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

Rishabh Jain
Computer Science and Engineering
The Pennsylvania State University
University Park, PA, USA
rishabh@psu.edu

Vivek M. Bhasi
Computer Science and Engineering
The Pennsylvania State University
University Park, PA, USA
vmbhasi@psu.edu

Adwait Jog
Computer Science
University of Virginia
Charlottesville, VA, USA
ajog@virginia.edu

Anand Sivasubramaniam
Computer Science and Engineering
The Pennsylvania State University
University Park, PA, USA
axs53@psu.edu

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

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

Abstract—Personalized recommendation is a ubiquitous application on the internet, with many industries and hyperscalers extensively leveraging Deep Learning Recommendation Models (DLRMs) for their personalization needs (like ad serving or movie suggestions). With growing model and dataset sizes pushing computation and memory requirements, GPUs are being increasingly preferred for executing DLRM inference. However, serving newer DLRMs, while meeting acceptable latencies, continues to remain challenging, making traditional deployments increasingly more GPU-hungry, resulting in higher inference serving costs. In this paper, we show that the embedding stage continues to be the primary bottleneck in the GPU inference pipeline, leading up to a $3.2\times$ embedding-only performance slowdown.

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 by up to 53%. 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. Experimental evaluations using A100 GPUs with large models and datasets show that our proposed techniques improve performance by up to 103% for the embedding stage, and up to 77% for the overall DLRM inference pipeline.

Index Terms—Recommendation Systems, Multi-threading, Warp-Level-Parallelism, Embeddings, memory-latency bound, Long latency load stalls, Prefetching, Cache residency control

I. INTRODUCTION

Recommendation Systems are the driving force for many internet applications such as social networks [1]–[3], entertainment [4]–[6], and e-commerce [4], [7], [8]. Modern recommendation systems provide personalized suggestions to enhance user experience through Deep Learning Recommendation Models (DLRM) [9]. 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
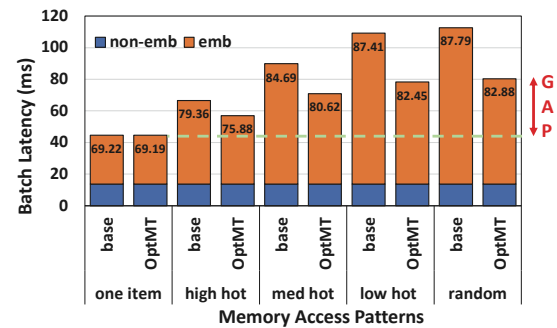

Fig. 1: Shown is the degradation in inference performance as hotness lowers (working footprint decreases) from left to right. The numbers inside the bars indicate the embedding stage contributions. Here, OptMT provides higher WLP which enhances performance over off-the-shelf PyTorch (base). Yet, a significant gap continues to exist compared to the fastest loads (one item case). We cite this as the research *gap*.

DLRMs [10], while being deployed on a variety of platforms including CPUs([10]–[14]), GPUs([13], [15]–[18]), and accelerators([18]–[24]). With the ever-increasing compute and memory requirements of DLRMs, it is increasingly being preferred to execute them on GPUs [13], [18] 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 [10]–[12], [15], [17] have shown that the embedding stage is memory intensive (due to frequent and irregular memory accesses) and the non-embedding stages are compute intensive.

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 1. GPUs are well-known for their Multi-Threading (MT) or Warp-Level-Parallelism (WLP)[1] support to hide memory latencies with computation [25]. 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 III-C).

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 it did improve the performance (reduction in batch latency) by 53%, as seen in Figure 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 IV), 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 [15]–[17], [26], distributed inference strategies [27], [28], accelerator designs [18]–[24], and algorithmic-system designs [29]–[32]. 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 [33] 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.

II. 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.

A. DLRM Inference Using GPUs

Many industries use GPUs to execute DLRM inference [13], [15], [16], [18]. 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 [34], [35]. Regardless of the number of GPUs

---

[1]We use MT and WLP interchangeably in the paper.

[2]Note that due to register spilling optimal MT may not be the maximum WLP supported by the GPU. We quantify this effect in Figure 6.
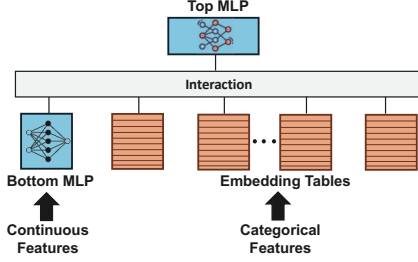
Fig. 2: A schematic of a DLRM architecture. The continuous features (e.g., age, location) are processed by Bottom MLP, and categorical features (e.g., movie genre, item ID) by the Embedding Stage. Their outputs are combined in the Feature Interaction Stage, and then fed into the Top MLP, which predicts top-k items with highest Click Through Rate (CTR).

used, each GPU executes one or more embedding tables serially [36], [37].

