GRACE: A Scalable Graph-Based Approach to Accelerating Recommendation Model Inference

Haojie Ye
University of Michigan
Ann Arbor, Michigan, USA
yehaojie@umich.edu

Sanketh Vedula
Technion
Haifa, Israel
sanketh@campus.technion.ac.il

Yuhan Chen
University of Michigan
Ann Arbor, Michigan, USA
chenyh@umich.edu

Yichen Yang
University of Michigan
Ann Arbor, Michigan, USA
yangych@umich.edu

Alex Bronstein
Technion
Haifa, Israel
bron@cs.technion.ac.il

Ronald Dreslinski
University of Michigan
Ann Arbor, Michigan, USA
rdreslin@umich.edu

Trevor Mudge
University of Michigan
Ann Arbor, Michigan, USA
tnm@umich.edu

Nishil Talati
University of Michigan
Ann Arbor, Michigan, USA
talatin@umich.edu

ABSTRACT

The high memory bandwidth demand of sparse embedding layers continues to be a critical challenge in scaling the performance of recommendation models. While prior works have exploited heterogeneous memory system designs and partial embedding sum memoization techniques, they offer limited benefits. This is because prior designs either target a very small subset of embeddings to simplify their analysis or incur a high processing cost to account for all embeddings, which does not scale with the large sizes of modern embedding tables. This paper proposes GRACE—a lightweight and scalable graph-based algorithm-system co-design framework to significantly improve the embedding layer performance of recommendation models. GRACE proposes a novel Item Co-occurrence Graph (ICG) that *scalably* records item co-occurrences. GRACE then presents a new system-aware ICG clustering algorithm to find frequently accessed item combinations of *arbitrary lengths* to compute and memoize their partial sums. High-frequency partial sums are stored in a software-managed cache space to reduce memory traffic and improve the throughput of computing sparse features. We further present a cache data layout and low-cost address computation logic to efficiently lookup item embeddings and their partial sums. Our evaluation shows that GRACE significantly outperforms the state-of-the-art techniques SPACE and MERCI by 1.5× and 1.4×, respectively.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582029

KEYWORDS

DLRM, Embedding Reduction, Algorithm-System Co-Design

ACM Reference Format:
Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex Bronstein, Ronald Dreslinski, Trevor Mudge, and Nishil Talati. 2023. GRACE: A Scalable Graph-Based Approach to Accelerating Recommendation Model Inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3582016.3582029

1 INTRODUCTION

Deep Learning Recommendation Models (DLRMs) are widely employed to predict rankings of news feeds and entertainment content [18, 21]. An earlier work [26] shows that DLRMs consume a majority of AI inference cycles of data centers. DLRM exhibits a mix of workload characteristics with fully connected *dense* neural network layers and *sparse* embedding layers. The sparse embedding layers are the primary performance bottlenecks of DLRM execution due to their high memory bandwidth requirement [24, 26, 28, 34, 38, 43, 44]. Because this application runs at a population scale, the execution bottlenecks significantly increase the Total Cost of Ownership (TCO) and power consumption of data centers [5, 35]. Therefore, improving DLRM performance directly results in saving millions of dollars in cost and carbon emission [66].

The key challenge in accelerating the DLRM embedding layer performance is to exploit spatial and temporal locality. This challenge is because of the irregular nature of the workload's memory access pattern over large embedding tables. Recently, several techniques have attempted to improve the DLRM embedding layer inference performance either by caching partial sums of embeddings leading to reduced memory traffic [34, 48] or by exploiting the heterogeneous memory systems [1, 34, 38]. These approaches, however, fall short in the following manner. First, FAE [1] and RecNMP [38] employ heterogeneous memory systems to exploit the power-law in the item access frequency distribution; however, they do not improve the memory traffic. Second, SPACE [34] employs

a heuristic threshold to select a small subset of popular items and stores exhaustive combinations of two-item partial sums that leads to low memory bandwidth reduction. Third, MERCI [48] employs an expensive user trace processing technique to store partial sums of more than two items. It has three main drawbacks: (i) the algorithm does not scale to large embedding tables, (ii) the algorithm operates on the level of sub-groups of embeddings and it does not capture a global view of user-item interactions; thus the resulting partial sum formation is based on a limited scope of user-item interactions, leading to sub-optimal memory traffic reduction, and (iii) its design is unaware of memory heterogeneity. An ideal design goal is to significantly *reduce memory traffic* while *exploiting memory heterogeneity* in a *scalable* fashion.

This paper presents GRACE—a scalable graph-based algorithm-system co-design that significantly improves the memory system performance of DLRM embedding reduction on commodity hardware. Due to the software-only nature of its design, GRACE can be immediately deployable in today's data centers. The design goals of GRACE are four-fold: (1) exploit spatial and temporal locality in the workload, (2) significant memory traffic reduction, (3) memory heterogeneity awareness, and (4) scalability to large embedding table sizes. To this end, we cast the problem of scalably identifying popular item combinations of arbitrary lengths to a graph problem. Using the outcome of this problem, we present a generic system design framework to improve DLRM performance.

Specifically, GRACE analyzes the item preferences of different users to construct an Item Co-occurrence Graph (ICG). Nodes in this graph represent items, and edge weights represent the number of times two items are co-accessed. Mapping co-occurrence frequencies to a graph offers a global view of co-occurrence events that can scale to a large number of users/items. We then propose a novel clustering algorithm for ICG that finds frequently accessed item combinations. Each resulting cluster is a set of co-accessed items. To best exploit this algorithmic framework, GRACE stores the partial sums of frequently co-accessed item combinations into a software-managed cache space. The ICG clustering algorithm is cache layout aware. GRACE effectively navigates the trade-off between memory traffic reduction and heterogeneous memory bandwidth utilization by appropriately distributing partial sums and single-item embeddings into cached and non-cached spaces. ICG construction and clustering, and partial sum cache data injection are performed offline without affecting ongoing inference cycles. At runtime, GRACE exploits both cached partial sums and frequently accessed single-item embeddings to significantly reduce the memory traffic and improve spatial and temporal locality.

To showcase the effectiveness of GRACE, we use a case study of a heterogeneous CPU-GPU system, widely deployed in today's data centers [25, 45, 63, 76] for executing DLRMs. In this system, the capacity-limited GPU memory acts as a software-managed cache. Our evaluation[1] shows that GRACE outperforms the state-of-the-art techniques SPACE [34] and MERCI [48] by 1.5× and 1.4×, respectively. We further show that GRACE reduces the memory traffic of embedding reduction by 1.5× and 1.1× over SPACE and MERCI. GRACE also improves performance over prior works by balancing the traffic between the heterogeneous memory system. The graph clustering algorithm in GRACE scales well with the increase in the number of user/items, reducing the processing cost by 8.3× compared to MERCI. The scalable nature of GRACE enables analysis of large user-item interaction traces and embedding tables in a practical fashion. We demonstrate the generality of GRACE by presenting case studies of two additional hardware platforms: 1) a homogeneous GPU memory and 2) a DIMM-HBM heterogeneous memory with Processing-In-Memory (PIM) capability. These studies show consistent improvements of GRACE over prior systems.

Compared to the state-of-the-art system MERCI, GRACE makes the following novel contributions. First, GRACE fundamentally redesigns the problem of finding frequently accessed item combinations by formulating it as a graph problem. This formulation provides a global view of the user-item access trace, as opposed to MERCI, which operates with a limited scope of user-item interactions. Second, GRACE proposes a scalable clustering algorithm whose complexity grows linearly with the number of items and is independent of the number of users. To compare, the runtime complexity of MERCI is quadratic in the number of items and increases linearly with the number of users. Third, GRACE design is memory heterogeneity-aware, which caters to the data center system modeling of the DLRM workload deployment [24, 63], while MERCI is designed only for homogeneous memory systems.

To summarize, the key contributions of GRACE are as follows:

- Casting the problem of finding popular item combinations in DLRM to a graph problem.
- Introduction of novel Item Co-occurrence Graph (ICG) that scalably records co-accessed item combinations for DLRM.
- A system-aware and scalable graph clustering algorithm aimed at finding arbitrary-length popular item combinations within the capacity-limited cache space.
- GRACE—an algorithm-system co-design that reduces memory traffic and exploits heterogeneous memory system to improve end-to-end DLRM throughput by 1.40× and 1.35× compared to the state-of-the-art frameworks SPACE [34] and MERCI [48], respectively.
- GRACE is open-source for the benefit of the broader research community: https://github.com/Linestro/GRACE.

2 BACKGROUND

2.1 Personalized Recommendation Models

The goal of DLRM is to predict the Click-Through Rate (CTR) [13, 18, 54, 72, 78], *i.e.,* the probability of a user clicking on an advertised item. A major data center operator Meta (previously Facebook) has claimed [26] that DLRM models consume more than 60% of their AI inference cycles in production, which makes them a leading candidate for optimization. In contrast to traditional deep neural network (DNN) models, DLRM features a hybrid architecture of multi-layer perceptron (MLP) models and embedding layers. The "dense" input features (*e.g.*, age, gender, and location of the user) are processed by the first MLP to generate dense features. The sparse input features (*e.g.*, previous user-item interactions), on the other hand, are processed by the embedding layers. An embedding layer contains a large embedding table that stores feature vectors of different items. A user's past interactions with items are used

[1]We use an in-house implementation for SPACE [34], and open-source implementation [3] of MERCI [48] by the authors.

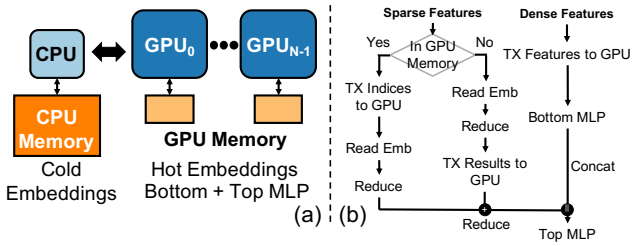

Figure 1: (a) A heterogeneous CPU-GPU system executing DLRM inference, and (b) workflow of DLRM inference execution with a heterogeneous system.

to index these tables to extract items' features. These features are then reduced to represent the summary of the user's interests. This layer performs sparse computation because a user only interacts with a handful of items out of millions of available items. These sparse and dense features are thereafter concatenated and fed into another MLP layer to predict the CTR.

2.2 DLRM Inference with GPU Support

DLRM systems in production [25, 45, 63, 76] employ a hybrid CPU-GPU design to execute MLPs and memory-bandwidth–demanding embedding layers in DLRM models. A simplified depiction of executing DLRM models on a hybrid CPU-GPU system is presented in Fig. 1(a). GPU executes MLPs to exploit higher compute throughput. The high-bandwidth GPU memory is used to handle the memory bandwidth-intensive reduction operations of the embedding layers. However, the embedding tables that store all item features can amount from tens of GBs to TBs, making it impossible to fit the entire table into GPU memory. Thus, the GPU memory acts as a software-managed cache space to store a portion of the embedding tables [1, 25, 34, 45, 63]. Low-bandwidth CPU memory with high capacity is employed to store and reduce the rest of the embedding entries that do not fit in the GPU. We further show in Fig. 1(b) the state-of-the-art DLRM inference framework that incorporates a GPU. After receiving a batch of user requests, the requested embedding indices are transferred (TX) to the GPU and are evaluated for whether each of them is on CPU or GPU. The embedding reduction operations will distribute to the corresponding memory and CPU/GPU reduces the embeddings to produce the results for each user before the results are finalized on GPU for top MLP layers.

2.3 Exploiting Popular Choices in DLRMs

Real-world DLRM inputs follow a power-law distribution [1, 22, 48, 58, 68], where a small collection of popular items accounts for a large fraction of embedding table accesses. Below, we summarize prior works that exploit power-law distribution for optimization.

- ***FAE** [1]* proposes a framework that constructs an empirical distribution of item access frequencies by profiling a portion of the user-item access trace. The framework then calibrates a popularity threshold and uses the GPU memory to store the highly accessed embeddings.
- ***RecNMP** [38]* proposes a small cache structure to each rank-level near-memory processing module to bypass the DRAM loads of frequently accessed items.
- ***SPACE** [34]* employs a hybrid memory architecture with HBM and DIMM, where HBM stores popular user choices. SPACE introduces two new concepts called *gather locality* and *reduction locality*. The power-law nature of the item access frequencies implies that preferential treatment of popular items (*i.e.*, placing them in HBM) can promote gather locality. Reduction locality, on the other hand, is availed by storing partial reductions of *any* two popular item vectors. Specifically, SPACE uses `psum2`, *i.e.*, reduction of embedding vectors of pairs of popular items. To exploit these two types of locality, SPACE pre-processes the user-item access trace to extract popular item choices and their combinations. These popular embedding vectors are stored in capacity-limited HBM that enables high-bandwidth access, while other embedding vectors are extracted from DIMMs.
- ***MERCI** [48]* generalizes SPACE by storing partial sums of more than two items. MERCI inspects the user-item interaction trace, analyzes popular co-accessed items, and merges them into clusters. Within the cluster, all partial sums are stored using the additional DRAM storage.

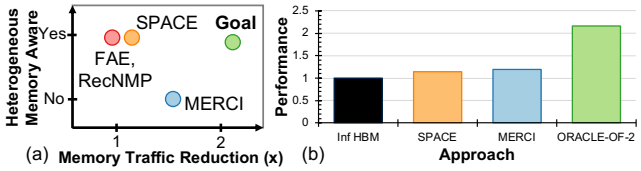

Figure 2: The landscape of DLRM embedding layer optimization design space and their respective performance over an infinite GPU memory model.

