler flag “-maxrregcount” to control the assigned registers, where the needed registers are $\leq (max_registers_per_SM)/((desired_active_warps) * (warp_size))$. (iv) After applying OptMT, assess if the kernel is still memory latency bound. If yes, carefully-tuned pinning and prefetching can help. (v) Assess if there is scope for applying L2 pinning by checking for any high reuse behavior towards certain data accesses and comparing the working footprint of kernel with the L2 cache size. If yes, sort the data addresses in descending order based on their reuse amount and apply the steps shown in Figure 5.10. (vi) If the performance is still memory latency bound and memory bandwidth is not saturated (under 80% usage), use prefetching and evaluate performance for different buffer locations as guided in Figure 5.8 by sweeping across the prefetch distances. Note that, when the MT is low, a higher prefetch distance is expected, and vice versa. (vii) Combine both prefetching and pinning.

Generalizability: Following the above Static Profiling Framework, we believe that memory-bound workloads (other than DLRMs) executing on GPUs can benefit from our key contributions with potential applications being Graph Neural Networks [204] and Graph Mining [205].

Scalability and Industrial Adoption: Although we consider model sizes which can fit within one GPU, as our proposed techniques optimize the embedding table granularity, our solutions are applicable for large-scale distributed inference scenarios [149]. Further, the forward pass in the training pipeline [206–208] could benefit from our schemes. By offering a readily deployable and performant solutions with prefetching and pinning, our work opens doors for wider industrial adoption of optimized DLRM inference pipelines.

5.8 Conclusion

With the ever-increasing compute and memory bandwidth requirements of DLRMs, they are increasingly getting adopted on GPUs. However, improving DLRM inference

performance on GPUs needs co-examination of DLRM models and the underlying architectural artifacts. In this work, we show that the embedding stage continues to dominate the DLRM inference pipeline, causing a performance gap of up to $3.2\times$ in the worst case. We show that standard embedding kernels underutilize the warp level parallelism (WLP) offered by the GPU hardware, and can be improved via compiler optimizations. Yet, the optimal WLP is insufficient in fully hiding the long latency load stalls. To tackle this, we propose specialized techniques (software prefetching and L2 pinning), and also combine them. Without requiring any modifications in the hardware or models, our experimental evaluations on A100 and H100 GPU over large models and a variety of datasets indicate performance improvements by up to 103% for the embedding stage, and up to 77% for the overall inference. We set a new benchmark for any future research, and believe that our proposed designs can be generally applied to a wide range of memory-bound kernels.

Chapter 6 |

HetT-HetD: Embracing Heterogeneity in Embedding Tables and Devices for Performant Recommendation Systems

6.1 Introduction and Motivation

As highlighted by prior works and the analyses presented in the preceding chapters, the embedding stage continues to be a dominant performance bottleneck in Deep Learning Recommendation Models (DLRMs). As highlighted by prior works and previous chapters, embedding stage continues to remain a major bottleneck in many DLRM workloads. Chapter 2, 3, and 4 focused on CPU-only hardware and explored performance optimizations on single- and multi-core. Chapter 5 looked into GPU-only hardware performance enhancements.

Despite these advances, we argue that *no single device can optimally handle all stages or all embedding tables*. As highlighted by Chapter 4, the embedding tables within the same model can exhibit widely varying memory-reuse behaviors and capacity requirements, while the non-embedding stages remain predominantly compute-bound. At the same time, modern hardware platforms are becoming increasingly capable: CPUs now feature larger cache hierarchies and expanded DRAM capacities, making them highly suitable for high-reuse and large-capacity embedding tables. Conversely, GPUs, equipped with high-bandwidth memory (HBM) and thousands of CUDA cores, can benefit the low-reuse embedding tables and compute-intensive stages.

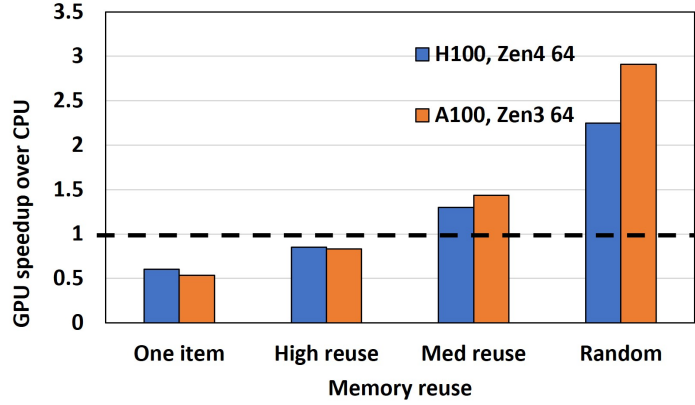


Figure 6.1: GPU speedup over CPU for embedding stage using {H100 GPU, Zen4 64 CPU} and {A100 GPU, Zen3 64 CPU}.

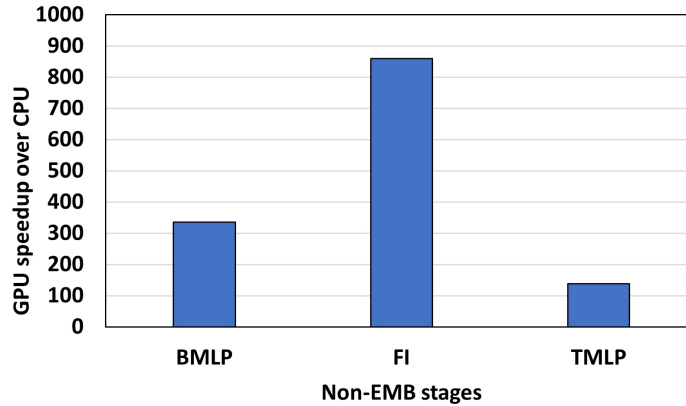


Figure 6.2: GPU speedup over CPU for non embedding stages using H100 GPU and Zen4 64 core CPU.

To better understand how each hardware helps different embedding tables and non-embedding stages, we conduct a performance comparison study as shown in Figure 6.1 and Figure 6.2. We observe that CPUs are favorable choices for high-reuse embedding tables, and GPUs are favorable for low-reuse tables. Further, GPUs heavily favor the bottom-MLP, feature-interaction, and top-MLP stages. Given that CPU can be competitive to GPU for certain embedding tables, we are motivated to deploy a heterogeneous CPU GPU execution.

6.2 Preliminary design

To enable an efficient and balanced heterogeneous execution across CPU and GPU, it is essential to clearly define the key design objectives guiding our framework. In this direction, we would like to define three goals towards achieving a performant heterogeneous deployment. They are:

- Goal#1: Leveraging both CPU and GPU effectively to ensure minimal idle time
- Goal#2: Avoid oversubscribing the GPU memory capacity to prevent embedding table movement over slow PCIe
- Goal#3: Load balancing the work across CPU and GPU to ensure maximal concurrent execution

Towards these goals, an intuitive approach could be to divide among the devices in proportion to their performance. For example, hardware measurements indicate that running an embedding stage with high hot dataset takes 25ms on CPU, and 29.4ms on the GPU. For random dataset, the same embedding stage takes 126ms on CPU and 56ms on GPU. Assume an embedding stage having T tables and B batch size. Thus, when doing batch parallelism, where we assign batch samples to a worker based on its performance, high hot dataset would mean $0.46B$ work on GPU and $0.54B$ work on CPU. Similarly, for random dataset it would mean $0.70B$ work on GPU and $0.30B$ work on CPU. Another intuitive approach could be table parallelism where tables are divided across the devices. In a similar way, high hot dataset would mean $0.46T$ work on GPU and $0.54T$ work on CPU. Similarly, for random dataset it would mean $0.70T$ work on GPU and $0.30T$ work on CPU. This approach is more fine-grained as each batch's sample leverages both the devices.

We note certain limitations with regard to employing batch parallelism. First, CPU and GPU might not be optimally used in cases where the embedding stage is composed of heterogeneous tables. As noted in the previous figure, certain embedding tables could run faster on certain devices. However, batch parallelism does not leverage this opportunity and maps the work based on the complete embedding execution time. For example, consider an embedding stage with 750 tables, of which 500 are high hot and 250 are random. Assuming the model can completely fit in a GPU, Batch parallelism achieves an execution time of 73 ms while table parallelism achieves 56 ms. Second, given that batch parallelism lowers the batch size assigned to each device, this lowers the execution

efficiency as larger batch size is known for higher throughput. Third, given that GPU’s memory budget is only few 10s of GBs (relatively small compared to CPUs), GPUs can quickly fall short in residing all the embedding tables. Thus, batch parallelism has severe limitations in meeting the requirements for first and second goal.

On the other hand, table parallelism does not observe these limitations as it allows to control execution by mapping the high-reuse tables on the CPU and low-reuse tables on the GPU. Thus, it leverages both CPU and GPU by mapping the appropriate tables on the devices. Further, it runs the tables with the original batch size, assisting with higher compute utilization with larger batch sizes. These advantages help with our first goal towards efficiently leveraging CPU and GPU. Moreover, the GPU need to only house the tables up to it’s memory budget, and remaining embedding tables can be offloaded and executed by the CPU. This helps with our second goal towards avoiding GPU memory oversubscription.

6.2.1 Key Design Decisions

Given that we have understood why to leverage heterogeneous devices for embedding tables, let’s establish the fundamental questions:

- What and where to offload? Embedding tables are run by both CPU and GPU. Non-embedding stages are offloaded to the GPU.
- When to offload? The tasks are decided statically in the beginning of inference server. Particularly, the mapping of embedding tables is chosen carefully based on the underlying hardware platform and table execution properties. In case the embedding model gets updated, new mapping can be quickly found and corresponding tables can be re-offloaded.
- How to offload? We need to design a right mechanism and policy to concurrently execute tables on cPu and GPU, and for the embedding table placement.

6.3 Optimization Design

To achieve a performant execution for meeting all of the three goals stated above, we first need an optimal mechanism for concurrently running all stages in DLRM on CPU and GPU with minimal exposed overheads orchestrating the work on CPU and GPU.

Next, we need an optimal policy to be deployed on top of our mechanism for efficiently mapping the tables on CPU and GPU.

6.3.1 Mechansim:

To identify an effective mechanism for concurrently executing embedding tables on the CPU and GPU, we begin by evaluating a textbook approach using CUDA streams. The goal is to enable concurrent execution across both devices assuming no data movement overheads. In this setup, a load-balanced number of embedding tables are mapped to the CPU and GPU.

To achieve concurrency, CUDA streams are employed to make GPU operations non-blocking. The GPU work is launched first, followed by CPU execution, with the CPU parent process sequentially launching embedding table computations on the GPU. However, the number of CUDA kernels to be launched is proportional to the number of embedding tables assigned to the GPU, which can easily reach into the hundreds. Many of these kernels complete within tens of microseconds. Consequently, sequential dispatch along with the short-lived GPU kernels offer limited opportunity for effective overlap with CPU execution. For the embedding tables running on CPU, we use table threading scheme as discussed in Chapter 3 which is more performant than in-built batch-threading scheme in pytorch. Overall, we observe an 18% deviation in performance compared to the idealized case of perfect overlap. We believe this performance to further degrade when data-movement overheads are considered.

