FASTCF: FPGA-based Accelerator for STochastic-Gradient-Descent-based Collaborative Filtering

Shijie Zhou
University of Southern California
Los Angeles, CA 90089
shijiezh@usc.edu

Rajgopal Kannan
US Army Research Lab
Los Angeles, CA 90094
Rajgopal.kannan.civ@mail.mil

Yu Min
University of Southern California
Los Angeles, CA 90089
yumin@usc.edu

Viktor K. Prasanna
University of Southern California
Los Angeles, CA 90089
prasanna@usc.edu

ABSTRACT

Sparse matrix factorization using Stochastic Gradient Descent (SGD) is a popular technique for deriving latent features from observations. SGD is widely used for Collaborative Filtering (CF), itself a well-known machine learning technique for recommender systems. In this paper, we develop an FPGA-based accelerator, FASTCF, to accelerate the SGD-based CF algorithm. FASTCF consists of parallel, pipelined processing units which concurrently process distinct user ratings by accessing a shared on-chip buffer. We design FASTCF through a holistic analysis of the specific design challenges for the acceleration of SGD-based CF on FPGA. Based on our analysis of these design challenges, we develop a bipartite graph processing approach with a novel 3-level hierarchical partitioning scheme that enables conflict-minimizing scheduling and processing of on-chip feature vector data to significantly accelerate the processing of this bipartite graph. First, we develop a fast heuristic to partition the input graph into induced subgraphs; this enables FASTCF to efficiently buffer vertex data for reuse and completely hide communication overhead. Second, we partition all the edges of each subgraph into matchings to extract the maximum parallelism. Third, we schedule the execution of the edges inside each matching to reduce concurrent memory access conflicts to the shared on-chip buffer. Compared with non-optimized baseline designs, the hierarchical partitioning approach results in up to 60× data dependency reduction, 4.2× bank conflict reduction, and 15.4× speedup. We implement FASTCF based on state-of-the-art FPGA and evaluate its performance using three large real-life datasets. Experimental results show that FASTCF sustains a high throughput of up to 217 billion floating-point operations per second (GFLOPS). Compared with state-of-the-art multi-core and GPU implementations, FASTCF demonstrates 13.3× and 12.7× speedup, respectively.

FPGA '18, February 25–27, 2018, Monterey, CA, USA

ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174252

KEYWORDS

Sparse matrix factorization; Training process; Bipartite graph representation

ACM Reference Format:
Shijie Zhou, Rajgopal Kannan, Yu Min, and Viktor K. Prasanna. 2018. FASTCF: FPGA-based Accelerator for STochastic-Gradient-Descent-based Collaborative Filtering. In *FPGA '18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174252

1 INTRODUCTION

Web-based services such as online shopping and social media have become extremely popular Internet services but also difficult to use effectively due to the surfeit of information available on the web. In order to provide accurate recommendations and enhance user satisfaction, many online companies such as Amazon, Netflix, and Facebook develop recommender systems [1, 2]. These systems analyze the patterns of user's interest in products and provide personalized recommendations that suit a user's taste. Collaborative Filtering (CF) is a widely used machine learning technique to design such recommender systems [1].

Sparse matrix factorization is an unsupervised machine learning technique to extract latent information from observations [1]. Stochastic Gradient Descent (SGD) is widely used to train the matrix factorization model for many applications, for which CF is a representative example. This approach has achieved the highest prediction accuracy in the Netflix challenge [2], which gives a partially observed rating matrix and asks to predict the missing ratings. In addition, SGD-based CF is adopted in many commercial recommender systems [1–4]. However, the training process of SGD-based CF algorithms is computation-intensive because the model needs to be iteratively updated for thousands of iterations [5]. When the volume of training data is huge, training time can become excessively long. Therefore, it becomes essential to develop hardware-based accelerators to reduce the training time.

Recently, there has been a growing interest in employing FPGA to accelerate machine learning techniques [6–10]. In this paper, we propose FASTCF, a high-throughput accelerator based on state-of-the-art FPGA to accelerate the training process of a popular SGD-based CF algorithm. FASTCF consists of parallel processing units concurrently working on distinct input data to sustain high throughput. On-chip buffers that store the feature data (vectors)

of users and items feed the pipleines and are exploited for data reuse. The proposed design is also applicable to accelerating other applications that use matrix factorization to derive hidden features from observations (e.g., text mining [11]).

Our design of FASTCF is holistic and generalized: It is motivated by a careful analysis of the challenges involved in accelerating SGD-based CF on FPGA. We identify three principal design challenges 1) limited on-chip memory which can limit throughput if the long latencies of external memory accesses are not managed 2) data dependencies among feature vectors which can prevent concurrent processing within the FPGA pipeline and 3) pipeline stalls due to access conflicts between different pipelines when accessing single R/W ported on-chip memory banks. (Note that multi-ported memory banks [25, 26] can solve the access conflict problem but we must pay a significant memory capacity penalty to do so. Multi-port banks require significantly larger on-chip memory capacity for solving a similar sized problem due to quadratic factor duplication (Sec 5.3)). 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. This is followed by a novel 3-level hierarchical partitioning scheme that enables conflict-minimizing scheduling and processing of on-chip feature vector data to significantly accelerate the processing of this bipartite graph. We handle the specific design challenges listed above through the following techniques:

- To overcome the first challenge, FASTCF first partitions the input 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 feature vectors of vertices in the on-chip buffer. By overlapping the communication overhead with computation, FASTCF 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×, and thus enables FASTCF to efficiently process distinct edges in parallel.
- To overcome the bank-conflict challenge, we develop a greedy algorithm to partition each matching into batches and schedule the execution of the batches to reduce conflicts due to concurrent accesses to the shared on-chip buffer (i.e., bank conflicts). This optimization results in up to 4.2× bank conflict reduction.