3 UNDERSTANDING THE CHALLENGES IN ACCELERATING DLRM INFERENCE

3.1 Growing Data Sizes and Demands

The recent development of DLRM observes a super-linear growth of capacity and bandwidth demands. The evolution in DLRM has resulted in much richer embedding features, leading to increased data volumes. The memory footprint of DLRM has increased by 16 times, reaching an order of terabytes within four years [52, 77]. Additionally, the inherently irregular nature of memory accesses over large embedding tables results in a significant portion of accesses that cannot be served using capacity-limited caches, increasing the off-chip memory bandwidth requirements. The bandwidth demand of DLRM embedding layers has increased by 30 times to 2TB/s, dramatically outpacing the bandwidth growth of accelerator memories and interconnections [63].

3.2 Limitations of Prior Works

Fig. 2(a) shows the landscape of optimization directions divided into memory traffic reduction and heterogeneous memory aware placement for better memory bandwidth utilization. The goal is to achieve both high memory traffic reduction and high memory bandwidth utilization of the heterogeneous memory at the same time. However, we show in the following that none of the prior works supports designs in both optimization directions, and thus, results in sub-optimal performance.

Memory traffic reduction. In what follows, we discuss prior works that attempt to improve the DLRM embedding layer performance. As many items are frequently accessed together, these works propose storing their partial sums, resulting in a memory traffic reduction. **SPACE** [34] uses a subset of the most popular

items and caches partial sums (*psum*) all two–item combinations of popular items to produce *reduction locality*. This reduces the memory bandwidth requirement. However, we show that SPACE only reduces the memory traffic by 1.09× on average (detailed in Fig. 10). While the most popular single items are likely to be accessed by different users, there is no guarantee of users accessing *all possible* two popular item combinations frequently. We find that only an average of 25% of popular *psum* of two items stored in SPACE represent 95% of the accesses to the cache space. This shows that a majority of cached partial sums in SPACE are accessed significantly less frequently. This caching space can be better utilized by storing other more frequent patterns. Also, SPACE only tracks *psums* among $O\left(\sqrt{n}\right)$ frequently accessed items, where n is the number of items, making the strategy unscalable to large datasets.

MERCI [48] finds that items that co-occur can benefit from memoization of their partial sums. Such memoization can be generalized to clustering the co-occurred items and storing all *psums* within each cluster. MERCI proposes generating the most cost-efficient item clusters to fill the cache space. However, the main drawback is the complexity of generating such clusters. MERCI first classifies each item as a single-item cluster and recursively measures the benefit of merging any two clusters. By merging two clusters, all the partial sums within the clustered items are stored to reduce the memory bandwidth requirement. The amount of bandwidth reduction is measured by inspecting the inverse map of the *full* training trace. The clustering incurs an overhead of $O\left(C \times n^2 \times m\right)$, where C is the maximum capacity of generating *psums*, n is the number of items and m is the number of users in the training trace. In practice, MERCI breaks the total item set into k sub-groups using an off-the-shelf algorithm [8] and only merges within the sub-group. This reduces the complexity to $O\left(C \times k \times \left(\frac{n}{k}\right)^2 \times m\right)$. Nevertheless, the complexity of MERCI grows super-linearly with the increase in the number of items and number of users, making the algorithm not scalable to large datasets.

Memory heterogeneity awareness. Both **FAE** [1] and **RecNMP** [38] set a heuristic threshold to distribute popular items to GPU/RankCache and exploit the high bandwidth memory of the heterogeneous system. **SPACE** stores popular embeddings (and *psums*) to produce gather locality.

To analyze the importance of memory heterogeneity awareness, we measure the performance of executing DLRM on a host machine that uses a heterogeneous memory system (configuration detailed in §6). For baseline performance analysis, we assume an infinite GPU memory capacity and naively migrate all embeddings (no *psums*) to the cache space, *i.e.,* the GPU memory. We further tested an oracle-of-2 framework that assumes *psum* of **any** 2 item embeddings can be accessed, and memory throughput on the heterogeneous memory system achieves a perfect balance between the CPU and GPU memory. While an oracle-of-3 or more is possible, we choose an oracle-of-2 to compare with MERCI and GRACE because it provides a reasonable roofline for the reduction factor. Fig. 2(b) shows that with the same additional capacity on GPU, two strategic frameworks SPACE and MERCI only outperform the baseline by 1.14× and 1.20× on average, while oracle-of-2 outperforms the baseline by 2.16×. We conclude 2 key reasons for this gap.

(a) Low memory traffic reduction ratio. The main speedup of SPACE stems from setting a heuristic threshold and storing popular item embeddings on HBM. This empirically distributes the traffic to both CPU memory and GPU memory, achieving a higher collective bandwidth. However, the traffic reduction ratio only goes up to 1.09× in SPACE due to the reduction strategy being unscalable to large datasets. FAE and RecNMP fall into the same category with a traffic reduction ratio of 1 (no reduction).

(b) Lack of heterogeneous memory awareness. MERCI finds items clustering assignments that maximize the memory traffic reduction using *psums*. However, this does not necessarily lead to optimal performance. By caching *psums* to the capacity-limited GPU, the bandwidth requirement is reduced, but this comes at the cost of excessively populating the cache space with *psums* that are rarely accessed. The occupied memory capacity for storing such *psums* prevents adding single-item embeddings to the GPU. This causes many item embeddings to be accessed from DIMM-based low bandwidth memory, throttling the overall memory throughput (detailed in §7.1). *An ideal clustering algorithm should be memory heterogeneity aware and balance the trade-off between memory traffic reduction and the heterogeneous memory throughput to achieve optimal performance.*

3.3 Challenges in Scalable System Design

Today's DLRM models involve several million items accessed by tens of millions of users [52, 77]. Scalably identifying frequently accessed item combinations that result in an effective memory traffic reduction remains a major challenge. Additionally, prior works do not systematically optimize for a *collective* bandwidth reduction of the heterogeneous memory system, resulting in a memory throughput imbalance.

4 GRACE ALGORITHMIC FRAMEWORK

This section presents a novel algorithmic framework of GRACE to tackle the aforementioned challenges. The framework designs the content of the capacity-limited cache space to maximize the DLRM inference performance. The designed cache space can contain both popular item embeddings and partial sums of item combinations of arbitrary lengths. We then present complexity and runtime overhead analysis to demonstrate the practicality of our algorithm.

4.1 Design Goals

The goal of the GRACE algorithmic framework is to make the most efficient use of the cache space to store frequently accessed items and their combinations, given the capacity limitation. In particular, the algorithmic framework must meet the following expectations:

- *No exhaustive caching.* As discussed in §3, storing all pairs of highly accessed items leads to an $O\left(n^2\right)$ space complexity, where n is the number of highly accessed cached items. In this setting, it is not guaranteed for all of the two frequently accessed items to be frequently *co-accessed*; caching partial sums of rarely co-accessed items wastes cache space. Thus, the algorithm must *not* exhaustively cache all the possible partial sums of highly accessed items.
- *Scalable with trace size.* The algorithm to build the cache space must have low complexity. In practice, the user-item interaction trace size can grow infinitely, and the number of users and items can scale to many millions. Therefore, a

high-complexity algorithm to find popular partial sums to cache can lead to prohibitive analysis times.
- *System awareness.* The algorithm should account for different dataset characteristics and underlying system configurations, and be extensible to multiple embedding tables to achieve optimal performance in realistic deployment environments.

4.2 Algorithm Details

Given the user-item interaction trace, the goal of the algorithm is to find the most frequently accessed items and item combinations. Naively counting frequencies of all item combinations results in a combinatorial explosion, thus it is not feasible even for a small number of item combinations. To tackle this problem, here we introduce the notion of an *Item Co-occurrence Graph (ICG)*. In an ICG, the nodes represent items, and edge weights represent the frequency of co-occurrence of items across the sampled user access patterns. We cast the problem of scalably tracking frequencies of arbitrary-sized item combinations as a *graph problem* on the ICG. The user-item interaction trace can have different orders of items being accessed (*i.e.*, irregular accesses) by users, and the trace size can grow infinitely. Key advantages of representing user-item interaction trace via ICG are (i) the graph size is *invariant* to the number of users, (ii) it is an *order-agnostic* representation of user-item trace, (iii) the number of nodes in the graph grows only *linearly* in the number of items. Heavily weighted edges in the ICG efficiently capture highly co-accessed combinations of items gathered from all user-item interactions. Thus, ICG provides a succinct global view over the user-item interaction trace, and allows for the design of efficient graph analysis algorithms that scale to large numbers of users and items. In what follows, we present a unified algorithmic framework that identifies frequently accessed single items and their combinations using the ICG. This is a two–phase algorithm: the first phase records user preferences and constructs of the ICG, and the second phase clusters this graph to find popular items and their combinations.

ICG Construction Phase. We are provided with sampled historical data of items accessed by users. Each user has a list of accessed items organized in a data structure `user_accesses`. We use these user-item accesses to construct the ICG which contains the frequency of co-occurrence of items aggregated across users. Alg. 4 (see Appendix §A.1) presents the pseudo-code of this graph construction phase. To build the ICG, we first randomly sample users. For each sampled user, we buffer all pairs of items accessed by the user as item co-occurrences. We then use this item co-occurrence buffer to construct a weighted graph by increasing the edge-weight by 1 for each co-occurrence. The buffer of edges/item co-occurrences can be constructed online by a fire-and-forget process without impacting the performance of ongoing DLRM inference; the weighted ICG is constructed offline during the cache design phase. Further discussion on our proposed usage model is presented in the sequel (§5.1).

ICG Clustering Phase. This phase clusters the ICG. The goal of this algorithm is to identify frequently occurring item combinations from the user access patterns. Post clustering, the nodes (items) from the same cluster are deemed to be accessed together frequently. One way to cluster the graphs is by employing off-the-shelf graph clustering algorithms such as Metis [46]. Notably, these clustering algorithms optimize for different criteria and do not create clusters that minimize DLRM bandwidth as we show in §7.

Algorithm 1 Pseudocode for partitioning ICG into clusters

```
1:  procedure ClusterICG( )                                  ▷ Offline ICG clustering
2:      Input: G: Item Co-occurrence Graph (ICG)
3:      Input: nodes: Vertex set of G sorted by their degrees
4:      Input: capacity_budget: Number of cache lines allowed in cache space
5:      Output: cluster_list: Assignment of ICG nodes into different clusters
6:
7:      node_idx=0; cluster_id=0; occupied_space=|nodes|
8:      active_list[u]=0, ∀ u ∈ nodes          ▷ indicator whether a node is clustered
9:      while node_idx < |nodes| do
10:         anchor_node = nodes[node_idx]
11:         remaining_memory = capacity_budget - occupied_space
12:         // Create a cluster using an anchor node
13:         cluster = FormCluster(G, anchor_node, active_list,
14:                                              remaining_memory)
15:         // Calculate occupied space, break if OOM
16:         occupied_space += 2^cluster.size() - 1 - cluster.size()
17:         if occupied_space >= capacity_budget then
18:             break
19:         cluster_list[cluster_id] = cluster
20:         cluster_id += 1
21:         while !active_list[nodes[node_idx]] do
22:             node_idx += 1          ▷ increment until reaching the first active node
23:     return cluster_list
```

GRACE proposes a novel clustering algorithm that *clusters the graph with the objective of maximizing bandwidth reduction in DLRM*. Our proposed algorithm is *caching space-aware*, *i.e.*, it also accounts for capacity-limited cache space for clustering decisions. Post clustering, GRACE caches the partial sums of embeddings of all item combinations within each cluster. During inference, these cached partial sums are used to (i) reduce memory traffic, and (ii) avail efficient memory accesses to increase end-to-end DLRM throughput.

Alg. 1 presents the pseudocode of the proposed ICG clustering phase. The proposed algorithm uses a *greedy* approach to form the clusters. The inputs to the clustering algorithm are: (i) the ICG generated in the *ICG Construction Phase*; (ii) a sorted vertex list, where nodes are sorted by their degrees in ICG; (iii) a capacity budget, denoting the number of lines of item embeddings/*psums* allowed in the cache space. We maintain an *active list* of vertices that are not clustered and update this list as the algorithm progresses. The algorithm loops over all active vertices and attempts to greedily form new clusters. Within each loop, the largest degree vertex that is active is chosen as an *anchor node* and is passed to FormCluster() to form a cluster of an arbitrary size. Upon forming a cluster, `occupied_space` is updated. For each cluster, the algorithm saves *all combinations* of its constituent items, taking an additional size of $2^{\texttt{cluster\_size}} - 1 - \texttt{cluster\_size}$ compared to originally stored item embeddings. The algorithm terminates when the `occupied_space` reaches the capacity budget. We now detail how to form clusters.

