Accelerating Stochastic Gradient Descent Based Matrix Factorization on FPGA

Shijie Zhou, Rajgopal Kannan, and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—Matrix Factorization (MF) based on Stochastic Gradient Descent (SGD) is a powerful machine learning technique to derive hidden features of objects from observations. In this paper, we design a highly parallel architecture based on Field-Programmable Gate Array (FPGA) to accelerate the training process of the SGD-based MF algorithm. We identify the challenges for the acceleration and propose novel algorithmic optimizations to overcome them. By transforming the SGD-based MF algorithm into a bipartite graph processing problem, we propose a 3-level hierarchical partitioning scheme that enables conflict-minimizing scheduling and processing of edges to achieve significant speedup. First, we develop a fast heuristic graph partitioning approach to partition the bipartite graph into induced subgraphs; this enables to efficiently use the on-chip memory resources of FPGA for data reuse and completely hide the data communication between FPGA and external memory. Second, we partition all the edges of each subgraph into non-overlapping matchings to extract the maximum parallelism. Third, we propose a batching algorithm to schedule the execution of the edges inside each matching to reduce the memory access conflicts to the on-chip RAMs of FPGA. Compared with non-optimized FPGA-based baseline designs, the proposed optimizations result in up to 60× data dependency reduction, 4.2× bank conflict reduction, and 15.4× speedup. We evaluate the performance of our design using a state-of-the-art FPGA device. Experimental results show that our FPGA accelerator sustains a high computing throughput of up to 217 billion floating-point operations per second (GFLOPS) for training very large real-life sparse matrices. Compared with highly-optimized GPU-based accelerators, our FPGA accelerator achieves up to 12.7× speedup. Based on our optimization methodology, we also implement a software-based design on a multi-core platform, which demonstrates 1.3× speedup compared with the state-of-the-art multi-core implementation.

Index Terms—Machine learning, sparse matrix factorization, training acceleration, bipartite graph representation, FPGA accelerator

1 Introduction

Matrix Factorization (MF), which factors a sparse matrix into two low-rank matrices, is a widely used machine learning technique for many applications such as collaborative filtering [1], topic modeling [10], and text mining [11]. This technique can achieve high prediction accuracy because it is able to derive latent features from observations [1]. Therefore, the model trained by the matrix factorization techique is also called latent factor model [1]. Stochastic Gradient Descent (SGD) is gradient descent optimization technique used to minimize certain objective functions [47]. It is a popular learning algorithm to train the MF model. However, the training process of the SGD-based MF algorithm is very computation-intensive. This is because the MF model needs to be iteratively updated based on the training data for thousands of iterations [1]. When the volume of training data is huge, the training time can become excessively long. Therefore, it is essential to design hardware accelerators to accelerate the training process. However, existing accelerators suffer from the expensive synchronization overhead and the limited parallelism of this algorithm [22]. Accelerating SGD-based MF is still a challenging problem.

Field-Programmable Gate Array (FPGA) is an attractive platform to accelerate computation-intensive applications. FPGA has been widely used to accelerate the inference process of machine learning applications [3], [4], [5], [6], [7], but using FPGA to accelerate the training process remains a challenging research area [8], [9]. In this paper, we design a parallel architecture based on state-of-the-art FPGA to accelerate the training process of the SGD-based MF algorithm. Our design methodology is holistic and generalized: It is motivated by a thorough analysis of the challenges involved in accelerating SGD on FPGAs and designed to be adaptable to accelerating SGD-based MF on a variety of computing platforms. We identify three principal acceleration challenges (1) limited on-chip memory which can limit throughput if the long latencies of external memory accesses are not managed, (2) data dependencies that can prevent concurrent processing within the FPGA pipeline, and (3) access conflicts between different pipelines when accessing dual-port on-chip RAMs. Based on our analysis of these design challenges, we develop a bipartite graph processing approach in which the input training data is first transformed into a bipartite graph representation and followed by a novel 3-level hierarchical partitioning, scheduling and processing of on-chip feature vector data that significantly accelerates the processing of this bipartite graph. In addition, we note that our holistic design approach is general enough to be adapted to accelerating multi-core implementations. Thus we also observe that our optimized multi-core implementation outperforms the state-of-the-art multi-core design by 1.3× due to our optimizations.

We summarize the main contributions of this paper

- *Shijie Zhou is with Microsoft Corporation, Redmond, WA, 98052. E-mail: shijieinusc@gmail.com*
- *Rajgopal Kannan is with the US Army Research Lab, Los Angeles, CA, 90094. E-mail: Rajgopal.kannan.civ@mail.mil*
- *Viktor K. Prasanna is with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA, 90089. E-mail: prasanna@usc.edu*

Manuscript received XX, XX, 2019; revised XX, XX, 2019.

below.

- We develop a highly parallel architecture to accelerate the training process of SGF-based MF. The accelerator consists of parallel processing units concurrently working on distinct input data to sustain high processing throughput. It also has an on-chip buffer to store the feature data to achieve efficient data reuse.
- By using a bipartite graph representation of MF, we propose three novel optimizations to overcome the challenges in acceleration.
  - We first partition the input bipartite graph into induced subgraphs. In lieu of more sophisticated partitioning algorithms with higher preprocessing costs, we develop a simple and fast partitioning heuristic that satisfies a necessary condition for storing the feature vectors of vertices in the on-chip buffer. By overlapping the communication overhead with computation, the accelerator can absorb the long latencies of external memory accesses.
  - To overcome the data dependency challenge, we maximize the available parallelism by partitioning the edges of each induced subgraph into matchings. This reduces the data dependencies among the edges by up to $60\times$.
  - To overcome the bank-conflict challenge, we develop a batching algorithm to partition each matching into batches and schedule the execution of the batches to reduce the conflict accesses to the shared on-chip buffer (i.e., bank conflicts). This optimization results in up to $4.2\times$ bank conflict reduction.
- Experimental results show that the proposed FPGA accelerator sustains a high throughput of up to 217 GFLOPS for training large real-life sparse datasets. Compared with state-of-the-art GPU implementations, our FPGA accelerator achieves up to 12.7× speedup and 24.8× throughput improvement.
- Our proposed optimizations are generic to general purpose processors as well. We adapt our techniques to mutli-core platform and implement a software-based design. The optimized software-based design achieves 1.3× speedup compared with state-of-the-art multi-core implementation with 1.5× fewer cores.

The rest of the paper is organized as follows. Section 2 covers the background and defines the problem; Section 3 introduces the challenges in accelerating the SGD-based MF algorithm; Section 4 presents our proposed optimizations; Section 5 describes the architecture of the accelerator; Section 7 summaries the related work; Section 8 discusses the generality of our design; Section 9 concludes the paper.

2 Background

2.1 Matrix Factorization in Machine Learning

MF is a technique to factor a sparse matrix into the product of two low-rank dense matrices. In machine learning applications, MF is used to predict unknown data based on a collection of existing observations that are represented in the matrix format. For example, MF is used to predict customer ratings on products in collaborative filtering [1]. This technique is powerful because it is able to discover some latent features from the observations. An example latent feature of a customer might be his/her income level, which has an impact on his/her interest in products but cannot be directly derived from the observed data (e.g., ratings on products). Hence, the output model of MF is called latent factor model. It consists of two dense matrices that are called (latent) feature matrices. There are three primary approaches used for training the model, including Alternating Least Square (ALS), Stochastic Gradient Descent (SGD), and Gradient Descent (GD) [12], [13], [43].

ALS-based MF alternately fixes one of the two feature matrices. When one feature matrix is fixed, the other feature matrix is recomputed by solving a least-squares problem. ALS-based MF algorithm is inherently parallelizable and requires fewer iterations to converge than the SGD-based MF algorithm. However, it is not scalable to large-scale datasets due to its cubic algorithmic complexity in each iteration [43]. SGD-based MF randomly initializes both the feature matrices and then iteratively improves them. In each iteration, SGD-based MF sequentially processes all the training data. Based on each training data, it updates the feature matrices in the opposite direction of the gradient. GD-based MF also iteratively updates the feature matrices. But it accumulates the intermediate updates and applies the update only after all the training data have been processed in an iteration. Hence, the key difference between SGD-based MF and GD-based MF is that SGD-based MF updates the features matrices once per training data, while GD-based MF updates the features matrices once per iteration. As a result, GD-based MF requires more iterations to converge as well as more training time than SGD-based MF [36], [44]. In this paper, we focus on accelerating SGD-based MF.

