A Novel Multi-CPU/GPU Collaborative Computing Framework for SGD-based Matrix Factorization

Yizhi Huang
Hunan University and Zhejiang Lab
Changsha, China
huangyizhi@hnu.edu.cn

Yanlong Yin
Zhejiang Lab
Hangzhou, China
yyin@zhejianglab.com

Yan Liu*
Hunan University
Changsha, China
liuyan@hnu.edu.cn

Shuibing He*
Zhejiang University and Zhejiang Lab
Hangzhou, China
heshuibing@zju.edu.cn

Yang Bai
Hunan University
Changsha, China
baiyang@hnu.edu.cn

Renfa Li
Hunan University
Changsha, China
lirenfa@hnu.edu.cn

ABSTRACT

This paper presents a heterogeneous collaborative computing framework for SGD-based Matrix Factorization, named HCC-MF. HCC-MF can train the feature matrix efficiently using multiple CPUs and GPUs. It performs collaborative computing with data parallelism, where a server CPU is in charge of management and synchronization and other heterogeneous worker CPUs and worker GPUs performs calculation with their data assignments. HCC-MF adopts two data partition strategies, "data partition with heterogeneous load balance" and "data partition with hidden synchronization." We build a time cost model to guide the data distribution among multiple workers and we design several communication optimization techniques with consideration of datasets' and processors' characteristics. Experimental results indicate that HCC-MF can utilize more than 88% of the platform's computing power, yielding a speedup of 2.9 compared with advanced SGD-based MF, CuMF_SGD, on large-scale data sets.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**.

KEYWORDS

heterogeneous collaborative computing, matrix factorization, multi-CPU/GPU.

ACM Reference Format:
Yizhi Huang, Yanlong Yin, Yan Liu, Shuibing He, Yang Bai, and Renfa Li. 2021. A Novel Multi-CPU/GPU Collaborative Computing Framework for SGD-based Matrix Factorization. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3472456.3472520

*Corresponding author.

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472520

1 INTRODUCTION

Matrix factorization (MF) is a collaborative filtering technique and has been widely used in recommendation systems. Recently, MF and some of its variants, such as ConVF[14] and NCRPD-MF[9], have helped recommendation systems achieve high accuracy hence attracted much attention. Stochastic gradient descent (SGD) is a commonly used algorithm for solving MF. Some well-known SGD-based MF algorithms include: FPSGD[2] for multi-core CPU system, GPUSGD[13] and CuMF_SGD[27] for GPU system, MSGD[17] for multi-GPU system, and HSGD[28] (which combines FPSGD and CuMF_SGD) for single CPU-GPU system. Nowadays, multi-CPU/GPU systems are well adopted in various computing-intensive scenarios and SGD-based MF is very suitable for data parallelism, so it is natural to utilize multi-CPU/GPU collaborative computing systems for better performance.

However, it faces three challenges. i) Among multiple processors (including CPUs and GPUs), improper data distribution can lead to load imbalance. ii) The synchronization overhead caused by distributed parallel SGD may cancel out the benefits brought by parallelism. iii) Excessive communication overhead between different processors may also offset the benefits brought by parallelism. The first challenge may cause buckets effect in multi-processor collaborative computing, i.e., a slower node will drag down the performance of the entire system. The other two challenges will affect the collaborative computing system's actual acceleration rate and its utilization of computing resources.

In this paper, we design a heterogeneous multi-CPU/GPU collaborative computing framework for SGD-based MF to address the aforementioned challenges, named HCC-MF. Our work includes *modeling the time cost*, two *data partition strategies*, and *communication optimization strategies*. ch more than 97% of the best performance, and there is almost no computational time overhead.

i) *Modeling the time cost.* We establish a time cost model in HCC-MF based on the complexity of the SGD-based MF and the whole collaborative computing process. The whole process includes computing, communication and synchronization. The model is a piecewise function based on the proportion of synchronization overhead, its independent variable is the size of the input data. The data partition strategy is closely related to whether the synchronization overhead can be ignored. Therefore, we design two different data partition strategies for HCC-MF.

ii) *Data partition strategy when synchronization overhead is negligible.* When the computing cost is much greater than the synchronization cost, we focus on accelerating the computing phase and do not pay much attention to the synchronization phase. We prove that when the processor load is balanced, the computing time overhead reaches the theoretical minimum. We bring the runtime memory bandwidth into consideration, and design a data partition strategy to balance computing time on each heterogeneous processor, named "data partition with heterogeneous load balance".

iii) *Data partition strategy when synchronization overhead cannot be neglected.* When the synchronization overhead is large enough, we try to change the balanced data workload and hide the synchronization overhead with computational cost. We name this strategy "data partition with hidden synchronization". These two strategies are simple and efficient and experiments show that in the corresponding scenario, HCC-MF uses these two strategies to perform better than the basic strategy.

iv) *Communication optimization strategies.* We design a series of communication optimization strategies to reduce the communication overhead between multi-CPU/GPU. We analyze the training data characteristics of the recommendation system and propose two communication optimization strategies, *"Transmitting Q Matrix only"* and *"Transmitting FP16 Data"*. More importantly, we hide part of the communication overhead using GPU's copy engine (including discrete GPU and integrated GPU in CPU) by using the asynchronous multi-stream mechanism. In our implementation, we use shared pinned memory and multi-threaded copy for data transmission. During transmission, we try to reduce the data copy operations between different processors, and the data copy usually happens only once in one epoch. Thus, our implementation can ensure that the data transmission bandwidth of HCC-MF can reach the maximum bandwidth of the transmission channel when the amount of data is appropriate.

Our contributions are as follows.

- We design and implement HCC-MF. To our best knowledge, it is the first multi-CPU/GPU collaborative computing framework for SGD-based MF.
- We design multiple optimization strategies about computing and communication for HCC-MF to improve the efficiency of collaborative computing.
- We conduct thorough evaluations to show that HCC-MF can use almost all the computing power of each CPU and GPU in the system for collaborative computing SGD-based MF on large scale data sets. And we also quantitatively analyzed the effective range of collaborative computing and the defects of the framework.

The rest of this paper is organized as follows. Section 2 introduces background knowledge and provides our research motivation derived from some exploratory experiments. Section 3 presents the design and implementation of the proposed framework HCC-MF. Section 4 presents the evaluations. Section 5 presents related works. Section 6 concludes our work and discusses future work.

2 BACKGROUND AND MOTIVATION

2.1 SGD-based Matrix Factorization

The overall process of MF (Matrix Factorization) in the recommendation system is shown in Figure 1, in which the rating matrix **R** reflects the user's interests in the products. To decide whether to recommend a product to a user, the recommendation system needs to accurately calculate the missing interest values in **R** (the pink blocks in Figure 1). To this end, MF decomposes **R** into two matrices that contain latent features of the user-item interaction, i.e., the user matrix (often denoted as **P**) and the item matrix (often denoted as **Q**). The product of matrix **P** and matrix **Q**, denoted as $\mathbf{R}^P$, needs to be an approximation of **R**. Thus, it can be used to predict the missing interest values in **R**.

SGD (stochastic gradient descent) is a commonly used algorithm to update the **P** and **Q** in MF. MF methods that invoke SGD can be called SGD-based MF. The standard SGD is a serial algorithm. Before performing the current calculation, it must wait for the previous values to be calculated, as shown in the formula in Figure 1. Recht proposed the Hogwild! algorithm[21] and proved that asynchronous SGD can also achieve convergence under sparse data conditions in a shared memory system. It provides a theoretical basis for us to calculate SGD-based MF in parallel on each processor.