To address the limited overlap caused by sequential kernel dispatch and the dependency of GPU progress on the CPU parent process, we decouple the GPU handler from the CPU execution by defining two process territories. Figure 6.3 shows the embedding-only concurrent design for an underlying AMD EPYC Zen4 CPU having 64 cores. We use python multi-processing to separate the work of CPU and GPU. Each worker process does pre-processing tasks like model and dataset loading before hitting a barrier which is used to synchronize the start of inference server. After the barrier, CPU and GPU runs their assigned portion of embedding tables (still assuming no data movement overheads). With this approach, we observe that the runtime matches ideal performance overlap. However, this approach is limited to purely studying embedding-only concurrent execution.

Towards achieving a concurrent end-to-end execution, we have to carefully orchestrate the execution of both embedding and non-embedding stages. Figure 6.4 showcases the data-flow graph to be realized for one batch of work. The non-embedding stages being compute-bound are mapped onto the GPU. For the embedding stage, a portion of tables

are mapped to CPU and remaining fraction is mapped to the GPU. We later discuss how to identify the tables for mapping. Our implementation follows the following steps:

- **Two-process CPU–GPU pipeline.**
 - We use `torch.multiprocessing` with the `spawn` backend to create two OS processes: a CPU producer (`cpu_worker`) and a GPU consumer (`gpu_worker`).
 - A `Barrier(2)` ensures both processes enter the main loop in lockstep before overlapped execution begins.
 - A multiprocessing `Queue` is used to hand off per-batch work items from the CPU process to the GPU process.
- **Parallel CPU embedding computation + pinned outputs.**
 - All embedding tables are implemented as `nn.EmbeddingBag` modules in a `DLRM_Net` model.
 - For each batch, we launch all embedding lookups in parallel using TorchScript futures: `torch.jit._fork(emb_l[k], lS_i[k], lS_o[k])`.
 - After each lookup completes, we immediately move the result into page-locked host memory by calling `tensor.pin_memory()` (also via `_fork`).
 - Thus, the CPU model returns a list of pinned tensors, one per embedding table, which are ready for asynchronous host-to-device copies.
- **Pre-allocated device buffers for embedding outputs.**
 - On the CPU process, we pre-allocate a fixed array of device tensors, one per table: `dev_buf[i] = torch.empty((2048, 128), device=cuda0)`.
 - These buffers are reused across all batches, eliminating allocator overhead from the critical path.
 - The pointers to these device buffers are later shared with the GPU process via CUDA IPC, so the consumer can directly read the data without additional copies.
- **Asynchronous host-to-device copies on a dedicated CUDA stream.**
 - The CPU process creates a dedicated CUDA stream (`torch.cuda.Stream(cuda0)`) for data movement.

- For each batch, we enqueue one `cudaMemcpyAsync`-equivalent operation per table by calling: `dev_buf[i].copy_(ly[i], non_blocking=True)`.
 - Because the sources `ly[i]` reside in pinned memory and we pass `non_blocking=True`, these copies are truly asynchronous with respect to the host CPU thread.
 - The copies are ordered on the data-movement stream, and the CPU immediately continues to compute the next batch’s embeddings while the DMA engines transfer the current batch.
- **Inter-process CUDA event-based synchronization.**
 - After enqueueing all H2D copies for a batch, the CPU process records a CUDA event on the same stream: `sync_ev = torch.cuda.Event(interprocess=True)` and `sync_ev.record(stream)`.
 - The event is created with `interprocess=True`, enabling it to be shared across processes via CUDA IPC.
 - The CPU process then pushes a tuple `(batch_index, dev_buf, sync_ev)` to the multiprocessing queue.
 - On the GPU process, we call `stream.wait_event(sync_ev)`, which guarantees that subsequent work on the GPU compute stream only begins after all H2D copies for that batch have completed.
 - **GPU-side computation with mixed CPU and GPU embeddings.**
 - The GPU process maintains its own copies of the bottom MLP and a second set of embedding tables; these are moved to the GPU once at startup.
 - For each batch z :
 - * The GPU process first computes the bottom MLP on preloaded GPU inputs (`bmlp_inp[z]`).
 - * It performs embedding lookups for the GPU-resident tables (`gpu_model`) to obtain `ly1`.
 - * It receives the device buffers `ly_cpu` and the event `sync_ev` from the queue, waits on the event, and then performs feature interaction on the concatenated list `ly1 + ly_cpu`.
 - * Finally, it applies the top MLP to the interacted features to produce the batch output.

- Because the CPU-produced embeddings are already resident in device memory when the event fires, the feature interaction and top MLP operate purely on GPU data.
- **End-to-end pipeline overlap.**
 - This design forms a three-stage pipeline:
 1. CPU process computes embedding bags for batch $k + 1$ and pins the outputs.
 2. Asynchronous H2D copies transfer the pinned outputs for batch $k + 1$ into pre-allocated device buffers.
 3. GPU process computes bottom MLP, GPU embeddings, feature interaction, and top MLP for batch k , consuming the data from the previous iteration.
 - The cross-process CUDA event ensures correct ordering of the copy and compute stages, while allowing CPU compute, DMA transfers, and GPU compute to execute concurrently.
 - As a result, we effectively overlap CPU embedding computation, PCIe/NVLink data movement, and GPU-side computation for different batches, improving end-to-end inference throughput.

Towards running complete DLRM model on CPU and GPU, Figure 6.5 shows the timeline diagram where we run embedding tables on CPU, BMLP stage on GPU, embedding tables on GPU. For running the FI stage on GPU, it waits for the outputs from both CPU and GPU after which TMLP stage is executed on the GPU.

6.3.2 Policy:

As discussed earlier, the embedding stage exhibits heterogeneity across tables in terms of memory reuse and capacity requirements. Executing all tables exclusively on the CPU favors those with high reuse and large sizes due to CPU’s higher cache and larger DRAM but results in suboptimal performance for low-reuse tables. Conversely, executing all tables solely on the GPU benefits low-reuse tables due to higher memory bandwidth; however, this approach demands multiple GPUs to accommodate the entire embedding stage and yields lower performance for high-reuse tables. In case a single GPU is used, the data transfer over PCIe to load embedding table becomes the bottleneck. Prior work [114]

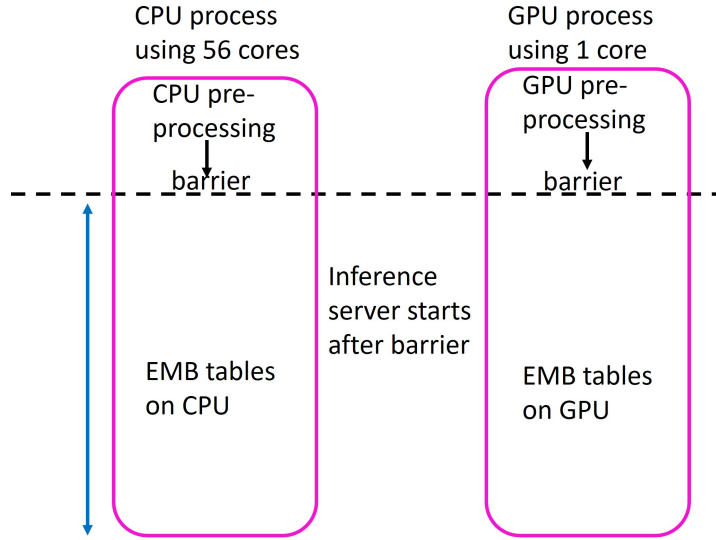


Figure 6.3: Concurrent execution design for embedding stage on CPU and GPU

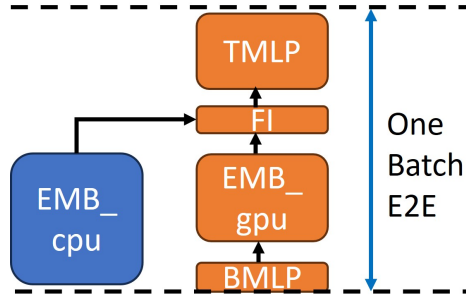


Figure 6.4: End-to-end dataflow for one batch of DLRM inference where blue represents tasks mapped onto CPU and orange represents tasks mapped onto GPU. A portion of embedding stage is mapped onto CPU. GPU first executes the bottom-MLP (BMLP) stage, followed by the remaining portion of embedding tables. Feature interaction (FI) stage waits for the output for embedding portions from both CPU and GPU and is followed by the top-MLP (TMLP) stage.

assumes power-law behavior in embedding accesss and proposes two optimizations for training DLRMs: (1) embedding stage onto CPU and non-embedding stage onto GPU (2) improving with putting high-reuse entries on the GPU. We note that approach (1) resembles the pure-CPU embedding deployment and is thereby inefficient. And, approach (2) brings overheads due to embedding entry to device mapping sequential check. Further (2) could suffer from frequent data transfers between CPU and GPU in the presence of medium and low reuse tables with high pooling operations – all of these could impact the tight latency requirements of inference.

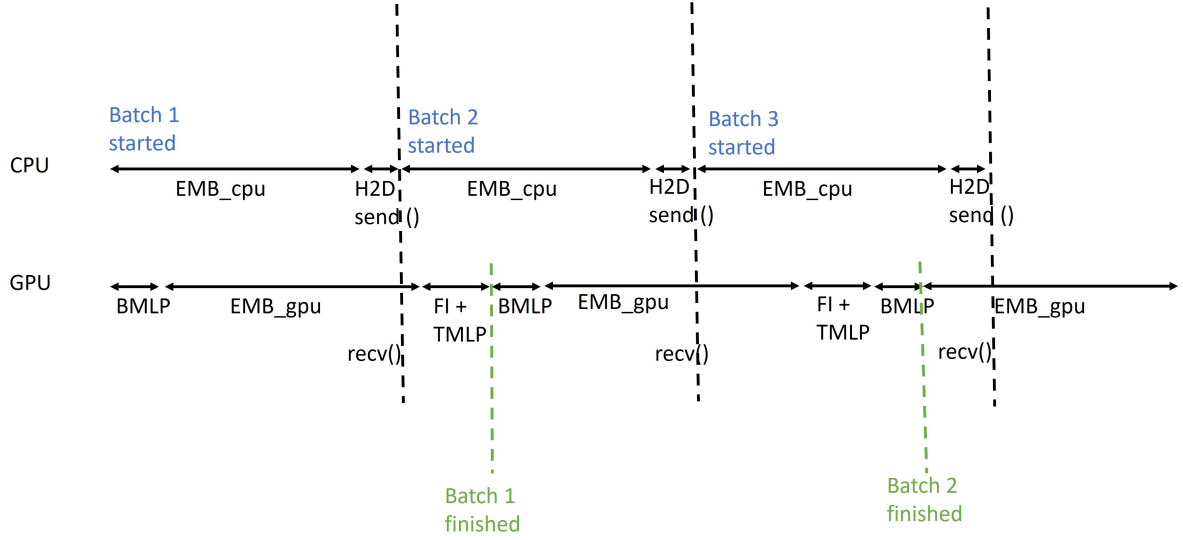


Figure 6.5: Concurrent execution design for end to end DLRM on CPU and GPU

Given that the DLRM embedding stage involves hundreds to thousands of embedding tables, the key question becomes: *how can we systematically decide the appropriate device assignment for each table?*

To address this, we develop a policy-driven mapping strategy that assigns embedding tables to CPU or GPU based on their access characteristics, size, and the load-balance constraints between devices. We make the following rationales: (1) Computing GPU time proxy based on CPU time proxy using a linear equation (2) For GPU, we would like to pick the tables which have high execution time and low memory capacity. For this, we divide the GPU time value with memory capacity of the table, and sort the tables based on this score. (3) The final picks are based on previous score, load balancing on CPU and GPU, and avoiding memory capacity oversubscription on the GPU. Figure 6.6 shows an example.

