CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs

Xiaolong Xie*
Center for Energy-efficient Computing and Applications,
EECS, Peking University, Beijing, China
xiexl_pku@pku.edu.cn

Wei Tan
IBM Thomas J. Watson Research Center
Yorktown Heights, NY, USA
wtan@us.ibm.com

Liana L. Fong
IBM Thomas J. Watson Research Center
Yorktown Heights, NY, USA
llfong@us.ibm.com

Yun Liang
Center for Energy-efficient Computing and Applications,
EECS, Peking University , Beijing, China
ericlyun@pku.edu.cn

ABSTRACT

Stochastic gradient descent (SGD) is widely used by many machine learning algorithms. It is efficient for big data applications due to its low algorithmic complexity. SGD is inherently serial and its parallelization is not trivial. How to parallelize SGD on many-core architectures (e.g. GPUs) for high efficiency is a big challenge.

In this paper, we present **cuMF_SGD**, a parallelized SGD solution for matrix factorization on GPUs. We first design high-performance GPU computation kernels that accelerate individual SGD updates by exploiting **model parallelism**. We then design efficient schemes that parallelize SGD updates by exploiting **data parallelism**. Finally, we scale cuMF_SGD to large data sets that cannot fit into one GPU's memory. Evaluations on three public data sets show that cuMF_SGD outperforms existing solutions, including a 64-node CPU system, by a large margin using only one GPU card.

CCS CONCEPTS

• **Computing methodologies** → **Factor analysis**;
• **Computer systems organization** → **Heterogeneous (hybrid) systems**;
• **Theory of computation** → *Massively parallel algorithms*;

KEYWORDS

Matrix Factorization, GPGPU, Parallel Computing.

ACM Reference format:
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 ACM Symposium on High-Performance Parallel and Distributed Computing, Washington , DC, USA, June 26-30, 2017 (HPDC '17),* 14 pages.
https://doi.org/http://dx.doi.org/10.1145/3078597.3078602

*Work done while the author was with IBM as an summer research intern.

HPDC '17, June 26-30, 2017, Washington , DC, USA

ACM ISBN 978-1-4503-4699-3/17/06…$$15.00
https://doi.org/http://dx.doi.org/10.1145/3078597.3078602

1 INTRODUCTION

Matrix Factorization (MF) is a popular algorithm that is widely used in modern machine learning applications, including collaborative filtering [12, 34], topic modeling [62], word embedding [43], and tensor decomposition [33]. It also has a natural connection with the embedding layers in deep neural network [43]. Without loss of generality, we use collaborative filtering as the example. Figure 1 shows a $m \times n$ rating matrix R which is sparse and with $N = 9$ observed samples. The goal of matrix factorization is to obtain two lower-rank feature matrices P ($m \times k$) and Q ($k \times n$) such that $R \approx P \times Q$. The feature vector dimension k is typically much smaller than m and n. Feature matrices P and Q can be used to predict the unknown samples in R, or as the features of the corresponding entities in subsequent machine learning tasks.

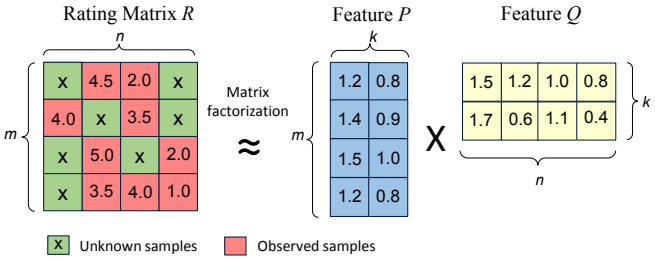


Figure 1: Matrix factorization, m=4, n=4, k=2.

Due to its algorithmic simplicity, Stochastic Gradient Descent (SGD) is often used in modern machine learning applications [17, 44]. Instead of calculating the gradient on the whole training set, SGD randomly picks up one sample from the training set and updates using the gradient on that particular sample. Such a simple approach has been demonstrated efficient in e.g., deep learning and matrix factorization [8, 65]. However, as SGD is inherently serial, how to parallelize it to exploit parallel processors becomes a major challenge [17]. Besides SGD, Coordinate Gradient Descent (CGD) [60] and Alternate Least Square (ALS) [34] are also used to solve MF problems. Previous works have demonstrated that CGD is prone to reach local optima [15]. ALS is inherently parallel and converges faster than SGD [51]. However, ALS is not efficient when processing large data sets due to its high algorithmic complexity. In this paper, we use SGD as the algorithm to solve MF and compare with ALS-based solutions.

CPU-based SGD solutions for MF, including both shared memory-based [16] and distributed system-based [63] have been studied. In contrast, we propose to use GPUs to accelerate SGD-based MF. Our insight is, **SGD-based MF is memory bound**. Previous works [15, 63] spend effort to improve the cache efficiency of CPUs. However, due to the limited cache capacity, shared memory-based solutions face serious performance degradation when processing large data sets. Distributed systems partition the data sets to fit into CPU caches, however, they are limited by the slow network. Therefore, we propose to use GPU to accelerate SGD-based MF as it provides large amount of computational resources [32], extreme high off-chip memory bandwidth [28, 30], and energy efficiency [49, 50].

Recent effort has been spent to use GPUs to accelerate the SGD-based MF. Kaleem et al. [31] evaluated different workload scheduling policies that parallelize SGD on GPUs. Jin et al. [27] propose an MF solution based on matrix blocking. Canny et al. [11] release BIDMach, a machine learning library that supports matrix factorization. However, to the best of our knowledge, none of them outperforms state-of-the-art CPU solutions. The main reason is, they simply port CPU-based algorithms to GPUs and do not fully exploit GPU architecture [59]. Unlike these efforts, our study is to fully explore and exploit the power of GPUs.

We develop **CuMF_SGD**[1], an SGD-based MF solution on GPUs, with the goal to scale to large data sets and to achieve near-linear scalability when increasing the level of parallelism. We first design high-performance GPU kernels to fully utilize the memory bandwidth on GPUs for individual SGD updates. We then evaluate the existing SGD parallelization policies, and design lightweight scheduling policies specifically for GPUs to minimize the scheduling overhead. Finally, we design workload partition schemes to accommodate large data sets and minimize the CPU-GPU communication overhead. We contribute to the state-of-the-art of machine learning and high-performance computing in the following aspects:

- We characterize the SGD-based MF problem and identify that it is bound by memory bandwidth. We propose an accelerated solution, cuMF_SGD, by exploiting GPUs and their extremely high off-chip memory bandwidth.
- We show how cuMF_SGD exploits *model parallelism* through high-performance GPU kernels, and *data parallelism* through lightweight scheduling schemes. In this way, cuMF_SGD is able to scale to large data sets that cannot fit into the device memory.
- We evaluate cuMF_SGD on two modern GPU generations with benchmark data sets. Evaluations illustrate that cuMF_SGD achieves good performance on all data sets on both platforms. Compared with previous works [16], cuMF_SGD achieves 3.1X to 28.1X performance improvements. Moreover, cuMF_SGD outperforms a 64-node CPU cluster with only one GPU card. CuMF_SGD is also able to scale to multiple GPUs and different generations of GPUs.

We organize the remainder of the paper as follows. Section 2 present the baseline SGD algorithm, GPU architectural details, and workload characterization. Section 3 shows the overview of cuMF_SGD. Section 4 presents the GPU kernel design, Section 5 shows the SGD scheduling algorithms, and Section 6 shows how cuMF_SGD scales to large data sets. Section 7 presents the experiment results and analysis, Section 8 discusses the related work and Section 9 concludes this paper.

[1] http://github.com/cumf/cumf_sgd/

2 BACKGROUND

We first present the basics of SGD-based matrix factorization in Section 2.1. Then Section 2.2 presents the details of experimental setup and Section 2.3 characterizes the workload.

2.1 SGD-based Matrix Factorization

Given a sparse $m \times n$ matrix R, the goal of matrix factorization is to train a $m \times k$ dense feature matrix P and a $k \times n$ dense feature matrix Q such that:

$$\mathbf{R} \approx \mathbf{P} \times \mathbf{Q} \tag{1}$$

We use $r_{u,v}$ to refer to the sample at the u_{th} row and the v_{th} column of R. We use p_u to represent the u_{th} row of P and q_v to represent the v_{th} column of Q. In a recommender system, $r_{u,v}$ indicates the preference or rating of u_{th} user on v_{th} item, p_u is used to represent the preference of the u_{th} user and q_v is used to represent the feature of the v_{th} item. The training process of matrix factorization is to minimize the root mean square error (RMSE) between the original matrix and trained model:

$$\sum_{r_{u,v} \in R}^{N} (r_{u,v} - \mathbf{p}_u \mathbf{q}_v)^2 + \lambda_p \mid\mid \mathbf{p}_u \mid\mid^2 + \lambda_q \mid\mid \mathbf{q}_v \mid\mid^2 \tag{2}$$

where λ_p and λ_q are regularization parameters to avoid overfitting and N is the number of non-zero samples in matrix R. Such a simple model and its variants are now widely used in recommender systems [34], topic modeling [62], word embeddings [43], and others.

To solve Eq.2, it is necessary to go through $R_{m \times n}$. Consider $R_{m \times n}$ may contain up to billions of samples, the process is time-consuming. To address the problem, *stochastic gradient descent* (SGD), is often employed in modern machine learning applications [17]. Instead of going through rating matrix $R_{m \times n}$, SGD only randomly picks one sample at each step and Eq.2 is reduced to:

$$(r_{u,v} - \mathbf{p}_u \mathbf{q}_v)^2 + \lambda_p \mid\mid \mathbf{p}_u \mid\mid^2 + \lambda_q \mid\mid \mathbf{q}_v \mid\mid^2 \tag{3}$$

Algorithm 1 shows a typical SGD-based matrix factorization algorithm. The input of the algorithm includes the rating matrix $R_{m \times n}$, the feature dimension k (typically ranges from $O(10)$ to $O(100)$), the learning rate γ, and the regularization parameter λ. In this paper, we use the same λ for both P and Q. The output is the trained model (i.e., feature matrices P and Q). Lines 1-3 show the data pre-processing, lines 5-12 show the SGD training process, and lines 14-16 show the data post-processing. The SGD training process is the most time-consuming part and not different with previous work, and we focus on optimizing this part. The training process is composed of two loops. Each iteration (also known as an *epoch*) of the outer loop represent a full pass of the rating matrix. The number of iterations of the outer loop is set by users. In each step of the inner loop, one sample is randomly picked to decrease Eq.3 (details shown in Line 5-12). The SGD training is finished when the given number of iterations (Ite) is reached or the model converges.