Experimental results show that FASTCF sustains high throughput of up to 217 GFLOPS for training. Compared with a state-of-the-art multi-core implementation running on a 24-core Intel Xeon processor, FASTCF achieves 13.3× speedup. Compared with a state-of-the-art GPU implementation running on a 2880-core Tesla K40C, FASTCF attains 12.7× speedup.

The rest of the paper is organized as follows. Section 2 covers the background; Section 3 introduces the SGD-based CF algorithm and the challenges in accelerating it; Section 4 presents our 3-level hierarchical partitioning approach; Section 5 describes the architecture of FASTCF; Section 6 reports the experimental results; Section 7 discusses the related work; Section 8 concludes the paper.

2 BACKGROUND

CF relies on existing user ratings to predict the ratings that have not been given [5]. By collecting and analyzing past rating information from many users (collaborating), CF identifies new user-item associations and makes predictions (filtering). Most of the CF algorithms fall into two categories, namely memory-based and model-based algorithms [1, 5].

Memory-based CF algorithms use user ratings to compute the similarity between users or alternatively, between items. Several similarity metrics, such as Pearson correlation and Cosine similarity [6], can be used to compute the similarity. Once a matrix of similarities is formed, the prediction of a particular user's rating of an item is made based on similar users (i.e., users that have high similarity with this user) or similar items. Although memory-based CF algorithms are simple and fast, they cannot efficiently handle sparse datasets [5]. In addition, their prediction performance is not as accurate as model-based CF algorithms [5].

Model-based CF algorithms aim to develop a model of user ratings using machine learning techniques. After the model is obtained, model-based CF algorithms produce the prediction of a user's rating by computing the expected value of the rating in the model. Matrix factorization model, which is also called latent factor model [3], has shown great success to achieve high prediction accuracy for CF, and is widely adopted in recommender systems [1, 5, 14]. Stochastic Gradient Descent (SGD) and Alternate Least Square (ALS) are two primary methods to perform matrix factorization for CF [1, 13, 14]. ALS can converge in fewer iterations than SGD, but ALS is hardly scalable to large-scale datasets due to its cubic time complexity in each iteration [14]. In this paper, we focus on accelerating the SGD-based CF algorithm.

There are also hybrid approaches to combine CF technique with other recommendation techniques (e.g., content-based recommender [5]). Such hybrid CF algorithms can overcome the problems of native CF such as loss of information. However, the complexity and expense for the implementation are significantly increased as well [5].

3 ALGORITHM AND CHALLENGES

In this section, we briefly introduce the SGD-based CF algorithm and discuss the challenges in accelerating it.

3.1 SGD-based CF

Let U and V denote a set of users and items, $|U|$ and $|V|$ denote the number of users and items, respectively. The **input** training dataset is a partially filled rating matrix $R = \{r_{ij}\}_{|U|\times|V|}$, in which r_{ij} represents the rating of item v_j given by user u_i ($0 \le i < |U|$, $0 \le j < |V|$).

Assuming each user and item is associated with H latent features[1], the **output** model of the training process contains two matrices, P (a $|U| \times H$ matrix) and Q (a $|V| \times H$ matrix), such that their product approximates R (i.e., $R \approx P \times Q^T$). P and Q are called user feature matrix and item feature matrix, respectively. 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.

[1] A typical value of H is 32 [15–17].

The prediction of the rating of item v_j by user u_i is the dot product of p_i and q_j:

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

Given a known 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 regularized squared error based on all the known ratings:

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

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

$$p_i^{new} = \beta \cdot p_i + err_{ij} \cdot \alpha \cdot q_j \quad (3)$$

$$q_j^{new} = \beta \cdot q_j + err_{ij} \cdot \alpha \cdot p_i \quad (4)$$

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**; therefore, the ratings of the same item or given by the same user cannot be concurrently processed because they will result in the updates for the same q_j or p_i. Additional details of this algorithm can be found in [1, 4].

3.2 Challenges

There are three challenges in accelerating the SGD-based CF algorithm using FPGA.

First, since the feature vectors of users and items are repeatedly accessed and updated during the processing of ratings, it is desirable to store them in the on-chip memory of FPGA. 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 latency, which results in accelerator pipeline stalls and even no speedup [19].

Second, data dependencies exist among ratings, making it challenging to efficiently exploit the massive parallelism of FPGA for concurrent processing. More specifically, the ratings of the same item or given by the same user cannot be processed concurrently. This is because SGD requires to incrementally update feature vectors once per rating; concurrent processing of such ratings can lead to read-after-write data hazard. We define such data dependency among ratings as **feature vector dependency**.

Third, FPGA accelerators usually employ parallel processing units to increase processing throughput [8, 19, 20]. However, the on-chip RAMs (e.g., block RAM and UltraRAM) of FPGA support only dual-port accesses (one read port and/or one write port) [25–27]. When multiple processing units need concurrent accesses to 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 and thus performance deterioration.

In order to overcome these challenges, we use a bipartite graph representation of CF (Section 4.1) and propose a 3-level hierarchical partitioning approach (Section 4.2).