The detailed steps are below:

- **Step 1: Characterization of Embedding Tables** Each embedding table is profiled to extract three key parameters:
 - **Average Memory Access Time (AMAT):** represents temporal locality.
 - **Average Pooling Factor (PF):** average number of accesses per table
 - **CPU Table Load (CPU TL):** proxy for CPU time execution similar to Chapter 4. This is a product of AMAT and PF.

- **Step 2: Modeling GPU Performance** We assume a linear relationship between CPU and GPU performance across embedding reuse. Using a linear equation, the GPU TL can be modeled based on CPU TL. Given that AMAT on CPU ranges from 1 to 15 and corresponding GPU speedup over CPU ranges from 0.6 to 2.25, we solve a linear equation ‘ $y = mx + c$ ’

$$y = mx + c$$

where x is CPU TL and y represents GPU TL.

From the profiling data, the slope and intercept are determined as:

$$y = 0.118x + 0.482 \quad \Rightarrow \quad \text{GPU TL} = \frac{\text{CPU TL}}{y}$$

- **Step 3: Identifying GPU-Preferable Tables** The next step leverages two key intuitions:
 1. Tables with smaller memory footprints are better candidates for GPU placement, as they fit within limited GPU HBM capacity.
 2. Tables with higher “GPU TL” benefit more from GPU acceleration. This assumes tables with low reuse have higher TL and may not always hold when compared with high reuse tables with large PF.

A metric, *GPU TL per GB*, is derived for all tables, and they are sorted in decreasing order of this value to create a ranked list of embedding tables most beneficial for GPU placement.

- **Step 4: Capacity and Load-Balance Constraints.** After sorting, the algorithm incrementally assigns tables to GPU following this ranking while checking two critical constraints:
 - **GPU Memory Capacity Constraint:** The cumulative size of selected tables must not exceed available GPU memory.
 - **Load-Balance Constraint:** The cumulative latency (TL) between CPU and GPU must remain balanced to ensure both devices are effectively utilized.

If either constraint is violated, the allocation process stops, and the remaining tables are assigned to CPU.

Table ID	Size (GB)	AMAT	PF	CPU TV	GPU TV	GPU TV / GB
9	0.02	15	6	90	39.96448	1998.224
4	0.05	15	10	150	66.60746	1332.149
10	0.08	5	10	50	46.64179	583.0224
1	0.25	1	10	10	16.66667	66.66667
2	0.5	5	2	10	9.328358	18.65672
8	1	3	5	15	17.94258	17.94258
3	2	15	3	45	19.98224	9.991119
6	1	3	1	3	3.588517	3.588517
7	1	2	1	2	2.785515	2.785515
5	3	3	1	3	3.588517	1.196172

Figure 6.6: Example policy for achieving an optimal mapping of tables on CPU and GPU

6.4 Evaluation

Methodology: We conduct our experiments on a heterogeneous CPU–GPU platform consisting of an AMD Zen4 CPU with 64 cores and an NVIDIA H100 GPU. The evaluation uses both *high-hot* and *random* datasets to capture varying degrees of memory reuse and access locality. The DLRM model under study comprises 500 embedding tables. We consider two configurations: (1) Homogeneous configuration: All 500 embedding tables follow a uniform access pattern. (2) Heterogeneous configuration: 250 tables are high-hot (high reuse) and 250 tables are random (low reuse). The embedding dimension is 128.

Results: We consider variety of baselines including pure CPU, pure GPU, naive CPU GPU, and our proposed scheme. The naive CPU GPU design maps embedding stage onto CPU and non-embedding stage onto GPU. The pure GPU scheme considers one GPU and thus suffers from PCIe data transfer overheads to load the embedding tables.

Figure 6.7 and 6.8 compares the performance of our scheme with baseline. In the homogeneous dataset, we note that our scheme performs up to 1.9x better than naive CPU-GPU design. In the heterogeneous dataset, we also evaluate two policies: (1) equal mapping refers to mapping all high reuse tables on CPU and low reuse tables on GPU (2) random mapping refers to mapping 125 high reuse and 125 random tables on both CPU and GPU. Overall, we note that our scheme performs 3.7x better than naive CPU-GPU design.

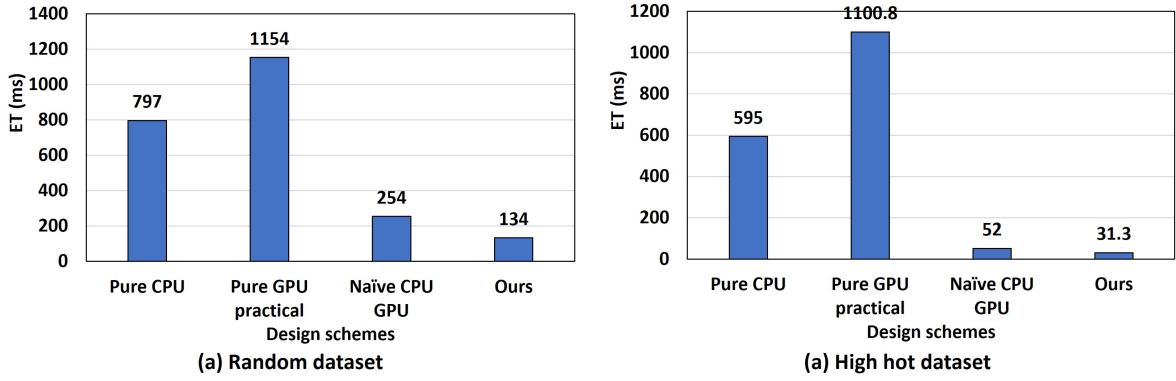


Figure 6.7: End to end speedup on homogeneous datasets.

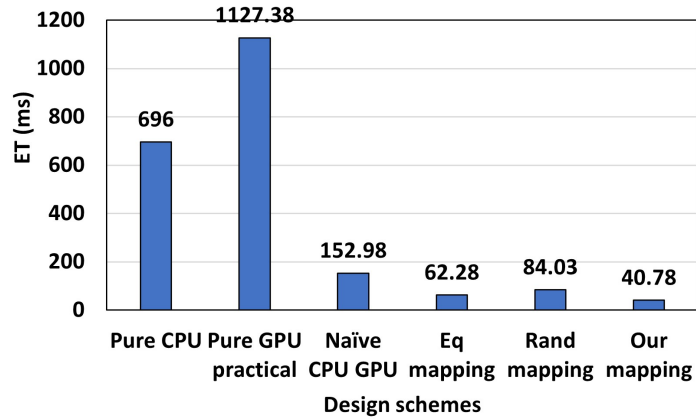


Figure 6.8: End to end speedup on heterogeneous datasets.

6.5 Conclusion

In summary, our study notes that CPU performance can be competitive with that of GPUs, even for large batch sizes. We discuss how the embedding stage can be effectively partitioned across the CPU and GPU, allowing each device to exploit its architectural strengths—large caches and memory capacity on the CPU, and high bandwidth on the GPU. We show that fine-tuning the partitioning and execution strategy of this heterogeneous deployment is crucial to achieving high overall performance, as balanced load distribution and efficient coordination between devices lead to significantly improved end-to-end DLRM execution.

Chapter 7 |

Conclusions and Future Work

7.1 Conclusions

As the compute, memory capacity, and memory bandwidth requirements of DLRMs grows, there is an increasing demand for both CPUs and GPUs. However, enhancing DLRM inference performance on these devices requires a simultaneous examination of DLRM models and the underlying hardware’s architectural elements. Through a series of five contributions, we systematically identified architectural bottlenecks, proposed software-driven application- and architecture-aware solutions, and demonstrated substantial performance improvements—all achieved without modifying the underlying hardware.

Chapter 2 analyzes several DLRM workloads and find that the embedding stage dominates end-to-end execution time. By classifying datasets based on access hotness and profiling the cache reuse behavior, we identify memory stalls as key bottleneck. To address these, we propose customized software prefetching and computation overlapping via hyperthreading to minimize the embedding stage overhead and overall execution latency. Our experimental results show that the proposed techniques improve the performance of embedding heavy & mixed models for various hotness datasets by up to 1.59x.

As the latest DLRMs show embedding stages growing with more tables, larger embedding dimensions, and pooling operations, assigning one batch per core limits scalability and violates latency goals. Chapter 3 highlights the limited multi-core scalability of existing techniques towards meeting low-latency inference goals. To address this, we propose improving the performance of a single batch by parallelizing it over multiple cores and introduce two new parallelization techniques: Table Threading (TT) and Hierarchical Threading (HT). We demonstrate up to 9.9x speedup on homogeneous

datasets and 2.95x speedup on heterogeneous datasets.

Chapter 4 highlights that HT is still imperfect in multi-core scaling over heterogeneous datasets. The heuristics and performance metrics developed in this work (TL heuristic and WMLP metric) have been successful in mitigating the inefficiencies. In addition, we demonstrate how load imbalance across threads and CCDs, suboptimal core allocation for embedding tables, and inefficient utilization of MLP resources can be addressed by our proposed mechanisms and techniques. Our technique show $1.67\times$ higher speedup over prior state-of-the-art DLRM with 96 cores.

Given the rapidly growing compute and memory bandwidth demands of modern DLRMs, GPUs have emerged as the natural choice for inference due to their massive parallelism and high-bandwidth memory (HBM) support. Chapter 5 studies the DLRM performance on GPUs and finds that the embedding stage remains the primary bottleneck. Through detailed microarchitectural characterization, we identify low kernel occupancy and long memory stalls, and address them via compiler optimizations, software prefetching, and L2 pinning. Evaluations on NVIDIA A100 and H100 GPUs show up to $2.03\times$ embedding and $1.77\times$ overall speedup, achieved entirely through software-only, plug-and-play techniques without any hardware or model modifications.

Building on insights from CPU- and GPU-specific optimizations, Chapter 6 introduces HetT-HetD, a heterogeneous execution framework that leverages the complementary strengths of CPUs and GPUs. While GPUs handle low-reuse tables and compute-intensive stages efficiently, CPUs are better suited for high-reuse and large-capacity tables. We achieve this balance through asynchronous CPU-GPU data movement and a heuristic-based scheduling policy for table-to-device mapping. Our evaluations demonstrate up to $3.7\times$ end-to-end speedup over naive CPU-GPU design.

7.2 Broader Insights

While our proposed techniques are developed in the context of DLRM inference, the underlying principles extend naturally to other recommendation and deep learning models that rely on embedding lookups, such as DIN and DIEN. These models can directly benefit from the same CPU-GPU co-execution and scheduling strategies. Furthermore, similar opportunities exist for optimizing DLRM training workloads through heterogeneous execution and memory-aware scheduling.

From an architectural standpoint, our findings highlight the importance of software prefetching mechanisms for efficiently handling large-dimensional vector operations

common in both deep learning and high-performance computing (HPC) kernels. In addition, we observe that multi-core scaling on chiplet-based many-core CPUs presents a promising direction for further enhancing the performance of embedding-heavy workloads by improving memory locality and parallel efficiency across compute tiles.

