HEAT

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

Chengming Zhang  
Shaden Smith  
Baixi Sun  
Jiannan Tian  
Jonathan Soifer  
Xiaodong Yu  
Shuaiwen Leon Song  
Yuxiong He  
Dingwen Tao

**Indiana University Bloomington**  
**Microsoft**  
**Indiana University Bloomington**  
**Microsoft**  
**Indiana University Bloomington**  
**Microsoft**  
**Argonne National Laboratory**  
**Microsoft**  
**Microsoft**  
**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

- 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

- **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.
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.
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.

<table>
<thead>
<tr>
<th>Loss</th>
<th>AmazonBooks</th>
<th>Yelp18</th>
<th>Gowalla</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall@20</td>
<td>NDCG@20</td>
<td>Recall@20</td>
</tr>
<tr>
<td>BPR Loss</td>
<td>0.0338</td>
<td>0.0261</td>
<td>0.0549</td>
</tr>
<tr>
<td>Pairwise Hinge Loss</td>
<td>0.0352</td>
<td>0.0267</td>
<td>0.0562</td>
</tr>
<tr>
<td>Binary Cross-Entropy</td>
<td>0.0479</td>
<td>0.0371</td>
<td>0.0617</td>
</tr>
<tr>
<td>Softmax Cross-Entropy</td>
<td>0.0478</td>
<td>0.0367</td>
<td>0.0639</td>
</tr>
<tr>
<td>Mean Square Error</td>
<td>0.0337</td>
<td>0.0267</td>
<td>0.0624</td>
</tr>
<tr>
<td>Cosine Contrastive Loss</td>
<td><strong>0.0559</strong></td>
<td><strong>0.0447</strong></td>
<td><strong>0.0698</strong></td>
</tr>
</tbody>
</table>

Performance of MF under different loss functions.
PROFILING
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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>ET</th>
<th>FP</th>
<th>BP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmazonBooks</td>
<td>dense</td>
<td>257.4</td>
<td>19.9%</td>
<td>67.0%</td>
</tr>
<tr>
<td></td>
<td>sparse</td>
<td>946.6</td>
<td>6.2%</td>
<td>92.8%</td>
</tr>
<tr>
<td>Yelp18</td>
<td>dense</td>
<td>129</td>
<td>21.2%</td>
<td>65.1%</td>
</tr>
<tr>
<td></td>
<td>sparse</td>
<td>386.3</td>
<td>9.1%</td>
<td>89.3%</td>
</tr>
<tr>
<td>Gowalla</td>
<td>dense</td>
<td>94.9</td>
<td>20.7%</td>
<td>66.8%</td>
</tr>
<tr>
<td></td>
<td>sparse</td>
<td>251.8</td>
<td>9.2%</td>
<td>89.2%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>u_emb</th>
<th>i_emb</th>
<th>u_norm</th>
<th>i_norm</th>
<th>mem_cp</th>
<th>bmm</th>
<th>loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmazonBooks</td>
<td>9.6%</td>
<td>39.8%</td>
<td>5.9%</td>
<td>22.3%</td>
<td>5.0%</td>
<td>7.1%</td>
<td>9.7%</td>
</tr>
<tr>
<td>Yelp18</td>
<td>9.1%</td>
<td>35.3%</td>
<td>5.1%</td>
<td>28.3%</td>
<td>4.8%</td>
<td>7.2%</td>
<td>9.6%</td>
</tr>
<tr>
<td>Gowalla</td>
<td>8.3%</td>
<td>33.2%</td>
<td>5.6%</td>
<td>31.1%</td>
<td>4.8%</td>
<td>7.3%</td>
<td>9.1%</td>
</tr>
</tbody>
</table>

Breakdown of the forward phase of SimpleX.
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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>users</th>
<th>items</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodreads</td>
<td>0.81M</td>
<td>1.56M</td>
<td>4.2%</td>
<td>30.1%</td>
</tr>
<tr>
<td>Google</td>
<td>4.57M</td>
<td>3.12M</td>
<td>11.3%</td>
<td>80.2%</td>
</tr>
<tr>
<td>Amazon</td>
<td>20.98M</td>
<td>9.35M</td>
<td>38.4%</td>
<td>OoM</td>
</tr>
</tbody>
</table>

