MERCI: Efficient Embedding Reduction on Commodity Hardware via Sub-query Memoization

Yejin Lee
Seoul National University
Republic of Korea
yejinlee@snu.ac.kr

Seong Hoon Seo
Seoul National University
Republic of Korea
andyseo247@snu.ac.kr

Hyunji Choi
Seoul National University
Republic of Korea
hyunjichoi@snu.ac.kr

Hyoung Uk Sul
Seoul National University
Republic of Korea
stuartsul@snu.ac.kr

Soosung Kim
Seoul National University
Republic of Korea
soosungkim@snu.ac.kr

Jae W. Lee
Seoul National University
Republic of Korea
jaewlee@snu.ac.kr

Tae Jun Ham
Seoul National University
Republic of Korea
taejunham@snu.ac.kr

ABSTRACT

Deep neural networks (DNNs) with embedding layers are widely adopted to capture complex relationships among entities within a dataset. Embedding layers aggregate multiple embeddings—a dense vector used to represent the complicated nature of a data feature—into a single embedding; such operation is called *embedding reduction*. Embedding reduction spends a significant portion of its runtime on reading embeddings from memory and thus is known to be heavily memory bandwidth-bound. Recent works attempt to accelerate this critical operation, but they often require either hardware modifications or emerging memory technologies, which makes it hardly deployable on commodity hardware. Thus, we propose MERCI, **m**emoization for **e**mbedding **r**eduction with **c**lusteri**n**g, a novel memoization framework for efficient embedding reduction. MERCI provides a mechanism for memoizing partial aggregation of correlated embeddings and retrieving the memoized partial result at a low cost. MERCI substantially reduces the number of memory accesses by 44% (29%), leading to 102% (74%) throughput improvement on real machines and 40.2% (28.6%) energy savings at the expense of 8× (1×) additional memory usage.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; • **Computer systems organization**; • **Software and its engineering** → *Software system structures*;

KEYWORDS

Memoization, Recommender Systems, Embedding Lookup

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '21, April 19–23, 2021, Virtual, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446717

ACM Reference Format:
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 (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3445814.3446717

1 INTRODUCTION

The capacity of deep neural networks (DNNs) is scaling rapidly to learn more complex and implicit relationships between entities within a dataset. *Embedding* is a data type widely used to represent such complicated nature of the dataset [4, 18, 40, 41, 48, 49]. It is a vector projection of high-dimensional sparse feature space to a low-dimensional dense space that preserves the semantics of the original features, for DNNs to process it easily as an input.

For instance, in an online shopping website, each product on sale can be treated as a feature and be represented with distinct embedding. Embeddings are trained to extract the semantics of features (i.e., products); therefore, co-appearing products (e.g., a notebook and a pencil) will appear relatively adjacent in the embedding space. In this setup, an online user can be represented by a set of products that he/she has recently browsed or purchased. Thus, to process a *query* on this user, it takes to gather this set of (products) embeddings and reduce them (e.g., sum, average, max, inner product) into a single embedding. This operation is called *embedding reduction*.

Embedding reduction is widely used in various applications. For instance, when an embedding represents a word [33, 39], embedding reduction is used to represent a sentence or a document. Another typical use case is in a personalized recommender system where it performs embedding reduction to represent users (like in the previous example) or products [10, 22, 53]. It is known that processing a single query takes a reduction of tens or even hundreds of embeddings in production-scale recommender systems such as Facebook's Deep Learning Recommendation Model (DLRM) [23] and Alibaba's Deep Interest Network (DIN) [58]. Unfortunately, embedding reduction takes a significant portion of the total DNN runtime. Previous studies [16, 19, 24, 41, 42, 51] report that embedding reduction (`SparseLengthsSum` in Caffe2[20]) takes significant portion,

50% to 75% of the total inference latency for the models introduced in DLRM[19, 24, 32]. This operation is known to be memory-bound with a relatively small amount of arithmetic computation (i.e., having a low arithmetic intensity [24, 32]). This inefficiency is mainly due to sparse indexing, which generates a large number of irregular memory accesses (details in Section 2.2). Since off-chip memory accesses are more expensive than on-chip computation in terms of latency and energy consumption [9, 26, 52], it is critical to reduce this high cost of memory accesses for efficient embedding reduction.

This problem has recently drawn much attention from the research community, and techniques have been proposed to accelerate this critical DNN operation. For example, RecNMP [32] and TensorDIMM [34] propose near-memory processing (NMP) architectures to reduce the performance and energy cost of embedding reduction. Bandana [19] leverages non-volatile memory (NVM) technology as a lower-cost alternative to existing fully DRAM-based storage while providing sufficient bandwidth by effective DRAM caching. However, these proposals are not readily applicable to the mainstream hardware as they require either hardware modifications or emerging memory technologies.

Memoization[38] is a classic technique that stores computation results for an input (query) and reuses them when the same query arrives again. It leverages time-space tradeoffs to reduce the number of memory accesses as well as compute cycles at the cost of additional space. While applying memoization for efficient embedding reduction is appealing, making it performant on the commodity hardware poses several challenges. First, a memoization table must be compact enough to fit in memory while providing broad coverage (hit rate). Second, the benefit of memoization should outweigh the cost of the original embedding reduction operation. If a memoization table lookup takes more memory accesses than accessing all the embeddings it covers, memoization will not be practical.

This paper proposes MERCI, a novel memoization framework for efficient embedding reduction on the commodity hardware. We observe that there often exists a *correlation structure* among features in a real-world dataset. In the previous example of online shopping, the features of (*notebook*, *pencil*) or (*bread*, *butter*) have a high chance to appear together in a user's browse history, but those of (*bread*, *pencil*) are not as much. To increase the coverage of memoization, we propose fine-grained (sub-query) memoization to perform partial reduction when processing a query. We introduce Correlation-Aware Variable-Sized Clustering to identify clusters of frequently co-appearing features of variable length with high coverage and small table size, as well as a feature remapping scheme to quickly locate a partially reduced embedding with a small number of instructions. Our evaluation of MERCI on a 16-core Intel CPU using both six synthetic and eight real-world datasets demonstrates a geomean speedup of 102%, memory access reduction by 44%, and system energy savings of 40.2% in embedding reduction.

In summary, our work makes the following contributions:

- We identify opportunities for applying memoization to efficient embedding reduction for the first time.
- We introduce Correlation-Aware Variable-Sized Clustering, a novel clustering scheme that carefully weighs the benefits and costs of memoization to form clusters of co-appearing features to memoize.
- We present MERCI, a memoization framework for efficient embedding reduction. MERCI utilizes Correlation-Aware Variable-Sized Clustering, feature ID remapping, and efficient query processing to minimize additional memory accesses, hence maximizing the benefit of memoization.
- We prototype MERCI on two commodity platforms to demonstrate its effectiveness in reducing the number of memory accesses, which translates to substantial performance gains and energy savings.

2 BACKGROUND AND MOTIVATION

2.1 Embedding Reduction

```
1  // N: Number of embeddings
2  // D: Dimension of each embedding vector
3  float emb_table[N][D];
4  def embedding_reduction (vector<int> query[B],
5                            float &res[B][D]):
6    for qid=0 to B-1:
7      for i=0 to query[qid].size()-1:
8        int cur_feature = query[qid][i];
9        float[D] embedding_vec = emb_table[cur_feature];
10       /* Embedding Reduction */
11       for d = 0 to D-1:
12         res[qid][d] += embedding_vec[d];
```

Figure 1: Pseudocode of embedding reduction.

Embeddings. An embedding is a relatively low-dimensional continuous vector that often represents a single categorical feature. For example, embedding generally represents a word in natural language processing (NLP) models [18, 35]. Without embedding, each word in a corpus would be assigned a dimension and is represented as a one-hot vector, resulting in an extremely sparse vector. Even worse, such representation does not contain semantic or syntactic relations between words; hence, in many cases, a neural network (NN) model maps this sparse vector to a low-dimensional dense vector that captures a word's semantic meaning and keeps similar words in close distance. Such processing makes it easier for machine learning (ML) systems to infer the meaning of input values.

