HEAT: A Highly Efficient and Affordable Training System for Collaborative Filtering Based Recommendation on CPUs

Chengming Zhang
Indiana University
czh5@iu.edu

Shaden Smith
Microsoft
shaden.smith@microsoft.com

Baixi Sun
Indiana University
sunbaix@iu.edu

Jiannan Tian
Indiana University
jti1@iu.edu

Jonathan Soifer
Microsoft
jonso@microsoft.com

Xiaodong Yu
Argonne National Laboratory
xyu@anl.gov

Shuaiwen Song
Microsoft
shuaiwen.song@microsoft.com

Yuxiong He
Microsoft
yuxhe@microsoft.com

Dingwen Tao*
Indiana University
ditao@iu.edu

ABSTRACT

Collaborative filtering (CF) has been proven to be one of the most effective techniques for recommendation. Among all CF approaches, SimpleX is the state-of-the-art method that adopts a novel loss function and a proper number of negative samples. However, there is no work that optimizes SimpleX on multi-core CPUs, leading to limited performance. To this end, we perform an in-depth profiling and analysis of existing SimpleX implementations and identify their performance bottlenecks including (1) irregular memory accesses, (2) unnecessary memory copies, and (3) redundant computations. To address these issues, we propose an efficient CF training system (called HEAT) that fully enables the multi-level caching and multi-threading capabilities of modern CPUs. Specifically, the optimization of HEAT is threefold: (1) It tiles the embedding matrix to increase data locality and reduce cache misses (thus reduce read latency); (2) It optimizes stochastic gradient descent (SGD) with sampling by parallelizing vector products instead of matrix-matrix multiplications, in particular the similarity computation therein, to avoid memory copies for matrix data preparation; and (3) It aggressively reuses intermediate results from the forward phase in the backward phase to alleviate redundant computation. Evaluation on five widely used datasets with both x86- and ARM-architecture processors shows that HEAT achieves up to 65.3× speedup over existing CPU solution and 4.8× speedup and 7.9× cost reduction in Cloud over existing GPU solution with NVIDIA V100 GPU.

CCS CONCEPTS

• **Information systems → Collaborative filtering**; • **Computing methodologies → Shared memory algorithms**.

KEYWORDS

Recommender system; multi-core processor; performance.

*Corresponding author: Dingwen Tao.

ACM Reference Format:
Chengming Zhang, Shaden Smith, Baixi Sun, Jiannan Tian, Jonathan Soifer, Xiaodong Yu, Shuaiwen Song, Yuxiong He, and Dingwen Tao. 2023. HEAT: A Highly Efficient and Affordable Training System for Collaborative Filtering Based Recommendation on CPUs. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the development and popularization of the Internet and smart devices, these platforms have become ideal tools for collecting various data/information that can be used to identify user preferences [27]. However, the exponential growth in the amount of digital information and the explosion in the number of Internet users can lead to the problem of information overload that prevents timely access to things of interest on the Internet [23]. As a result, recommender systems (a.k.a recommendation systems) that use a user's choices, interests, or observed behavior to filter out key information from a massive amount of dynamically collected information are more in demand than ever [32].

Collaborative filtering (CF) has been proven to be one of the most effective techniques for building recommender systems due to its ability to recommend completely dissimilar content. Learning effective latent factors directly from the user-item rating matrix through matrix factorization (MF) is the most effective method among CF-based approaches [29] (will be discussed in §2.2). Despite the effectiveness of MF-based CF, training MF-based CF is challenging due to two reasons: (1) **Performance:** Irregular memory access causes a significant performance degradation when using sparse real-world user-item rating matrices [14]. (2) **Cost:** Weekly or monthly updates on large training datasets (millions of users and items) causes a drastic cost increase. Recommendation tasks in the industry are time-sensitive and profit-oriented, which means that training time and cost are critical for entrepreneurs.

Although today's various accelerators such as graphics processing units (GPUs) are widely used to train machine learning models today, in this work we focus on using multi-core CPUs for training recommendation models due to the following three main reasons.

CF training with GPUs is expensive. Nowadays, users typically accelerate machine learning applications [43] using GPUs that are capable of performing floating-point operations in a massively parallel fashion. However, CF training on large datasets using GPUs is expensive. For example, training an MF-based CF model for 3,000 epochs on a dataset with 200 million users and 200 million items takes about 297.7 hours using 100 16 GB V100 GPUs. Assuming this training service is deployed on an AWS p3.2xlarge instance [3] at an hourly cost of around 3 dollars, the total cost for one-time training is approximately 91,081 dollars. Furthermore, the model needs to be retrained every month or week due to the dataset update.

GPU memory is limited. Besides the user-item rating matrix as input, training MF-based CF also requires holding two large embedding matrices in memory, i.e., a user embedding matrix $S \in \mathbb{R}^{|U| \times K}$ and an item embedding matrix $T \in \mathbb{R}^{|I| \times K}$. Here $|U|$ is the number of users, $|I|$ is the number of items, and K is usually between 64 and 128. For instance, the Amazon Product Reviews dataset [18, 19] contains 21 million users and 9 million items. However, currently the most powerful GPUs have only 80 GB of memory [11], which can only accommodate embedding matrices of up to 4 million users/items when K is 128 and and the data type is float32. For larger datasets, we need to split embedding matrices and rating matrix across different GPUs, which introduces high global communication overhead. In comparison, the popular CPU nodes in HPC systems typically have 512 GB or even 1-2 TB of memory.

MF-based CF is not suitable for GPUs. An MF-based CF features low computation intensity and highly irregular memory accesses. For MF-based CF training, we need to maximize the similarity of a positive user-item pair while minimizing the similarity of a negative user-item pair. This procedure requires fetching (1) a K-dimensional user embedding (vector) from the user embedding matrix, (2) one K-dimensional positive item embedding from the item embedding matrix, and (3) random n K-dimensional negative item embeddings from the item embedding matrix. Based on these sampled vectors, we need to compute the similarity of the user-item pairs using vector-dot product operations. These computation patterns (i.e., embeddings are first accessed in an irregular fashion and then used in a regular way with spatial locality for low computation-intensive vector products) make MF-based CF training more suitable for CPUs than GPUs.

Prior works have focused on optimizing the performance MF-based CF training. For example, MSGD [30] improves training performance on GPUs by removing dependencies on user and item pairs. However, MSGD does not support sampling multiple negative terms, which leads to inferior training results (i.e., low accuracy). Recently, Mao *et al.* proposed SimpleX [31], a state-of-the-art CF method, that has a novel loss function and a large negative sampling rate, greatly outperforming other existing methods. However, SimpleX only uses PyTorch to implement its approach without considering the computational efficiency. Specifically, (1) training on sparse user-item rating matrices and random sampling for multiple negative items lead to irregular memory accesses to embedding matrices. (2) The similarity computation before the loss computation is usually based on parallel matrix-matrix multiplication, which introduces expensive memory copies to concatenate sampled vectors into matrices. (3) Automatic differentiation engines in machine learning frameworks (such as `autograd` in PyTorch) ignore potential data reuse in the backward phase (see §4.4).

To this end, we propose a Highly Efficient and Affordable Training system (called HEAT[1]) for collaborative filter based recommendation on multi-core CPUs based on the SimpleX approach. First, we propose to take advantage of modern CPUs' memory hierarchies to reduce embedding read latencies. Specifically, we propose an effective tiling method that partitions item embedding matrices to fit into multi-level caches according to their sizes. Second, we adopt a multi-threaded training method [36], where each thread independently and parallelly reads its corresponding user and item embeddings, calculates their gradients, and updates them rather than all embeddings, and fuse forward and backward phases to reduce the size of the per-thread memory footprint. Third, we identify reuse opportunities for intermediate results during the backward pass of training. This reuse is missed in automatic differentiation systems that work at a more fine-grained operator level. To the best of knowledge, *this is the first work that enables high-performance and low-cost CF training for recommendation based on the SimpleX approach on multi-core CPUs.*

The main contributions of this paper are summarized as follows:

- We deeply analyze the performance of two state-of-the-art MF-based CF solutions and identify their performance bottlenecks.
- We propose to tile the item embedding matrix according to multi-level cache sizes to reduce read latency. Furthermore, we propose a light-weight algorithm to find the optimal tiling size and cache eviction policy (e.g., refresh interval).
- We develop a parallel method for similarity computation based on vector products rather than matrix-matrix multiplication to avoid matrix data preparation (i.e., memory copies).
- We propose to save the result of the partial derivative of embeddings in the forward computation and reuse them in the backward computation to avoid redundant calculations.
- We propose two implementation optimizations to improve the performance of weight updates by alleviating read/write conflicts in shared memory and reducing the amount of updates.
- Evaluation on three real-world datasets with AMD 7742 CPUs and Fujitsu A64FX CPUs shows that HEAT achieves up to 65.3× and 4.8× speedups over state-of-the-art CPU and GPU solutions, respectively. We also derive some takeaways for CF training on different CPU architectures.