4 GRAPH REPRESENTATION AND HIERARCHICAL PARTITIONING

4.1 Graph Representation

We transform SGD-based CF into a bipartite graph-processing problem so that graph theories can be leveraged to optimize the performance. The input rating matrix is converted into a bipartite graph G, whose vertices can be divided into two disjoint sets, U (user vertices) and V (item vertices). Each known rating in R is represented as an edge connecting a user vertex and an item vertex in G. We store G in the coordinate (COO) format [22], which is a commonly used graph representation [16–18, 22–24]. This format stores the graph as an edge list E; each edge 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, and r_{ij} corresponds to the rating value of v_j given by u_i. Algorithm 1 illustrates the SGD-based CF using bipartite graph representation. Each vertex maintains a feature vector of length H. 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.

Algorithm 1 SGD-based CF using graph representation

Let p_i denote the feature vector of user vertex u_i $(0 \le i < |U|)$
Let q_j denote the feature vector of item vertex v_j $(0 \le j < |V|)$
Let $edge_{ij}$ denote the edge connecting u_i and v_j
CF_Train $(G(U, V, E))$

```
1:  for each user/item vertex do
2:      Randomly initialize its feature vector
3:  end for
4:  while Overall_squared_error_converages = false do
5:      Overall_squared_error= 0
6:      for each edge_ij ∈ E do
7:          Read feature vectors p_i and q_j
8:          Compute r̂_ij based on Eq. (1)
9:          Compute err_ij based on r_ij and r̂_ij
10:         Update p_i and q_j based on Eq. (3) and (4)
11:         Overall_squared_error+ = err_ij^2
12:     end for
13: end while
14: Return all the feature vectors of vertices
```

4.2 3-Level Hierarchical Partitioning

4.2.1 First-Level Partitioning: On-chip Buffering and Communication Hiding. In order to address the first challenge described in Section 3.2, we partition G into induced subgraphs to achieve two goals: (1) the feature vectors of the vertices in each induced subgraph can fit in the on-chip buffer; (2) the computation for

processing each induced subgraph can completely hide the communication cost.

Let L (N) denote the on-chip buffer 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 the details to partition U and V in Algorithm 3. Let E_{xy} denote an subset of E that consists of all the edges connecting the vertices belonging to U_x and V_y in G ($0 \le x < l, 0 \le y < n$). Then U_x, V_y, and E_{xy} form an **Induced Subgraph (*IS*)** of G [28]. Since each user (item) vertex subset has no more than L (N) vertices, the feature vectors of all the vertices in each *IS* can fit in the on-chip buffer.

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, all the induced subgraphs are sequentially processed by FASTCF based on Algorithm 2. 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 into the on-chip buffer; therefore, the processing units of FASTCF can directly access the feature vectors from the on-chip buffer.