Figure 2 shows a simplified diagram of a typical DLRM [9], [10]. 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 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 ([11], [12], [19], [22], [29]) and non-embedding stages to be compute intensive [15].

B. Key Properties of GPU Microarchitecture

GPUs, also known as throughput processing engines, contain a hierarchical array of compute cores (CUDA cores). Figure 3 shows a simplified diagram of the GPU organization. Modern GPUs (Nvidia based) contain 100s of SMs (Streaming Multiprocessors) [38], [39], 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 [40] 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 a region for persisting accesses [41]. Section IV-C 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
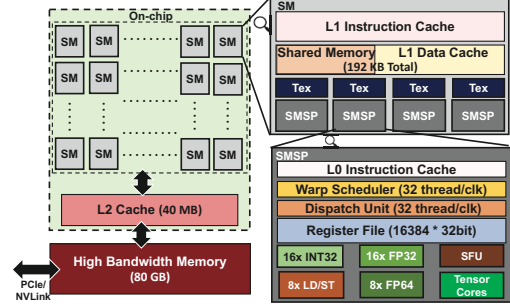

Fig. 3: Simplified Nvidia A100 GPU organization.

the embedding stage. Additionally, a modest on-chip cache hierarchy helps in capturing the locality in memory accesses. Table I shows the access latencies [42] for different access locations. Note that fetching data is more costlier compared to CPUs. Table II 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 ∼7x large sizes over V100 for L1 and L2, respectively, and (2) RTX 4090 offers ∼12x large L2 cache over RTX 3090 Ti.

TABLE I: Access Latencies for various levels of the A100 GPU memory hierarchy based on [42].

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

TABLE II: 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

C. Related Works

DLRM Optimizations: Prior works [15]–[17], [26] have looked into scheduling frameworks and heterogeneous platforms for inference serving and [27] discusses the system design for effective distributed inference. [29]–[31] 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), making it orthogonal, and thus it can be combined the prior works.

Accelerator designs: [18]–[24] 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 [33], our plug-and-play solutions can be instantly leveraged (Section IV).

Scheduling and Virtualization: Several warp and CTA scheduling works have been proposed in the past [43]–[45] 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 III-C). Register-file virtualization techniques [46]–[48] 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 IV).

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 [49]–[52] 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 IV-B).

L2 Cache Management: Recently, with Nvidia's Ampere architecture and onwards [38], [39], the GPUs feature a CUDA/PTX-based programmer control for L2 cache management [41]. [53]–[55] 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 IV-C).

III. 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.

A. 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 1 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) × (average lookups per
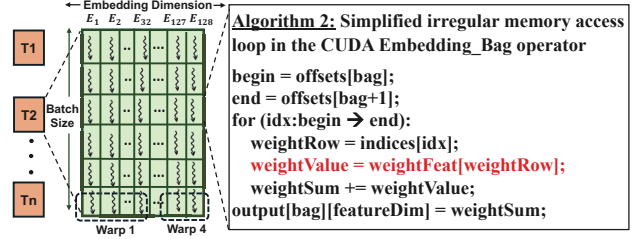

Fig. 4: Parallel implementation of embedding stage by work partitioning across CUDA threads. Here, 1000s of CUDA threads independently work on one output matrix element.

sample) × (ED) × (precision). For example, for our chosen configuration (described in Section V), the amount of data processed per table is 2048 × 150 × 128 × 4B = 150 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 1 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;

To better understand the incorporated parallelism, Figure 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 *"Embedding-Bag_updateOutputKernel_sum_mean"* [56]), which generates an output matrix of dimension (BS) × (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 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 1). 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
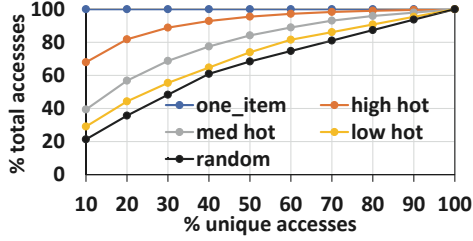
Fig. 5: Coverage study for different memory access patterns: it shows the % of total accesses (y axis) that are covered by the % of unique accesses (x axis).

operation results in the irregular loads which have a significant impact on (and causes variation in) the performance of the embedding stage (Figure 1).

B. 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 [10], [11], [15]. In our study, we use the datasets from a recent work [11], which extracts homogeneous datasets using Meta's production traces [57].

Building on prior works [10], [11], [15], [58], 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 III 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 III: 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 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.

C. 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 [59], [60] provide larger caches and HBM capacity, which directly help embeddings data reuse behavior and bandwidth needs. In this spirit, various previous works [15]–[17], [34], [58], [61] 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 [10], [11], [15], 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 4 [56] 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 III).