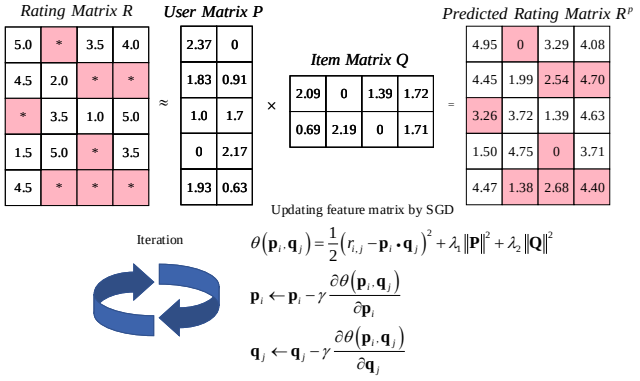


Figure 1: SGD-based MF uses the root mean square error of P∗Q and R, as well as L2-norm, to construct the loss function. The SGD algorithm will iteratively update P and Q to make the loss function converge. The values in the pink squares of $\mathbf{R}^P$ are the predicted values of * in R.

2.2 Muti-CPU/GPU architecture

Multi-CPU/GPU architecture means that multiple CPUs and multiple GPUs are on a single machine. Recently, this architecture has been widely used in HPC systems and workstations that pursue performance. For example, among the top 5 of the TOP500 list in 2020, there are three machines (i.e., Summit, Sierra, and Selene) that adopt this architecture in their individual nodes[24]. In multi-CPU/GPU architecture, multiple cores inside a CPU are interconnected by IMC[5] and they share the physical memory uniformly. CPUs not located on the same node are connected by high-speed interconnection technology, such as QPI (Quick Path Interconnect)[3], UPI

(Ultra Path Interconnect)[5], etc. Multiple GPUs can be directly connected to the CPU through PCI-E, or through NVLINK (only for some specific NVIDIA GPUs). As shown in Figure 2, as long as these connection channels are sufficient, processors can communicate in parallel without losing bandwidth. Currently, there are many unified programming frameworks for CPU and GPU, such as CUDA[22], OpenCL[19], oneAPI[10], and ROCm[1], etc., which can help us conveniently program and implement a collaborative computing framework for SGD-based MF on heterogeneous CPU and GPU.

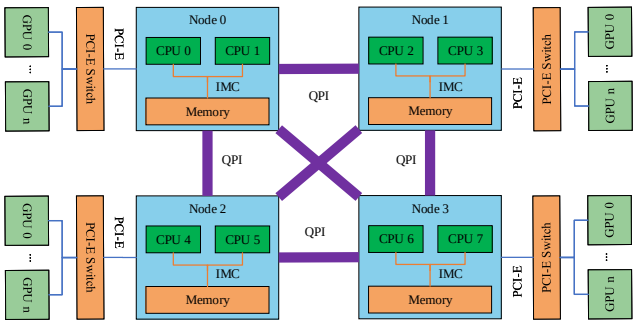


Figure 2: A typical multi-CPU/GPU architecture, widely used in servers, workstations or clusters.

2.3 The Under-utilized CPUs

Recently, as the computing performance gap between GPU and CPU has become larger[22], researchers prefer to put computing tasks on the GPU(s)[8, 11, 20, 26]. The CPU(s) in the multi-CPU/GPU system is usually only used to submit GPU tasks and control GPU tasks, and its computing performance is under-utilized. However, some studies have shown that for SGD calculations, the CPU can also provide acceleration[12, 15, 16]. In order to verify whether this conclusion is valid for SGD-based MF, we test the SGD-based MF on different single CPU and single GPU through FPSGD algorithm and CuMF_SGD algorithm, and then perform CPU/GPU collaborative computing to compute SGD-based MF in a crude and direct manner (that is, free to adjust the configuration of data partition and communication). As shown in the "Good collaboration" part in Figure 3(a), **SGD-based MF can be accelerated through multi-CPU/GPU collaborative computing, even if CPU is weaker than GPU. More importantly, it may be more economical to use additional less powerful CPUs for collaborative computing than to choose a single more powerful GPU** (as shown in Figure 3(b)). For example, the "6242-2080S" can achieve performance close to Telsa V100, but only needs less than 1/3 of its price, as shown in Figure 3. Therefore, for SGD-based MF, we should devote the CPUs in multi-CPU/GPU system to the calculation to improve performance.

2.4 Collaborative computing should be carefully designed

Data distribution. Figure 3(a) shows that the performance of SGD-based MF varies greatly on different CPU/GPU. And the execution time of collaborative computing depends on the execution time of the slowest node in the system. Therefore, if the amount of data assigned to the processors is not appropriate, the efficiency of multi-CPU/GPU collaborative computing will be very low caused short-board effect, such as the "6242-2080S (Unbalanced data)" in Figure 3(a).

Communication. There is a very troublesome communication problem, which may even completely offset the benefits of collaborative computing. It is that SGD-based MF needs to transmit feature matrices **P** and **Q**, and the amount of transmitted data is only related to the dimension of the rating matrix **R**. Even if there is a little input data, processing a rating matrix with a larger dimension will generate higher communication overhead. In this case, the efficiency of collaborative computing is very poor, such as the "6242-2080S (Bad communication)" in Figure 3(a).

Summary: For SGD-based MF, we can obtain better performance and economic benefits through multi-CPU/GPU collaborative computing. However, we must design a good collaborative computing process to deal with data distribution and communication on the processors.

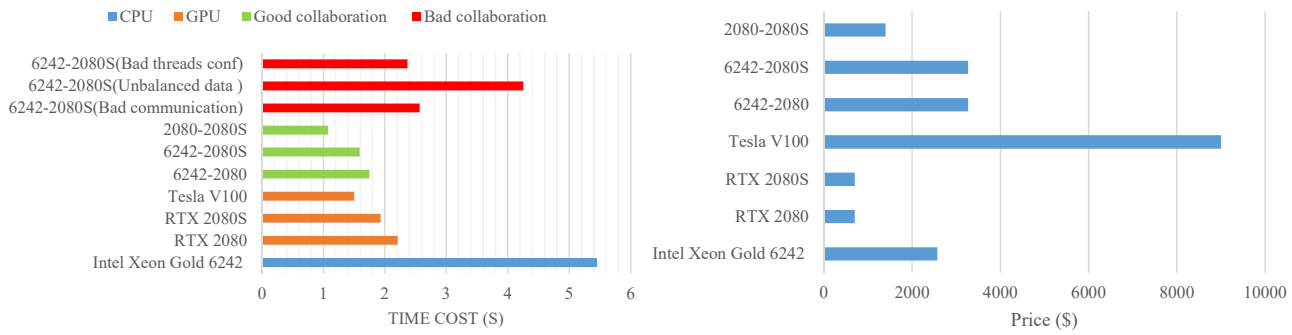
3 DESIGN OF HCC-MF

3.1 Framework description

We design a collaborative computing framework HCC-MF, hoping to efficiently utilize all the processors (including CPUs and GPUs) in multi-CPU/GPU system like Figure 2. For this, we try to increase each processor's local computing and reduce the interaction among processors and the management of HCC-MF.

The workflow of framework is shown in Figure 4, where the involved steps are denoted as ① to ⑦. First of all, before the training starts, the framework will preprocess the input rating matrix (steps ① to ③), which involves operations such as shuffling, sorting, and data partition. These steps will prepare training data and initialize feature data for each worker. Afterwards, it comes to the training process. At the beginning of a training epoch, the workers will pull feature matrix from the server in parallel (see step ⑤). Then the worker computes SGD-based MF in the data parallelism way (step ⑥). After the local feature matrices $\mathbf{P}_n$ and $\mathbf{Q}_n$ is updated, the local feature matrix will be pushed to the server in parallel (step ⑦). Then, the server will synchronize the data from each worker to the global feature matrices **P** and **Q** (the step ④ called sync). The operations of "pull→computing→push→sync" will be repeated until the objective function converges.