Algorithm 2 Scheduling of induced subgraph processing

```
1:  while Overall_squared_error_converages = 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 ∈ 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 [8, 29], we can pipeline the processing of induced subgraphs while overlapping communication and computation of each *IS* with its predecessor/successor. Let $P(IS_z)$ denote the computation time to process all the edges of an induced subgraph IS_z; let T_z denote the *intra-subgraph* communication time (i.e., the total time for data transfers occurring during the processing of IS_z). As shown in Figure 1, we can pipeline the processing of induced subgraphs by overlapping the computation time $P(IS_z)$ of IS_z with the *writing* of feature vectors from IS_{z-1} and the *reading* of feature vectors from IS_{z+1}. Therefore, $T_z = T^{wr}_{z-1} + T^{rd}_{z+1}$. Here, T_z in general may include reads or writes of both user and item feature vectors. We can easily derive the **sufficient condition** for complete overlap of communication and computation: $P(IS_z) \ge T_z, \forall z \in [0, l \times n)$.

A vertex-index-based partitioning approach [30] has been widely used to perform graph partitioning for hardware accelerators [17, 20, 21]. This approach simply assigns a group of vertices with contiguous indices to each vertex subset. Although this approach is fast, it can lead to significant data imbalance such that some subgraphs may have very few edges; in this scenario, the communication cost cannot be completely hidden by the computation. Therefore, a desirable partitioning approach should balance the number of

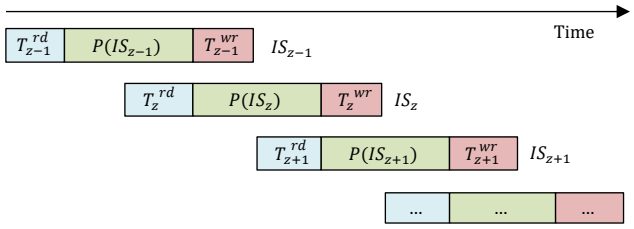

Figure 1: Pipelined induced subgraph processing

edges among the induced subgraphs. Many sophisticated graph partitioning approaches have been developed to achieve balanced partitioning and simultaneously minimize some other metrics [31]. For example, vertex-cut algorithm [31] balances the number of edges among subgraphs and meanwhile minimizes the number of vertex replicas in distributed computing environment. 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 is acceptable as long as the derived sufficient condition is satisfied.

We propose a simple and fast heuristic partitioning approach. Our empirical results show that this approach leads to the balanced partitioning such that each *IS* has sufficient edges for the computation to completely hide the communication cost. 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 subset sizes are bound by L and N as defined earlier. Algorithm 3 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 non-increasing order. Then we greedily assign each vertex into the vertex subset which has not been full and has the minimum subset degree, until all the vertices are assigned, subject to the subset size condition.

When each vertex is assigned to a vertex subset, we assign a new vertex index to it (Algorithm 3, 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 Second-Level Partitioning: Data Dependency Reduction. The second-level partitioning addresses the second challenge described in Section 3.2. Note that the edges having the same user vertex or item vertex cannot be concurrently processed due to the feature vector dependencies. We partition the edges in each *IS* into set of **matchings**, such that each matching consists of a set of

Algorithm 3 Partition U into l subsets $U_0, \cdots, U_{l-1}$

Let $u_{i \cdot degree}$ be the number of edges connected to u_i $(0 \le i < |U|)$
Let $U_{x \cdot size}$ be the number of vertices in U_x $(0 \le x < l)$
Let $U_{x \cdot degree}$ be the subset degree of U_x $(\sum u_{i \cdot degree}, \forall u_i \in U_x)$
Partition (U, L, l)

1: **for** x from 0 to $l-1$ **do**
2: $U_x = \varnothing$
3: $U_{x \cdot degree} = 0$
4: $U_{x \cdot size} = 0$
5: **end for**
6: Sort U based on vertex degree in descending order
7: **for** each $u_i \in U$ **do**
8: $subset\_id = -1$
9: $min\_degree = |E|$
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: $subset\_id = x$
13: $min\_degree = U_{x \cdot degree}$
14: **end if**
15: **end for**
16: $U_{subset\_id} = U_{subset\_id} \cup u_i$
17: $U_{subset\_id \cdot degree} = U_{subset\_id \cdot degree} + u_{i \cdot degree}$
18: $u_{i \cdot new\_user\_id} = subset\_id \times L + U_{subset\_id \cdot size}$
19: $U_{subset\_id \cdot size} = U_{subset\_id \cdot size} + 1$
20: **end for**
21: **Return** $U_0, \cdots, U_{l-1}$

vertex-disjoint edges. As a result, the edges in the same matching do not have any feature vector dependencies and can be processed in parallel.

We perform the second-level partitioning by using edge-coloring [28], which colors all the edges of a bipartite graph such that any two adjacent edges do not have the same color. After all the edges are colored, the edges having the same color form a matching. However, the classic edge-coloring algorithm in [28] 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) is not fully utilized. We modify the edge-coloring algorithm by keeping track of the number of edges in each matching during the partitioning; when an edge can be partitioned into multiple matchings, we select the matching having the minimum number of edges.