TABLE IV: 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

Table IV describes the off-the-shelf PyTorch characterization using various NCU metrics. Recall that Figure 1 highlighted that random performs 3.2× slower than the fastest one item case, even though both observe the same number of loads. This is 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 [11], 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 I. 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 [62]) 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 [59], 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 [56] 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 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" [63] 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 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
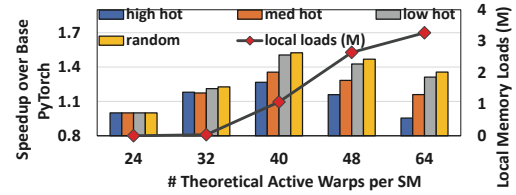

Fig. 6: Synthetically varying the number of registers allocated to improve WLP. The primary y-axis is speedup over off-the-shelf PyTorch, and the secondary y-axis is the register spilling penalty based on extra local memory loads (in millions). OptMT refers to the highest speedup at 40 warps.

(compared to the baseline) as the spilling penalty overshadows the potential benefits from multi-threading.

TABLE V: 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

Table V 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) 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*.

D. 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.

IV. 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.

A. 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 III-D.

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
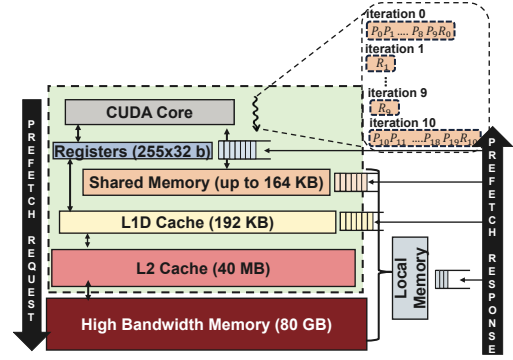

Fig. 7: Buffer locations used for various prefetching schemes. A CUDA thread is shown executing SMPF, a batch of 10 prefetches(P) are launched every 10th iteration, and a reduce(R) operation every iteration.

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 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.

B. 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 [64] and hardware [65], [66]. However, on-chip prefetching in GPUs is *uncommon*, due to GPUs extensive reliance on WLP for latency hiding [25]. 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. To comprehensively explore the design space of prefetching, we formulate a series of key questions as discussed next:

What to Prefetch? Recall that Section III-A 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 III), which leads to long scoreboard stalls (Tables IV and V). 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

```
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;
```

a) Register-based Prefetching

```
INITIALIZE: prefetch_bfr[256][10],
begin = offsets[bag], end = offsets[bag+1];
pf_cntr = 0;
START:
for (idx:begin → end):
    int trigger_prefetch = pf_cntr%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_cntr = pf_cntr + 1;
    weightValue = prefetch_bfr[thread_id]
                          [trigger_prefetch];
    weightSum += weightValue;
output[bag][featureDim] = weightSum;
```

b) Shared Memory-based
Prefetching

Fig. 8: Prefetching implementations: (a) RPF (b) SMPF

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 7 shows a total of 4 buffer stations – register, shared memory, L1 D$, and local memory. We do not pick L2 since its access latency is quite high (261.5 cycles [42]). Note that each buffer station has pros and cons. Specifically, in terms of access latency, the register is optimal, whereas, in terms of size, L1 D$ 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 [67]. 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 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" [68] to prefetch the data into L1D$, similar to commonly used CPU intrinsics [64]. To achieve timeliness (and ultimately optimal performance), we vary the prefetch distance to find the optimal value. Figure 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.
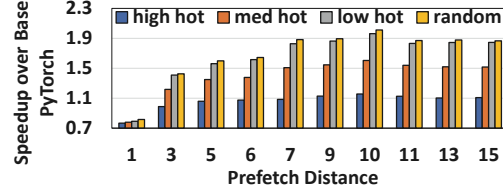

Fig. 9: Performance impact of prefetch distance in SMPF.

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 9, a distance of 1 hurts the performance for all datasets. Also, similar to [45], 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 9, for random case, while the long scoreboard stalls significanlty reduce from 18.6 cycles(Table IV) 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 I), their limited count can become a bottleneck, as noted with limited WLP and register spilling (Section III-C).(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 [52] or throttling of demand memory requests [69].

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.

C. 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 II). Further, Nvidia GPUs (from Ampere architecture onwards) have recently released support for CUDA programmer-based L2 cache access management (residency control [41], which enables a portion of the L2 cache to be used for persistent data access). Given that off-chip memory accesses are very costly (Table I), 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), we propose an L2 pinning (L2P) design that can benefit from L2 residency control for the embedding accesses. Towards this, we first discuss the
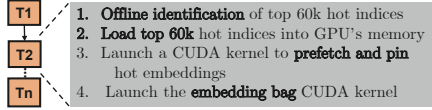
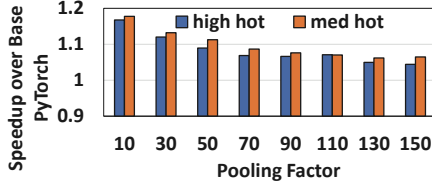Fig. 10: High-level design of L2P

Fig. 11: Detailed study of L2 pinning over various pooling factors. Speedup is reported over off-the-shelf PyTorch.