As the above workflow, HCC-MF is designed as an "asynchronous + synchronous" collaborative computing mode based on parameter server[18]. Each processor as a worker calculates its own data and updates its training results asynchronously. Processor as a server is responsible for synchronous management. Synchronization management is necessary. The reason is that, no matter how the data is divided, it is always possible for multiple workers to process the same row (column) of data. In this way, they will update the training results to same row (column) in the global feature matrix **P** (**Q**). This will cause data race of write-after-write (WAW). As the number of processors in the system increases or the sparsity of input data decreases, the risk increases.



(a) The performance of SGD-based MF on different platforms

(b) The prices of different platforms

Figure 3: (a) In the test, we use the Netflix dataset for training and record the training time of 20 epochs. The "Bad collaboration" is the result of random configuration and no communication optimization in collaborative computing. The "Good collaboration" is the result of a carefully adjusted configuration. The "CPU" and "GPU" are result of training on corresponding independent processor. (b) Hardware platform costs.

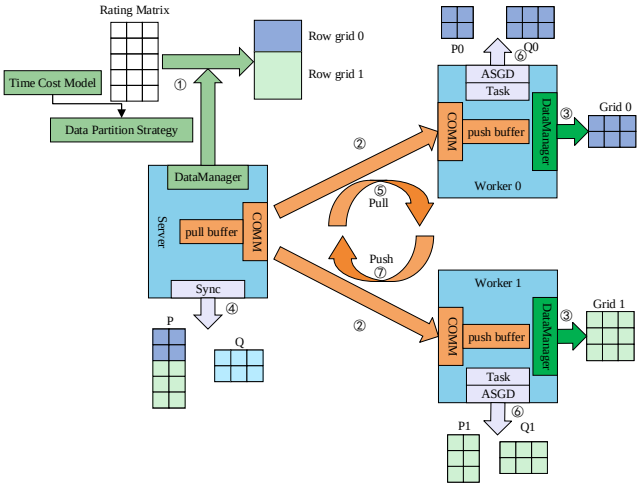


Figure 4: The workflow of HCC-MF.

Table 1: Model parameters in HCC-MF

Symbols	Descriptions
m,n	Rows and columns of rating matrix
nnz	The number of elements in rating matrix
k	Columns of matrix P and rows of matrix Q
c, g, p	The number of CPUs, GPUs and total processors
T_i	The time cost of i-th worker
T_{i_pull}	The pull time of i-th worker
T_{i_c}	The computing time of i-th worker
T_{i_push}	The push time of i-th worker
T_{i_sync}	The synchronization time of i-th worker
P_i	Runtime computing performance of i-th worker
P_{server}	Runtime computing performance of server
B_i	Runtime memory bandwidth of i-th worker
B_{server}	Runtime memory bandwidth of server
B_{i_bus}	Runtime bus bandwdith connecting i-th worker and server
T_{i_e}	The independent execution time in i-th worker
x_i	Data partition parameters of i-worker
λ	The threshold of ignoring synchronization overhead

3.2 Time cost of HCC-MF

We establish a time cost model to guide the detailed design data partition and communication of HCC-MF. The model describes the overall time cost of a training epoch in collaborative computing. As shown in Figure 5, the time overhead of an epoch is composed of the longest time among all workers and its subsequent synchronization time on the server, which can be written as follow:

$$T = max\{T_i\} + T_{sync}, \tag{1}$$

where $T_i = T_{i\_pull} + T_{i\_c} + T_{i\_push}$. When we know the type of bus that connects the worker and the server, in theory, the cost of pull and push are the same, and they are both equal to a constant, $k(m+n)/B_{bus\_i}$. The time for the worker to process one input data is $7k/P_i + (16k+4)/B_i$. Since the performance of the processor is always much greater than the memory bandwidth, i.e., $P_i \gg B_i$, the $7k/P_i$ term can be ignored, therefore

$$T_i \approx \frac{x_i nnz\,(16k+4)}{B_i} + \frac{2k(m+n)}{B_{bus\_i}}. \tag{2}$$

The server synchronizes a feature parameter with three read and write memory operations and one multiply-add operation. We assume that there are t synchronizations after the max time cost, then we get the synchronizations time:

$$\begin{aligned} T_{sync} &= \sum_{i=1}^{t} T_{i\_sync} = \sum_{i=1}^{t} \left[\frac{3k(m+n)}{B_{server}} + \frac{k(m+n)}{P_{server}} \right] \\ &\approx \frac{3tk(m+n)}{B_{server}}. \end{aligned} \tag{3}$$

Substitute Equations (2) and (3) into Equation (1), then:

$$T = max\left\{\frac{x_i nnz\,(16k+4)}{B_i} + \frac{2k(m+n)}{B_{bus\_i}}\right\} + \frac{3tk(m+n)}{B_{server}}, \tag{4}$$

where the constraints are $\sum_{i=1}^{p} x_i = 1$. And t is also a function related to the data partition $\mathbf{x}$. When the computational cost is much higher than the synchronization cost, the synchronization overhead is too small to affect the overall overhead. We can ignore it and the time cost model will become a piecewise function:

$$T = \begin{cases} max\left\{T_i(x_i)\right\} & max\left\{T_i\right\}/T_{sync} \geq \lambda, \\ max\left\{T_i(x_i)\right\} + T_{sync}(\mathbf{x}) & max\left\{T_i\right\}/T_{sync} < \lambda. \end{cases} \tag{5}$$

where λ is a threshold that indicates that synchronization overhead can be ignored. Its value should be changed with the scale of execution time. In our test environment, HCC-MF can complete training on different data sets within 30s, and the fastest can be less than 1s. Therefore, we take its value as 10 in experiments.

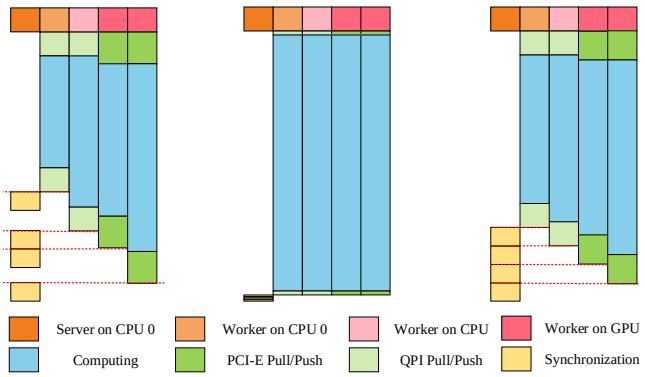


Figure 5: Different timing sequence in a training epoch. The left sub-figure is the original timing sequence without any optimization. The middle one is the optimized timing sequence without considering the synchronization overhead. The right one is the optimized timing sequence with considering the synchronization overhead.

3.3 Data distribution management

Data distribution management is a key module on server named "DataManager" which is responsible for assigning data to each worker. In "DataManager", the data grid affect the synchronization management of server and the calculation content of workers, and the data partition strategy determines the load of the workers. In this section, we discuss the design of data grid and data partitioning strategy.

Row (column) grid. In HCC-MF, the server divides the data in the form of row (column) grid. It means the server divides the rating matrix $\mathbf{R}$ to each group of rows (columns) which will be assigned to a corresponding worker, as the step ⑥ in Figure 4. When input rating matrix has more rows than columns, the row grid is adopted, otherwise, the column grid is used. However, using the row grid, the server must synchronize submitted results of each worker to avoid data race. This only requires adding a multiply-add operation after each data submitting. For distributed systems, submitting results is always necessary, the increase in synchronization overhead is limited. More importantly, in terms of communication overhead, using row grids will have advantages, which we will discuss in detail in section 3.4. Therefore, in the "DataManager" on server, we adopt the form of row (column) grid to divide the data.