In §2, we present the background about recommender systems and matrix factorization based collaborative filtering. In §3, we present our profiling and analysis of existing solutions. In §4, we describe the design of our HEAT. In §5, we evaluate HEAT on different datasets and compare it with other works. In §6, we discuss related work. In §7, we conclude our work and discuss future work.

2 BACKGROUND

In this section, we present the background about recommender systems and collaborative filtering techniques for recommendation.

2.1 Basics of Recommender Systems

Input Data. There are two main types of collected feedback (i.e., rating matrix): (1) explicit feedback [28] that is directly provided by users, such as likes and ratings, and (2) implicit feedback [6] that

[1]The code is available at https://github.com/hipdac-lab/HEAT.

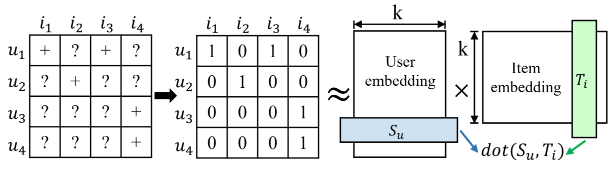

Figure 1: Basic concept of MF-based CF.

is obtained from users' interactions, such as click data, purchases, and implicit visit information. Recent research on recommender systems has shifted from explicit feedback to implicit feedback [13] because the majority of a user's preference-related signal is implicit. Thus, we focus on implicit feedback in this work.

Filtering Techniques. Recommender systems mainly include content-based [8] filtering and collaborative filtering (CF) [21] techniques. Content-based filtering is based on the items' information and recommends items that have attributes similar to those that users like. However, the technique is notorious for its inability to recommend dissimilar items [39]. To address this issue, collaborative filtering makes recommendations by learning preferences or taste information from many other users' interactions [21] and is able to provide diverse recommendations. CF techniques can be further classified into user–user CF [38], item-item CF [40], dimensionality reduction [7], and probabilistic methods [26]. Specifically, user–user and item–item CF techniques directly use feedback to calculate similarities between users or items. But vectors in feedback are highly sparse and have extremely large dimensions. For example, an item is a $|U|$-dimensional vector and a user is a $|I|$-dimensional vector, where U is the set of all users and I is the set of all items. This causes high overhead of computing resources and memory space. To address this issue, dimensionality reduction techniques such as matrix factorization (MF) reduce the dimension of the rating space to a constant number K [41] thereby reducing computational complexity and memory requirements (will be detailed in the next section). Other techniques such as probabilistic methods seek to create probabilistic models of users' behaviors and employ those models to predict users' future behaviors.

Software Frameworks. There are two popular training frameworks to implement recommender systems: (1) PyTorch provides a lookup table (`torch.nn.Embedding`) to store embeddings of a fixed dictionary and size. Users can build a complete model using necessary modules (e.g., loss function, optimizer) provided by PyTorch. Besides, users can utilize the `autograd` module to compute gradients and then update the embeddings. However, on the one hand, using `torch.nn.Embedding` with dense gradient would directly update all embeddings, which leads to unnecessary operations since only part of the embeddings are involved in one training iteration; on the other hand, using `torch.nn.Embedding` with sparse gradient causes worse performance (detailed in Section §3.1). (2) TorchRec [24] is a production-quality recommender systems package in the open-source PyTorch ecosystem. It provides model and data parallelism and represents sparse inputs by jagged tensors. Moreover, TorchRec supports computations on sparse data through FBGEMM [25] and overlaps communication and computation through train_pipeline. However, similar to PyTorch, TorchRec also suffers from the above dense/sparse embedding update issue.

2.2 MF-based Collaborative Filtering

The purpose of MF-based CF training is to maximize the similarity of embeddings of a positive user-item pair while minimizing the similarity of embeddings of a negative user-item pair. We can use dot product similarity or cosine similarity as expressed in Equation 2. Assume U is the set of all users and I is the set of all items. The implicit feedback can be expressed as $X \subseteq U \times I$ as depicted in Figure 1. Particularly, "+" indicates a user's preference for an item. Such corresponding items are called **positive items**. "?" represents either negative (not interested) or missing (not interacted) values. The items corresponding to the negative values are called **negative items**. MF-based techniques train two low-dimensional matrices, i.e., a user embedding matrix $S \in \mathbb{R}^{|U| \times K}$ and an item embedding matrix $T \in \mathbb{R}^{|I| \times K}$, to approximate X as expressed in Equation (1). Then, the main task is to predict missing ratings in X using the corresponding embeddings.

$$X \approx \hat{X} = ST^t \tag{1}$$

$$\hat{x}_{u,i} = \begin{cases} S_u \cdot T_i = \sum_{k=0}^{K} S_{u,k} T_{i,k} \quad \text{(dot)} \\ \frac{S_u \cdot T_i}{\|S_u\|_2 \|T_i\|_2} = \frac{\sum_{k=0}^{K} S_{u,k} T_{i,k}}{\sqrt{\sum_{k=0}^{K} S_{u,k}^2} \sqrt{\sum_{k=0}^{K} T_{i,k}^2}} \quad \text{(cosine)} \end{cases} \tag{2}$$

$$\mathcal{L}(u,i) = (1 - \hat{x}_{u,i}) + \frac{\mu}{|\mathcal{N}|} \sum_{j \in \mathcal{N}} \max(0, \hat{x}_{u,j} - \theta) \tag{3}$$

Prior MF-based CF works can be generally classified into two directions. The first direction only targets recall (i.e., accuracy) and adopts simple similarity functions (e.g., dot product) and point-wise loss functions (e.g., mean square error, binary cross entropy) for user-item pairs without using negative items. For example, representative works such as CuMF_ALS [44], CuMF_SGD [48], and MSGD [30] focus more on the computational efficiency of matrix factorization than on recall. The second direction brings higher accuracy and creates a user-specific item ranking by using the concept of positive/negative items, novel loss functions, and more sophisticated similarity functions (e.g., cosine similarity) with sampling methods. For example, [37] proposed Bayesian personalized ranking (BPR) loss function, while [17, 46] proposed a contrastive loss (i.e., a Euclidean distance-based loss). More recently, SimpleX [31] proposes a cosine contrastive loss (CCL) and utilizes *multiple* negative samples, which outperforms other approaches regarding accuracy. Equation (3) is the CCL, where (u, i) is a positive user-item pair, $\mathcal{N}$ is the number of randomly sampled negative samples, μ is a hyperparameter, and θ is the threshold to filter negative samples.

3 PERFORMANCE PROFILING AND ANALYSIS

In this section, we characterize the performance of SimpleX on both CPU and GPU. Note that we focus on the PyTorch implementation rather than the TorchRec implementation since TorchRec optimizes sparse computation and communication/computation overlap, which does not address the performance bottleneck of MF-based CF training methods. Thus, for simplicity, we only show the performance breakdown of the PyTorch implementation to motivate the design of our HEAT.

We set the embedding dimension to 128 and the number of negatives to 64, and the batch size to 1024. We use PyTorch 1.10.0 and CUDA 10.2. We use the PyTorch profiler [35] to perform the

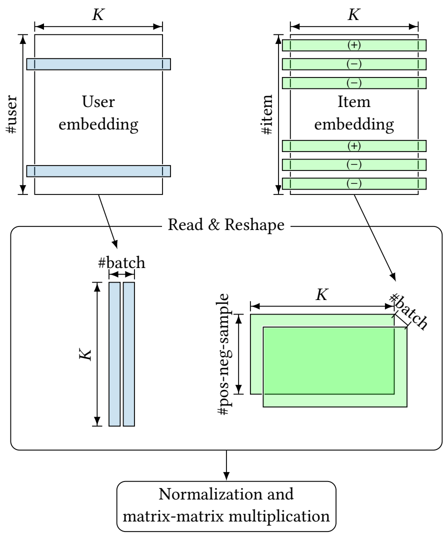

Figure 2: Overview of SimpleX. "+"/"-" denote positive/negative embedding.

Table 1: Profiling of embedding update in SimpleX. "ET", "FP", "BP" are short for epoch time, forward percentage, backward percentage, respectively.

Dataset	Method	ET	FP	BP
AmazonBooks	dense	257.4	19.9%	67.0%
	sparse	946.6	6.2%	92.8%
Yelp18	dense	129	21.2%	65.1%
	sparse	386.3	9.1%	89.3%
Gowalla	dense	94.9	20.7%	66.8%
	sparse	251.8	9.2%	89.2%

breakdown. We use three real-world large datasets (containing more than millions of users and items) for profiling, i.e., Goodreads Book Reviews (Goodreads) [45], Google Local Reviews (2018) (Google) [33], and Amazon Product Reviews (Amazon) [19].