Forming a cluster. Alg. 2 presents the pseudocode for forming individual clusters. The function receives four inputs: (a) the ICG, (b) `anchor_node`—a starting node from which we attempt to form a cluster, (c) active list of nodes that are not yet clustered; (d) remaining cache capacity. Given an anchor node, all its neighbors that are part of the active list become the candidates to be added to the cluster. We use a cost-benefit model to estimate the cost efficiency obtained by including a new node in the cluster. To select the best candidate to add to the existing cluster, we compute the estimated benefit of each of the candidates to the cluster, and admit the node that yields the maximum expected benefit. This algorithm

Algorithm 2 Pseudocode for an algorithm to form a new cluster under a capacity budget.

```
1: procedure FormCluster(g, anchor_node, active_list)                ▷ Offline
2:     Input: G: Item co-occurrence graph (ICG)
3:     Input: anchor_node: The node selected to create the cluster
4:     Input: active_list: List of unclustered nodes
5:     Input: remaining_capacity: Remaining memory within the cache space
6:     Output: cluster: Formed cluster containing anchor_node
7:     Constant: MAX_CLUSTER_SIZE: Maximum size of any cluster
8:
9:    candidates = {anchor_node}
10:   best_candidate = anchor_node
11:   current_benefit = 0
12:   while true do
13:       cluster.append(best_candidate)
14:       candidates.remove(best_candidate)
15:       active_list.erase(best_candidate)
16:       // Potential candidates while inducting the next node into cluster
17:       candidates.add(active_list ∩ neigh(best_candidate))
18:       best_candidate = -1
19:       current_benefit *= tolerance_factor
20:       if cluster.size() >= MAX_CLUSTER_SIZE then
21:           break
22:       if 2^cluster.size()+1 − 1 >= remaining_memory then
23:           break
24:       for all candidate ∈ candidates do
25:           // Estimate benefit of adding the candidate to the cluster
26:           est_benefit = EstimateBenefit(G, cluster, candidate)
27:           if est_benefit > current_benefit then
28:               current_benefit = est_benefit
29:               best_candidate = candidate
30:       if best_candidate < 0 then
31:           return cluster
32:   return cluster
```

is greedy because it chooses the *next* best node from the candidate set to insert into the clusters. When a new node is admitted to the cluster, it is removed from the candidate set and the active list. For the next iteration, the candidate set is updated to contain the neighbors of all the nodes in the cluster so far that are in the active list. In each round, after determining a new node to join the cluster, we record the total estimated benefit so far. When new candidates are evaluated, they are deemed valid to join the cluster if the cost efficiency yielded by their addition to the cluster is greater than the previous cost efficiency (within a specified tolerance level). This procedure terminates when one of the following criteria is satisfied: (i) no valid candidates are found to add to the cluster based on the estimated benefits; (ii) the cluster size exceeds a maximum cluster limit imposed externally; (iii) the cluster exceeds the total memory budget in the cache space. Finally, the formed cluster is returned.

Cost-benefit model for joining a cluster. The goal of the cost-benefit model is to estimate the benefit of admitting a candidate node into a given cluster. Measuring the *exact* benefit of adding a node to a cluster of items requires going over the entire trace of user accesses to measure the frequency of *all* subsets of items. The resulting complexity would be *exponential* with the size of the cluster. Therefore, it is prohibitively expensive and unrealistic even for small datasets. The key idea of our approach is to exploit the item co-occurrence graphs to *estimate* the expected savings of a cluster without explicitly counting the frequency of all combinations. Our estimates rely on inclusion-exclusion rules in combinatorics [10]. This allows us to build lower and upper bounds on the frequency of larger tuples (triplets, quadruplets, and beyond) by only measuring the frequency of pairs (*i.e.* the number of co-occurrences). These lower and upper bounds on frequencies directly allow us to estimate the lower and upper bounds of the expected bandwidth reduction resulting from caching all subsets of a given cluster.

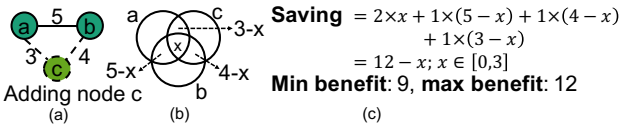

Figure 3: An example demonstrating the cost-benefit model of adding a node to an existing cluster.

We provide an intuitive explanation of our cost-benefit estimation using an example. In Fig. 3, suppose we are provided with a cluster that already contains items a and b, and our goal is to estimate the benefit of adding item c to the cluster. As depicted in Fig. 3(a), suppose items a and b are co-accessed 5 times, items b and c are co-accessed 4 times, and items a and c are co-accessed 3 times. However, note that the graph, since it encodes only pairwise relations, does not offer any information on how often all three items are accessed together. We can represent this information in the form of a Venn diagram where (a, b), (b, c), and (c, a) correspond to different sets, as depicted in Fig. 3(b). We assume that the intersection of three sets has x elements. Storing the partial sum of a, b & c, denoted by $psum(a, b, c)$, reduces the number of embedding fetches from 3 to 1 when all these items are accessed together. Storing the partial sums of pairs, on the other hand, would save one embedding fetch if the pair is co-accessed. Based on this knowledge, we can calculate the total savings of caching all pairs and the triplet as shown in Fig. 3(c) as a function of x. Given the number of co-accesses between $(a, b) = 5$, $(b, c) = 4$, and $(c, a) = 3$, the maximum frequency of (a, b, c) could be 3 and the minimum frequency of (a, b, c) could be 0. Therefore, caching all combinations of a, b, and c yields worst-case and best-case savings of 9 and 12, respectively.

Forming a cluster with nodes a, b, and c implies that we cache these embeddings and their partial sums: *emb(a)*, *emb(b)*, *emb(c)*, and additionally *psum(a, b)*, *psum(b, c)*, *psum(a, c)*, and *psum(a, b, c)*, *i.e.,* 4 additional cached partial sums. Consequently, the cost-benefit model estimates the maximum and minimum benefit of adding a node c to the cluster of nodes a and b would be 9/4 (`min_expected_saving` in Algorithm 3) and 12/4 (`max_expected_saving` in Algorithm 3). In practice, we observe that the exact benefit of adding a node to a cluster is around the midpoint of the maximum and minimum estimated benefits. We use a linear interpolation factor α between the lower and upper bounds of the benefit to estimate the cost efficiency of the proposed cluster as shown in Algorithm 3. §B.4 discusses the sensitivity of tuning of this estimation. GRACE uses a graph-based algorithm, which readily applies to multiple embedding tables (shown in §A.3).

4.3 A Walk-Through Example

To best understand the proposed algorithms, Fig. 4 shows a walk-through example of our ICG building and clustering phases. Fig. 4(a) shows the user-item interaction trace, where 5 different users are accessing unique items. In this example, we set the maximum cache capacity to 10 cached items, tolerance factor to 0.4, and α to 0.5. Note that our algorithms are not restricted to these parameters and can work for any parameter setting, these parameters are chosen for simplicity.

Fig. 4(b) shows the ICG that is formed as a result of shown user preference trace. In this example, the node IDs correspond to items

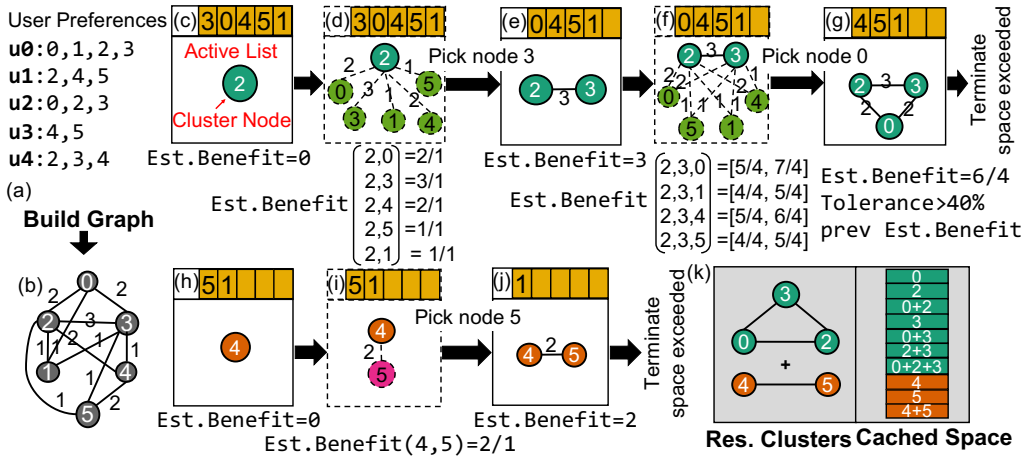

Figure 4: (a) User-item preference trace, (b) resulting ICG, (c-j) a walk-through example of the ICG clustering algorithm, and (k) resulting clusters and cached embeddings.

Algorithm 3 Pseudocode to estimate the cost benefit of a node joining an existing cluster

```
1:  procedure EstimateBenefit(G, cluster, candidate)        ▷ Offline
2:      Input: G: Item co-occurrence graph (ICG)
3:      Input: cluster: the current formed cluster so far
4:      Input: candidate: candidate node to join cluster
5:      Output: est_benefit: estimated savings per cache line
6:
7:      g = subgraph(G, cluster)
8:      // Construct g', the resulting cluster if candidate was added to g
9:      g' = g.add_node(G, candidate)
10:     // Estimate cost benefit by the creation of g'
11:     lower_bound = min_expected_saving(g')
12:     upper_bound = max_expected_saving(g')
13:     // g' is the resulting cluster if candidate was added, thus |g'| ≥ 2
14:     est_benefit = ((1−α)×lower_bound + α×upper_bound) / (2^(|g'|) − 1 − |g'|)
15:     return est_benefit
```

from 0 to 5. The edge weights of ICG represent the number of times items corresponding to its source and destination nodes are co-accessed. For example, items 2 and 3 are co-accessed by three users, *i.e.,* users 0, 2, and 4, hence, a weight of 3 is assigned to the edge between ICG nodes 2 and 3. This graph is a result of the ICG building phase, the next phase is clustering this graph.

The ICG clustering algorithm starts by assigning all nodes to the active list, and picking the first node to start forming clusters. As shown in Fig. 4(b), because node 2 has the highest degree (*i.e.,* item 2 is the most popular), the first node that starts building clusters is node 2 (Fig. 4(c)). Based on line 24 of Algorithm 2, all the neighbors of node 2 from the active list are picked to estimate the benefit–per–cached–space of adding them to an existing cluster. Based on the ICG connectivity, node 3 has the best estimated benefit of 3/1 for getting added to the cluster. Therefore, our algorithm picks node 3, and forms a cluster of nodes 2 and 3. Note that this cluster takes 3 cache spaces, which is less than the cache budget of 10. Therefore, this algorithm continues and it attempts to find new nodes to add to the same cluster.

As shown in Fig. 4(f), the cluster expansion continues by examining the neighbors of ICG nodes 2 and 3 to the existing cluster. Using nodes 0, 1, 4, and 5, the algorithm calculates the cost of adding each of these nodes to an existing cluster of nodes 2 and 3. The figure shows the range of benefits calculated by our algorithm, and using an α of 0.5, node 0 has the highest estimated benefit of 6 / 4 (the denominator of 4 is because the cluster of three nodes would consume 4 additional caching locations). Because this benefit is within a tolerance limit of the previously estimated benefit (*i.e.,* 6 / 4 > 0.4 × 3), node 0 is added to the cluster. At this point, 7 out of 10 cache spaces are claimed, and adding any more nodes to the cluster would result in more than 10 cache spaces. Therefore, this clustering algorithm terminates, and it picks up a new node 4 from the active list to form a fresh cluster. The result of this iteration of clustering is a 2-node cluster with nodes 4 and 5.

Fig. 4(k) shows the result of this clustering algorithm, where two clusters are formed with 2 and 3 nodes. It also shows the consumption of cache space taken by these two clusters. Here, 0 + 2 means the partial sum of items 0 and 2. Of note are two important details: (i) clusters can be of **different** sizes (size of 2 and 3 in this example); (ii) the partial sums of **all combinations** of items in a cluster are cached. The cache layout is carefully tailored to compute addresses easily (detailed in §5.3). In practice, the cache space budget is much higher, and this algorithm forms several clusters of different sizes.

4.4 Overhead Analysis

Complexity Analysis. Denote the number of users by m, and the average length of item interactions per user by p. The complexity of ICG construction (Algorithm 4) is $O\left(mp^2\right)$. Let n be the number of nodes (items) in ICG, d be the average degree per node, and k be the average size of a cluster. The complexity of a single evaluation of the cost model is $O\left(k^2\right)$. In Algorithm 2, the `while(true)` loop is iterated k times; each iteration makes d calls to the EstimateBenefit() function (Algorithm 3). Therefore the overall complexity of FormCluster() is $O\left(dk^3\right)$, executed $\frac{n}{k}$ times. Thus, the overall complexity of clustering the ICG is $O\left(ndk^2\right)$.

We highlight the following merits of our algorithmic framework: (i) the ICG construction phase is *linear* in the number of users; (ii) the clustering algorithm is *linear* in the number of items. This allows our approach to scale to a large number of users and items. The ICG clustering complexity is quadratic to k. Our evaluation shows that k goes up to 8 for the best DLRM performance, making the clustering algorithm practical.

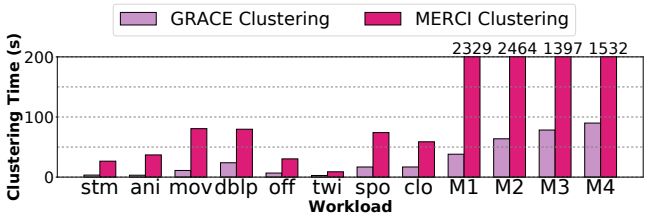

Figure 5: Clustering time comparison of GRACE and MERCI using a 128-thread implementation among different datasets.

