Optimizing CPU Performance for Recommendation Systems At-Scale

Rishabh Jain
The Pennsylvania State University
University Park, PA, USA
rishabh@psu.edu

Scott Cheng
The Pennsylvania State University
University Park, PA, USA
ypc5394@psu.edu

Vishwas Kalagi
The Pennsylvania State University
University Park, PA, USA
vqk5180@psu.edu

Vrushabh Sanghavi
Intel
Portland, Oregon, USA
vrushabh.h.sanghavi@intel.com

Samvit Kaul
Intel
Folsom, CA, USA
samvit.kaul@intel.com

Meena Arunachalam
Intel
Portland, Oregon, USA
meena.arunachalam@intel.com

Kiwan Maeng
The Pennsylvania State University
University Park, PA, USA
kvm6242@psu.edu

Adwait Jog[*]
William & Mary /
University of Virginia
Virginia, USA
ajog@virginia.edu

Anand Sivasubramaniam
The Pennsylvania State University
University Park, PA, USA
axs53@psu.edu

Mahmut T. Kandemir
The Pennsylvania State University
University Park, PA, USA
mtk2@psu.edu

Chita R. Das
The Pennsylvania State University
University Park, PA, USA
cxd12@psu.edu

ABSTRACT

Deep Learning Recommendation Models (DLRMs) are very popular in personalized recommendation systems and are a major contributor to the data-center AI cycles. Due to the high computational and memory bandwidth needs of DLRMs, specifically the embedding stage in DLRM inferences, both CPUs and GPUs are used for hosting such workloads. This is primarily because of the heavy irregular memory accesses in the embedding stage of computation that leads to significant stalls in the CPU pipeline. As the model and parameter sizes keep increasing with newer recommendation models, the computational dominance of the embedding stage also grows, thereby, bringing into question the suitability of CPUs for inference. In this paper, we first quantify the cause of irregular accesses and their impact on caches and observe that off-chip memory access is the main contributor to high latency. Therefore, we exploit two well-known techniques: (1) Software prefetching, to hide the memory access latency suffered by the demand loads and (2) Overlapping computation and memory accesses, to reduce CPU stalls via hyperthreading to minimize the overall execution time. We evaluate our work on a single-core and 24-core configuration with the latest recommendation models and recently released production traces. Our integrated techniques speed up the inference by up to 1.59x, and on average by 1.4x.

[*]Work done while was with William & Mary. Currently with the University of Virginia.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Computing methodologies → Modeling methodologies; Machine learning**.

KEYWORDS

Recommendation Systems, Embeddings, Irregular memory accesses, Reuse distance, Prefetching, Hyperthreading, CPU

ACM Reference Format:
Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut T. Kandemir, and Chita R. Das. 2023. Optimizing CPU Performance for Recommendation Systems At-Scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589112

1 INTRODUCTION

Recommendation systems have made a profound impact in numerous domains such as e-commerce [64, 73, 75, 76], entertainment [9, 10, 15], and social networks [54], 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 [54]. With time, based on continuous usage, the corpus of features and data continue to

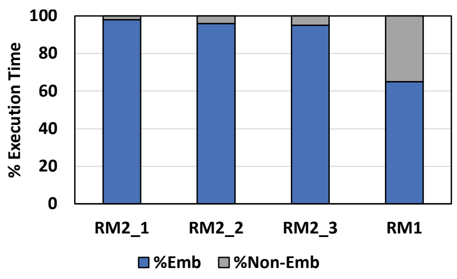


Figure 1: Execution time breakdown of different DLRMs.

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 [17, 19]. Due to the high computational and memory bandwidth needs of DLRMs [1], 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 [43]. On the other hand, CPU has been a popular choice for inference [17, 19, 40, 47]. Unlike training, inference cannot use a very large batch size due to its tight latency requirement [17, 40], thus, making the GPU less effective and thereby, positioning the CPU as an appealing economical alternative [21, 41, 45, 46, 55, 58]. As the growth of the memory requirement has been much higher than the growth of the computational demands of the MLP layer [47], this trend is expected to hold in the new generations of DLRM as well. Using recently released recommendation models/data [51, 56, 70–72], 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 1 shows that the embedding stage constitutes a significant amount of execution time for different recommendation models (detailed later in Section 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 [23, 37] and/or using Processing-In-Memory/Storage mechanisms [13, 39, 42, 68]. 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 [51, 54, 70, 71] and production traces from Meta [56], 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.