4.2.3 Third-Level Partitioning: Bank Conflict Reduction. The architecture of the accelerator has M parallel processing units sharing an on-chip buffer, which is organized in $2M^*$ ($M^* \ge M$) banks with separate banks for users and items (see Section 5.3); therefore, a batch of M edges from a matching can be concurrently processed at a time. However, due to the dual-port nature of on-chip RAM [25–27], each bank can support only 1 read and 1 write request 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. Thus the latency for resolving the bank conflict(s) within a batch is equal to the *maximum* number of accesses to the same bank within the batch. We develop the following greedy heuristic for reducing the bank conflicts. We sort all the edges in a matching in non-increasing order of their bank conflict index (BCI). The BCI of an edge is defined as the number of other edges in the matching that have bank conflict with this edge. We partition each matching into batches of size M by sequentially traversing the sorted edges. We greedily assign an edge to the first batch where its addition *does not increase* the current latency to resolve the bank conflicts of the batch. Note that this is different from assigning the edge to the batch where it has the minimum bank conflict.

5 ACCELERATOR DESIGN

5.1 Overall Architecture

The overall architecture of FASTCF is depicted in Figure 2. As shown, two DRAM chips, $DRAM_0$ and $DRAM_1$, are connected to FPGA as the external memory. $DRAM_0$ stores all the edges and $DRAM_1$ stores the feature vectors of all the vertices, respectively. When processing an *IS*, the feature vectors of all the vertices belonging to the *IS* are read from $DRAM_1$ and stored in the Feature Vector Buffer (FVB), which is organized as banks of UltraRAM. FPGA fetches edges from $DRAM_0$ and stores them into a first-in-first-out Edge Queue (EQ). Whenever the EQ is not full, FPGA pre-fetches edges from $DRAM_0$. A batch of edges are fed into the Bank Conflict Resolver (BCR) at a time and output in one or multiple clock cycles, such that the edges output in the same clock cycle do not have bank conflict accesses to the FVB. Then, each edge is checked by the Hazard Detection Unit (HDU) to determine whether it is 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, pipeline stalls occur until the dependency is resolved. The PE consists of multiple processing units that process distinct edges in parallel. These processing units access the feature vectors of vertices from the FVB.

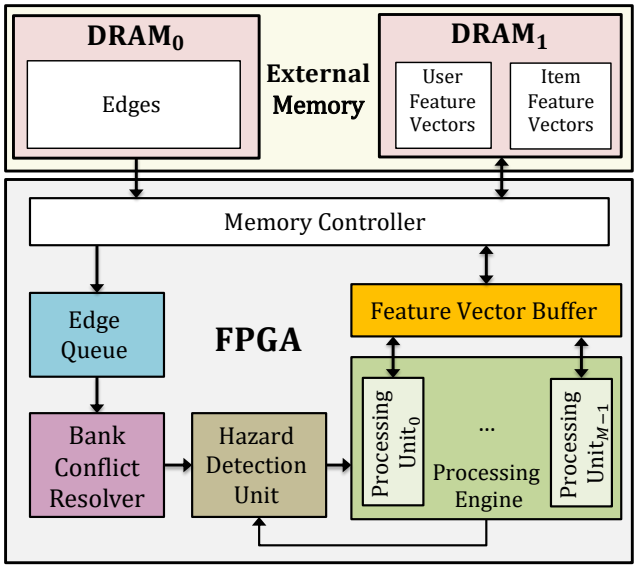

Figure 2: Overall architecture

5.2 Processing Engine

The processing engine (PE) consists of M parallel processing units that concurrently process distinct edges. We show the architecture of each processing unit in Figure 3. Each input edge is processed as follows: based on the user and item vertex indices, the processing unit reads the feature vectors, p_i and q_j, from the FVB; then, 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; once the prediction error err_{ij} is obtained, p_i^{new} and q_j^{new} are computed based on Eq. (3) and (4); finally, p_i^{new} and q_j^{new} are written into the FVB.

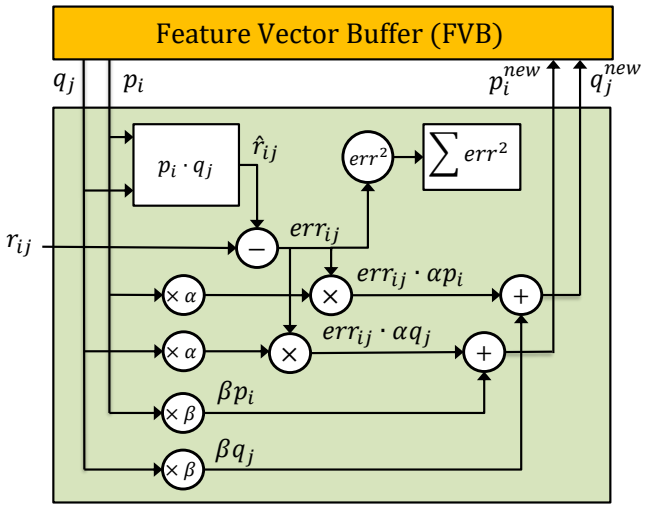

Figure 3: Architecture of processing unit

The dot product of p_i and q_j is computed in a binary-reduction-tree fashion [32], requiring H multipliers and $(H-1)$ adders in total. Hence, each processing unit contains $7H$ multipliers, $(3H-1)$ adders, 1 subtractor, 1 squarer, and 1 accumulator, sustaining a peak throughput of $(10H+2)$ floating point operations per clock cycle. The processing unit is fully pipelined so that one edge can be processed 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 M processing units need concurrently access the Feature Vector Buffer (FVB) to read and write distinct feature vectors, there can be up to $2M$ read requests[2] and $2M$ write requests in each clock cycle. However, native on-chip RAMs of FPGA provide only two ports for reading and/or writing [25–27]. There are three major approaches to build multiport memory using dual-port on-chip RAMs, including multi-pumping [33], replication [25], and banking [34]. Multi-pumping gains ports by running the processing units with $M\times$ lower frequency than the multiport memory. However, this can significantly deteriorate the clock rate of the processing units for a large M (e.g., M=8) [25, 26]. Replication-based approaches, such as Live Value Table (LVT) and XOR [25], 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 [26]. However, the size of the RAM needed in implementing this grows quadratically with the number of ports, such that $M \times M$ replicas are required to support M read ports and M write ports. Additionally, the clock rate can degrade below 100 MHz when the width and depth of the memory are large (e.g., 1Kbit × 16K) [26].

In order to support large buffer capacity and sustain high clock rate, FASTCF adopts the banking approach [34] to build the 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 illustrated in Figure 4, the banked 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 M^* banks ($M^* \geq M$) and each bank is a dual-port UltraRAM [27]. Therefore, the FVB provides up to $2M^*$ read ports and $2M^*$ write ports. 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 $(i\%M^*)$-th user bank and q_j is stored in the $(j\%M^*)$-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.

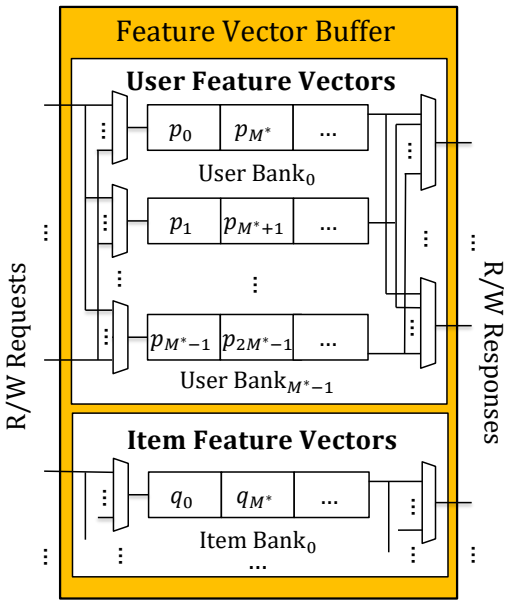

Figure 4: Banked FVB

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. To address this issue, we develop a Bank Conflict Resolver (BCR) to avoid any bank conflict accesses. The BCR fetches a batch of M edges at a time and outputs them to the Hazard Detection Unit (HDU). The BCR ensures that all the edges output in the same clock cycle have the feature vectors of their vertices stored in distinct banks of the FVB. However, this can lead to additional clock cycles to resolve the bank conflicts within a

[2] M for user feature vectors and M for item feature vectors

batch; in the worst case, when all the edges in a batch have conflict with each other, the BCR takes M clock cycles to output all the edges in the batch.

5.4 Hazard Detection Unit