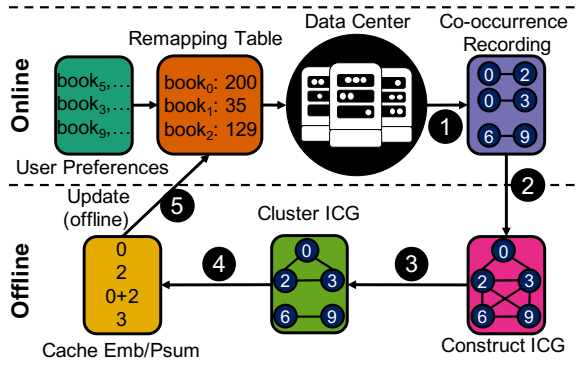

Figure 6: Usage model of GRACE.

Runtime Analysis. To evaluate the runtime overhead of GRACE clustering algorithm, we implement a parallel version of this algorithm in C++ using OpenMP. To best match our estimation to a data center deployment scenario, we run this clustering algorithm on a high-end server-grade CPU discussed in §6.2. Using a 128-thread implementation, Fig. 5 compares the clustering speeds of GRACE and MERCI. GRACE achieves 8.3× faster clustering on average among all datasets, and 26.6× among the mixed datasets that have a larger number of items. *This shows that the GRACE algorithmic framework meets one of its key goals, i.e., designing a practical and scalable algorithm.* With the low-cost scalable clustering algorithm, GRACE can adapt to frequent user-item preference behavior changes even at an update frequency of hours.

5 GRACE SYSTEM DESIGN

The algorithmic framework of GRACE is generic and can apply to various types of memory systems. Here, we consider the use case of a CPU-GPU heterogeneous system and present GRACE system design. Our system modeling choice is motivated by the fact that this type of system is widely adopted in today's data centers that execute DLRMs [25, 45, 63, 76].

5.1 Usage Model

Fig. 6 depicts a high-level overview of the usage model of GRACE. It consists of online profiling of user-item interactions, offline ICG construction, clustering, and populating the cache space with partial sums of clusters. GRACE is immediately deployable on commodity hardware platforms. In what follows, we detail the GRACE online and offline components.

Online profiling. While running DLRM inference in a data center, GRACE samples a subset of users, and records their item interactions. Specifically, it records which items are accessed together (*i.e.*, pairwise item co-occurrence recording as depicted in Fig. 6) and lazily updates the ICG. The lazy nature of graph updates means that the incoming edges to the graph can be buffered and processed at a later point in time. This ensures that the recording phase does not interfere with the performance of the ongoing DLRM inference.

Offline analysis. As presented in Fig. 6, GRACE collects the edges recorded during the online profiling phase, and constructs the ICG offline. The constructed ICG is then clustered to find frequently accessed item combinations as discussed in §4. For each cluster, GRACE identifies and fetches the embedding vectors of the constituent nodes, computes *psums*, and caches embedding vectors and *psums* into the cache space. GRACE generates clusters in decreasing order of expected cost-benefit efficiency, as shown in Algorithm 1. It then stores clusters of *psums* accordingly to the GPU and then the CPU. As shown by prior industrial [63] and academic [1, 34, 48] works, DLRMs employ a remapping table to keep track of cached items. GRACE re-purposes this remapping table to reflect the cached item set. More details on how to filter cached and non-cached accesses, and how to determine the addresses of *psums* are presented in §5.3. Note that the clustering of ICG and computing/caching *psums* does not affect DLRM inference latency as they are carried out offline. Using *psums* does not change the reduction results; GRACE does not affect DLRM inference accuracy.

Justification of the usage model. As shown in prior works [50], data center operators typically employ feedback-driven [11] and post-link optimizations [47, 56] to improve the performance of their workloads. In the case of DLRM workload, for example, a data center operator like Meta may profile and record user-item interactions for a week, analyze them offline to create a cache space, and deploy the updated system for inference in subsequent weeks. Several prior works [5, 11, 39–41, 56] have successfully demonstrated that profile-guided techniques, similar to GRACE, are practical and they are deployed in data centers today.

5.2 Heterogeneity Awareness

One of the key goals of GRACE is to achieve a **heterogeneous memory-aware** framework that also optimizes for high aggregate memory utilization. To compare, MERCI optimizes for a single metric of maximizing the memory traffic reduction. Large clusters formed by MERCI, while yielding a greater bandwidth reduction, prevent most embeddings from being stored in a capacity-limited cache space. In that case, the main memory becomes the throttling bottleneck in processing user requests, and it prevents a high heterogeneous memory utilization.

To combat this, GRACE can be tuned to form appropriately sized clusters to store a greater diversity of item embeddings and combinations to effectively use the cache space. GRACE uses the parameters `MAX_CLUSTER_SIZE` and `tolerance_factor` discussed in Algorithm 1 to navigate the complex trade-off space of memory traffic reduction and balance of the heterogeneous memory bandwidth. To be able to apply GRACE to arbitrary input and get the best speedup, GRACE uses a lightweight decision engine to find the best parameter given dataset characteristics. We use three features from the dataset input: *number of items*, *average pooling factor*, and *average node degree in ICG* to train the decision engine. We use the decision tree implementation from scikit-learn [60] library and use an optimized CART (Classification and Regression Trees) algorithm. The decision engine can pick the optimal or near-optimal combination of maximum cluster size and tolerance without exhaustively

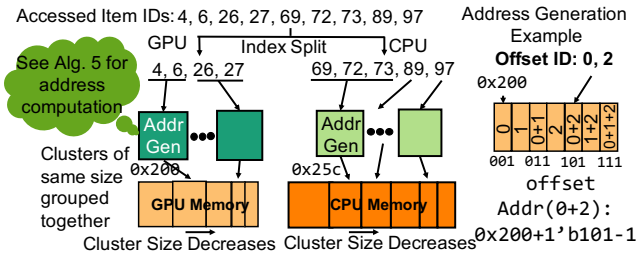

Figure 7: Example of GRACE cache space layout and address generation. User access IDs are compared with the starting address of each unique length of the cluster group to find any clustered accesses. Those clustered accesses have their partial sum ready in the memory, the address of which can be directly computed based on their IDs.

experimenting with all possible combinations. The decision engine accuracy performance is detailed in Appendix §C with a full sweep of different maximum cluster sizes and tolerance combinations shown in Fig. 20.

5.3 Address Generation

As detailed in §5.1, GRACE constructs and clusters an ICG, and stores *psums* into a cache space offline. To efficiently use these *psums* to improve end-to-end performance, it is crucial to design an *efficient* cache address computation logic. To this end, we propose a cache data layout and corresponding address generation technique for efficient GRACE system design. The goal of our designed combined cache data layout and address generation is to compute the address based on the accessed user index at an extremely low cost in software. Fig. 7 shows the proposed layout of cached data, where the clusters of the same sizes are grouped together and laid out adjacent to one another in the address space. Notably, GRACE employs a software-managed cache space to avoid hardware additions to commercial hardware platforms. Because software injects cache lines offline, it does not disturb the ongoing inference cycles. To understand cache layout and address generation with a simple example, assume that the largest size cluster is 4 nodes. GRACE first stores all 4-node clusters, then 3-node clusters, and so on. The item IDs are remapped in the order of clusters. Algorithm 5 (see Appendix §A.2) presents the pseudocode for generating redirected addresses. Given the embedding index of the accessed item, the address generation logic can quickly derive whether the index belongs to the CPU or the GPU. The memory location of the cluster, the cluster size, cluster ID, and offset within the cluster are used to determine specific *psum* addresses (Fig. 7 right). With the above cache layout, the user index only needs to compare with the starting address of each unique length of the cluster group (in practice at most 8 entries) and then compute the address of the embedding vector/partial sum without accessing additional data structures.

5.4 End-To-End System Design

Fig. 7 presents the end-to-end system execution of GRACE. The *psums* are pinned in the CPU and GPU memory using GRACE software. Similar to an earlier work, we execute on a heterogeneous CPU/GPU system [14], and we use 1 GB super pages to avoid any paging overhead. GRACE re-purposes the remapping table [1, 34, 48, 63] in software to process the incoming user requests of embedding layers. The remapped and sorted indices split the requests to either CPU or GPU. With a software-defined cache space, the item indices can directly translate into the cluster ID and offset within the cluster. Using Alg. 5, the redirected addresses are used to correctly serve the requested indices with *psums* in the heterogeneous memory. The item embedding reduction is then executed on both CPU and GPU simultaneously before the CPU results are sent and reduced with the GPU results (see Fig. 1(b)). Each batch synchronously reduces item embeddings and computes sparse features on GPUs before processing the top MLP layers. The address generation process of the batch of users is overlapped with the item embedding reduction of the previous batch to ensure that address generation is not on the critical path. We show in §7.1 that the latency of address generation is negligible compared to embedding reduction time.

6 METHODOLOGY

6.1 Real-World Datasets

We use a variety of datasets from different web service vendors, shown in Table 1. We choose datasets of different sizes and average pooling factors. The average pooling factor of a dataset is defined as the number of items, on average, reduced by each user to compute sparse embeddings.

Table 1: Real-world datasets from web service vendors.

Category	Dataset Name	Avg. Pool. Factor	#Items
Small	Steam (stm) [36, 59, 71]	71.8	10,978
	Anime (ani) [2]	106.3	11,200
	MovieLens20M (mov) [27]	144.4	26,744
Medium	DBLP (dblp) [62]	61.8	540,459
	AmazonOffices (off) [29]	64.0	598,943
	Twitch (twi) [61]	30.5	739,991
Large	AmazonSports (spo) [29]	96.1	1,505,707
	AmazonClothes (clo) [29]	82.0	2,345,346

In addition to evaluating uniform datasets, we also present an evaluation with a mixture of the datasets to model the real-world recommendation system that has multiple embedding tables of different sizes and properties. Table 2 lists the mix of our datasets. For each dataset, we split by 50:50 ratio to profile the behavior and estimate inference performance (we sweep the training/test ratio in §B.3). We use the embedding dimension of 1024 and the user batch size of 1024. For end-to-end speedup analysis, we use DLRM models in Table 3.

Table 2: Experimented mixture of datasets.

Dataset Name	Mixture of Dataset	Classes
M1	twi-mov-ani-stm	1M+3S
M2	clo-off-dblp-ani	1L+2M+1S
M3	spo-off-dblp-twi	1L+3M
M4	clo-spo-off-dblp	2L+2M

6.2 System Configuration

For sampled user traces, we build ICG and form clusters using the GAPBS [6] framework. We deploy all the inference tasks on a high-end server and measure the performance. For a heterogeneous memory system deployment, we use an Intel Xeon Platinum 8380 CPU with 80 physical cores and 512GB 32-channel DDR4-3200 main memory as the CPU host. We use NVIDIA A40 with 48 GB GDDR6

memory as the GPU device. The embedding table reduction operation executed on CPU uses AVX-512 instructions, same as [25]. We use OpenMP to parallelize multiple embedding-reducing operations from different users and we verified the full utilization of CPU bandwidth. We also verified the correctness of the embedding layer functionality since GRACE does not alter any reducing results of embedding layer operations.

While GRACE presents a generic algorithmic and system design framework to improve DLRM inference throughput, we evaluate GRACE and prior works [34, 48] using a heterogeneous CPU-GPU system for a fair comparison. GRACE, however, can be generalized and adapted to any heterogeneous memory configuration with a main memory and cache space.

Table 3: DLRM models for end-to-end performance analysis.

DLRM Model	Bottom MLP	Top MLP	Num. of table
RM1 [24, 26]	128-64-32	256-64-1	8
RM2 [24, 26]	256-128-64	128-64-1	32
RM3 [24, 26]	2560-1024-256-32	512-256-1	10
RM4 [78]	-	200-80-2	3

6.3 State-of-the-Art Baselines

Infinite GPU memory. This solution models an infinite GPU memory capacity that can host full embedding tables in GPU memory, regardless of size. It does not store any *psums*.

CPU only. This baseline models hosting full embedding tables in the CPU memory. We use all 80 cores for executing DLRM.

Off-the-shelf clustering techniques. We use the state-of-the-art graph clustering algorithm Metis [46] that can apply to ICG clustering. Metis uses a recursive k-way multi-level graph partitioning algorithm to form clusters.

FAE [1]. We model an ideal performance of FAE that places a subset of highly accessed item embedding vectors in a cache space. There is no memory traffic reduction mechanism in FAE. To report the optimal performance of FAE, we sweep every possible cut-off frequency value for each dataset separately. The performance is also an indicator of the upper bound performance of utilizing the heterogeneous memory without reducing the memory traffic.

SPACE [34]. SPACE is a state-of-the-art recommendation system inference framework that uses static analysis of user preferences to find popular items. It significantly outperforms other hybrid DRAM management frameworks [15, 16, 65]. SPACE caches the single popular items and an exhaustive set of combinations of two items in GPU memory. SPACE offers a rich design space in terms of the fraction of single versus partial embedding sums stored in the cache space. We report the results for the best-performing parameter setting by extensively sweeping the value of this fraction.

MERCI [48]. MERCI is a state-of-the-art framework to generate clusters of *psums* to reduce the memory traffic. Although MERCI is proposed for a DIMM-only system, we assume that their clusters of *psums* are stored in GPU memory. We use the open-source implementation [3] from authors. Additionally, we navigate the subgroup size-performance trade-off in MERCI to find the best-performing parameters and report the optimal performance numbers.

