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

Chengming Zhang	*Indiana University Bloomington*
Shaden Smith	*Microsoft*
Baixi Sun	*Indiana University Bloomington*
Jiannan Tian	*Indiana University Bloomington*
Jonathan Soifer	*Microsoft*
Xiaodong Yu	*Argonne National Laboratory*
Shuaiwen Leon Song	*Microsoft*
Yuxiong He	*Microsoft*
Dingwen Tao	*Indiana University Bloomington*

ICS 2023: International Conference on Supercomputing (In Conjunction with the Federated Computing Research Conference)
Orlando, Florida, United States, June 22, 2023

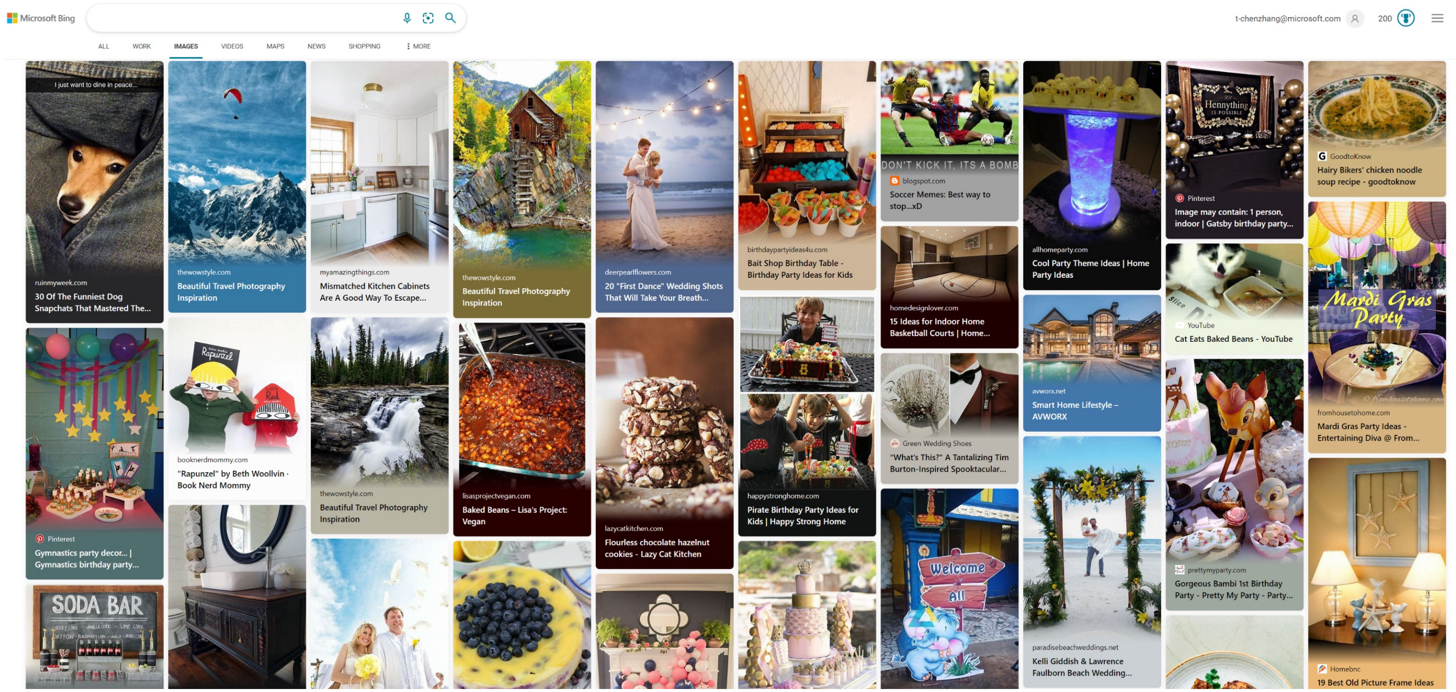
Introduction

INDIANA UNIVERSITY
BLOOMINGTON

➢ Recommendation systems overview

- Candidate generation. Generate potential recommendations (embeddings) for a user. (focus on)
- Scoring. Scores and ranks the candidates.
- Users' application.

➢ **Example: Bing image feed is personalized with our collaborative filtering as a recall path.**



BACKGROUND

Background: Basics of Recommender Systems

INDIANA UNIVERSITY
BLOOMINGTON

➢ Input data

- Explicit feedback: likes and ratings.
- Implicit feedback: users' interactions, e.g. click data, purchases, and implicit visit information.

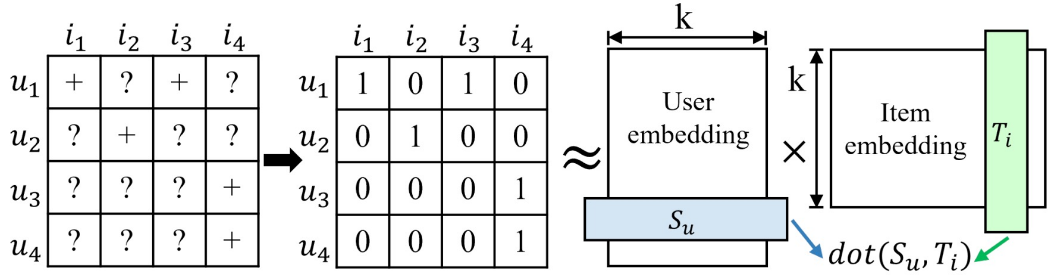
➢ Filtering techniques

- Content-based: notorious for its inability to recommend dissimilar items
- Collaborative filtering (CF): provide diverse recommendations.

Background: Basics of Recommender Systems

➢ CF techniques

- User-user CF, item-item CF, dimensionality reduction, and probabilistic methods.
- Dimensionality reduction uses matrix factorization (MF) reduce rating space to K.
- MF reduces computational complexity and memory requirements.



Implicit feedback $X \subseteq U \times I$. A user embedding matrix $S \in \mathcal{R}^{|U| \times K}$ and an item embedding matrix $T \in \mathcal{R}^{|I| \times K}$.

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

➢ Software Frameworks

- PyTorch provides a lookup table (torch.nn.Embedding) to store embeddings.
- TorchRec is a production-quality recommender systems package.