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 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 CUDA kernel. We use the inline PTX instruction "prefetch.global.L2::evict_last" [68], 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 I).

Performance Expectations and Overheads? Although hardware caches capture hotness to an extent (Table IV), 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 11).

Clearly, embedding access patterns can change over time, potentially reducing the effectiveness of L2 pinning. To ad-

dress this challenge, similar to prior research [70], 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.

D. 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$), it still suffers from suboptimal timeliness and puts pressure on the off-chip bandwidth. Similarly, even though pinning lowers the load latency by bringing and holding the frequently-accessed embeddings in the L2 cache, it still lacks because (1) 30MB of L2 set-aside cannot fully cover the working set required by the datasets, especially in the low hot and random cases, and (2) access latency with L2 is significantly high (261.5 cycles; see Table I), compared to registers, shared memory, or L1D$. 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.

V. METHODOLOGY

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

TABLE VI: System specifications used for evaluation

CPU	AMD EPYC 7763
RAM	1 TB
GPU	Nvidia A100-SXM4-80GB
# SMs	108
Register File per SM	64K × 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) [71] 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 [72].

Model: Taking inspiration from [10], [11], [15], [35], 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 $\sim$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 VII describes a mixture of tables for heterogeneous evaluation.

TABLE VII: 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 [11], we use the publicly released homogenized production traces from Meta [57], [73]. Thus, we consider a variety of hotness (memory access patterns): one item, high hot, medium hot, low hot, and random. Section III-B 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 III). Inspired by [11], [15], [57], [58], 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.

VI. EVALUATION

Based on real system measurements, we evaluate the benefits of our proposed latency hiding schemes: OptMT (Section III-C), Software Prefetching [RPF, SMPF, LMPF, L1DPF], L2 Pinning (L2P), as well as their combined versions (Section IV). 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 [36], [56] (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.

A. Key Results

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 12 evaluates the performance of various design points across different datasets. As highlighted earlier in Section III-C, 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 VI-B1). 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× speedup (random case). Interestingly, the highest benefit is 13.5% in
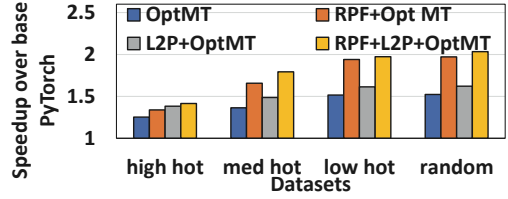

Fig. 12: Embedding-only improvement in latency of the proposed techniques with OptMT over off-the-shelf PyTorch.
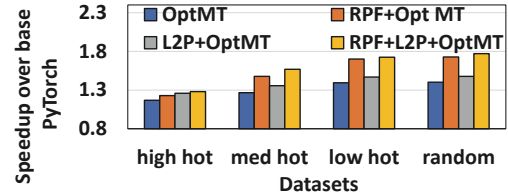

Fig. 13: End-to-end improvement in latency of the proposed techniques with OptMT over off-the-shelf PyTorch.

the med hot case over the previous optimal (RPF+OptMT), thus maximally complementing each other.

2) Boost in End-to-End DLRM Performance: Given that DLRM is composed of 4 stages (Section II-A) which collectively influence the batch latency, we also evaluate the benefit of our proposed schemes with respect to the end-to-end latency (Figure 13). Note that the trends in speedup remain similar to Figure 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 1 highlighted the performance gap between the fastest (one item dataset) and slowest (random dataset) loads being 3.2× and 2.1× 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 (higlighted in Figure 14) to only 1.57×, 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).

3) Microarchitectural Justifications: To better understand the above gains, we profile the proposed schemes using NCU [74]. Table VIII and Table IX show the microarchitectural measurements for RPF+OptMT and RPF+L2P+OptMT designs, respectively. Due to limitations in NCU profiling [75],
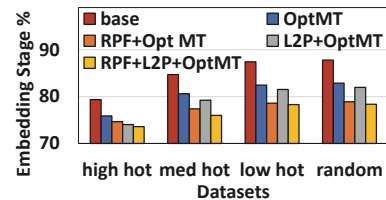

Fig. 14: Embedding stage contribution in the end-to-end latency.