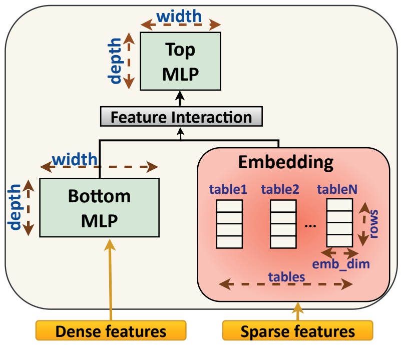


Figure 2: A generic DLRM architecture.

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.

2.1 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 [6, 7, 12, 38, 40, 59, 69]. Some common SLA targets (as seen in [17]) are shown in Table 1.

Figure 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) [17, 54].

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

Table 1: Different model characteristics from [17].

In general, embedding stages are "memory-intensive", while others are "compute-intensive" [17, 22, 40]. 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, [17, 19] 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 1, Table 2).

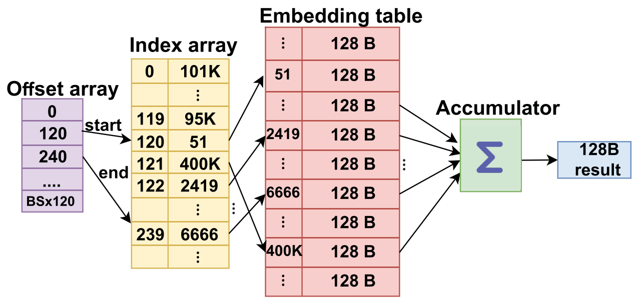


Figure 3: Illustration on the working of the embedding_bag operator for one sample (assume BS means batch size, embedding_dim = 128, and lookups per sample as 120). Three levels of indirection are involved.

2.2 Memory Accesses in the Embedding Stage

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

```
for i in 0 ... num_batches do
    for j in 0 ... num_tables do
        for k in 0 ... batch_size do
            for l in 0 ... lookup_per_sample do
                for m in 0 ... embedding_dim do
                    vec.ld accm;
                    vec.ld row_block;
                    vec.add accm, accm, row_block;
                    vec.st accm;
```

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

```
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]
        add_ker(output_ptr, row_ptr, ddim);
```

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 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 [54]. Algorithm 2 shows the core memory access loop in the embedding_bag operator. To better understand the memory access behavior, Figure 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 3), which are carried out in loading the embedding row vector, can result in highly irregular memory accesses.*

2.3 Related Work

Comparison with Prior Inference Optimizations. [24, 37, 39, 45, 68] 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 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. [2, 36] 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, [36] lacks control over the prefetch amount, which we found to be an important knob for optimization (Section 4.2). [5, 60] 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-CPUs [29] employ simple hardware prefetchers due to area/power constraints and target only next-line/stride accesses which are insufficient for irregular accesses (Section 4.2).

Multithreading and Hyperthreading. [19] showed that naive adoption of hyperthreading for DLRM can hurt the tail latency. Many other previous works [17–19, 40] 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 [9, 54, 65, 66, 74–76]. 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.

2.4 Our Novelty

We do not claim novelty in running DLRMs on CPUs. As several prior studies note [11, 17, 19, 21, 40, 41, 45–47, 55, 58], 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 3.1.1). This in turn can make some previously proposed techniques([23, 24, 37, 39, 45, 68]) 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 hardware or software prefetcher [2, 5, 14, 29, 30, 36, 60] does not work very well (Section 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 [19] 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 [53, 62], thereby allowing sibling thread to avail more resources.

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

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: Model architecture parameters in our study.

in-depth analysis of how the performance of the embedding stage is impacted by the newer datasets [56] and larger models[51, 70–72] as well as the newer CPU platforms used for execution. Though previous works [17, 19, 24, 37, 39, 68] 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 scalability and bandwidth study and share the key architectural insights.

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 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 [56] (more details in 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 4(a) compares the batch latency across datasets, and Figure 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 4, we can say that the wide gap between one item and other datasets is because of the irregular memory accesses.

3.1.1 *Access count study for different input loads.*

Earlier works [37, 39, 43] 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 5 shows the hot embedding behavior in the three production datasets. As depicted in Figure 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.