2.2 Problem Definition

Without loss of generality, we define the problem based on the context of collaborative filtering for recommender systems [1]. Let U and V denote a set of users and items, $|U|$ and $|V|$ denote the number of users and items, respectively. As shown in Figure 1, the **input** training dataset is a partially observed rating matrix $R = \{r_{ij}\}_{|U|\times|V|}$, in which r_{ij} represents the rating of item v_j given by user u_i ($0 \leq i < |U|$, $0 \leq j < |V|$). The **output** of the training process contains two low-rank matrices, P (a $|U|\times H$ matrix) and Q (a $|V|\times H$ matrix), which are refereed as user feature matrix and item feature matrix, respectively. A typical value of the rank of P and Q (i.e., H) is 32 [15], [16], [17]. The i-th row of P (denoted as p_i) constitutes a **feature vector** of user u_i and the j-th row of Q (denoted as q_j) constitutes a feature vector of item v_j, respectively. The prediction of the rating of item v_j given by user u_i is the dot product of p_i and q_j:

$$\hat{r}_{ij} = p_i \cdot q_j = \sum_{h=0}^{H-1} p_{ih} \cdot q_{jh} \tag{1}$$

Given an observed rating r_{ij}, the prediction error is computed as $err_{ij} = r_{ij} - \hat{r}_{ij}$. The objective of the training process is to obtain such P and Q that minimize the overall

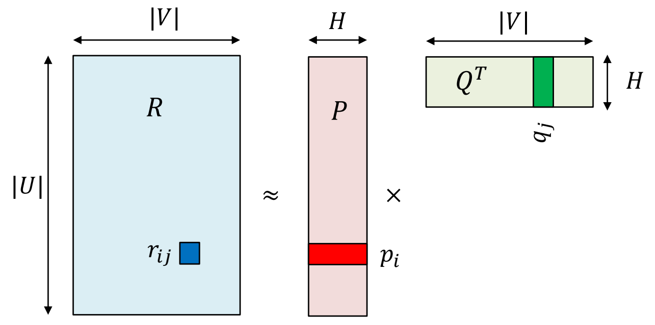

Fig. 1: Input and output of training process

regularized squared error based on all the observed ratings:

$$\min_{P,Q} \sum_{u_i \in U, v_j \in V} err_{ij}^2 + \lambda \cdot (||p_i||^2 + ||q_j||^2) \tag{2}$$

In the objective function, λ is a constant used to introduce regularization to prevent overfitting. In order to minimize the objective function, SGD is used to update the feature vectors. As shown in Algorithm 1, SGD randomly initializes all the feature vectors and then updates them by iteratively traversing all the observed ratings. The training process terminates when the overall squared error (i.e., $\sum err_{ij}^2$) converges. By taking an observed rating r_{ij}, p_i and q_j are updated by a magnitude proportional to a constant α (i.e., learning rate) in the opposite direction of the gradient, yielding the following updating equations:

$$p_i^{new} = \beta \cdot p_i^{old} + err_{ij} \cdot \alpha \cdot q_j^{old} \tag{3}$$

$$q_j^{new} = \beta \cdot q_j^{old} + err_{ij} \cdot \alpha \cdot p_i^{old} \tag{4}$$

In Eq. (3) and (4), β is a constant whose value is equal to $(1-\alpha\lambda)$. The algorithm requires to incrementally update the feature vectors **once per rating**. As a result, the ratings of the same item or given by the same user cannot be concurrently processed because they will result in concurrent updates to the same p_i or q_j. Additional details of this algorithm can be found in [1], [2].

Algorithm 1 SGD-based MF

```
Train ($R$)
1:  Randomly initialize $P$ and $Q$
2:  while Termination_condition = false do
3:      Overall_squared_error= 0
4:      for each observed rating $r_{ij} \in R$ do
5:          Compute $\hat{r}_{ij}$ based on $p_i$ and $q_j$
6:          Compute $err_{ij}$ based on $r_{ij}$ and $\hat{r}_{ij}$
7:          Update $p_i$ and $q_j$ based on Eq. (3) and (4)
8:          Overall_squared_error+ $= err_{ij}^2$
9:      end for
10: end while
11: Return $P$ and $Q$
```

3 Challenges in Acceleration

To achieve efficient acceleration of the SGD-based MF algorithm using FPGA, three challenges need to be carefully addressed.

3.1 Large Data Volume but Limited On-chip Memory Resources

Since the feature vectors of users and items are repeatedly accessed and updated during the processing, it is desirable to store them in the on-chip memory of FPGA for data reuse. However, for large training dataset that involves a large number of users and items, the feature vectors cannot fit in the on-chip memory. In this scenario, external memory such as DRAM is required to store them. However, accessing feature vectors from external memory can incur long access latencies, which result in massive accelerator pipeline stalls and significant performance deterioration [19], [21]. Therefore, the first challenge is how to use the limited on-chip memory resources to achieve efficient data reuse.

3.2 Limited Parallelism due to Data Dependencies

SGD is inherently a serial algorithm because it requires to incrementally update the training model once per training data. As a result, the ratings of the same item or given by the same user have data dependencies and cannot be processed in parallel. This is because concurrent processing of such ratings leads to Read-After-Write (RAW) data hazard. We define such data dependency among ratings as **feature vector dependency**. Hence, the second challenge is how to reduce the feature vector dependencies so that the massive parallelism offered by FPGA can be efficiently exploited.

3.3 Concurrent Accesses to Dual-port On-chip RAMs

FPGA accelerators usually employ parallel processing units to increase processing throughput [3], [19], [20]. However, the native on-chip RAMs (e.g., block RAM and UltraRAM) of FPGA support only dual-port accesses (one read port and/or one write port) [26], [27], [29]. When multiple processing units concurrently access the same RAM based on distinct memory addresses, these memory accesses have to be serially served. This leads to additional latency to resolve the access conflicts as well as performance deterioration. Therefore, the third challenge is how to schedule the execution to reduce such access conflicts.

4 Graph Representation and Algorithmic Optimizations

In order to overcome the three challenges described in Section 3, we represent the MF problem using a bipartite graph representation (Section 4.1) and propose three novel algorithmic optimizations (Section 4.2).

4.1 Graph Representation

We transform SGD-based MF into a bipartite graph-processing problem so that graph theories can be leveraged to optimize the performance. The input matrix is converted into a weighted bipartite graph G, whose vertices can be divided into two disjoint sets, U (user vertices) and V (item vertices). Each observed rating in R is represented as an edge connecting a user vertex and an item vertex in G. G is represented using the coordinate (COO) format [22], which is a commonly used graph representation [16], [17], [18], [22], [23], [24]. In this format, each edge (i.e., $edge_{ij}$) is represented as a <u_i, v_j, r_{ij}> tuple, in which u_i and v_j refer to the user and item vertices connected by the edge, and r_{ij} (i.e., edge weight) corresponds to the rating value of v_j given by u_i; all the edges are stored in an edge list E; each user/item vertex maintains a feature vector whose length is H. Table 1 summaries the frequently used graph notations in this paper. Algorithm 2 illustrates the SGD-based MF algorithm based on the bipartite graph representation. All the edges in E are iteratively processed to update the feature vectors of vertices until the overall squared error converges. When the training process terminates, the feature vectors of all the user vertices and item vertices constitute the output feature matrices P and Q, respectively.

TABLE 1: Bipartite graph notations for MF

Notation	Description
u_i	the user vertex with index i ($0 \le i < \lvert U \rvert$)
v_j	the item vertex with index j ($0 \le j < \lvert V \rvert$)
p_i	the feature vector of u_i
q_j	the feature vector of v_j
H	the length of each feature vector
$edge_{ij}$	the edge connecting u_i and v_j
r_{ij}	the weight of $edge_{ij}$

Algorithm 2 SGD-based MF using graph representation

```
MF_Train ($G(U, V, E)$)
1:  for each user/item vertex do
2:    Randomly initialize its feature vector
3:  end for
4:  while Termination_condition = false do
5:    for each $edge_{ij} \in E$ do
6:      Read feature vectors $p_i$ and $q_j$
7:      Compute $\hat{r}_{ij}$ based on Eq. (1)
8:      Compute $err_{ij}$ based on $r_{ij}$ and $\hat{r}_{ij}$
9:      Update $p_i$ and $q_j$ based on Eq. (3) and (4)
10:   end for
11: end while
12: Return all the feature vectors of vertices
```

4.2 Algorithmic Optimizations

4.2.1 Optimization 1: Graph Partitioning and Communication Hiding

In order to address the challenge described in Section 3.1, we partition G into induced subgraphs (IS) to achieve two goals: (1) the feature vectors of the vertices in each induced subgraph can fit in the on-chip memory of FPGA; (2) the computation for processing each induced subgraph can completely hide the communication cost.

Let L (N) denote the on-chip storage capacity in terms of the number of feature vectors for user (item) vertices. We partition U into l disjoint vertex subsets $\{U_0, \ldots, U_{l-1}\}$, each of size at most L, where $l = \lceil \frac{|U|}{L} \rceil$. Similarly, V is partitioned into $\{V_0, \ldots, V_{n-1}\}$, each of size at most N, where $n = \lceil \frac{|V|}{N} \rceil$. We will introduce our proposed algorithm to perform the partitioning of U and V in details later in this section. Let E_{xy} denote a subset of E that consists of all the edges connecting the vertices belonging to U_x and V_y in G ($0 \le x < l, 0 \le y < n$). U_x, V_y, and E_{xy} form an **Induced Subgraph (**IS**)** of G [30]. The **necessary condition** for the on-chip buffering of all the feature vectors of each IS is:

$$|U_x| \le L, \forall x \in [0, l) \quad \& \quad |V_y| \le N, \forall y \in [0, n) \tag{5}$$

Since we ensure that each user (item) vertex subset has no more than L (N) vertices during the partitioning, the necessary condition for the on-chip buffering of the feature vectors is satisfied.

Because there are l user vertex subsets and n item vertex subsets, the total number of induced subgraphs after the partitioning is $l \times n$. Then, in each iteration of the training process, the induced subgraphs are sequentially processed by our FPGA accelerator based on Algorithm 3. Note that during the processing of the edges in E_{xy}, all the feature vectors of the vertices in U_x and V_y have been prefetched and buffered into an on-chip buffer of FPGA; therefore, the processing units of the accelerator can directly access the feature vectors from the on-chip buffer, rather than from the external memory.

Algorithm 3 Scheduling of induced subgraph processing

```
MF_Train (Induced Subgraphs)
1:  while Termination_condition = false do
2:    for $x$ from 0 to $l - 1$ do
3:      Load feature vectors of $U_x$ into on-chip buffer
4:      for $y$ from 0 to $n - 1$ do
5:        Load feature vectors of $V_y$ into on-chip buffer
6:        Process all the edges $\in E_{xy}$
7:        Write feature vectors of $V_y$ into external memory
8:      end for
9:      Write feature vectors of $U_x$ into external memory
10:   end for
11: end while
```

Using double buffering [3], [31], we pipeline the processing of induced subgraphs to hide the communication cost for data transferring between FPGA and external memory. To illustrate our idea, we define the following terms.