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

- (1) **Initializes** user/item embedding matrices on CPU \(\Rightarrow\) **limitation** (3)
- (2) Chooses either the original random sampler or proposed random tiling sampler \(\Rightarrow\) **limitation** (1)
- (3) Behavior aggregation layer generates a new user embedding
- (4) Calculates similarities **in parallel** \(\Rightarrow\) **limitation** (2)
- (5) Calculates gradients through an optimized gradient computation kernel \(\Rightarrow\) **limitation** (4)
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.
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:**
- $L$: # of items
- $M$: total iterations
- $N_1$: tile size
- $N_2$: refresh interval
- $n_\pi$: number of negatives
- $n_\sigma$: number of positives
- $r$: average positive hit ratio
- $s_{12}$, $s_3$: L2, L3 cache size
- $t_m$, $t_{12}$, $t_{13}$: latency of reading data from memory, L2 cache, and L3 cache
- $P$: expected speedup
- $\alpha$, $\beta$: percentage of positive, negative speedup

**Outputs:**
- $\overline{N}_1$: optimized tile size
- $\overline{N}_2$: 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. $neg\_time\_random \leftarrow M \times n_\pi \times t_m$
5. // Estimate latency of reading cache
6. $s_t \leftarrow N_1 \times sizeof(\text{embedding row}) \times \text{num\_threads}$
7. if $s_t < s_{t2}$ then
8. \hspace{1em} $t_c \leftarrow t_{t2}$
9. else if $s_t \geq s_{t2}$ and $s_t < s_3$ then
10. \hspace{1em} $t_c \leftarrow t_{t3}$
11. else
12. \hspace{1em} $t_c \leftarrow t_m$
13. end

14. // Time of reading negatives using tiling
15. $neg\_time\_tiling \leftarrow n_\pi \times M \times \frac{N_1}{N_2} \times ((N_2 - N_1) \times t_c + N_1 \times t_m)$
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. $pos\_speedup \leftarrow \frac{n_\sigma \times t_m}{n_\pi \times t_c + n_\pi \times (1 - r) \times t_m}$
18. // Percentage of speedup
19. $\alpha \leftarrow \frac{pos\_speedup}{P}$
20. // Calculate $N_1$, $N_2$
21. $N_1 \leftarrow f_0(s_{t2}, s_3, \text{num\_threads}, \text{emb\_dim})$
22. $N_20 \leftarrow \frac{M \times N_1}{P}$
23. $N_{21} \leftarrow \frac{N_1}{\alpha \times \beta}$
24. if $N_{20} < N_{21}$ then
25. \hspace{1em} $N_2 \leftarrow N_{20}$
26. else
27. \hspace{1em} $\overline{N}_2 \leftarrow N_{21}$
28. end
29. $\overline{N}_1 \leftarrow N_1$
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 $\hat{x}_{u,i}$ 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.
  - $\frac{\partial \hat{x}_{u,i}}{\partial S_u}$ 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.

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

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

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

\[ 17 \]
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.

```c
// 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_counts; ++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.
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)
Performance breakdown of HEAT on CPU. Note that sim and aggr are short for similarity computation and aggregation.
Evaluation

- **Training cost**
  - AWS p3.2xlarge (116 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.](image)
Evaluation

- **Training accuracy** • \( \text{Recall} = \frac{TP}{TP+FN} \), 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>AmazonBooks</th>
<th>Yelp18</th>
<th>Gowalla</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall@20</td>
<td>NDCG@20</td>
<td>Recall@20</td>
</tr>
<tr>
<td>MF-CCL</td>
<td>0.0559</td>
<td>0.0447</td>
<td>0.0698</td>
</tr>
<tr>
<td>SimpleX</td>
<td>0.0583</td>
<td>0.0468</td>
<td>0.0701</td>
</tr>
<tr>
<td>HEAT-CCL</td>
<td>0.0521</td>
<td>0.0416</td>
<td>0.0651</td>
</tr>
<tr>
<td>HEAT-ACCL</td>
<td>0.0541</td>
<td>0.0429</td>
<td>0.0683</td>
</tr>
</tbody>
</table>

Comparison of training results under different frameworks and datasets.
Impacts of Tiling Sizes and Refresh Intervals on Performance and Accuracy

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

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>AmazonBooks</th>
<th>Yelp18</th>
<th>Gowalla</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall@20</td>
<td>Tile</td>
<td>Interval</td>
</tr>
<tr>
<td>RCCL</td>
<td>0.0506</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RACCL</td>
<td>0.0527</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TCCL</td>
<td>0.0498</td>
<td>1024</td>
<td>4096</td>
</tr>
<tr>
<td>TACCL</td>
<td>0.0518</td>
<td>1024</td>
<td>3072</td>
</tr>
</tbody>
</table>

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

➢ 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).
Fujitsu 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

➢ 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