The optimization work on matrix factorization contains two streams: *algorithm* and *system*. The algorithmic stream tries to optimize update schemes such as learning rate (γ) in gradient descent, in order to **reduce the number of epochs (iterations)** needed to converge [16]. The system stream tries to accelerate the computation, in order to **run each epoch faster** [15, 51, 61, 63]. We focus on the system stream and the proposed techniques can be combined with other algorithmic optimizations.

Algorithm 1 A Typical SGD-based Matrix Factorization Algorithm.

Input: $R_{m\times n}$ (N samples), k (feature dimension), γ (learning rate), λ (regularization parameter);

1: ▷ Data pre-processing.
2: $random\_shuffle(\mathbf{R}_{m\times n})$;
3: $\mathbf{P}_{m\times k}, \mathbf{Q}_{k\times n} \leftarrow random(0, sqrt(1/(k * scale\_factor))$;
4:
5: ▷ SGD training
6: **for** $iteration \leftarrow 1$ to Ite **do**
7: **for** Randomly select $r_{u,v}$ from $\mathbf{R}_{m\times n}$ **do**
8: $error \leftarrow r_{u,v} - \mathbf{p}_u \times \mathbf{q}_v$;
9: $\mathbf{p}_u \leftarrow \mathbf{p}_u + \gamma(error * \mathbf{q}_v^T - \lambda * \mathbf{p}_u)$
10: $\mathbf{q}_v \leftarrow \mathbf{q}_v + \gamma(error * \mathbf{p}_u^T - \lambda * \mathbf{q}_v)$
11: **end for**
12: **end for**
13:
14: ▷ Data post-processing.
15: $model\_shuffle(\mathbf{P}, \mathbf{Q})$;
16: $model\_save(\mathbf{P}, \mathbf{Q})$;

Output: $\leftarrow \mathbf{P}$,$\mathbf{Q}$

As we see in Algorithm 1, the SGD training process is serial. One SGD update contains a dot product (Line 8) and a few vector operations (Line 9, 10) at length k. How to efficiently execute individual SGD updates, i.e., exploit the *model parallelism*, becomes one major design factor of matrix factorization. As one SGD update can not saturate the resources on modern processors, how to parallelize the SGD updates is also a challenge (*data parallelism*). Our proposed cuMF_SGD exploits both model parallelism and data parallelism.

2.2 Experimental Setup

Platform. Table 1 shows the configurations of the platforms used in this paper. The Maxwell platform is with Intel Xeon CPUs and NVIDIA TITAN X GPUs. The TITAN X GPU is Maxwell architecture [3] with CPUs and GPUs connected via PCIe 3.0. The Pascal platform is equipped with IBM PowerNV8 CPUs and NVIDIA Pascal P100 GPUs. The P100 GPU is Pascal architecture [5], newer than Maxwell architecture. The CPUs and GPUs are connected by NVLink that is much faster than PCIe. Overall, the Pascal platform is more powerful than the Maxwell platform in terms of both computational power and interconnection bandwidth.

Data sets. We use three public data sets: *Netflix*, *Yahoo!Music*, and *Hugewiki*. Details of them are shown in Table 2. They are also used in other matrix factorization systems [16, 38]. *Netflix* and *Yahoo!Music* comes with a test set but *Hugewiki* does not. For *Hugewiki*, we randomly sample and extract out 1% of the data as the test set.

Table 1: Configuration of the Maxwell [3] and Pascal [5] Platform.

Maxwell Platform	
CPU	12-core Intel Xeon CPU E5-2670*2 (up to 48 threads), 512 GB memory.
GPU	TITAN X GPU*4, per GPU: 24 SMs, 12 GB device memory, 360GB/s memory bandwidth, per SM: 128 CUDA cores, 2K threads.
Inter-connection	PCIe 3.0, up to 16 GB/s.
Pascal Platform	
CPU	2*10 PowerNV 8 processors with SMT 8 and NVLink, 512 GB memory.
GPU	Pascal P100 GPU*4, per GPU: 56 SMs, 16 GB device memory, 780GB/s memory bandwidth, per SM: 64 CUDA cores, 2K threads.
Inter-connection	NVLink, up to 80 GB/s.

Table 2: Details of workload data sets.

Dataset	*Netflix*	*Yahoo!Music*	*Hugewiki*
m	480,190	1,000,990	50,082,604
n	17,771	624,961	39,781
k	128	128	128
$Train\ Set$	99,072,112	252,800,275	3,069,817,980
$Test\ Set$	1,408,395	4,003,960	31,327,899

2.3 Workload Characterization

Before optimizing the SGD-based MF, we first characterize the workload features. We start with analyzing the computation to memory ratio. We adopt a metric, $Flops/Byte$, which is defined as the ratio of floating point operations to memory access density (byte) (Eq. 4). When $Flops/Byte$ is extremely high (e.g. in matrix multiplication), the application is *compute bound*, otherwise, the application is *memory bound*.

$$Flops/Byte = \frac{\#FloatingPointOps}{\#MemoryOps(Byte)} \tag{4}$$

Eq. 5 shows how to compute the $Flops/Byte$ metric for SGD-based MF. Consider that, for $k = 128$ and $sizeof(r_{u,v}) = 12$ (2 integers and 1 float in COO format), the $Flops/Byte$ is 0.43 ops/byte. Given the fact that a modern CPU processor provides ~600 GFLOPS computational horsepower and ~60 GB/s off-chip memory bandwidth (600/60=10), SGD-based MF has low $Flops/Byte$ ratio and is bound by memory.

$$Flops/Byte = \frac{6k + \sum_{i=1}^{logk} \frac{k}{2^i}}{sizeof(r_{u,v}) + 4k * sizeof(float)} \tag{5}$$

For shared memory based solutions, to address the memory problem, cache efficiency has to be carefully optimized. Chin et al. release LIBMF [15, 16], a CPU-based high-performance SGD solution to MF. As reported in the original paper, they optimize the cache efficiency at the expense of sacrificing randomness. We evaluate LIBMF on our Maxwell platform and show the effective memory bandwidth in Figure 2(a). On the smallest *Netflix* data set, LIBMF achieves 194GB/s effective memory bandwidth. The bandwidth is much higher than the theoretical memory bandwidth due to cache effect. However, on the largest *HugeWiki* data set, the effective memory bandwidth drops by 45%, to 106GB/s. The main reason is that increased data size leads to decreased cache efficiency, and thus degrades the performance. Hence, single-node CPU solution is not scalable to large data sets.

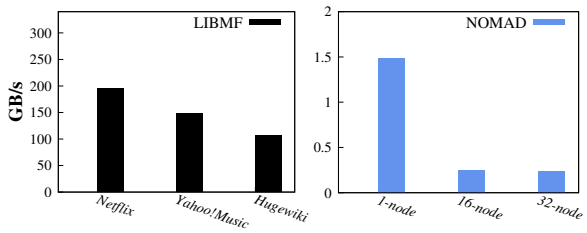


Figure 2: (a) The effective memory bandwidth of LIBMF drops when solving large data sets. (b) NOMAD achieves lower memory efficiency when scaling to multiple nodes.

One solution to the cache performance degradation is to distribute the workload to multiple nodes. Yun et al. develop *NOMAD*, a distributed SGD-based MF solution [63]. By distributing the data to multiple nodes, the working set can fit into to L3 cache of CPUs. Hence, the cache efficiency is improved. However, the overall performance is bound by the slow network speed [47]. On the *Netflix* data set, NOMAD only achieves ~5.6X speedup when scaling from 1 node to 32, which is far from perfect scaling. Figure 2 (b) shows the achieved memory efficiency of NOMAD on *Netflix* data with different number of nodes. The memory efficiency is the ratio of effective memory bandwidth to total memory bandwidth of all nodes. We observe that the efficiency of distributed solution is extremely low.

In this paper, we propose to accelerate SGD-based MF on GPUs. One main aspect is to maximize the usage of GPUs' high off-chip memory bandwidth. Compared with CPUs, GPUs do not rely on the cache and are scalable to process large data sets. For example, NVIDIA Maxwell architecture GPU TITAN X is equipped with up to 360GB/s memory bandwidth, which is multiple times higher than CPUs (<100 GB/s). New generation NVIDIA P100 GPUs provide even higher bandwidth (780 GB/s). Moreover, NVIDIA GPUs introduce NVLink as new generation interconnect network between CPUs and GPUs. NVLink provides up to 80 GB/s memory bandwidth. For data sets that can not fit into GPU's memory, the high-speed CPU to GPU bandwidth makes GPUs more efficient than distributed systems.

3 OVERVIEW

In this section, we present the overview of cuMF_SGD, as shown in Figure 3. Parallelizing SGD on GPUs is a not a trivial task. SGD is inherently serial, which does not fit the flavor of GPUs [17]. Many machine learning solutions [37] employs batch SGD on GPUs to exploit model parallelism. They process a batch of, say, 16 to 256, samples to saturate GPUs. Given that the each individual SGD update in MF is lightweight, thousands of SGD updates are required to saturate GPUs. However, increasing the size of each batch may hurt convergence and in the end prolong the training time. Hence, we carefully design cuMF_SGD to fully utilize the resources on GPUs.

The design of cuMF_SGD is composed of two streams. The first stream is to exploit the *model parallelism*. We design high-performance GPU kernels to optimize the execution of each individual SGD update. In this paper, we term a group of threads that

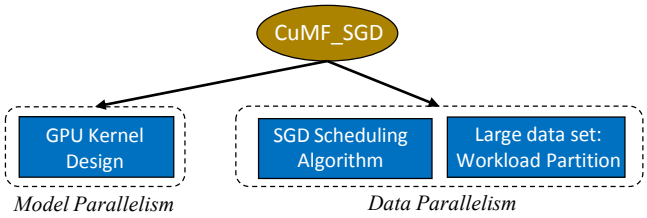


Figure 3: Overview of cuMF_SGD.