Similarly, embedding is often used in recommender systems[41, 49]. For instance, a recommender system at an online shopping website often employs a NN model to learn the semantics of products and create an embedding for each product. These embeddings are later used as categorical feature inputs to the recommender system. In practice, modern recommender systems utilize an extensive range of embeddings for different categorical features. One example of a user-related categorical feature is a set of products a user has recently browsed.

Embedding Reduction. In many cases multiple features can constitute a single categorical variable. For example, a user can have multiple recently browsed products or multiple favorite brands. To represent these categories as a single embedding vector, embedding reduction operation should be performed. This operation loads an embedding vector for each feature (e.g., a single product), and performs a reduction operation (e.g., sum, average, max, inner product) on them. In NLP models, embedding reduction generates a sentence or a document embedding by aggregating embedding vectors for words in a given sentence or document[33]. Many other ML models also adopt embedding reduction [7, 10, 13, 22], and all popular NN

```
1   // for i = 0 to query[qid].size()-1;
2   REDUCTION:
3       // %rcx: feature index i; initialized with 0
4       // %r9: query[qid].size()
5   FEAT_LOOP:
6       // for d = 0 to D-1:
7       //    res[d] += embedding_vec[d];
8                   ...
9       88.64%    vmovups (%rsi), %ymm2
10       6.96%    vadddps (%rax), %ymm2, %ymm0
11       1.68%    vmovups %ymm0, (%rax)
12                ... // loop unrolling
13  FEAT_REPEAT:
14                inc %rcx
15                cmp %rcx, %r9
16                jne FEAT_LOOP
```

Figure 2: Instruction-level runtime breakdown for embedding reduction using Amazon-Books dataset[1].

frameworks like Tensorflow (`embedding_lookup_sparse(...)`) [1], PyTorch (`EmbeddingBag(...)`) [44], and Caffe2 (`SparseLengthsSum`) [20] support this operation.

Figure 1 shows the pseudocode of embedding reduction with the sum operator. The embedding reduction operation takes a set of queries (a batch of queries), and iterates over each feature in each query (Line 6-7). For each feature, it retrieves the corresponding embedding vector from the embedding table (Line 8-9) and performs an element-wise reduction for the vector (Line 11-12). The result of the reduction for each query (i.e., `res`) is the output of this operation. Note that, although sum reduction is adopted here, other reduction operators such as min, max, or inner product can be employed.

2.2 Bottleneck Analysis

Embedding reduction has a small number of arithmetic computations compared to the number of memory reads it generates. In a CPU, which is a common deployment platform to accommodate a large embedding table [24, 27, 32], SIMD optimization (e.g., Intel AVX) is usually applied. Figure 2 shows the instruction-level profiling results of the embedding reduction operation on the CPU using `perf annotate`. The profiling results demonstrate that embedding reduction operation is an extremely memory bandwidth-bound operation as most of the runtime is spent on the SIMD load instruction (`vmovups`). In this code, the SIMD add instruction is very efficient with wide vector processing, but our internal profiling result shows that memory accesses to the embedding table yield a high cache miss rate (e.g., 52.8%) in L3 cache despite the existence of temporal locality to a certain degree. This inefficiency is due to the following reasons. First, embedding reduction accesses embedding table with sparse index; thus, cache miss rate increases. Second, the embedding tables are often much larger than L3 cache sizes. For example, the embedding table for one million 256B embeddings takes 256MB, which is an order of magnitude larger than a typical L3 cache size. There are cases where multiple embedding tables are accessed at the same time by different threads [32], thus further increasing the memory pressure. Third, there are other data structures that contend for the cache space (e.g., `query`, `res`). Finally, between batches of embedding reduction, other layers of the DNN model may be executed to evict most of the embedding vectors from the cache hierarchy.

[1] See Section 7.1 for details on datasets.

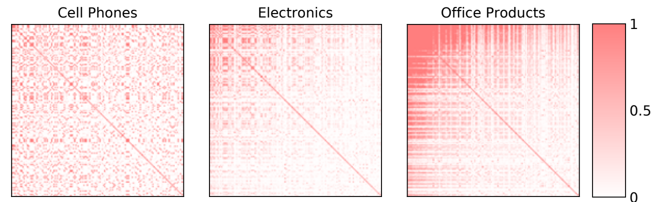

Figure 3: Correlation heat map for product pairs of top 150 items in the Amazon Review dataset.

In fact, the embedding reduction operation is the primary performance bottleneck in many NN models. One example is Facebook's DLRM [41], in which their profiling results [23, 32] show that this operation accounts for 50% to 75% of the total runtime in their models (i.e., RMC1, RMC2). Furthermore, with a scaling of the dataset and adoption of specialized DNN accelerators to shrink the portion of the compute-intensive layers, the bottleneck is likely to be more critical in the future.

2.3 Opportunities for Sub-query Memoization

Memoization [38] is a classic technique that stores the result of computation for an input (query) and reuses it when the same query arrives again. Memoization leverages space-time trade-offs and is the most effective when a small subset of inputs is likely to occur repeatedly. Memoization can be an effective solution to reduce memory accesses in embedding reduction by replacing N embedding table lookups with a single memoization table lookup, where N is the number of the embeddings that the given query should aggregate. However, this coarse-grained (i.e., query granularity) memoization has a limited coverage as memoization can be applied only when the *exact* same query arrives again.

Instead, we identify new opportunities for *fine-grained* (i.e., sub-query) memoization to enable partial reduction by exploiting the correlation structure that exists in many real-world categorical features. This partial reduction is possible as a reduction operator (e.g., sum) is commutative and associative by definition. For example, if a query contains a user's recently browsed items, there is a high probability that if *notebook* appears in the query, then *pen* would also appear together. Similarly, (*notebook*, *pencil*) and (*pencil*, *eraser*) are co-appearing pairs that are frequently browsed. In contrast, (*pants*, *pencil*) and (*shampoo*, *apple*) are pairs that are much less likely to appear together. We can exploit such patterns and memoize the partial reduction for the frequently co-appearing features to replace two or more memory accesses with single memory access.

Figure 3 illustrates such opportunities indeed exist in a real-world dataset. This Amazon Review dataset contains lists of items that are bought/viewed together, which are commonly utilized as a categorical input of the recommender system. The heat map depicts the pair-wise correlation of Top 150 frequently appearing items in this dataset. Thus, both X and Y-axis enumerate the 150 items, and a dot in the figure quantifies the correlation of a particular item pair in the range of zero (low, white) to one (high, red). This heat map shows pairs of items jointly appear frequently in the lists (queries), which can be excellent targets of memoization. Although the figure only shows the pair-wise co-appearance, there often exists a cluster of co-appearing features (items) that have a high chance to appear together. Thus, we propose to exploit this correlation structure to

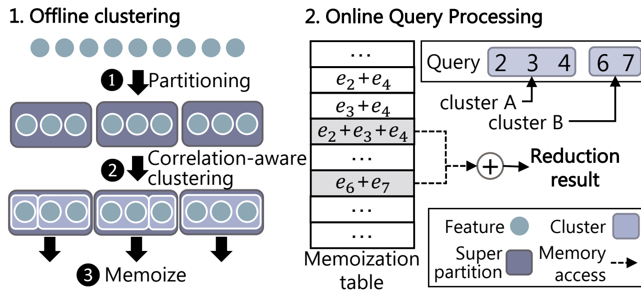

Figure 4: Overview of MERCI

perform memoization at a *sub-query* granularity (as few as two embeddings) to provide much greater coverage than the coarse-grained memoization scheme at a query granularity.

3 MERCI OVERVIEW

Although conceptually simple, building a performant memoization system for embedding reduction is a challenging task. If done naïvely using a dense array, the cost of memoization can nullify its benefit. For example, storing partial sums of all possible combinations would not be possible due to memory constraint; for N embeddings, the required memory space is $2^N \times$ {Embedding Vector Size} where N often exceeds a million in many popular NN models. Utilizing sparse data structure (e.g., a sparse hash table) for maintaining the memoized values and only storing partial sums of frequently co-appearing features can solve this problem, but triggers considerable additional memory access to retrieve a memoized value.