7.3 Future Work

7.3.1 DLRM Training

The dissertation demonstrated powerful techniques for improving the performance of DLRM inference. Though the training workload leverages GPU for throughput optimizations, many of the proposed techniques can be beneficial. This is because embedding loads continue to persist for two reasons: (1) forward pass in training is the same as inference (2) updating of embedding vectors in backward pass involves loading them first for gradient calculation. Thus, the ideas proposed in Chapter 5 which improves the performance of embedding loads can be leveraged by training workloads. Further, CPUs can be leveraged as demonstrated in Chapter 6.

7.3.2 Multi-device load balancing

As noted by prior works, load balancing has been a critical concern when employing multiple devices for both DLRM training and inference. This is challenging since embedding tables differ in their memory capacity and execution time parameters. The proposed techniques in Chapter 4 and 6 demonstrates how to analytically model the time and memory-level-parallelism requirements per table, and using it to achieve an optimal table placement. These techniques can be extended for a large number of workers including heterogeneous devices. Specifically, each device can be modeled based on it's memory capacity budget, and execution time on diverse set of memory accesses.

7.3.3 Analyzing accuracy vs.performance tradeoffs

Given that DLRM only predicts the likelihood of an item to be clicked by an user, several approximations can be studied to improve the performance of inference. These are following: (1) Given that low-reuse loads exist which are in the critical path of inference, could we drop some of the low-reuse loads without sacrificing much on performance? (2) Further, since memory capacity is a major requirement of embedding tables, could we

assess the accuracy contributions of each table and drop some of the embedding tables which have minimal accuracy contributions but occupy a reasonable memory footprint?

(3) Moreover, since some of the tables execution time is significantly large, could we drop their computation in cases where they don't have enough accuracy impact?

7.3.4 Going Beyond DLRMs

Several other applications exist which are memory-bound or have a mix of memory- and compute-bound stages. Specifically, applications like GNN and graph mining have similar properties. Further, newer recommendation models like generative recommenders [209] rely on element-wise operations (like SiLU activation, scaling, attention masking, several bias operations, layer normalization, tensor manipulation, and bucket weight lookup) which are memory-bound and accounts for significant fraction of encoder latency. These applications can leverage the techniques proposed in our dissertation for performant execution.

Bibliography

- [1] TROESTER, K. and R. BHARGAVA (2023) “AMD Next Generation “Zen 4” Core and 4th Gen AMD EPYC™ 9004 Server CPU,” in *2023 IEEE Hot Chips 35 Symposium (HCS)*, pp. 1–25.
- [2] GUPTA, U., S. HSIA, V. SARAPH, X. WANG, B. REAGEN, G.-Y. WEI, H.-H. S. LEE, D. BROOKS, and C.-J. WU (2020) “Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 982–995.
- [3] AMD (2022), “AMD Epyc 9654 CPU, code-named Genoa,” "<https://www.amd.com/en/product/12191>".
- [4] LUO, W., R. FAN, Z. LI, D. DU, Q. WANG, and X. CHU (2024) “Benchmarking and Dissecting the Nvidia Hopper GPU Architecture,” *arXiv preprint arXiv:2402.13499*.
- [5] META (2024), “Facebook Recommendation System,” "<https://ai.meta.com/blog/ai-unconnected-content-recommendations-facebook-instagram/>".
- [6] ——— (2024), “Instagram Recommendation System,” "<https://engineering.fb.com/2023/08/09/ml-applications/scaling-instagram-explore-recommendations-system/>".
- [7] TIKTOK (2024), “TikTok Recommendation System,” "<https://www.tiktok.com/transparency/en-us/recommendation-system/>".
- [8] AMAZON (2024), “Amazon Recommendation System,” "<https://aws.amazon.com/personalize/>".
- [9] NETFLIX (2024), “Netflix Recommendation System,” "<https://research.netflix.com/research-area/recommendations>".
- [10] HULU (2024), “Hulu Recommendation System,” "<https://help.hulu.com/article/hulu-personalized-recommendations#:~:text=While%20you're%20looking%20for,get%20to%20know%20you%20better.>".

- [11] META (2024), “Meta Q1 2024 Financial Report,” "https://s21.q4cdn.com/399680738/files/doc_financials/2024/q1/Earnings-Presentation-Q1-2024.pdf".
- [12] GOOGLE (2024), “Google Q2 2024 Financial Report,” "<https://abc.xyz/assets/ae/e9/753110054014b6de4d620a2853f6/goog-10-q-q2-2024.pdf>".
- [13] GVR (2024), “Recommendation Engine Market Size,” "<https://www.grandviewresearch.com/industry-analysis/recommendation-engine-market-report>".
- [14] NAUMOV, M., D. MUDIGERE, H.-J. M. SHI, J. HUANG, N. SUNDARAMAN, J. PARK, X. WANG, U. GUPTA, C.-J. WU, A. G. AZZOLINI, D. DZHULGAKOV, A. MALLEVICH, I. CHERNIAVSKII, Y. LU, R. KRISHNAMOORTHY, A. YU, V. KONDRATENKO, S. PEREIRA, X. CHEN, W. CHEN, V. RAO, B. JIA, L. XIONG, and M. SMELYANSKIY (2019), “Deep Learning Recommendation Model for Personalization and Recommendation Systems,” 1906.00091.
- [15] GUPTA, U., C.-J. WU, X. WANG, M. NAUMOV, B. REAGEN, D. BROOKS, B. COTTEL, K. HAZELWOOD, M. HEMPSTEAD, B. JIA, ET AL. (2020) “The architectural implications of facebook’s dnn-based personalized recommendation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 488–501.
- [16] META (2024), “Meta Q2-2024 Earnings Call,” "<https://investor.atmeta.com/investor-events/event-details/2024/Q2-2024-Earnings-Call/default.aspx>".
- [17] ALPHABET (2024), “Alphabet Q2-2024 Earnings Call,” "<https://www.sec.gov/Archives/edgar/data/1652044/000165204424000076/googexhibit991q22024.htm>".
- [18] ACUN, B., M. MURPHY, X. WANG, J. NIE, C.-J. WU, and K. HAZELWOOD (2021) “Understanding training efficiency of deep learning recommendation models at scale,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, pp. 802–814.
- [19] KE, L., U. GUPTA, M. HEMPSTEADIS, C.-J. WU, H.-H. S. LEE, and X. ZHANG (2022) “Hercules: Heterogeneity-Aware Inference Serving for At-Scale Personalized Recommendation,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, pp. 141–144.
- [20] WU, C.-J., D. BROOKS, K. CHEN, D. CHEN, S. CHOUDHURY, M. DUKHAN, K. HAZELWOOD, E. ISAAC, Y. JIA, B. JIA, T. LEYVAND, H. LU, Y. LU, L. QIAO, B. REAGEN, J. SPISAK, F. SUN, A. TULLOCH, P. VAJDA, X. WANG, Y. WANG, B. WASTI, Y. WU, R. XIAN, S. YOO, and P. ZHANG (2019) “Machine Learning

- at Facebook: Understanding Inference at the Edge,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 331–344.
- [21] KANNAN, R. S., L. SUBRAMANIAN, A. RAJU, J. AHN, J. MARS, and L. TANG (2019) “Grand slam: Guaranteeing slas for jobs in microservices execution frameworks,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–16.
 - [22] BHASI, V. M., J. R. GUNASEKARAN, P. THINAKARAN, C. S. MISHRA, M. T. KANDEMIR, and C. DAS (2021) “Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms,” in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 153–167.
 - [23] SINGHVI, A., A. BALASUBRAMANIAN, K. HOUCK, M. D. SHAIKH, S. VENKATARAMAN, and A. AKELLA (2021) “Atoll: A scalable low-latency serverless platform,” in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 138–152.
 - [24] BHASI, V. M., J. R. GUNASEKARAN, A. SHARMA, M. T. KANDEMIR, and C. DAS (2022) “Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms,” in *Proceedings of the 13th Symposium on Cloud Computing*, pp. 257–272.
 - [25] NAUMOV, M., D. MUDIGERE, H.-J. M. SHI, J. HUANG, N. SUNDARAMAN, J. PARK, X. WANG, U. GUPTA, C.-J. WU, A. G. AZZOLINI, ET AL. (2019) “Deep learning recommendation model for personalization and recommendation systems,” *arXiv preprint arXiv:1906.00091*.
 - [26] HSIA, S., U. GUPTA, M. WILKENING, C.-J. WU, G.-Y. WEI, and D. BROOKS (2020) “Cross-stack workload characterization of deep recommendation systems,” in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, pp. 157–168.
 - [27] PYTORCH (2024), “Embedding Bag CUDA Kernel in PyTorch,” "<https://github.com/pytorch/pytorch/blob/da7db5d345a10ffb5092b26c5159f56faec1d0ea/aten/src/ATen/native/cuda/EmbeddingBag.cu#L115>".
 - [28] FIROOZSHAHIAN, A., J. COBURN, R. LEVENSTEIN, R. NATTOJI, A. KAMATH, O. WU, G. GREWAL, H. AEPALA, B. JAKKA, B. DREYER, A. HUTCHIN, U. DIRIL, K. NAIR, E. K. AREDESTANI, M. SCHATZ, Y. HAO, R. KOMURAVELLI, K. HO, S. ABU ASAL, J. SHAJRAWI, K. QUINN, N. SREEDHARA, P. KANSAL, W. WEI, D. JAYARAMAN, L. CHENG, P. CHOPDA, E. WANG, A. BIKUMANDLA, A. KARTHIK SENGOTTUVEL, K. THOTTEMPUDI, A. NARASIMHA, B. DODDS, C. GAO, J. ZHANG, M. AL-SANABANI, A. ZEHTABIOSKUIE, J. FIX, H. YU, R. LI, K. GONDKAR, J. MONTGOMERY, M. TSAI, S. DWARAKAPURAM, S. DESAI, N. AVIDAN, P. RAMANI, K. NARAYANAN, A. MATHEWS, S. GOPAL, M. NAUMOV, V. RAO, K. NORU, H. REDDY, P. VENKATAPURAM, and A. BJORLIN (2023)

- “MTIA: First Generation Silicon Targeting Meta’s Recommendation Systems,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA ’23*, Association for Computing Machinery, New York, NY, USA.
URL <https://doi.org/10.1145/3579371.3589348>
- [29] JOUPPI, N., G. KURIAN, S. LI, P. MA, R. NAGARAJAN, L. NAI, N. PATIL, S. SUBRAMANIAN, A. SWING, B. TOWLES, ET AL. (2023) “Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” in *Proceedings of the 50th annual international symposium on computer architecture*, pp. 1–14.
 - [30] NVIDIA (2024), “Nvidia Hopper GPU WhitePaper,” "<https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper>".
 - [31] ——— (2024), “BlackWell GPU Technical Brief,” "https://cdn.prod.website-files.com/61dda201f29b7efc52c5fbaf/6602ea9d0ce8cb73fb6de87f_nvidia-blackwell-architecture-technical-brief.pdf".
 - [32] AMD (2024), “AMD Instinct MI300,” "<https://www.amd.com/en/products/accelerators/instinct/mi300.html>".
 - [33] META (2023), “Embedding lookup Production dataset,” "https://github.com/facebookresearch/dlrm_datasets".
 - [34] INFERENCE, M. D. (2024), “MLPerf Datacenter Inference 2024,” "<https://mlcommons.org/benchmarks/inference-datacenter/>".
 - [35] WANG, M., Y. LIN, G. LIN, K. YANG, and X.-M. WU (2020) “M2GRL: A multi-task multi-view graph representation learning framework for web-scale recommender systems,” in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2349–2358.
 - [36] ZHOU, G., N. MOU, Y. FAN, Q. PI, W. BIAN, C. ZHOU, X. ZHU, and K. GAI (2019) “Deep interest evolution network for click-through rate prediction,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, pp. 5941–5948.
 - [37] ZHAO, W., J. ZHANG, D. XIE, Y. QIAN, R. JIA, and P. LI (2019) “Aibox: Ctr prediction model training on a single node,” in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pp. 319–328.
 - [38] ZHOU, G., X. ZHU, C. SONG, Y. FAN, H. ZHU, X. MA, Y. YAN, J. JIN, H. LI, and K. GAI (2018) “Deep interest network for click-through rate prediction,” in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 1059–1068.