- $|IS_\tau|$: the number of edges in an induced subgraph $IS_\tau, \tau \in [0, l \times n)$
- Ω: the number of edges that can be processed per unit of time
- $P(IS_\tau)$: the computation time to process all the edges of IS_z (i.e., $P(IS_\tau) = \frac{|IS_\tau|}{\Omega}$)
- T_τ^{rd}: the communication time due to reading the feature vectors of IS_τ from external memory
- T_τ^{wr}: the communication time due to writing the feature vectors of IS_τ into external memory
- T_τ: the *intra-subgraph* communication time (i.e., the total time for data transfers, including both reads and writes, occurred during the processing of IS_τ)

As shown in Figure 2, we pipeline the processing of induced subgraphs by overlapping the computation time $P(IS_\tau)$ of IS_τ with the *writing* of feature vectors from $IS_{\tau-1}$ and the *reading* of feature vectors from $IS_{\tau+1}$. Therefore,

$T_\tau = T^{wr}_{\tau-1} + T^{rd}_{\tau+1}$. Here, T_τ in general may include reads or writes of both user and item feature vectors. We can derive the **sufficient condition** for a **complete** overlap of communication and computation:

$$P(IS_\tau) \geq T^{wr}_{\tau-1} + T^{rd}_{\tau+1}, \forall \tau \in [0, l \times n) \quad (6)$$

By replacing the $P(IS_\tau)$ in Inequality (6) with $\frac{|IS_\tau|}{\Omega}$, we obtain:

$$|IS_\tau| \geq \Omega \times T_\tau, \forall \tau \in [0, l \times n) \quad (7)$$

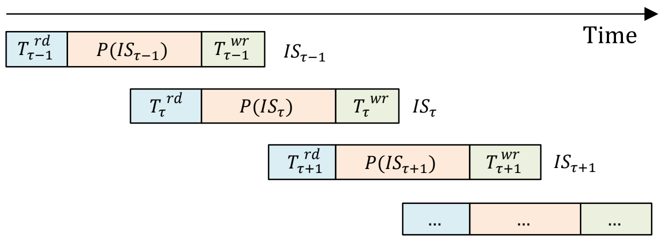

Fig. 2: Pipelined induced subgraph processing

Therefore, besides satisfying the necessary condition for on-chip buffering of feature vectors, a desirable graph partitioning approach should also ensure that each obtained IS has sufficient amount of edges for the computation to completely hide the communication. Note that as parallelism (i.e., Ω) increases, it becomes more challenging to satisfy the sufficient condition, especially when the bipartite graph is sparse. Graph partitioning is a classic problem and many sophisticated graph partitioning algorithms have been developed [33]. However, these sophisticated approaches usually introduce significant preprocessing overhead. Our intuition is that it is not necessary to invest significantly in developing complex partitioning algorithms (in terms of preprocessing time), rather any reasonably fast algorithm that satisfies Inequality (5) and (7) is acceptable. A vertex-index-based partitioning approach [32] has been widely used to perform graph partitioning for hardware accelerators [17], [20]. This approach simply assigns a group of vertices with contiguous indices to each vertex subset subject to the subset size condition (i.e., Inequality (5)). Although this approach is fast, it can lead to significant data imbalance such that some induced subgraphs may have very few edges; in this scenario, the communication cost cannot be completely hidden by the computation. Therefore, we propose a new heuristic graph partitioning approach, which is also simple and fast, but leads to a balanced partitioning effect such that each obtained IS has sufficient edges to satisfy Inequality (6) for a sparse bipartite graph.

We define the *subset degree* of a vertex subset as the total number of edges that connect to the vertices in the subset. When we partition U and V into vertex subsets, we attempt to pack vertices into each disjoint vertex subset such that the subset degrees are close to each other. However, the most important criteria is to ensure that the number of vertices in each vertex subset is bound by L and N as defined earlier. Algorithm 4 illustrates our approach to partition U into $U_0, \cdots, U_{l-1}$; V is partitioned based on the same methodology. We first identify the vertex degree of each vertex (i.e., the number of edges connected to the vertex) and sort all the vertices based on the vertex degree in descending order. Then, we greedily assign each vertex into the vertex subset which has the minimum subset degree, until all the vertices are assigned, subject to the subset size condition.

Algorithm 4 Partition U into l subsets $U_0, \cdots, U_{l-1}$

```
Let $u_{i \cdot degree}$ denote the number of edges connected to $u_i$ ($0 \leq i < |U|$)
Let $U_{x \cdot size}$ denote the number of vertices in $U_x$ ($0 \leq x < l$)
Let $U_{x \cdot degree}$ denote the subset degree of $U_x$ (i.e., $U_{x \cdot degree} = \sum u_{i \cdot degree}, \forall u_i \in U_x$)
Partition ($U$, $L$, $l$)
 1: for $x$ from 0 to $l-1$ do
 2:     $U_x \leftarrow \varnothing$
 3:     $U_{x \cdot degree} \leftarrow 0$
 4:     $U_{x \cdot size} \leftarrow 0$
 5: end for
 6: Sort $U$ based on vertex degree in descending order
 7: for each $u_i \in U$ do
 8:     $subset\_id \leftarrow -1$
 9:     $min\_degree \leftarrow |E|$
10:     for $x$ from 0 to $l-1$ do
11:         if $min\_degree > U_{x \cdot degree}$ and $U_{x \cdot size} < L$ then
12:             $subset\_id \leftarrow x$
13:             $min\_degree \leftarrow U_{x \cdot degree}$
14:         end if
15:     end for
16:     $U_{subset\_id} \leftarrow U_{subset\_id} \cup u_i$
17:     $U_{subset\_id \cdot degree} \leftarrow U_{subset\_id \cdot degree} + u_{i \cdot degree}$
18:     $u_{i \cdot new\_user\_id} \leftarrow subset\_id \times L + U_{subset\_id \cdot size}$
19:     $U_{subset\_id \cdot size} \leftarrow U_{subset\_id \cdot size} + 1$
20: end for
21: Return $U_0, \cdots, U_{l-1}$
```

When each vertex is assigned to a vertex subset, we assign a new vertex index to it (Algorithm 4, Line 18), which indicates the vertex subset that it belongs to and its index in the vertex subset. After U and V are partitioned, we reorder the vertices based on the new indices such that the feature vectors of the vertices belonging to the same vertex subset are stored contiguously in external memory. Since user and item vertices are reordered, we also re-index the user and item indices of each edge and partition the edges into induced subgraphs based on the new indices.

4.2.2 Optimization 2: Parallelism Extraction

To address the challenge described in Section 3.2, we further partition the edges in each IS into a list of non-overlapping **matchings**, such that each matching consists of a set of independent edges without any common vertices. Therefore, the edges in the same matching do not have any feature vector dependencies and can be independently processed in parallel.

We partition the edges in each IS into matchings based on the graph theory of edge-coloring [30], which assigns "colors" to the edges of a bipartite graph such that any two adjacent edges do not have the same color. After all the edges have been colored, the edges having the same color form a matching. A classic edge-coloring algorithm has been introduced in [30]. However, this algorithm can result in small matchings, in which there are very few edges (e.g., only 1 edge). When processing such small matchings, the parallelism provided by the hardware accelerator (i.e., parallel processing units) cannot be fully utilized. Therefore, we propose a new edge-coloring algorithm which avoids small matchings. As shown in Algorithm 5, we maintain all the matchings using a singly linked list (i.e., M_List). During the edge-coloring process, the linked list keeps track of the size of each matching (i.e., the number of edges in the matching), and arranges the matchings based on their sizes in a non-descending order. When coloring an edge, we traverse the linked list and assign the edge to the first matching whose color is appropriate. Therefore, when an edge has multiple color options, the color of the matching that has the minimum number of edges is selected.

Algorithm 5 Partition edges of an IS into matchings

```
Let $M\_List$ denote a linked list of matchings
Let $M$ denote the matching being examined
Let $M.size$ denote the number of edges in the matching $M$
Let $M.color$ denote the color of the matching $M$
Let $M_{next}$ denote the next matching in $M\_List$ linked by $M$
**Partition** ($IS$)
 1: Create empty matching $M$ with $M.color$
 2: $M\_List.addFirst(M)$
 3: for each edge $e_{ij} \in IS$ do
 4:   $M \leftarrow M\_List.head$
 5:   while $M \neq$ Null do
 6:     if $u_i$ or $v_j$ has an edge colored by $M.color$ then
 7:       $M \leftarrow M_{next}$
 8:     else
 9:       Color $e_{ij}$ using $M.color$
10:       $M \leftarrow M \cup e_{ij}$
11:       $M.size \leftarrow M.size + 1$
12:       while $\exists M_{next}$ and $M_{next}.size < M.size$ do
13:         Swap $M$ and $M_{next}$ in $M\_List$
14:       end while
15:       Go to Line 3
16:     end if
17:   end while
18:   Create empty matching $M$ with $M.color$
19:   $M \leftarrow M \cup e_{ij}$
20:   $M.size \leftarrow 1$
21:   $M\_List.addFirst(M)$
22: end for
23: Return $M\_List$
```

4.2.3 Optimization 3: Edge Scheduling

The architecture of our accelerator has K parallel processing units sharing an on-chip buffer, which is organized in $2K^*$ ($K^* \geq K$) memory banks with separate banks for users and items (see Section 5.3); therefore, a batch of K edges are fed into the processing units and processed at a time. However, due to the dual-port nature of on-chip RAMs [26], [27], [29], each bank can serve only 1 read access and 1 write access per clock cycle. If there is a bank conflict between two or more accesses within a batch, the memory requests to process the edges have to be serially served. Hence, the latency (in terms of clock cycles) to resolve the bank conflict(s) within a batch is equal to the *maximum* number of accesses to the same bank within the batch. In order to reduce the bank conflicts, we develop a batching algorithm to schedule the processing of edges. As shown in Algorithm 6, the algorithm aims to partition the edges of a matching into batches, with each batch having K edges. We define a threshold value, Δ, to restrict the upper bound of bank conflicts allowed within a batch. Δ is initially set to 0. Then we sequentially traverse the edges and assign an edge into a batch if its addition does not violate the threshold condition or the batch size condition. If there are still unassigned edges after all the edges have been traversed, we increase Δ by 1 and traverse the unassigned edges again. The same procedures repeat until all the edges have been signed into a batch.