MERCI proposes a way to get the best of both approaches. It moves through two phases that we call offline clustering and online query processing. Figure 4 shows the overall overview of MERCI.

Offline clustering. The offline clustering phase consists of two steps. ❶ MERCI first partitions N features into a set of coarse-grained, fixed-length partitions called **super-partitions** by utilizing an existing hypergraph partitioning algorithm on the training dataset (Step 1). Each super-partition contains features that are likely to appear together based on the history of queries. Then, ❷ MERCI applies Correlation-Aware Variable-Sized Clustering to each super-partition, dividing features in a super-partition into fine-grained, variable-length clusters (Step 2). Finally, ❸ MERCI creates a memoization table that holds all possible partial sum combinations for each cluster. Section 4 describes the details of these steps.

Online query processing. MERCI utilizes the memoization table created from the previous phase to serve incoming queries. Once a query arrives—for example, one requiring accesses to the feature set of $\{2, 3, 4, 6, 7\}$ as in Figure 4—MERCI first identifies which features belong to the same cluster. In this example, features $\{2, 3, 4\}$ belong to cluster A and $\{6, 7\}$ to cluster B. Hence, two memoized partial sums (embeddings) are retrieved (i.e., $e_2 + e_3 + e_4$ and $e_6 + e_7$) and summed up to generate the final output. While the baseline scheme—performing embedding reduction without memoization—would have to load five embeddings (i.e., e_2, e_3, e_4, e_6, e_7), MERCI only loads two. Because the embedding reduction is often memory bandwidth-bound, such reduction in memory accesses can lead to performance improvement. Section 5 describes how the online query processing phase utilizes clusters to optimize the memoization table and accelerate the embedding reduction at runtime.

4 OFFLINE CLUSTERING

4.1 Step 1: Hypergraph Partitioning

The first step of the offline clustering phase is coarse-grained, fixed-length partitioning of all N features utilizing a hypergraph partitioning algorithm. Hypergraph partitioning is a popular algorithm that aims to generate a specified number of *equal-sized* partitions while minimizing the number of accessed partitions for a given set of queries; that is, features in the same partition have a high chance of occurring together. We first partition all N features into equal-sized *super-partitions* of size S with an existing hypergraph partitioning algorithm implementation called PaToH [8]. As N (number of features) can often exceed millions in today's NNs, the hypergraph partitioning algorithm first reduces this large problem size to S, and the proposed fine-grained clustering algorithm called Correlation-Aware Variable-Sized Clustering (Section 4.2) is performed on each super-partition to manage the complexity. While PaToH works well for our purpose, there exists a variety of different hypergraph partitioning algorithms [8, 17, 30, 31, 47] with different time-quality trade-offs [47], and there is no fundamental limitation in using a different algorithm.

One question that might arise is whether we can use a hypergraph-partitioning algorithm to create fine-grained clusters, instead of the proposed two-step algorithm, as it also groups frequently co-appearing features. It may be possible but suboptimal for the following reasons. First, it only generates equal-sized partitions, which either cannot support a set of co-appearing features larger than the (fixed) partition size or waste memory space for a smaller set. The size of a group of the co-appearing features would vary. Our algorithm can reflect diversity by generating clusters of size one to twenty or more. Second, the hypergraph partitioning algorithm does not give a fine-grained knob for memory constraints. Once the cluster size is set to S (embeddings), the space overhead is fixed to $N/S \times (2^S - 1) \times$ {Embedding Vector Size}. On the other hand, our algorithm can set memory limits in the granularity of less than 1% of the original embedding table size. Thus, the hypergraph partitioning algorithm can reduce the search space while increasing the locality, but in itself, it is not sufficient for our purpose.

4.2 Step 2: Correlation-Aware Variable-Sized Clustering

Once the hypergraph partitioning algorithm divides all N features into super-partitions containing S features each, the next step is to apply Correlation-Aware Variable-Sized Clustering to each super-partition individually. It is a clustering algorithm that aims to further divide the S features into fine-grained variable-sized *clusters* such that the resulting memoization table yields the maximum benefit for a given memory constraint. All possible combinations of the partial sum within a cluster are then stored for memoization.

Sketch of the Algorithm. The Correlation-Aware Variable-Sized Clustering algorithm is applied to all N/S super-partitions independently. Let c_i be a set of feature IDs that belongs to cluster i. Initially, we let each feature form a distinct cluster by itself (i.e., S clusters of size one). In this state, there is nothing to memoize since no cluster has more than one feature. The algorithm then selects and merges two clusters; to select these clusters, it considers both the *benefit* (i.e., decrease in the expected number of memory accesses

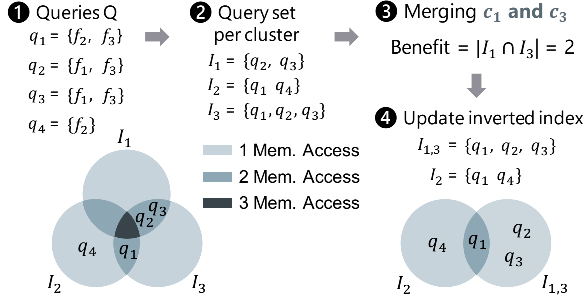

Figure 5: Illustration of the process for computing benefits for a particular merge decision.

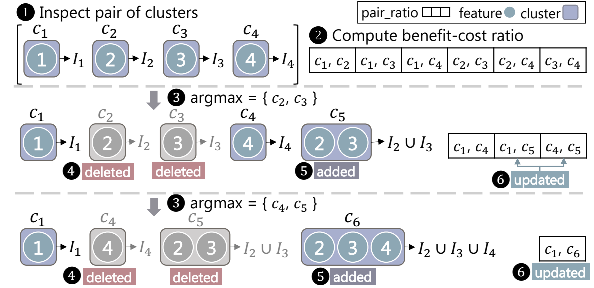

Figure 6: Illustration of the Correlation-Aware Variable-Sized Clustering.

if merged) and the *cost* (i.e., increase in memory usage if merged). This cost and benefit are estimated using queries in the training set, as explained below. Once two clusters are merged, the algorithm repeats the next iteration until it hits the memory size constraint.

Estimating the Cost. Because all possible combinations of features in a cluster is memoized, a cluster with a features occupies $(2^a - 1) \times \{\text{Embedding Vector Size}\}$ of memory space. $2^a - 1$ is the cardinality of power set (excluding empty set) for a set with a elements. With this in mind, we can compute the memory cost of merging cluster A having a features and cluster B with b features. The newly formed cluster would require $2^{a+b} - 1$ partial sums to be stored; since the original memory usage was $2^a - 1 + 2^b - 1$, the increase in memory usage (i.e., the cost of merge) is $(2^{a+b} - 2^a - 2^b + 1) \times \{\text{Embedding Vector Size}\}$.