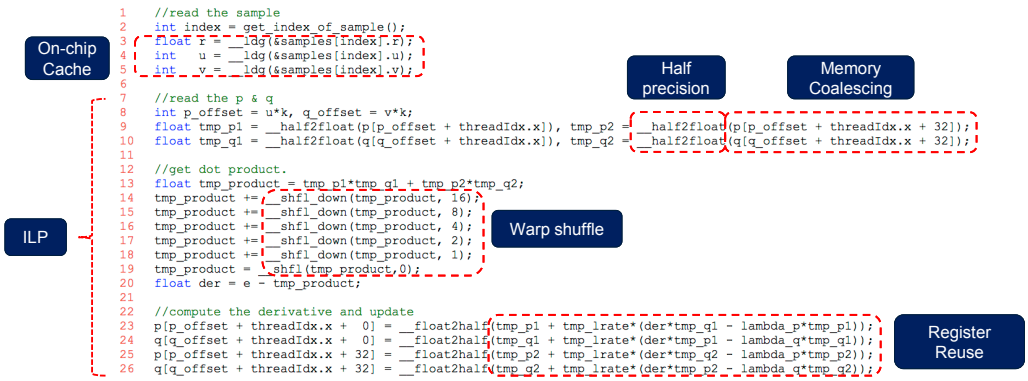
work coordinately to perform one SGD update as a *parallel worker*. On CPUs, a parallel worker is usually composed of one thread. On GPUs, we set a parallel worker as one thread block to exploit the SIMD features. We optimize the kernel using various GPU optimization techniques, including warp shuffle, memory coalescing, and on-chip caching. Details are shown in Section 4. The other stream is to exploit the *data parallelism*. There are hundreds of parallel workers (thread blocks) running on one GPU, SGD updates are scheduled and executed in parallel. How to design efficient scheduling algorithm that incurs minimal scheduling overhead becomes a key design factor. We design lightweight SGD scheduling algorithms that are effective in terms of both system throughput and convergence, with details discussed in Section 5. The GPU memory capacity is limited (~10GB per GPU). When processing large data sets, the data set has to be partitioned to fit into the GPU's memory. The memory transfer between CPU and GPU happens frequently to migrate the data. We design workload partitioning algorithm and optimize the CPU-GPU memory transfer to maximize the system performance. The algorithm is presented in Section 6.

4 GPU KERNEL DESIGN

In this section, we present how to design the GPU kernel of cuMF_SGD. When designing the kernel, we assume that the SGD workloads have been assigned to parallel workers. Hence, we only focus on how to efficiently execute the SGD updates within each parallel worker.

In MF, one SGD update consists of four steps: 1) read one sample ($r_{u,v}$) from the rating matrix, 2) read two feature vectors (p_u, q_v), 3) compute prediction error($r_{u,v} - p_u q_v$), and 4) update the features. Except for the first step, other three steps are all vector operations at length k. k is the feature dimension and typically ranges from $O(10)$ to $O(100)$. On a CPU, a parallel worker can be one or more threads of a process, where vector instructions such as SSE and AVX can be used to accelerate the computation. GPUs are SIMD architectures [48], where a thread block is a vector group. Hence, in cuMF_SGD, we use a thread block as a parallel worker. Figure 4 shows a code snippet of the computational part of cuMF_SGD, where we use $k = 64$ as an example. We highlight the major optimization techniques in Figure 4 and explain them in the following.

Warp shuffle. Warp shuffle instructions [19] are used to compute the dot product $p_u \times q_v$ and broadcast the result. Compared with traditional shared memory system-based approaches, this warp shuffle-based approach performs better because: (1) warp shuffle instructions have extra hardware support, (2) register is faster than shared memory, and (3) no thread synchronization is




```
1   //read the sample
2   int index = get_index_of_sample();
3   float r = __ldg(&samples[index].r);
4   int   u = __ldg(&samples[index].u);
5   int   v = __ldg(&samples[index].v);
6
7   //read the p & q
8   int p_offset = u*k, q_offset = v*k;
9   float tmp_p1 = __half2float(p[p_offset + threadIdx.x]), tmp_p2 = __half2float(p[p_offset + threadIdx.x + 32]);
10  float tmp_q1 = __half2float(q[q_offset + threadIdx.x]), tmp_q2 = __half2float(q[q_offset + threadIdx.x + 32]);
11
12  //get dot product.
13  float tmp_product = tmp_p1*tmp_q1 + tmp_p2*tmp_q2;
14  tmp_product += __shfl_down(tmp_product, 16);
15  tmp_product += __shfl_down(tmp_product, 8);
16  tmp_product += __shfl_down(tmp_product, 4);
17  tmp_product += __shfl_down(tmp_product, 2);
18  tmp_product += __shfl_down(tmp_product, 1);
19  tmp_product = __shfl(tmp_product,0);
20  float der = e - tmp_product;
21
22  //compute the derivative and update
23  p[p_offset + threadIdx.x +  0] = __float2half(tmp_p1 + tmp_lrate*(der*tmp_q1 - lambda_p*tmp_p1));
24  q[q_offset + threadIdx.x +  0] = __float2half(tmp_q1 + tmp_lrate*(der*tmp_p1 - lambda_q*tmp_q1));
25  p[p_offset + threadIdx.x + 32] = __float2half(tmp_p2 + tmp_lrate*(der*tmp_q2 - lambda_p*tmp_p2));
26  q[q_offset + threadIdx.x + 32] = __float2half(tmp_q2 + tmp_lrate*(der*tmp_p2 - lambda_q*tmp_q2));
```

Figure 4: The exemplify kernel code of cuMF_SGD, where $k = 64$. The used optimization techniques are highlighted.

involved. To exploit the warp shuffle feature, we fix the thread blocks size as warp size(32).

On-chip cache. Since Fermi architecture, NVIDIA GPUs feature on-chip L1 cache and allow programmers to control the cache behavior of each memory instruction (cache or bypass). While many GPU applications do not benefit from the cache due to cache contention [36, 56, 57], some memory instructions may benefit from the cache as the accessed data may be frequently reused in the near future (temporal reuse) or by other threads (spatial reuse). Following the model provided by [56], we observe that the memory load of the rating matrix benefits from cache and use the intrinsic instruction *__ldg* [2] to enable cache-assisted read.

Memory coalescing. On GPUs, when threads within one warp access the data within one cache line, the access is coalesced to minimize the bandwidth consumption [29, 59]. This is called memory coalescing. In cuMF_SGD, the read/write of P and Q are carefully coalesced to ensure that consecutive threads access consecutive memory addresses.

ILP. Modern GPUs support compiler-aided super scalar to exploit the instruction-level parallelism (ILP). In cuMF_SGD, when $k > 32$, a thread is responsible for processing $k/32$ independent scalars. Hence, with awareness of the low-level architecture information, we reorder the instructions to maximize the benefit of ILP.

Register usage. The register file is an important resource on GPUs. As the total number of registers on GPUs is fixed, if each thread uses too many registers, the register consumption may become the limitation to concurrency. In our case, the CUDA compiler reports that allocating 33 registers for each thread is enough to fit all active variables. The concurrency is only limited by the number of thread blocks of GPUs [2, 55]. Hence, we allocate as many as possible registers to each thread such that every reusable variable is kept in the fastest register file.

Half-precision. As addressed before, SGD is memory bound. Most of the memory bandwidth is spent on the read/write to the feature matrices. Recently, GPU architectures support the storage of half-precision (2 bytes vs. 4 bytes of single-precision) and fast transformation between floating point and half-precision. In practice, after parameter scaling, half-precision is precise enough to store the feature matrices and does not incur accuracy loss. CuMF_SGD uses half-precision to store feature matrices, which halves the memory bandwidth need when accessing feature matrices.

5 WORKLOAD SCHEDULING ALGORITHM

The original SGD algorithm is serial, with samples in the rating matrix picked up randomly and updated in sequence. To exploit the data parallelism and execute SGD updates in parallel, a workload scheduling algorithm that assigns tasks to parallel workers becomes necessary. The parallelization of SGD updates for MF is based on the observation that in Algorithm 1, one SGD update on sample $r_{u,v}$ only updates the u^{th} row of $P(P_u)$ and v^{th} row of $Q(Q_u)$. Consider two samples, $r_{u1,v1}$ and $r_{u2,v2}$, they can be updated simultaneously if

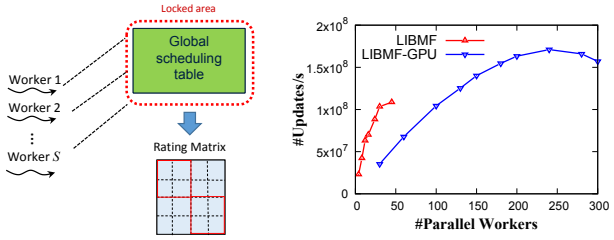
$$u1 \neq u2 \;\&\&\; v1 \neq v2 \tag{6}$$

We term them as *independent* updates. We term simultaneous dependent updates as *conflicts*. To evaluate the efficiency of workload scheduling algorithm, we define a metric, *updates per second* as the performance indicator:

$$\#Updates/s = \frac{\#Iterations \times N}{Elapsed\ Time} \tag{7}$$

where $\#Iterations$, N, $Elapsed\ Time$ are the number of iterations, the number of non-zero samples in the input matrix R, and the elapsed time in seconds, respectively. As we discussed before, the SGD-based MF is memory bound. According to the roofline model [54], the application is limited by the memory bandwidth. Hence, the design goal of workload scheduling algorithm is to minimize the scheduling overhead and exhaust the memory bandwidth on GPUs.

We start our scheduling algorithm design from examining existing solutions. We analyze and evaluate the workload scheduling algorithm proposed in CPU-based LIBMF [15], as it is publicly available. Figure 5(a) shows the basics of scheduling algorithm used in LIBMF. It evenly divides the rating matrix into $a \times a$ blocks and uses s threads ($s < a$). It also uses a table to manage the matrix blocks to comply to Eq. 6. When a thread is idle, it accesses the table and finds an independent block. Then, the thread executes the SGD updates in the block in serial. Overall, it requires atomic operations to manage the table and $O(a^2)$ time complexity to search the table.