- [39] CHENG, H.-T., L. KOC, J. HARMSSEN, T. SHAKED, T. CHANDRA, H. ARADHYE, G. ANDERSON, G. CORRADO, W. CHAI, M. ISPIR, ET AL. (2016) “Wide & deep learning for recommender systems,” in *Proceedings of the 1st workshop on deep learning for recommender systems*, pp. 7–10.
- [40] COVINGTON, P., J. ADAMS, and E. SARGIN (2016) “Deep neural networks for youtube recommendations,” in *Proceedings of the 10th ACM conference on recommender systems*, pp. 191–198.
- [41] GOMEZ-URIBE, C. A. and N. HUNT (2015) “The netflix recommender system: Algorithms, business value, and innovation,” *ACM Transactions on Management Information Systems (TMIS)*, **6**(4), pp. 1–19.
- [42] KWON, Y. and M. RHU (2022) “Training personalized recommendation systems from (GPU) scratch: look forward not backwards,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 860–873.
- [43] LUI, M., Y. YETIM, Ö. ÖZKAN, Z. ZHAO, S. TSAI, C. WU, and M. HEMPSTEAD (2021) “Understanding Capacity-Driven Scale-Out Neural Recommendation Inference,” in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*, IEEE, pp. 162–171.
URL <https://doi.org/10.1109/ISPASS51385.2021.00033>
- [44] SARMA, A., H. JIANG, A. PATTNAIK, J. KOTRA, M. T. KANDEMIR, and C. R. DAS (2019) “CASH: compiler assisted hardware design for improving DRAM energy efficiency in CNN inference,” in *Proceedings of the International Symposium on Memory Systems*, pp. 396–407.
- [45] LIU, Y., Y. WANG, R. YU, M. LI, V. SHARMA, and Y. WANG (2019) “Optimizing {CNN} Model Inference on {CPUs},” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1025–1040.
- [46] KUMAR, A., A. SIVASUBRAMANIAM, and T. ZHU (2022) “Overflowing emerging neural network inference tasks from the GPU to the CPU on heterogeneous servers,” in *Proceedings of the 15th ACM International Conference on Systems and Storage*, pp. 26–39.
- [47] HAZELWOOD, K., S. BIRD, D. BROOKS, S. CHINTALA, U. DIRIL, D. DZHULGAKOV, M. FAWZY, B. JIA, Y. JIA, A. KALRO, ET AL. (2018) “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 620–629.
- [48] PARK, J., M. NAUMOV, P. BASU, S. DENG, A. KALAIAH, D. KHUDIA, J. LAW, P. MALANI, A. MALEVICH, S. NADATHUR, ET AL. (2018) “Deep learning infer-

ence in facebook data centers: Characterization, performance optimizations and hardware implications,” *arXiv preprint arXiv:1811.09886*.

- [49] LEE, Y., S. H. SEO, H. CHOI, H. U. SUL, S. KIM, J. W. LEE, and T. J. HAM (2021) “MERCI: efficient embedding reduction on commodity hardware via sub-query memoization,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 302–313.
- [50] MUDIGERE, D., Y. HAO, J. HUANG, Z. JIA, A. TULLOCH, S. SRIDHARAN, X. LIU, M. OZDAL, J. NIE, J. PARK, ET AL. (2022) “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*, pp. 993–1011.
- [51] ZHAO, M., N. AGARWAL, A. BASANT, B. GEDIK, S. PAN, M. OZDAL, R. KOMURAVELLI, J. PAN, T. BAO, H. LU, ET AL. (2022) “Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 1042–1057.
- [52] ZHA, D., L. FENG, B. BHUSHANAM, D. CHOUDHARY, J. NIE, Y. TIAN, J. CHAE, Y. MA, A. KEJARIWAL, and X. HU (2022) “Autoshard: Automated embedding table sharding for recommender systems,” in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4461–4471.
- [53] ZHA, D., L. FENG, Q. TAN, Z. LIU, K.-H. LAI, B. BHUSHANAM, Y. TIAN, A. KEJARIWAL, and X. HU (2022) “Dreamshard: Generalizable embedding table placement for recommender systems,” *arXiv preprint arXiv:2210.02023*.
- [54] KAL, H., S. LEE, G. KO, and W. W. RO (2021) “SPACE: locality-aware processing in heterogeneous memory for personalized recommendations,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 679–691.
- [55] HWANG, R., T. KIM, Y. KWON, and M. RHU (2020) “Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 968–981.
- [56] KE, L., U. GUPTA, B. Y. CHO, D. BROOKS, V. CHANDRA, U. DIRIL, A. FIROOZSHAHIAN, K. HAZELWOOD, B. JIA, H.-H. S. LEE, ET AL. (2020) “Recnmp: Accelerating personalized recommendation with near-memory processing,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 790–803.

- [57] EISENMAN, A., M. NAUMOV, D. GARDNER, M. SMELYANSKIY, S. PUPYREV, K. HAZELWOOD, A. CIDON, and S. KATTI (2019) “Bandana: Using non-volatile memory for storing deep learning models,” *Proceedings of Machine Learning and Systems*, **1**, pp. 40–52.
- [58] WILKENING, M., U. GUPTA, S. HSIA, C. TRIPPEL, C.-J. WU, D. BROOKS, and G.-Y. WEI (2021) “RecSSD: near data processing for solid state drive based recommendation inference,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 717–729.
- [59] KWON, Y., Y. LEE, and M. RHU (2019) “Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 740–753.
- [60] RESEARCH, M. (2021), “Embedding Lookup Synthetic Dataset,” "https://github.com/facebookresearch/dlrm_datasets".
- [61] IBRAHIM, M. A., O. KAYIRAN, and S. AGA (2021) “Efficient Cache Utilization via Model-aware Data Placement for Recommendation Models,” in *The International Symposium on Memory Systems*, pp. 1–11.
- [62] AINSWORTH, S. and T. M. JONES (2017) “Software prefetching for indirect memory accesses,” in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 305–317.
- [63] JAMILAN, S., T. A. KHAN, G. AYERS, B. KASIKCI, and H. LITZ (2022) “Apt-get: Profile-guided timely software prefetching,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 747–764.
- [64] TALATI, N., K. MAY, A. BEHROOZI, Y. YANG, K. KASZYK, C. VASILADIOTIS, T. VERMA, L. LI, B. NGUYEN, J. SUN, ET AL. (2021) “Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, pp. 654–667.
- [65] BASAK, A., S. LI, X. HU, S. M. OH, X. XIE, L. ZHAO, X. JIANG, and Y. XIE (2019) “Analysis and optimization of the memory hierarchy for graph processing workloads,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 373–386.
- [66] INTEL (2022), “Hardware Prefetchers in Intel CPU,” "<https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>".
- [67] GUPTA, U., S. HSIA, J. ZHANG, M. WILKENING, J. POMBRA, H.-H. S. LEE, G.-Y. WEI, C.-J. WU, and D. BROOKS (2021) “RecPipe: Co-designing models and

- hardware to jointly optimize recommendation quality and performance,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 870–884.
- [68] ZHAO, Z., L. HONG, L. WEI, J. CHEN, A. NATH, S. ANDREWS, A. KUMTHEKAR, M. SATHIAMOORTHY, X. YI, and E. CHI (2019) “Recommending what video to watch next: a multitask ranking system,” in *Proceedings of the 13th ACM Conference on Recommender Systems*, pp. 43–51.
 - [69] WANG, R., B. FU, G. FU, and M. WANG (2017) “Deep & cross network for ad click predictions,” in *Proceedings of the ADKDD’17*, pp. 1–7.
 - [70] WANG, R., R. SHIVANNA, D. CHENG, S. JAIN, D. LIN, L. HONG, and E. CHI (2021) “Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems,” in *Proceedings of the Web Conference 2021*, pp. 1785–1797.
 - [71] DAGHAGHI, S., N. MEISBURGER, M. ZHAO, and A. SHRIVASTAVA (2021) “Accelerating slide deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more,” *Proceedings of Machine Learning and Systems*, **3**, pp. 156–166.
 - [72] GCC (2022), “GCC Data Prefetch Support,” "<https://gcc.gnu.org/projects/prefetch.html>".
 - [73] INTEL (2022), “Intel C++ Compiler Classic Developer Guide and Reference,” "<https://www.intel.com/content/www/us/en/developer/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/advanced-optimization-options/qopt-prefetch-qopt-prefetch.html>".
 - [74] TULLSEN, D. M. and J. A. BROWN (2001) “Handling long-latency loads in a simultaneous multithreading processor,” in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, IEEE, pp. 318–327.
 - [75] NAITHANI, A., S. AINSWORTH, T. M. JONES, and L. EECKHOUT (2021) “Vector runahead,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 195–208.
 - [76] HASSAN, R., A. HARRIS, N. TOPHAM, and A. EFTHYMIU (2007) “Synthetic trace-driven simulation of cache memory,” in *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW’07)*, vol. 1, IEEE, pp. 764–771.
 - [77] INTEL (2022), “Pin - A Dynamic Binary Instrumentation Tool,” "<https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html>".