Estimating the Benefit. Merging two clusters improve the chance of features in a query to be in the same cluster (and hence memoized). Therefore, the number of memory accesses for processing a query would likely decrease. The amount of decrease is considered as the benefit of merging two clusters. The queries in the training set are analyzed to estimate the benefit. To explain, we define Q as the set of queries being analyzed, and I is an inverted index data structure [14] for each cluster.

$$Q = \{q_i \mid i = 0, 1, ..., q\},\ I_i = \{q_j \mid \exists f_k \in c_i,\ s.t.\ f_k \in q_j\} \quad (1)$$

Following the numbering in Figure 5, ❶ $Q = \{q_1, q_2, q_3, q_4\}$ is a set of queries used for training, and each query is identified as a set of feature IDs it accesses (e.g., f_1, f_2, ...), where each feature corresponds to a specific embedding vector. ❷ Using this information, an inverted index I_i is built and maintained for each cluster i. I_i contains IDs of queries that contain at least one feature belonging to cluster i. For instance, if f_1 appears in q_2 and q_3, I_1 becomes $\{q_2, q_3\}$ as in Figure 5. ❸ Then, the benefit of merging cluster c_1 and c_3 is computed as the cardinality of $I_1 \cap I_3$, which is equivalent to the number of queries that contain at least one feature from each of c_1 and c_3. That is, if c_1 and c_3 are merged, $c_1 \bigcup c_3$ would be memoized, and thus queries q_2 and q_3 would require one memory access instead of two saving one memory access for embedding reduction. Note that, our scheme originally chooses the pair of clusters considering both *benefit* and *cost*; however, in this example, cost is the same for all pairs of clusters ($a = 1, b = 1$) and thus we only consider the benefit.

$$benefit(c_i, c_j) = |\{\ q_k \mid q_k \in I_i \cap I_j\ \}| \quad (2)$$

❹ After the algorithm merges c_i and c_j, it updates the state of the clusters and the inverted index as shown below. For the next iteration, this merged cluster is treated as a single one cluster ($c_{i,j}$) for benefit-cost analysis. In consequence, various sizes of clusters can be generated by the granularity of one feature.

$$c_{i,j} = \{\ f_k \mid f_k \in c_i \cup c_j\ \},\ I_{i,j} = \{\ q_k \mid q_k \in I_i \cup I_j\ \} \quad (3)$$

By computing *benefit* and *cost*, we evaluate **benefit-cost ratio** (i.e., $\frac{benefit}{cost}$) of all possible pairs of clusters. With the benefit-cost ratio, we can quantify the effectiveness of every possible merge of two clusters. If the pair has a high benefit-cost ratio, merging those clusters increases the benefit of memoization with relatively low additional memory usage. To summarize, a unique feature of our clustering scheme is that it selects clusters to merge by considering the benefits of forming a larger cluster (decrease in memory accesses) and its cost (increase in memory usage).

4.3 Algorithm Details

Figure 6 illustrates an example walk-through of our Correlation-Aware Variable-Sized Clustering algorithm with a running example. First, ❶ it inspects every pair of the clusters, and ❷ records the benefit-cost ratio of the merged cluster (i.e., `pair_ratio`). Then, ❸ it selects the cluster pair with the highest benefit-cost ratio (i.e., clusters 2 and 3) and merges them into a single cluster. In doing so, ❹ it deletes the two original clusters (i.e., c_2 and c_3) and ❺ adds the new aggregated cluster c_5 (= $c_{2,3}$). Finally, ❻ the cluster pairs and their ratios are updated along with the inverted index of the merged cluster. Note that merging two clusters implies that queries with a feature in either of the two clusters now access the merged cluster c_5. Thus, we set an inverted index of the merged cluster as a union of two existing inverted indexes (i.e., $I_5 = I_2 \cup I_3$). The process of ❸ - ❻ is repeated with newly updated `pair_ratio`, treating c_5 as a single cluster like others. In Figure 6, cluster 4 and 5 are selected to be merged to become c_6. Although not shown in the figure, the current memory usage is also updated after a merge. The algorithm exits if the memory usage reaches the user-specified limit.

Figure 7 presents a pseudocode of the Correlation-Aware Variable-Sized Clustering algorithm. Function `correlation_aware_clustering` is the top-level function, which merges clusters until the current memory usage reaches the user-defined memory limit (i.e., `CAPACITY_LIMIT`) (Line 36). It first calculates the benefit-cost ratios for all possible pairs of clusters of size one (Line 28-32) by calling `getBCRatio` (Line 9-14). It then finds the pair with the maximum benefit-cost ratio (Line 38) and merges the selected clusters (Line

```
1  struct Cluster{
2   int cid;
3   int size;
4   /* Queries including any feature in this cluster */
5   set<int> I;
6   /* Features in this cluster */
7   set<int> features;
8  }
9  float getBCRatio(Cluster a, Cluster b) {
10  benefit = intersection(a.I, b.I);
11  cost = pow(2, a.size + b.size)
12         - pow(2,a.size) - pow(2,b.size) + 1;
13  return benefit/cost;
14 }
15 Cluster merge(Cluster a, Cluster b, int &nextcid) {
16  /* Merge cluster i and cluster j */
17  Cluster merged;
18  merged.cid = nextcid++;
19  merged.size = a.size + b.size;
20  merged.features = union(a.features, b.features);
21  merged.I = union(a.I, b.I);
22  return merged;
23 }
24 // S: The number of initial clusters(features)
25 def correlation_aware_clustering (set<Cluster> &c):
26  /* Pair of clusters and its benefit-to-cost ratio */
27  map<int, map<int,float>> pair_ratio;
28  /* Initial evaluation of benefit-cost ratio
29     for cluster pairs */
30  for i=0 to S-1:
31   for j=i+1 to S-1:
32    pair_ratio[i][j] = getBCRatio(c[i], c[j]);
33  int nextcid = S;
34  /* Repeat merging until user-specified capacity limit */
35  capacity = S;
36  while (capacity < CAPACITY_LIMIT):
37   /* Returns cluster indices for the maximum ratio */
38   (i, j) = argmax(pair_ratio);
39   /* Merge two selected clusters */
40   Cluster merged = merge(c[i], c[j], nextcid);
41   /* Remove two clusters from c */
42   c.erase(c[i]), c.erase(c[j]);
43   /* Remove pairs containing c[i] or c[j] from pair_ratio*/
44   pair_ratio.erase(c[i].cid), pair_ratio.erase(c[j].cid);
45   /* Update ratios for cluster pairs including merged*/
46   for cl in c:
47    pair_ratio[cl.cid][merged.cid] =
48                                  getBCRatio(cl, merged);
49   /* Add merged cluster info */
50   c.insert(merged);
51   /* Recompute memory usage here */
```

Figure 7: Pseudocode of the Correlation-Aware Variable-Sized Clustering algorithm.

40). Function `merge` assigns a new ID to the merged cluster and updates its size, features, and inverted index, as discussed before (Line 15-23). Finally, the original cluster pair is erased from the cluster set (c) and the benefit-cost ratio map (`pair_ratio`) (Line 41-44), and the newly merged cluster is inserted with benefit-cost ratios calculated against all the other clusters (Line 45-50).

4.4 Parallelization of Correlation-Aware Variable-Sized Clustering Algorithm

By exploiting the super-partition-level parallelism, our clustering algorithm can be effectively parallelized. However, naïvely scheduling each thread to execute on its own super-partition may end up violating the memory usage constraint without coordination. Thus, we address this by employing a minimum benefit-cost ratio; each thread will stop merging once the benefit-cost ratio of the next merge fails to exceed this minimum ratio, and once clustering completes for all super-partitions, the total memory consumption is computed. The minimum ratio is automatically adjusted according to the calculated memory consumption, and the process continues until the expected memory consumption converges to the user-specified limit. This parallelization technique enables our clustering algorithm to utilize multiple cores in parallel, reducing its runtime by a significant factor (e.g., 9.7× on a 16-core machine with 32 hardware threads).

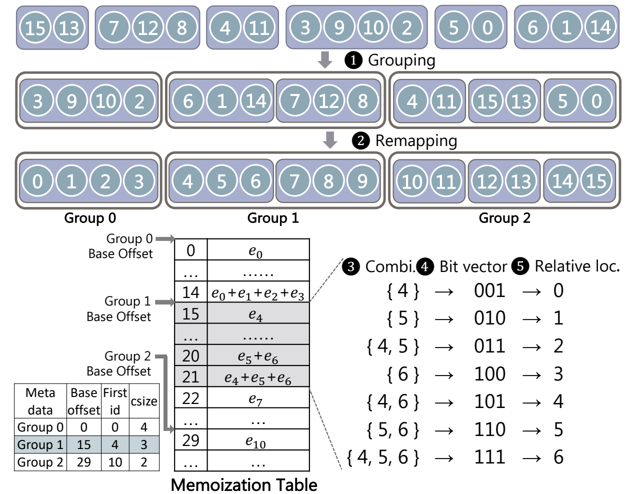

Figure 8: Preprocessing phase of online query processing.