When the edges from the Bank Conflict Resolver and the edges being processed in the PE belong to different matchings, read-after-write data hazards due to feature vector dependencies may occur. The Hazard Detection Unit (HDU) is constructed by BRAMs and responsible for detecting feature vector dependencies and preventing read-after-write data hazards. We design the HDU using a fine-grained locking mechanism. For each vertex, we assign a 1-bit flag. A flag with value 1 means the feature vector of the corresponding vertex is being computed by the PE, and thus cannot be accessed. For each input edge, the HDU checks the flags of its user and item vertices; if both the flags are 0, the edge is fed into the PE and the flags are set to 1; otherwise, the pipeline stalls until both the flags become 0. When the PE writes any updated feature vector into the FVB, it also sends signals to the HDU to set the flag of the corresponding vertex back to 0. Therefore, deadlock will not occur.

6 EXPERIMENTAL RESULTS

6.1 Experimental Setup

Our FPGA designs are implemented on a state-of-the-art Virtex UltraScale+ xcvu9pflgb2104 FPGA [37]. The target 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 RAM. The FPGA uses two DDR4 chips as the external memory. Each DRAM has 16 GB capacity and a peak bandwidth of 19.2 GB/s. The host CPU is an 8-core Intel Xeon E5-2686 processor. Each core of the host CPU runs at 2.3 GHz and has a 32 KB L1 cache and a 256 KB L2 cache. All the cores share a 45 MB L3 cache. The host CPU and the FPGA are connected through PCIe 3.0×16 bus.

We use large real-life datasets (Table 1) to evaluate our designs. These datasets have been widely used in related works [12, 16, 22, 35]. 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 a standard learning rate $\alpha = 0.0001$ and regularization parameter $\lambda = 0.02$ [4]. We use execution time and throughput (sustained floating point operations per second (GFLOPS)) as our performance metrics.

Table 1: Large real-life datasets used for experiments

Dataset	# users $\|U\|$	# items $\|V\|$	# ratings $\|E\|$	Description
Libimseti [23]	135 K	168 K	17,359 K	Dating ratings
Netflix [4]	480 K	17 K	100,480 K	Movie ratings
Yahoo [24]	1,200 K	136 K	460,380 K	Music ratings

6.2 Resource Utilization, Clock Rate, and Power Consumption

Table 2 shows the resource utilization, clock rate, and power consumption of FASTCF for $M = 8$ (i.e., the number of processing units = 8). The reported results are post-place-and-route results evaluated by Xilinx Vivado Design Suite 2017.2. For $M = 8$, FASTCF uses up to 58.9% slice LUTs and 63.0% DSPs in the FPGA device. Therefore, we could not increase M 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 to 85 MHz when 75% UltraRAMs of the FPGA device were used.

Table 2: Resource utilization, clock rate, and power consumption of FASTCF

Slice LUT (%)	Register (%)	DSP (%)	On-chip RAM (%)	
			Block RAM	UltraRAM
58.9	27.1	63.0	1.2	37.5
Clock rate (MHz)		Power (Watt)		
150		13.8		

6.3 Pre-processing Time and Training Time

Table 3 and Table 4 report the pre-processing time and training time, respectively. The pre-processing is performed by the host CPU based on our proposed 3-level partitioning approach; the training is performed by FASTCF. Note that the pre-processing is performed only once, while the training is an iterative process; thus, the pre-processing time can be amortized and is negligible compared with the total training time. In Table 4, we also report the total training time and the average execution time for each iteration.

Table 3: Pre-processing time

Dataset	1st-level	2nd-level	3rd-level	Total
Libimseti	0.4 sec	4.4 sec	2.7 sec	7.5 sec
Netflix	1.0 sec	10.7 sec	7.0 sec	18.7 sec
Yahoo	5.5 sec	42.3 sec	23.0 sec	70.8 sec

Table 4: Training time

Dataset	Total training time	# iterations to converge	Avg. T_{exec} per iteration
Libimseti	360.8 sec	11,568	0.03 sec
Netflix	876.4 sec	5,766	0.15 sec
Yahoo	2536.5 sec	3,714	0.68 sec

6.4 Performance vs. Parallelism

To explore the impact of parallelism on the performance, we vary the number of processing units (M) from 1 to 8. Figure 5 shows the throughput performance for various M. We observe that the throughput performance significantly improves as M increases for all the three datasets. For $M = 8$, FASTCF sustains 165 GFLOPS

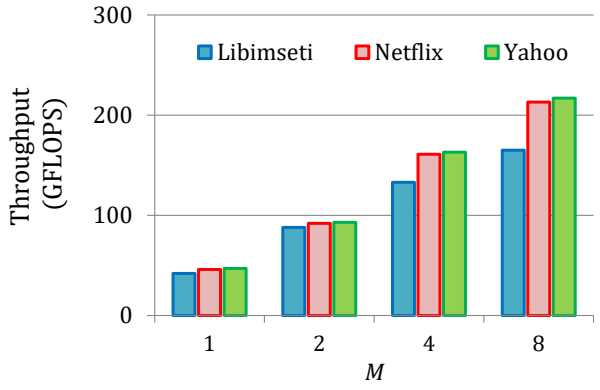

Figure 5: Throughput performance for various M

for Libimseti, 213 GFLOPS for Netflix, and 217 GFLOPS for Yahoo, respectively. However, we also observe that the number of pipeline stalls due to feature vector dependencies increases as M increases. This is because when a matching is to be processed, there can be up to $M \times D$ edges of other matching(s) remaining in the M processing units, where D is the pipeline depth of each processing unit. Thus, a larger M increases the chances of feature vector dependencies. It can also be observed that the throughput performance of Libimseti is worse than Netflix and Yahoo for $M = 4$ and 8. This is because the Libimseti dataset is much sparser than the other two datasets, thus resulting in more small matchings that cannot fill up the processing units. Since the edges belonging to different matchings can have common vertices, when multiple small matchings are consecutively processed, the pipeline stalls due to feature vector dependencies are more likely to occur.

6.5 Impact of the Optimizations

To show the effectiveness of our proposed 3-level hierarchical partitioning approach, we compare our optimized design with non-optimized FPGA-based baseline designs. All the comparisons are based on $M = 8$.

6.5.1 Bank Conflict Reduction. We first explore the effectiveness of the third-level partitioning in reducing the number of bank conflicts. Here, the baseline design used for the comparison only performs the first-level and second-level partitionings during the pre-processing. Table 5 summarizes the results of the comparison. We observe that our optimized design reduces the number of bank conflicts by 2.4× to 4.2× and thus results in 1.3× to 1.5× speedup.