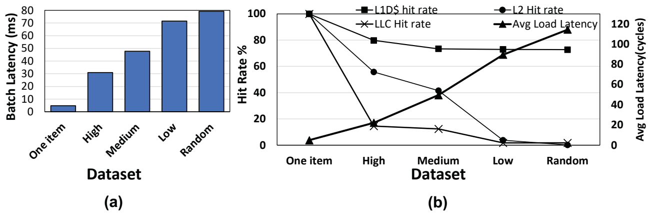
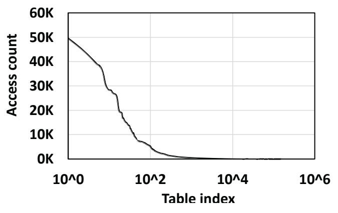
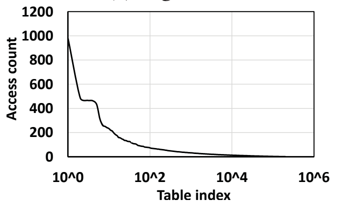


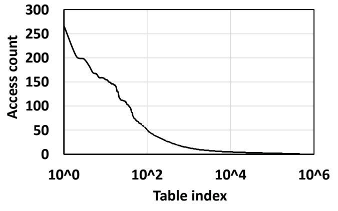
Figure 4: RM2_1 embedding stage performance comparison across datasets. (a) Batch latency (b) average load latency, and cache hit rates for L1D, L2, L3.



(a) High Hot



(b) Medium Hot



(c) Low Hot

Figure 5: Hot embedding access count (sorted) in 3 datasets.

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 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 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 [20] 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 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.

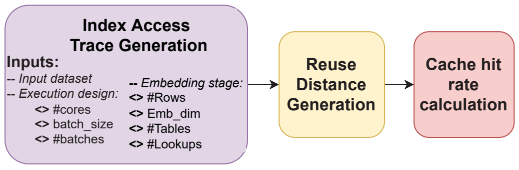


Figure 6: Our model for capturing the reuse distances, and for indicating cache hit rates and cold misses.

(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.

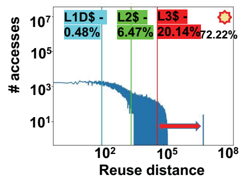
(3) *Inter-batch:* As each batch accesses all the embedding tables, we can think about the presence of locality across batches. For a table, this would happen when the same indices are accessed across batches. However, in this situation, the reuse distance is very large as many unique accesses happen in between corresponding to one less of the total tables. The thick red arrow in Figure 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.

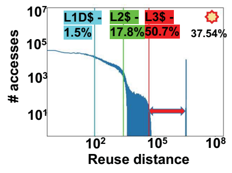
Discussion on trace collection process: It is important to highlight how our trace generation compares with collecting traces from a real execution. In a real run, one could either modify the backend codebase to dump the load addresses corresponding to embedding vector accesses or instrument the memory accesses using a tool like PinTool [33]. 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.

3.2 Multi-core Scaling and Bandwidth study

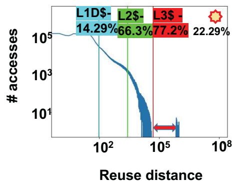
Previous works [17, 19, 23, 37, 39] have highlighted limitations of memory bandwidth in CPUs. However, modern CPUs have significantly higher memory bandwidth Cascade Lake [26], Ice Lake [27]. Also, Intel has heavily tuned the PyTorch operators for optimizing the performance on CPUs [31]. To improve throughput during inference, we propose to utilize all CPU cores, such that a batch is mapped to each core. Figure 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 [32], the execution time



(a) Low Hot



(b) Medium Hot



(c) High Hot

Figure 7: Reuse distance study for different datasets in rm2_1 model for 24 cores & batch size = 64.

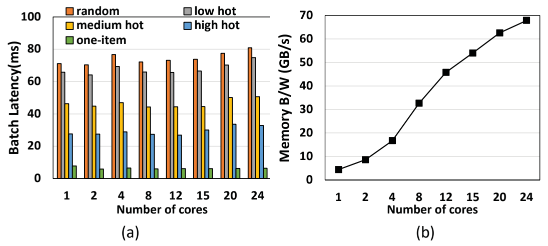


Figure 8: Multi-core scalability: (a) Execution time (b) Memory bandwidth used.

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.