3.1 Embedding Update in SimpleX

The core component in the PyTorch implementation of SimpleX [31] is `torch.nn.Embedding`, a simple lookup table storing embeddings. SimpleX fetches a batch of embeddings from torch.nn.Embedding to perform one training iteration. Logically, we only need to generate the gradients of the involved embeddings and update those embeddings. Thus, we can leverage `torch.nn.Embedding`'s capability which allows users to enable sparse gradient computation and embedding update (by setting the parameter `sparse` to True).

Table 1 shows the profiling results of the embedding update in SimpleX with dense or sparse gradients. In the case of dense gradient, the backward phase takes more than 60% of the epoch time. We observe that `torch.nn.Embedding` updates all embeddings in every iteration. In the case of sparse gradient, although we theoretically reduce the computation complexity, the actual epoch time of training with sparse gradient is almost 3× higher than that of

Table 2: Breakdown of the forward phase of SimpleX. The ratio of each component's time to the forward time. Amazon is short for AmazonBooks.

Dataset	u_emb	i_emb	u_norm	i_norm	mem_cp	bmm	loss
Amazon	2.8%	33.2%	2.1%	40.5%	7.6%	10.2%	3.1%
Yelp18	3.4%	32.4%	4.2%	37.7%	7.1%	10.6%	3.6%
Gowalla	3.7%	31.4%	2.7%	40.9%	7.3%	10.4%	3.5%

Table 3: Memory usage of SimpleX. OoM is short for out of memory.

Dataset	users	items	CPU	GPU
Goodreads	0.81M	1.56M	4.2%	30.1%
Google	4.57M	3.12M	11.3%	80.2%
Amazon	20.98M	9.35M	38.4%	OoM

dense gradient, where the backward phase takes more than 90% of the epoch time. This motivates us to design a training method that supports updating embedding sparsely and efficiently in parallel.

3.2 Computation Efficiency of SimpleX

SimpleX utilizes `torch.bmm`, a batch matrix-matrix product for similarity computation. Before that, it needs to concatenate and then reshape embeddings. Specifically, as shown in Figure 2, SimpleX reads a batch of user embeddings and item embeddings, and reshapes them to let batch dimension be the first dimension. After reshaping, SimpleX performs normalization and matrix-matrix multiplication. `torch.bmm` can fully enable the underlying high-performance BLAS library on multi-core CPUs.

Table 2 shows the breakdown of the forward phase of SimpleX. The forward phase includes reading user embeddings (u_emb), reading item embeddings (i_emb), normalization of user embeddings (u_norm), normalization of item embeddings (i_norm), concatenation and reshaping of embeddings (mem_cp), a batch matrix-matrix product (bmm), and a loss function (loss). We observe that the time of mem_cp and the time of bmm are comparable. This inspires us to avoid explicit concatenation and reshaping. To normalize the embedding tensor E, it needs to sum the square of E along the dimension of the embedding dimension, and then calculate the square root of the summation, and then reverse values of the square root, i.e., the norm $R = \left(\sqrt{E^2.sum(dim = 1)}\right)^{-1}$. We observe that this normalization takes more than 40% of the forward time because of two main reasons: (1) The underlying library does not have good support for the above operations. (2) Reading the entire matrix during the computation and the writing of the generated intermediate tensor R back to the memory cause additional memory access time. In addition, the time of reading item embeddings takes around 30% of the forward time, which is caused by irregular memory accesses.

3.3 Memory Usage of SimpleX

The sizes of user and item embedding matrices in MF-based CF are linearly scaled to the size of training dataset (i.e., item-user rating matrix). Table 3 shows the memory usage of SimpleX on both CPUs and GPU. The total memory capacity of GPU and CPU is 32 GB and 256 GB, respectively. We observe that SimpleX almost runs out of the GPU memory when the numbers of users and items are over 3 millions. This is because it needs to save not only the embedding matrices, but also the gradient matrices (scaled with user/item sizes) and optimizer states (scaled with batch size). The out of memory happens when training on the Amazon dataset due to the limited GPU memory. This observation further strengthens our motivation to use multi-core CPUs with larger memory as our target platform.

4 DESIGN METHODOLOGY

In this section, we propose our multi-threading MF-based CF training system with optimizations to improve the training performance.

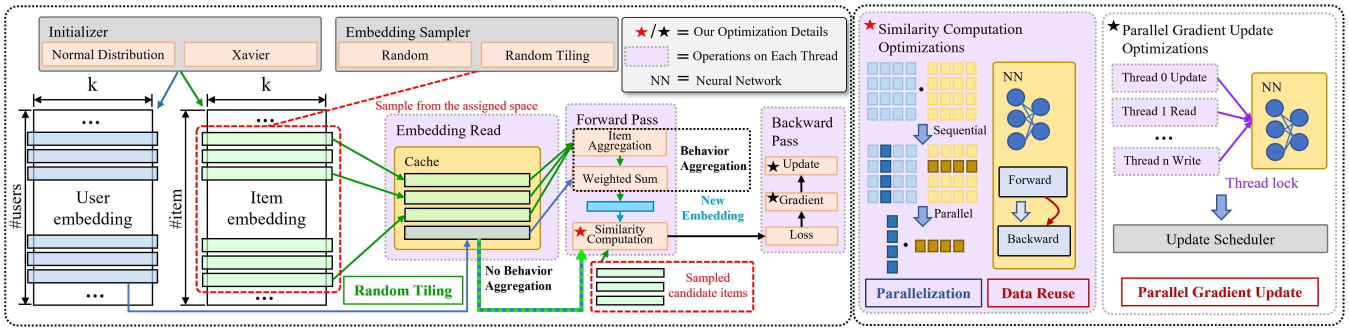

Figure 3: Overview of our proposed HEAT's workflow/dataflow. HEAT has four main optimizations: "random tiling" (§4.2), "parallelization" (§4.3), "data reuse" (§4.4) and "parallel gradient update" (§4.5).

Table 4: Breakdown of the forward phase of HEAT with random sampling.

Dataset	u_emb	i_emb	compute	loss
Amazon	5.1%	63.2%	25.9%	4.3%
Yelp18	5.3%	62.4%	26.6%	4.6%
Gowalla	5.5%	61.4%	26.4%	4.5%

4.1 Overview of HEAT

Figure 3 shows the key components of HEAT: (1) It initializes user/item embedding matrices with values either drawn from the normal distribution $\mathcal{N}(mean, std^2)$ or initialized by Xavier [16]. (2) It chooses either the original random sampler or our proposed random tiling sampler that increases the cache hit ratio (see in §4.2) to sample one user, one positive item, and n negative items from embeddings. Then, it reads the user's corresponding embeddings and these positive/negative items. (3) The behavior aggregation layer with our proposed optimization of gradient update (see §4.5) generates a new user embedding via aggregating embeddings of historically interacted items of the user when enabling behavior aggregation. (4) It calculates similarities in parallel (see §4.3) of user-item pairs and calculate the loss. (5) It calculates gradients through an optimized gradient computation kernel (see §4.4). (6) It updates and writes back corresponding embeddings.

4.2 Random Tiling

Cache size oriented tiling. The original method randomly samples n negative items following a uniform distribution and reads their embeddings. As shown in Table 4, the time of reading item embeddings exceeds 60% of the total forward time. This is due to two reasons: (1) randomly sampled negative items lead to irregular memory accesses, which causes poor data locality, low cache hit rate, and high latency. (2) Each embedding consists of $K (K \geq 64)$ floating-point numbers, which will further exacerbate this problem when K is relatively large. Meanwhile, reading user embeddings takes less than 5% of the total forward time because we only sample one user in each iteration.

To utilize modern CPU's memory hierarchy, especially multi-level caches, we propose to tile the item embedding matrix according to the cache size and make sure a tile of items and their embeddings can be fitted into the cache. Then, we randomly sample negative items directly from the cached tile of items, which increases the cache hit rate and thus reduces the latency of reading embeddings. Assume N_1 and N_2 are tiling size and refresh interval, respectively. The sampling space of the original strategy is whole items. However, the sampling space is shrunk to the tiling size N_1 after applying the tiling strategy. To reduce the impact of the tiling strategy on training results as much as possible, we hope to have as large a sampling space as possible while ensuring acceleration. Thus, we will refresh the cached tile every N_2 iterations to enlarge the sampling space. The sampling space becomes $\frac{M}{N_2} \times N_1$, where M is the number of total iterations.

Figure 4 illustrates the proposed random tiling strategy in each thread. Specifically, each thread preallocates a suitable cache space to buffer random sampled N_1 embeddings. In each iteration, each thread also randomly samples n negative embeddings from the cached tile to compute the gradients, and update corresponding embeddings. After N_2 iterations, each thread randomly samples N_1 embeddings again to refresh the cache space. Since the behavior aggregation layer aggregates embeddings of a user's historical interaction items (i.e., positive items) and one user's negative items may be transformed into another user's positive items, the tiling method can also benefit the behavior aggregation layer.

