Microprocessors and Microsystems 73 (2020) 102997

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Recommender system implementations for embedded collaborative filtering applications

Francisco Pajuelo-Holguera [a], Juan A. Gómez-Pulido [a,*], Fernando Ortega [b], José M. Granado-Criado [a]

[a] Department of Technologies of Computers and Communications, Universidad de Extremadura, Spain
[b] Dep. Sistemas Informáticos, ETSI Sistemas Informáticos, Universidad Politécnica de Madrid, Spain

ARTICLE INFO

Article history:
Received 13 January 2019
Revised 19 June 2019
Accepted 9 January 2020
Available online 10 January 2020

Keywords:
Collaborative filtering
Matrix factorization
Recommender systems
Reconfigurable computing
FPGAs
High level synthesis

ABSTRACT

This paper starts proposing a complete recommender system implemented on reconfigurable hardware with the purpose of testing on-chip, low-energy embedded collaborative filtering applications. Although the computing time is lower than the one obtained from usual multicore microprocessors, this proposal has the advantage of providing an approach to solve any prediction problem based on collaborative filtering by using an off-line, highly-portable light computing environment. This approach has been successfully tested with state-of-the-art datasets. Next, as a result of improving certain tasks related to the on-chip recommender system, we propose a custom, fine-grained parallel circuit for quick matrix multiplication with floating-point numbers. This circuit was designed to accelerate the predictions from the model obtained by the recommender system, and tested with two small datasets for experimental purposes. The accelerator is built from two levels of parallelism. On the one hand, several predictions run in parallel through the simultaneous multiplication of different vectors of two matrices. On the other hand, the operation of each vector is executed in parallel by multiplying pairs of floating-point values to later add the corresponding results in parallel as well. This circuit was compared with other approaches designed for the same purpose: circuits built using automatized tools of high-level synthesis, a general-purpose microprocessor, and high-performance graphical processing units. The performance of the prediction accelerator in terms of time surpassed that of the other approaches. We also evaluated the scalability of the circuit to practical problems using the high-level synthesis approach, and confirmed that implementations based on reconfigurable hardware allow acceptable speedups of multi-core processors.

1. Introduction

Recommender systems (RS) [1] are intelligent systems that make personalized recommendations for users of large databases. The recommendations are obtained according to user behavior when requesting and handling information involving *data analytics* (DA) and *machine learning* (ML) techniques. Mainly, RS provide personalized recommendations to the users based on their preferences [2,3]. RS are also known as filters because they block the irrelevant information to the users.

The algorithms developed for RS focus on prediction, and are applied to other systems where knowledge of user behavior is important, not only for recommendation purposes, but also for analysis. For example, a student's performance on some tasks in the academic process can be predicted when RS tackles it as a ranking prediction problem: this is the well-known *predicting student performance* (PSP) problem [4].

The most popular implementation of RS is *collaborative filtering* (CF) [5,6]. CF is based on the idea that users with similar tastes in the past will have similar tastes in the future [7]. For example, if Alice and Bob have rated the same movies as positive, the new movies that Alice rates as positive might like to Bob.

CF can be applied to several fields [3,8]: movies, books, e-commerce, e-learning, etc. CF is built using a matrix that relates

Abbreviations: ALS, alternating least squares; CF, collaborative filtering; CNN, convolutional neural network; CPU, central processing units; DA, data analytics; DL, deep learning; DSP, digital signal processor; FPGA, field programmable gate array; GPU, graphics processing unit; HLS, high-level synthesis; HPRC, high-performance reconfigurable computing; K, number of latent factors; MB, Microblaze processor; MF, matrix factorization; ML, machine learning; NA, number of floating-point adders; NM, numbers of floating-point multipliers; NPE, number of prediction elements; OS, operating systems; PPC, prediction parallel circuit; PSP, predicting student performance; RC, reconfigurable computing; RMSE, root mean square error; RS, recommender systems; SGD, stochastic gradient descent; SoC, system-on-chip; VHDL, VHSIC Hardware Description Language.