(a) Idea of LIBMF. *(b) Scalability Study.*

Figure 5: (a) LIBMF uses a centralized table to manage parallel workers. (b) LIBMF scales to only 30 CPU threads and 240 GPU thread blocks.

We evaluate LIBMF on the Maxwell platform using the *Netflix* data set. Figure 5(b) shows the evaluation results that the performance of LIBMF saturates around 30 concurrent workers (CPU threads), which is consistent with the previous study [41].

As GPUs are more sensitive to synchronization overhead than CPUs [9], we optimize the scheduling algorithm by reducing its time complexity. In each scheduling step, we first search all a columns and a rows to find the independent rows or columns. Then we randomly choose one independent block that is in the independent rows and columns. By doing so, the time complexity of the scheduling algorithm is reduced to $O(a)$. We combine the optimized scheduling algorithm with our designed GPU kernel(Section 4). However, as shown in Figure 5(b) (labled as LIBMF-GPU), it can only scale to 240 thread blocks, much lower than the hardware limit (768 thread blocks on Maxwell).

The reason why LIBMF cannot scale to many parallel workers is that it uses a global scheduling table to manage all parallel workers. At each time, only one parallel worker can access the table and it is also time-consuming to find a free block to process. Therefore, when the number of workers increases, the waiting time also increases. As the number of workers grows, the waiting time becomes dominating. This shows that cuMF_SGD can not simply re-use existing scheduling policies. To overcome the scheduling overhead, we propose two GPU-specific scheduling schemes, *batch-Hogwild!* and *Wavefront-update*. Batch-Hogwild! avoids matrix blocking-based scheduling and improves the cache efficiency by process samples in batch. Wavefront-update is still blocking-based, but only requires a local look-up instead of the expensive global lookup as in LIBMF.

5.1 Batch-Hogwild!

We propose batch-Hogwild!, a variant of Hogwild! [44] with improved cache efficiency. Hogwild! is efficient as its lock-free scheme incurs negligible scheduling overhead. It is not efficient, however, in terms of data locality [15]. In Hogwild!, each parallel worker randomly selects one sample from the rating matrix at each step. After each update, Hogwild! may not access the consecutive samples in the rating matrix and corresponding rows and columns in the feature matrices for a long time interval, leading to low cache efficiency. As shown in Section 4, we carefully align the memory access of the feature matrices to achieve perfect memory coalescing and the high memory bandwidth on GPUs, such that eliminating accessing feature matrices as a performance bottleneck. To accelerate the access to rating matrix, we exploit the spatial data locality using L1 data cache. We let each parallel worker, instead of fetching one sample randomly at a time, fetch f consecutive samples and update them serially. The data locality is fully exploited when the following constraint is met,

$$f \gg \lceil \frac{CacheLineSize}{sizeof(r_{u,v})} \rceil \quad (8)$$

Note that these samples are consecutive in their memory storage; because we shuffle samples, they are still random in terms of their coordinates in R. By doing so, the data locality is fully exploited. Consider the L1 cache line size is 128 bytes and the size of each sample is 12 bytes (one floating point and two integers), $f \gg \lceil 128/12 \rceil$ is enough to exploit the locality. We evaluate different values of f and find that they yield similar benefit. Therefore we choose $f = 256$ without loss of generality.

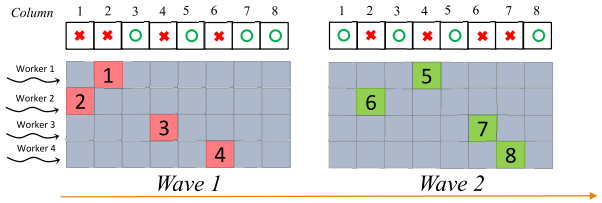


Figure 6: Wavefront-update. Each parallel worker is assigned to a row and a randomized column update sequence. For example, when Worker 3 completes Block 3 in Wave 1, it releases Column 4 such that Worker 1 can start Block 5 in Wave 2.

5.2 Wavefront-update

As previously discussed, existing scheduling schemes [15, 23] impose a global synchronization, where all workers look up a global table to **find both row and column coordinates** to update. This is expensive and has been shown not scalable to the hundreds of workers on GPUs. To overcome this, we propose wavefront-update, a lightweight scheme that **locks and looks up columns only**.

In the exemplary Figure 6, we use four parallel workers to process an R which is partitioned into 4×8 blocks. Each worker is assigned to a row in this 4×8 grid, and each generates a permutation of $\{1, 2, 3, \ldots, 7, 8\}$ as its column update sequence. By this, an epoch is conducted in eight waves given this sequence. In each wave, one worker update one block, and workers do not update blocks in the same column. Assume *Worker 1* has the sequence defined as $\{2, 4, \ldots\}$ and *Worker 3* has sequence $\{4, 6, \ldots\}$. With this sequence, *Worker 1* updates *Block 1* in wave 1 and *Block 5* in wave 2. To avoid conflicts, we propose a lightweight synchronization scheme between waves using the column lock array. As shown in the figure, we use an array to indicate the status of each column. Before a worker moves to next wave, it checks the status of the next column defined in its sequence. For example, after *Worker 1* finishes *Block 1*, it needs to check the status of column 4 and does not need to care about the status of other columns. When *Worker 3* finishes *Block 3* and releases column 4, *Worker 1* is allowed to move to wave

2. There are two main benefits by doing so: firstly to reduce the two-dimension look-up table in [15, 23] to a one-dimension array; secondly to minimize the workload imbalance problem, as a worker can start the next block earlier without waiting for all other workers to finish.

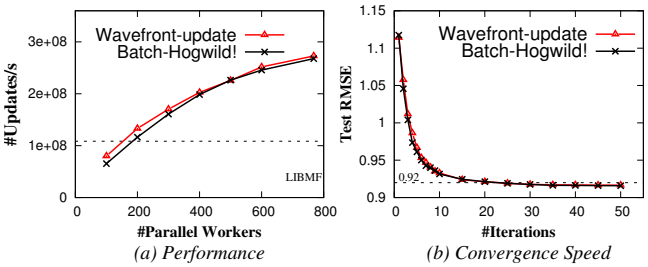


Figure 7: Performance Comparison of *batch-Hogwild!* and *wavefront-update* on *Netflix* data set. Both schemes scale much better than LIBMF.

5.3 Evaluation of scheduling schemes

We evaluate both scheduling schemes in terms of performance and convergence speed using the *Netflix* data set on the Maxwell platform. We use metric $\#Updates/s$ to quantify the performance. Figure 7(a) shows the scalability of *batch-Hogwild!* and *wavefront-update* with a different number of parallel workers (i.e., thread blocks). When increasing the number of parallel workers, both schemes achieve near-linear scalability. When the number of parallel workers hits the hardware limit of 768 on the Maxwell GPU, both techniques achieve ~0.27 billion updates per second, the rate which is 2.5 times faster than LIBMF. Therefore, we conclude that our proposed schemes can perfectly solve the scalability problem of the scheduling policy and fully exploit the equipped hardware resources on GPUs. We also evaluate the convergence speed of both schemes. We use the *root mean square root* error on the standard test data set (*Test RMSE*) as the indication of convergence. Figure 7(b) shows the decrease of *Test RMSE* as the number of iterations increases. Overall, batch-Hogwild! converges a little bit faster than wavefront-update. The reason is that batch-Hogwild! enforces more randomness in update sequence, as compared with the block-based wavefront-update. Based on this observation, we use batch-Hogwild! as the default scheme on one GPU experiments.

6 WORKLOAD PARTITION

In the previous discussions, we assume that the rating matrix and feature matrices fully reside in GPU's memory. However, the limited GPU memory capacity [45] prevents cuMF_SGD from solving large-scale problems. For example, NVIDIA TITAN X GPU has 12 GB device memory that can only store 1 billion samples (one sample needs one float and two integers). Nowadays, real-world problems may have 10^{11} samples [51]. Techniques such as *Unified Virtual Memory* [2] allow GPU to access CPU's memory but with high overhead. Consider these factors, to solve large-scale MF problems that can not fit into one GPU's memory, we need to partition the data sets and stage the partitions to GPUs in batches. Moreover, we should overlap the data transfer with computation to alleviate the delay caused by slow CPU-GPU memory transfer. Please note that the partitions can be processed by one or multiple GPUs.

6.1 Workload Partitioning

Figure 8 shows our proposed multi-GPU solution for large data sets. The main idea is to partition the rating matrix R into multiple blocks; each block is small enough to fit into one GPU's memory such that independent blocks can be updated concurrently on different GPUs. The multi-GPU solution works as follows,

(1) Divide the rating matrix R into $i \times j$ blocks. Meanwhile, divide feature matrix p into i segments and feature matrix q into j segments accordingly.
(2) When a GPU is idle, randomly select one matrix block from those independent blocks and dispatch it to the GPU.
(3) Transfer the matrix block and corresponding feature sub-matrices p and q to the GPU. Then update the matrix block using the single GPU implementation discussed in Section 5. After the update, transfer p and q back to CPU.
(4) Iterate from 2 until convergence or the given number of iterations is reached.