Tiling size and refresh interval tuning. In order to avoid manually tuning N_1 and N_2 by trial and error, we propose Algorithm 1 to systematically determine N_1 and N_2 given an expected speedup P. Specifically, (1) the negative sampling space of random tiling is determined by $\frac{N_1}{N_2}$ (Line 2) and affects the training results. Thus, $\frac{N_1}{N_2}$ affects the training results. (2) We determine the latency of reading negative embeddings by estimating which level of cache can buffer a tile of embeddings (Lines 5-13). (3) We calculate the total time of reading negative embeddings and speedup after using random tiling, and the speedup can be approximated as $\frac{N_2}{N_1}$ (Lines 15-16). (4) We calculate the speedup of reading positive embeddings after using random tiling (Line 17). (5) Negative and positive speedups for α, β (in percentile) of the total speedup (Line 19). In our design, we set α, β to 0.15 and 0.85, respectively. (6) We first obtain N_1 via function f_0. The main idea of f_0 is to determine a suitable N_1 through the number of threads and the embedding size to ensure

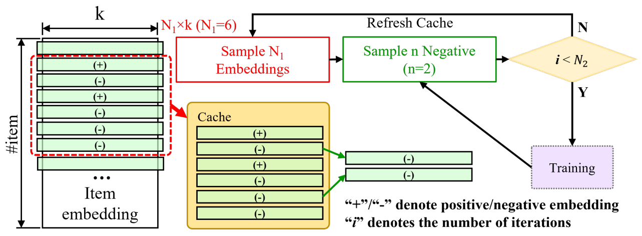

Figure 4: Random tiling strategy in each thread.

Algorithm 1: Proposed tuning method for tiling size & refresh interval.

```
Inputs  : $I$: # of items, $M$: total iterations; $N_1$: tile size; $N_2$: refresh interval; $n_n$: number
          of negatives; $n_p$: number of positives; $r$: average positive hit ratio; $s_{l2}, s_{l3}$: L2,
          L3 cache size; $t_m, t_{l2}, t_{l3}$: latency of reading data from memory, L2 cache, and
          L3 cache; $P$: expected speedup; $\alpha, \beta$: percentage of positive, negative speedup
Outputs : $\widehat{N_1}$: optimized tile size; $\widehat{N_2}$: optimized refresh interval
1  // Negative sampling space of tiling
2  $neg\_space \leftarrow \frac{M}{N_2} \times N_1 = M \times \frac{N_1}{N_2}$
3  // Time of reading negatives using random sampling
4  $neg\_time\_random \leftarrow M \times n_n \times t_m$
5  // Estimate latency of reading cache
6  $s_t \leftarrow N_1 \times sizeof(embedding\ row) \times num\_threads$
7  if $s_t < s_{l2}$ then
8      $t_c \leftarrow t_{l2}$
9  else if $s_t \geq s_{l2}$ and $\mathrm{s_t} < \mathrm{s_{l3}}$ then
10     $t_c \leftarrow t_{l3}$
11 else
12     $t_c \leftarrow t_m$
13 end
14 // Time of reading negatives using tiling
15 $neg\_time\_tiling \leftarrow n_n \times \frac{M}{N_2} \times ((N_2 - N_1) \times t_c + N_1 \times t_m)$
16 $neg\_speedup \leftarrow \frac{neg\_time\_random}{neg\_time\_tiling} = \frac{t_m}{t_c + (t_m - 1) \times \frac{N_1}{N_2}} \approx \frac{N_2}{N_1}$
17 $pos\_speedup \leftarrow \frac{n_p \times t_m}{n_p \times r \times t_c + n_p \times (1 - r) \times t_m}$
18 // Percentage of speedup
19 $\alpha \leftarrow \frac{pos\_speedup}{P}$ $\beta \leftarrow \frac{neg\_speedup}{P}$
20 // Calculate $N_1$ $N_2$
21 $N_1 \leftarrow f_0(s_{l2},\ s_{l3},\ num\_threads,\ emb\_dim)$
22 $N_{20} \leftarrow \frac{M \times N_1}{I}$
23 $N_{21} \leftarrow \frac{N_1}{\beta \times P}$
24 if $N_{20} < N_{21}$ then
25     $\widehat{N_2} \leftarrow N_{20}$
26 else
27     $\widehat{N_2} \leftarrow N_{21}$
28 end
29 $\widehat{N_1} \leftarrow N_1$
```

that $num\_threads \times N_1$ embeddings can be held in the L2 cache (Line 21). (7) We can either choose the negative sampling space $I = M \times \frac{N_1}{N_2}$ or the negative speedup $neg\_speedup \approx \frac{N_2}{N_1}$ to calculate N_2 (Lines 22-23). (8) We select a smaller N_2 to ensure high accuracy since smaller N_2 larger negative sampling space (Lines 24-28).

4.3 Parallelization of Similarity Computation

Modern CPUs are usually multi-core architectures and support the multi-threading paradigm to further exploit instruction-level parallelism. A multi-core processor typically uses a single thread in a single physical core. In order to utilize hardware parallelism (e.g., multiple cores in CPUs, CUDA threads in GPUs) and high-performance libraries (e.g., BLAS, LAPACK), PyTorch abstracts input data into tensors (i.e., multi-dimensional matrix) and calculations into tensor operations. PyTorch-based SimpleX follows the same design philosophy. As discussed in §3.2, SimpleX first concatenates embeddings, then reshapes them, and adopts matrix-matrix multiplication to calculate the similarity.

However, directly adopting such a parallel computing design in CF training introduces two severe performance problems: (1) concatenating sampled embeddings into matrices and reshaping introduce expensive memory copies; and (2) normalization of embedding tensor needs writing of the generated intermediate tensor R back to the memory, which causes additional memory access time. To conquer the above limitations and make full use of the multi-core architecture and the multi-threading paradigm in CF training, we propose a new parallel method that different threads directly

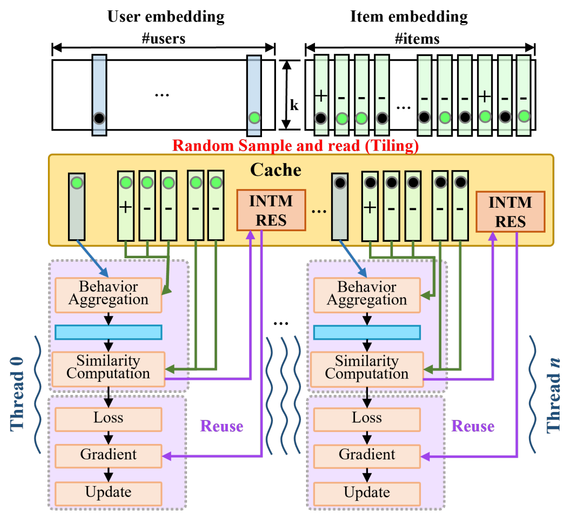

Figure 5: Overview of our training workload partition strategy. Different colored circles represent the embeddings sampled for different threads. "+" and "-" denote positive and negative embeddings, respectively.

perform dot products after reading sampled user/item embeddings without concatenation and reshaping.

Figure 5 depicts our proposed parallelization of similarity computation strategy. Specifically, for each iteration, each thread first fetches one user embedding S_u, one positive embedding T_{pos}, and n negative embeddings T_{neg}. Then, each thread performs the dot product of user embedding and positive/negative embeddings $S_u \cdot T_{pos}$ or $S_u \cdot T_{neg}$. Meanwhile, to facilitate calculations of cosine similarities and reuse these embeddings, each thread also does the dot product of each embedding with itself, since $||S_u||_2 = \sqrt{S_u \cdot S_u}$, $||T_{pos}||_2 = \sqrt{T_{pos} \cdot T_{pos}}$ and $||T_{neg}||_2 = \sqrt{T_{neg} \cdot T_{neg}}$. Each thread finally generates gradients using the optimized similarity and gradient computation and then updates corresponding embeddings.

This strategy also facilitates updating embedding matrices in a sparse fasion. Theoretically, we only need to generate the gradients of involved embeddings and update them. Note that although PyTorch allows users to set the parameter "sparse" to True to enable sparse gradients so as to update embeddings sparsely, it leads to worse performance as demonstrated in §3.1. By comparison, in our proposed method, different threads independently and in parallel are responsible for gradient calculations of involved embeddings. Besides, different threads can write embeddings matrices independently. Therefore, different threads are able to update these embeddings instead of updating all embeddings.