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 VIII: 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	49.65	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 Utilization (%)	2.6	13.9	31.5	35

TABLE IX: 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 Utilization (%)	1.5	12.3	30.7	34.9

B. Sensitivity Analysis

1) Winning Prefetching Scheme: Earlier, Section IV 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 9. Interestingly, we find that all schemes perform best at a prefetch distance of 2. Figure 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 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 I). Thus, prefetching achieves {34%, 66%, 94%, 97%} speedups for the {high, med, low} hot and random, respectively.
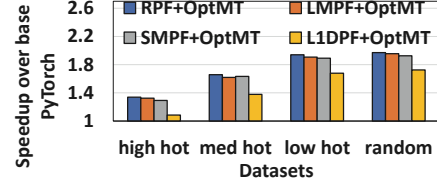

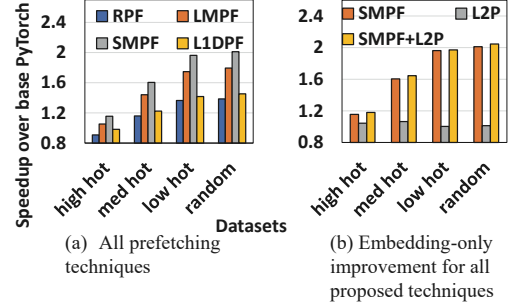Fig. 15: Comparison of all prefetching techniques with OptMT over off-the-shelf PyTorch.

(a) All prefetching techniques

(b) Embedding-only improvement for all proposed techniques

Fig. 16: Comparison of techniques for off-the-shelf Pytorch.

2) Improvement over Baseline PyTorch without OptMT: Earlier, Figure 6 highlighted that OptMT improves the performance over baseline PyTorch, and our proposed techniques can work in conjunction (Figure 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, and L1DPF}, respectively) (Figure 16.a). Compared to Figure 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 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 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 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
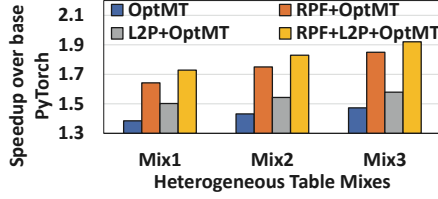
Fig. 17: Embedding-only improvement of the proposed techniques with OptMT over off-the-shelf PyTorch and heterogeneous tables.

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.

3) Heterogeneous Table Mixing: Given that DLRMs are executed in both homogeneous [11], [15] and heterogeneous settings [22], [58], we evaluate a synthetic case where non-uniform tables (Table VII) are present within the embedding stage by having a mixture of hotness. Figure 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.

4) Evaluation on H100 NVL GPU: We also evaluate the applicability our proposed schemes on an H100-NVL GPU [76], which is increasingly being embraced by the datacenters [77], [78]. 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 IV)). 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 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 6, higher gains of MT are visible for low hot and random cases. Finally, Figure 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%).

VII. DISCUSSION

A Static Profiling Framework for adoption of our proposed designs: Achieving the ideal performance (like
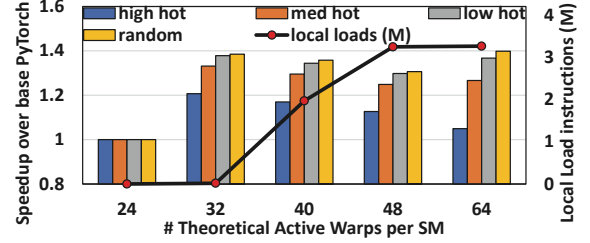

Fig. 18: On H100 NVL GPU, the number of registers allocated is varied to find optimal WLP. The primary y-axis is speedup over off-the-shelf PyTorch and the secondary y-axis is the register spilling penalty based on extra local memory loads (in millions). OptMT on H100 refers to the highest speedup at 32 warps.
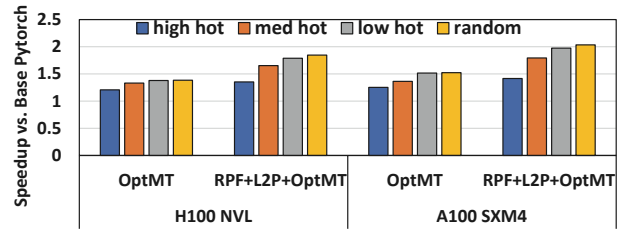

Fig. 19: Comparison of Embedding-only improvement in latency of the integrated scheme over off-the-shelf PyTorch for H100 NVL and A100 GPU.

Figure 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 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 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 [79] and Graph Mining [80].

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 [28]. Further, the forward pass in the training pipeline [81]–[83] 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.

VIII. 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.

ACKNOWLEDGEMENTS

All product names used here are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] Meta, "Facebook Recommendation System," 2024. [Online]. Available: https://ai.meta.com/blog/ai-unconnected-content-recommendations-facebook-instagram/

[2] ——, "Instagram Recommendation System," https://engineering.fb.com/2023/08/09/ml-applications/scaling-instagram-explore-recommendations-system/, 2024.

[3] TikTok, "TikTok Recommendation System," https://www.tiktok.com/transparency/en-us/recommendation-system/, 2024.