Background: MF-based Collaborative Filtering

INDIANA UNIVERSITY
BLOOMINGTON

Training purpose

- Maximize the similarity of a positive user-item pair while minimizing the similarity of a negative user-item pair.

SOTA CF method - SimpleX

- Adopt a novel loss function: cosine contrastive loss (CCL).
- A large negative sampling rate.
- Greatly outperforming other existing methods.

Loss	AmazonBooks		Yelp18		Gowalla	
	Recall@20	NDCG@20	Recall@20	NDCG@20	Recall@20	NDCG@20
BPR Loss	0.0338	0.0261	0.0549	0.0445	0.1616	0.1366
Pairwise Hinge Loss	0.0352	0.0267	0.0562	0.0453	0.1318	0.0996
Binary Cross-Entropy	0.0479	0.0371	0.0617	0.0503	0.1321	0.1159
Softmax Cross-Entropy	0.0478	0.0367	0.0639	0.0522	0.1545	0.1276
Mean Square Error	0.0337	0.0267	0.0624	0.0513	0.1528	0.1315
Cosine Contrastive Loss	**0.0559**	**0.0447**	**0.0698**	**0.0572**	**0.1837**	**0.1493**

Performance of MF under different loss functions.

PROFILING

Profiling and Analysis (1/4)

INDIANA UNIVERSITY
BLOOMINGTON

- **Characterize the performance of SimpleX on both CPU and GPU**

- **Embedding update in SimpleX**

- SimpleX randomly fetches a batch of embeddings to perform one training iteration.
- Logically, only need to update involved embeddings.
- Actual epoch time of training with sparse gradient is almost 3x higher than that of dense gradient.

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%

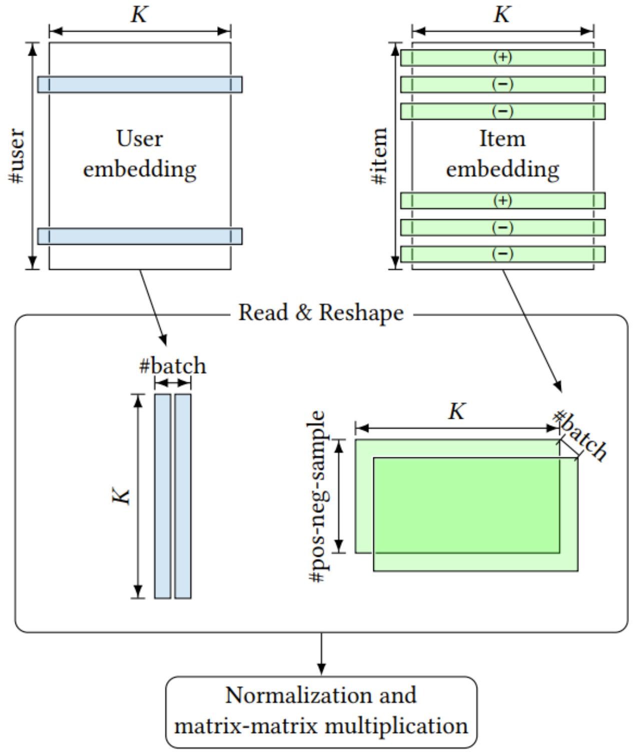
Profiling of embedding update in SimpleX. ET, FP, BP are short for epoch time, forward percentage, backward percentage, respectively.

Profiling and Analysis (2/4)

INDIANA UNIVERSITY
BLOOMINGTON

➢ Computation efficiency of SimpleX

- SimpleX needs to concatenate and then reshape embeddings to utilizes torch.bmm.
- Time of mem_cp and the time of bmm are comparable.
- Normalization takes more than 20% of the forward time.



← Overview of SimpleX. +/- denote positive/negative embedding.

Dataset	u_emb	i_emb	u_norm	i_norm	mem_cp	bmm	loss
AmazonBooks	9.6%	39.8%	5.9%	22.3%	5.0%	7.1%	9.7%
Yelp18	9.1%	35.3%	5.1%	28.3%	4.8%	7.2%	9.6%
Gowalla	8.3%	33.2%	5.6%	31.1%	4.8%	7.3%	9.1%

Breakdown of the forward phase of SimpleX.

Profiling and Analysis (3/4)

INDIANA UNIVERSITY
BLOOMINGTON

➢ Memory usage of SimpleX

- Sizes of user and item embedding matrices in MF-based CF are linearly scaled to the size of training dataset.
- Runs out of the GPU memory when the numbers of users and items are over 3 millions.

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

Memory usage of SimpleX. OoM is short for out of memory.

Profiling and Analysis (4/4)

INDIANA UNIVERSITY
BLOOMINGTON

- **Summarization of SimpleX limitations.**

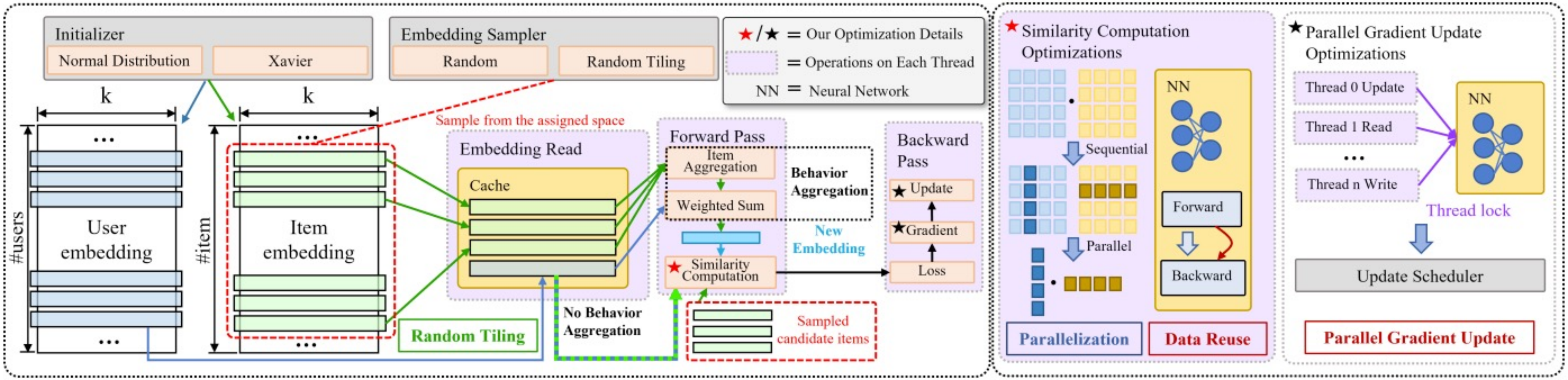
1) Irregular memory accesses: training on sparse user-item rating matrices and random sampling for multiple negative items.
2) Extra memory copies: similarity computation needs to concatenate sampled vectors into matrices.
3) Out of memory: limited GPU memory causes error.
4) Ignore potential data reuse: automatic differentiation engines in backward phase.