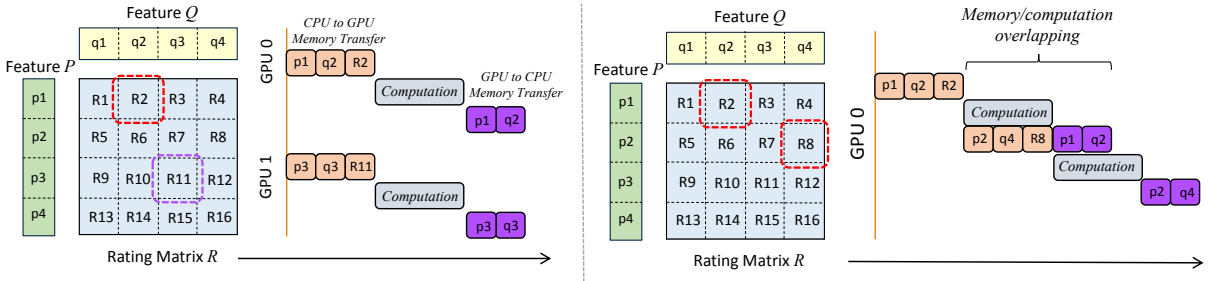
We further explain the proposed scheme using the example shown in Figure 8(a). In Step 1, we divide R into 4×4 blocks and use two GPUs. In Step 2, we send block $R2$ to GPU 0 and $R11$ to GPU 1. Again, consider the nature of MF, updating $R2$ only touches sub-matrices $p1$ & $q2$ while updating $R11$ only touches $p3$ & $q3$. Hence, GPU 0 only needs to store $R2$, $p1$, and $q2$ in its device memory while GPU 1 only needs to store $R11$, $p3$, and $q3$. By doing so, the problem is divided and conquered by multiple GPUs. After deciding the block scheduling order, cuMF_SGD transfers $p1$, $q2$, $R2$ to GPU 0 and $p3$, $q3$, $R11$ to GPU 1. Then cuMF_SGD performs the computation on two GPUs in parallel. The GPU-side computation follows the rules we discussed in Section 5. After finishing the computation, the updated $p1$, $q2$, $p3$, and $q3$ are transferred back to CPU memory. Note that we don't have to transfer $R2$ or $R11$ back to CPU memory as they are read-only.

Scalability problem. We mentioned that LIBMF faces serious scalability issue, as the scheduling overhead increases quickly with the number of workers [41]. Our multiple-GPU scheduling scheme has similar complexity with that of LIBMF. However, it does not face the same scalability issue as we only need to schedule to a few GPUs instead of hundreds of workers.

6.2 Optimizing Data Transfer

GPUs' memory bandwidth are much higher than the CPU-GPU memory transfer bandwidth. For example, NVIDIA TITAN X GPU provides 360 GB/s device memory bandwidth while the CPU-GPU memory bandwidth is only ~16 GB/s (PCIe v3.0 16x). In the single-GPU implementation, CPU-GPU memory transfer only happens at the start and end of MF, and therefore not dominant. However, when the data set can not fit into the GPU memory, memory transfer happens frequently and has a significant impact on the overall performance.

Given the memory transfer overhead, we overlap the memory transfer and computation when solving large problems, as shown in Figure 8(b). Due to space limitation, we only plot one GPU. The



(a) Multi-GPU solution. *(b) Optimized multi-GPU solution.*

Figure 8: (a) Multi-GPU solution of cuMF_SGD, where the rating matrix is partitioned and each partition can fit into a GPU's device memory. (b) Optimizing the multi-GPU solution by overlapping memory transfer with computation.

key idea is, at the block scheduling time, instead of randomly selecting one independent block for the GPU, the optimized technique randomly **selects multiple blocks at a time**. Those blocks are pipelined to overlap the memory transfer and computation. In that case, we schedule two blocks to GPU 0, and overlap the memory transfer of the second block ($R8$) with the computation of the first block ($R2$). Note that the two blocks scheduled to one GPU do not need to be independent as they are updated in serial; meanwhile, blocks scheduled to different GPUs have to be independent of each other to avoid conflicts. By doing so, we can minimize the overhead of slow CPU-GPU memory transfer and improve the overall performance.

Discussion. Allocating more blocks to one GPU would yield more performance benefit as more memory/computation overlapping can be achieved. However, the number of available blocks is limited by how we divide the rating matrix R. Consider we divide R to $i \times i$ and we have two GPUs running in parallel, the number of blocks per GPU cannot be more than $i/2$. In practice, i is determined by the size of the rating matrix R and the available hardware resources on the GPU. We will discuss it in Section 7.6.

6.3 Implementation Details

Multiple GPUs management. We implement it using multiple CPU threads within one process. Within the process, there is one *host thread* and multiple *worker threads*, where each GPU is bound to one worker thread. The host thread manages the workload scheduling and informs worker threads of the scheduling decision. Each worker thread then starts the data transfer and launches compute kernels on a GPU.

Overlapping. Each worker thread will overlap the computation and CPU-GPU memory transfers. We use CUDA *streams* to achieve this. A *stream* contains a list of GPU commands that are executed in serial, and commands in different streams are executed in parallel if hardware resources permit. Each worker thread uses three streams that manage CPU-GPU memory transfer, GPU-CPU memory transfer, and GPU kernel launch, respectively.

7 EXPERIMENTS

We implement cuMF_SGD using CUDA C (source code available at http://github.com/cumf/cumf_sgd/), evaluate its performance on public data sets, and demonstrate its advantage in performance and cost. The following experiments are designed to answer the following questions:

- Compared with state-of-the-art SGD-based MF solutions [11, 16, 63], is cuMF_SGD better and why? (Section 7.2)
- What is the implication of using different generations of GPUs? (Section 7.3)
- Compared with the ALS-based GPU library **cuMF_ALS** that we published earlier [51], what is the advantage of cuMF_SGD? (Section 7.4)
- Parallelizing SGD is always tricky and may lead to converge problems, how does cuMF_SGD perform with different parallelization parameters? (Section 7.5)
- Is there any limitation that may incur convergence problems with matrix blocking-based algorithms? (Section 7.6)
- When scaling up to multiple GPUs, is cuMF_SGD still efficient? (Section 7.7)

7.1 Machine Learning Parameters

As mentioned in the introduction and background, this paper focuses on system-level optimization, not algorithmic-level optimization. Therefore, we do not spend much effort on machine learning parameter tuning. Instead, we use the parameters adopted by earlier works [15, 16, 51, 63]. For the learning rate, we adopt the learning rate scheduling technique used by Yun et al. [63], where the learning rate γ_t at epoch t is monotonically reduced in the following routine:

$$\gamma_t = \frac{\alpha}{1 + \beta \cdot t^{1.5}} \tag{9}$$

α is the given initial learning rate and β is another given parameter. The parameters used by cuMF_SGD are listed in Table 3.

Table 3: Machine learning parameters used for all three data sets.

Dataset	λ	α	β
Netflix	0.05	0.08	0.3
Yahoo!Music	1.0	0.08	0.2
Hugewiki	0.03	0.08	0.3

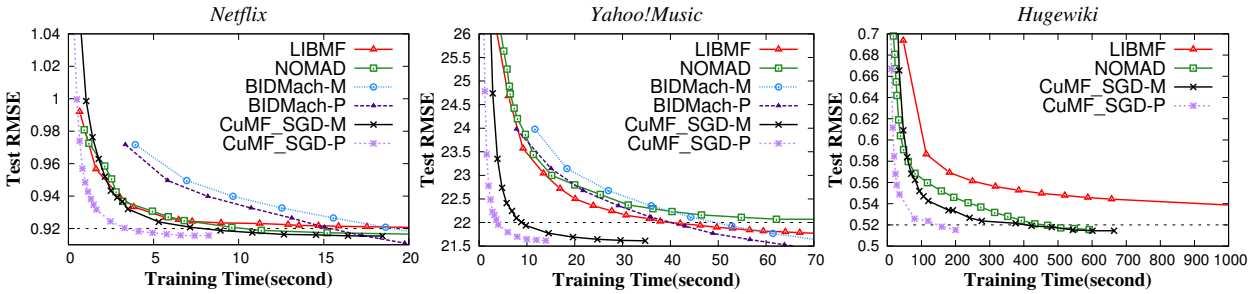


Figure 9: Test RMSE over training time on three data sets. CuMF_SGD converges faster than all other approaches with only one GPU card.

7.2 Comparison of SGD-based approaches

We only compare with MF-specific solutions as they represent the performance upper bound of MF solutions. Machine learning frameworks, such as TensorFlow [6], MXNet [14], also support matrix factorization. Their goal is to provide a unified interface for all machine learning applications. Hence, they are not neccessarily efficient in terms of performance. We compare cuMF_SGD with the following state-of-the-art approaches.

- **LIBMF** [15]. LIBMF is a representative matrix blocking-based solution on shared-memory systems. Its main design purpose is to balance the workload across CPU threads and accelerate the memory access using caches. It also leverages SSE instructions and a novel learning rate schedule to speed up the convergence [16]. The Maxwell platform supports up to 48 concurrent physical threads. We exhaustively evaluate all possible numbers of threads (1~48) on the Maxwell platform and we choose to use 40 CPU threads as it yields fastest convergence. LIBMF divides the rating matrix into $a \times a$ blocks. We evaluate different values for a(40~160) and select the optimal value(100). We set its initial learning rate as 0.1 as suggested in the original paper.
- **NOMAD** [63]. NOMAD is a representative distributed matrix factorization solution. It uses a 64-node HPC cluster to solve MF. It proposes a decentralized scheduling policy to reduce the synchronization overhead and discusses how to reduce the inter-node communication overhead. We cite the best results presented in the original paper, i.e., using 32 nodes for *Netflix* and *Yahoo!Music* data sets and using all 64 nodes for *Hugewiki* data set on the HPC cluster. Each node employs 4 CPU cores. That is, NOMAD launches 128 parallel workers for *Netflix* and *Yahoo!Music*, 256 parallel workers for *Hugewiki*.
- **BIDMach**. BIDMach [11] is a machine learning acceleration library that supports SGD-based MF on GPU. We evaluate BIDMach on both Maxwell and Pascal platforms using the default GPU configurations. We name the results on Maxwell as *BIDMach-M* and those on Pascal as *BIDMach-P*. We are not able to successfully run BIDMach for *Hugewiki* due to memory allocation error. *Hugewiki* has over 3B non-zero samples. BIDMach requires ~62GB memory space, exceeding single GPU's memory(12 GB on Maxwell, 16GB on Pascal).
- **CuMF_SGD**. We evaluate cuMF_SGD on both Maxwell and Pascal platforms, with all three data sets. We name the results on Maxwell as *cuMF_SGD-M* and those on Pascal as *cuMF_SGD-P*. We use one GPU in this subsection. The number of parallel workers (thread blocks) is set as the maximum of the corresponding GPU architecture (768 on Maxwell platform and 1792 on Pascal platform). We use half precision to store feature matrices, however, *Hugewiki* still requires ~49GB memory space, exceeding the GPU's memory. We divide it into 64×1 blocks and at each scheduling time, we schedule 8 blocks to overlap memory transfer and computation. Each block only occupies 0.77GB memory space. CuMF_SGD needs to keep two blocks in the memory to overlap computation and memory transfer. Overall, cuMF_SGD only occupies 1.54GB memory space in GPU's memory.