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 on lookup patterns (up to 16 times), as shown in Figure 4.
(3) Reuse distance calculations are representative of the memory working sets. For large DLRMs, caches (even as large as 768MB[4]) 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 [3, 27, 35] has significantly increased compared to CPUs evaluated in past works [17, 19]. 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.

4 PROPOSED OPTIMIZATIONS FOR IMPROVING THE EFFICIENCY OF CPU PIPELINE

Since there are three levels of indirection in memory accesses (Figure 3), and the reuse distance easily exceeds the cache capacities (Figure 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.

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 [52]. 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 4).

Intel CPUs are typically equipped with a total of four prefetchers [29] 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 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 [14, 30], 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 10(a) with respect to the baseline results (hardware prefetcher on). As the results indicate, these off-the-self techniques[2] 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.

[1]By default, all the hardware prefetchers are enabled.
[2]gcc is used with "-fprefetch-loop-arrays", and icc is used with "-qopt-prefetch=5"

4.2 Application-Specific Software Prefetching

Plainly employing state-of-the-art software or hardware prefetchers [2, 5, 14, 29, 30, 36, 60] could be naïve and costly [44]. 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 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 1 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 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" [34] 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 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 4×512B = 2KB amount of prefetch injections, which is reasonably low compared to the L1D$ cache capacity. Figure 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 3 captures the above points of discussion and illustrates the prefetch implementation.

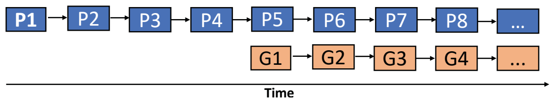


Figure 9: Software prefetching timeline: Gather[G] (the demand loads) follow the look-ahead loads[P] at an ideal prefetch distance of four.