Algorithm 6 Partition edges of a matching into batches

```
Let $K$ denote the maximum number of edges that can be assigned in a batch
Let $Count\_BC(B, e)$ denote a function to count the number of edges in batch $B$ that have bank conflict with edge $e$
**Partition** ($M$)
 1: Create $b = \lceil \frac{M.size}{K} \rceil$ empty batches, $B_0, \cdots, B_{b-1}$
 2: $\Delta \leftarrow 0$
 3: while $M \neq \varnothing$ do
 4:   for each edge $e \in M$ do
 5:     for $i$ from 0 to $b-1$ do
 6:       if $B_i.size < K$ and $Count\_BC(B_i, e) \leq \Delta$ then
 7:         $B_i \leftarrow B_i \cup e$
 8:         $M \leftarrow M \backslash e$
 9:         Break
10:       end if
11:     end for
12:   end for
13:   $\Delta \leftarrow \Delta + 1$
14: end while
15: Return $B_0, \cdots, B_{b-1}$
```

5 FPGA ACCELERATOR DESIGN

5.1 Overall Architecture

The overall architecture of our FPGA accelerator is depicted in Figure 3. The external memory connected to the FPGA accelerator stores all the edges and the feature vectors of all the user and item vertices. Before an IS is processed, the feature vectors of all the vertices belonging to the IS have been stored in the Feature Vector Buffer (FVB), which is organized as memory banks of UltraRAMs (see Section 5.3). When processing an IS, FPGA fetches the edges from the external memory and stores them into a first-in-first-out Edge Queue (EQ). Whenever the EQ is not full, FPGA prefetches edges from the external memory. A batch of edges are fed into the Bank Conflict Resolver (BCR) (see Section 5.3) at a time and output in one or multiple clock cycles, such that the edges output in the same clock cycle do not result in any bank conflict accesses to the FVB. Then, the edges output by the BCR are checked by the Hazard Detection Unit (HDU) to determine whether they are data-hazard free to be processed. If an edge has no feature vector dependency with any edge being processed in the Processing Engine (PE), it is sent into the PE; otherwise, accelerator stalls occur until the dependency is resolved. The PE consists of K processing units that process distinct edges in parallel. These processing units access the feature vectors of user and item vertices from the FVB.

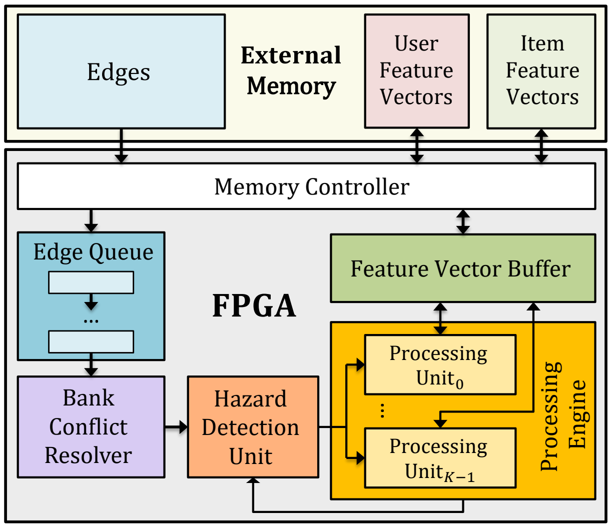

Fig. 3: Overall architecture

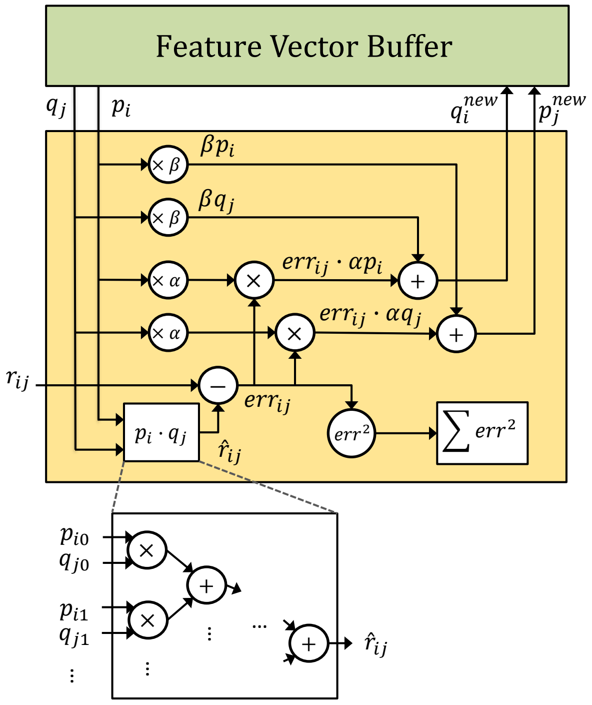

Fig. 4: Architecture of processing unit

5.2 Processing Engine

The processing engine (PE) consists of K parallel processing units that concurrently process distinct edges. We show the architecture of each processing unit in Figure 4. Each input edge is processed based on the follow steps.

1) Based on the user and item vertex indices of the edge, the processing unit reads the feature vectors (i.e., p_i and q_j) from the FVB
2) The prediction $\hat{r}_{ij}$ is computed based on p_i and q_j; meanwhile, p_i and q_j are multiplied with the constants (i.e., α and β) to obtain αp_i, αq_j, βp_i, and βq_j
3) Once the prediction error err_{ij} is obtained, p_i^{new} and q_j^{new} are computed based on Eq. (3) and (4)
4) The updated feature vectors (i.e., p_i^{new} and q_j^{new}) are written into the FVB

The dot product of p_i and q_j is computed in a binary-reduction-tree fashion, requiring H (i.e., the length of each feature vector) multipliers and $(H-1)$ adders in total. Hence, the processing unit depicted in Figure 4 contains $7H$ multipliers, $(3H-1)$ adders, 1 subtractor, 1 squarer, and 1 accumulator. This results in a peak computing throughput of $(10H+2)$ floating point operations per clock cycle. The processing unit is fully pipelined to be able to process one edge per clock cycle. We use three pipeline stages to compute each floating point operation. Hence, the pipeline depth of the processing unit is $3(\log H + 4)$.

5.3 Feature Vector Buffer

Since the K processing units of the PE need to concurrently access the Feature Vector Buffer (FVB) to read and write distinct feature vectors, there can be up to $2K$ read requests[1] and $2K$ write requests in each clock cycle. However, the native on-chip RAMs of FPGA, such as BRAMs and UltraRAMs, provide only two ports for reading and/or writing [26], [27], [29]. There are three major approaches to build multiport memory using dual-port on-chip RAMs, including multi-pumping [34], replication [26], and banking [35]. Multi-pumping gains additional ports by running the processing units with $K\times$ lower frequency than the memory. Consequently, this significantly deteriorates the clock rate of the processing units for a large K (e.g., $K = 8$) [25], [26], [27]. Replication-based approaches, such as LVT [25] and XOR [26], create replicas of all the stored data to provide additional ports and keep track of which replica has the most recently updated value for each data element. However, the amount of the RAMs needed in implementing this approach grows quadratically with the number of ports, such that $K \times K$ replicas are required to support K read ports and K write ports. Additionally, the clock rate can degrade below 100 MHz when the width and depth of the memory are large (e.g., 1Kbit $\times$ 16K) [27], [28].

In order to support large buffer capacity and sustain high clock rate, we adopt the banking approach [35] to build the multiport FVB. This approach divides the memory into equal sized banks and interleaves these banks to provide higher access bandwidth (i.e., more read and write ports). As shown in Figure 5, the FVB contains two parts of equal size, one for storing user feature vectors and the other for storing item feature vectors. Each part is divided into K^* banks ($K^* \geq K$) and each bank is implemented using a dual-port UltraRAM [29]. Therefore, the FVB provides $2K^*$ read ports and $2K^*$ write ports in total. Feature vectors

1. K for user feature vectors and K for item feature vectors

of vertices are stored into the FVB in a modular fashion based on the vertex indices, such that p_i is stored in the $(i\%K^*)^{th}$ user bank and q_j is stored in the $(j\%K^*)^{th}$ item bank. Hence, the feature vector of any user (item) can be accessed from the FVB based on the user (item) vertex index without complex index-to-address translation.

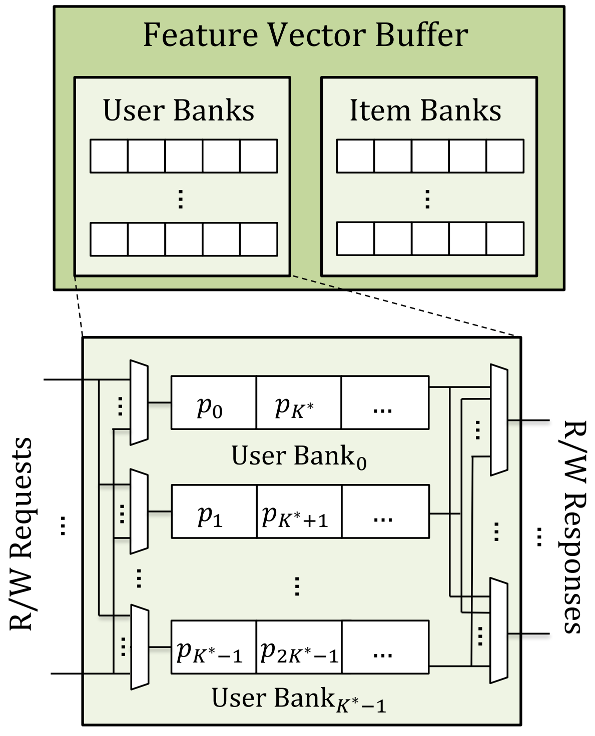

Fig. 5: Multiport FVB based on banking

However, the banked FVB cannot handle concurrent accesses to the same bank for distinct feature vectors. Such memory accesses are defined as *bank conflict* accesses, and may occur for both user and item feature vectors. To address this issue, we develop a Bank Conflict Resolver (BCR) to avoid any bank conflict accesses. As illustrated in Figure 6, the BCR fetches a batch of K edges from the Edge Queue (EQ) at a time and employs parallel detectors to detect the potential bank conflicts among the edges. Then, it outputs the edges to the Hazard Detection Unit (HDU) in one or mutliple clock cycles, such that the edges output in the same clock cycle have the feature vectors stored in distinct banks of the FVB. Therefore, concurrent accesses to the same bank will not occur. However, this also leads to additional clock cycles to resolve the bank conflicts within a batch; in the worst case, when all the edges in a batch have conflict with each other, the BCR takes K clock cycles to output all the K edges in the batch.