Data partition with heterogeneous load balance. According to the first case of the time cost model, we can establish the optimization objective function:

$$\theta(\mathbf{x}) = min\left\{T\right\} = min\left\{max\left\{\frac{x_i nnz\,(16k+4)}{B_i} + \frac{2k(m+n)}{B_{bus\_i}}\right\}\right\}.$$

In order to solve the equation conveniently, we assume that the memory bandwidth during the computing of the worker will not change with the scale of the input data. It should be noted that in this section we only discuss data partition, so other condition variables, such as the number of threads, are fixed. Thus, it has a general form $\theta(\mathbf{x}) = min\left\{max\left\{Ax+B\right\}\right\}$. And both A and B are constants. For this function, we can obtain a useful theorem:

Theorem 1: When $\sum_{i=1}^{m} x_i = 1$, $\exists\left\{x_1, x_2, ..., x_i, ..., x_m\right\}$ minimizes the $T(x) = max\left\{Ax+B\right\}$, if and only if $a_1x_1 + b_1 = a_2x_2 + b_2 = ... = a_mx_m + b_m$.

proof 1: Assuming it is not $a_1x_1 + b_1 = a_2x_2 + b_2 = ... = a_mx_m + b_m$, minimizes the $T(x)$. Because $\sum_{i=1}^{m} x_i = 1$, then there is a pair $\left\{x_i', x_j'\right\}$ in the solution set, let $a_ix_i' + b_i \neq a_ix_i + b_i$, $a_jx_j' + b_j \neq a_jx_j + b_j$, and $x_i' + x_j' = x_i + x_j$. If $a_ix_i' + b_i < a_ix_i + b_i$, then $x_i' < x_i$, $x_j' < x_j$ and $a_jx_j' + b_j > a_jx_j + b_j$. At this time, $T(x)$ is not the minimum value, which contradicts the assumption. The theorem 1 is proved.

For the HCC-MF, a_i is the time cost independently calculated by the i-th worker . And $b_i = k(m+n)/B_{bus\_i}$, that is the communication time of the i-th worker. Since the bandwidth of x16 PCI-E Gen3 is close to that of QPI (16GB/s v.s. 16~20.8GB/s), we consider the communication overhead on each worker to be equal. Thus, according to Theorem 1, when $a_1x_1 = a_2x_2 = ... = a_px_p$, the objective function reach minimum. It means that the computing cost of each worker is equal, the overall cost is the minimum, like the middle diagram Figure 5. Then we get the solution of objective function $\mathbf{x}$ that represents the number of rows assigned to workers:

$$x_i = \frac{1}{\sum_{j=1}^{p} \frac{a_i}{a_j}} = \frac{1}{\sum_{j=1}^{p} \frac{T_{j\_e}}{T_{j\_e}}}. \tag{6}$$

This is the basic data partition strategy, named "DP0".

Table 2: Memory bandwidth (GB/s) of different data partitions under the same configuration. "IW" is shot for "Independent Worker" which means that each worker processes all the input data individually. "DP0" means dividing input data to workers according to the DP0 strategy.

Worker	6242	6242l-10	2080	2080S
IW	67.3001	39.31905	378.616	407.095
DP0	67.75335	39.5995	388.7935	412.042

However, in reality, memory bandwidth is not necessarily constant. With the help of profiling tools Intel PCM[4] and NVIDIA

Nsight Systems[6], we find that, while varying input data size of the worker, the bandwidth of the CPU is constant, but GPUs' bandwidth has a little change. The results are shown in Table 2. In addition, we neglected some non-critical factors when modeling, which are i) P_i when deriving equation 2, and P_{server} when derive equation 3. These factors make that after dividing the data according to DP0, the computational time of CPU and GPU is unbalanced (the gap is small), but the computational time of the same type of processor is still balanced. Therefore, we need to fix the data partition strategy to ensure a balanced computational time.

We have obtained an observation from a lot of tests: when the input data changes in a small range, the memory bandwidth at runtime will not change much, and it can be considered as a constant. Using this feature, we can linearly find a new data partition based on the data partition, computing cost, and unbalanced cost obtained by DP0, such as the lines 5-11 in the Algorithm 1. It is a data partition strategy with heterogeneous load balance named DP1 as the Algorithm 1. Since the computing time of each worker by using DP0 is very close, using the DP1, the computational cost can be balanced by executing the algorithm a few times (usually only once).

Algorithm 1 Compensation algorithm

Input: Old data partition $\{x_{b\_1}, x_{b\_2}, ..., x_{b\_p}\}$; The computing time $\{t_1, t_2, ..., t_p\}$;
Output: New data partition $\{x_1, x_2, ..., x_p\}$

1: $T_{avg\_cpu} \leftarrow \frac{1}{c}\sum_{i=1}^{c} T_{i\_cpu}, T_{avg\_gpu} \leftarrow \frac{1}{g}\sum_{i=1}^{g} T_{i\_gpu}$
2: **while** $\frac{|T_{avg\_cpu} - T_{avg\_gpu}|}{min(T_{avg\_cpu}, T_{avg\_gpu})} > 0.1$ **do**
3: $\quad T_{avg\_cpu} > T_{avg\_gpu}\ ?\ l \leftarrow 1\ :\ l \leftarrow -1$
4: $\quad \Delta T \leftarrow \frac{l(T_{avg\_cpu} - T_{avg\_gpu})}{c+g}$
5: $\quad$ **for** $i = 1 \rightarrow c$ **do**
6: $\quad\quad x_{i\_cpu} \leftarrow \frac{x_{b\_i\_cpu}(t_{i\_cpu} - lg\Delta T)}{t_{i\_cpu}}$
7: $\quad$ **end for**
8: $\quad$ **for** $j = 1 \rightarrow g$ **do**
9: $\quad\quad x_{j\_gpu} \leftarrow \frac{x_{b\_j\_gpu}(t_{j\_gpu} + lc\Delta T)}{t_{j\_gpu}}$
10: $\quad$ **end for**
11: $\quad \{x_{b\_1}, x_{b\_2}, ..., x_{b\_p}\} \leftarrow \{x_{cpu}\} \cup \{x_{gpu}\}$
12: $\quad \{t_1, t_2, ..., t_p\} \leftarrow sgd\_update(\{x_{b\_1}, x_{b\_2}, ..., x_{b\_p}\})$
13: $\quad T_{avg\_cpu} \leftarrow \frac{1}{c}\sum_{i=1}^{c} T_{i\_cpu}, T_{avg\_gpu} \leftarrow \frac{1}{g}\sum_{i=1}^{g} T_{i\_gpu}$
14: **end while**
15: $\{x_1, x_2, ..., x_p\} \leftarrow \{x_{b\_1}, x_{b\_2}, ..., x_{b\_p}\}$
16: **return** $\{x_1, x_2, ..., x_p\}$

Data partition with hidden synchronization. In the second case, synchronization overhead will affect the overall time overhead. Since the T_{sync} is a variable, Equation (5) can no longer be equivalent to linear equations. Existing constraints are too few, it is difficult to find an exact solution for the objective function, $\theta(\mathbf{x}) = min\{max\{T_i(x_i)\} + T_{sync}(\mathbf{x})\}$. Therefore, we designed a greedy heuristic method to find sub-optimal solution. First, we use the time overhead and data partition generated by DP1 as the initial conditions. Then, we hope that the time cost of each worker changes according to the law which the synchronization overhead of the i-th worker can always be hidden by the computational cost of the $i+1$-th, as shown right diagram in Figure 5. Accordingly, we get the following equation:

$$T_{(i\pm n)\_c} = T_{i\_c} \pm nT_{i\_sync}. \tag{7}$$

Finally, we take the initial conditions as the median value, and compensate synchronization time to each worker up (+) or down (-) according to Equation (7). Thus, we can get new data partition with hidden synchronization named DP2 by using method similar to the line 6 in Algorithm 1.