DESIGN

Overview of HEAT

➢ Overview of HEAT

- (1) Initializes user/item embedding matrices on CPU ⟹ **limitation (3)**
- (2) Chooses either the original random sampler or proposed random tiling sampler ⟹ **limitation (1)**
- (3) Behavior aggregation layer generates a new user embedding
- (4) Calculates similarities in parallel ⟹ **limitation (2)**
- (5) Calculates gradients through an optimized gradient computation kernel ⟹ **limitation (4)**

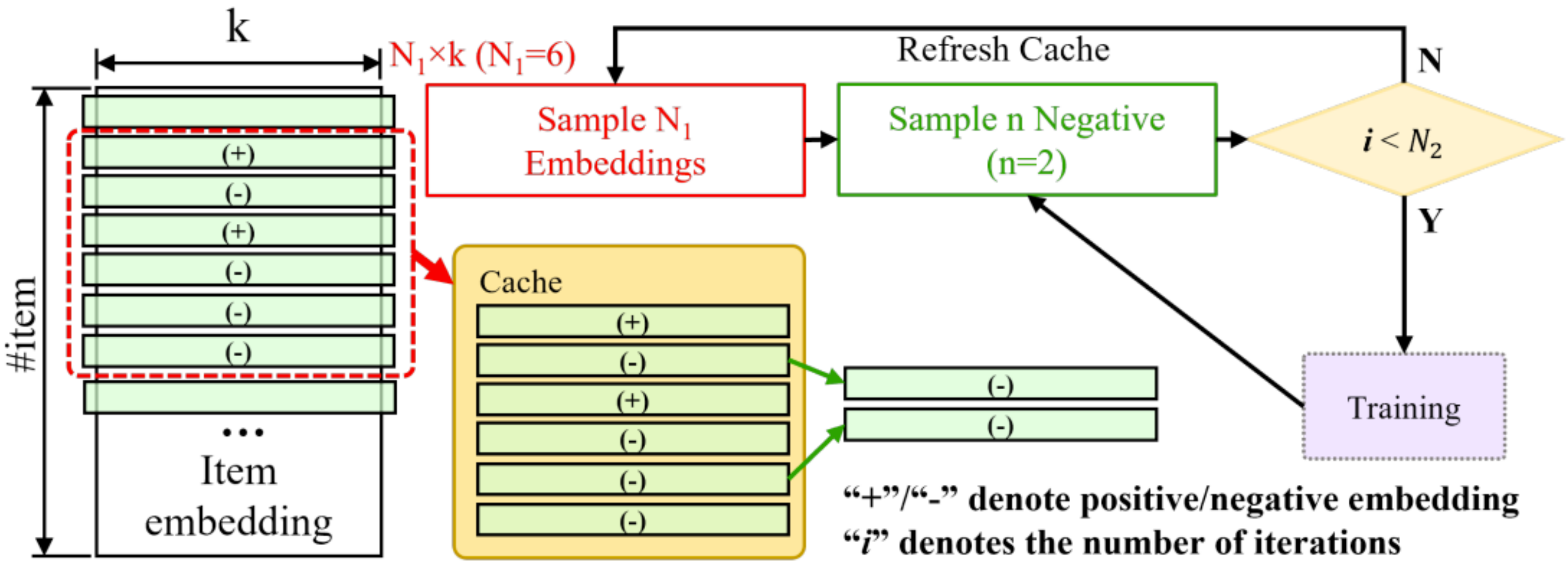


Overview of our proposed HEAT workflow/dataflow.

Design Methodology

➢ **Cache size oriented tiling**

- Each thread buffers randomly sampled N_1 embeddings.
- Each thread then randomly samples n negative embeddings from buffer to compute.
- After N_2 iterations, each thread randomly samples N_1 embeddings again to refresh the cache space.



Random tiling strategy in each thread.

Design Methodology

➢ Tiling size and refresh interval tuning

- Negative sampling space of random tiling is determined by $\frac{N_1}{N_2}$.
- Using random tiling, and the speedup can be approximated as $\frac{N_2}{N_1}$.
- First obtain N_1 according to L2 cache.
- Using negative sampling space or the negative speedup to calculate N_2.

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; α, β: percentage of positive, negative speedup

Outputs : $\widehat{N_1}$: optimized tile size; $\widehat{N_2}$: optimized refresh interval