4.4 Aggressive Data Reuse

With matrix factorization, the rating matrix X is approximated by the matrix product of two low-rank matrices $S \in \mathbb{R}^{|U| \times K}$ and $T \in \mathbb{R}^{|I| \times K}$. Each row S_u in S can be seen as a feature vector describing a user u and similarly each row T_i of T describes an item i. We need to use the feedback X and a suitable loss function to train S and T. The training procedure is (1) pick a user-item pair $(u,\ i)$ from X. (2) calculate the similarity $\hat{x}_{u,i}$ of the user-item pair, we can use dot product similarity or cosine similarity. We focus on cosine similarity since it delivers better training results as demonstrated in SimpleX. (3) generate loss and loss gradient using the suitable loss function. (4) do gradient backpropagation to obtain partial derivatives (gradients) of involved embeddings. (5) utilize the obtained gradients to update engaged embeddings.

$$\frac{\partial \hat{x}_{u,i}}{\partial S_u} = \frac{T_i \cdot \sqrt{\sum S_u^2}\sqrt{\sum T_i^2} - \frac{1}{2}\left(\sqrt{\sum S_u^2}\right)^{-\frac{1}{2}} \cdot 2S_u \cdot \sqrt{\sum T_i^2}\sum S_u T_i}{\left(\sqrt{\sum S_u^2}\sqrt{\sum T_i^2}\right)^2} = \frac{T_i \cdot \sum S_u^2 - \sum S_u T_i \cdot S_u}{\sum S_u^2 \sqrt{\sum S_u^2}\sqrt{\sum T_i^2}} \quad (4)$$

$$\frac{\partial \hat{x}_{u,i}}{\partial T_i} = -\frac{S_u \cdot \sum T_i^2 - \sum S_u T_i \cdot T_i}{\sum T_i^2 \sqrt{\sum T_i^2}\sqrt{\sum S_u^2}} \quad (5)$$

The partial derivative of $\hat{x}_{u,i}$ with respect to the variable S_u is defined in Equation (4). $\frac{\partial \hat{x}_{u,i}}{\partial S_u}$ mainly consists of $\sum S_u^2$ the sum of squares of S_u, $\sum T_i^2$ the sum of squares of T_i, and $\sum S_u T_i$ the dot product of S_u and T_i.

We also need to calculate $\sum S_u^2$, $\sum T_i^2$, and $\sum S_u$ when calculating the cosine similarity $\hat{x}_{u,i}$ of the user-item pair in the forward phase. Thus, to avoid redundant calculation of the values of $\sum S_u^2$, $\sum T_i^2$, and $\sum S_u T_i$, we will cache the values of these variables in the forward phase to achieve data reuse.

Similarly, the partial derivative of $\hat{x}_{u,i}$ with respect to the variable T_i is defined in Equation (5). $\frac{\partial \hat{x}_{u,i}}{\partial T_i}$ is also related to $\sum S_u^2$, $\sum T_i^2$, and $\sum S_u T_i$. Thus, we can reuse $\sum T_i^2$, $\sum S_u^2$, and $\sum S_u T_i$ in the calculation of $\frac{\partial \hat{x}_{u,i}}{\partial T_i}$ in the backward computation.

4.5 Optimized Parallel Gradient Update

To parallelize similarity computation (in §4.3), different threads independently and in parallel are responsible for similarity computation, gradient calculations, and embedding updates of involved embeddings. This parallelization strategy brings another challenge when enabling the behavior aggregation layer.

As aforementioned, the traditional MF methods only need to read one user embedding, one positive embedding, and multiple negative embeddings in each iteration, and then feed these embeddings into the model to calculate gradients and then update the corresponding embeddings. But SimpleX uses an extra behavior aggregation layer to process interacted item sequence of each user to better model user behaviors. The essence of the behavior aggregation layer is a small fully connected layer, its input/output dimension is the same as the embedding dimension.

This layer aggregates the user's embedding and embeddings of the user's historical interaction items to generate a new embedding. Then, we feed the new embedding, a positive embedding, and multiple negative embeddings, into the model to update the corresponding embeddings. The effectiveness of the behavior aggregation layer has been proven in many previous works, such as YouTubeNet [12] and ACF [9]. It has three common aggregation choices, i.e., average pooling, self-attention, and user-attention

In HEAT, each thread performs the training procedure independently to avoid synchronization among threads, which will degrade the overall performance. Each thread reads the weight matrix of the behavior aggregation layer to perform forward and backward computations using different training data. This training mode is similar to data parallel distributed training. We can follow asynchronous distributed stochastic gradient descent (SGD) and specify one thread as the parameter server for the global weights of the behavior aggregation layer, other threads request weights replicas from the parameter server to process a mini batch to calculate gradients and send them back to the parameter server which updates the global weights accordingly. However, this method causes high overheads of memory and synchronization among threads due to multiple weights replicas and gradients exchange.

To solve this issue, inspired by a prior work (called Hogwild!) [36] that uses shared memory to hold global weights, which enables processes to access global weights without lock mechanism, we also let all threads access global weights without lock mechanism. However, Hogwild! cannot be directly applied to HEAT since Hogwild! targets to the sparse optimization problem but the optimization of the behavior aggregation layer is a dense optimization problem. In our HEAT design, we let all threads share one weight matrix, thus each thread just holds a pointer to the weight matrix, and then generates gradients to update the weight matrix directly. The conflict will happen when one thread tries to update the weight matrix while other threads try to read/write the weight matrix since there is only one copy of the data in the memory. To alleviate the conflict, we let each thread first accumulate the gradients locally, and update the global weight matrix every m iterations.

Listing 1 describes the simplified training workflow of the behavior aggregation layer. Specifically, (1) we enable multi-threading processing and let aggregator_weights be shared by all threads (Line 7). (2) We calculate weight gradients (weights_grad) locally and accumulate gradients to the accu_weights_grad (Lines 18-20). (3) We update the global weight matrix every mini_batch_size (Lines 22-27). According to our experimental results, we set mini_batch_size to 32 to avoid accuracy drop.

```
1  //Input: total iteration I, init_weights0,
2  //activation data act_data, outputs gradient outs_grad
3  //mini_batch_size
4  //Output: updated aggregator_weights
5  typedef Array<float, Dynamic, Dynamic> XMatrix
6  XMatrix aggregator_weights(emb_dim, init_weights0)
7  #pragma omp parallel shared(aggregator_weights) {
8   int i_counts = 0; // iteration counts
9   XMatrix weights_grad = Zero(emb_dim, emb_dim);
10  XMatrix accu_weights_grad = Zero(emb_dim, emb_dim);
11 #pragma omp for
12  for (int i=0; i<I; ++i) {
13   for (int k=0; k<emb_dim; ++k) {
14    weights_grad.row(k) = act_data(0, k) * outs_grad;
15   }
16   accu_weights_grad += weights_grad;
17   if (i_counts>0 && i_counts % mini_batch_size==0) {
18    weights_grad = accu_weights_grad / mini_batch_size;
19    aggregator_weights -= l_r * weights_grad;
20    accu_weights_grad = Zero(emb_dim, emb_dim);
21   } } }
```

Listing 1: Psuedocode of our behavior aggregator design.

5 PERFORMANCE EVALUATION

5.1 Experimental Setup

Datasets. We evaluate HEAT on five real-world datasets as they have been preprocessed for fairness and ease of comparison. Specifically, (1) we perform most of our experiments on three datasets,

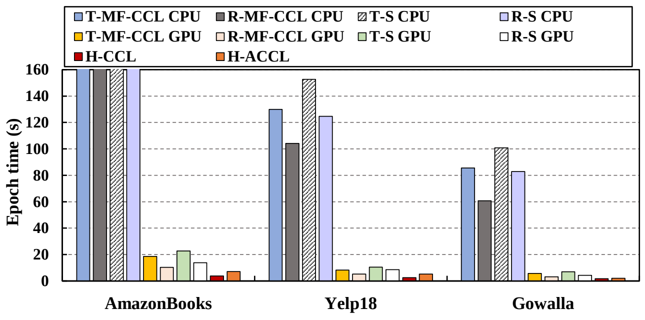

Figure 6: Comparison of epoch time between SimpleX and HEAT. Simplex without aggregation layer degenerates to MF-based model.

Amazon-Books, Yelp2018, and Gowalla, which are commonly used in recent CF works [10, 20]. Amazon-Books, Yelp2018, Gowalla have 52643, 31668, 29858 users, and 91599, 38048, 40981 items, respectively. (2) To demonstrate that HEAT is affordable, we further evaluate on two larger datasets, Goodreads Book Reviews (Goodreads) and Google Local Reviews (2018).