Figure 9 shows the test RMSE w.r.t. the training time. Table 4 summarizes the training time required to converge to a reasonable RMSE (0.92, 22.0, and 0.52 for *Netflix*, *Yahoo!Music*, and *Hugewki*, respectively). Results show that **with only one GPU, cuMF_SGD-P and cuMF_SGD-M perform much better (3.1X to 28.2X) on all data sets compared than all existing works**, including NOMAD on a 64-node HPC cluster. In the following, we analyze the reasons.

Table 4: Training time speedup normalized to LIBMF.

Data set	*Netflix*	*Yahoo!Music*	*Hugewiki*
LIBMF	23.0s	37.9s	3020.7s
NOMAD	9.6s(2.4X)	108.7s(0.35X)	459.1s(6.6X)
BIDMach-M	18.6s(1.24X)	48.6s(0.78X)	-
BIDMach-P	15.0s(1.53X)	39.5s(0.96X)	-
CuMF_SGD-M	7.5s(3.1X)	8.8s(4.3X)	442.3s(6.8X)
CuMF_SGD-P	3.3s(7.0X)	3.8s(10.0X)	107.0s(28.2X)

Compared with LIBMF. As shown in Figure 9 and Table 4, cuMF_SGD outperforms LIBMF on all data sets, on both Maxwell and Pascal. More precisely, cuMF_SGD-M is 3.1X - 6.8X as fast as LIBMF and cuMF_SGD-P is 7.0X - 28.2X as fast. CuMF_SGD outperforms LIBMF because it can do more updates per second, as shown in Figure 10(a). We have already mentioned that matrix factorization is memory bound, LIBMF is also aware of that and strives to keep all frequently used data in the CPU cache. However,

the limited cache capacity on a single CPU makes LIMBF suboptimal in large data sets. As shown in Figure 10(b), LIBMF achieves an effective memory bandwidth of 194 GB/s[2] on the *Netflix* data set (with 99M samples) – close to cuMF_SGD-M. However its achieved bandwidth drops almost by half, to 106 GB/s on the larger *Hugewiki* data set (with 3.1B samples) – while cuMF_SGD achieves similar bandwidth in all data sets.

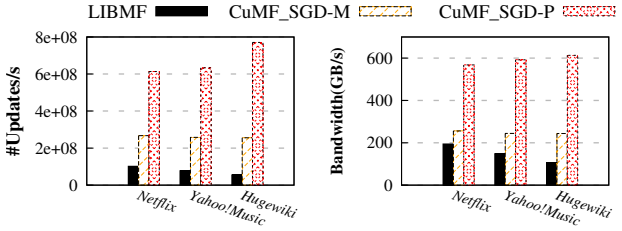


Figure 10: Achieved *#Updates/s* and memory bandwidth of LIBMF, cuMF_SGD-M, and cuMF_SGD-P. The achieved memory bandwidth explains the advantage of cuMF_SGD.

Simply porting LIBMF to GPUs leads to resource under-utilization due to the scalability problem of it scheduling policy (recall Figure 5). In contrast, the workload scheduling policy and memory/computation pattern of cuMF_SGD are delicately designed to fully exploit the computation and memory resources on GPUs. Hence, as shown in Figure 10 (b), cuMF_SGD achieves much higher bandwidth than LIBMF. Moreover, cuMF_SGD uses half-precision (2 bytes for a float number) to store feature matrices. As a result, it can perform twice updates as LIBMF with the same bandwidth consumption.

Compared with NOMAD. As presented in [63], NOMAD uses 32 nodes for *Netflix* and *Yahoo!Music* and 64 HPC nodes for *Hugewiki*. Despite the tremendous hardware resources, NOMAD is still outperformed by cuMF_SGD on all data sets. As observed in Section 2, MF is a memory bound application and data communication happens frequently between parallel workers. When NOMAD distributes parallel workers to different nodes, the network bandwidth, which is much lower than intra-node communication bandwidth, becomes the bottleneck. Consequently, NOMAD achieves suboptimal scalability when scaling from single node to multiple nodes, especially for small data sets. For example, on *Yahoo!Music*, NOMAD performs even worse than LIBMF that uses only one node.

NOMAD (on a 64-node HPC cluster) has similar performance with cuMF_SGD-M on *Hugewiki*, while it is much slower than cuMF_SGD-P. Obviously, cuMF_SGD is not only faster, using a single GPU card, it is also more cost-efficient.

Table 5: Achieved *#Updates/s* of BIDMach and cuMF_SGD.

Data set	*Netflix*	*Yahoo!Music*	*Hugewiki*
BIDMach-M	25.2M	21.6M	-
BIDMach-P	29.6M	32.3M	-
CuMF_SGD-M	267M	258M	256M
CuMF_SGD-P	613M	634M	710M

[2]The achieved memory bandwidth measures the data processed by the compute units per second, and can be higher than the theoretical off-chip memory bandwidth thanks to the cache effect.

Compared with BIDMach. BIDMach implements SGD-based MF on GPUs. Different from cuMF_SGD, BIDMach employs the ADAGRAD [20] algorithm to fine tune the learning rate for faster convergence. However, as shown in Figure 9 and Table 4, BIDMach is still outperformed by cuMF_SGD. CuMF_SGD is designed to execute the SGD updates efficiently (Section 4) with low scheduling overhead (Section 5) and minimize the CPU-GPU transfer overhead (Section 6). CuMF_SGD is able to fully exploit the hardware resources on GPUs and achieves higher throughputs (Table 5) than BIDMach. Besides, cuMF_SGD yields better cross-architecture scalability than BIDMach as cuMF_SGD achieves more speedup when porting from Maxwell to Pascal GPUs. In addition, cuMF_SGD can also use ADAGRAD or other learning rate schedulers, for faster convergence. We leave it as future work.

7.3 Implication of GPU Architectures

We have evaluated cuMF_SGD on the two current generations of GPUs, Maxwell and Pascal. As we presented, cuMF_SGD performs consistently well on both platforms. We believe that cuMF_SGD is able to scale to future GPU architectures with minor tuning effort. In this section, we explain the performance gap between Maxwell and Pascal in three aspects: computation resources, off-chip memory bandwidth, and CPU-GPU memory bandwidth.

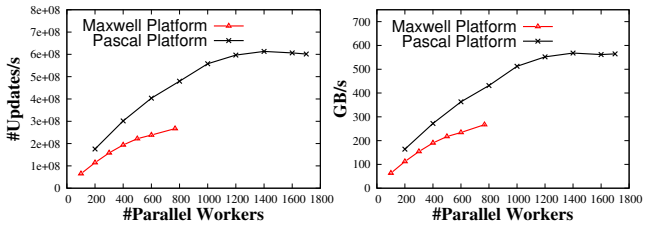


Figure 11: *#Updates/s* and achieved memory bandwidth cuMF_SGD on Maxwell and Pascal platforms, using the *Netflix* data set. CuMF_SGD performs better on the more recent Pascal platform.

Computation resources. We show the *#Updates/s* metric of two platforms with different numbers of parallel workers using *Netflix* in Figure 11(a). Results show that Pascal platform scales to more parallel workers and achieves much higher *#Updates/s* than Maxwell. This is because the Maxwell platform has 24 streaming multiprocessors (SMs) within each GPU, with each SM allowing up to 32 parallel workers (thread blocks). Hence, one Maxwell GPU allows up to 768 parallel workers. Meanwhile, the Pascal GPU used has 56 SMs and allows 32 thread blocks on each SM. Hence, a Pascal GPU allows up to 1792 parallel workers, which is 2.3 times of that of Maxwell GPU. Overall, a Pascal GPU is more powerful than a Maxwell GPU in term of the amount of computation resources.

Off-chip memory bandwidth. As we discussed before, SGD is memory bound. Optimized for throughput, GPUs are able to overlap memory access and computation by fast context switch among concurrent threads [2]. When there are enough threads concurrently running on GPUs, long memory latencies can be hidden, which is exactly what happens with cuMF_SGD. In this scenario, memory bandwidth, instead of memory latency, becomes the limitation of

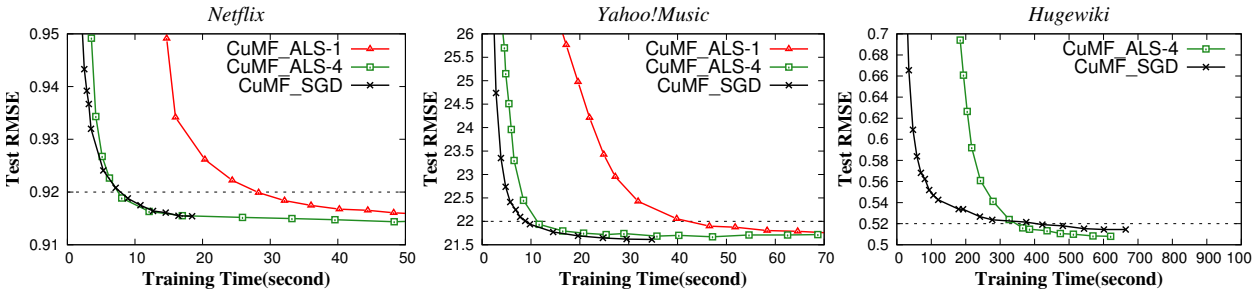


Figure 12: CuMF_SGD vs. CuMF_ALS. With one GPU, cuMF_SGD converges ~4X faster than cuMF_ALS-1 (one GPU) and similar to cuMF_ALS-4 (four GPUs).

the performance. Pascal platforms provide twice as much theoretical peak off-chip memory bandwidth (780 GB/s per GPU) as Maxwell platforms(360 GB/s per GPU). Figure 11(b) shows the achieved memory bandwidth on two platforms with different number of parallel workers. On Maxwell and Pascal, cuMF_SGD achieves up to 266 GB/s and 567 GB/s memory bandwidth, respectively.