5.4 Hazard Detection Unit

Since the edges of distinct matchings can have common vertices, feature vector dependencies exist among the edges of distinct matchings. Therefore, when the edges output from the Bank Conflict Resolver (BCR) and the edges being processed in the Processing Engine (PE) belong to distinct matchings, read-after-write data hazards may occur. The

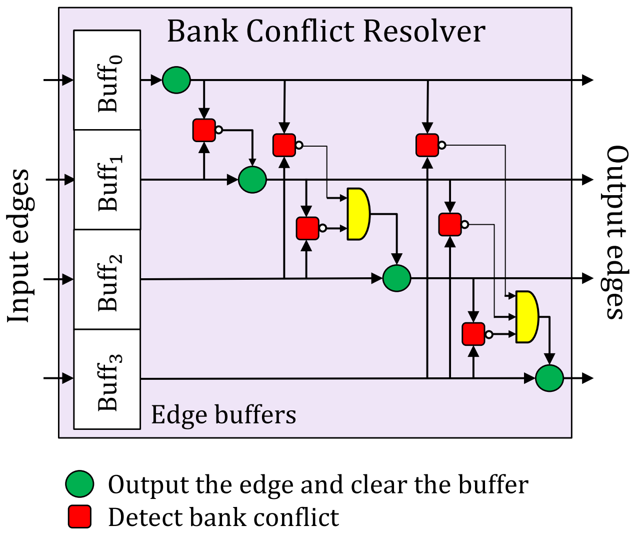

Fig. 6: Architecture of Bank Conflict Resolver for $K = 4$

Hazard Detection Unit (HDU) is responsible for detecting feature vector dependencies and preventing read-after-write data hazards. We design the HDU using BRAMs based on a fine-grained locking mechanism. As shown in Figure 7, the HDU maintains a 1-bit *lock* for each vertex of the IS being processed. A lock with value 1 means the feature vector of the corresponding vertex is being computed by the PE, and thus cannot be accessed at this time. For each input edge, the HDU checks the locks of both the user and item vertices; if both the locks are 0 (i.e., unlocked), the edge is fed into the PE and the locks are set to 1 (i.e., locked); otherwise, the HDU generates a pipeline stall signal to stall the accelerator until both the locks become 0. Note that when the PE writes an updated feature vector into the FVB, it also sends unlock signals to the HDU to set the lock of the corresponding vertex back to 0. Therefore, deadlock will not occur.

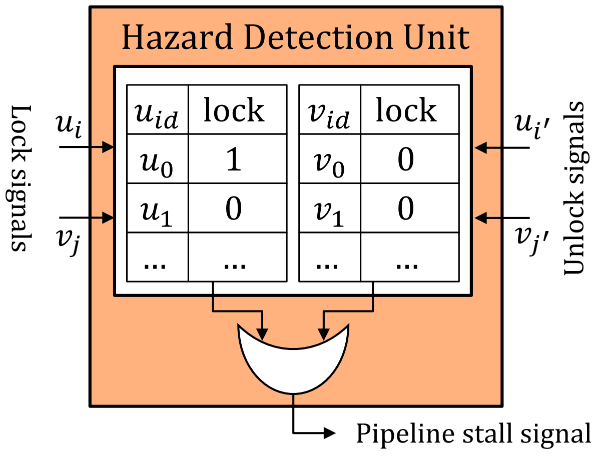

Fig. 7: Architecture of Hazard Detection Unit

6 Experimental Results

6.1 Experimental Setup

6.1.1 Platforms

We conduct experiments based on a state-of-the-art Virtex UltraScale+ xcvu9pflgb2104 FPGA [38]. This FPGA device has 1,182,240 slice LUTs, 2,364,480 slice registers, 6,840 DSPs, and up to 43.3 MB of on-chip RAMs. Two DDR4 chips are connected to the FPGA as the external memory. Each DRAM has 16 GB capacity and a peak access bandwidth of 19.2 GB/s. Our FPGA designs are implemented in RTL using `Verilog` (available at http://www-scf.usc.edu/~shijiezh/FPGA18); no high-level synthesis tool is used.

In order to show that our proposed optimizations are applicable to general purpose processors as well, we also implement a software-based design on a multi-core platform (see Section 6.6). The multi-core platform has a 16-core Intel Xeon E5-2650 processor running at 2.6 GHz. Each core has a 32 KB L1 cache and a 256 KB L2 cache. All the cores share a 20 MB L3 cache. The multi-core platform is equipped with 128 GB main memory with a peak access bandwidth of 38.4 GB/s.

6.1.2 Datasets and Machine Learning Parameters

We use several large real-life sparse matrices to evaluate our designs. They have been widely used in related works [12], [16], [22], [36]. Table 2 summaries the characteristics of the datasets. In our experiments, the length of each feature vector is 32 (i.e., $H = 32$) with each element represented using IEEE 754 single precision format. We adopt standard learning rate $\alpha = 0.0001$ and regularization parameter $\lambda = 0.02$ [2]. Note that our focus is accelerating the computations of SGD-based MF; tuning the machine learning parameters to improve the prediction accuracy is out of the scope of this paper.

TABLE 2: Large real-life matrices used for experiments

Dataset	# users ($\|U\|$)	# items ($\|V\|$)	# ratings ($\|E\|$)	Density ($\frac{\|E\|}{\|U\|\times\|V\|}$)
Libim [23]	135 K	168 K	17,359 K	7.6×10^{-4}
Netflix [2]	480 K	17 K	100,480 K	1.2×10^{-2}
Yahoo [24]	1,200 K	137 K	460,380 K	2.8×10^{-3}

6.1.3 Performance Metrics

We evaluate the performance of our designs based on the following metrics.

- Resource utilization: the percentages of basic FPGA resources utilized by the accelerator, including slice LUT, register, BRAM, UltraRAM, and DSP.
- Power: the power consumption of the FPGA accelerator, including both the static power and dynamic power.
- Execution time: the elapsed time to complete one iteration of the SGD-based MF algorithm; note that this is independent with the values of α and λ.
- Throughput: the number of floating point operations performed per second (GFLOPS).

6.2 Resource Utilization, Clock Rate, and Power Consumption

Table 3 and Table 4 show the resource utilization, clock rate, and power consumption of our FPGA accelerator for various number of processing units. The reported results are obtained through post-place-and-route simulations using Xilinx Vivado Design Suite 2018.1 [41]. For $K = 8$, the accelerator uses up to 63.9% slice LUTs in the FPGA device. Therefore, we could not increase K further to 16 due to the resource limitations. The feature vector buffer (FVB) is organized in 32 banks and the capacity of the FVB is empirically set to 64K feature vectors (32K for user vertices and 32K for item vertices). We did not increase the capacity of the FVB to 128K because we observed that the clock rate degraded below 100 MHz when 75% UltraRAMs of the FPGA device were used.

TABLE 3: Resource utilization for various number of processing units

K	LUT	Register	DSP	On-chip RAM	
				Block RAM	UltraRAM
1	7.0 %	4.6 %	3.8 %	1.2 %	37.5 %
2	14.3 %	8.2 %	7.5 %	1.2 %	37.5 %
4	30.7 %	17.2 %	15.1 %	1.2 %	37.5 %
8	63.9 %	33.4 %	30.2 %	1.2 %	37.5 %

TABLE 4: Clock rate and power consumption for various number of processing units

K	Clock Rate (MHz)	Power (Watt)
1	171	5.8
2	167	7.3
4	161	12.1
8	150	20.1

6.3 Pre-processing Time and Training Time

Table 5 and Table 6 report the pre-processing time and training time, respectively. The pre-processing includes the three proposed optimizations in Section 4, and is performed by the multi-core platform introduced in Section 6.1.1. The training is performed by our FPGA accelerator. Note that the pre-processing is performed only once, while the training is an iterative process. Therefore, the pre-processing time can be amortized and is negligible compared with the training time ($<$ 3% of the training time). In Table 6, we also report the total number of iterations, the execution time for each iteration, and the Root-Mean-Squared-Error (RMSE) (i.e., $\sqrt{\frac{\sum err^2}{|E|}}$) after the training process is completed.

TABLE 5: Pre-processing time

Dataset	Opt. 1	Opt. 2	Opt. 3	Total
Libim	0.4 sec	4.4 sec	1.6 sec	6.4 sec
Netflix	1.0 sec	10.7 sec	5.4 sec	17.1 sec
Yahoo	5.5 sec	42.3 sec	18.9 sec	66.7 sec

TABLE 6: Training time

Dataset	Total training time	# iterations to converge	T_{exec} per iteration	Root-Mean-Square-Error
Libim	360.8 sec	11,568	0.03 sec	1.05
Netflix	876.4 sec	5,766	0.15 sec	0.72
Yahoo	2536.5 sec	3,714	0.68 sec	0.94

6.4 Performance vs. Parallelism

In this section, we vary the number of processing units (K) from 1 to 8 to explore its impact on pipeline stalls, bank conflicts, and throughput performance.