Platforms. We perform our experiments on three types of platforms: (1) a regular memory (RM) node from the Bridges-2 supercomputer [34] equipped with x86-architecture processors. Each RM node has two 64-core AMD EPYC 7742 CPUs with 32 MB L2 cache and 256 MB L3 cache; (2) a compute node from the Ookami [5] cluster equipped with Fujitsu ARM A64FX processors. Each A64FX processor features 48 cores, 512-bit wide SIMD, 32 MB L2 cache, and 32 GB HBM2 memory with 1024 GB/s bandwidth; and (3) a GPU node from the Bridges-2 supercomputer equipped with one NVIDIA Tesla 32 GB V100 GPU to perform GPU experiments.

Baselines. SimpleX mainly consists of MF, behavior aggregation layer, and cosine contrastive loss (CCL). Note that Simplex without aggregation layer degenerates to an MF-based model. We compare HEAT with five baselines: PyTorch-implemented MF with CCL (T-MF-CCL), TorchRec-implemented MF with CCL (R-MF-CCL), PyTorch-implemented SimpleX (T-S), TorchRec-implemented SimpleX (R-S), and CuMF_SGD.

Implementation details. We implement HEAT using C++. Specifically, we implement computation kernels using Eigen [15] for vector-dot product, and vector-matrix product. Eigen is a C++ template library for linear algebra. We use OpenMP to support our shared-memory multi-threading computation. We use Intel oneAPI C++ compiler [22] to compile C++ source code. We also use the Intel MKL library for BLAS operations and LAPACK operations. We use ARM C/C++ compiler [1] which provides armclang and armclang++. We link HEATto ARM performance library (ARMPL) to enable BLAS or LAPACK as Eigen's backend for dense matrix products. ARMPL [2] provides optimized standard core math libraries such as BLAS, LAPACK, FFT, and sparse routines with OpenMP.

5.2 Training Time

For training epoch time, we first compare HEAT with T-MF-CCL, R-MF-CCL, T-S, R-S. We run HEAT on the CPU and run T-MF-CCL, R-MF-CCL, T-S, and R-S on both the CPU and GPU. For this comparison, we use the embedding dimension of 128, 64 negative samples, and 100 historical items for fairness and ease of comparison. Moreover, we also compare HEAT with CuMF_SGD (i.e., the state-of-the-art GPU-based MF solution with high performance)

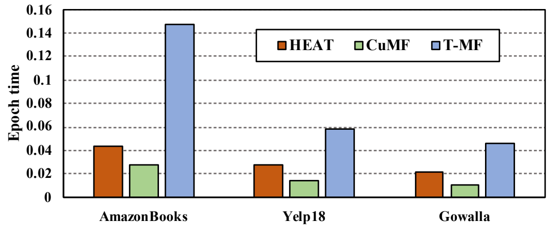

Figure 7: Comparison of epoch time among CuMF_SGD (GPU), TorchRec (GPU), and HEAT (CPU).

and TorchRec-based MF (T-MF). For this comparison, we use the embedding dimension of 128, one negative sample, dot-product similarity, and mean square error loss because CuMF_SGD only supports these settings.

Figure 6 shows the training epoch time comparison on the CPU and GPU among T-MF-CCL, R-MF-CCL, T-S, R-S, HEAT with CCL (H-CCL), HEAT with behavior aggregation layer and CCL (H-ACCL). Compared with the CPU baselines, H-CCL achieves 65.3×, 50.9×, and 47.8× speedup over T-MF-CCL on AmazonBooks, Yelp18, and Gowalla, respectively. H-CCL also achieves 41.1× on average over R-MF-CCL. H-ACCL achieves 43.7×, 28.8×, and 47.3× speedup over T-S on AmazonBooks, Yelp18, and Gowalla, respectively. H-ACCL also achieves 33.2× on average over R-S. Compared with the GPU baselines, HEAT with CCL achieves 4.8×, 3.2×, and 3.2× speedup over T-MF-CCL on AmazonBooks, Yelp18, and Gowalla, respectively. H-CCL also achieves 2.2× on average over R-MF-CCL. H-ACCL provides 3.2×, 2.0×, and 3.3× speedup over T-S on AmazonBooks, Yelp18, and Gowalla, respectively. H-ACCL achieves 1.9× speedup on average over R-S. We get such significant speedups because (1) we let each thread run independently and avoid synchronization between threads, (2) we aggressively reuse data in forward and backward computation to improve the performance, and (3) we only update the involved embeddings in each thread.

Figure 7 shows a comparison of training epoch time among CuMF_SGD on the GPU, TorchRec-based MF on the GPU, and HEAT on the CPU. The performance of HEAT and CuMF is comparable. However, CuMF_SGD implements the most basic CUDA-based (stochastic gradient descent) SGD solution for MF problems. CuMF_SGD only supports basic mean squared error loss function, 1 negative sample, and sets the embedding dimension to a fixed value of 128 for performance. HEAT achieves 2.1× speedup on average over TorchRec-based MF.

In addition, we break down the epoch time into different phases. Figure 8 shows that in HEAT-CCL the time of reading embeddings takes 39.3%, which proves the necessity of our tiling strategy. The

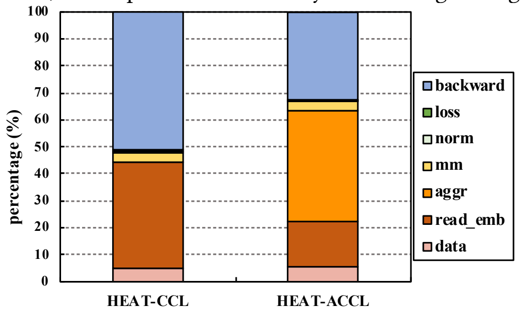

Figure 8: Performance breakdown of HEAT on CPU. Note that norm, mm, and aggr are short for normalization, matrix multiplication, and aggregation.

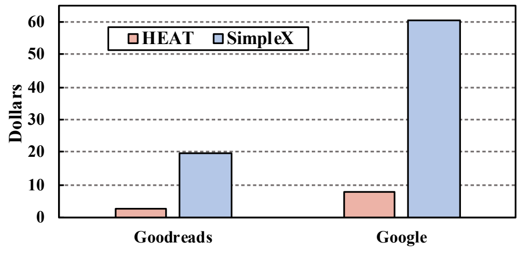

Figure 9: Comparison of total training cost ($) for 100 epochs.

normalization only takes up 0.38%, which shows our similarity computation is very efficient. Moreover, in HEAT-ACCL, the percentage of reading embeddings and aggregation reaches 16.5% and 41.4%, respectively, which indicates further optimization on the aggregation computation in future work.

5.3 Training Cost

Next, to demonstrate that our training system is highly affordable, we compare the training cost of our HEAT on the CPU and SimpleX on the GPU on two large datasets, i.e., Goodreads Book Reviews (Goodreads) and Google Local Reviews (2018) (Google). We use AWS p3.2xlarge instance as the GPU platform, which is equipped with one 16 GB V100 GPU. The price of p3.2xlarge is $3.06 per hour. We need two p3.2xlarge to fit these two large datasets since each GPU has only 16 GB memory. We use AWS c5a.16xlarge as the CPU platform, which is equipped with one AMD EPYC 7R32 and 128 GB memory. The price of c5a.16xlarge is $2.46 per hour [4]. Figure 9 shows the comparison of the total training cost of the two methods for 100 epochs. Compared with SimpleX on the GPU, HEAT can reduce the cost by 7.9×.

5.4 Training Accuracy

After that, we report the training results on different datasets using the same evaluation metrics (e.g., Recall@20 and NDCG@20) and parameter configuration as SimpleX in Table 5, to demonstrate that our proposed multi-threading training system does not affect the training accuracy. Both SimpleX and HEAT's negative sampler obey the uniform distribution. We use the metric "recall", which is a widely used indicator to assess the proportion of positive samples successfully predicted by the CF model to all actually positive samples. It is calculated as Recall = $\frac{TP}{TP+FN}$, where TP and FN stand for true positive and false negative in the confusion matrix, respectively. "NDCG" is short for normalized discounted cumulative gain. The difference between the Recall@20 of HEAT and the Recall@20 of SimpleX is within ±0.0003. Therefore, we can conclude that the proposed multi-threading training framework has negligible impact on the training accuracy.

5.5 Impacts of Tiling Sizes and Refresh Intervals on Performance and Accuracy

Furthermore, we show the effectiveness of our proposed random tiling strategy and the proposed tuning algorithm for tiling size and refresh interval. We perform experiments on AmazonBooks dataset and set the embedding dimension to 128, the number of negatives to 64, and the number of historical items to 100.

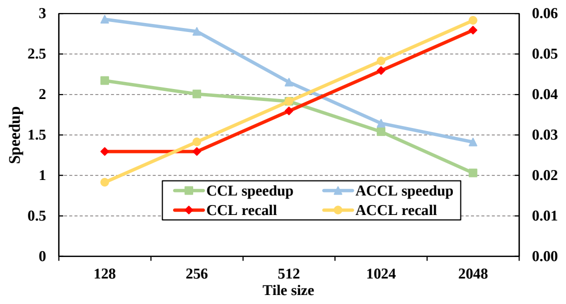

Figure 10: Speedup & recall with different tiling sizes on AmazonBooks.