CPU-GPU memory bandwidth. *Netflix* and *Yahoo!Music* data sets are small enough to fit into the GPU device memory. For *Hugewiki*, memory transfer occurs multiple times as the data cannot fit into GPU device memory and the overhead is non-negligible. In Section 6.2, we propose to overlap data transfer with computation. Despite of this optimization, the CPU-GPU memory bandwidth still has a noticeable impact on the overall performance as the perfect overlapping cannot be achieved. On the Maxwell platform, the memory transfer between CPU and GPU is via PCIe v3.0 16x with 16 GB/s bandwidth (we observe that on average, the achieved bandwidth is 5.5 GB/s). The very recent Pascal platform is with NVLink [4] that can provide 80 GB/s in theory (we observe an average 29.1 GB/s CPU-GPU memory transfer bandwidth, which is 5.3X as that on Maxwell). This also explains why cuMF_SGD achieves much more speedup on *Hugewiki* using Pascal platform (28.2X) than that on Maxwell platform (6.8X).

7.4 Comparison with cuMF_ALS

Our earlier work **CuMF_ALS** [51] represents the state-of-art ALS-based matrix factorization solution on GPUs. We use one GPU for cuMF_SGD, and one and four GPUs for cuMF_ALS. Figure 12 compares their performance on three data sets on Maxwell. We observe that cuMF_SGD, with one GPU, is faster than cuMF_ALS-1 and achieves similar performance with cuMF_ALS-4.

It is expected that cuMF_SGD is faster than cuMF_ALS, for the following reason. Each epoch of SGD needs memory access of $O(N*k)$ and computation of $O(N*k)$. Each epoch of ALS needs memory access of $O(N*k)$ and computation of $O(N*k^2+(m+n)*k^3)$. Thus, ALS's epochs run slower due to its much more intensive computation. Although ALS needs fewer epochs to coverage, as a whole it converges slower.

Despite the fact that cuMF_ALS is slower than cuMF_SGD, both solutions are maintained at https://github.com/cuMF/ because they serve different purposes: SGD converges faster and is easy to do incremental update, while ALS is easy to parallelize and is able to deal with non-sparse rating matrices [34].

7.5 Convergence analysis of cuMF_SGD

The original SGD algorithm is serial. To speed it up, we discuss how to parallelize it on one GPU in Section 5 and on multiple GPUs in Section 6.1. It is well-known that SGD parallelization may have subtle implications on convergence [15, 63]. In this Section, we provide convergence analysis of cuMF_SGD.

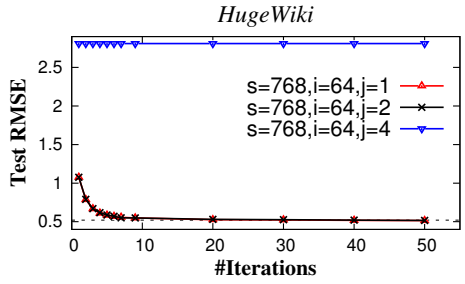


Figure 13: Convergence speed of cuMF_SGD on *Hugewiki* with different parallelization parameters.

Section 5.1 proposes the batch-Hogwild! scheme to schedule the workload and cuMF_SGD adopts it to exploit the data parallelism. As a vectorized version of Hogwild!, batch-Hogwild! inherits the limitation of Hogwild!. Given a rating matrix of $m \times n$ and s parallel workers, convergence is ensured only when the following condition is satisfied [44]:

$$s \ll min(m, n)$$

For multiple GPUs, Section 6 proposes to first divide the rating matrix R into $i \times j$ blocks and process one block on one GPU in parallel if possible. In this case, the above condition needs to change to:

$$s \ll min(\lfloor m/i \rfloor, \lfloor n/j \rfloor)$$

To evaluate the convergence speed of cuMF_SGD and find out the potential convergence problem with it, we conduct the following experiments.

We first fix i and j for all three datasets and vary s. *Netflix* and *Yahoo!Music* data sets are small enough to fit into one GPU's memory, we fix $i = 1$ and $j = 1$. We fix $i = 64$ and $j = 1$ for *Hugewiki*. On both Maxwell and Pascal platforms, we enumerate all possible values of s([1, 768] on Maxwell, [1, 1792] on Pascal) and collect the performance metrics. We observe that the $\#Updates/s$ varies

with s as we discussed in Section 7.3. As a result, *Test RMSE* w.r.t. training time varies with s. At the mean-time, we observe that *Test RMSE* w.r.t. $\#Iterations$ does not change. Therefore, for the used data sets, the convergence speed of cuMF_SGD does not decrease with parallelism increasing within one GPU.

We conduct another experiment to find out the convergence limitation in cuMF_SGD. Among the three data sets, *Hugewiki* is the largest one and intuitively, cuMF_SGD should gain more speedup through employing multiple GPUs. However, the n of *Hugewiki* is only 40 thousand, preventing cuMF_SGD from further increasing the parallelism beyond one GPU. When we try to partition the *Hugewiki* data set into more blocks (increase i and j), convergence is not ensured. We show an empirical study on *Hugewiki* in Figure 13. *Hugewiki* has $min(m, n) = 40k$ and we fix $s = 768$; convergence is achieved when $j \leq 2$ ($40k/20/768 \approx 2$), and fails when $j = 4$. The result shows that, to ensure the converge, we can not infinitely increase the parallelism and the following regulation has to be complied,

$$s < 1/20 * min(\lfloor m/i \rfloor, \lfloor n/j \rfloor)$$

We believe this is a limitation for all Hogwild!-style solutions.

7.6 Convergence analysis of matrix blocking

Matrix blocking is used to parallelize SGD-based MF by many applications, e.g., LIBMF. CuMF_SGD uses matrix blocking to tackle with workload partitioning. The purpose of matrix blocking is to avoid conflicts between parallel workers. However, we observe that matrix blocking can have a negative impact on convergence. We use LIBMF as a case study. Figure 14 illustrates the convergence speed of LIBMF on *Netflix* data set with different parameters. In this study, we fix the number of parallel workers $s = 40$; without loss of generality, we divide R into $a \times a$ blocks and vary the value of a. Figure 14 shows that when a is less than or close to s, convergence speed is much slower or even cannot be achieved. We have similar observations on other data sets and using cuMF_SGD. We briefly explain the reason with a simple example shown in Figure 15.

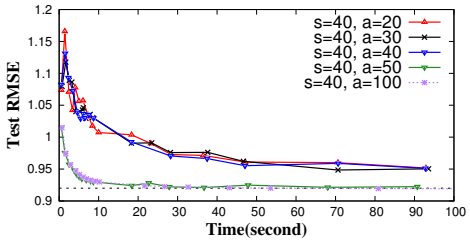


Figure 14: Convergence speed of LIBMF on *Netflix*. We fix #parallel-workers $s = 40$ and vary value a to partition to $a \times a$ blocks.

In Figure 15, we divide the rating matrix into 2×2 blocks and use 2 parallel workers. In theory, 4 blocks can have $4 \times 3 \times 2 \times 1 = 24$ possible update orders. We also show all update orders in Figure 15. However, only orders 1~8 out of the total 24 orders are feasible so as to avoid update conflicts. For example, when *Block 1* is issued to one worker, only *Block 4* can be issued to another worker. Hence, *Blocks 2* and *3* cannot be updated between 1 and 4, which precludes order 9~12. This demonstrated that when $s \geq a$, all independent blocks have to be updated concurrently to make all workers busy, which enforces certain update order constraints and hurts the randomness. As a consequence, convergence speed can deteriorate. In practice, when cuMF_SGD uses two GPUs, R should at least be divided into 4×4 blocks.

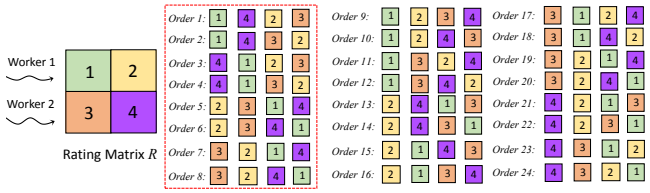


Figure 15: A simple example to demonstrate the limitation of matrix blocking. The rating matrix is divided into 2×2 blocks and updated using two parallel workers.

7.7 Scale up to multiple GPUs

System wise, cuMF_SGD is designed to scale to multiple GPUs. However, algorithmic wise, the scaling is restricted by factors such as problem dimension and number of parallel workers, as discussed earlier in Section 7.5 and Section 7.6. Among the three data sets used in this paper, *Netflix* and *Hugewiki* have very small n($20k$ and $40k$, receptively), preventing cuMF_SGD from solving them on multiple GPUs. In comparison, *Yahoo!Music* can be solved on multiple GPUs as the dimension of it R is $1M \times 625k$. We divide its R into 8×8 blocks and run it with two Pascal GPUs. Figure 16 shows the convergence speed. With 2 Pascal GPUs, cuMF_SGD takes 2.5s to converge to RMSE 22, which is 1.5X as fast as 1 Pascal GPU (3.8s). The reason behind this sub-linear scalability is that the multi-GPU cuMF_SGD needs to spend time on CPU-GPU memory transfer so as to synchronize two GPUs.

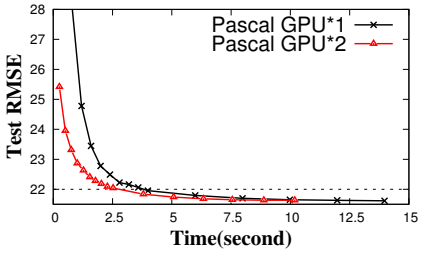


Figure 16: Convergence of cuMF_SGD on *Yahoo!Music*: two Pascal GPUs is 1.5X as fast as one.

8 RELATED WORK

Algorithms. SGD has been widely used to solve matrix factorization [34]. Serial SGD can be parallelized to improve performance [8, 67]. ALS is naturally easy to parallelize and it can also be used in dense matrix factorization [33]. Coordinate descent is another algorithm to solve matrix factorization [26, 61]. It updates

the feature matrix along one coordinate direction in each step. Our earlier work [51] focuses on ALS algorithm.