6.4.1 Pipeline Stalls

Figure 8 shows *the number of pipeline stalls per edge* for various K. We compute the number of pipeline stalls per edge as the total number of pipeline stalls incurred due to feature vector dependencies divided by the total number of traversed edges; therefore a smaller value corresponds to better performance. We observe that the number of pipeline stalls per edge slightly increases as K increases. This is because when a new matching is to be processed, there can be up to $K \times D$ edges of other matching(s) remaining in the processing units, where D is the pipeline depth of each processing unit. Since the edges belonging to different matchings can have feature vector dependencies, a larger K increases the chances of pipeline stalls. We also observe that the Libim dataset has more pipeline stalls than the other two datasets. The reason is that the Libim dataset is much sparser than the other two datasets (see Table 2), thus resulting in more small matchings that cannot fill up the processing units. When multiple such small matchings are consecutively processed, the pipeline stalls due to feature vector dependencies are more likely to occur.

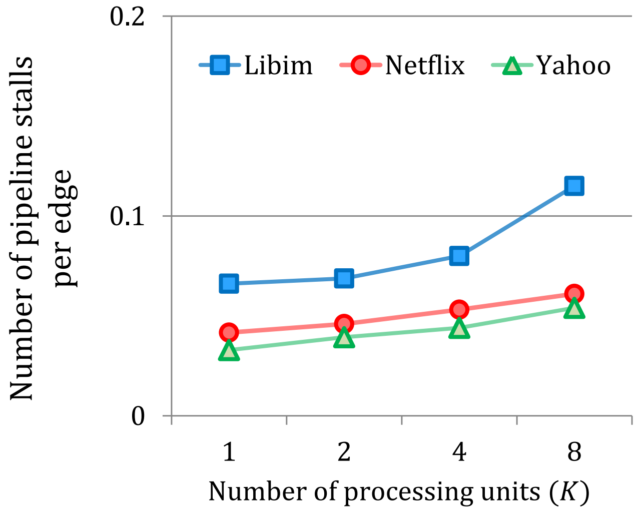

Fig. 8: Number of pipeline stalls per edge for various K

6.4.2 Bank Conflicts

Figure 9 shows *the number of bank conflicts per edge* for various K, which is computed as the total number of bank conflicts incurred during the processing divided by the total number of traversed edges. This metric indicates the average number of bank conflicts that an edge can result in; thus a smaller value is more desirable. It can be observed that the number of bank conflicts per edge significantly increases as K increases. This is because it becomes more likely that different processing units concurrently access the same bank of the feature vector buffer as K increases. Note that when $K = 1$ (i.e., there is only one processing unit), bank conflict never occurs. Another observation is that the Libim dataset has more bank conflicts per edge than the other two datasets. This is also due to the sparseness of the Libim dataset that leads to a lot of small matchings. When we perform the edge scheduling optimization (i.e., Algorithm 6) for these small matchings, it is very challenging to separate the edges that have bank conflict into distinct batches due to a small number of batches.

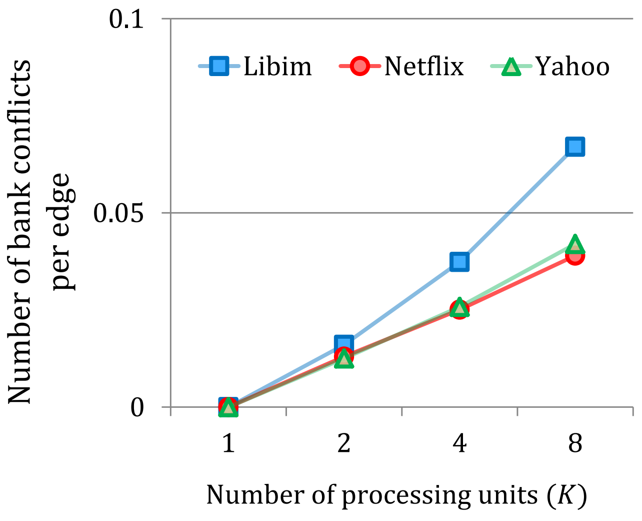

Fig. 9: Number of bank conflicts per edge for various K

6.4.3 Throughput

Figure 10 shows the throughput performance for various K. We observe that the throughput performance significantly improves as K increases for all the three datasets. For $K = 8$, our FPGA accelerator sustains a high throughput of 165 GFLOPS for Libim, 213 GFLOPS for Netflix, and 217 GFLOPS for Yahoo, respectively. We also observe that the sustained throughput for Libim is lower than Netflix and Yahoo. As explained in Sections 6.4.1 and 6.4.2, this is because the Libim dataset is much sparser than the other two datasets, resulting in more pipeline stalls and bank conflicts per edge.

6.5 Impact of the Optimizations

To show the effectiveness of our proposed three optimizations, we compare our optimized design with several non-

TABLE 7: Bank conflict reduction due to Optimization 3

Dataset	# bank conflicts per edge		Reduction	T_{exec} per iteration (sec)		Speedup
	Optimized	Baseline		Optimized	Baseline	
Libim	0.067	0.161	2.4×	0.03	0.04	1.3×
Netflix	0.039	0.166	4.2×	0.15	0.23	1.5×
Yahoo	0.042	0.164	3.9×	0.68	1.03	1.5×

TABLE 8: Pipeline stall reduction due to Optimization 2

Dataset	# pipeline stalls per edge		Reduction	T_{exec} per iteration (sec)		Speedup
	Optimized	Baseline		Optimized	Baseline	
Libim	0.115	3.313	28.8×	0.03	0.40	13.3×
Netflix	0.061	3.133	51.3×	0.15	2.19	14.6×
Yahoo	0.054	3.259	60.3×	0.68	10.45	15.4×

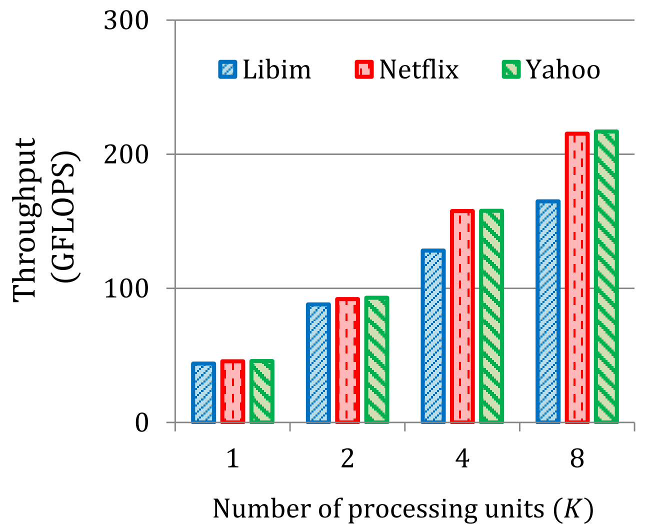

Fig. 10: Throughput for various K

optimized FPGA-based baseline designs. All the comparisons are based on $K = 8$.

6.5.1 Bank Conflict Reduction

We first explore the effectiveness of Optimization 3 (Section 4.2.3) in reducing the number of bank conflicts. Here, the baseline design used for the comparison only performs Optimization 1 and Optimization 2 during the pre-processing. Table 7 summarizes the results of the comparison. We observe that Optimization 3 reduces the number of bank conflicts by 2.4× to 4.2×. As a result, the execution time per iteration is improved by 1.3× to 1.5×.

6.5.2 Data Dependency Reduction

We further explore the impact of Optimization 2 (Section 4.2.2) which aims to reduce the number of pipeline stalls due to feature vector dependencies. For comparison purpose, the baseline design performs Optimization 1 and Optimization 3 only. Table 8 summarizes the effectiveness of this optimization. We observe that the optimized design dramatically reduces the number of bank conflicts by 28.8× to 60.3× and thus achieves 13.3× to 15.4× speedup.

6.5.3 Communication Cost Reduction

Lastly, we study the impact of Optimization 1 (Section 4.2.1) to reduce the communication cost. We define communication cost as the data transfer time between the FPGA and the external memory. The baseline design for the comparison also performs Optimization 2 and Optimization 3; however, when partitioning the input graph into induced subgraphs, the baseline design adopts the widely used vertex-index-based partitioning approach [32] (described in Section 4.2.1) rather than our proposed approach (Algorithm 4). We first compare the partitioning effect of these two approaches. Table 9 lists the maximum size, minimum size, and average size of induced subgraphs with respect to number of edges after the input graph is partitioned. It can be observed that the induced subgraphs obtained by our approach have similar amount of edges, while the induced subgraphs obtained by [32] can vary significantly in size.

TABLE 9: Comparison of partitioning effect

Dataset	Approach	$\lvert IS\rvert_{max}$	$\lvert IS\rvert_{min}$	$\lvert IS\rvert_{avg}$
Libim	This paper	591 K	524 K	579 K
	[32]	703 K	175 K	
Netflix	This paper	6,699 K	6,699 K	6,699 K
	[32]	6,929 K	4,501 K	
Yahoo	This paper	2,487 K	2,288 K	2,465 K
	[32]	3,086 K	288 K	

Table 10 summarizes the results of the comparison with respect to communication cost. For all the three datasets, the optimized design is able to completely hide the communication cost; while the baseline design can not completely hide the communication cost for Libim and Yahoo datasets. Based on the sufficient condition derived in Section 4.2.1, the required number of edges in each induced subgraph for a complete overlap of computation and communication is 438 K for Libim, 574 K for Netflix, and 574 K for Yahoo, respectively. Since the baseline design has small induced subgraphs which do not satisfy the requirement, the communication cost cannot be completely hidden.