- [78] ——— (2019), “Intel Cascade Lake Architecture,” ["https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html"](https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html).
- [79] ——— (2021), “Intel Ice Lake Architecture,” ["https://ark.intel.com/content/www/us/en/ark/products/codename/74979/products-formerly-ice-lake.html"](https://ark.intel.com/content/www/us/en/ark/products/codename/74979/products-formerly-ice-lake.html).
- [80] ——— (2022), “Intel Extension for PyTorch,” ["https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html"](https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html).
- [81] ——— (2022), “Intel VTune Profiler,” ["https://github.com/intel/intel-extension-for-pytorch"](https://github.com/intel/intel-extension-for-pytorch).
- [82] ——— (2023), “Intel Xeon Platinum 8480+ Processor,” ["https://ark.intel.com/content/www/us/en/ark/products/231746/intel-xeon-platinum-8480-processor-105m-cache-2-00-ghz.html"](https://ark.intel.com/content/www/us/en/ark/products/231746/intel-xeon-platinum-8480-processor-105m-cache-2-00-ghz.html).
- [83] AMD (2021), “AMD EPYC 7763,” ["https://www.amd.com/en/products/cpu/amd-epyc-7763"](https://www.amd.com/en/products/cpu/amd-epyc-7763).
- [84] MUTLU, O., J. STARK, C. WILKERSON, and Y. N. PATT (2003) “Runahead execution: An alternative to very large instruction windows for out-of-order processors,” in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, IEEE, pp. 129–140.
- [85] LEE, J., H. KIM, and R. VUDUC (2012) “When prefetching works, when it doesn’t, and why,” *ACM Transactions on Architecture and Code Optimization (TACO)*, **9**(1), pp. 1–29.
- [86] INTEL (2022), “Prefetch Intrinsic,” ["https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm_prefetch"](https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm_prefetch).
- [87] TULLSEN, D. M., S. J. EGGERS, and H. M. LEVY (1995) “Simultaneous multithreading: Maximizing on-chip parallelism,” in *Proceedings of the 22nd annual international symposium on Computer architecture*, pp. 392–403.
- [88] MARR, D. T., F. BINNS, D. L. HILL, G. HINTON, D. A. KOUFATY, J. A. MILLER, and M. UPTON (2002) “Hyper-Threading Technology Architecture and Microarchitecture.” *Intel Technology Journal*, **6**(1).
- [89] MLPERF (2022), “MLPerf Datacenter Inference Submissions v2.1,” ["https://mlcommons.org/en/inference-datacenter-21/"](https://mlcommons.org/en/inference-datacenter-21/).
- [90] CALLAHAN, D., K. KENNEDY, and A. PORTERFIELD (1991) “Software prefetching,” *ACM SIGARCH Computer Architecture News*, **19**(2), pp. 40–52.
- [91] TARAM, M., X. REN, A. VENKAT, and D. TULLSEN (2022) “{SecSMT}: Securing {SMT} Processors against {Contention-Based} Covert Channels,” in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3165–3182.

- [92] RESEARCH, M. (2022), “DLRM configuration for Criteo Kaggle Training,” "https://github.com/facebookresearch/dlrm/blob/main/bench/dlrm_s_criteo_kaggle.sh".
- [93] GUPTA, U. (2020), “DLRM configuration in DeepRecSys RMC3,” "https://github.com/harvard-acc/DeepRecSys/blob/master/models/configs/dlrm_rm3.json".
- [94] MLPERF (2022), “MLPerf benchmarking on CPUs using Intel Extension for PyTorch,” "https://github.com/mlcommons/inference_results_v2.1/tree/master/closed/Intel/code/dlrm-99.9/pytorch-cpu".
- [95] WENG, Q., W. XIAO, Y. YU, W. WANG, C. WANG, J. HE, Y. LI, L. ZHANG, W. LIN, and Y. DING (2022) “{MLaaS} in the Wild: Workload Analysis and Scheduling in {Large-Scale} Heterogeneous {GPU} Clusters,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 945–960.
- [96] INTEL (2017), “Intel Xeon Gold 6136 Processor,” "<https://www.intel.com/content/www/us/en/products/sku/120479/intel-xeon-gold-6136-processor-24-75m-cache-3-00-ghz/specifications.html>".
- [97] ——— (2021), “Intel Xeon Silver 4314 Processor,” "<https://www.intel.com/content/www/us/en/products/sku/215269/intel-xeon-silver-4314-processor-24m-cache-2-40-ghz/specifications.html>".
- [98] DEAN, J. and L. A. BARROSO (2013) “The tail at scale,” *Communications of the ACM*, **56**(2), pp. 74–80.
- [99] JAIN, R., S. CHENG, V. KALAGI, V. SANGHAVI, S. KAUL, M. ARUNACHALAM, K. MAENG, A. JOG, A. SIVASUBRAMANIAM, M. T. KANDEMIR, and C. R. DAS (2023) “Optimizing CPU Performance for Recommendation Systems At-Scale,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA '23*, Association for Computing Machinery, New York, NY, USA. URL <https://doi.org/10.1145/3579371.3589112>
- [100] GUPTA, U., C.-J. WU, X. WANG, M. NAUMOV, B. REAGEN, D. BROOKS, B. COTTEL, K. HAZELWOOD, M. HEMPSTEAD, B. JIA, H.-H. S. LEE, A. MALEVICH, D. MUDIGERE, M. SMELYANSKIY, L. XIONG, and X. ZHANG (2020) “The Architectural Implications of Facebook’s DNN-Based Personalized Recommendation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–501.

- [101] GOOGLE (2024), “Machine families resource and comparison guide | Compute Engine Documentation | Google Cloud,” ["https://cloud.google.com/compute/docs/machine-resource"](https://cloud.google.com/compute/docs/machine-resource).
- [102] AMAZON (2024), “Amazon EC2 Instance Types – AWS,” ["https://aws.amazon.com/ec2/instance-types/"](https://aws.amazon.com/ec2/instance-types/).
- [103] MICROSOFT (2024), “Cloud Computing Services | Microsoft Azure,” ["https://azure.microsoft.com/en-us/"](https://azure.microsoft.com/en-us/).
- [104] FACEBOOK (2023), “Facebook Generalized Matrix Multiplication,” ["https://github.com/pytorch/FBGEMM"](https://github.com/pytorch/FBGEMM).
- [105] AMD (2022), “AMD Epyc 9684X CPU, code-named Genoa X,” ["https://www.amd.com/en/products/cpu/amd-epyc-9684x"](https://www.amd.com/en/products/cpu/amd-epyc-9684x).
- [106] GUPTA, U., S. HSIA, J. ZHANG, M. WILKENING, J. POMBRA, H.-H. S. LEE, G.-Y. WEI, C.-J. WU, and D. BROOKS (2021) “Recipe: Co-designing models and hardware to jointly optimize recommendation quality and performance,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 870–884.
- [107] GUPTA, U., S. HSIA, V. SARAPH, X. WANG, B. REAGEN, G.-Y. WEI, H.-H. S. LEE, D. BROOKS, and C.-J. WU (2020) “Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 982–995.
- [108] IBRAHIM, M. A., O. KAYIRAN, and S. AGA (2022) “Efficient Cache Utilization via Model-Aware Data Placement for Recommendation Models,” in *Proceedings of the International Symposium on Memory Systems*, MEMSYS ’21, Association for Computing Machinery, New York, NY, USA.
URL <https://doi.org/10.1145/3488423.3519317>
- [109] WILKENING, M., U. GUPTA, S. HSIA, C. TRIPPEL, C.-J. WU, D. BROOKS, and G.-Y. WEI (2021) “RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’21, Association for Computing Machinery, New York, NY, USA, p. 717–729.
URL <https://doi.org/10.1145/3445814.3446763>
- [110] AMD (2023), “AMD Zen Deep Neural Network,” ["https://www.amd.com/en/developer/zendnn.html"](https://www.amd.com/en/developer/zendnn.html).
- [111] ARDESTANI, E. K., C. KIM, S. J. LEE, L. PAN, J. AXBOE, V. RAMPERSAD, B. AGRAWAL, F. YU, A. YU, T. LE, ET AL. (2022) “Supporting massive DLRM

- inference through software defined memory,” in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, IEEE, pp. 302–312.
- [112] EISENMAN, A., K. K. MATAM, S. INGRAM, D. MUDIGERE, R. KRISHNAMOORTHY, K. NAIR, M. SMELYANSKIY, and M. ANNAVARAM (2022) “Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, USENIX Association, Renton, WA, pp. 929–943.
URL <https://www.usenix.org/conference/nsdi22/presentation/eisenman>
 - [113] KALAMKAR, D., E. GEORGANAS, S. SRINIVASAN, J. CHEN, M. SHIRYAEV, and A. HEINECKE (2020) “Optimizing Deep Learning Recommender Systems Training on CPU Cluster Architectures,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15.
 - [114] KWON, Y. and M. RHU (2022) “Training personalized recommendation systems from (GPU) scratch: Look forward not backwards,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 860–873.
 - [115] KE, L., U. GUPTA, M. HEMPSTEAD, C.-J. WU, H.-H. S. LEE, and X. ZHANG (2022) “Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, pp. 141–154.
 - [116] SETHI, G., B. ACUN, N. AGARWAL, C. KOZYRAKIS, C. TRIPPEL, and C.-J. WU (2022) “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*, pp. 344–358.
 - [117] SUN, X., H. WAN, Q. LI, C.-L. YANG, T.-W. KUO, and C. J. XUE (2022) “RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1056–1070.
 - [118] ZHAO, M., N. AGARWAL, A. BASANT, B. GEDIK, S. PAN, M. OZDAL, R. KOMURAVELLI, J. PAN, T. BAO, H. LU, S. NARAYANAN, J. LANGMAN, K. WILFONG, H. RASTOGI, C.-J. WU, C. KOZYRAKIS, and P. POL (2022) “Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA ’22, Association for Computing Machinery, New York, NY, USA, p. 1042–1057.
URL <https://doi.org/10.1145/3470496.3533044>
 - [119] SHI, H.-J. M., D. MUDIGERE, M. NAUMOV, and J. YANG (2020) “Compositional Embeddings Using Complementary Partitions for Memory-Efficient Recommendation Systems,” in *Proceedings of the 26th ACM SIGKDD International Conference*

- on *Knowledge Discovery & Data Mining*, KDD '20, Association for Computing Machinery, New York, NY, USA, p. 165–175.
URL <https://doi.org/10.1145/3394486.3403059>
- [120] LIU, H., L. ZHENG, Y. HUANG, C. LIU, X. YE, J. YUAN, X. LIAO, H. JIN, and J. XUE (2023) “Accelerating Personalized Recommendation with Cross-Level Near-Memory Processing,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, Association for Computing Machinery, New York, NY, USA.
URL <https://doi.org/10.1145/3579371.3589101>
 - [121] KAL, H., S. LEE, G. KO, and W. W. RO (2021) “Space: locality-aware processing in heterogeneous memory for personalized recommendations,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 679–691.
 - [122] KE, L., U. GUPTA, B. Y. CHO, D. BROOKS, V. CHANDRA, U. DIRIL, A. FIROOZSHAHIAN, K. HAZELWOOD, B. JIA, H.-H. S. LEE, M. LI, B. MAHER, D. MUDIGERE, M. NAUMOV, M. SCHATZ, M. SMELYANSKIY, X. WANG, B. REAGEN, C.-J. WU, M. HEMPSTEAD, and X. ZHANG (2020) “RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 790–803.
 - [123] HUANG, Y., X. WEI, X. WANG, J. YANG, B.-Y. SU, S. BHARUKA, D. CHOUDHARY, Z. JIANG, H. ZHENG, and J. LANGMAN (2021) “Hierarchical Training: Scaling Deep Recommendation Models on Large CPU Clusters,” in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, Association for Computing Machinery, New York, NY, USA, p. 3050–3058.
URL <https://doi.org/10.1145/3447548.3467084>
 - [124] GUPTA, V., D. CHOUDHARY, P. TANG, X. WEI, X. WANG, Y. HUANG, A. KEJARIWAL, K. RAMCHANDRAN, and M. W. MAHONEY (2021) “Training Recommender Systems at Scale: Communication-Efficient Model and Data Parallelism,” in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, Association for Computing Machinery, New York, NY, USA, p. 2928–2936.
URL <https://doi.org/10.1145/3447548.3467080>
 - [125] WANG, Z., Y. WANG, B. FENG, D. MUDIGERE, B. MUTHIAH, and Y. DING (2022) “EL-Rec: Efficient Large-Scale Recommendation Model Training via Tensor-Train Embedding Table,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22, IEEE Press.
 - [126] BALASUBRAMANIAN, K., A. ALSHABANAH, J. D. CHOE, and M. ANNAVARAM (2021) “CDLRM: Look Ahead Caching for Scalable Training of Recommendation