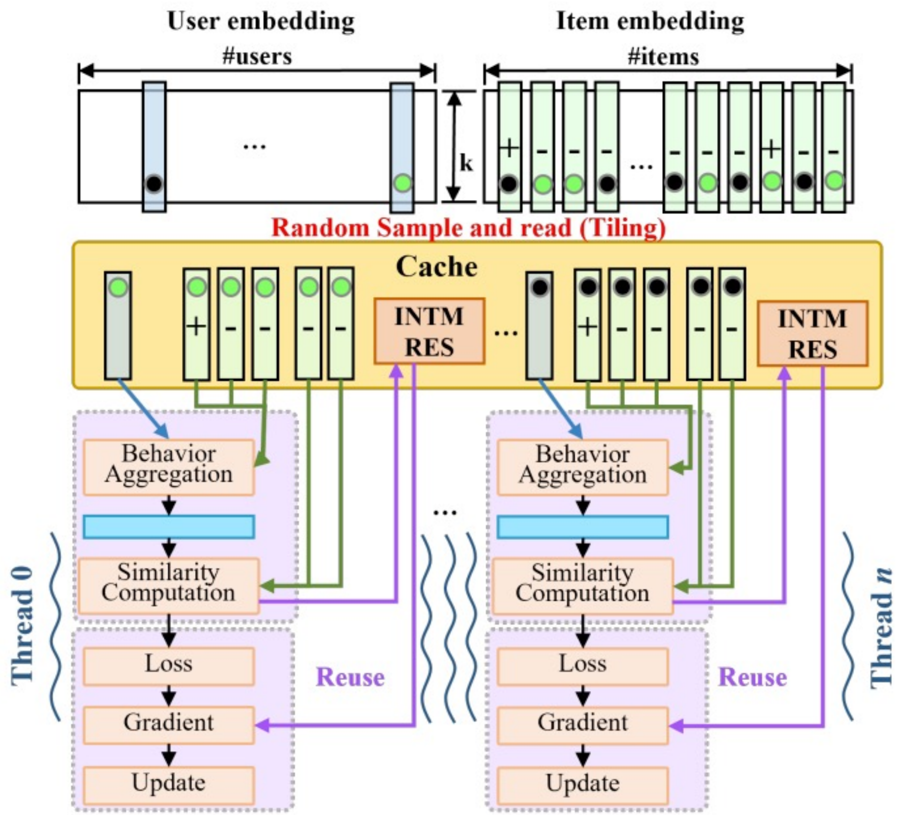
// Negative sampling space of tiling
$neg\_space \leftarrow \frac{M}{N_2} \times N_1 = M \times \frac{N_1}{N_2}$
// Time of reading negatives using random sampling
$neg\_time\_random \leftarrow M \times n_n \times t_m$
// Estimate latency of reading cache
$s_t \leftarrow N_1 \times sizeof(embedding\ row) \times num\_threads$
if $s_t < s_{l2}$ **then**
  $t_c \leftarrow t_{l2}$
else if $s_t \geq s_{l2}$ and $s_t < s_{l3}$ **then**
  $t_c \leftarrow t_{l3}$
else
  $t_c \leftarrow t_m$
end
// Time of reading negatives using tiling
$neg\_time\_tiling \leftarrow n_n \times \frac{M}{N_2} \times ((N_2 - N_1) \times t_c + N_1 \times t_m)$
$neg\_speedup \leftarrow \frac{neg\_time\_random}{neg\_time\_tiling} = \frac{t_m}{t_c + (t_m - 1) \times \frac{N_1}{N_2}} \approx \frac{N_2}{N_1}$
$pos\_speedup \leftarrow \frac{n_p \times t_m}{n_p \times r \times t_c + n_p \times (1-r) \times t_m}$
// Percentage of speedup
$\alpha \leftarrow \frac{pos\_speedup}{P}\ \beta \leftarrow \frac{neg\_speedup}{P}$
// Calculate N_1 N_2
$N_1 \leftarrow f_0(s_{l2},\ s_{l3},\ num\_threads,\ emb\_dim)$
$N_{20} \leftarrow \frac{M \times N_1}{I}$
$N_{21} \leftarrow \frac{N_1}{\beta \times P}$
if $N_{20} < N_{21}$ **then**
  $\widehat{N_2} \leftarrow N_{20}$
else
  $\widehat{N_2} \leftarrow N_{21}$
end
$\widehat{N_1} \leftarrow N_1$

Design Methodology

INDIANA UNIVERSITY
BLOOMINGTON

➢ **Parallelization of similarity computation**

- Each thread fetches one user embedding, one positive embedding, and n negative embeddings.
- Each thread then directly performs the dot product of user embedding and positive/negative embeddings.
- Updating embedding matrices in a sparse fashion independently and in parallel.



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

Design Methodology

➢ Aggressive data reuse

- $S \in R^{|U| \times K}$, $T \in R^{|U| \times K}$; S_u describing a user u, T_i describes an item i.
- The training procedure is (1) pick a user-item pair (u, i).
  (2) Calculate the similarity $\widehat{x_{u,i}}$ of the user-item pair.
  (3) Generate loss and gradient using the suitable loss function.
  (4) do gradient backpropagation to obtain partial derivatives (gradients) of involved embeddings.
- $\frac{\partial \widehat{x_{u,i}}}{\partial S_u}$ mainly consists of $\sum s_u^2$, $\sum T_i^2$, and $\sum S_u T_i$
- Cache the values of $\sum s_u^2$, $\sum T_i^2$, and $\sum S_u T_i$ in the forward when calculating the cosine similarity.

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

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

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

Design Methodology

➢ Optimized parallel gradient update

- Let aggregator_weights be shared by all threads.
- Calculate weight gradients locally and accumulate it.
- Update the global weight matrix every x steps.

```
//Input: total iteration I, init_weights0,
//activation data act_data, outputs gradient outs_grad
//mini_batch_size
//Output: updated aggregator_weights
typedef Array<float, Dynamic, Dynamic> XMatrix
XMatrix aggregator_weights(emb_dim, init_weights0)
#pragma omp parallel shared(aggregator_weights) {
 int i_counts = 0; // iteration counts
 XMatrix weights_grad = Zero(emb_dim, emb_dim);
 XMatrix accu_weights_grad = Zero(emb_dim, emb_dim);
#pragma omp for
 for (int i=0; i<I; ++i) {
  for (int k=0; k<emb_dim; ++k) {
   weights_grad.row(k) = act_data(0, k) * outs_grad;
  }
  accu_weights_grad += weights_grad;
  if (i_counts>0 && i_counts % mini_batch_size==0) {
   weights_grad = accu_weights_grad / mini_batch_size;
   aggregator_weights -= l_r * weights_grad;
   accu_weights_grad = Zero(emb_dim, emb_dim);
  } } }
```

EVALUATION

Evaluation

➢ Experimental setup

- Five real-world datasets:
    - Amazon-Books
    - Yelp2018
    - Gowalla
    - Goodreads
    - Google Local Reviews

- Platforms:
    - Bridges-2: 64-core, AMD EPYC 7742 CPU; NVIDIA Tesla 32 GB V100 GPU
    - Ookami: 48 cores, ARM A64FX.