Oracle-of-2. We model an oracle with a *psum of 2* that can find partial sums of **any** 2 item embeddings. While an oracle-of-3 or larger is possible in theory, we choose the oracle-of-2 because it provides a reasonable reduction factor roofline to compare the modeled systems. For this oracle, we assume that it perfectly balances the memory bandwidth in a heterogeneous memory system (§6.2).

7 EVALUATION RESULTS

7.1 Performance Analysis

GRACE vs. prior works. Fig. 8 compares the embedding layer throughput of GRACE with CPU only, Metis clustering [46], FAE [1], SPACE [34], MERCI [48], and an Oracle-of-2 normalized to an infinite GPU memory solution. GRACE and prior works use extra memory capacity to store embedding vectors/partial sums that is equal to 1× the size of the original embedding table. The figure shows that CPU slows down the execution by 3.7× compared to the infinite GPU memory baseline. This is because the compute throughput and peak memory bandwidth of GPU are much higher than CPU. FAE achieves 1.1× better performance than the baseline. We sweep every possible cut-off frequency for each dataset separately and report the best performance. FAE only marginally improves the performance of the baseline because of no memory traffic reduction. Moreover, the figure shows that, on average, the off-the-shelf clustering algorithm (Metis) only achieves 0.34× the baseline performance. The slowdown is attributed to the fact that a generic clustering algorithm does not contribute to partial sum reductions or heterogeneous memory utilization effectively. **This clearly motivates the design of a novel ICG clustering algorithm to best accelerate DLRM inference.**

Fig. 8 shows that GRACE outperforms SPACE and MERCI by 1.5× and 1.4×, on average. As discussed in §6.3, we report the best performance of SPACE and MERCI based on extensive parameter tuning. This significant performance improvement is attributed to the GRACE algorithm and system design that (a) finds popular item combinations of arbitrary lengths in a scalable fashion, and (b) is memory heterogeneity aware by effectively navigating the trade-off of expanding the partial sum sizes or storing more item embeddings. GRACE effectively bridges the performance gap between prior works and oracle-of-2 by 52.1%. The performance of SPACE and MERCI are limited due to limited reduction in memory traffic and memory heterogeneity unawareness, respectively. *This, in turn, shows that GRACE finds high-quality popular item combinations to effectively reduce the memory traffic, and rejects over-sized clusters to prevent DRAM throttling. This optimizes the collective heterogeneous memory bandwidth.* Interestingly, we find that GRACE performance is positively correlated with the average ICG node degree. The best-performing datasets over prior works (stm, ani, mov, and M1) have the highest ICG average node degrees of (1405, 1148, 2107, and 900).

Heterogeneous memory time split. To further understand the performance of different baselines, Fig. 9 shows the time split for the embedding reduction in CPU and GPU memories. Ideally, a system that splits memory traffic to balance the execution times spent on CPU and GPU can achieve high throughput.

SPACE moderately reduces the GPU execution time by 15% compared to a system with infinite GPU memory, which determines the overall throughput of SPACE. Because GPU has a much higher memory bandwidth available compared to CPU, a moderate memory traffic redirected from GPU to CPU memory will result in a significant increase in CPU memory time. *This result underscores*

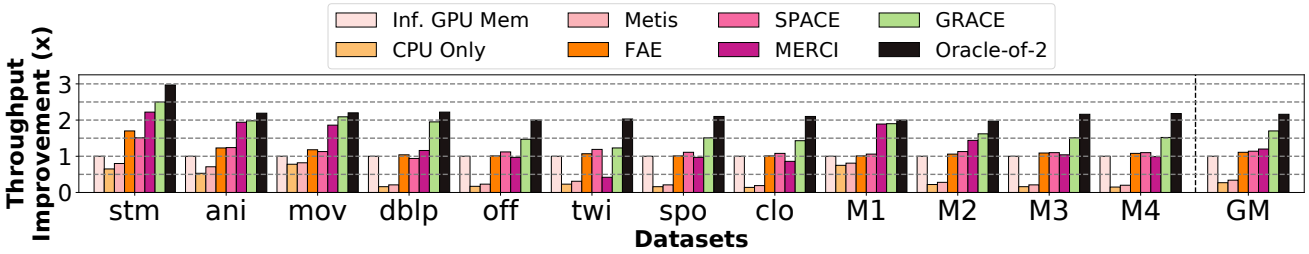

Figure 8: Embedding layer throughput of Infinite GPU Memory, CPU only, Metis [46], FAE [1], SPACE [34], MERCI [48], GRACE and Oracle-of-2 normalized to Infinite GPU Memory. All works use 1× additional table capacity to store partial sums.

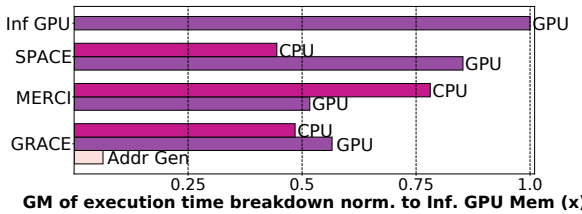

Figure 9: Embedding layer execution time breakdown across CPU and GPU memory, averaged among all datasets, normalized to the infinite GPU memory execution time. Ideally, the memory system achieves a balanced execution.

the value of achieving high memory traffic reduction to speed up the workload. MERCI performance, on the other hand, is determined by the embedding reduction time on the CPU. This is because reducing memory traffic is the sole design objective of MERCI, which leads to large cluster sizes. In a real-world heterogeneous memory setting, this leads to spilling of many embedding psums to CPU memory, inadvertently increasing its execution time.

The GRACE design effectively navigates the complex design space of reducing memory traffic by storing large clusters versus distributing more memory traffic to the heterogeneous memory system. Fig. 9 shows that GRACE achieves a near-perfect execution time split between CPU and GPU to maximize overall application throughput. The address generation time is shown separately because it refers to the latency to compute memory addresses and distribute psums/item embedding requests to CPU and GPU. This is off the critical path of the embedding reduction latency as it can be pipelined with the previous batch reduction. Finally, the figure also shows that the address computation time is negligible (6%) compared to the time to load and reduce item embeddings. The GRACE runtime system thus can fully hide address computation latency by overlapping it with the embedding reduction of the previous batch.

Memory traffic reduction. Fig. 10 shows the comparison among different baselines. The oracle-of-2 achieves a 50% reduction in the memory traffic as it stores psums of all two-item embeddings (not practical). SPACE only reduces memory traffic by 9% because it stores partial sums of two item combinations of a very small subset of items. MERCI and GRACE can reduce memory traffic by 37% and 40%. Note that dblp has a memory reduction factor of 2.3× for GRACE, which is higher than the modeled oracle-of-2. Although MERCI inspects full training traces to generate optimal clusters, the high complexity of such inspection forces the algorithm to break into sub-groups. During this process, co-accesses between different sub-groups are ignored and MERCI may miss the opportunity to analyze a global set of co-accessed items. GRACE, on the other hand, does not have such constraints and the ICG captures accesses of all items. *This result also shows that even in a traditional DIMM-only memory system, GRACE results in higher memory reduction and outperforms MERCI.*

Tail latency comparison. Fig. 11 shows the 95th percentile latency of processed batches of compared works, normalized to the infinite GPU memory solution. The figure shows that GRACE consistently outperforms the state-of-the-art works in terms of tail latency as well as throughput. Specifically, GRACE improves SPACE by 1.54× and MERCI by 1.41×.

End-to-end DLRM performance. By speeding up the embedding reduction phase, GRACE also significantly improves the end-to-end throughput of DLRM. Fig. 12 shows that GRACE offers a significant end-to-end performance improvement of 1.6× over infinite GPU memory on embedding-heavy models such as RM2. In MLP-heavy models such as RM3, GRACE achieves 1.2× speedup, outperforming prior works. DLRMs are executed at a population scale. Even a single percent performance improvement in data center applications leads to a significant reduction in Total Cost of Ownership (TCO) and global carbon footprint [5, 35]. DLRMs consume more than 60% of AI inference cycles [26]. Fig. 12 shows that GRACE provides significant end-to-end performance improvement of 1.2-1.6× compared to MERCI and offers a low-cost solution, obviating intrusive hardware modifications. Therefore, GRACE can be immediately adopted in today's data centers.

Understanding the improvements over MERCI. MERCI is the state-of-the-art framework aiming at memory traffic reduction in DLRM. Interestingly, we observe that GRACE outperforms MERCI in both memory traffic reduction and end-to-end throughput. The reason behind this improvement is twofold. 1) GRACE has a global view of user-item interactions, irrespective of the dataset size. This is enabled by our novel graph construction that allows scalable analysis at a global dataset scale. §4.4 shows that GRACE analysis scales from both algorithmic complexity and runtime standpoints. MERCI's analysis, on the other hand, has a fundamental limitation that it operates at a sub-group level and fails to capture a global view of user-item interactions due to scalability issues. 2) While MERCI only aims to improve memory traffic reduction, the proposed GRACE algorithm is memory heterogeneity-aware. GRACE not only improves memory traffic reduction, but also results in a balanced memory traffic distribution further improving end-to-end throughput.

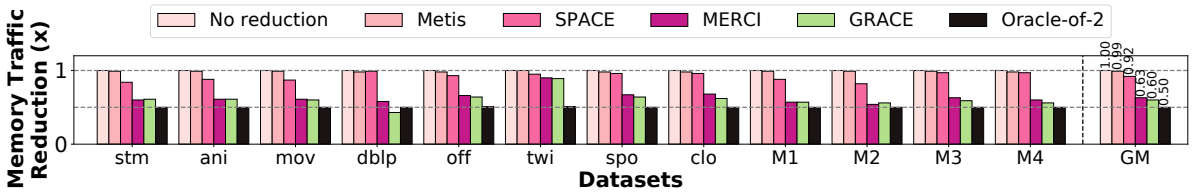

Figure 10: Memory traffic reduction in DLRM inference of compared works (lower is better).

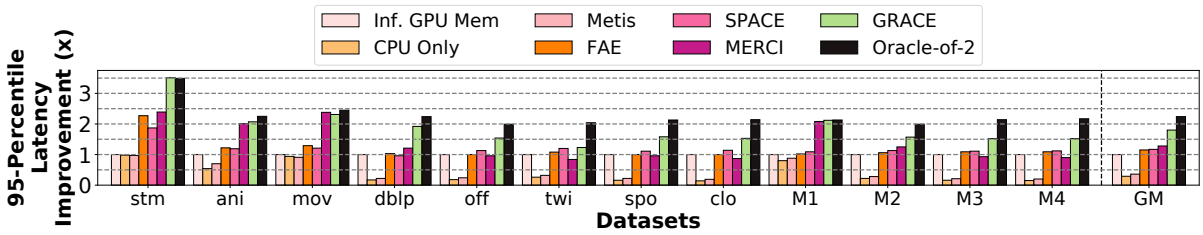

Figure 11: Embedding layer 95th percentile latency of compared works normalized to Infinite GPU Memory.

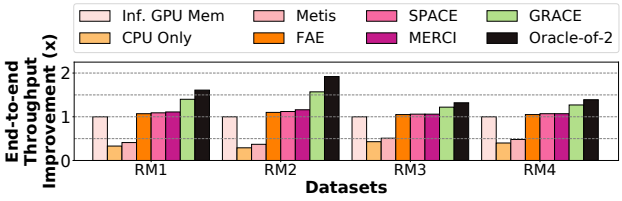

Figure 12: End-to-end DLRM inference performance of compared works normalized to Infinite GPU Memory.

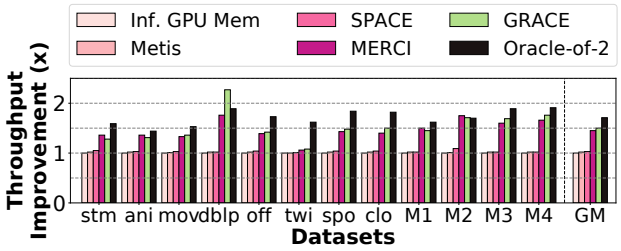

Figure 13: Embedding layer throughput comparison in a homogeneous GPU memory platform.

Comparison using additional hardware configurations. GRACE algorithm-system co-design is agnostic to any specific hardware configuration. While a CPU-GPU platform represents a baseline modeling for a majority of our evaluation, next, we show the performance of GRACE using two other hardware platforms.

First, we compare the performance of various baselines on a homogeneous GPU memory in Fig. 13. This experiment assumes an infinite GPU memory. Because this platform does not have heterogeneous memory, the performance is directly correlated with memory traffic reduction. GRACE outperforms MERCI marginally by 5%. This corroborates with the traffic reduction ratio in Fig. 10. Furthermore, GRACE significantly outperforms Metis and SPACE due to improved memory traffic reduction.

Second, Fig. 14 shows the embedding layer throughput improvement on a system having hybrid DIMM-HBM memory with Process-In-Memory (PIM) technology (evaluation similar to SPACE [34]). We simulate the embedding reduction operations in PIM using a trace-based simulation methodology in Ramulator [42]. The modeled DRAM consists of 8 DDR4-3200 channels and 2 stacks of bandwidth-optimized cache space of HBM2 (specifications adopted from [53, 55]). Fig. 14 shows an interesting trend that FAE outperforms SPACE. This shows that heterogeneous memory awareness is increasingly important when the bandwidth capability varies across different platforms. The figure also shows that the memory heterogeneity-aware design of GRACE can adapt to different technological parameters, and consistently offer the best performance compared to the state-of-the-art. By optimizing memory traffic reduction and distribution, GRACE outperforms MERCI by 1.5×.