- Models,” in *Proceedings of the 15th ACM Conference on Recommender Systems*, RecSys ’21, Association for Computing Machinery, New York, NY, USA, p. 263–272. URL <https://doi.org/10.1145/3460231.3474246>
- [127] AGARWAL, S., C. YAN, Z. ZHANG, and S. VENKATARAMAN (2023) “Bagpipe: Accelerating Deep Recommendation Model Training,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP ’23, Association for Computing Machinery, New York, NY, USA, p. 348–363. URL <https://doi.org/10.1145/3600006.3613142>
- [128] PUNNIYAMURTHY, K., B. M. BECKMANN, and K. HAMIDOUCHE (2023) “GPU-initiated Fine-grained Overlap of Collective Communication with Computation,” *CoRR*, **abs/2305.06942**, 2305.06942. URL <https://doi.org/10.48550/arXiv.2305.06942>
- [129] JIANG, J., R. TIAN, J. DU, D. HUANG, and Y. LU (2023) “MixRec: Orchestrating Concurrent Recommendation Model Training on CPU-GPU platform,” in *2023 IEEE 41st International Conference on Computer Design (ICCD)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 366–374. URL <https://doi.ieeecomputersociety.org/10.1109/ICCD58817.2023.00062>
- [130] ADNAN, M., Y. E. MABOUD, D. MAHAJAN, and P. J. NAIR (2022) “Heterogeneous Acceleration Pipeline for Recommendation System Training,” *CoRR*, **abs/2204.05436**, 2204.05436. URL <https://doi.org/10.48550/arXiv.2204.05436>
- [131] WEI, Y., M. LANGER, F. YU, M. LEE, J. LIU, J. SHI, and Z. WANG (2022) “A GPU-Specialized Inference Parameter Server for Large-Scale Deep Recommendation Models,” in *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys ’22, Association for Computing Machinery, New York, NY, USA, p. 408–419. URL <https://doi.org/10.1145/3523227.3546765>
- [132] YE, H., S. VEDULA, Y. CHEN, Y. YANG, A. BRONSTEIN, R. DRESLINSKI, T. MUDGE, and N. TALATI (2023) “GRACE: A Scalable Graph-Based Approach to Accelerating Recommendation Model Inference,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 282–301.
- [133] HILDEBRAND, M., J. LOWE-POWER, and V. AKELLA (2023) “Efficient Large Scale DLRM Implementation on Heterogeneous Memory Systems,” in *High Performance Computing: 38th International Conference, ISC High Performance 2023, Hamburg, Germany, May 21–25, 2023, Proceedings*, Springer-Verlag, Berlin, Heidelberg, p. 42–61. URL https://doi.org/10.1007/978-3-031-32041-5_3

- [134] WANG, Y., S. LI, Q. ZHENG, A. CHANG, H. LI, and Y. CHEN (2023) “EMS-i: An Efficient Memory System Design with Specialized Caching Mechanism for Recommendation Inference,” *ACM Trans. Embed. Comput. Syst.*, **22**(5s). URL <https://doi.org/10.1145/3609384>
- [135] FANG, J., G. ZHANG, J. HAN, S. LI, Z. BIAN, Y. LI, J. LIU, and Y. YOU (2022) “A Frequency-aware Software Cache for Large Recommendation System Embeddings,” *CoRR*, **abs/2208.05321**, 2208.05321. URL <https://doi.org/10.48550/arXiv.2208.05321>
- [136] JAIN, R., S. CHENG, V. KALAGI, V. SANGHAVI, S. KAUL, M. ARUNACHALAM, K. MAENG, A. JOG, A. SIVASUBRAMANIAM, M. T. KANDEMIR, and C. R. DAS (2023) “Optimizing CPU Performance for Recommendation Systems At-Scale,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA ’23*, Association for Computing Machinery, New York, NY, USA. URL <https://doi.org/10.1145/3579371.3589112>
- [137] HSIA, S., U. GUPTA, M. WILKENING, C.-J. WU, G.-Y. WEI, and D. BROOKS (2020) “Cross-stack workload characterization of deep recommendation systems,” in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, pp. 157–168.
- [138] JAIN, R., V. M. BHASI, A. JOG, A. SIVASUBRAMANIAM, M. T. KANDEMIR, and C. R. DAS (2024) “Pushing the Performance Envelope of DNN-based Recommendation Systems Inference on GPUs,” in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, pp. 1217–1232.
- [139] GUPTA, U., C.-J. WU, X. WANG, M. NAUMOV, B. REAGEN, D. BROOKS, B. COTTEL, K. HAZELWOOD, M. HEMPSTEAD, B. JIA, ET AL. (2020) “The architectural implications of facebook’s dnn-based personalized recommendation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 488–501.
- [140] NAIR, K., A.-C. PANDEY, S. KARABANNAVAR, M. ARUNACHALAM, J. KALAMATIANOS, V. AGRAWAL, S. GUPTA, A. SIRASAO, E. DELAYE, S. REINHARDT, ET AL. (2024) “Parallelization Strategies for DLRM Embedding Bag Operator on AMD CPUs,” *IEEE Micro*.
- [141] BHASI, V. M., A. SHARMA, R. JAIN, J. R. GUNASEKARAN, A. PATTNAIK, M. T. KANDEMIR, and C. DAS (2024) “Towards SLO-Compliant and Cost-Effective Serverless Computing on Emerging GPU Architectures,” in *Proceedings of the 25th International Middleware Conference*, pp. 211–224.
- [142] NAIR, K., M. ARUNACHALAM, J. KALAMATIANOS, R. JAIN, V. AGRAWAL, A. C. PANDEY, S. Y. KARABANNAVAR, A. SIRASAO, E. DELAYE, ET AL. (2025), “Systems and methods for parallelization of embedding operations,” US Patent App. 18/974,332.

- [143] INTEL (2024), “Intel Extension For Pytorch,” "<https://github.com/intel/intel-extension-for-pytorch>".
- [144] META (2024), “DLRM using PyTorch,” "<https://github.com/facebookresearch/dlrm>".
- [145] SU, Y., M. RAINEY, N. WANNINGER, N. DHIANTRAVAN, J. LIANG, U. A. ACAR, P. DINDA, and S. CAMPANONI (2024) “Compiling Loop-Based Nested Parallelism for Irregular Workloads,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 232–250.
- [146] META (2024), “PyTorch DLRM,” "https://github.com/facebookresearch/dlrm/blob/639e3d25a59b35e6b703506a5764e611cdf8bea/dlrm_s_pytorch.py#L590".
- [147] ——— (2024), “PyTorch Embedding Bag Operator,” "<https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/EmbeddingBag.cpp>".
- [148] ARDALANI, N., C.-J. WU, Z. CHEN, B. BHUSHANAM, and A. AZIZ (2022) “Understanding Scaling Laws for Recommendation Models,” *arXiv preprint arXiv:2208.08489*.
- [149] MATAM, K. K., H. RAMEZANI, F. WANG, Z. CHEN, Y. DONG, M. DING, Z. ZHAO, Z. ZHANG, E. WEN, and A. EISENMAN (2024) “{QuickUpdate}: a {Real-Time} Personalization System for {Large-Scale} Recommendation Models,” in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 731–744.
- [150] ZHONG, Y., D. S. BERGER, C. WALDSPURGER, I. AGARWAL, R. AGARWAL, F. HADY, K. KUMAR, M. D. HILL, M. CHOWDHURY, and A. CIDON (2024) “Managing Memory Tiers with CXL in Virtualized Environments,” in *Symposium on Operating Systems Design and Implementation*.
- [151] CHOI, Y., J. KIM, and M. RHU (2024) “ElasticRec: A Microservice-based Model Serving Architecture Enabling Elastic Resource Scaling for Recommendation Models,” *arXiv preprint arXiv:2406.06955*.
- [152] CHEN, S., H. TAN, A. C. ZHOU, Y. LI, and P. BALAJI (2024) “UpDLRM: Accelerating Personalized Recommendation using Real-World PIM Architecture,” *arXiv preprint arXiv:2406.13941*.
- [153] PARK, T., S. YANG, J. SEOK, H.-J. LEE, J. KIM, and C. E. RHEE (2024) “Accelerating Large-Scale DLRM Inference through Dynamic Hot Data Rearrangement,” in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, pp. 1–5.

- [154] META (2024), “Multi-threading in PyTorch,” ["https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html"](https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html).
- [155] CHIPS and CHEESE (2024), “Ryzen-9950x,” ["https://chipsandcheese.com/p/amds-ryzen-9950x-zen-5-on-desktop"](https://chipsandcheese.com/p/amds-ryzen-9950x-zen-5-on-desktop).
- [156] ZHENG, M., V. T. RAVI, W. MA, F. QIN, and G. AGRAWAL (2012) “Gmprof: A low-overhead, fine-grained profiling approach for gpu programs,” in *2012 19th International Conference on High Performance Computing*, IEEE, pp. 1–10.
- [157] ZHAO, Q., I. CUTCUTACHE, and W.-F. WONG (2010) “Pipa: Pipelined profiling and analysis on multicore systems,” *ACM Transactions on Architecture and Code Optimization (TACO)*, **7**(3), pp. 1–29.
- [158] LAI, F., W. ZHANG, R. LIU, W. TSAI, X. WEI, Y. HU, S. DEVKOTA, J. HUANG, J. PARK, X. LIU, Z. CHEN, E. WEN, P. RIVERA, J. YOU, C. CHENG JASON CHEN, and M. CHOWDHURY (2023) “{AdaEmbed}: Adaptive Embedding for {Large-Scale} Recommendation Models,” in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, USENIX Association, Boston, MA, pp. 817–831.
URL <https://www.usenix.org/conference/osdi23/presentation/lai>
- [159] JHA, G. K., A. THOMAS, N. JAIN, S. GOBRIEL, T. ROSING, and R. IYER (2024) “Mem-Rec: Memory Efficient Recommendation System using Alternative Representation,” in *Asian Conference on Machine Learning*, PMLR, pp. 518–533.
- [160] HWANG, R., T. KIM, Y. KWON, and M. RHU (2020) “Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 968–981.
- [161] KE, L., X. ZHANG, J. SO, J.-G. LEE, S.-H. KANG, S. LEE, S. HAN, Y. CHO, J. H. KIM, Y. KWON, K. KIM, J. JUNG, I. YUN, S. J. PARK, H. PARK, J. SONG, J. CHO, K. SOHN, N. S. KIM, and H.-H. S. LEE (2022) “Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM,” *IEEE Micro*, **42**(1), pp. 116–127.
- [162] KWON, Y., Y. LEE, and M. RHU (2019) “Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 740–753.
- [163] ADNAN, M., Y. E. MABOUD, D. MAHAJAN, and P. J. NAIR (2024) “Heterogeneous acceleration pipeline for recommendation system training,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 1063–1079.