[4] Amazon, "Amazon Recommendation System," https://aws.amazon.com/personalize/, 2024.

[5] Netflix, "Netflix Recommendation System," https://research.netflix.com/research-area/recommendations, 2024.

[6] Hulu, "Hulu Recommendation System," https://help.hulu.com/article/hulu-personalized-recommendations#:~:text=While%20you're%20looking%20for,get%20to%20know%20you%20better., 2024.

[7] eBay, "eBay Recommendation System," https://innovation.ebayinc.com/tech/engineering/building-a-deep-learning-based-retrieval-system-for-personalized-recommendations/, 2024.

[8] AliBaba, "eBay Recommendation System," https://www.alibabacloud.com/blog/getting-started-with-recommendation-system_597740, 2024.

[9] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: http://arxiv.org/abs/1906.00091

[10] U. Gupta, 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, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[11] R. Jain, S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Optimizing cpu performance for recommendation systems at-scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589112

[12] K. Nair, A.-C. Pandey, S. Karabannavar, M. Arunachalam, J. Kalamatianos, V. Agrawal, S. Gupta, A. Sirasao, E. Delaye, S. Reinhardt *et al.*, "Parallelization strategies for dlrm embedding bag operator on amd cpus," *IEEE Micro*, 2024.

[13] M. D. Inference, "MLPerf Datacenter Inference 2024," https://mlcommons.org/benchmarks/inference-datacenter/, 2024.

[14] G. K. Jha, A. Thomas, N. Jain, S. Gobriel, T. Rosing, and R. Iyer, "Mem-rec: Memory efficient recommendation system using alternative representation," in *Asian Conference on Machine Learning*. PMLR, 2024, pp. 518–533.

[15] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "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, 2020, pp. 982–995.

[16] L. Ke, U. Gupta, M. Hempstead, C.-J. Wu, H.-H. S. Lee, and X. Zhang, "Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 141–154.

[17] S. Hsia, U. Gupta, M. Wilkening, C.-J. Wu, G.-Y. Wei, and D. Brooks, "Cross-stack workload characterization of deep recommendation systems," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 157–168.

[18] A. Firoozshahian, 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, "Mtia: First generation silicon targeting meta's recommendation systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589348

[19] L. Ke, 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, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 790–803.

[20] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, "Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 968–981.

[21] U. Gupta, S. Hsia, J. Zhang, M. Wilkening, J. Pombra, H.-H. S. Lee, G.-Y. Wei, C.-J. Wu, and D. Brooks, "Recpipe: Co-designing models and hardware to jointly optimize recommendation quality and performance," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 870–884.

[22] H. Kal, S. Lee, G. Ko, and W. W. Ro, "Space: locality-aware processing in heterogeneous memory for personalized recommendations," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 679–691.

[23] L. Ke, 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, "Near-memory processing in action: Accelerating personalized recommendation with axdimm," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, 2022.

[24] Y. Kwon, Y. Lee, and M. Rhu, "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*, 2019, pp. 740–753.

[25] Nvidia, "CUDA Programming Guide: The benefits of using GPUs," https://docs.nvidia.com/cuda/cuda-c-programming-guide/#the-benefits-of-using-gpus, 2024.

[26] L. Ke, X. Zhang, B. Lee, G. E. Suh, and H.-H. S. Lee, "Disaggrec: Architecting disaggregated systems for large-scale personalized recommendation," *arXiv preprint arXiv:2212.00939*, 2022.

[27] M. Lui, Y. Yetim, Ö. Özkan, Z. Zhao, S.-Y. Tsai, C.-J. Wu, and M. Hempstead, "Understanding capacity-driven scale-out neural recommendation inference," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 162–171.

[28] K. K. Matam, H. Ramezani, F. Wang, Z. Chen, Y. Dong, M. Ding, Z. Zhao, Z. Zhang, E. Wen, and A. Eisenman, "{QuickUpdate}: a {Real-Time} personalization system for {Large-Scale} recommendation models," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 731–744.

[29] H. Ye, S. Vedula, Y. Chen, Y. Yang, A. Bronstein, R. Dreslinski, T. Mudge, and N. Talati, "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*, 2023, pp. 282–301.

[30] M. Adnan, Y. E. Maboud, D. Mahajan, and P. J. Nair, "Accelerating recommendation system training by leveraging popular choices," *arXiv preprint arXiv:2103.00686*, 2021.

[31] Y. Kwon and M. Rhu, "Training personalized recommendation systems from (gpu) scratch: Look forward not backwards," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 860–873.

[32] D. Zha, L. Feng, Q. Tan, Z. Liu, K.-H. Lai, B. Bhushanam, Y. Tian, A. Kejariwal, and X. Hu, "Dreamshard: Generalizable embedding table placement for recommender systems," *Advances in Neural Information Processing Systems*, vol. 35, pp. 15 190–15 203, 2022.