Table 4: Absolute throughput numbers (*i.e.*, #batches processed per second) of an infinite GPU memory baseline.

Dataset	stm	ani	mov	dblp	off	twi
Throughput	976	1125	668	1594	2787	4965
Dataset	spo	clo	M1	M2	M3	M4
Throughput	1924	2332	361	492	667	583

Absolute performance numbers. To enable better reproducibility of results and future comparison with GRACE, Table 4 shows the absolute throughput numbers of an infinite GPU memory baseline as shown in Fig. 8. This result is obtained by running a full embedding layer on the GPU platform (§6.2). The reported absolute numbers are of the same order as a recent industrial work DeepRecSys [24] (our baselines are more optimistic than DeepRecSys as our embedding layer execution baseline is natively optimized using CUDA/C++ compared to PyTorch). All other absolute numbers reported by our compared works can be inferred by scaling the absolute numbers with speedups shown in the figures.

7.2 Sensitivity Analysis

GPU memory capacity. In practice, the capacity budget for the high bandwidth memory can be less than 1.0× because of the capacity-limited GPU memory and large embedding table sizes. Multiple embedding tables also share the GPU memory capacity resources. To measure the effectiveness of GRACE in a more constrained environment, we sweep the allowed cache space capacity to 0.5 × and 0.25 × embedding table size. Fig. 15 its effect on embedding reduction performance. With more constrained cache space,

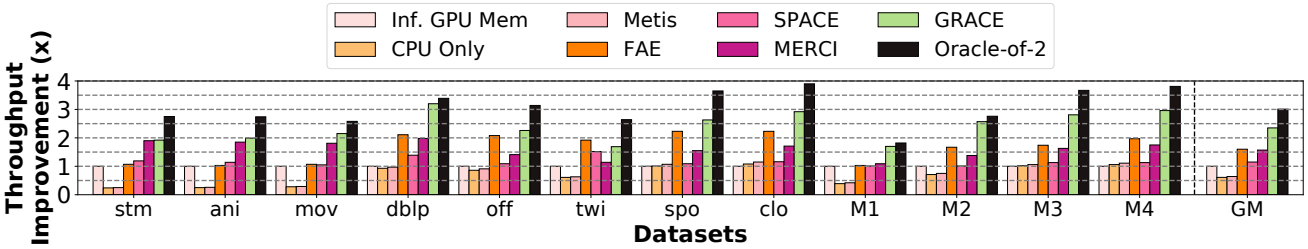

Figure 14: Embedding layer throughput comparison for a DIMM-HBM heterogeneous memory with PIM capability.

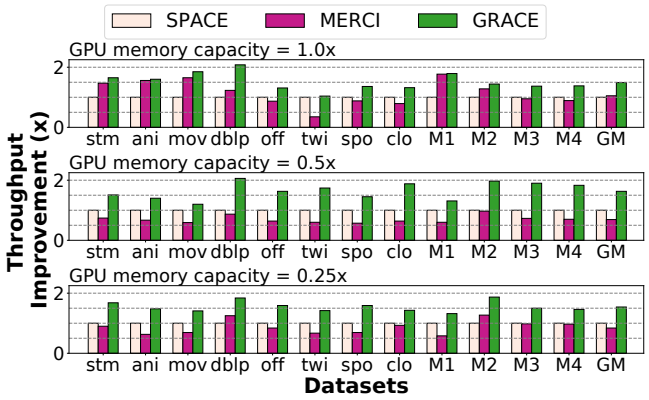

Figure 15: Performance sensitivity of compared works for different GPU memory capacities normalized to SPACE.

MERCI further exacerbates DIMM memory throughput by using the constrained space for storing *psums*. GRACE , however, can adapt to the more constrained space by rejecting clusters at an earlier threshold. GRACE balances the throughput of the heterogeneous memory system. On average, GRACE outperforms SPACE by 1.63× and 1.54× in the tested configurations of constrained cache space.

7.3 Additional Results

Extensive experiments and further insights on energy analysis and sensitivity studies are presented in detail in Appendix §B.

8 RELATED WORK

Profiling of recommendation systems [24, 26] shows that the embedding layer accounts for more than 25% and 80% of the inference latency in the Meta RM1 and RM2 model. These RM models consume more than 60% of Meta's data center AI inference cycles. While using specialized DNN accelerators or employing batched inference can significantly [9, 12, 17, 23, 30–33, 51] improve the DNN layer throughput, the embedding layer performance is still bottlenecked by the memory bandwidth [24, 26, 28, 34, 38, 43, 44, 76].

Exploiting the embedding table locality. Analysis of the user-item interactions has been studied in prior works [1, 7, 20, 34, 37, 38, 43, 58, 63, 64, 68, 71, 75], such as reducing the dimension and exploiting the power-law characteristics observed in the embedding table operations. SPACE [34] is the most recent work that exploits both singular hot items and exhaustive combinations of partial sums of two hot items. MERCI [48] captures the most efficient *psums* to reduce the memory traffic. GRACE explores the design space of both memory traffic reduction and heterogeneous memory utilization and GRACE analysis is derived by analyzing real-world service vendors instead of artificially generated distribution [24, 25].

Near memory processing and memory technology for improving embedding table operations. Near memory processing is explored in many prior works [38, 43, 57, 69, 70]. They serve as the heterogeneous memory module and significantly increase the available memory bandwidth. Fafnir [4] uses a tree-like reduction hierarchy among different ranks to improve the reduction efficiency near the memory logic. Other memory technologies have been studied in prior works including SSD (solid-state drive) [67, 73] and NVM (non-volatile memory) [19] to aid the embedding layer operation, which has a dual challenge of large capacity and a high bandwidth requirement. GRACE assumes a hybrid DRAM model and uses GPU memory as a software-managed cache and this algorithm framework can generalize to these new memory technologies.

9 CONCLUSION

This paper proposed GRACE—a novel algorithm-system co-design framework to significantly accelerate DLRM inference by speeding up the embedding reduction stage. To reduce the memory traffic of sparse DLRM layers, GRACE proposed mapping the problem of finding popular item combinations to a graph problem. GRACE presented an Item Co-occurrence Graph (ICG) to scalably analyze popular item combinations. GRACE then proposed a low-cost graph clustering algorithm that finds popular item combinations of *arbitrary lengths* and inserts these frequently accessed item combinations into a software-managed cache space. The GRACE runtime system exploited partial embedding sums to significantly reduce memory traffic. Our evaluation showed that GRACE significantly outperforms state-of-the-art prior works SPACE and MERCI by 1.5× and 1.4×, respectively.

10 DATA AVAILABILITY STATEMENT

The code of this work is also available on Zenodo [74].

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd Laurent Bindschaedler for their insightful feedback. The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This work was also supported by the United States-Israel BSF grant number 2020135.

A APPENDIX: ADDITIONAL ALGORITHMS

A.1 ICG Construction Algorithm

Algorithm 4 presents the pseudocode of the graph construction phase into ICG. All pairs of items accessed by the user as item co-occurrences. We use an item co-occurrence buffer to construct a weighted graph by increasing the edge-weight by one for each co-occurrence.

Algorithm 4 Pseudocode for item co-occurrence graph (ICG) construction

```
procedure BuildICG(user_accesses)                                  ▷ Offline
    Input: user_accesses: Historical data of item accesses by sampled users
    Output: G: Item co-occurrence graph (ICG)

   recorded_edges = []
   for all user_access ∈ user_accesses do                          ▷ Online lazy recording
      num_items = user_access.size()                               ▷ #items accessed by user
      // For each user access, iterate over all unique item pairs
      for i in (0, num_items) do
         for j in (i + 1, num_items) do
            item_i = user_access[i]; item_j = user_access[j];
            // Lazy buffering of graph edges
            recorded_edges.append(edge(item_i, item_j))

   Initialize empty graph G
   for edge in recorded_edges do                                   ▷ Offline graph construction
      if edge not in G  then
         G.add_weighted_edge(edge, 0)
      G.increment_edge_weight(edge)
   return G
```

A.2 Address Generation Algorithm

We detail the address generation process in Algorithm 5. One user from a batch accesses remapped indices. The remapped indices are sorted, streamed in, and compared with the starting address of grouped clusters on CPU and GPU. After finding the group, the index can directly translate into cluster ID and offset within the cluster because clusters are grouped by size.

If the access results in a miss to the previous cluster, the address of accessing the single-item embedding at `item_offset` is located at $2^{\texttt{item\_offset}} - 1$ from the starting address of the cluster. If the access is a hit of the previous cluster, it indicates that *psum* is accumulated previously. Because the memory layout of embeddings and partial sums is in a bitmap fashion within a cluster, one-bit activation will direct to the address with the *psum* that accumulates the hit index. The new *psum* address is generated by adding a $2^{\texttt{item\_offset}}$ to the previous item embedding/*psum* address.

After directing all access indices to the proper address across the heterogeneous memory, the corresponding embedding items/partial sums are reduced to satisfy the batch of users' embedding layer requests.

A.3 Multiple Embedding Table Support

Real-world DLRM models have multiple embedding tables that share the available capacity-limited HBM resources. We propose Algorithm 6 to design cache space for multiple embedding tables. Similar to previous industrial proposals [24, 43, 54, 78], we assume that the embedding tables are independent, *i.e.*, items in an embedding table do not reduce with items in a different table. Algorithm 6 shows the clustering pseudo-code to support multiple embedding tables. The procedure only differs when building the ICG. Because different embedding tables have *mutually exclusive* nodes, for each item index, we add an offset of the previous table size to avoid duplication of the same item ID in different embedding tables. We build the ICG accordingly and the resulting ICG has a collection of all nodes of different embedding tables.

With this ICG, the problem of cluster forming for all embedding tables is evaluated in the same graph using the same ClusterICG algorithm in Algorithm 2. In this ICG, we can systematically analyze the clustering benefit-cost efficiency and assign the HBM capacity

Algorithm 5 Pseudocode for Address Generation on User Accesses

```
procedure AddressGen(user_accesses)                                ▷ Online
    Input: user_accesses: Incoming data of item accesses by runtime users
    Input: start_addr: Starting address of each cluster size group
    Output: redirected_accesses: Redirected addresses of item embeddings and partial sums
that users request for correct reduction

   prev_c_id = None
   prev_g_size = None
   temp_addr = None
   redirected_accesses = []
   for all access ∈ user_accesses do
      // Find which group, cluster, and whether CPU/GPU the access belongs to
      g_size = access.get_g_size();
      c_id = (access - start_addr[g_size]) / g_size;
      item_offset = (access - start_addr[c_id]) % g_size;
      // If access has a diff. cluster from prev access, commit prev redirected addr
      if c_id != prev_c_id or g_size != prev_g_size then
         redirected_accesses.append(temp_addr);
         temp_addr = start_addr[g_size] + c_id × (2^g_size − 1)
         temp_addr += (2^item_offset − 1);
      else
         temp_addr += (2^item_offset);
      // Accumulate to the redirected addr to get item embeddings / partial sums
      prev_c_id = cluster;
      prev_g_size = g_size;
   redirected_accesses.append(temp_addr);
   return redirected_accesses
```

Algorithm 6 Pseudocode for building the item co-occurrence graph (ICG) for multiple embedding tables

```
procedure BuildMultiTableICG(user_accesses_per_table)              ▷ Offline
    Input: user_accesses_per_table: Historical data of item accesses by sampled users for
each embedding table
    Output: G': Unified item co-occurrence graph (ICG) of multiple embedding tables

   Offset = []
   item_id_offset = 0
   for all t ∈ embedding_tables do                 ▷ Item renaming for each embedding table
      Offset.append(item_id_offset);
      item_id_offset += total_size_emb[t];

   recorded_edges = []
   for all t ∈ embedding_tables do                 ▷ Iterate through multiple tables
      for all user_access ∈ user_accesses_per_table[t] do
         num_items = user_access.size()                            ▷ #items accessed by user
         // For each user access, iterate over all unique item pairs
         for i in (0, num_items) do
            for j in (i + 1, num_items) do
               item_i = user_access[i] + Offset[t];
               item_j = user_access[j] + Offset[t];
               // Lazy buffering of graph edges
               recorded_edges.append(edge(item_i, item_j))

   Initialize empty graph G'
   for edge in recorded_edges do                                   ▷ Offline graph construction
      if edge not in G'  then
         G'.add_weighted_edge(edge, 0)
      G'.increment_edge_weight(edge)
   return G'

procedure ClusterMultiTableICG( )                                  ▷ Offline ICG clustering
    Input: G': Item Co-occurrence Graph (ICG) for multiple embedding tables
    Input: renamed_nodes: Vertex set of G' sorted by their degrees
    Input: capacity_budget: Number of cache lines allowed in cache space
    Output: cluster_list: Assignment of ICG nodes of multiple tables
   // The same clustering algorithm directly applies on the renamed nodes.
   cluster_list = ClusterICG(G',renamed_nodes,capacity_budget)
   return cluster_list
```

budget to each of the embedding tables. No heuristics are required to find a proper distribution of capacity for each embedding table. This shows the benefit of casting such a problem into a graph problem. Different embedding tables equivalently become disjoint partitions of an overall ICG, and can be evaluated altogether.