5 ONLINE QUERY PROCESSING

This section explains how MERCI utilizes the clusters identified by the offline clustering scheme in Section 4 to create a table structure for memoization (Section 5.1). Note that this is a one-time process performed when new clusters are formed. Then we present how MERCI exploits this data structure to serve incoming queries for embedding reduction (Section 5.2).

5.1 Preprocessing

Preprocessing for memoization consists of two steps. First, MERCI remaps feature IDs to identify the cluster for each feature with only a few additional memory access. Second, the memoization table that stores partial reduction for various combinations of embedding vectors is constructed. Below, we explain each step in detail.

Step 1: Remapping Feature IDs. At runtime, MERCI needs to identify the cluster that a specific feature within a query belongs to. The naïve solution would be to maintain a mapping table that maps each feature ID to a pointer to its cluster information. However, such an approach will incur additional memory access, since it is likely that not all mapping tables and data structures containing information on each cluster are cached. Instead, our approach statically remaps feature IDs before deployment to minimize the information needed to access at runtime.

Figure 8 (❶, ❷) illustrates the feature ID remapping process. ❶ First, clusters with the same size are grouped into a cluster group, and all cluster groups are sorted by descending order of cluster size.

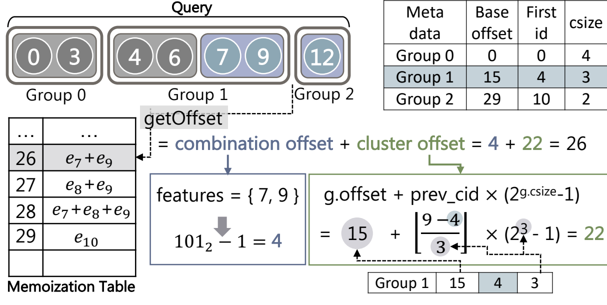

Figure 9: Illustration of MERCI query processing.

The order of the clusters within a cluster group is irrelevant. ❷ Then, starting from the first feature (ID 3 in the first cluster) to the last feature (ID 0 in the last cluster), we assign new IDs 0 to N-1 in order, where N is the total number of features. With this feature ID remapping, features in the same cluster or a cluster group have contiguous feature IDs.

Step 2: Memoization Table Construction. Once the remapping completes, MERCI constructs the memoization table containing partial sums of all $2^{cluster_size} - 1$ feature combinations in each cluster. The memoization table is a 2D array (implemented as a 1D array) storing R vectors ($R = \sum_{\forall clusters}(2^{cluster_size} - 1)$), each having D dimensions (i.e., identical to the embedding vector dimension). This memoization table follows the order of clusters determined during the feature ID remapping step.

Within a single cluster, we utilize the following mechanism to determine the partial sum's location for a specific combination. Figure 8 (❸-❺) shows a simple example of filling out memoization table for cluster {4, 5, 6}. ❸ First, we generate all possible combinations for every cluster. ❹ Then we represent the combination as a one-hot bit vector whose kth least significant bit is set when it contains the kth feature. For example, a combination of 2nd, 3rd feature (i.e., {5, 6}) is represented as $110_{(2)}$. ❺ Then, the integer representation of this number minus one (except for empty set) is utilized as the relative offset of the partial sum within the cluster. This relative offset is then added to the base offset of the cluster to find the absolute location in the memoization table. For example, in the figure, seven partial sums are stored in the memoization table consecutively starting from base offset 15.

To facilitate the retrieval of the partial reduction results during the runtime, MERCI utilizes a tiny additional metadata array, as shown in Figure 8. For each cluster group, the base offset within the memoization table for this cluster group, as well as the first feature ID and cluster size (`csize`) of this cluster group are stored.

5.2 Query Processing

When a batch of queries arrive, they are distributed to each thread, and all threads run a subset of queries in parallel. For each query, MERCI iterates through features and identifies the clusters they belong to using remapped feature IDs and the cluster group metadata array. If multiple features in a query fall into the same cluster, this implies an opportunity for partial reduction as their partial sum is already memoized. Thus, MERCI retrieves sub-query partial sums for all clusters and calculates final reduction results.

Figure 9 illustrates how MERCI processes a query in detail. Given

```
1  def getOffset (GroupInfo& g, int prev_cid,
2                                      vector<int> features):
3    int combi_offset = 0;
4    int first_f = features[0];
5    /* Iterate features for bit vector representation */
6    for f in features:
7      combi_offset |= (1 << (f-first_f));
8    /* Compute cluster offset */
9    int cluster_offset = g.offset
10                   + prev_cid*(pow(2, g.csize)-1);
11   return combi_offset + cluster_offset;
12
13 def query_processing (vector<int> query[B],
14                                           float &res[B][D]):
15   for qid = 0 to B-1:
16     /* Initialize cid & gid with first feature */
17     GroupInfo g = getGroup(query[qid][0]);
18     int prev_cid = (query[qid][0] - g.first_id)/g.csize;
19     int prev_gid = g.gid;
20
21     vector<int> features; // Features in the same cluster
22     for fid in query[qid]:
23       int cid = (fid - g.first_id)/g.csize;
24       /* cluster not changed */
25       if (prev_cid == cid && g.gid == prev_gid):
26         features.push_back(fid);
27       /* cluster changed */
28       else:
29         /* Get memoization table index for features */
30         int offset = getOffset(g, prev_cid, features);
31         for d=0 to D-1:
32           res[qid][d] += memoization_table[offset][d];
33         prev_cid = cid;
34         prev_gid = g.gid;
35         features = {fid};
36         GroupInfo new_g = getGroup(fid)
37         /* Group changed, update group info */
38         if(prev_gid != new_g.gid)
39           g = new_g
40     /* Omitted: Handle the last cluster here */
```

Figure 10: Query processing pseudocode, omitting the details for multi-threading.

a query {0, 3, 4, 6, 7, 9, 12}, MERCI processes each feature sequentially starting from first feature in the query (i.e., feature 0). For each feature, MERCI finds out each feature's cluster group by comparing its ID with groups' first IDs in group meta-data array. Figure 9 assumes feature 0, 3, 4 and 6 are already processed, and feature 7 is about to be processed. Feature 7 is in Cluster Group 1 because ID 7 is less than Group 2's first ID 10 but greater than Group 1's first ID 4. Then, its cluster ID can be obtained by dividing offset within a cluster group (feature ID - group first ID) by cluster size (`csize`) of that group. Hence, Feature 7 is in Cluster 1 (= $\lfloor \frac{7-4}{3} \rfloor$). Likewise, Feature 9 is in Cluster Group 1, Cluster 1, and Feature 12 is in Cluster Group 2, Cluster 0. Then, MERCI detects a change in the cluster and collects previous features in the same cluster ({7, 9}) to calculate the memoization table offset of their reduction. The location of cluster's reduction results (i.e., *cluster offset*) is computed by {cluster group base offset} + cluster ID$\times(2^{cluster_size} - 1)$. For instance, Cluster Group 1, Cluster 1's *cluster offset* is 22 (= 15 + 1$\times$ $(2^3 - 1)$). The exact offset is *cluster offset+combination offset*, which can be obtained by representing the current feature set using a bit vector, as explained in Section 5.1 and Figure 8.

Figure 10 again shows this process in a pseudocode. Function

query_processing stores reduction results of B queries. For features in a query, it identifies their cluster group IDs (g.gid) and the cluster IDs (cid) (Line 23, 36). Then it compares g.gid and cid of the current feature (fid) with that of the previous feature to collect features in the same cluster (Line 24-26). If the cluster has changed, combination offset (Line 3-7) and the previous feature set's cluster offset (Line 9) is calculated in function getOffset, and the reduction result is updated (Line 31-32). Note that the last feature set of the query needs to be handled after the loop, but we omit that part for the brevity.