To better visualize software prefetching, we can look into the timeline of look-ahead loads and demand loads, shown in Figure 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 3 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);
```

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 [19] used hyperthreading to run two DLRM inference instances on the same core and showed it to be detrimental to performance. Many other previous works [17–19, 40] 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 [63] like Hyper-Threading [48] in Intel CPUs.

Figure 11 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 [17, 50, 54] where Embedding is followed by Bottom-MLP. DP-HT implements mapping two inference instances on the hyper-threads. 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

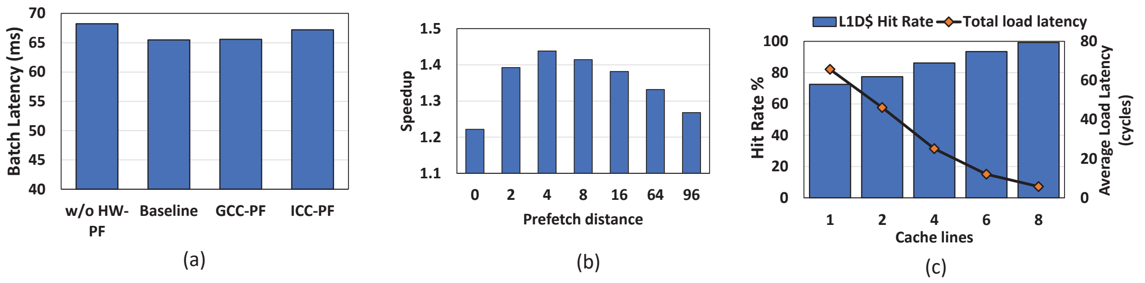


Figure 10: (a) Comparing state-of-the-art compiler-inserted prefetching with baseline for the RM2_1 model, (b) Impact of prefetch distance on the performance of 24 cores, (c) Impact of prefetch amount on the L1D$ hit rate and average load latency.

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 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.

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 [8]. 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 [61–63]. 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 [24], 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.

5 METHODOLOGY

Datasets: We use the recently released production traces from Meta [56], 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

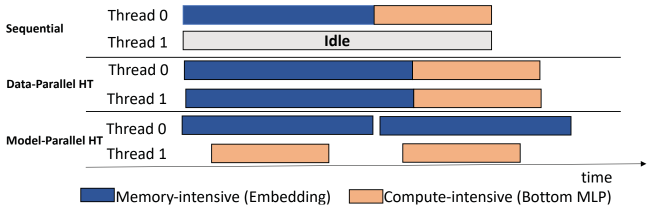


Figure 11: Overlapping compute and memory access via hyperthreading.

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 3: CPU configuration parameters.

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 [13, 19].

Models: As highlighted in Section 2, [19] and [17] 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 [17] 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 [17] may not be accurate. Therefore, we use the guidelines from [19] and [17] on model configurations to project practical models adhering to the upper bounds mentioned in the recent works [51, 70–72]. 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 [16, 18, 57]. The adopted model configuration details are given in Table 2. The batch size is chosen as 64 to maximize throughput while meeting the SLA requirements associated with each class as mentioned in [17] (Table 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) [26] is used in the evaluation. Table 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) [31], which provides optimized operators and is used in MLPerf benchmarking [49, 50]. Our schemes are built on top of IPEX. The batch size used is 64 which meets the SLA latency target shared in [17].

6 PERFORMANCE EVALUATION

In this section, we analyze the performance benefits of our optimization designs. Our experimentation configuration is described in Section 5. We report the inference performance for single and multi-core, in terms of batch latency (averaged over 120 batches) and speedup. Similar to [17, 22], 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, we consider three datasets {High, Medium, Low} and three models RM2_1, RM2_2, RM2_3 for our evaluation.

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 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 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 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.

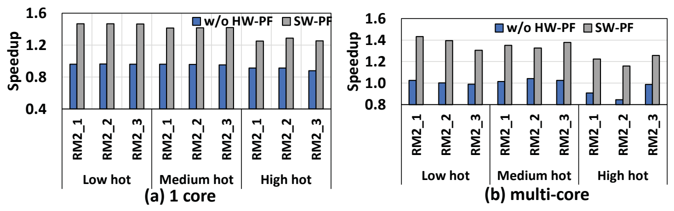


Figure 12: Comparison of performance of embedding-only stage for embedding heavy models for different datasets: (a) single and (b) many cores.

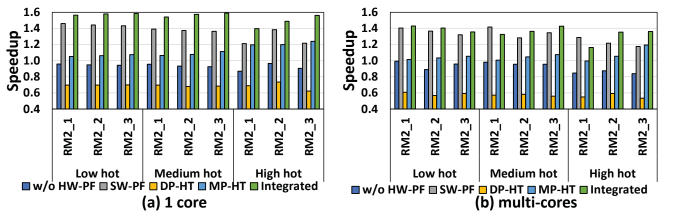


Figure 13: End-to-end performance gain on embedding heavy models for different datasets.: (a) single and (b) many cores.

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).

6.2.1 Embedding Heavy Models. Similar to Figure 12, Figure 13 compares the performance of all design points for different datasets, models, and cores. First, we observe that turning off hardware 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 4 details the actual execution time for embedding-only multi-core case.

6.2.2 Mixed Models. Figure 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

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

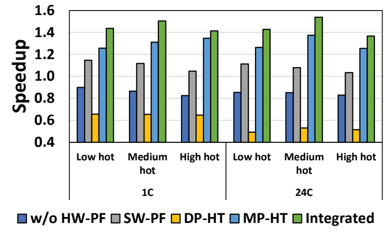


Figure 14: End-to-end performance gain on mix-model RM_1 for different datasets.

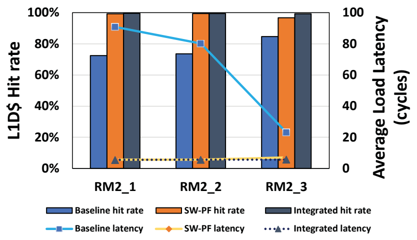


Figure 15: L1D$ hit rate and average load latency for RM2_1 model (embedding-only) on Low-Hot dataset.

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 4.4.

6.3 Empirical Characterization

Using VTune, we profile the embedding stage to obtain low-level CPU metrics. Figure 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 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 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).

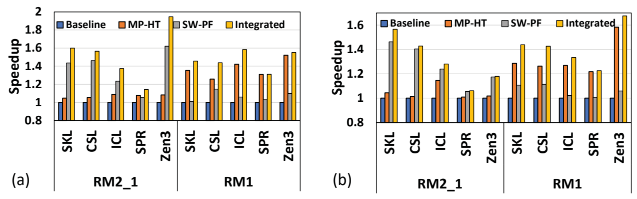


Figure 16: Speedup comparison across different CPUs for (a) single-core (b) multi-core

6.4 Evaluating Across Different CPU Platforms