- Baselines:
    - T-MF-CCL: PyTorch-implemented MF with CCL.
    - R-MF-CCL: TorchRec-implemented MF with CCL.
    - T-S: PyTorch-implemented SimpleX.
    - R-S: TorchRec-implemented SimpleX.
    - CuMF_SGD: SOTA GPU-based MF solution.

Evaluation

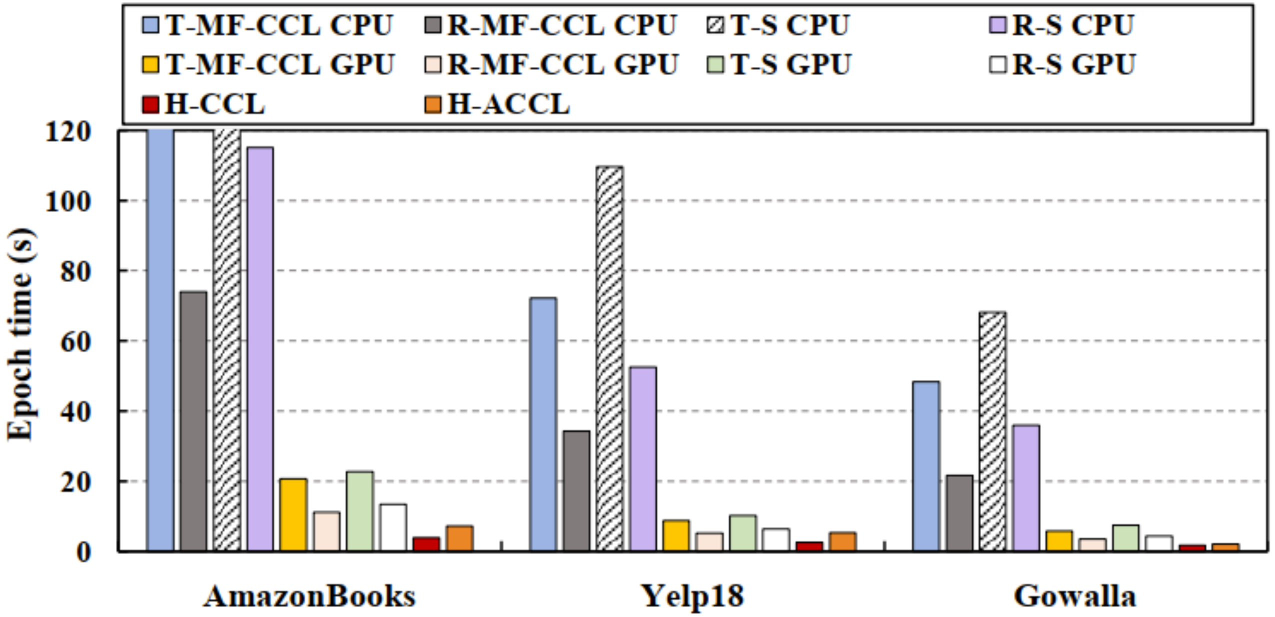
➢ Comparison of training time

➢ Compared with the CPU baselines

- H-CCL achieves 33.5x on average over T-MF-CCL.
- H-ACCL achieves 29.8x speedup on average over T-S.

➢ Compared with the GPU baselines

- H-CCL achieves 3.7x on average over T-MF-CCL.
- H-ACCL achieves 2.9x speedup on average over T-S.

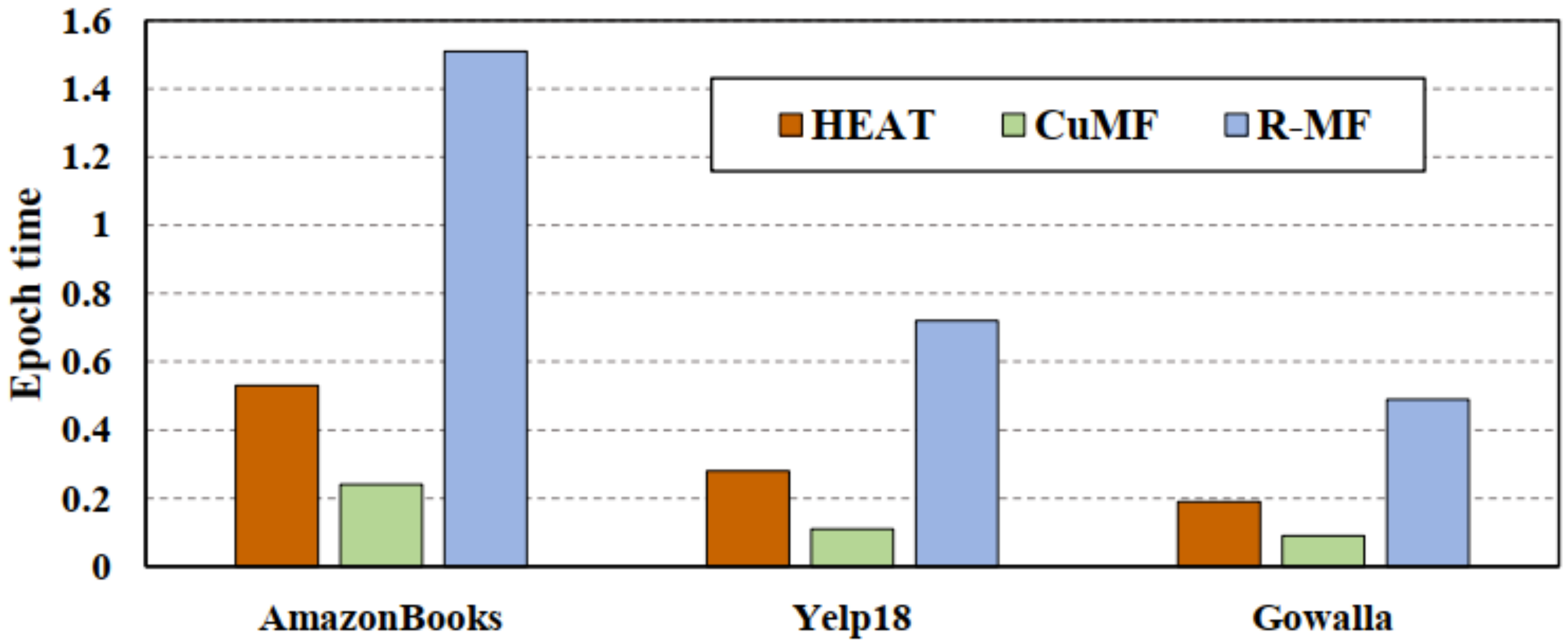


Evaluation

INDIANA UNIVERSITY
BLOOMINGTON

- **Compression CuMF_SGD and TorchRec**

- HEAT achieves 2.6x speedup on average over TorchRec-based MF.
- Performance of HEAT and CuMF is comparable.