6 DISCUSSION

6.1 Time Complexity of Correlation-Aware Variable-Sized Clustering

Correlation-Aware Variable-Sized Clustering is performed on each super-partition, and hence it only considers clusters within the same super-partition as potential merge candidates. The time complexity of our clustering scheme is $O(NS|Q|)$, where N is the number of embedding vectors (i.e., the number of features), S the size of each super-partition, and $|Q|$ the number of queries in the training set. For each merge, the scheme needs to evaluate the benefit and the cost of merging the different pairs of clusters. Here, there exist at most S remaining clusters, and evaluating the benefit of merging two clusters requires at most $|Q|$ operations as it is simply an intersection of two inverted indices whose size is bounded to $|Q|$. As a result, each merge requires at most $O(S|Q|)$ operations. In the worst case, the merge needs to be performed N times (i.e., S merges for N/S super-partitions), making the total time complexity $O(NS|Q|)$. We have empirically confirmed that a choice of small S (e.g., 128) is nearly as effective as a larger S such as 1024, and expect that even larger S does not substantially boost the performance. This implies that the number of co-appearing features for a single feature does not exceed 128 on average in the datasets used for evaluation.

If we assume that there was only one super-partition (e.g., $N = S$), the time complexity of a single merge becomes $O(N|Q|)$. In the worst case, the merge needs to be performed for N times, meaning that the time complexity without the hypergraph partitioning step (Section 4.1) is $O(N^2|Q|)$. This is impractical, especially given that N is often an order of millions. Therefore, the step of hypergraph partitioning is justified to keep the time complexity manageable.

6.2 Capacity Cost

Some large-scale recommendation systems [56, 57] already require an enormous capacity to store embeddings, and at a glance, it may seem applying MERCI on such systems is impractical due to the additional capacity cost that memoization requires. For such models, naively using MERCI for all embedding tables may incur an excessive capacity cost. To avoid such a huge capacity cost, we envision that it is possible to selectively apply MERCI for some embedding tables (or a subset of a single embedding table) that are i) frequently accessed, ii) reasonably sized, and iii) have high locality. In fact, several existing literature state that the large-scale recommendation model utilizes multiple separate embedding tables [37], and some embedding tables exhibit higher locality than the others [19]. Furthermore, as explained in Section 4.2, MERCI allows users to specify the limit of capacity overhead from the memoization table.

Table 1: Dataset analysis.

Name	# of Features	# of Queries	Avg. Query Len.	Embedding Tbl. Size	MemTable Size (+8×)
Synthetic datasets					
Synthetic 1	1,000K	1,000K	60.0	244MB	2.08GB
Synthetic 2	1,000K	1,000K	54.0	244MB	2.06GB
Synthetic 3	1,000K	1,000K	51.0	244MB	2.19GB
Synthetic 4	2,000K	2,000K	60.0	488MB	4.18GB
Synthetic 5	2,000K	2,000K	54.0	488MB	4.09GB
Synthetic 6	2,000K	2,000K	51.0	488MB	4.36GB
Real-world datasets					
Books	3,187K	32,305K	72.796	568MB	5.20GB
Electronics	759K	10,711K	55.746	115MB	1.06GB
Clothing	2,345K	4,137K	81.953	224MB	2.06GB
Sports	1,506K	5,998K	96.019	196MB	1.75GB
Office Products	599K	3,736K	64.088	85MB	0.73GB
Home & Kitchen	1,806K	11,270K	51.476	248MB	2.23GB
Last.fm	636K	534K	95.611	104MB	0.88GB
DBLP	540K	479K	61.780	102MB	0.87GB

6.3 Handling Embedding Table Updates and Query Access Pattern Changes

Embedding vectors are known to be frequently retrained every few hours [19]. In that case, the memoization table needs to be updated as well. However, the time to update the memoization table is relatively smaller than time to retrain the embedding vectors. And simply updating the embedding vectors does not require MERCI to perform offline clustering (i.e., Hypergraph partitioning and Correlation-Aware Variable-Sized Clustering) again. In contrast, when the query access pattern changes, hypergraph partitioning (i.e., Step 1 in 4.1) and clustering (i.e., Step 2 in 4.2) need to be performed again. In practice, this happens much less frequently than the embedding vector change in many recommender systems. For all our workloads, clustering and partitioning are completed within 10 minutes with a 16-core machine (the same as in 7). Naturally, this time can be further reduced with the use of a better machine or the use of multiple machines.

7 EVALUATION

7.1 Datasets

Real-world Datasets. For the assessment of our algorithm, we utilize popular public datasets for the recommender systems: the Amazon Review dataset (books, electronics, clothing, shoes, and jewelry, sports and outdoors, office products, and home and kitchen) [28], Last.fm Million Songs dataset [5], and DBLP Co-Authors Network dataset [46]. Although it would be ideal to use the feature traces from actual recommender models such as Facebook DLRM [41], such production traces are not publicly released.

Each dataset was parsed into a format suitable for our use. Queries and features were defined in a way they would be in a recommender system. For instance, in the Amazon Review dataset, we defined a feature as a product for sale on Amazon, and query as a group of products (features) a reviewer bought or viewed together. Then, queries were randomly partitioned into train and test sets at the ratio of 8:2. Queries in the train set are used by Correlation-Aware Variable-Sized Clustering to calculate the benefit and cost during offline clustering, and queries in the test set were utilized

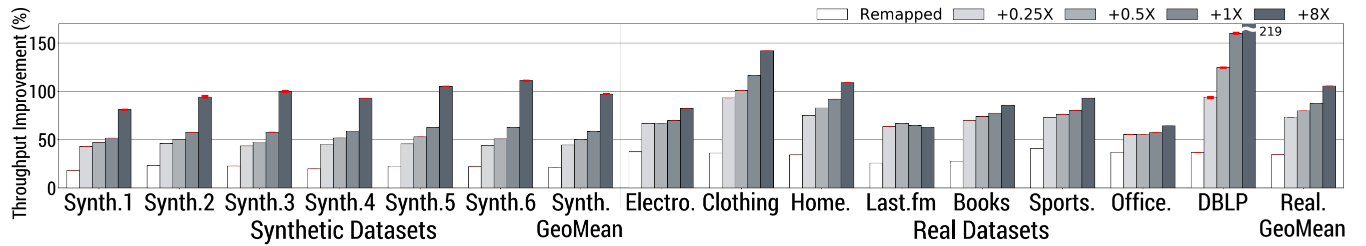

Figure 11: Throughput improvement.

to simulate query processing with the memoization table. Specific statistics regarding each dataset are delineated in Table 1.

Synthetic Datasets. We also evaluate MERCI on synthetic datasets. For synthetic dataset generation, we employed a technique named Stochastic Block Model (SBM) [54], which is a well known approach for creating a random graph with community structures. We configured the parameters such that 128 features form a single correlative group, and the total number of features be N. Furthermore, each query comes with an average of p features from the same group and an average of q features from different groups. We evaluated datasets with (p, q) pairs of (48, 3), (48, 6), (48, 12) for $N = 1M$ and $N = 2M$. Queries are also randomly partitioned into train and test sets at the ratio of 8:2.

7.2 Methodology

We implemented baseline and MERCI's embedding reduction operation in C++. In both cases, queries are distributed to multiple threads. We functionally verified the correctness of the implementation and checked that the baseline implementation achieves high-performance by confirming that it fully utilizes the system's memory bandwidth. MERCI implementation is available at https://github.com/SNU-ARC/MERCI. We measured the runtime mostly on Amazon Web Services (AWS) EC2 m5.8xlarge instance [3], which provides 16 Intel Xeon Platinum 8259CL CPU cores with 128GiB of DRAM. We also checked the performance sensitivity to machines by evaluating MERCI on desktop-class Intel Core i7-10700K CPU with 64GiB of DRAM. Note that accessing hardware counters is possible only on the local desktop, but not on the Amazon server. Thus, we perform energy and LLC miss analysis on the desktop machine. All evaluations were performed on Ubuntu 18.04 LTS.