[33] Meta, "Meta's 2024 ML Infrastructure," https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/, 2024.

[34] F. Lai, 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, "{AdaEmbed}: Adaptive embedding for {Large-Scale} recommendation models," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 817–831. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/lai

[35] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 993–1011. [Online]. Available: https://doi.org/10.1145/3470496.3533727

[36] Meta, "PyTorch DLRM," https://github.com/facebookresearch/dlrm/blob/639e3d25a59b35e6b703506a5764e611cdfe8bea/dlrm_s_pytorch.py#L590, 2024.

[37] ——, "Embedding ubenchmark in param," "https://github.com/facebookresearch/param/blob/0a073429d2139b5947212863b32b222a09239cd3/train/compute/pt/pytorch_emb.py#L49", 2024.

[38] Nvidia, "A100 GPU White Paper," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2024.

[39] ——, "Nvidia Hopper GPU WhitePaper," https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper, 2024.

[40] "Scoreboarding in GPUs," 2024, http://gpgpu-sim.org/manual/index.php/Main_Page#Scoreboard.

[41] Nvidia, "L2 cache residency control," https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-l2-access-management, 2024.

[42] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu, "Benchmarking and dissecting the nvidia hopper gpu architecture," *arXiv preprint arXiv:2402.13499*, 2024.

[43] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 395–406, 2013.

[44] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 2013, pp. 157–166.

[45] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *2015 IEEE 21st International symposium on high performance computer architecture (HPCA)*. IEEE, 2015, pp. 174–185.

[46] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 420–432.

[47] D. Voitsechov, A. Zulfiqar, M. Stephenson, M. Gebhart, and S. W. Keckler, "Software-directed techniques for improved gpu register file utilization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, pp. 1–23, 2018.

[48] Y. Oh, M. K. Yoon, W. J. Song, and W. W. Ro, "Finereg: Fine-grained register file management for augmenting gpu throughput," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 364–376.

[49] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 332–343.

[50] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Apogee: Adaptive prefetching on gpus for energy efficiency," in *Proceedings of the*

1230

22nd international conference on Parallel architectures and compilation techniques. IEEE, 2013, pp. 73–82.

[51] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, M. Annavaram, and W. W. Ro, "Adaptive cooperation of prefetching and warp scheduling on gpus," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 609–616, 2018.

[52] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer, "Pacman: prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 442–453.

[53] Y. Fu, E. Bolotin, A. Jaleel, G. Dalal, S. Mannor, J. Subag, N. Korem, M. Behar, and D. Nellans, "Autoscratch: Ml-optimized cache management for inference-oriented gpus," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[54] T. Adufu and Y. Kim, "L2 cache access pattern analysis using static profiling of an application," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2023, pp. 97–102.

[55] ——, "Optimizing performance using gpu cache data residency based on application's access patterns," in *2023 24st Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2023, pp. 42–47.

[56] PyTorch, "Embedding Bag CUDA Kernel in PyTorch," https://github.com/pytorch/pytorch/blob/da7db5d345a10ffb5092b26c5159f56faec1d0ea/aten/src/ATen/native/cuda/EmbeddingBag.cu#L115, 2024.

[57] Meta, "Embedding lookup Production dataset," https://github.com/facebookresearch/dlrm_datasets, 2023.

[58] G. Sethi, B. Acun, N. Agarwal, C. Kozyrakis, C. Trippel, and C.-J. Wu, "Recshard: statistical feature-based memory optimization for industry-scale neural recommendation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 344–358.

[59] "NVIDIA A100 Tensor Core GPU." 2024, https://www.nvidia.com/en-us/data-center/a100/.

[60] "NVIDIA H100 Tensor Core GPU," 2024, https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/.

[61] Z. Lin, L. Feng, E. K. Ardestani, J. Lee, J. Lundell, C. Kim, A. Kejariwal, and J. D. Owens, "Building a performance model for deep learning recommendation model training on gpus," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2022, pp. 48–58.

[62] "NVIDIA Nsight Systems," 2024, https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[63] "nvcc maxrregcount flag," 2024, https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#maxrregcount-amount-maxrregcount.

[64] GCC, "Data Prefetching with GCC," https://gcc.gnu.org/projects/prefetch.html, 2024.

[65] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, pp. 1–29, 2012.

[66] B. Falsafi and T. F. Wenisch, *A primer on hardware prefetching*. Springer Nature, 2022.

[67] StackOverflow, "Where does Local Memory reside?" https://stackoverflow.com/questions/72381905/seeking-a-better-understanding-of-local-memory-in-cuda-where-does-it-live-how, 2024.

[68] NVIDIA, "Parallel Thread Execution ISA Version 8.4," https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-conversion-instructions-prefetch-prefetchu, 2024.