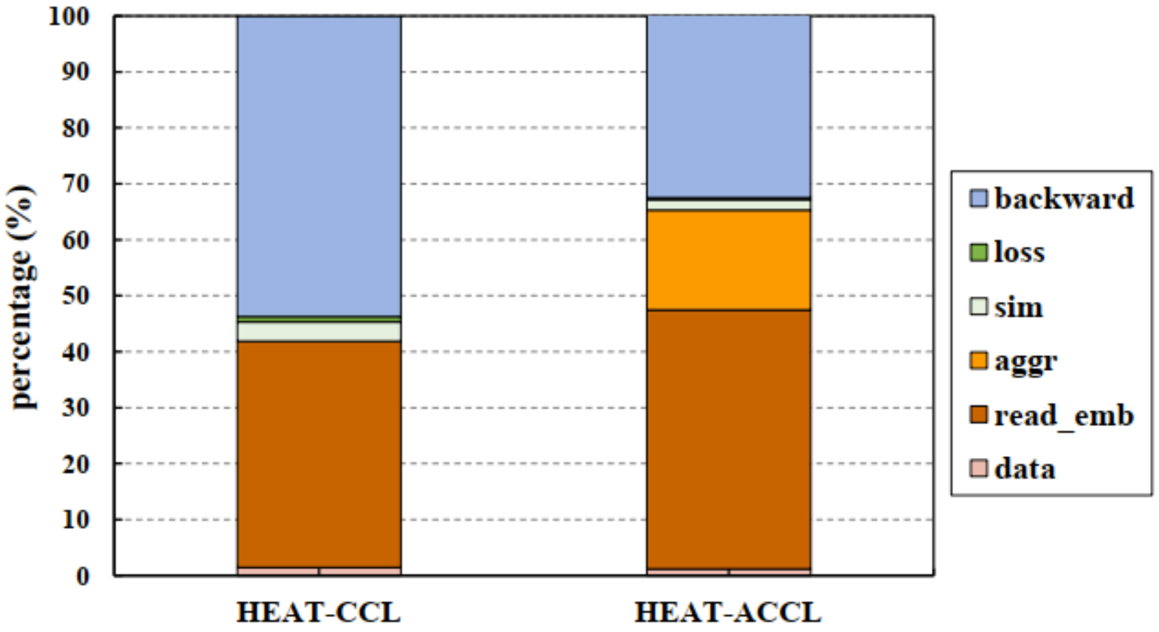
Comparison of epoch time among CuMF_SGD (GPU), TorchRec (GPU), and HEAT (CPU)

Evaluation

INDIANA UNIVERSITY
BLOOMINGTON

➢ Performance break down

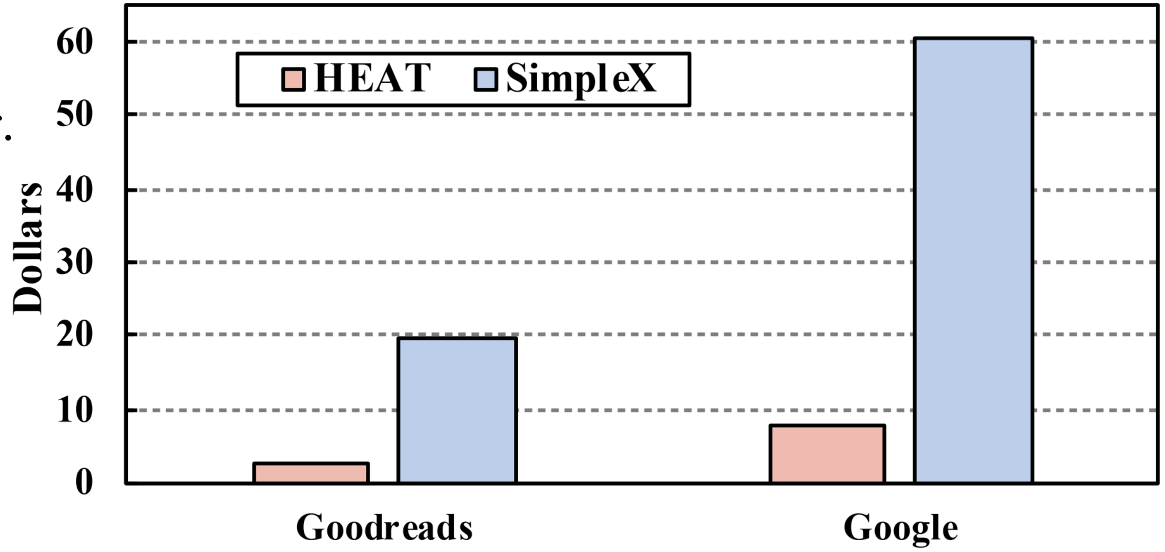
- In HEAT-CCL time of reading embeddings takes **40.4%**.
- Similarity computation including dot product and normalization only takes up **3.4%**.



Performance breakdown of HEAT on CPU. Note that sim and aggr are short for similarity computation and aggregation.

Evaluation

- **Training cost**
  - AWS p3.2xlarge (1 16 GB V100 GPU): $3.06/hour.
  - AWS c5a.16xlarge (CPU): $2.46/hour.
  - Compared with SimpleX on the GPU, HEAT can reduce the cost by 7.9x.



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

Evaluation

INDIANA UNIVERSITY
BLOOMINGTON

- **Training accuracy**
  - $Recall = \frac{TP}{TP+FN}$, where true positive ($TP$), false negative ($FN$) from confusion matrix (the larger the better).
  - NDCG normalized discounted cumulative gain (the larger the better).
  - Recall@ difference is within 0.01.