7.3 Performance Evaluation

Throughput. Figure 11 delineates the throughput improvement of MERCI. The x-axis denotes the size of the memoization table over the original embedding table. We limited the additional memory usage incurred by MERCI's memoization table to 0.25, 0.5, 1 and 8 times the size of the original embedding table. Note that MERCI uses both the original embedding table and the memoization table, so this implies the pure memory consumption from memoization. All configurations in the figure (*Remapped*, +0.25x, +0.5x, +1x, +8x) hypergraph-partitioned N features into super-partitions of size 128. The embedding dimension is set to a constant value of 64 (i.e., 64 elements per embedding vector), and all measurements were repeated five times and averaged. Error bars are expressed in red lines but too minuscule to notice.

The bars labeled *Remapped* refer to *remapped-baseline* whose height denote runtime speedup without memoization. In *Remapped*, the N features are hypergraph-partitioned into super-partitions of size 128 and remapped so that those in the same super-partition are assigned consecutive IDs. Hence, *Remapped* shows the pure effect of locality-aware ID remapping without our clustering algorithm and memoization. As shown in Figure11, considering locality at coarse-grained granularity improves by 29%. Clearly, clustering and memoization give substantial extra speedup on top of ID remapping.

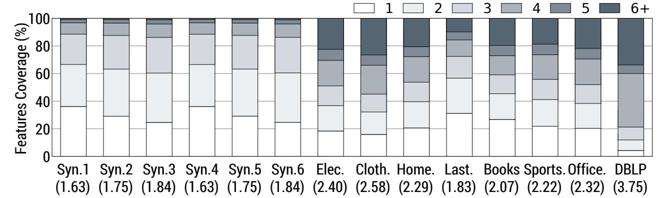

Figure 12: Feature coverage per memoization size.

Across all datasets, MERCI manifests significant throughput improvement of 62%–262%, and achieves a geomean speedup of 102% when the memoization table size is at +8×. As shown in the figure, it is possible to obtain a decent speed up of 52%–160%, and 74% on average when the table size is limited at +1×. Even for smaller table size which is limited at +0.25× and +0.5×, MERCI achieves 60% and 66% on average, respectively. In general, increasing the memoization table size leads to further speedup, but with diminishing returns. This is because MERCI first utilizes the capacity for the most popular co-appearing combinations and then utilizes extra capacity for the less frequently co-appearing ones.

Memoization Size Analysis. Figure 12 analyzes feature coverage of memoization table access by memoization size (i.e., the number of features aggregated together). This is an actual clustering result that derived the speedup in Figure 11. Each section in a stacked bar represents the percentage of features covered by a given memoization size, summing up to the total number of demanded features in the test set (i.e., $\sum_{qid=1}^{|Q|} query[qid].size()$). For instance, in the Amazon Office Products dataset, 18.6% of all features were retrieved as a reduction of four features, thus quartered table access count for that portion. On average, 75.4% of features were accessed by a memoization size greater than one. In the x-axis, the number in parenthesis indicates the average number of features retrieved per memoization table access. The result shows that, and single table access covers from 1.63 to 3.75 features on average.

The figure illustrates that MERCI effectively memoized the embedding table as a large portion of memory accesses reads amassed features. Even when MERCI retrieves the memoization value of size 1, it is still superior to the baseline because it benefits from locality-aware feature ID remapping, as discussed in Figure 11.

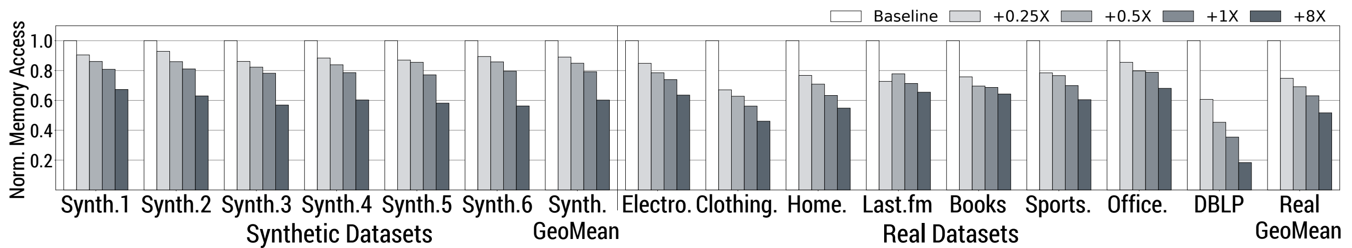

Figure 13: Memory access count reduction.

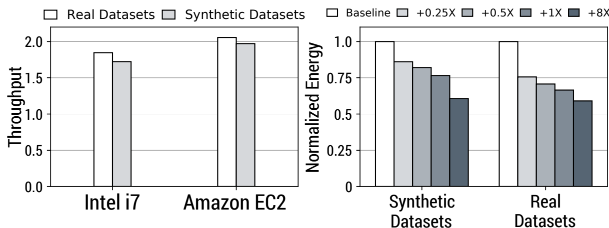

Figure 14: (a) Machine sensitivity (b) Energy consumption.

7.4 Evaluation on Desktop Platform

Machine Sensitivity. Figure 14(a) explores the MERCI performance sensitivity on different machine configurations with +8× memoization table. As shown in Figure 11, MERCI demonstrates largely similar performance improvements by 102% on the server system (AWS EC2 m5.8xlarge instance) and 78% on the desktop system (Intel Core i7-10700K CPU with two memory channels). The result indicates that MERCI can achieve speedup on any system whose embedding reduction performance is bound by memory bandwidth. Technically, MERCI may not show a performance improvement on some systems with very abundant memory bandwidth and a small number of cores. In practice, however, such systems are rare as they would heavily underutilize the memory bandwidth in many conventional operations.

Energy Savings. Figure 14(b) shows total energy consumption normalized to baseline energy consumption. We measured energy consumption with Intel Running Average Power Limit (RAPL) interface [15]. MERCI significantly saves energy consumption by 40.2% on average (up to 63.5%) at +8× configuration.

Memory Access Count. Figure 13 shows memory access count (i.e., reads+writes) during MERCI's query processing normalized to baseline memory access count. For memory access count measurement, we utilized Intel VTune Profiler [12]. As in previous sections, memoization table sizes were set at +0.25, +0.5, +1 and +8 times the original embedding table, and the embedding dimension was set to a constant value of 64. The graph shows that MERCI's total memory access count decreases by 48%, 40% for real and synthetic datasets at +8× memoization table. MERCI successfully accelerated memory-bound embedding reduction operation by reducing the actual count of memory accesses, which decreases as we use more memory for storing memoization results. We also measured the memory bandwidth utilization of both the baseline and MERCI using Intel VTune Profiler. Both systems almost fully utilize the available memory bandwidth (e.g., 90+% of the theoretical peak bandwidth), and this indicates that the memory access reduction shown in Figure 13 directly translates to the throughput improvement.

8 RELATED WORK

Frequent Pattern Mining. Frequent pattern mining algorithms such as apriori [2], FP-growth [25], and DHP [43] algorithm can be utilized to identify sets of frequently co-appearing features. However, the main drawback of these algorithms is that they simply find multiple sets of co-appearing features, allowing a single feature to belong in multiple sets. In such a case, unlike our clustering approach, retrieving the reduction results becomes much more difficult. Specifically, i) identifying the relevant partial sums for a given query and ii) finding where they are located become serious challenges. It is our design choice to give up some extra reduction opportunities for efficient retrieval of partial sums.

Hardware Solutions for the Embedding Reduction. Recently, Facebook[19, 24, 41, 42], Google[16], and Alibaba[51] emphasize that embedding reduction is memory-bound and takes a significant portion of runtime. Several works addressed this problem with hardware support. For example, [32, 34] adopted near-memory processing (NMP) architecture to exploit the abundant internal bandwidth to perform reduction, and only passes the reduction outcome to the external device through links with lower bandwidth. Centaur [29] is a chiplet-based hybrid accelerator that also includes embedding reduction as its target. These solutions report that they achieve up to an order of magnitude performance improvements or traffic reductions based on their simulation results. However, solutions that require hardware support are often expensive. On the other hand, our proposal is an immediately deployable solution that provides a substantial speedup at the cost of extra memory capacity.