* Corresponding author.
E-mail address: jangomez@unex.es (J.A. Gómez-Pulido).

https://doi.org/10.1016/j.micpro.2020.102997
0141-9331/

users with items. This matrix stores the ratings (explicit or implicit) of the users to the items. No additional information such as items' features or users' properties is required. The rating matrix has a high level of sparsity, because users only rate a small number of available items. The rating matrix usually stores thousands of million ratings that thousands of hundreds of users have performed to thousands of hundreds of items.

The main goal of CF is to fill the gaps of the sparse ratings matrix [9]. This task is usually performed by the *matrix factorization* (MF) algorithm [10,11]. MF makes predictions as a linear combination of factors, allowing for better scalability. It generates a model from which we can make predictions [12]. The prediction model is composed of two matrices such that when we make predictions for a certain user and task, the corresponding row in the first matrix and column in the second are multiplied.

MF assumes that users' ratings are conditioned by K latent factors that describes the items of the RS. For example, in a movies' RS, users' ratings are conditioned by the genres of the movies that each user rate. If a user likes action movies and dislikes romance movies, is highly probable that he or she rates positively any action movie and rates negatively any romance movie. MF algorithms try to find these hidden factors through the rating matrix.

One of the most important features of RS relates to the large amount of data involved because of the number of users and items in databases. The needs of predictions and data handling requires extensive computational resources, especially if we want to respond to real-time requests by many users. Nevertheless, some computing contexts as mobile devices make difficult to satisfy real-time requirements. Therefore, in this work we focus on two research aspects. First, we implemented a RS on a *field programmable gate array* (FPGA) device [13]. This embedded, low-energy design has the advantage of providing a first approach of the RS for off-line, highly-portable light computing environments. This approach was successfully tested considering state-of-the-art datasets. Second, we explored the acceleration of the predictions using the real-parallelism feature that FPGA provides according to two possibilities. On the one hand, at the bottom-level, we can design a circuit that parallelizes the matrix multiplication involved in the prediction operation, by multiplying the corresponding pairs of elements in the selected rows and columns, and finally adding the results in parallel as well. On the other hand, we can make several predictions in parallel at the upper-level by replicating these operations, satisfying prediction requests in parallel. Thus, if we design a system considering these two levels of parallelism, overall performance improves.

We propose using FPGA devices for the design and acceleration of RS because the *reconfigurable computing* (RC) [14] technology combines software flexibility with hardware performance by exploiting parallelism. Thus, a circuit carefully designed for specific purposes, even in arithmetic and algorithmic domains, can surpass the performance delivered by usual microprocessors or *central processing units* (CPU) in similar experimental conditions, as RC has been shown to do for many applications [15]. Moreover, we use FPGAs instead of other technologies, such as *graphics processing units* (GPUs), because they provide better performance at fine-grained levels of parallelization and consume less power in most cases [16].

The remainder of this paper is structured as follows: We summarize related work in Section 2. In Section 3, we discuss the basis of the RS. Next, Section 4 explains the design and implementation of a complete on-chip RS, which was tested with state-of-the-art datasets and measured in computing time and energy consumption terms. In Section 5, we detail our proposal of fine-grained parallel implementation of the prediction by considering the two parallel levels discussed above, where fast floating-point matrix multiplication is the target of interest. Two datasets are introduced as testbenches for the hardware implementation. Other hardware solutions based on high-level synthesis, graphics processing units and commonly used microprocessors are presented for performance comparison purposes. Some issues about speedup, power consumption, and circuit's scalability are described in this section too. Finally, the conclusions of this paper are detailed in Section 6.