Table 5: Bank conflict reduction

Dataset	# clocks to resolve bank conflicts per iteration		Reduction	T_{exec} per iteration (sec)		Speedup
	Opt.	Base.		Opt.	Base.	
Libim.	1,165 K	2,798 K	2.4×	0.03	0.04	1.3×
Netflix	3,960 K	16,686 K	4.2×	0.15	0.23	1.5×
Yahoo	19,393 K	75,524 K	3.9×	0.68	1.03	1.5×

6.5.2 Data Dependency Reduction. We further explore the impact of the second-level partitioning to reduce the number of pipeline stalls due to feature vector dependencies. The baseline design used for the comparison performs the first-level and third-level partitionings only. Table 6 summarizes the effectiveness of this optimization. We observe that the optimization dramatically reduces the number of pipeline stalls due to feature vector dependencies by 28.7× to 60.1×; as a result, the execution time per iteration is reduced by 13.3× to 15.4×.

Table 6: Pipeline stall reduction

Dataset	# stalls due to dependencies per iteration		Reduction	T_{exec} per iteration (sec)		Speedup
	Opt.	Base.		Opt.	Base.	
Libim.	2,005 K	57,524 K	28.7×	0.03	0.40	13.3×
Netflix	6,151 K	314,884 K	51.2×	0.15	2.19	14.6×
Yahoo	24,954 K	1,500,295 K	60.1×	0.68	10.45	15.4×

6.5.3 Communication Cost Reduction. Lastly, we study the impact of the first-level partitioning to reduce the communication cost. We define communication cost as the data transfer time between the FPGA and the external memory. Here, the baseline design also performs all the three levels of partitionings, but the first-level partitioning is based on the simple vertex-index-based partitioning approach [30] (described in Section 4.2.1) rather than our proposed approach (Algorithm 3). Table 7 summarizes the results of the comparison. For all the three datasets, the optimized design is able to completely hide the communication cost; while the baseline design cannot completely hide the communication cost for Libimseti and Yahoo datasets. This is because the vertex-index-based partitioning approach performs unbalanced graph partitioning and thus results in small induced subgraphs, which do not have sufficient edges to completely hide the communication cost.

Table 7: Communication cost reduction

Dataset	Unhidden communication cost per iteration (sec)		T_{exec} per iteration (sec)		Speedup
	Opt.	Base.	Opt.	Base.	
Libimseti	0	0.005	0.031	0.036	1.16×
Netflix	0	0	0.15	0.15	1.00×
Yahoo	0	0.04	0.68	0.72	1.06×

6.6 Comparison with State-of-the-art

We compare the performance of FASTCF with a state-of-the-art multi-core implementation [35] and a state-of-the-art GPU implementation [22]. Native [35] implements SGD-based CF on a 24-core Intel E5-2697 processor. It has shown the fastest training speed among the existing multi-core implementations [15, 36]. In [22], SGD-based CF algorithm is implemented on a 2880-core Tesla K40C GPU. The GPU design develops several scheduling schemes for parallel thread execution on the GPU. 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 have significant synchronization overhead. As a result, the achieved speedup by the GPU acceleration is quite limited.

Table 8 compares the performance of FASTCF with [22, 35] for training the same dataset (Netflix). Our design achieves 13.3× and 12.7× speedup compared with [35] and [22], respectively. Note that the feature vector length (H) used in FASTCF is larger than [22, 35]. Therefore, from throughput perspective, FASTCF achieves 21.3× and 25.4× improvement compared with [35] and [22], respectively. Moreover, the power consumption of FASTCF (13.8 W) is far less than the multi-core (130 W) and GPU (235 W) platforms.

Table 8: Comparison with state-of-the-art multicore and GPU implementations based on Netflix dataset

Approach	Platform	H	T_{exec} per iteration	Speedup
[35]	24-core Intel E5-2697	20	2.00 sec	1.0×
[22]	2880-core Tesla K40C	16	1.90 sec	1.1×
FASTCF	Virtex UltraScale+	32	0.15 sec	13.3×

7 RELATED WORK

7.1 Graph-processing Frameworks

There are several graph-processing frameworks that support CF. Representative examples include GraphMat [16], Graphicionado [17], and GraphLab [15]. However, most of these frameworks implement Gradient-Descend-based CF [15–17] because it can be easily expressed as a vertex-centric program. GD-based CF accumulates the intermediate updates for each feature vector and performs the update after all the ratings have been traversed in an iteration. Therefore, it updates each feature vector only once per iteration and thus requires more iterations to converge and more training time than SGD-based CF (e.g., 40× more iterations to train Netflix [35]). Native [35] implements SGD-based CF on multi-core platform. It pre-processes the input training matrix by partitioning it into submatrices, and concurrently processes the submatrices that do not have feature vector dependencies using distinct CPU cores. However, the design only exploits submatrix-level parallelism (i.e, each submatrix is serially processed by a CPU core) and the submatrices can vary significantly in size. This can result in load imbalance among the CPU cores and thus increase the synchronization overhead.

7.2 GPU-based CF Accelerators

GPUs are widely used to accelerate machine learning applications [38]. GPU-based accelerators for memory-based CF [39] and ALS-based CF [13] have been developed. However, it has been shown that GPUs are not suitable for accelerating SGD-based CF [22, 38, 40]. The main reasons include (1) the fine-grained synchronization of updated feature vectors is expensive on GPU platforms [38], and (2) the SIMD execution of GPU further inflates the cost of thread divergence when synchronization conflicts occur [22]. Siede et. al [40] investigate the theoretical efficiency of SGD on GPUs, and conclude that fundamental changes in the algorithm are necessary to attain significant speedup. In [22], SGD-based CF is implemented on a Tesla K40C GPU. The design develops and compares several scheduling schemes for parallel execution of SGD on GPU, including dynamic scheduling schemes using locks and lock-free static scheduling schemes. However, none of the schemes is able to efficiently exploit the GPU acceleration and the achieved speedup compared with a CPU implementation is small (< 1.1×).