3.4 Communication optimization

It is necessary to optimize communication overhead. First, in HCC-MF, only the feature matrix **P** and **Q** are transmitted between the worker and the server. Therefore, the transmission overhead is related to the dimensions of the rating matrix. The larger the dimension of the matrix, the larger the transmission overhead. Second, communication overhead may account for a large proportion of the total cost. According to the Equation 2, the ratio of communication overhead to computing cost can be obtained to be about $B_i(m+n)/(8x_i \times nnz \times B_{bus\_i})$. Generally, considering the cache, the ratio of B_i and B_{bus_i} ranges from 10 to 100. Then, if the difference between matrix dimensions and matrix elements is less than three orders of magnitude, that is $nnz/(m+n) < 10^3$, communication overhead and computing time will be in the same order of magnitude. Therefore, we must optimize the communication of the system. According to the characteristics of data transmission, we design three communication optimization strategies to ensure efficient parameter data transmission in the HCC-MF. The core ideas are as follows:

- Reduce the amount of data transmission without compromising data accuracy (data accuracy will affect the convergence speed of training).
- Construct an asynchronous execution pipeline to overlay transmission with computing.

These strategies include: *"Transmitting* **Q** *matrix only"*, *"Transmitting FP16 Data"* and *"Asynchronous Computing-Transmission"*.

Strategy 1: Transmitting Q matrix only. It means that, during the SGD-based MF training, all workers only need to pull or push **Q** matrix in every epoch, except for the last push in which both **P** and **Q** matrix need to be pushed. The MF application has practical significance. Each row in the rating matrix **R** is the user's rating for a series of products. In a large number of random user scenarios, the rows are independent of each other (the columns are same). In this way, the rows of the feature matrix **P** will not interfere with each other in the SGD update, as shown in Figure 4. Combined with the row grid data partition, the local feature matrix **P** of the worker does not need to be updated to the global feature matrix **P** during training. Therefore, for the epochs in the middle of the training process, each worker only needs to transmit **Q** matrix. Correspondingly, if there are more items than users, the strategy can also be switched to *"Transmitting* **P** *matrix only"*. According to this transmission strategy, the amount of transmitted data is only about $n/(m+n)$, but the accuracy of training data will not be affected, where $(m+n)$ is the original amount of data transmission

without employing this strategy. In the data sets of the actual recommendation systems, the number of users is much greater than the number of products, i.e, $m \gg n$. For example, in the Netflix data set, there are 480,190 users and 17,771 items. Our strategy can reduce a large proportion of the original transmission (the proportion lower bound is 1/2, obtained when $m = n$). In this case, the strategy can reduce data transmission by about 96.4% in practice.

Strategy 2: Transmitting FP16 Data. In the actual recommendation system, common scoring methods include 5-point scale, 10-point scale and 100-point scale. The number of possible values of the score is finite. The step size of the score value is usually relatively large. For example, we generally use 1 point or 0.5 points as the step length in the 100-point system. That means, the feature matrices **P** and **Q** do not need too high precision to represent the score matrix **R**. Therefore, we can compress the feature matrix data and then transmit it. This approach will not affect the accuracy of the target rating matrix. We implement the IEEE-754 FP32 and FP16 conversion functions on the GPU and CPU respectively in the COMM component for data compression and decompression. In order to ensure conversion efficiency, we use AVX instructions and multi-threaded parallel acceleration conversion on the CPU. In theory, the strategy of transmitting compressed data can further reduce the data transmission by 1/2.

Strategy 3: Asynchronous Computing-Transmission. The first two strategies are designed for the case where m is much greater than n. While in practical applications, it is still possible to encounter a situation where the number of rows and columns is roughly the same. In this situation, the first two strategies can only reduce the communication overhead to 1/4 in theory. If the rating matrix has a particularly large number of rows and columns, the communication overhead will still be high. For this reason, we establish multiple asynchronous "pull-computing-push" pipelines on each worker, trying to hide the communication overhead of pull and push with computational cost, as show in Figure 6. The copy engine on the GPU and the multi-stream mechanism provided by CUDA make it easy for us to implement this strategy for the GPU workers. For CPU workers, this strategy can only be implemented if the CPU has an integrated GPU. We can build such multiple pipelines by using the copy engine of integrated graphics (like Intel BLT engine). In this case, synchronization on the server will occur in the middle of the process, and the CPU acting as the server will no longer be time-sharing as worker.

3.5 Implementation of HCC-MF

Transparent utilization of the CPUs and GPUs. We implement the HCC-MF that can use all CPUs and GPUs in the system to accelerate SGD-based MF. In HCC-MF, one CPU in the system is designated as the parameter server, and all the other connected CPUs and GPUs are workers. In addition, when "Asynchronous Computing-Transmission" is not used, a special worker will be created to time-sharing reuse the server's CPU. The server and the workers are designed as process instances, so that both CPU and GPU are transparent to users. Server and CPU workers is bound to the corresponding CPU through setting CPU affinity. GPU workers is bound to the parameter server. Then the corresponding GPU is selected using device API, like CUDA API "cudaSetDevice" or

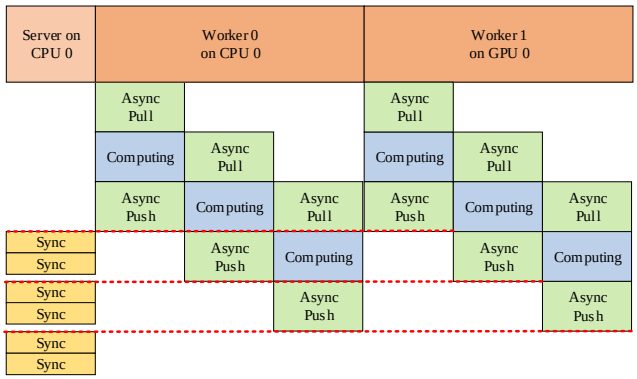


Figure 6: Asynchronous computing-transmission does not reduce computational time. Asynchronous pull and push make the transmission hidden by calculation. In theory, it can reduce the transmission overhead to $1/streams$.

OpenCL API "clCreateContext". Then, the server will create a thread to synchronize the received training results from workers, named "Sync". The worker will create its own thread pool and train the feature matrix asynchronously, as shown in the Figure 4.

Implementation of communication module. "*COMM*" is a communication module between the server and workers, which connects the server and the workers through IPC (Inter-Process Communication). The communication optimization strategies mentioned in the previous chapter are integrated in "COMM". It provides two simple interfaces, "*Pull*" and "*Push*", to transfer feature matrix between worker and server. In order to make full use of the independent QPI and PCI-E channel in the system shown in Figure 2, the "*COMM*" create a piece of shared memory on each server and worker, called "*pull buffer*" and "*push buffer*" respectively, as shown in Figure 4. The "*pull buffer*" is mapped into the address space of each worker and the "*push buffer*" is mapped into the server by "COMM". In this way, the "*Pull*" and "*Push*" of each worker is parallel. More importantly, if the feature matrix is directly stored in the respective shared memory, then the communication between the worker and the server only needs to copy once.