Parallel SGD solutions have been discussed in multi-core [15, 16, 42], multi-node [52, 63], MapReduce [23, 35] and parameter-servers [46] settings. Existing works are mostly inspired by Hogwild! [44] that allows lock-free update, use matrix-blocking partitioning to avoid conflicts, or use a combination of them. LIBMF [15, 16] is a representative shared-memory system-based solution. Evaluations have shown that it outperforms all previous single-node approaches. Although it has been optimized for cache efficiency, it is still not efficient at processing large-scale data sets. Moreover, the high complexity of its scheduling policy makes it infeasible to scale to many cores. NOMAD [63] partitions the data on HPC clusters to improve the cache performance. At the meantime, it proposes to minimize the communication overhead. Compared with LIBMF, it has similar performance on one machine and is able to scale to 64 nodes. However, none of the above solutions use GPU as accelerators.

Parallelization is also used in coordinate descent [61]. Compared with SGD, coordinate descent has lower overhead and runs faster at the first few epochs of training. However, due to the algorithmic limitation, coordinate descent is prone to reach local optima [15] in the later epochs of training. Compared with CGD and SGD, ALS is inherently easy to parallel, ALS-based parallel solutions are widely discussed [1, 22, 39, 40, 66]. Our earlier work, cuMF_ALS [51] focuses on optimizing ALS to matrix factorization on GPUs. As ALS algorithm has more compute intensive epochs, it runs slower than cuMF_SGD.

GPU solutions. The emerging of CUDA and OpenCL programming models makes GPUs popular as accelerators [21, 24]. Applications, including storage system [7], graph processing [25], hash table [53], neural network [18, 58], linear algebra [13], have enjoyed the tremendous computational horsepower equipped on GPUs. Prior to our work, [10] applies Restricted Boltzmann Machines on GPUs to solve MF. [64] implements both SGD and ALS on GPU to solve MF. [27] proposes matrix blocking-based MF solution on GPUs and [31] evaluates the workload scheduling overhead for SGD on GPUs. BIDMach [11] supports SGD-based MF and uses GPUs as accelerators. To the best of our knowledge, cuMF_SGD outperforms all existing solutions because we optimize both memory access and workload scheduling.

9 CONCLUSION

Matrix factorization is widely used in recommender systems and other applications. SGD-based MF is limited by memory bandwidth that single and multi-CPU systems cannot effectively provision. We propose a GPU-based solution, by observing that GPUs offer abundant memory bandwidth and fast intra-node connection. We design workload partitioning and scheduling schemes to dispatch tasks inside a GPU and across GPUs, without impacting the randomness required by SGD. We also develop highly-optimized GPU kernels for individual SGD updates. With only one Maxwell or Pascal GPU, cuMF_SGD runs **3.1X-28.2X** as fast compared with state-of-art CPU solutions on 1-64 CPU nodes. Evaluations also show that cuMF_SGD scales well on multiple GPUs for large data sets. In future, we plan to extend cuMF_SGD to multiple nodes and investigate how to deal with incremental training.

REFERENCES

[1] Recommending items to more than a billion people, 2015. https://code.facebook.com/posts/861999383875667/recommending-items-to-more-than-a-billion-people/.

[2] NVIDIA CUDA programming guide., 2016. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[3] NVIDIA Maxwell Architecture . https://developer.nvidia.com/maxwell-compute-architecture, 2016.

[4] NVIDIA NVLink, 2016. http://www.nvidia.com/object/nvlink.html.

[5] NVIDIA Pascal Architecture, 2016. http://www.geforce.com/hardware/10series/architecture.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[7] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu. GPUs as storage system accelerators. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1556–1566, 2013.

[8] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[9] M. Butler, K. Sajjapongse, and M. Becchi. Improving application concurrency on GPUs by managing implicit and explicit synchronizations. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, pages 535–544. IEEE, 2015.

[10] X. Cai, Z. Xu, G. Lai, C. Wu, and X. Lin. GPU-accelerated restricted boltzmann machine for collaborative filtering. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2012.

[11] J. Canny and H. Zhao. Bidmach: Large-scale learning with zero memory allocation. In *BigLearning, NIPS Workshop*, 2013.

[12] S. Chang, Y. Zhang, J. Tang, D. Yin, Y. Chang, M. A. Hasegawa-Johnson, and T. S. Huang. Streaming recommender systems. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, 2017.

[13] J. Chen, L. Tan, P. Wu, D. Tao, H. Li, X. Liang, S. Li, R. Ge, L. Bhuyan, and Z. Chen. GreenLA: green linear algebra software for GPU-accelerated heterogeneous computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 57. IEEE Press, 2016.

[14] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[15] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2015.

[16] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin. A learning-rate schedule for stochastic gradient methods to matrix factorization. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2015.

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[18] B. Del Monte and R. Prodan. A scalable GPU-enabled framework for training deep neural networks. In *Green High Performance Computing (ICGHPC), 2016 2nd International Conference on*, pages 1–8. IEEE, 2016.

[19] C. del Mundo and W.-c. Feng. Enabling efficient intra-warp communication for Fourier transforms in a many-core architecture. In *Supercomputing, 2013. Proceedings of the 2013 ACM/IEEE International Conference on*, 2013.

[20] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[21] J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, pages 216–225. IEEE, 2011.

[22] M. Gates, H. Anzt, J. Kurzak, and J. Dongarra. Accelerating collaborative filtering using concepts from high performance computing. In *Big Data, 2015 IEEE International Conference on*, 2015.

[23] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011.

[24] A. Goswami, J. Young, K. Schwan, N. Farooqui, A. Gavrilovska, M. Wolf, and G. Eisenhauer. GPUShare: Fair-sharing middleware for GPU clouds. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1769–1776. IEEE, 2016.

[25] S. Heldens, A. L. Varbanescu, and A. Iosup. Dynamic load balancing for high-performance graph processing on hybrid CPU-GPU platforms. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, pages 62–65. IEEE Press, 2016.

[26] C.-J. Hsieh and I. S. Dhillon. Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011.

[27] J. Jin, S. Lai, S. Hu, J. Lin, and X. Lin. GPUSGD: A GPU-accelerated stochastic gradient descent algorithm for matrix factorization. *Concurrency and Computation: Practice and Experience*, 2015.

[28] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *ACM SIGPLAN Notices*, volume 48, pages 395–406. ACM, 2013.

[29] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das. Anatomy of GPU memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 223–234. ACM, 2015.

[30] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated scheduling and prefetching for GPGPUs. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 332–343. ACM, 2013.

[31] R. Kaleem, S. Pai, and K. Pingali. Stochastic gradient descent on GPUs. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 81–89. ACM, 2015.

[32] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[33] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[34] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.

[35] B. Li, S. Tata, and Y. Sismanis. Sparkler: supporting large-scale matrix factorization. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 2013.

[36] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 67–77. ACM, 2015.

[37] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *SIGKDD*, pages 661–670. ACM, 2014.

[38] Z. Liu, Y.-X. Wang, and A. Smola. Fast differentially private matrix factorization. In *Proceedings of the 9th ACM Conference on Recommender Systems*, RecSys'15, 2015.

[39] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 2012.

[40] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 2016.

[41] Y. Nishioka and K. Taura. Scalable task-parallel SGD on matrix factorization in multicore architectures. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW '15, pages 1178–1184, Washington, DC, USA, 2015. IEEE Computer Society.

[42] J. Oh, W.-S. Han, H. Yu, and X. Jiang. Fast and robust parallel SGD matrix factorization. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015.

[43] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[44] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[45] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, 2008.

[46] S. Schelter, V. Satuluri, and R. Zadeh. Factorbird-a parameter server approach to distributed matrix factorization. *arXiv preprint arXiv:1411.0602*, 2014.

[47] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges. Infiniband scalability in Open MPI. In *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[48] D. Song and S. Chen. Exploiting SIMD for complex numerical predicates. In *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*, 2016.

[49] S. Song and K. W. Cameron. System-level power-performance efficiency modeling for emergent GPU architectures. In *PACT*, pages 473–474, 2012.

[50] J. Tan, S. L. Song, K. Yan, X. Fu, A. Marquez, and D. Kerbyson. Combating the reliability challenge of GPU register file at low supply voltage. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 3–15. IEEE, 2016.

[51] W. Tan, L. Cao, and L. Fong. Faster and Cheaper: Parallelizing large-scale matrix factorization on GPUs. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, 2016.

[52] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *IEEE 12th International Conference on Data Mining*. IEEE, 2012.

[53] A. Todd, H. Truong, J. Deters, J. Long, G. Conant, and M. Becchi. Parallel gene upstream comparison via multi-level hash tables on GPU. In *22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[54] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[55] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48.

[56] X. Xie, Y. Liang, G. Sun, and D. Chen. An efficient compiler framework for cache bypassing on GPUs. In *IEEE/ACM International Conference on Computer-Aided Design*, 2013.

[57] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for GPUs. In *International Symposium on High Performance Computer Architecture*, HPCA'15, pages 76–88, 2015.

[58] F. Yan, O. Ruwase, Y. He, and E. Smirni. SERF: efficient scheduling for fast deep neural network serving via judicious parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 26. IEEE Press, 2016.

[59] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, 2010.

[60] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, 2012.

[61] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, 2012.

[62] H.-F. Yu, C.-J. Hsieh, H. Yun, S. Vishwanathan, and I. S. Dhillon. A scalable asynchronous distributed algorithm for topic modeling. In *Proceedings of the 24th International Conference on WWW*, pages 1340–1350. ACM, 2015.

[63] H. Yun, H.-F. Yu, C.-J. Hsieh, S. V. N. Vishwanathan, and I. Dhillon. NOMAD: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proc. VLDB Endow.*, 2014.

[64] D. Zastrau and S. Edelkamp. Stochastic gradient descent with GPGPU. In *Annual Conference on Artificial Intelligence*. Springer, 2012.

[65] T. Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM, 2004.

[66] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *International Conference on Algorithmic Applications in Management*. Springer, 2008.

[67] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *NIPS*, 2010.