Data centers contain a range of CPU platforms [17, 19, 21, 40, 67]. To understand how our optimizations perform across different CPU architectures, we consider 5 different CPUs: Intel's SkyLake(SKL, 24 cores)[25], Cascade Lake(CSL, 48 cores)[26], IceLake(ICL, 32 cores)[28], Sapphire Rapids (SPR, 56 cores)[35], and AMD's EPYC 7763(Zen3, 64 cores)[3]. To evaluate both embedding heavy and mixed-model situations, RM2_1 and RM1 are tested on the low hot dataset. Figure 16 shows the performance of our proposed optimizations across different CPU platforms, for single-core (Fig. 16(a)) and for multi-core (using all CPU cores, Fig. 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 (B) 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 effective, and the integrated scheme reaps the best performance.

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 [12, 17, 40]. Similar to [17], 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 17 plots the p95 tail latency for various schemes. Arrival times are carefully chosen to consider both SLA-compliant and saturation regions.

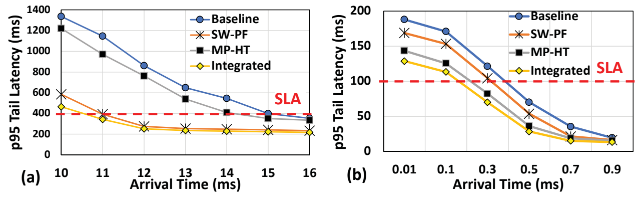


Figure 17: Tail (P95) latency comparison across designs by varying the arrival time for models: (a) RM2_1 (b) RM1. SLA latency target is marked in red.

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.

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 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. 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.

ACKNOWLEDGEMENTS

We would offer thanks to anonymous reviewers for their detailed feedback to improve the paper (and artifact) in various insightful ways. This research was partially funded by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and NSF grants {#1763681, #1931531, #2028929, #2116962, #2211018, #1912495, #1909004, #1714389}. We also thank the NSF Chameleon Cloud projects CH-819640 and CHI-220948 for their generous compute grant. All product names used here are for identification purposes only and may be trademarks of their respective companies. Lastly, we would like to thank Cyan Subhra Mishra, Siddhartha Balakrishna Rai, Anup Sarma, Sonali Singh, and Dr. Sampson for their intellectual discussions. Also, special thanks to HPCL members.

REFERENCES

[1] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2021. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 802–814.

[2] Sam Ainsworth and Timothy M. Jones. 2017. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 305–317. https://doi.org/10.1109/CGO.2017.7863749

[3] AMD. 2021. AMD EPYC 7763. "https://www.amd.com/en/products/cpu/amd-epyc-7763".

[4] AMD. 2022. AMD Zen3 3D V-Cache. "https://www.amd.com/en/press-releases/2022-03-21-3rd-gen-amd-epyc-processors-amd-3d-v-cache-technology-deliver-outstanding".

[5] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan 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, 373–386.

[6] Vivek M Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. 2022. Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing*. 257–272.

[7] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*. 153–167.

[8] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software prefetching. *ACM SIGARCH Computer Architecture News* 19, 2 (1991), 40–52.

[9] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[10] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[11] Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, and Anshumali Shrivastava. 2021. Accelerating slide deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more. *Proceedings of Machine Learning and Systems* 3 (2021), 156–166.

[12] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[13] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using non-volatile memory for storing deep learning models. *Proceedings of Machine Learning and Systems* 1 (2019), 40–52.

[14] GCC. 2022. GCC Data Prefetch Support. "https://gcc.gnu.org/projects/prefetch.html".

[15] Carlos A Gomez-Uribe and Neil Hunt. 2015. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)* 6, 4 (2015), 1–19.

[16] Udit Gupta. 2020. DLRM configuration in DeepRecSys RMC3. "https://github.com/harvard-acc/DeepRecSys/blob/master/models/configs/dlrm_rm3.json".

[17] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean 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, 982–995.

[18] Udit Gupta, Samuel Hsia, Jeff Zhang, Mark Wilkening, Javin Pombra, Hsien-Hsin Sean Lee, Gu-Yeon Wei, Carole-Jean Wu, and David 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*. 870–884.

[19] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill 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, 488–501.

[20] Rahman Hassan, Antony Harris, Nigel Topham, and Aris Efthymiou. 2007. Synthetic trace-driven simulation of cache memory. In *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*,

Vol. 1. IEEE, 764–771.

[21] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya 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, 620–629.

[22] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. 2020. Cross-stack workload characterization of deep recommendation systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 157–168.

[23] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo 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, 968–981.