4 EVALUATION

In this section, we evaluate the proposed HCC-MF on real open source datasets. To demonstrate the performance of HCC-MF, we design several experiments around the following key questions:

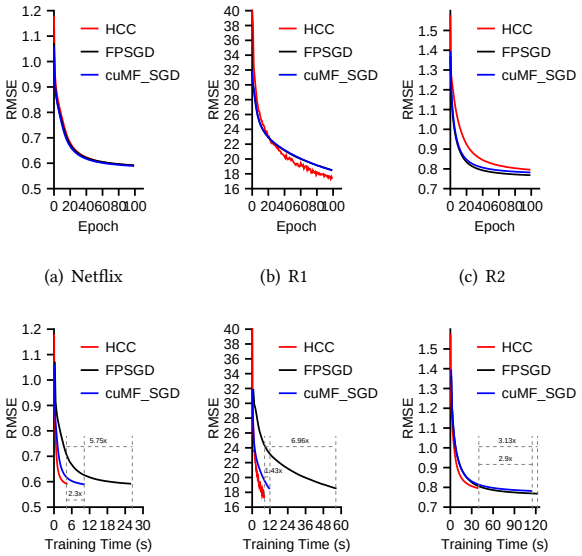
- In a real multi-CPU/GPU system, can HCC-MF effectively improve the execution performance of an SGD-based MF application? (Section 4.2)
- Does the core design proposed in Section 3 effectively improve the performance of collaborative computing? (Section 4.3 and Section 4.4)
- What is the HCC-MF's utilization of computing resources in multi-CPU/GPU system? (Section 4.5)
- What are the limitations of HCC-MF? (Section 4.6)

4.1 Experimental setup

We conduct experiments on a multi-CPU/GPU workstation with two Intel(R) Xeon(R) Gold 6242 CPUs, one NVIDIA RTX 2080 GPU (GPU_1, configured with 41,216 threads) and one NVIDIA RTX 2080 Super GPU (GPU_0, configured with 43,008 threads). The two GPUs are directly connected to CPU_0 with PCI-E 3.0 x16 and the CPU_1 (configured with 24 threads) are connected to CPU_0 with Intel UPI (Intel Ultra Path Interconnect). In the overall performance test, CPU_0 is configured with 16 threads to obtain the maximum performance. In other tests, in order to increase the heterogeneity between CPU_0 and CPU_1, we configure CPU_0 with 10 threads. We adopt relevant data sets and training parameter settings of MF according to existing studies, as shown in Table 3, where Netflix, R1, R2, and Movielens are real data sets, while R1* is a data set generated by adding data according to the uniform distribution on the basis of R1 in order to verify the data distribution strategy.

Table 3: Parameters of data sets and SGD-based MF training, where the learning rate γ is 0.005.

Data Set	m	n	nnz	λ_1, λ_2
Netflix	480190	17771	99072112	0.01
Yahoo! Music R1	1948883	1101750	115579437	1
R1*	1948883	1101750	199999997	1
Yahoo! Music R2	1000000	136736	383838609	0.01
Movielens-20m	138494	131263	20000260	0.01



(a) Netflix (b) R1 (c) R2

(d) Netflix (e) R1 (f) R2

Figure 7: (a), (b) and (c) is the comparison of the convergence rate on different data sets, indicating the convergence effect under a certain training epoch. (d), (e) and (f) is comparison of the training speed on different data sets.

4.2 Overall performance

We evaluate the overall performance of HCC-MF from three aspects: training effect, training speed, and utilization of processors. As far as we know, currently there is no program or framework for MF applications on multi-CPU-GPU heterogeneous systems. We compare HCC-MF with FPSGD and CuMF_SGD, which are the state-of-the-art SGD-based MF approaches on a single processor, to show the performance of HCC-MF. It should be noted that we did not directly use their open source code. We modify the open source implementation of these methods (faster than their open source implementation)[1], and use them as the computing task of HCC-MF. Considering the fairness of comparison and correctness of collaborative computing evaluation, we use the modified versions of FPSGD and CuMF_SGD as baselines.

On the one hand, we show that HCC-MF can accelerate the training speed while still having the same convergence rate as the single processor algorithms. Parallel SGD may update a parameter at the same time, which will cause convergence problems, such as slow convergence or non-convergence[25]. We first tested the training effect after 100 epoch training, the results of convergence rate and training speed are shown in Figure 7. It can be seen that on different data sets, the convergence rate of SGD-based MF in HCC-MF is equivalent to that in single-processor methods, but the training speed in HCC-MF is faster than that in single-processor methods. For example, on R2, the training speed of HCC-MF is 2.9x that of CuMF_SGD and 3.1x that of FPSGD. Of course, it must be noted that on R1, due to the use of the "asynchronous calculation-transmission strategy", a small part of the training results is lost. To obtain high training speed, the execution in worker is completely asynchronous. Therefore, several asynchronous streams in a same worker may train the same row of parameters concurrently, resulting in the coverage of the training results. However, Hogwild![21] proves that this influence is relatively small if the data are sparse and random enough. The data HCC-MF processes can meet these requirements. Because the rating matrix is always sparse, and there are only a few asynchronous streams in HCC-MF. As shown in Figure 7 (b) and (e), the loss function fluctuates during training, but it maintains a downward trend until it converges.

On the other hand, we evaluate whether HCC-MF fully utilizes the computing performance of the heterogeneous system. The traditional peak performance cannot well evaluate the real performance of the application, and the speedup is not enough to reflect the utilization of heterogeneous processors by system. Therefore, we define the "computing power" instead of peak performance and the "computing power utilization" instead of speedup to further evaluate the overall performance of the multi-CPU/GPU system. "Computing power" reflects the computing performance of a processor for a certain application. In SGD-based MF, *"computing power" means the number of rating matrix elements that can be updated per*

[1]The main modifications: i) We added the implementation of AVX and SSE instructions to the parameter update kernel in FPSGD, which is only implemented in ordinary C language in the open source code. Under the same configuration, the open source version training Netflix 20 times will take 22.9s, while our version only needs 12.9s. ii) we have added the implementation version of AVX512F to the kernel function of inner product and parameter update. Using AVX512F, on our test platform, our implementation training Netflix 20 times only takes 5.5s. iii) Refer to the FPSGD, we added the block sorting by row to the grid_problem of CuMF_SGD to improve cache hit rate. 20 Netflix training time is reduced from 2.84s to 2.25s on RTX 2080.

unit time on a certain processor, which can be written as:

$$computing\_power = \frac{nnz \times epochs}{cost\_time} \tag{8}$$

Ideal "computing power" of system is the sum of the "computing power" of all the processors in the system. "Computing power utilization" is used to describe the utilization of heterogeneous processors by HCC-MF. We define it as the ratio of the HCC-MF's actual "computing power" to the ideal "computing power" of system. In Table 4, the first four data columns present the "computing power" of several processors to independently calculate SGD-based MF, the column "Ideal" presents the ideal "computing power" of the system and the HCC-MF column is the tested "computing power" of HCC-MF in the experiments. According to Table 4, on Netflix and R2 data sets, HCC-MF shows high utilization (more than 85%) of the ideal computing power. On R1, HCC-MF also has a good utilization (62%) of the ideal computing power. On MovieLens, HCC-MF utilize 46% of the ideal computing power, that's because this type of data set is not suitable for multi-CPU/GPU acceleration, we will discuss it in section 4.6.