7.3 FPGA-based CF Accelerators

There have not yet been many efforts to exploit FPGA to accelerate CF. In [6], an FPGA-based accelerator for memory-based CF algorithms is proposed. The design accelerates three memory-based CF algorithms and achieves up to 16× speedup compared with multi-core implementations. However, the training dataset of the design is very small (4K users, 1K items, and 1M ratings), which can fit in the on-chip memory of state-of-the-art FPGAs. To the best of our knowledge, FASTCF is the first design to exploit FPGA to accelerate model-based CF algorithm for large training datasets.

8 CONCLUSION AND FUTURE WORK

In this paper, we presented FASTCF, an FPGA-based accelerator for SGD-based CF. FASTCF consisted of parallel processing units sharing an on-chip feature vector buffer. To optimize the performance of FASTCF, we proposed a novel 3-level hierarchical partitioning approach by using a bipartite graph representation of CF. Our focus was to obtain simple and fast heuristics based on identifying sufficient conditions for significant acceleration of the SGD-based CF algorithm on FPGA. By holistically considering the architectural characteristics of the FPGA platform, the proposed partitioning approach resulted in a complete overlap of communication and computation, up to 60× data dependency reduction, and 4.2× bank conflict reduction. As a result, our accelerator sustained a high throughput of up to 217 GFLOPS for training large real-life datasets. Compared with the state-of-the-art multi-core implementation and GPU implementation, FASTCF demonstrated 13.3× and 12.7× speedup, respectively.

In the future, we will explore multi-FPGA architectures, in which each FPGA device employs FASTCF, to further reduce the training time. We also plan to generalize our partitioning approach to support other SGD-based algorithms.

ACKNOWLEDGMENTS

This work is supported by the U.S. National Science Foundation grants ACI-1339756 and CNS-1643351. This work is also supported in part by Intel Strategic Research Alliance funding.

REFERENCES

[1] Y. Koren, R. Bell, and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems," in IEEE Computer, vol. 42, iss. 8, 2009.
[2] C. A. Gomez-Uribe and N. Hunt, "The Netflix Recommender System: Algorithms, Business Value, and Innovation," ACM Transactions on Management Information Systems (TMIS), vol. 6, iss. 4, 2016.
[3] B. Chen, D. Agarwal, P. Elango, and R. Ramakrishnan, "Latent Factor Models for Web Recommender Systems," http://www.ideal.ece.utexas.edu/seminar/LatentFactorModels.pdf

[4] "Netflix Update: Try This at Home," http://sifter.org/~simon/journal/20061211.html
[5] X. Su and T. M. Khoshgoftaar, "A Survey of Collaborative Filtering Techniques," Advances in Artificial Intelligence, 2009.
[6] X. Ma, C. Wang, Q. Yu, X. Li, and X. Zhou, "An FPGA-based Accelerator for Neighborhood-based Collaborative Filtering Recommendation Algorithms," in Proc. of International Conference on Cluster Computing (CLUSTER), pp. 494-495, 2015.
[7] J. Zhang and J. Li, "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network," in Proc. of International Symposium on Field-Programmable Gate Arrays February (FPGA), pp. 25-34, 2017.
[8] 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.
[9] G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre, "CaffePresso: An Optimized Library for Deep Learning on Embedded Accelerator-based Platforms," in Proc. of International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES), 2016.
[10] 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.
[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," 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, https://pdfs.semanticscholar.org/1a42/f06f368cf9b2ba8565e81d8e048caa5c2c9e.pdf.
[24] "Ratings and Classification Data," https://webscope.sandbox.yahoo.com/catalog.php?datatype=r
[25] 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.
[26] 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.
[27] "UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices," https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf
[28] R. J. Wilson, "Introduction to Graph Theory," ISBN 0-582-24993-7, 1996.
[29] 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.
[30] 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.
[31] 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.
[32] G. R. Morris, V. K. Prasanna, and R. D. Anderson, "A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer," in Proc. of International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 3-12, 2006.
[33] 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.
[34] J. Wawrzynek, K. Asanovic, J. Lazzaro, and Y. Lee, "Banked Multiport Memory," https://inst.eecs.berkeley.edu/~cs250/fa10/lectures/lec08.pdf.
[35] 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 SIGMOD, pp. 979-990, 2014.
[36] A. Lenharth, "Parallel Programming with the Galois System," http://iss.ices.utexas.edu/projects/galois/downloads/europar2014-tutorial.pdf
[37] "Virtex UltraScale+ FPGA Data Sheet," https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf
[38] 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.
[39] Z. Wang, Y. Liu, and S. Chiu, "An Efficient Parallel Collaborative Filtering Algorithm on Multi-GPU Platform," in Journal of Supercomputing, vol. 72, iss. 6, pp. 2080-2094, 2016.
[40] 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.