Feature-aware Optimizations. Bandana [19] utilizes hypergraph partitioning to place embedding vectors that are likely to be accessed together in a same 4KB NVM block. Bandana aims to reduce DRAM capacity consumption under the same number of memory access count while our work aims to reduce the number of memory access count itself.

Memoization. Since memoization was first introduced at [38], memoization is widely adopted as a key technique to accelerate specific target. COREx[21] scales datacenter accelerators via memoization. Specifically, it proposes an accelerator and a storage layer that memoize and reuse the outcome of previously accelerated computations when the accelerator needs to compute the same thing. Other proposals [6, 11, 36, 45, 50] identify computation redundancy caused by similarities in the input within the various granularity (e.g., instruction, function, task level) and memoize them. Thus these works avoid processing the same set of instructions and rather replace such memoized regions with much simpler operations. In [55], they propose a technique that can be used to accelerate memoization.

9 CONCLUSION

In this paper, we propose MERCI, a memoization framework that addresses the memory bandwidth bottleneck of embedding reduction without distinct hardware support. Embedding reduction is an accumulative operation that is comprehensively utilized in modern neural network models and is well-known to be bounded by memory bandwidth. Its defect is that it does not endorse the correlation among inputs features and always load embeddings in succession. Hence, MERCI tackles this complication with memoization. Bolstered by our novel clustering scheme, MERCI accelerates embedding reduction by identifying co-appearing features, memoizing partial reduction of such features, and constructing a memoization table that supports fast access to memoized values on-demand.

ACKNOWLEDGMENTS

This work was supported by a research grant from SK Hynix and by research grants from the Korea Government (MSIT): Institute for Information & communications Technology Promotion (IITP) grant (2014-0-00035, Research on High Performance and Scalable Manycore Operating System) and the National Research Foundation of Korea (NRF) grant (NRF-2020R1A2C3010663, Research on NAND Flash-Based DNN Training System). Tae Jun Ham is the corresponding author.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

[2] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. *Proceedings of the International Conference on Very Large Data Bases (VLDB).*

[3] Amazon. Amazon EC2 M5 Instances. https://aws.amazon.com/ec2/instance-types/m5/.

[4] B. Barz and J. Denzler. 2019. Hierarchy-Based Image Embeddings for Semantic Image Retrieval. *In proceedings of IEEE Winter Conference on Applications of Computer Vision (WACV).*

[5] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. *In proceedings of the International Conference on Music Information Retrieval (ISMIR).*

[6] I. Brumar, M. Casas, M. Moreto, M. Valero, and G. S. Sohi. 2017. ATM: Approximate Task Memoization in the Runtime System. *In proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[7] Miguel Campo, Cheng-Kang Hsieh, Matt Nickens, J. J. Espinoza, Abhinav Taliyan, Julie Rieger, Jean Ho, and Bettina Sherick. 2018. Competitive Analysis System for Theatrical Movie Releases Based on Movie Trailer Deep Video Representation. *CoRR* abs/1807.04465 (2018).

[8] Ümit V. Çatalyürek and Cevdet Aykanat. 2011. PaToH (Partitioning Tool for Hypergraphs). In *Encyclopedia of Parallel Computing*. 1479–1487.

[9] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *In proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[10] 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. *In proceedings of Workshop on Deep Learning for Recommender Systems (DLRS).*

[11] D. A. Conners and W. W. Hwu. 1999. Compiler-directed dynamic computation reuse: rationale and initial results. *In proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO).*

[12] Intel Corporation. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[13] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. *In proceedings of the ACM Conference on Recommender Systems (RecSys).*

[14] W. Bruce Croft, Donald Metzler, and Trevor Strohman. 2010. *Search engines: information retrieval in practice* (1st ed.). Addison-Wesley, Boston.

[15] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. 2010. RAPL: Memory power estimation and capping. *ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED).*

[16] J. Dean, D. Patterson, and C. Young. 2018. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro* 38, 2 (2018).

[17] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, and Umit V Catalyurek. 2006. Parallel hypergraph partitioning for scientific computing. *In proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *In proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers) (NAACL).*

[19] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. *Proceedings of Machine Learning and Systems (MLSys).*

[20] Facebook. Caffe2. https://caffe2.ai.

[21] Adi Fuchs and David Wentzlaff. 2018. Scaling Datacenter Accelerators with Compute-Reuse Architectures. *In proceedings of the Annual International Symposium on Computer Architecture (ISCA).*

[22] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. *In proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, Carles Sierra (Ed.).

[23] 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 proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[24] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. *In proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[25] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. *In proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD).*

[26] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *In proceedings of the International Symposium on Computer Architecture (ISCA).*

[27] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[28] 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 International Conference on World Wide Web (WWW).*

[29] R. Hwang, T. Kim, Y. Kwon, and M. Rhu. 2020. Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. *In proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[30] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. 2017. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. *In proceedings of the VLDB Endowment* (2017).

[31] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1999. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration Systems* 7, 1 (1999).

[32] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David 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, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. *In proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[33] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings to Document Distances. *In proceedings of the International Conference on Machine Learning (ICML).*

[34] 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 IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[35] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019).

[36] Zhenhong Liu, Amir Yazdanbakhsh, Dong Kai Wang, Hadi Esmaeilzadeh, and Nam Sung Kim. 2019. AxMemo: Hardware-Compiler Co-Design for Approximate Code Memoization. *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19).*

[37] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding Capacity-Driven Scale-Out Neural Recommendation Inference.

[38] Donald Michie. 1968. "Memo" functions and machine learning. *Nature* 218, 5136 (1968).

[39] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. *In proceedings of the International Conference on Neural Information Processing Systems (NIPS).*

[40] Maxim Naumov. 2019. On the Dimensionality of Embeddings for Sparse Features and Data. *CoRR* abs/1901.02103 (2019).

[41] 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).

[42] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Shanker Khudia, James Law, Parth Malani, Andrey Malevich, Nadathur Satish, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim M. Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *CoRR* abs/1811.09886 (2018).

[43] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. 1995. An Effective Hash-Based Algorithm for Mining Association Rules. *In proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD).*

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. 8024–8035.

[45] Stephen E. Richardson. 1992. *Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation.* Technical Report.

[46] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. *AAAI*.

[47] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. K-way hypergraph partitioning via n-level recursive bisection. *In proceedings of Workshop on Algorithm Engineering and Experiments (ALENEX).*

[48] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019).

[49] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. 2019. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer. *CoRR* abs/1904.06690 (2019).

[50] Tomoaki Tsumura, Ikuma Suzuki, Yasuki Ikeuchi, Hiroshi Matsuo, Hiroshi Nakashima, and Yasuhiko Nakashima. 2007. Design and Evaluation of an Auto-Memoization Processor. *In proceedings of the IASTED International Multi-Conference: Parallel and Distributed Computing and Networks (PDCN).*

[51] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. *CoRR* abs/1803.02349 (2018).

[52] P. Wang, Z. Liu, H. Wang, and D. Wang. 2017. Data-centric computation mode for convolution in deep neural networks. *In proceedings of the International Joint Conference on Neural Networks (IJCNN).*

[53] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. *CoRR* abs/1708.05123 (2017).

[54] Yuchung J Wang and George Y Wong. 1987. Stochastic blockmodels for directed graphs. *J. Amer. Statist. Assoc.* 82, 397 (1987).

[55] Guowei Zhang and Daniel Sanchez. 2019. Leveraging Caches to Accelerate Hash Tables and Memoization. *In proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[56] 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. *arXiv preprint arXiv:2003.05622* (2020).

[57] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. *In proceedings of the ACM International Conference on Information and Knowledge Management (CIKM).*

[58] 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 ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD).*