[24] Mohamed Assem Ibrahim, Onur Kayiran, and Shaizeen Aga. 2021. Efficient Cache Utilization via Model-aware Data Placement for Recommendation Models. In *The International Symposium on Memory Systems*. 1–11.

[25] 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".

[26] Intel. 2019. Intel Cascade Lake Architecture. "https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html".

[27] Intel. 2021. Intel Ice Lake Architecture. "https://ark.intel.com/content/www/us/en/ark/products/codename/74979/products-formerly-ice-lake.html".

[28] Intel. 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".

[29] Intel. 2022. Hardware Prefetchers in Intel CPU. "https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html".

[30] Intel. 2022. Intel C++ Compiler Classic Developer Guide and Reference. "https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/advanced-optimization-options/qopt-prefetch-qopt-prefetch.html".

[31] Intel. 2022. Intel Extension for PyTorch. "https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html".

[32] Intel. 2022. Intel VTune Profiler. "https://github.com/intel/intel-extension-for-pytorch".

[33] 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".

[34] Intel. 2022. Prefetch Intrinsic. "https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm_prefetch".

[35] Intel. 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".

[36] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 747–764.

[37] Hongju Kal, Seokmin Lee, Gun Ko, and Won Woo 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, 679–691.

[38] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[39] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin 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, 790–803.

[40] Liu Ke, Udit Gupta, Mark Hempsteadis, Carole-Jean Wu, Hsien-Hsin S Lee, and Xuan Zhang. 2022. Hercules: Heterogeneity-Aware Inference Serving for At-Scale Personalized Recommendation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 141–144.

[41] Adithya Kumar, Anand Sivasubramaniam, and Timothy 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*. 26–39.

[42] Youngeun Kwon, Yunjae Lee, and Minsoo 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*. 740–753.

[43] Youngeun Kwon and Minsoo Rhu. 2022. Training personalized recommendation systems from (GPU) scratch: look forward not backwards. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 860–873.

[44] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 1 (2012), 1–29.

[45] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W Lee, and Tae Jun 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*. 302–313.

[46] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} Model Inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1025–1040.

[47] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark 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, 162–171. https://doi.org/10.1109/ISPASS51385.2021.00033

[48] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. 2002. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* 6, 1 (2002).

[49] 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".

[50] MLPerf. 2022. MLPerf Datacenter Inference Submissions v2.1. "https://mlcommons.org/en/inference-datacenter-21/".

[51] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo 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*. 993–1011.

[52] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale 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, 129–140.

[53] Ajeya Naithani, Sam Ainsworth, Timothy M Jones, and Lieven Eeckhout. 2021. Vector runahead. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 195–208.

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

[55] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886* (2018).

[56] Meta Research. 2021. Embedding Lookup Synthetic Dataset. "https://github.com/facebookresearch/dlrm_datasets".

[57] Meta Research. 2022. DLRM configuration for Criteo Kaggle Training. "https://github.com/facebookresearch/dlrm/blob/main/bench/dlrm_s_criteo_kaggle.sh".

[58] Anup Sarma, Huaipan Jiang, Ashutosh Pattnaik, Jagadish Kotra, Mahmut Taylan Kandemir, and Chita 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*. 396–407.

[59] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*. 138–152.

[60] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen 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, 654–667.

[61] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. {SecSMT}: Securing {SMT} Processors against {Contention-Based} Covert Channels. In *31st USENIX Security Symposium (USENIX Security 22)*. 3165–3182.

[62] Dean M Tullsen and Jeffery A Brown. 2001. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 318–327.

[63] Dean M Tullsen, Susan J Eggers, and Henry M Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*. 392–403.

[64] Menghan Wang, Yujie Lin, Guli Lin, Keping Yang, and Xiao-ming 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*. 2349–2358.

[65] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*. 1–7.

[66] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed 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*. 1785–1797.