TABLE 10: Communication cost reduction

Dataset	Unhidden communication cost per iteration (sec)		T_{exec} per iteration (sec)	
	Opt.	Base.	Opt.	Base.
Libim	0	0.005	0.031	0.036
Netflix	0	0	0.15	0.15
Yahoo	0	0.04	0.68	0.72

6.6 Performance of Multi-core Implementation

Based on our design methodology, we also implement a software-based design on the multi-core platform introduced in 6.1.1. To optimize the software-based design, we first partition the input graph into induced subgraphs based on the capacity of the last level cache (i.e., L3 cache); then, each induced subgraph is partitioned into non-overlapping matchings based on Algorithm 5. During the training process, since there is no feature vector dependency among the edges within a matching, each matching can be processed by the 8 cores of the multi-core platform in parallel without any atomic operations. The software-based design is implemented using `C` language and has been parallelized with 16 threads using multi-threading technique [45]. Table 11 reports the performance of our multi-core implementation for various datasets. We observe that our FPGA accelerator achieves an average speedup of 10× compared with our multi-core implementation.

TABLE 11: Performance of our multi-core implementation

Dataset	T_{exec} per iteration (sec)	Throughput (GFLOPS)
Libim	0.35	16.0
Netflix	1.51	21.4
Yahoo	6.55	22.6

We further compare the performance of our multi-core implementation with a highly-optimized multi-core implementation, Native [36], which has shown the fastest training speed among existing multi-core implementations [15], [37]. Native implements SGD-based MF on a 24-core Intel E5-2697 processor. It partitions the input training matrix into submatrices, and exploits submatrix-level parallelism to concurrently process the submatrices that do not have feature vector dependencies using distinct CPU cores. However, the submatrices can vary significantly in size, thus resulting in load imbalance among the CPU cores and increasing the synchronization overhead. Table 12 shows the comparison results based on the same dataset (i.e., Netflix). It can be observed that our optimized multi-core implementation achieves 1.3× speedup with 1.5× fewer CPU cores and 2.2× lower memory bandwidth.

TABLE 12: Comparison with state-of-the-art multi-core implementation for training Netflix dataset

Approach	Platform		T_{exec} per iteration	Speedup
	# cores	Mem. BW		
[36]	24	86.5 GB/s	2.0 sec	1.3×
This paper	16	38.4 GB/s	1.5 sec	

6.7 Performance Comparison with State-of-the-art GPU Implementations

In this section, we compare the performance of our FPGA accelerator with two state-of-the-art GPU implementations [22], [43]. In [22], several scheduling schemes for parallel thread execution on GPU are developed and compared. However, the lock-free static scheduling schemes are not able to efficiently exploit the thousands of cores on the GPU, and the dynamic scheduling schemes require memory locks to handle feature vector dependencies and thus result in significant synchronization overhead. In [43], the GPU design focuses on optimizing the memory performance of GPU by exploiting warp shuffling, memory coalescing, and half-precision (i.e., using 16 bits to represent a floating point number) techniques. Table 13 summaries the comparison results of our FPGA design with [22], [43] for training the same dataset (Netflix). Our design achieves 12.7× and 2.5× speedup compared with [22] and [43], respectively. Note that the speedup is achieved with fewer cores (i.e., processing units), lower clock frequency, and lower memory bandwidth. In addition, the power consumption of our FPGA accelerator is over 10× lower than the thermal design power of the GPU platforms. Therefore, from energy-efficiency perspective (i.e., performance per Watt), our FPGA design achieves even larger improvement.

7 Related Work

7.1 Multi-core-based Acceleration

Multi-core-based designs for SGD-based MF usually partition the input matrix into submatrices and use a global table shared by all the CPU threads to schedule the execution of each submatrix [40]. When a thread is idle, it locks the global table and finds an unprocessed submatrix from the table, such that the submatirx has no feature vector dependencies with the submatrices being processed by the other threads; then, the thread updates and unlocks the global table, and sequentially processes all the training data in the submatrix. There are also several multi-core-based graph-processing frameworks that support GD-based MF which can be expressed as a vertex-centric program. Representative examples include GraphLab [15] and GraphMat [16]. However, GD is less efficient than SGD when the volume of training data is large [44]. The reason is that GD needs to generate an update based on the entire training dataset, while SGD generates an update based on a single training data. Nadathur et al. observe that GD-based MF runs 40× slower than SGD-based MF for training the Netflix dataset given the same convergence criterion [36].

7.2 GPU-based Acceleration

GPUs are widely used to accelerate machine learning applications [39]. In [43], ALS-based MF and SGD-based MF are compared based on the GPU platform. The observation is that SGD-based MF results in 4× less training time than ALS-based MF. However, it has been shown that GPUs are not suitable for accelerating SGD-based MF [22], [39], [42]. The main reasons include (1) the fine-grained synchronization of updated feature vectors is very expensive on GPU platforms [39], and (2) the SIMD execution of

TABLE 13: Comparison with GPU implementations for training Netflix dataset

Approach	Platform				T_{exec} per iteration	Speedup	Throughput	Improvement
	# cores	Frequency	Mem. BW	Power				
[22]	2880	745 MHz	288 GB/s	235 W	1.90 sec	1.0×	8.6 GFLOPS	1.0×
[43]	3072	1000 MHz	360 GB/s	250 W	0.38 sec	5.0×	171.4 GFLOPS	19.9×
This paper	8	150 MHz	38 GB/s	20 W	0.15 sec	12.7×	213.3 GFLOPS	24.8×

GPU further inflates the cost of thread divergence when synchronization conflicts occur [22]. In [22], Rashid et al. compare several scheduling schemes for parallel execution of SGD on GPU, including lock-based dynamic scheduling schemes and lock-free static scheduling schemes. However, none of these schemes is able to efficiently exploit the GPU acceleration; as a result, the achieved speedup compared with a CPU implementation is very small ($< 1.1\times$). Siede et. al [42] investigate the theoretical efficiency of SGD on GPUs, and conclude that fundamental changes in the algorithm are necessary to achieve significant speedup. In [46], a variation of SGD-based MF algorithm called MSGD is proposed. In order to reduce feature vector dependencies and exploit more GPU acceleration, MSGD updates only the feature vector of either user or item per input rating (SGD updates both the feature vectors); the work [46] mathematically proves that such modification does not significantly impact the prediction accuracy.

7.3 FPGA-based Acceleration

There have not been many efforts to exploit FPGA to accelerate SGD-based MF. In [47], an FPGA accelerator for SGD-based deep learning algorithms is developed. In order to improve hardware efficiency, the accelerator is designed based on a model which allows asynchronous updates (i.e., read-after-write data hazard is acceptable) and uses low-precision computations (i.e., floating-point numbers are represented using low-precision fixed-point data type). However, it is highly likely that applying such model to SGD-based MF will negatively impact the quality of the training output. To the best of our knowledge, our proposed accelerator is the first FPGA-based design to accelerate the training process of SGD-based MF for large input matrices.

8 Generality of Our Design

Our holistic design approach is general enough to be adapted to various machine learning and parallel computing problems. The generality of our design is in four aspects. Firstly, the accelerator can be easily modified to train matrix factorization models that require tuning parameters; for example, the learning rate and regularization parameter can serve as inputs to the accelerator and be adjusted in each iteration based on the achieved accuracy improvement [2]. Secondly, the on-chip shared memory of GPU platform is also organized as memory banks and suffers bank conflict issue; our approach to reduce bank conflicts is applicable to GPU platform as well, which can enable a more efficient batching of input data and thus improve memory performance. Thirdly, our bipartite graph partitioning approach can be used to partition sparse matrices in such a way that the obtained submatrices have sufficient computations; this enables efficient load balancing when mapping the computations of submatrices to distinct parallel computing units. Lastly, the implementation of our locking mechanism can be directly used to design architecture for many applications that require to incrementally update data during processing; examples include other SGD-based training problems [47] and networking applications [48].

9 Conclusion and Future Work

In this paper, we presented an FPGA-based accelerator to speedup the training process of SGD-based MF. The accelerator consisted of parallel processing units with a shared on-chip feature vector buffer. To optimize the performance, we proposed three novel optimizations by using a bipartite graph representation of MF. Our focus was to obtain simple and fast heuristics based on identifying sufficient conditions for significant acceleration of the SGD-based MF on FPGA. By holistically considering the architectural characteristics of the FPGA platform, the proposed optimizations resulted in a complete overlap of communication and computation, up to 60.3× data dependency reduction, and 4.2× bank conflict reduction. As a result, our FPGA accelerator sustained a high throughput of up to 217 GFLOPS for training large real-life sparse datasets. Compared with the state-of-the-art GPU implementations, our FPGA accelerator achieved up to 12.7× speedup and 24.8× throughput improvement. We also demonstrated that the proposed optimizations were applicable to general purpose processors. Our optimized software-based design achieved 1.3× speedup compared with the state-of-the-art multi-core implementation.

In the future, we will explore multi-FPGA architectures, in which each FPGA device employs our FPGA accelerator, to further reduce the training time and handle even larger input matrices. We are also interested in exploring high-level synthesis tools to generate the architecture of our design and comparing the performance.

Acknowledgments

This work is supported by Intel Strategic Research Alliance funding. This work is also supported in part by the U.S. National Science Foundation under grants ACI-1339756, CNS-1643351 and OAC-1911229. Equipment grant from Xilinx, Inc. is gratefully acknowledged.

References

[1] Y. Koren, R. Bell, and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems," in *IEEE Computer*, vol. 42, iss. 8, pp. 30-37, 2009.
[2] "Netflix Update: Try This at Home," [Online.] Available: http://sifter.org/~simon/journal/20061211.html

[3] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proc. of International Symposium on Field-Programmable Gate Arrays February (FPGA)*, pp. 161-170, 2015.

[4] D. H. Noronha, P. Leong, and S. Wilton, "Kibo: An Open-Source Fixed-Point Tool-kit for Training and Inference in FPGA-Based Deep Learning Networks," in *Proc. of International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pp. 178-185, 2018.

[5] M. Blott, T. Preuer, N. J. Fraser, G. Gambardella, K. Brien, Y. Umuroglu, and M. Leeser, "Scaling Neural Network Performance through Customized Hardware Architectures on Reconfigurable Logic," in *Proc. of International Conference on Computer Design (ICCD)*, pp. 419-422, 2017.