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	0.0521	0.0416	0.0651	0.0548	0.1742	0.1413
HEAT-ACCL	0.0541	0.0429	0.0683	0.0561	0.1793	0.1457

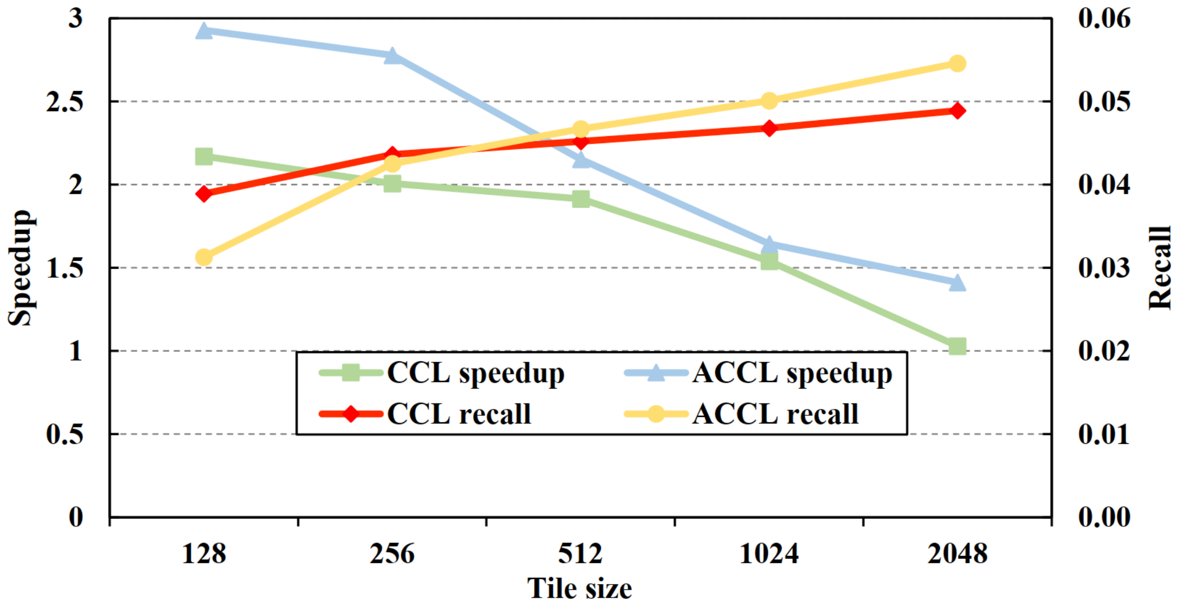
Comparison of training results under different frameworks and datasets.

Evaluation

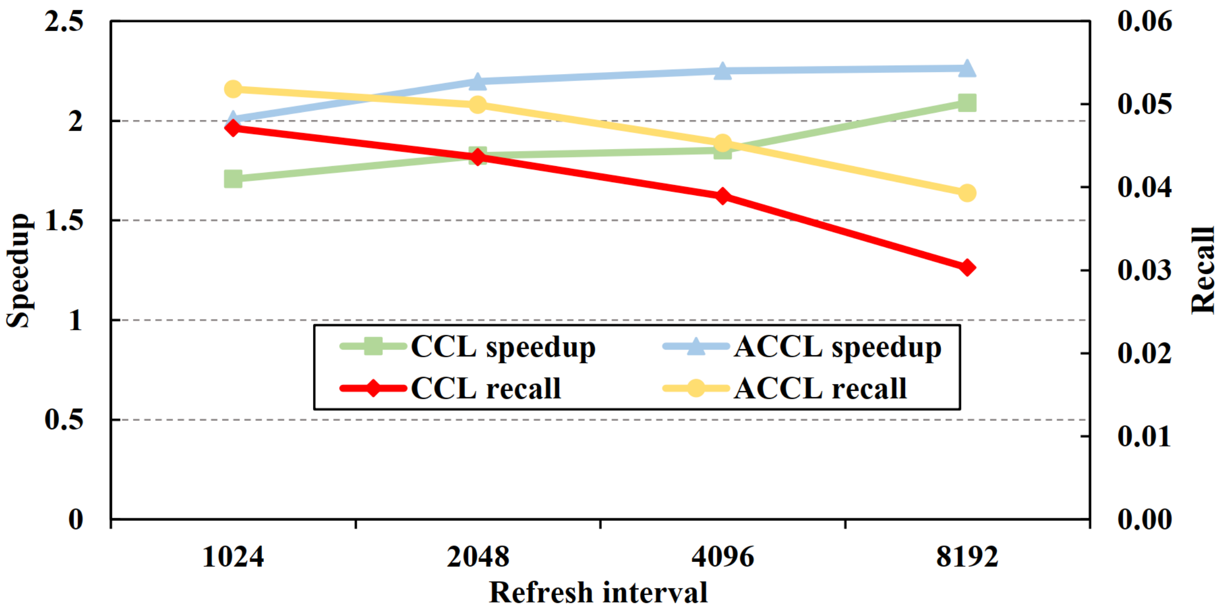
INDIANA UNIVERSITY
BLOOMINGTON

- **Impacts of Tiling Sizes and Refresh Intervals on Performance and Accuracy**

- Speedup exceeds **2x** when tiling size is less than 128.
- Recall gradually increase as tiling size increases.
- Speedup gradually increases with increasing refresh interval. But recall will gradually decrease.



Speedup & recall with different tiling sizes.



Speedup & recall with different refresh intervals.

Evaluation

➢ Impacts of tiling sizes and refresh intervals on performance and accuracy

- Random tiling sampler delivers a 1.6x speedup on average.
- Recall drop is within 0.003.