[69] P. G. Emma, A. Hartstein, T. R. Puzak, and V. Srinivasan, "Exploring the limits of prefetching," *IBM Journal of Research and Development*, vol. 49, no. 1, pp. 127–144, 2005.

[70] Y. Lee, S. H. Seo, H. Choi, H. U. Sul, S. Kim, J. W. Lee, and T. J. Ham, "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*, 2021, pp. 302–313.

[71] PyTorch, "PyTorch 2.1.0," https://github.com/pytorch/pytorch/tree/v2.1.0, 2024.

[72] Nvidia, "Nvidia's NVCC compiler," https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#opt-level-n-o, 2024.

[73] R. Jain, "Homogeneous Production Traces," https://github.com/rishucoding/reproduce_isca23_cpu_DLRM_inference, 2023.

[74] Nvidia, "Nvidia's Nsight Compute Tool," https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html, 2024.

[75] NVIDIA, "NCU profiling limitation," https://forums.developer.nvidia.com/t/ncu-profiling-with-cache-control/246113/, 2024.

[76] ——, "H100 NVL 96 GB," "https://www.nvidia.com/en-us/data-center/h100/", 2024, accessed: 2024-02-07.

[77] Meta, "Meta H100 Infrastructure 2024," https://engineering.fb.com/2024/03/12/datacenter-engineering/building-metas-genai-infrastructure/, 2024.

[78] Amazon, "Amazon EC2-P5 H100 Instance," https://aws.amazon.com/blogs/aws/new-amazon-ec2-p5-instances-powered-by-nvidia-h100-tensor-core-gpus-for-accelerating-generative-ai-and-hpc-applications/, 2024.

[79] X. Song, Y. Zhang, R. Chen, and H. Chen, "Ugache: A unified gpu cache for embedding-based deep learning," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 627–641.

[80] Y. Yuan, H. Ye, S. V. W. Kaza, and N. Talati, "Everest: Gpu-accelerated system for mining temporal motifs," *arXiv preprint arXiv:2310.02800*, 2023.

[81] A. Sharma, V. M. Bhasi, S. Singh, R. Jain, J. R. Gunasekaran, S. Mitra, M. T. Kandemir, G. Kesidis, and C. R. Das, "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)*, 2023, pp. 1–12.

[82] ——, "Analysis of distributed deep learning in the cloud," 2022. [Online]. Available: https://arxiv.org/abs/2208.14344

[83] S. Singh, A. Sarma, S. Lu, A. Sengupta, M. T. Kandemir, E. Neftci, V. Narayanan, and C. R. Das, "Skipper: Enabling efficient snn training through activation-checkpointing and time-skipping," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 565–581.

A. Abstract

The artifact covers the complete steps to setup DLRM inference on GPUs. It provides the codebase for the proposed schemes: (1) improve WLP by lowering register allocation (2) various prefetching designs (3) L2 pinning design (4) most performant combined design (RPF + L2P + OptMT). Also, the necessary datasets are shared. Overall, the steps are shared to help reproduce the figures in the results section.

B. Artifact check-list (meta-information)

- **Algorithm:** DLRM inference
- **Program:** DLRM implementation from Meta using PyTorch
- **Compilation:** gcc 11.4.0, nvcc 12.2
- **Model:** DLRM variants mentioned in Gupta et al – Section V
- **Data set:** Section V
- **Run-time environment:** Ubuntu 22.04.4 LTS
- **Hardware:** CPU: AMD EPYC 7763 64-Core Processor, GPU: Nvidia A100-SXM4-80G (complete details in Table VI)
- **Metrics:** Batch Latency (ms), Speedup over base PyTorch
- **Output:** Batch Latency (ms)
- **Experiments:** Figure 12
- **How much disk space required (approximately)?:** 80GB
- **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

C. Description

1) How to access: The codebase is available on Zenodo at https://doi.org/10.5281/zenodo.13325108 and Github at https://github.com/rishucoding/reproduce_MICRO24_GPU_DLRM_inference

2) Software dependencies: The required software dependencies are outlined in the repository.

3) Data sets: The required dataset files are added to the repository.

4) Models: Steps to save and load models are added in the repository.

D. 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) Install PyTorch
3) Evaluate baseline performance over various datasets.
4) Evaluate OptMT over various datasets.
5) Evaluate RPF+OptMT over various datasets.
6) Evaluate L2P+OptMT over various datasets.
7) Evaluate RPF+L2P+OptMT over various datasets.

E. Experiment workflow

We suggest to follow the README.md file in the above repository.

F. Evaluation and expected results

Figure 12 can be directly reproduced following the given steps.

G. Experiment customization

Models, datasets, and optimization designs can be customized to evaluate various configurations, and thus reproduce majority of results shown in Section VI.

H. Notes

Please raise a Github issue for any questions.