[6] T. B. Preußer, G. Gambardella, N. J. Fraser, and M. Blott, "Inference of Quantized Neural Networks on Heterogeneous All-programmable Devices," in *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 833-838, 2018.

[7] R. Zhao, W. Song, W. Zhang, T. Xing, J. Lin, M. B. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," in *Proc. of International Symposium on Field-Programmable Gate Arrays February (FPGA)*, pp. 15-24, 2017.

[8] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. C. Herbordt, "FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters," in *Proc. of International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 81-84, 2018.

[9] S. Zhou, R. Kannan, Y. Min, and V. K. Prasanna, "FASTCF: FPGA-based accelerator for stochastic-gradient-descent-based collaborative filtering," in *Proc. of International Symposium on Field-Programmable Gate Arrays February (FPGA)*, pp. 259-268, 2018.

[10] H. F. Yu, C. J. Hsieh, H. Yun, S. Vishwanathan, and I.S.Dhillon, "A Scalable Asynchronous Distributed Algorithm for Topic Modeling," in *Proc. of International Conference on WWW*, pp 1340-1350, 2015.

[11] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global Vectors for Word Representation," In *Proc. of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532-1543, 2014.

[12] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, "Exploring the Hidden Dimension in Graph Processing," in *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[13] W. Tan, L. Cao, and L. Fong, "Faster and Cheaper: Parallelizing Large-Scale Matrix Factorization on GPUs," in *Proc. of International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pp. 219-230, 2016.

[14] H. Yu, C. Hsieh, S. Si, and I. Dhillon. "Parallel Matrix Factorization for Recommender Systems," in Journal of Knowledge and Information Systems (KAIS), pp. 793-819, 2014.

[15] "GraphLab Collaborative Filtering Library," [Online.] Available: http://select.cs.cmu.edu/code/graphlab/pmf.html

[16] N. Sundaram, N. Satish, M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High Performance Graph Analytics Made Productive," in *Proc. of VLDB Endowment*, vol. 8, no. 11, pp. 1214-1225, 2015.

[17] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics," in *Proc. of International Symposium on Microarchitecture (MICRO)*, 2016.

[18] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating Large-Scale Single-source Shortest Path on FPGA," in *Proc. of International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, 2015

[19] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," in *Proc. of International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 8-15, 2012.

[20] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and Energy-efficient Graph Processing on FPGA," in *Proc of International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 103-110, 2016.

[21] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing Memory Performance for FPGA Implementation of PageRank," in *Proc. of International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2015.

[22] R. Kaleem, S. Pai, and K. Pingali, "Stochastic Gradient Descent on GPUs," in *Proc. of Workshop on General Purpose Processing using GPUs (GPGPU)*, pp. 81-89, 2015.

[23] L. Brozovsky and V. Petricek, "Recommender System for Online Dating Service," 2007, [Online.] Available: https://pdfs.semanticscholar.org/1a42/f06f368cf9b2ba8565e81d8e048caa5c2c9e.pdf.

[24] "Ratings and Classification Data," [Online.] Available: https://webscope.sandbox.yahoo.com/catalog.php?datatype=r

[25] C. E. LaForest and J. G. Steffan, "Efficient Multi-ported Memories for FPGAs," in *Proc. of International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 41-50, 2010.

[26] C. E. Laforest, M. G. Liu, E. R. Rapati, and J. G. Steffan, "Multi-ported Memories for FPGAs via XOR," in *Proc. of International Symposium on Field-Programmable Gate Arrays February (FPGA)*, pp. 209-218, 2012.

[27] S. N. Shahrouzi and D. G. Perera, "An Efficient Embedded Multiport Memory Architecture for Next-Generation FPGAs," in *Proc. of International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 83-90, 2017.

[28] S. Zhou, R. Kannan, and V. K. Prasanna, "Accelerating low rank matrix completion on FPGA," in *Proc. of International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017.

[29] "UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices," [Online.] Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

[30] R. J. Wilson, "Introduction to Graph Theory," ISBN 0-582-24993-7, 1996.

[31] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martnez, and C. Guestrin, "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation," in *Proc. of International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 25-28, 2014.

[32] R. Pearce, M. Gokhale, and N. M. Amato, "Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates," in *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 549-559, 2014.

[33] R. Chen, J. Shi, B. Zang, and H. Guan, "Bipartite-oriented Distributed Graph Partitioning for Big Learning," in Proc. of Asia-Pacific Workshop on Systems Article (APSys), 2014.

[34] H. E. Yantir, S. Bayar, A. Yurdakul, "Efficient Implementations of Multi-pumped Multi-port Register Files in FPGAs," in *Proc. of Euromicro Conference on Digital System Design (DSD)*, pp. 185-192, 2013.

[35] J. Wawrzynek, K. Asanovic, J. Lazzaro, and Y. Lee,

"Banked Multiport Memory," [Online.] Available: https://inst.eecs.berkeley.edu/~cs250/fa10/lectures/lec08.pdf.

[36] N. Satish, N. Sundaram, M. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the Maze of Graph Analytics Frameworks using Massive Graph Datasets," in *Proc. of ACM International Conference on Management of Data (SIGMOD)*, pp. 979-990, 2014.

[37] A. Lenharth, "Parallel Programming with the Galois System," [Online.] Available: http://iss.ices.utexas.edu/projects/galois/downloads/europar2014-tutorial.pdf

[38] "Virtex UltraScale+ FPGA Data Sheet," [Online.] Available: https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf

[39] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng, "Large Scale Distributed Deep Networks," in *Proc. of Neural Information Processing Systems Conference (NIPS)*, pp. 1232-1240, 2012.

[40] W. Chin, Y. Zhuang, Y. Juan, and C. Lin, "A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems," in *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, iss. 1, 2015.

[41] "Vivado Design Suite," [Online.] Available: https://www.xilinx.com/products/design-tools/vivado.html

[42] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On Parallelizability of Stochastic Gradient Descent for Speech DNNs," in *Proc. of International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 235-239, 2014.

[43] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs," in *Proc. of International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pp. 79-92, 2017.

[44] L. Bottou, "Stochastic Gradient Descent Tricks," in *Neural Networks, Tricks of the Trade*, pp. 430-445, 2012.

[45] "OpenMP," [Online.] Available: https://computing.llnl.gov/tutorials/openMP/

[46] H. Li, K. Li, J. An, and K. Li, "MSGD: A Novel Matrix Factorization Approach for Large-Scale Collaborative Filtering Recommender Systems on GPUs," in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, pp. 1530-1544, 2018.

[47] C. D. Sa, M. Feldman, C. Re, and K. Olukotun, "Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent," in *Proc. of International Symposium on Computer Architecture (ISCA)*, pp. 561-574, 2017.

[48] D. Tong and V. K. Prasanna, "Sketch Acceleration on FPGA and its Applications in Network Anomaly Detection," in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, pp. 929-942, 2018.

Shijie Zhou obtained the BS degree in electrical engineering from Zhejiang University in 2010, and the Ph.D degree in electrical engineering from University of Southern California (USC) in 2018. He is currently working in Microsoft as a hardware engineer. His research interests include parallel computing and reconfigurable computing for accelerating graph algorithms. He has published over 25 papers in peer reviewed conference and journal proceeding. He is a member of the ACM.

Rajgopal Kannan obtained his B. Tech degree in Computer Science and Engineering from IIT Bombay in 1991 and the Ph.D. in Computer Science from the University of Denver in 1996. He is currently a Computer Scientist at the Army Research Lab in the Computing Architectures Branch and a Research Adjunct Professor in Electrical Engineering at the University of Southern California. He was formerly a Professor in the Dept. of Computer Science at Louisiana State University (2000-2015). His academic research was funded by DARPA, NSF and DOE and he has published over 150 research papers in international journals and conferences with two patents awarded in the area of network optimization. His research interests are at the intersection of Graph Analytics, Machine Learning and Edge Computing - enabling application acceleration at the edge on low power devices, for example using Software-Defined Memory for memory bound applications. He is also interested in Cyber-Physical systems, especially data-driven models and analytics driving Smartgrid optimization and control.

Viktor K. Prasanna received the BS degree in electronics engineering from the Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University. He is Charles Lee Powell Chair in Engineering and is Professor of Electrical and Computer Engineering and Professor of Computer Science with the University of Southern California (USC). He is the director of the Center for Energy Informatics (CEI) and the USC-Infosys Center for Advanced Software Technologies (CAST) at USC. He is an associate member of the Center for Applied Mathematical Sciences (CAMS) and leads the Parallel Computing/FPGA (fpga.usc.edu) and Data Science (dslab.usc.edu) Labs. The Parallel Computing/FPGA Lab is focused on innovative applications of FPGAs for accelerating computations in diverse areas while the Data Science Lab is exploring novel applications of machine learning in complex engineered systems. He was an associate director of the USC Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). His research interests include High-Performance Computing, Parallel and Distributed Systems, Reconfigurable Computing, Cloud Computing, and Smart Energy Systems. He served as the editor-in-chief of the IEEE Transactions on Computers during 2003-06. Currently, he is the editor-in-chief of the Journal of Parallel and Distributed Computing. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and the IEEE International Conference on High Performance Computing (HiPC). He received the W. Wallace McDowell Award from the IEEE Computer Society in 2015 for his contributions to reconfigurable computing. He received an Outstanding Engineering Alumnus Award from the Pennsylvania State University in 2009 and the Distinguished Alumnus Award from the University Visveswaraya College of Engineering (UVCE) of Bangalore University in 2017. He was appointed Distinguished Professor, Beihang University (BUAA), Beijing, PRC, in 2018. He received a 2019 Distinguished Alumnus Award from the Indian Institute of Science (IISc). His work on regular expression matching received one of the most significant papers in FCCM during its first 20 years award in 2013. He is a fellow of the IEEE, the ACM, and the American Association for Advancement of Science (AAAS).