[67] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu 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)*. 945–960.

[68] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon 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*. 717–729.

[69] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 331–344. https://doi.org/10.1109/HPCA.2019.00048

[70] Daochen Zha, Louis Feng, Bhargav Bhushanam, Dhruv Choudhary, Jade Nie, Yuandong Tian, Jay Chae, Yinbin Ma, Arun Kejariwal, and Xia Hu. 2022. Autoshard: Automated embedding table sharding for recommender systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4461–4471.

[71] Daochen Zha, Louis Feng, Qiaoyu Tan, Zirui Liu, Kwei-Herng Lai, Bhargav Bhushanam, Yuandong Tian, Arun Kejariwal, and Xia Hu. 2022. Dreamshard: Generalizable embedding table placement for recommender systems. *arXiv preprint arXiv:2210.02023* (2022).

[72] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei 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*. 1042–1057.

[73] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 319–328.

[74] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. 2019. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 43–51.

[75] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 5941–5948.

[76] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1059–1068.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact provides the codebase for the proposed prefetching, and hyperthreading designs. We share the steps to duplicate the figures shown in the results section. The artifact covers the complete steps to setup the DLRM inference pipeline, and replicate major figures.

A.2 Artifact check-list (meta-information)

- **Algorithm:** DLRM inference pipeline
- **Program:** DLRM implementation in Intel's Extension for PyTorch (IPEX), Intel's VTune
- **Compilation:** gcc 9.4.0
- **Model:** DLRM variants mentioned in Gupta et al – Table II.
- **Data set:** https://github.com/facebookresearch/dlrm_datasets
- **Run-time environment:** Ubuntu 20.04.4 LTS
- **Hardware:** Intel Cascade Lake 6240R CPU
- **Metrics:** Batch Latency (ms), speedup, p95 Tail Latency(ms), cache hit rate(%), average load latency(CPU cycles)
- **Output:** Batch execution time, p95 Tail Latency
- **Experiments:** Steps outlined in the artifact.
- **How much disk space required (approximately)?:** 65GB
- **How much time is needed to prepare workflow (approximately)?:** 2-3 hours
- **How much time is needed to complete experiments (approximately)?:** Under 1 week
- **Publicly available?:** Yes

A.3 Description

A.3.1 How to access. The codebase is available on Zenodo and can be accessed at: https://doi.org/10.5281/zenodo.7957909. Also, it is uploaded on GitHub at https://github.com/rishucoding/reproduce_isca23_cpu_DLRM_inference/

A.3.2 Hardware dependencies. To replicate the major figures, we suggest using the Intel Cascade Lake 6240R CPU.

A.3.3 Software dependencies. Anaconda, python pip, relevant conda and pip packages, PyTorch, IPEX, Intel VTune, itt-python, Intel Model Zoo, numactl, intel msr-tools. Steps to setup the dependencies is outlined in the artifact.

A.3.4 Data sets. We have used the DLRM traces recently released from Meta (https://github.com/facebookresearch/dlrm_datasets). Our artifact contains the exact dataset files used for evaluation.

A.3.5 Models. We have used DLRM models suggested by Gupta et al. The exact model configurations are shown in Table II.

A.4 Installation

The detailed installation steps are mentioned in the artifact, and the following is a high-level summary of the steps:

(1) Install Anaconda
(2) conda create –name dlrm_cpu python=3.9 ipython; conda activate dlrm_cpu
(3) Install conda and pip packages
(4) Build pytorch from source
(5) Install Intel VTune and ensure that the analysis for memory, architecture, and hardware events is working.
(6) Install itt-python
(7) Build Intel's IPEX and apply the patch.
(8) Setup the DLRM inference using Intel's ModelZoo and apply patches.
(9) Run DLRM inference.

A.5 Experiment workflow

We suggest first to setup the DLRM inference pipeline by following README.md, and then following the steps shared in replicate_figures.md. These files are available in the shared artifact.

A.6 Evaluation and expected results

To obtain the results shown in Figures 12, 13, and 14, one needs to enable prefetching or hyperthreading. To enable prefetching: one needs to alter the EmbeddingBagKrnl.cpp file in IPEX and rebuild the IPEX. To enable hyperthreading: one needs to alter the thread_pool.{cpp,h}, and throughput_benchmark.{cpp,h} in pytorch and rebuild the pytorch. To run the Integrated scheme: one needs to enable both prefetching and hyperthreading. The detailed steps are shown in the artifact.

For a particular dataset and model, one needs to collect the execution time numbers for different schemes and then find the speedups. The baseline scheme is default with hardware prefetcher ON. To disable hardware prefetchers, one can use the intel msr-tools as described in the artifact.

To obtain Figure 15, one can use VTune, and follow the steps mentioned in the replicate_figures.md file.

A.7 Experiment customization

The models or dataset can be customized by changing the parameters passed in the inference launch script.