First, we show how the speedup and recall change with different tiling sizes when the refresh interval is fixed. Figure 10 depicts the speedup over HEAT with a random negative sampler gradually decreases with increasing tiling size. In particular, the speedup exceeds 2× when the tiling size is less than 128 because embeddings can be fully cached in the l2 cache. Meanwhile, the recall will gradually increase as the tiling size increases because the sampling space of the negative sampler increases.

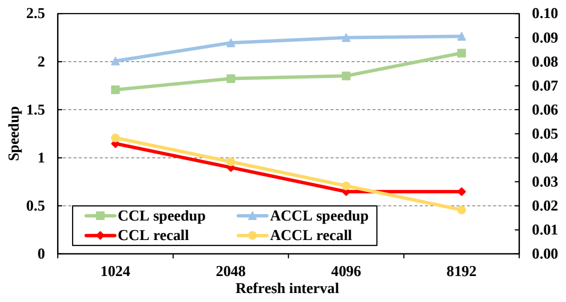

Figure 11: Speedup & recall with different refresh intervals on AmazonBooks.

Second, we show how the speedup and recall change with different refresh intervals when the tiling size is fixed. Figure 11 shows the speedup over HEAT with a random negative sampler gradually increases with increasing refresh interval. The reason is that increasing refresh interval raises the probability of data appearing in the cache, thereby reducing the time to read data. Simultaneously, the recall will gradually decrease as the refresh interval increases because the sampling space of the negative sampler decreases. From these two experiments and the derivation of §4.2, we conclude that we need to adjust tiling size and refresh interval simultaneously to get the optimal accuracy and performance.

In addition, Table 6 shows the tiling size and refresh interval corresponding to the optimal training results and speedup obtained by our Algorithm 1. HEAT with the random tiling sampler delivers 1.9× speedup on average over HEAT with the random sampler, while the recall drop is also negligible (i.e., within 0.001).

5.6 Behavior Aggregation Evaluation

To prove the proposed local gradient accumulation benefits the performance of the behavior aggregation layer, we compare the performance of HEAT with and without local gradient accumulation, as shown in Table 7. HEAT with our local gradient accumulation provides a 2.3× speedup on average due to fewer write conflicts. Moreover, its recall drop is within 0.0005.

Table 5: Comparison of training results under different frameworks and datasets.

Method	AmazonBooks		Yelp18		Gowalla	
	Recall@20	NDCG@20	Recall@20	NDCG@20	Recall@20	NDCG@20
MF-CCL	0.0559	0.0447	0.0698	0.0572	0.1837	0.1493
SimpleX	0.0583	0.0468	0.0701	0.0575	0.1872	0.1557
HEAT-CCL	0.0561	0.0445	0.0698	0.0573	0.1836	0.1495
HEAT-ACCL	0.0585	0.047	0.0702	0.0577	0.1869	0.1554

Table 6: Tiling size and refresh interval for optimal training accuracy and speedup. "R" and "T" represent random tiling sampler and random sampler, respectively.

Method	AmazonBooks				Yelp18				Gowalla			
	Recall@20	Tile	Interval	Speedup	Recall@20	Tile	Interval	Speedup	Recall@20	Tile	Interval	Speedup
RCCL	0.0511	N/A	N/A	N/A	0.0626	N/A	N/A	N/A	0.1732	N/A	N/A	N/A
RACCL	0.0535	N/A	N/A	N/A	0.0659	N/A	N/A	N/A	0.1769	N/A	N/A	N/A
TCCL	0.0509	1024	4096	1.6	0.0621	1024	2048	1.8	0.1728	512	2048	2.0
TACCL	0.0532	1024	2048	2.1	0.0652	1024	4096	2.2	0.1763	1024	4096	1.9

Table 7: Epoch time and recall w/ and w/o local gradient accumulation.

Metrics	AmazonBooks		Yelp18		Gowalla	
	W	W/O	W	W/O	W	W/O
Epoch	7.17	16.92	5.31	9.45	2.13	4.96
Recall	0.0535	0.0531	0.0652	0.0650	0.1763	0.1759

5.7 Scalability Evaluation

To demonstrate the scalability of HEAT, we choose the AmazonBooks dataset and set the embedding dimension to 128 and the number of negatives to 64. We increase the number of threads/-cores from 1 to 64 (commonly used in other CF works [42, 47]). Figure 12 illustrates that the epoch time (in log-scale) decreases linearly as the number of threads increases (with the parallel efficiency of 83.7%). HEAT can achieve this high scalability because (1) different threads are responsible independently for the gradient calculation and embedding update in parallel, and (2) there is no need for communication and synchronization across different threads.

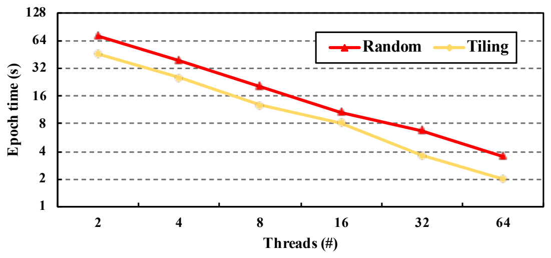

Figure 12: Scalability of HEAT with original random sampler (random) and our random tiling sampler (tiling).

5.8 Discussion of Different CPU Architectures

To explore suitable CPU architectures for CF applications, which feature highly irregular memory access and low computation intensity, we also evaluate the performance of HEAT on an ARM-architecture processor, i.e., Fujistu A64FX, since A64FX provides 1024 GB/s bandwidth and 48 compute cores with 512-bit wide SIMD.

We first compare the overall training performance between SimpleX and HEAT on the ARM CPU. SimpleX is implemented using ARMPL-optimized PyTorch, while HEAT is compiled by armclang++ and linked to ARMPL. Figure 13 shows the training epoch time comparison between SimpleX and HEAT. on AmazonBooks, Yelp18,

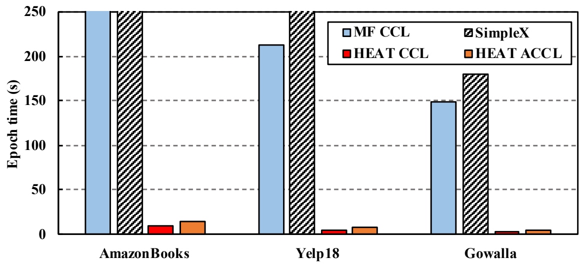

Figure 13: Comparison of training epoch time on ARM CPUs.

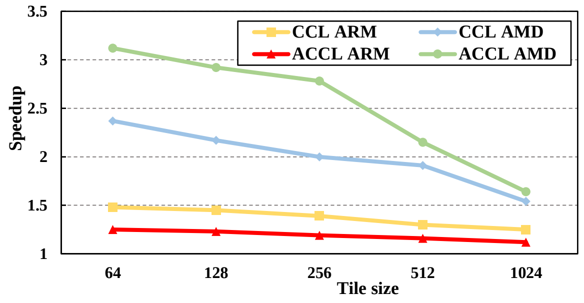

Figure 14: Comparison of tiling speedup on ARM and AMD CPUs.

and Gowalla datasets, HEAT with CCL achieves 50.4×, 42.6×, and 44.1× speedup over SimpleX without aggregation layer (degenerated to classic matrix factorization), respectively; HEAT with ACCL provides 41.7×, 37.9×, and 39.9× speedup over SimpleX with aggregation layer, respectively.

We then show a comparison of tiling speedup on the ARM CPU and on the AMD CPU in Figure 14. Specifically, our tiling optimization only provides up to 1.5 × speedup on the ARM CPU, whereas it achieves up to 3.1 × speedup on the AMD CPU. This is because of three reasons: (1) ARM only has two levels of caches (with a L2 cache of 32 MB), while AMD has three levels of caches (with a much larger L3 cache of 256 MB); a smaller cache leads to a higher cache miss rate. (2) Negative sampling following the uniform distribution leads to irregular memory access, which cannot give fully leverage the high memory bandwidth of HBM2. (3) The ARM CPU has less physical cores (i.e., 48 cores) than the AMD CPU, and the computation time takes more than 70% of the total time, resulting in limited optimization space for tiling.

6 RELATED WORK

BPR [37] proposes a generic optimization criterion for personalized ranking via maximizing posterior estimator derived from a Bayesian analysis of the problem [37]. Its core idea behind is to find suitable Θ to represent parameters of an arbitrary model via maximizing posterior estimator $p(\Theta| > u) \propto p(> u|\Theta)p(\Theta)$, where $> u$ represents a user's preference. BPR concentrates on the most common scenario with implicit feedback (e.g. clicks, purchases). However, BPR uses only one negative sample, which causes inferior results for many CF models [31].