Method	AmazonBooks				Yelp18				Gowalla			
	Recall@20	Tile	Interval	Speedup	Recall@20	Tile	Interval	Speedup	Recall@20	Tile	Interval	Speedup
RCCL	0.0506	N/A	N/A	N/A	0.0625	N/A	N/A	N/A	0.1691	0.1495	N/A	N/A
RACCL	0.0527	N/A	N/A	N/A	0.0675	N/A	N/A	N/A	0.1732	0.1554	N/A	N/A
TCCL	0.0498	1024	4096	1.5	0.0608	1024	3072	1.8	0.1663	512	4096	1.7
TACCL	0.0518	1024	3072	1.6	0.0657	1024	4096	1.5	0.1716	1024	4096	1.8

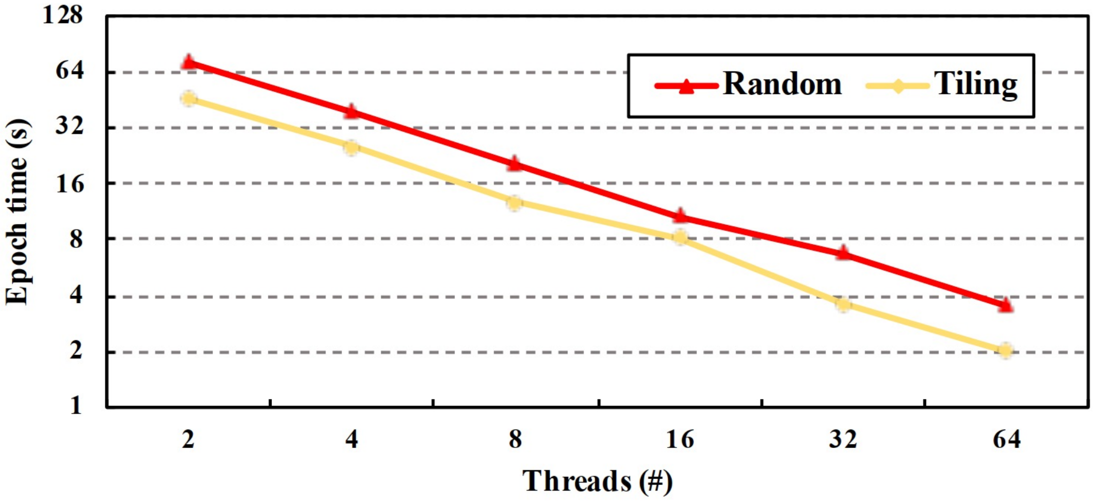
Tiling size and refresh interval for optimal training accuracy and speedup. ``R" and ``T" represent random tiling sampler and random sampler, respectively.

Evaluation

INDIANA UNIVERSITY
BLOOMINGTON

➢ **Scalability evaluation**

- Increase the number of threads/cores from 1 to 64.
- HEAT achieves the parallel efficiency of **63.7%.**



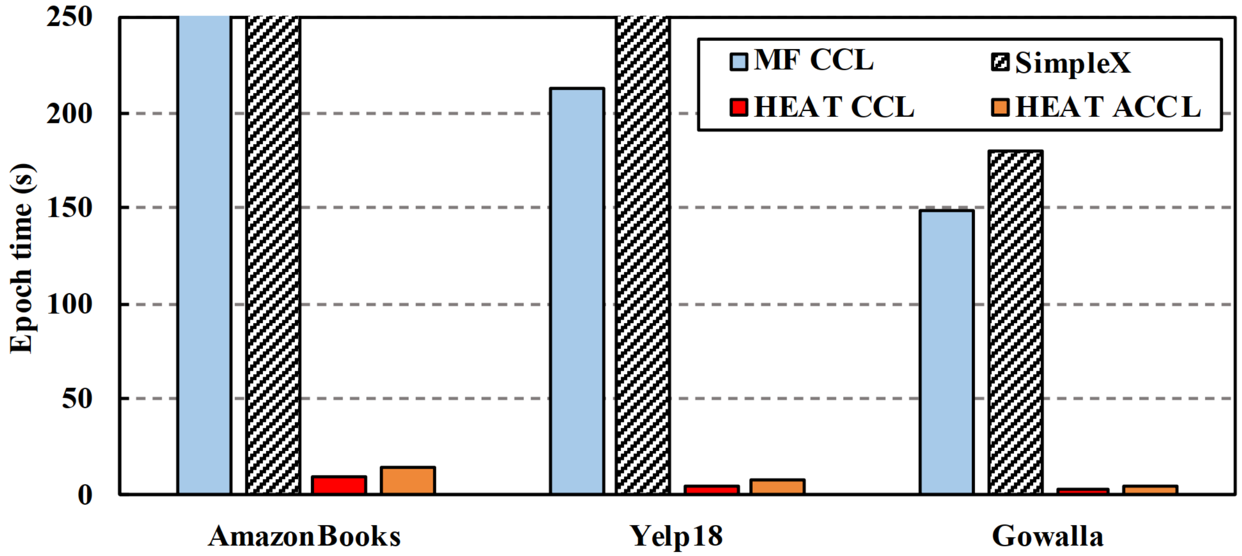
Scalability of HEAT with original random sampler (random) and our random tiling sampler (tiling).

Evaluation

INDIANA UNIVERSITY
BLOOMINGTON

➢ **Fujistu A64FX (ARM)**

- H-CCL achieves 45.7x on average over T-MF-CCL.
- H-ACCL achieves 39.8x on average over SimpleX.



Comparison of training epoch time on ARM CPUs.

BACK MATTER

Conclusion & Future Work

INDIANA UNIVERSITY
BLOOMINGTON

Conclusion

- Propose an efficient and affordable collaborative filtering-based recommendation training system.
- We propose to tile the item embedding matrix cache sizes to reduce read latency. Propose a light-weight algorithm to find the optimal tiling size and cache eviction policy.
- Save the result of the partial derivative of and reuse them.
- On AMD and ARM CPUs. HEAT achieves up to 45.2x and 4.5x speedups over existing CPU and GPU solutions, respectively, with 7.9x cost reduction.

Future work

- Deploy support distributed training with rating matrix partitioning and efficient communication.
- Apply our random tiling strategy to more recommendation models.

Thank you!

All questions and ideas are welcomed.

Contact

Chengming Zhang czh5@iu.edu
Dr. Dingwen Tao ditao@iu.edu

github.com/hipdac-lab/ICS23-HEAT

NSF

INDIANA UNIVERSITY
BLOOMINGTON

Microsoft

Argonne
NATIONAL LABORATORY