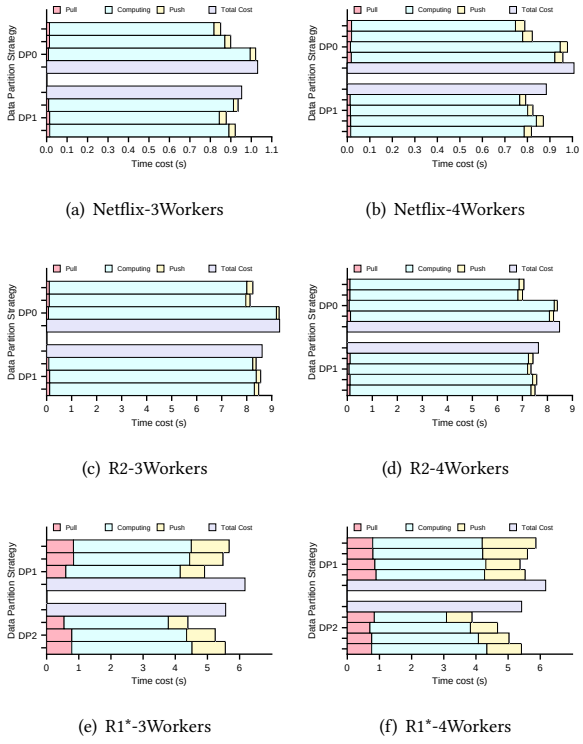
4.3 Data partition strategy evaluation

In this set of experiments, we verify that the timing sequence of HCC-MF is consistent with the proposed time cost model, and demonstrate the performance of HCC-MF's data partitioning strategy. According to our design, when synchronization overhead can be ignored, dividing data according to DP1 HCC-MF should obtain a more balanced load and better performance, when synchronization overhead needs to be considered, dividing data according to DP2 should hide synchronization overhead and obtain better performance.

We run the training process for 20 epochs and record the cumulative time cost of the pull, computing, and push operation(including push and sync), respectively. The results are shown in Figure 8. Comparing Figure 8 and Figure 5, we can find the actual execution process of HCC-MF is consistent with the proposed time cost model. Next, we demonstrate the performance of DP1 and DP2. On one hand, HCC-MF finds that the computational cost is much higher than the synchronization overhead on Netflix and R2, so it chooses the DP1 strategy. In this case, the computing time of DP1 is more balanced, and the overall overhead is also smaller, which are reduced by 12.2% and 10% respectively compared with DP0, as shown in the Figures 8 (b) and (d). On the other hand, for R1, HCC-MF considers the synchronization overhead and chooses DP2. In Figures 8 (c) and (f), the computing cost of DP2 is not balanced, but the synchronization overhead of the previous worker are hidden. It enables the HCC-MF to end the current iteration as soon as the last worker ends, while DP1 has to wait for a period of time. Therefore, DP2 gets a smaller overall time cost than DP1, for example, it decreased by 12.1% on R1*-4workers.

4.4 Communication evaluation

Communication performance is an important performance indicator of HCC-MF. From the above-recorded data, we can obtain the communication time cost of HCC-MF, which are the sum of the cumulative time cost of pull and push, as shown in Table 5. It



(a) Netflix-3Workers (b) Netflix-4Workers

(c) R2-3Workers (d) R2-4Workers

(e) R1*-3Workers (f) R1*-4Workers

Figure 8: Time statistics of 20 epochs using different data partition strategies.

can be seen that actual performance of our communication optimization strategies are also in line with the design. Explanations are as follows. Without any optimization, "COMM" (the communication module of HCC-MF) will transmit both **P** and **Q** matrices (abbreviated as P&Q in the table). When compare the computing time cost in Figure 8 with the communication time cost in Table 5, the communication overhead is indeed much larger than computational cost, which confirms that the benefits of parallelism are canceled out. Using *"Transmitting* **Q** *matrix only"* (abbreviated as Q in the table), theoretically, the communication performance will be accelerated by $20(m+n)/(m+20n)$ times on the basis of P&Q. So the theoretical communication speedup of Netflix, R1_NEW and R2 are: 19.4, 2.5 and 6.1. The actual communication speedup are: 18.3, 2.9, 7.5, which are very close to theoretical values. When *"Transmitting FP16 Data"* is further executed based on Q (abbreviated as half-Q in the table), it can improve the communication performance by more than two times compared with only employing Q, exceeding the theoretical value. This phenomenon may be caused by the combination of memory access and more data being cached.

Besides, "COMM" is a good implementation in terms of communication between processors. To illustrate it, we have implemented a communication module based on ps-lite, named "COMM-P", which has same function with "COMM". And we tested the communication time of "COMM-P" under different strategies, as the same configuration the above experiment. Shown in Table 5, using the same optimization strategy, the communication performance of

Table 4: HCC-MF's "computing power" of 20-epoch training time (Updates/s)

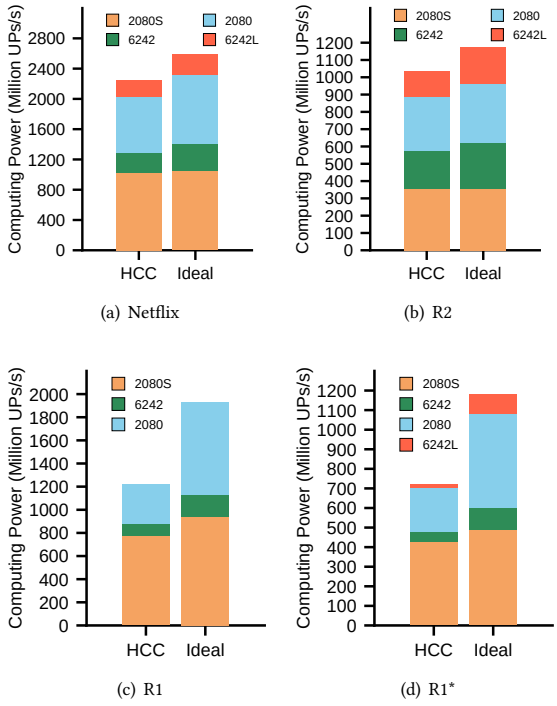
	6242-24T	6242-16T	2080	2080s	Ideal	HCC	utilization
Netflix	348790567	272502189.3	918333483.2	1052866849	2592493089	2228476993	86%
R1	190891071	191469060.9	801190194	939313585.8	2122863911	1310424456	62%
R2	266293289	212851540	339096219.3	354261902.7	1172502951	1034234303	88%
MovieLens-20m	261609815	250860330	835890148.7	905200490.3	2253560784	1025654359	46%

the "COMM" is significantly better than the "COMM-P". Because "COMM" can support point-to-point communication between workers without entering the kernel by using shared memory. More importantly, we ensure less temporary memory creation and release as well as fewer memory copies.

Finally, whether it is "COMM" or "COMMP", the same communication performance trend is reflected in each communication optimization strategy. It can explain that our communication strategy is stable.

4.5 Utilization under different system scales

By adding workers one by one, we test the "computing power" under different system scales to reflect HCC-MF's utilization of computing resources. Figure 9 is the stacked graph of "computing power" in HCC-MF. It can be found that the "computing power" always increases with the increase of workers. For Netflix and R2, which have low communication and synchronization costs, HCC-MF can use more than 80% of the "computing power" of each ordinary worker. The special worker that reuses the server's CPU can contribute more than 70% of the "computing power". For R1 and R1*, despite the substantial increase in communication overhead and synchronization overhead, each ordinary worker can still provide 45% of the "computing power" stably.



(a) Netflix (b) R2

(c) R1 (d) R1*

Figure 9: The result of increased "computing power" after adding heterogeneous processors in turn.

4.6 Limitations

HCC-MF has achieved good performance on some large-scale data sets, it still has some limitations. Through some experiments, we found that the HCC-MF cannot well accelerate the data set whose computing cost and communication cost are close, like MovieLens-20m. According to our design and the verification in Section 4.4, the communication overhead of HCC-MF is related to the scale of the rating matrix which is about $n/2(streams \times B_{buf})$. It will not decrease with the increase of workers. In this way, if the communication overhead is very close to the computational cost, by increasing the processor to reduce the computational cost, the overall overhead will not be reduced too much, like Table 6.