SimpleX [31] investigates the impacts of the loss function, and negative sampling in CF. It demonstrates the importance of selecting an appropriate loss function and a proper number of negative samples. Inspired by contrastive loss [17] in computer vision, SimpleX proposes a cosine contrastive loss (CCL) tailored for CF. However, SimpleX implemented its algorithm using PyTorch and did not consider the computational efficiency on either CPU or GPU.

CuMF_SGD [48] utilizes GPU's massive threads to update embeddings in parallel. CuMF_SGD implemented the basic stochastic gradient descent (SGD) solution using CUDA for MF problems. However, it cannot create user-specific item ranking using the concept of positive/negative items and only supports dot-product similarity, basic mean squared error loss function and requires a fixed embedding dimension (i.e., 128) to achieve high performance.

MSGD [30] is a MF approach for large-scale CF based recommender systems on GPUs. To parallelize SGD, MSGD removes dependencies between user and item pairs. It also splits the MF optimization objective into many separate sub-objectives. However, the optimizations of MSGD cannot be applied to multi-core CPUs because MSGD specially optimizes its parallelization approaches for coalesced memory access in GPUs. Furthermore, similar to CuMF_SGD, it does not support sampling multiple negative items, which is crucial to the final training results. Due to the lack of source code for MSGD, we compare the performance of HEAT with SimpleX and CuMF_SGD.

7 CONCLUSION AND FUTURE WORK

In this work, we propose an efficient and affordable collaborative filtering-based recommendation training system that incorporates features of the multi-level cache and multi-threading paradigms of modern CPUs. It has a series of optimizations to address the performance issues of irregular memory accesses, unnecessary memory copies, and redundant computations. Evaluation on five widely used datasets with AMD and ARM CPUs shows that HEAT achieves up to 65.3× and 4.8× speedups over existing CPU and GPU solutions, respectively, with 7.9× cost reduction.

In the future, we plan to first extend our work to support distributed training with rating matrix partitioning and efficient communication. Then, we will apply our random tiling strategy to more recommendation models such as graph neural network based CF.

ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation OAC-2303820 and OAC-2312673. The material was also supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contracts DE-AC02-06CH11357.

REFERENCES

[1] arm. https://developer.arm.com/Tools%20and%20Software/Arm%20Compiler%20for%20Linux, 2023. Online.

[2] arm. https://developer.arm.com/Tools%20and%20Software/Arm%20Performance%20Libraries, 2023. Online.

[3] AWS. https://instances.vantage.sh/aws/ec2/p3.2xlarge, 2023. Online.

[4] AWS. https://instances.vantage.sh/aws/ec2/c5a.16xlarge, 2023. Online.

[5] Md Abdullah Shahneous Bari, Barbara Chapman, Anthony Curtis, Robert J Harrison, Eva Siegmann, Nikolay A Simakov, and Matthew D Jones. A64fx performance: experience on ookami. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 711–718. IEEE, 2021.

[6] Immanuel Bayer, Xiangnan He, Bhargav Kanagal, and Steffen Rendle. A generic coordinate descent framework for learning from implicit feedback. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1341–1350, 2017.

[7] Daniel Billsus, Michael J Pazzani, et al. Learning collaborative information filters. In *Icml*, volume 98, pages 46–54, 1998.

[8] Iván Cantador, Miriam Fernández, David Vallet, Pablo Castells, Jérôme Picault, and Myriam Ribiere. A multi-purpose ontology-based approach for personalised content filtering and retrieval. In *Advances in Semantic Media Adaptation and Personalization*, pages 25–51. Springer, 2008.

[9] Jingyuan Chen, Hanwang Zhang, Xiangnan He, Liqiang Nie, Wei Liu, and Tat-Seng Chua. Attentive collaborative filtering: Multimedia recommendation with item-and component-level attention. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 335–344, 2017.

[10] Lei Chen, Le Wu, Richang Hong, Kun Zhang, and Meng Wang. Revisiting graph based collaborative filtering: A linear residual graph convolutional network approach. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 27–34, 2020.

[11] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

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

[13] Jingtao Ding, Guanghui Yu, Xiangnan He, Yuhan Quan, Yong Li, Tat-Seng Chua, Depeng Jin, and Jiajie Yu. Improving implicit recommender systems with view data. In *IJCAI*, pages 3343–3349, 2018.

[14] Xin Dong, Lei Yu, Zhonghuo Wu, Yuxia Sun, Lingfeng Yuan, and Fangxi Zhang. A hybrid collaborative filtering model with deep structure for recommender systems. In *Proceedings of the AAAI Conference on artificial intelligence*, volume 31, 2017.

[15] eigen. https://eigen.tuxfamily.org/index.php?title=Main_Page, 2023. Online.

[16] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[17] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

[18] Tanjim Ul Haque, Nudrat Nawal Saber, and Faisal Muhammad Shah. Sentiment analysis on large scale amazon product reviews. In *2018 IEEE international conference on innovative research and development (ICIRD)*, pages 1–6. IEEE, 2018.

[19] Ruining He and Julian McAuley. 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*, pages 507–517, 2016.

[20] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 639–648, 2020.

[21] Il Im and Alexander Hars. Does a one-size recommendation system fit all? the effectiveness of collaborative filtering based recommendation systems across different domains and search modes. *ACM Transactions on Information Systems (TOIS)*, 26(1):4–es, 2007.

[22] intel. https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.htm, 2023. Online.

[23] Folasade Olubusola Isinkaye, Yetunde O Folajimi, and Bolande Adefowoke Ojokoh. Recommendation systems: Principles, methods and evaluation. *Egyptian informatics journal*, 16(3):261–273, 2015.

[24] Dmytro Ivchenko, Dennis Van Der Staay, Colin Taylor, Xing Liu, Will Feng, Rahul Kindi, Anirudh Sudarshan, and Shahin Sefati. Torchrec: a pytorch domain library for recommendation systems. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 482–483, 2022.

[25] Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*, 2021.

[26] Brendan Kitts, David Freed, and Martin Vrieze. Cross-sell: a fast promotion-tunable customer-item recommendation method based on conditionally independent probabilities. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 437–446, 2000.

[27] Hyeyoung Ko, Suyeon Lee, Yoonseo Park, and Anna Choi. A survey of recommendation systems: recommendation models, techniques, and application fields. *Electronics*, 11(1):141, 2022.

[28] Yehuda Koren. Collaborative filtering with temporal dynamics. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 447–456, 2009.

[29] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[30] Hao Li, Kenli Li, Jiyao An, and Keqin Li. Msgd: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 29(7):1530–1544, 2017.

[31] Kelong Mao, Jieming Zhu, Jinpeng Wang, Quanyu Dai, Zhenhua Dong, Xi Xiao, and Xiuqiang He. Simplex: A simple and strong baseline for collaborative filtering. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 1243–1252, 2021.

[32] Chenguang Pan and Wenxin Li. Research paper recommendation with topic analysis. In *2010 International Conference On Computer Design and Applications*, volume 4, pages V4–264. IEEE, 2010.

[33] Rajiv Pasricha and Julian McAuley. Translation-based factorization machines for sequential recommendation. In *Proceedings of the 12th ACM Conference on Recommender Systems*, pages 63–71, 2018.

[34] psc. https://www.psc.edu/resources/bridges-2/, 2023. Online.

[35] PyTorch. https://pytorch.org/docs/stable/profiler.html, 2023. Online.

[36] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24, 2011.

[37] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618*, 2012.

[38] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, 1994.

[39] James Salter and Nick Antonopoulos. Cinemascreen recommender agent: combining collaborative and content-based filtering. *IEEE Intelligent Systems*, 21(1):35–41, 2006.

[40] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295, 2001.

[41] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. Collaborative filtering recommender systems. In *The adaptive web*, pages 291–324. Springer, 2007.

[42] Jinbo Song, Chao Chang, Fei Sun, Xinbo Song, and Peng Jiang. Ngat4rec: Neighbor-aware graph attention network for recommendation. *arXiv preprint arXiv:2010.12256*, 2020.

[43] Dave Steinkraus, Ian Buck, and PY Simard. Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 1115–1120. IEEE, 2005.

[44] Wei Tan, Liangliang Cao, and Liana Fong. Faster and cheaper: Parallelizing large-scale matrix factorization on gpus. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 219–230, 2016.

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

[46] Feng Wang and Huaping Liu. Understanding the behaviour of contrastive loss. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2495–2504, 2021.

[47] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. Neural graph collaborative filtering. In *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval*, pages 165–174, 2019.

[48] Xiaolong Xie, Wei Tan, Liana L Fong, and Yun Liang. Cumf_sgd: Fast and scalable matrix factorization. *arXiv preprint arXiv:1610.05838*, 2016.