B APPENDIX: ADDITIONAL RESULTS

B.1 Performance Analysis: MERCI Variants

One of the reasons GRACE outperforms MERCI is that GRACE imposes a limit on the maximum size of the cluster. As shown in Fig. 9, this can effectively limit large clusters and allows a larger number of item embeddings/psums to be placed on the size-limited GPU memory. This way, the GPU memory utilization increases, achieving a more balanced distribution of memory traffic. This motivates us to test variants of MERCI that can benefit from a similar advantage. Fig. 16 presents the performance of MERCI-k variants. Here, MERCI-k limits the maximum cluster size to k. We sweep k from 2 to 5, and k equal to infinity (equivalent to vanilla MERCI design: *MERCI-inf*). Fig. 16 shows that GRACE consistently outperforms all MERCI variants. While *MERCI-k* improves the memory traffic distribution compared to *MERCI-inf*, this further limits the memory traffic reduction. Fig. 16 shows that, compared to a MERCI variant that optimizes heterogeneous memory utilization, GRACE stills offers improved throughput by balancing both memory traffic reduction and distribution.

B.2 Energy Consumption Analysis

Fig. 17 compares the energy saving of different baselines normalized to the infinite GPU memory solution. Because GPU memory consumes much less energy per data transfer byte than DIMM-based memory [49, 53], infinite GPU memory achieves low power consumption. GRACE achieves the best energy consumption compared with other works and even performs marginally 4% better energy consumption than infinite GPU memory. This is because although GRACE keeps some embeddings in DIMM-based memory to exploit a larger collective bandwidth for the best performance, it efficiently utilizes *psum* to serve requests, which saves energy both on CPU and GPU. Note that if GRACE is specifically configured to optimize for energy consumption, *GRACE-Energy* can save 18% energy compared to an infinite GPU memory configuration.

B.3 Sensitivity Analysis

Training ratio sensitivity. Fig. 18 shows that GRACE consistently outperforms prior works even using a limited training set to learn popular item combinations. Because every single user-item interaction can produce multiple co-accessed patterns, even though a limited set of the profiled user-item interaction, GRACE can extract co-access patterns effectively. GRACE achieves even a higher speedup at a limited training set, achieving 1.66× and 1.46× over SPACE and MERCI respectively at train/test ratio of 10:90.

Anchor node selection policy. Fig. 19 shows the performance sensitivity of sweeping ICG anchor node selection in Algorithm 1. In addition to ICG node degree, the other available options include anchor node selection based on the item access frequency or all random. The degree-based clustering performs marginally better than frequency-based and outperforms random anchor node selection. This shows the robust nature of ICG clustering algorithm. It enables GRACE to outperform prior works without any specific requirement for choosing an anchor node for clustering.

B.4 Optimal Algorithmic Parameter Search

We show the parameter search process and their sensitivity involved in Algorithm. 2.

Performance sensitivity on α. We sweep the value of α to find its effect on performance. This parameter is used in estimating the benefit of adding nodes to existing clusters in the proposed clustering algorithm (see line 14 in Algorithm 3). We sweep α from 0 to 1, and find its effect on the performance of each workload. Table 5 presents the performance of each α normalized to the best performance, averaged across all workloads. This shows that GRACE performance does not change significantly for $\alpha \geq 0.25$, and achieves an optimal performance with $\alpha = [0.5, 0.75]$ (highlighted in green).

Table 5: Performance of different α normalized to the best performing α, averaged across datasets.

α	0	0.25	0.5	0.75	1.0
GM	0.934	0.987	0.997	0.997	0.996

`MAX CLUSTER SIZE` and `tolerance factor`. Fig. 20 shows the performance sensitivity of GRACE for different maximum cluster sizes and tolerance factors. Maximum cluster size is a parameter used in the proposed algorithm to limit the sizes of the formed clusters (see line 20 in Algorithm 2). Tolerance factor adds a margin for the estimated benefit to drop while still allowing a new node to be added to an existing cluster (see line 19 in Algorithm 2). Both hyperparameters modulate the cluster sizes. Fig. 20 only shows the most interesting data points for each dataset; in reality, we sweep the tolerance factor from 0 to 100.

Intuitively, allowing larger clusters leads to higher memory traffic saving if partial sums of several items are cached. This, however, limits the diversity of items to be cached due to limited cache space, limiting the ability to divert more memory traffic to the cache space. Therefore, there is a rich trade-off space between savings due to each cluster and the diversity of cached items. GRACE navigates this trade-off space to find the optimal hyperparameter for each workload. We show the tuning process of these parameters below.

C APPENDIX: DECISION ENGINE DESIGN FOR PREDICTING OPTIMAL PARAMETER SETTING

The choice of maximum cluster size and tolerance affects the quality of the selected popular items and their combinations, thus affecting the overall performance. The optimal choice is different for different input characteristics. Fig. 20 shows the speedup for each input graph under different maximum cluster size and tolerance combinations. To be able to apply GRACE to arbitrary input and get the best speedup, we build a decision engine to pick the optimal or near-optimal combination of maximum cluster size and tolerance without exhaustively experimenting with all possible combinations.

We use three features from the input: *number of items*, *average pooling factor*, and *average node degree in ICG* to train the

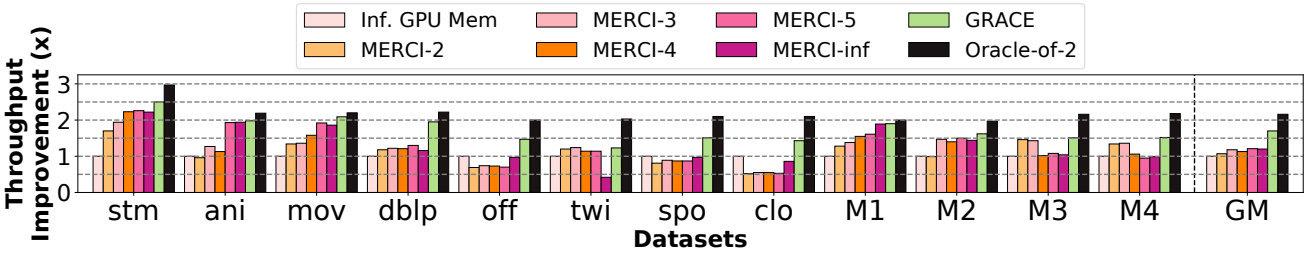

Figure 16: Throughput improvement of MERCI-k over compared baselines (higher is better).

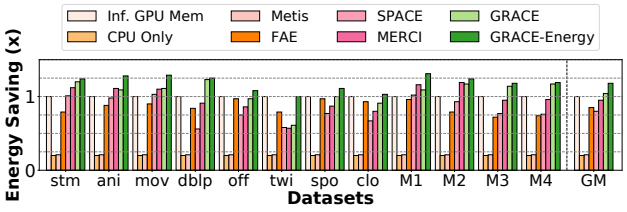

Figure 17: Energy saving in DLRM inference of compared works normalized to Infinite GPU Memory (higher is better).

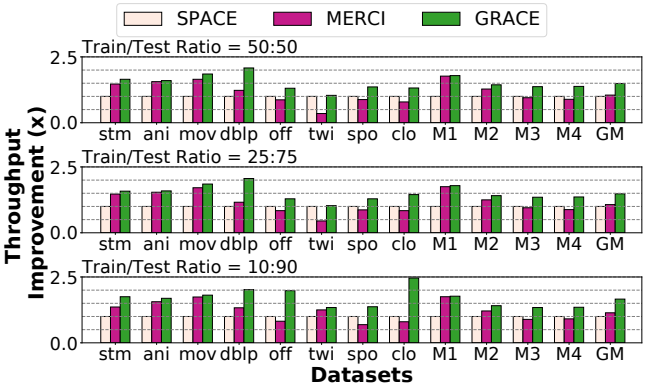

Figure 18: Performance sensitivity of compared works for different train/test ratios normalized to SPACE. GRACE consistently outperforms the compared works even when the training set is limited.

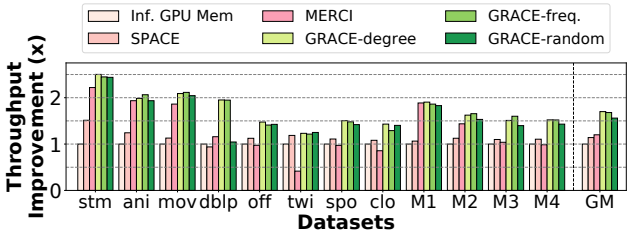

Figure 19: Performance sensitivity of compared works for different anchor node selection policies normalized to SPACE. The available options are selected based on ICG node degree, the item access frequency, and random.

decision engine. We use the decision tree implementation from scikit-learn [60] library, which uses an optimized CART (Classification and Regression Trees) algorithm. The decision tree finds the feature that yields the largest information gain at each level. The output of the decision engine is the maximum cluster size and tolerance combination that gives optimal performance. We define the combination that gives the best speedup result as the **strict optimal combination**, and we define a set of combinations that gives a speedup no less than relaxing coefficient (**rc**) to the optimal speedup as the **relaxed optimal combination set**. We use the strict optimal combination to train the decision engine, and consider the decision engine's prediction accurate if it gives a result that falls in the relaxed optimal combination set.

We train the decision engine with 80-20 random split for train-test data, and we repeat the procedure 50 times with different random seeds. The decision engine achieves an average accuracy of 92.7% under a 90% relaxing coefficient.

D ARTIFACT APPENDIX

D.1 Abstract

This paper presents an algorithm-system co-design for improving the performance of the embedding layer in Deep Learning Recommendation Models (DLRMs). This document briefly describes how to reproduce the main result of our paper. The performance results shown in the paper are machine-dependent. For example, Fig. 8, Fig. 13, and Fig. 14 show results on a CPU-GPU system, HBM-only system, and DIMM-HBM system with Processing-In-Memory (PIM) capability, respectively. To enable reproducing results in a timely fashion on different machines, we discuss the methodology to reproduce the main result of our paper that is machine-independent (Fig. 10). Specifically, our instructions include 1) how to download the input datasets, 2) how to pre-process these datasets, 3) how to reproduce the memory traffic reduction results for each baseline, and 4) how to generate a plot similar to Fig. 10. Expected result: compared to a no-reduction baseline, GRACE reduces the memory traffic by 1.7×.

D.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Deep Learning Recommendation Model (DLRM)
- **Program:** `c++` and `python3`
- **Compilation:** `g++ 9.4.0`
- **Dataset:** Steam (stm), Anime (ani), MovieLens20M (mov), DBLP (dblp), AmazonOffices (off), Twitch (twi), AmazonSports (spo), AmazonClothes (clo), mixture of datasets twi-mov-ani-stm (M1), mixture of datasets clo-off-dblp-ani (M2), mixture of datasets spo-off-dblp-twi (M3), mixture of datasets clo-spo-off-dblp (M4)

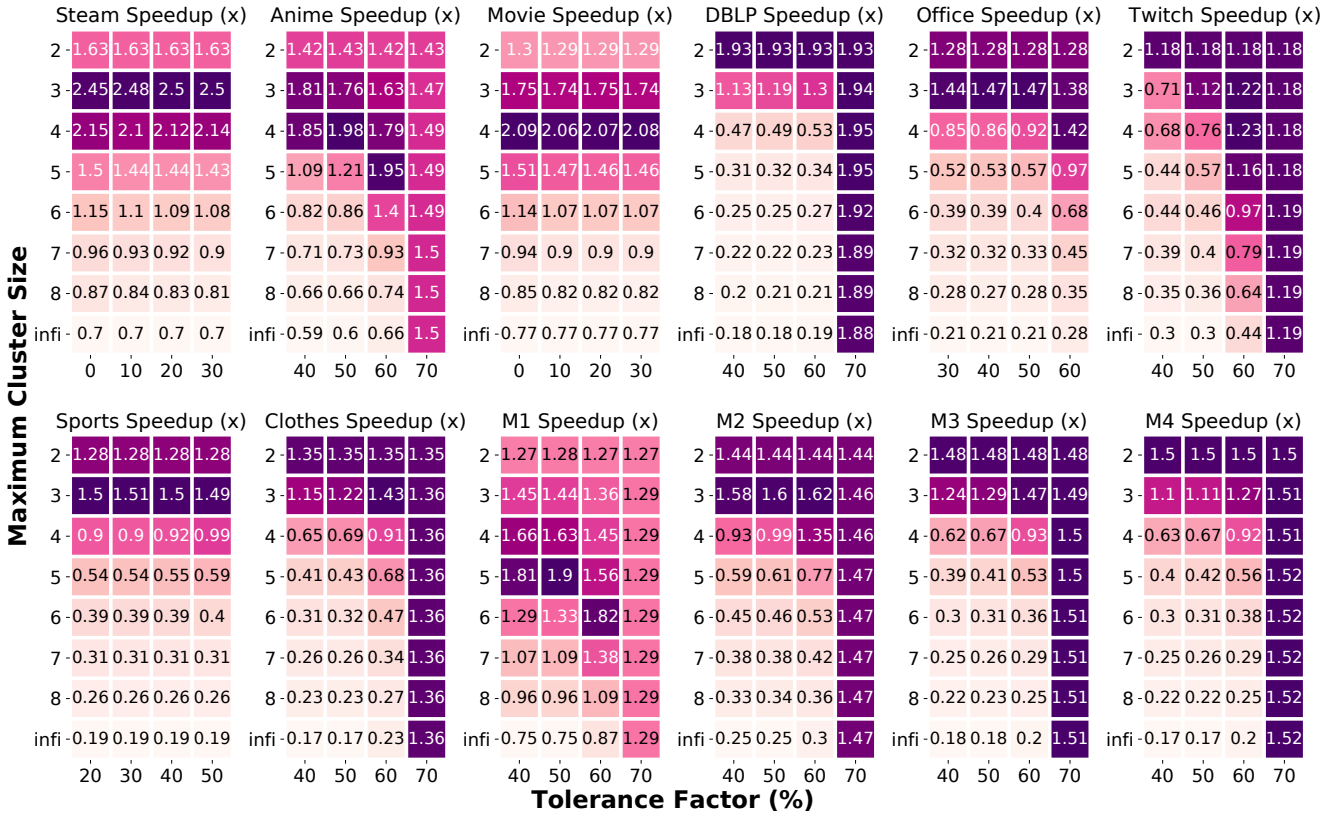

Figure 20: Performance sensitivity of GRACE to maximum cluster size and tolerance factor normalized to Infinite GPU Memory.