5 RELATED WORK

There are many parallel solutions for SGD-based MF, which can be mainly divided into distributed solutions and local solutions. i) *In distributed solutions*, DSGD[7] and DSGD++[23] divide the data by rows and synchronizes the feature matrix after each iteration. Their workflow is very consistent with the form of MapReduce and parameter server. We also adopt this type of workflow in HCC-MF. But DSGD and DSGD++ equally divide the input data into rows, which does not consider the difference in machine performance. It leads to the problem that the high-performance machines being blocked by the low-performance machines in a heterogeneous system. NOMAD[29] is an asynchronous and lock-free distributed framework for SGD-based MF. However, its lock-free mechanism is completely supported by the transmission of parameter messages. Differently, in NOMAD, a worker who finishes processing column will pass the column to other workers that will bring huge communication overhead. More importantly, even though the initial state of the worker is to process the data in the diagonal area, if the distribution of the elements in rating matrix is not balanced, the data race will occur. This kind of asynchronous and lock-free hides the risk of calculation errors on some data sets. ii) *In local solutions*, FPSGD and CuMF_SGD are two advanced solutions in shared memory system. They all try to make threads process independent data blocks to avoid synchronization. In addition, CuMF_SGD design cooperates with warp threads and coalesced memory access to improve GPU performance. However, it must use global locks to effectively protect the independence of the data block.

Table 5: The communication time of 20 epochs

		Netflix		R1_NEW		R2	
	Optimization	Cost time (s)	Speedup	Cost time (s)	Speedup	Cost time (s)	Speedup
COMM	P&Q	3.289744	1x	19.569929	1x	7.0763885	1x
	Q	0.180084684	18.3x	6.729931	2.9x	0.9467911	7.5x
	half-Q	0.056680425	58x	2.04014235	9.6x	0.31296455	22.6x
COMM-P	P&Q	21.8169325	1x	140.821585	1x	51.00871	1x
	Q	1.461305316	14.9x	50.57931	2.8x	7.190965	7.1x
	half-Q	0.53061025	41.1x	24.5123435	5.7x	4.039398	12.6x

Table 6: Limitation shown with MovieLens-20m

	worker	pull	computing	push	cost
HCC	2080S	0.088	0.368	0.103	0.559
	2080S-2080	0.097	0.2	0.132	0.449
		0.097	0.198	0.124	0.449
CuMF_SGD	2080S	N/A	N/A	N/A	0.559

6 CONCLUSION

We designed a multi-CPU/GPU collaborative computing framework HCC-MF for SGD-based MF with "asynchronous + synchronous" working mode. We build time cost model of collaborative computing workflow verified the effectiveness of the model through experiments on actual system. According to the model, we propose corresponding optimization strategies for the load balancing, synchronization, and communication problems it faces. Experiments show, the framework can effectively use the multi-CPUs/GPUs in the system for acceleration. HCC-MF can use up to 88% of the "computing power" of the entire platform to obtain 2.9x the acceleration of a single GPU on larger data sets. HCC-MF still has limitations in communication, which decreases its acceleration effect on some data sets close to the square matrix. We will try to solve this problem in the future.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (Grant 61872135, 61932010) and Zhejiang Lab (No. 2020KC0AC01), and Xiangjiang Artificial Intelligence Academy (Grant 202021B02).

REFERENCES

[1] AMD. 2021. AMD ROCm™ Open Ecosystem. Website. https://www.amd.com/en/graphics/servers-solutions-rocm.

[2] Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. 2015. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Transactions on Intelligent Systems and Technology (TIST)* 6, 1 (2015), 1–24.

[3] Intel Coorporation. 2013. An Introduction to the Intel® QuickPath Interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[4] Intel Coorporation. 2017. Intel® Performance Counter Monitor - A Better Way to Measure CPU Utilization. https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html.

[5] Intel Coorporation. 2019. SECOND GENERATION Intel® Xeon® Scalable Processors. https://www.intel.com/content/www/us/en/products/docs/processors/xeon/2nd-gen-xeon-scalable-datasheet-vol-1.html.

[6] NVIDIA Coorporation. 2020. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[7] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. Association for Computing Machinery, San Diego, California, USA, 69–77.

[8] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX, Boston, MA, USA, 485–500.

[9] Longke Hu, Aixin Sun, and Yong Liu. 2014. Your Neighbors Affect Your Ratings: On Geographical Neighborhood Influence to Rating Prediction. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval* (Gold Coast, Queensland, Australia) *(SIGIR '14)*. Association for Computing Machinery, New York, NY, USA, 345–354. https://doi.org/10.1145/2600428.2609593

[10] Intel. 2021. Intel® oneAPI Programming Guide. Website. https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html.

[11] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX, Renton, WA, USA, 947–960.

[12] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX, virtually, 463–479.

[13] Jing Jin, Siyan Lai, Su Hu, Jing Lin, and Xiaola Lin. 2016. GPUSGD: A GPU-accelerated stochastic gradient descent algorithm for matrix factorization. *Concurrency and Computation: Practice and Experience* 28, 14 (2016), 3844–3865.

[14] Donghyun Kim, Chanyoung Park, Jinoh Oh, Sungyoung Lee, and Hwanjo Yu. 2016. Convolutional Matrix Factorization for Document Context-Aware Recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys '16)* (Boston, Massachusetts, USA) *(RecSys '16)*. Association for Computing Machinery, New York, NY, USA, 233–240. https://doi.org/10.1145/2959100.2959165

[15] Tung D Le, Taro Sekiyama, Yasushi Negishi, Haruki Imai, and Kiyokuni Kawachiya. 2018. Involving cpus into multi-gpu deep learning. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, New York, NY, USA, 56–67.

[16] Feng Li, Yunming Ye, Zhaoyang Tian, and Xiaofeng Zhang. 2019. CPU versus GPU: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms. *Neural Computing and Applications* 31, 8 (2019), 4353–4365.

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

[18] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. 2013. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, Vol. 6. Association for Computing Machinery, San Diego, California, USA, 2.

[19] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. 2011. *OpenCL programming guide*. Addison-Wesley Professional, UK.

[20] J Myeongjae, V Shivaram, P Amar, et al. 2018. Multi-Tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications.

[21] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. arXiv:arXiv:1106.5730

[22] NVIDIA. 2019. CUDA C++ Programming Guide. Website. https://docs.nvidia.com/cuda/archive/10.2/cuda-c-programming-guide/index.html.

[23] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. 2012. Distributed matrix completion. In *2012 IEEE 12th international conference on data mining*. IEEE, IEEE,

Brussels, Belgium Belgium, 655–664.

[24] TOP500.org. 2020. TOP500 Certificates for top ranking systems in the 55th List. Website. https://www.top500.org/news/top500-certificates-top-ranking-systems-55th-list/.

[25] Fei Wang, Xiaofeng Gao, Jun Ye, and Guihai Chen. 2018. Is-asgd: Accelerating asynchronous sgd using importance sampling. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, Eugene, OR, USA, 1–11.

[26] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. 2021. Elastic Deep Learning in Multi-Tenant GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* (2021), 1–1.

[27] Xiaolong Xie, Wei Tan, Liana L Fong, and Yun Liang. 2017. CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on GPUs. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, Washington, DC, USA, 79–92.

[28] Yuanhang Yu, Dong Wen, Ying Zhang, Xiaoyang Wang, Wenjie Zhang, and Xuemin Lin. 2020. Efficient Matrix Factorization on Heterogeneous CPU-GPU Systems. arXiv:arXiv:2006.15980

[29] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, S. V. N. Vishwanathan, and Inderjit Dhillon. 2013. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. arXiv:arXiv:1312.0193