- [164] SUN, X., H. WAN, Q. LI, C.-L. YANG, T.-W. KUO, and C. J. XUE (2022) “Rm-ssd: In-storage computing for large-scale recommendation inference,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, pp. 1056–1070.
- [165] JAIN, R., T. CHOU, O. KAYIRAN, J. KALAMATIANOS, G. H. LOH, M. T. KANDEMIR, and C. R. DAS (2025) “Load and MLP-Aware Thread Orchestration for Recommendation Systems Inference on CPUs,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 589–603.
- [166] NVIDIA (2024), “CUDA Programming Guide: The benefits of using GPUs,” ["https://docs.nvidia.com/cuda/cuda-c-programming-guide/#the-benefits-of-using-gpus"](https://docs.nvidia.com/cuda/cuda-c-programming-guide/#the-benefits-of-using-gpus).
- [167] KE, L., X. ZHANG, B. LEE, G. E. SUH, and H.-H. S. LEE (2022) “DisaggRec: Architecting Disaggregated Systems for Large-Scale Personalized Recommendation,” *arXiv preprint arXiv:2212.00939*.
- [168] META (2024), “Meta’s 2024 ML Infrastructure,” ["https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/"](https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/).
- [169] ——— (2024), “Embedding uBenchmark in PARAM,” ["https://github.com/facebookresearch/param/blob/0a073429d2139b5947212863b32b222a09239cd3/train/compute/pt/pytorch_emb.py#L49"](https://github.com/facebookresearch/param/blob/0a073429d2139b5947212863b32b222a09239cd3/train/compute/pt/pytorch_emb.py#L49).
- [170] NVIDIA (2024), “A100 GPU White Paper,” ["https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf"](https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf).
- [171] (2024), “Scoreboarding in GPUs,” http://gpgpu-sim.org/manual/index.php/Main_Page#Scoreboard.
- [172] NVIDIA (2024), “L2 cache residency control,” ["https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-l2-access-management"](https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-l2-access-management).
- [173] JOG, A., O. KAYIRAN, N. CHIDAMBARAM NACHIAPPAN, A. K. MISHRA, M. T. KANDEMIR, O. MUTLU, R. IYER, and C. R. DAS (2013) “OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance,” *ACM SIGPLAN Notices*, **48**(4), pp. 395–406.
- [174] KAYIRAN, O., A. JOG, M. T. KANDEMIR, and C. R. DAS (2013) “Neither more nor less: Optimizing thread-level parallelism for GPGPUs,” in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, IEEE, pp. 157–166.

- [175] SETHIA, A., D. A. JAMSHIDI, and S. MAHLKE (2015) “Mascar: Speeding up GPU warps by reducing memory pitstops,” in *2015 IEEE 21st International symposium on high performance computer architecture (HPCA)*, IEEE, pp. 174–185.
- [176] JEON, H., G. S. RAVI, N. S. KIM, and M. ANNAVARAM (2015) “GPU register file virtualization,” in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 420–432.
- [177] VOITSECHOV, D., A. ZULFIQAR, M. STEPHENSON, M. GEBHART, and S. W. KECKLER (2018) “Software-directed techniques for improved gpu register file utilization,” *ACM Transactions on Architecture and Code Optimization (TACO)*, **15**(3), pp. 1–23.
- [178] OH, Y., M. K. YOON, W. J. SONG, and W. W. RO (2018) “FineReg: Fine-grained register file management for augmenting GPU throughput,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, pp. 364–376.
- [179] JOG, A., O. KAYIRAN, A. K. MISHRA, M. T. KANDEMIR, O. MUTLU, R. IYER, and C. R. DAS (2013) “Orchestrated scheduling and prefetching for GPGPUs,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 332–343.
- [180] SETHIA, A., G. DASIKA, M. SAMADI, and S. MAHLKE (2013) “APOGEE: Adaptive prefetching on GPUs for energy efficiency,” in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, IEEE, pp. 73–82.
- [181] OH, Y., K. KIM, M. K. YOON, J. H. PARK, Y. PARK, M. ANNAVARAM, and W. W. RO (2018) “Adaptive cooperation of prefetching and warp scheduling on gpus,” *IEEE Transactions on Computers*, **68**(4), pp. 609–616.
- [182] WU, C.-J., A. JALEEL, M. MARTONOSI, S. C. STEELY JR, and J. EMER (2011) “PACMan: prefetch-aware cache management for high performance caching,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 442–453.
- [183] FU, Y., E. BOLOTIN, A. JALEEL, G. DALAL, S. MANNOR, J. SUBAG, N. KOREM, M. BEHAR, and D. NELLANS (2023) “AutoScratch: ML-Optimized Cache Management for Inference-Oriented GPUs,” *Proceedings of Machine Learning and Systems*, **5**.
- [184] ADUFU, T. and Y. KIM (2023) “L2 Cache Access Pattern Analysis using Static Profiling of an Application,” in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, IEEE, pp. 97–102.

- [185] ——— (2023) “Optimizing Performance Using GPU Cache Data Residency Based on Application’s Access Patterns,” in *2023 24th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, IEEE, pp. 42–47.
- [186] (2024), “NVIDIA A100 Tensor Core GPU.” <https://www.nvidia.com/en-us/data-center/a100/>.
- [187] (2024), “NVIDIA H100 Tensor Core GPU,” <https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/>.
- [188] LIN, Z., L. FENG, E. K. ARDESTANI, J. LEE, J. LUNDELL, C. KIM, A. KEJARIWAL, and J. D. OWENS (2022) “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)*, IEEE, pp. 48–58.
- [189] (2024), “NVIDIA Nsight Systems,” <https://docs.nvidia.com/nsight-systems/UserGuide/index.html>.
- [190] (2024), “nvcc maxrregcount flag,” <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#maxrregcount-amount-maxrregcount>.
- [191] GCC (2024), “Data Prefetching with GCC,” "<https://gcc.gnu.org/projects/prefetch.html>".
- [192] FALSAFI, B. and T. F. WENISCH (2022) *A primer on hardware prefetching*, Springer Nature.
- [193] STACKOVERFLOW (2024), “Where does Local Memory reside?” "<https://stackoverflow.com/questions/72381905/seeking-a-better-understanding-of-local-memory-in-cuda-where-does-it-live-how>".
- [194] NVIDIA (2024), “Parallel Thread Execution ISA Version 8.4,” "<https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-conversion-instructions-prefetch-prefetchu>".
- [195] EMMA, P. G., A. HARTSTEIN, T. R. PUZAK, and V. SRINIVASAN (2005) “Exploring the limits of prefetching,” *IBM Journal of Research and Development*, **49**(1), pp. 127–144.
- [196] PYTORCH (2024), “PyTorch 2.1.0,” "<https://github.com/pytorch/pytorch/tree/v2.1.0>".
- [197] NVIDIA (2024), “Nvidia’s NVCC compiler,” "<https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#opt-level-n-o>".
- [198] JAIN, R. (2023), “Homogeneous Production Traces,” "https://github.com/rishucoding/reproduce_isca23_cpu_DLRM_inference".

- [199] NVIDIA (2024), “Nvidia’s Nsight Compute Tool,” ["https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html"](https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html).
- [200] ——— (2024), “NCU profiling limitation,” ["https://forums.developer.nvidia.com/t/ncu-profiling-with-cache-control/246113/"](https://forums.developer.nvidia.com/t/ncu-profiling-with-cache-control/246113/).
- [201] ——— (2024), “H100 NVL 96 GB,” ["https://www.nvidia.com/en-us/data-center/h100/"](https://www.nvidia.com/en-us/data-center/h100/).
- [202] META (2024), “Meta H100 Infrastructure 2024,” ["https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/"](https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/).
- [203] AMAZON (2024), “Amazon EC2-P5 H100 Instance,” ["https://aws.amazon.com/blogs/aws/new-amazon-ec2-p5-instances-powered-by-nvidia-h100-tensor-core-gpus/"](https://aws.amazon.com/blogs/aws/new-amazon-ec2-p5-instances-powered-by-nvidia-h100-tensor-core-gpus/).
- [204] SONG, X., Y. ZHANG, R. CHEN, and H. CHEN (2023) “UGACHE: A Unified GPU Cache for Embedding-based Deep Learning,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 627–641.
- [205] YUAN, Y., H. YE, S. V. W. KAZA, and N. TALATI (2023) “Everest: GPU-Accelerated System For Mining Temporal Motifs,” *arXiv preprint arXiv:2310.02800*.
- [206] SHARMA, A., V. M. BHASI, S. SINGH, R. JAIN, J. R. GUNASEKARAN, S. MITRA, M. T. KANDEMIR, G. KESIDIS, and C. R. DAS (2023) “Stash: A Comprehensive Stall-Centric Characterization of Public Cloud VMs for Distributed Deep Learning,” in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pp. 1–12.
- [207] ——— (2022), “Analysis of Distributed Deep Learning in the Cloud,” 2208.14344. URL <https://arxiv.org/abs/2208.14344>
- [208] SINGH, S., A. SARMA, S. LU, A. SENGUPTA, M. T. KANDEMIR, E. NEFTCI, V. NARAYANAN, and C. R. DAS (2022) “Skipper: Enabling efficient SNN training through activation-checkpointing and time-skipping,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 565–581.
- [209] ZHAI, J., L. LIAO, X. LIU, Y. WANG, R. LI, X. CAO, L. GAO, Z. GONG, F. GU, M. HE, ET AL. (2024) “Actions speak louder than words: Trillion-parameter sequential transducers for generative recommendations,” *arXiv preprint arXiv:2402.17152*.

Vita

Rishabh Jain

Rishabh Jain was born in Ajmer, Rajasthan, India. His early exposure to computers began when his father brought home a personal computer when Rishabh was only four years old. He was fascinated by how computers could perform so many tasks and how printers could bring anything created on the screen to life, which left a strong impression on him and nurtured his early interest in technology.

He completed his schooling at St. Stephen's Senior Secondary School, Ajmer, and earned his Bachelor of Engineering degree in Electrical and Electronics Engineering from Birla Institute of Technology and Science (BITS) Pilani Goa Campus in December 2018, graduating ahead of schedule. In February 2019, he joined the Indian Institute of Technology (IIT) Madras as a Project Associate, where he worked under the supervision of Prof. V. Kamakoti, Dr. Neel Gala, and Dr. Nitya Ranganathan on topics related to network-on-chip design, data prefetching, and performance monitoring counters for RISC-V based 'Shakti Processors'. His experience at IIT Madras strengthened his interest in computer architecture and motivated him to pursue doctoral research.

Rishabh began his Ph.D. program in Computer Science and Engineering at The Pennsylvania State University advised by Prof. Chita Das and Prof. Mahmut Kandemir. He started his first semester remotely from India and continued on campus in the United States beginning January 2021. During his doctoral studies, he completed three research internships with AMD Research and published multiple first-author papers in leading computer architecture and systems conferences, including ISCA, MICRO, and ASPLOS. His research also took him to various parts of the United States, the Netherlands, Belgium, and Hong Kong for conferences.

Upon completion of his Ph.D. in July 2025, Rishabh joined NVIDIA Corporation as a Deep Learning Performance Architect in August 2025. His research interests lie in GPU architecture, performance modeling, and hardware–software co-design for large-scale deep learning systems.