- **Run-time environment:** Implementation should run natively
- **Hardware:** CPU with 64 GB main memory or more
- **Execution:** Bash script for automatic compilation and execution
- **Metrics:** Memory traffic reduction
- **Output:** Memory access count in `hbm_only_*_log/` folders, reproduced Fig. 10 in the paper in `Fig10_plot/` folder
- **Experiments:** Memory access counts for GRACE, MERCI, SPACE, and Metis
- **How much disk space required (approximately)?:** 70GB
- **How much time is needed to prepare workflow (approximately)?:** 3 hours
- **How much time is needed to complete experiments (approximately)?:** 9 hours without Metis, 15 hours with Metis
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License (MERCI)
- **Data licenses (if publicly available)?:** Creative Commons Attribution ShareAlike License (dblp), CC0: Public Domain License (ani)
- **Workflow framework used?:** GAPBS

D.3 Description

D.3.1 How to access? The artifact code base can be downloaded from https://github.com/Linestro/GRACE. A third-party code base MERCI is needed and can be obtained by downloading from https://github.com/SNU-ARC/MERCI.git. The README file in the root directory of GRACE repository contains instructions to download open-source data sets and commands to reproduce Fig. 10.

D.3.2 Hardware dependencies: Any commodity CPU should be adequate for running the code implementation.

D.3.3 Software dependencies: We use python 3.9 and g++ 9.4.0 on Ubuntu 20.04.5 LTS (GNU/Linux 5.4.0-135-generic x86_64).

D.3.4 Datasets: We use real-world datasets for evaluation. The datasets are obtained from the following links:

- DBLP: [link]
- AmazonSports: [link1] [link2]
- AmazonOffices: [link1] [link2]
- AmazonClothes: [link1] [link2]
- Anime: [link]
- Twitch: [link]
- Movie: [link]
- Steam: [link]

D.4 Installation

Download the GRACE code base from https://github.com/Linestro/GRACE. In *GRACE/* folder, download the MERCI code base from https://github.com/SNU-ARC/MERCI.git

D.5 Experiment Workflow

Due to the large number of commands, please refer to `GRACE/README.md` for the commands for each step to run.
Step 1: Create necessary folders `GRACE/` and `GRACE/MERCI/`.

Step 2: Download and process datasets. (3 hours)
Step 3: Perform datasets cleaning. (10 minutes)
Step 4: Prepare mixed datasets. (2 hours)
Step 5: Generate ICG with training set. (3 hours)
Step 6: Reformat datasets into inference streaming set. (5 minutes)
Step 7: Reproduce memory access count for GRACE, MERCI, SPACE. (1 hour)
Step 8: (Optional) Reproduce memory access count for Metis. (3 hours)
Step 9: Reproduce Fig. 10 in the paper using the memory access count collected in steps 7 and 8.

D.6 Evaluation and Expected Results

After the runs have completed running, the raw results are in `hbm_only_grace_log/`, `hbm_only_merci_log/`, `hbm_only_space_log/`, and `hbm_only_metis_log/`.
The reproduced Fig. 10 is in `Fig10_plot/`, and it should be close to the Fig. 10 in the paper; a small error ($< \pm 5\%$) accepted because each time the training-testing set split (by default 50:50) is randomized.

REFERENCES

[1] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. 2021. Accelerating recommendation system training by leveraging popular choices. *arXiv preprint arXiv:2103.00686* (2021).
[2] My anime list. 2016. Anime recommendations database. https://www.kaggle.com/CooperUnion/anime-recommendations-database.
[3] SNU Architecture and Code Optimization (ARC) Lab. 2021. MERCI Code Repository. https://github.com/SNU-ARC/MERCI.
[4] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, and Hyesoon Kim. 2021. FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 908–920. https://doi.org/10.1109/HPCA51647.2021.00080
[5] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*. 462–473.
[6] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619
[7] Erik Brynjolfsson, Yu Hu, and Duncan Simester. 2011. Goodbye pareto principle, hello long tail: The effect of search costs on the concentration of product sales. *Management Science* 57, 8 (2011), 1373–1386.
[8] Ümit V Çatalyürek and Cevdet Aykanat. 2011. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of parallel computing*. Springer, 1479–1487.
[9] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710
[10] Mauro Cerasoli and Aniello Fedullo. 2002. The inclusion-exclusion principle. *Journal of Interdisciplinary Mathematics* 5, 2 (2002), 127–141.
[11] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 12–23.
[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622. https://doi.org/10.1109/MICRO.2014.58
[13] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. *CoRR* abs/1606.07792 (2016). arXiv:1606.07792 http://arxiv.org/abs/1606.07792
[14] Hyeonseong Choi and Jaehwan Lee. 2021. Efficient Use of GPU Memory for Large-Scale Deep Learning Model Training. *Applied Sciences* 11, 21 (2021), 10377.
[15] Chiachen Chou, Aamer Jaleel, and Moinuddin K Qureshi. 2015. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 198–210.
[16] Chia Chen Chou, Aamer Jaleel, and Moinuddin K Qureshi. 2014. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 1–12.
[17] Marshall Choy. [n. d.]. Accelerating the Modern Machine Learning Workhorse: Recommendation Inference. https://sambanova.ai/blog/accelerating-the-modern-ml-workhorse-recommendation-inference/
[18] 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.
[19] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. 2018. Bandana: Using non-volatile memory for storing deep learning models. *arXiv preprint arXiv:1811.05922* (2018).
[20] AA Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. 2021. Mixed dimension embeddings with application to memory-efficient recommendation systems. In *2021 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2786–2791.
[21] 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.
[22] Asela Gunawardana and Guy Shani. 2015. Evaluating recommender systems. In *Recommender systems handbook*. Springer, 265–308.
[23] Cong Guo, Yangjie Zhou, Jingwen Leng, Yuhao Zhu, Zidong Du, Quan Chen, Chao Li, Bin Yao, and Minyi Guo. 2020. Balancing efficiency and flexibility for DNN acceleration via temporal GPU-systolic array integration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[24] 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.
[25] Udit Gupta, Samuel Hsia, Jeff Jun Zhang, Mark Wilkening, Javin Pombra, Hsien-Hsin S. Lee, Gu-Yeon Wei, Carole-Jean Wu, and David Brooks. 2021. RecPipe: Co-designing Models and Hardware to Jointly Optimize Recommendation Quality and Performance. *CoRR* abs/2105.08820 (2021). arXiv:2105.08820 https://arxiv.org/abs/2105.08820
[26] Udit Gupta, Xiaodong Wang, Maxim Naumov, Carole-Jean Wu, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2019. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. *CoRR* abs/1906.03109 (2019). arXiv:1906.03109 http://arxiv.org/abs/1906.03109
[27] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)* 5, 4 (2015), 1–19.
[28] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629. https://doi.org/10.1109/HPCA.2018.00059
[29] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*. 507–517.
[30] 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.
[31] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B. Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. 2020. MicroRec: Accelerating Deep Recommendation Systems to Microseconds by Hardware and Data Structure Solutions. *CoRR* abs/2010.05894 (2020). arXiv:2010.05894 https://arxiv.org/abs/2010.05894
[32] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous GPU/CPU Clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 463–479.
[33] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati,

William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 http://arxiv.org/abs/1704.04760

[34] 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.

[35] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[36] Wang-Cheng Kang and Julian McAuley. 2018. Self-attentive sequential recommendation. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 197–206.

[37] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S Lee, and Xuan Zhang. 2022. Hercules: Heterogeneity-Aware Inference Serving for At-Scale Personalized Recommendation. *arXiv preprint arXiv:2203.07424* (2022).

[38] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David M. Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. 2019. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. *CoRR* abs/1912.12953 (2019). arXiv:1912.12953 http://arxiv.org/abs/1912.12953

[39] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-guided BTB prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 816–829.

[40] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 146–159.

[41] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-guided instruction cache replacement for data center applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 734–747.

[42] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters* 15, 1 (2015), 45–49.

[43] 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.

[44] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2021. Tensor casting: Co-designing algorithm-architecture for personalized recommendation training. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 235–248.

[45] Youngeun Kwon and Minsoo Rhu. 2022. Training personalized recommendation systems from (GPU) scratch: look forward not backwards. *arXiv preprint arXiv:2205.04702* (2022).

[46] Dominique LaSalle and George Karypis. 2016. A parallel hill-climbing refinement algorithm for graph partitioning. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 236–241.

[47] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the 28th International Conference on Compiler Construction*. 65–75.

[48] 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.

[49] Bingchao Li, Choungki Song, Jizeng Wei, Jung Ho Ahn, and Nam Sung Kim. 2016. Exploring new features of high-bandwidth memory for GPUs. *IEICE Electronics Express* (2016), 13–20160527.

[50] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: critical slice prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 300–313.

[51] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 369–381.

[52] 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 *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 162–171.

[53] Micron. 2015. DDR4 SDRAM Data sheet, MT40A2G4, MT40A1G8, MT40A512M16. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf

[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, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). arXiv:1906.00091 http://arxiv.org/abs/1906.00091

[55] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. 2017. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 41–54.

[56] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[57] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. 2021. TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 268–281. https://doi.org/10.1145/3466752.3480080

[58] Yoon-Joo Park and Alexander Tuzhilin. 2008. The long tail of recommender systems and how to leverage it. In *Proceedings of the 2008 ACM conference on Recommender systems*. 11–18.

[59] Apurva Pathak, Kshitiz Gupta, and Julian McAuley. 2017. Generating and personalizing bundle recommendations on steam. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1073–1076.

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

[61] Jérémie Rappaz, Julian McAuley, and Karl Aberer. 2021. Recommendation on Live-Streaming Platforms: Dynamic Availability and Repeat Consumption. In *Fifteenth ACM Conference on Recommender Systems*. 390–399.

[62] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Twenty-ninth AAAI conference on artificial intelligence*.

[63] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: Statistical Feature-Based Memory Optimization for Industry-Scale Neural Recommendation. *arXiv preprint arXiv:2201.10095* (2022).

[64] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. 2020. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 165–175.

[65] Jaewoong Sim, Alaa R Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent hardware management of stacked dram as part of memory. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 13–24.

[66] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[67] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2021. RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference. In *28th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2022)*.

[68] Idan Szpektor, Aristides Gionis, and Yoelle Maarek. 2011. Improving recommendation for long-tail queries via templates. In *Proceedings of the 20th international conference on World wide web*. 47–56.

[69] Nishil Talati, Ameer Haj Ali, Rotem Ben Hur, Nimrod Wald, Ronny Ronen, Pierre-Emmanuel Gaillardon, and Shahar Kvatinsky. 2018. Practical challenges in delivering the promises of real processing-in-memory machines. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1628–1633.

[70] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. 2016. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE*

Transactions on Nanotechnology 15, 4 (2016), 635–650.

[71] Mengting Wan and Julian McAuley. 2018. Item recommendation on monotonic behavior chains. In *Proceedings of the 12th ACM conference on recommender systems*. 86–94.

[72] Ruoxi Wang, Bin Fu, G. Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. *Proceedings of the ADKDD'17* (2017).

[73] 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.

[74] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex Bronstein, Trevor Mudge, Ronald Dreslinski, and Nishil Talati. 2023. Artifact of "GRACE: A Scalable Graph-Based Approach To Accelerating Recommendation Model Inference". Zenodo. https://doi.org/10.5281/zenodo.7699872

[75] Hongzhi Yin, Bin Cui, Jing Li, Junjie Yao, and Chen Chen. 2012. Challenging the long tail recommendation. *arXiv preprint arXiv:1205.6700* (2012).

[76] Buyun Zhang, Liang Luo, Xi Liu, Jay Li, Zeliang Chen, Weilin Zhang, Xiaohan Wei, Yuchen Hao, Michael Tsang, Wenjun Wang, et al. 2022. DHEN: A Deep and Hierarchical Ensemble Network for Large-Scale Click-Through Rate Prediction. *arXiv preprint arXiv:2203.11014* (2022).

[77] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems* 2 (2020), 412–428.

[78] 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.

Received 2022-10-20; accepted 2023-01-19