2. Literature review

The reconfigurable computing technology is drawing researchers attention in different possibilities to apply it to machine learning. Some ML algorithms, or parts of them, have been implemented on FPGAs for different purposes, mainly for acceleration tasks. For example, *convolutional neural networks* (CNN) and *deep learning* (DL) [17,18], K-Means for clustering [19–21], kernel density estimation [22] are a few examples in that direction. The possibility of considering FPGAs in this context led some companies as Amazon to offer tools and platforms to accelerate particular tasks when dealing with large databases and cloud services [23]. Some works attempt to implement CF and RS on FPGAs, considering different parts, focus and constraints. It is more usual to find implementations of some parts of a RS rather than the whole RS itself. For example, in [24] a *stochastic gradient descent* (SGD) algorithm [25] used for training RS models is implemented on FPGA considering single-precision floating-point. In this sense, the approach that we present in Section 4 tries to host on FPGA both modeling and training aspects of the RS when handling datasets.

As pointed out in Section 1, RS are appropriate for use in parallel, not only by using repeated processing elements in parallel for prediction purposes, but also by parallelizing the relevant matrix multiplication tasks. Both ways are used in our proposal to implement parallel circuits for prediction purposes using FPGAs from the perspective of fine-grained parallelization. This approach is supported by trends of research in analytics [26], where promising lines of work have exploited the features of FPGAs. For example, they have been applied to online analytical processing, text analytics, and time series processing. On this matter, the fine-grained parallel architecture proposed in this study contributes to recent work that has used FPGAs to accelerate functions in RS, for example, neighborhood-based collaborative filtering [27].

With regard to matrix multiplication in general, several studies have implemented approaches using FPGAs in pursuit of different goals: high speed [28], area reduction [29], low power consumption, and the pipeline approach [30], for instance. The majority of these implantations deal with fixed-point values for arithmetic operations [31], whereas fewer researchers have examined the floating-point domain [32]. In this sense, our research meets the usual needs of floating-point calculation when predicting performance, according to the kind of data and algorithmic operations in RS.

An instance of matrix multiplication is the implementation of MF techniques on FPGAs. Some attempts have been made in this vein [33], allowing floating-point operations [34] and adding pipelined calculation [30]. In this context, our proposal of parallel processing is restricted to the prediction phase of the RS, instead of model building using MF. Moreover, the matrix multiplication in our approach multiplies matrix vectors in parallel by considering the constraints of the floating-point arithmetic.

Our research starts focusing on designing a circuit that can perform matrix multiplication quickly, and can be replicated for parallel predictions. Once its speedup with regard to other solutions has been obtained, several approaches can be used where the core can be included to build massively parallel scalable systems ready for big data management, where larger matrices can be handled by solving different questions concerning memory and communication. These approaches usually involve several FPGA devices in

multi-core architectures, where basic cores such as ours can be used as coprocessors of small, embedded CPUs, such as *Microblaze* (MB) [35]. This coarse-grained parallelization belongs to the domain of *high-performance reconfigurable computing* (HPRC) [36], which exploits FPGAs [37] but require a design according to proper models of computation [38]. Some hardware solutions in the literature involve several boards, each of which has several FPGAs mounted on it, such as the RIVYERA computing system [39].

Other works focus on different MF techniques parallelized by means of GPUs. Thus, a parallel architecture to be performed on GPUs is proposed in [40] where SGD was adapted to remove the dependence on the user and item pair in MF for large-scale CF problems. With regard to non-negative MF, [41] proposes an online, scalable and single-thread-based generalized sparse MF for parallelization on GPUs, which describes properly the high-dimensional and sparse matrices, whereas [42] deals with manifold-regularized non-negative MF, proposing a single-thread-based model parallelized on GPUs in order to avoid large-scale matrix manipulation and remove the dependence among the feature vectors. Our contribution differs from these works in the focus (fine and coarse grained parallelization for on-chip, low-energy embedded collaborative filtering application) and the platform for the hardware implementation (FPGA).

3. Matrix factorization based collaborative filtering

Matrix factorization approaches build a model of the rating matrix R by its factorization into two dense matrices [43]: P, that contains the suitability of each latent factor with each user; and Q, that contains the suitability of each latent factor with each item. The values of these new matrices are learned from the rating matrix in such a way that they satisfy Eq. (1).

$$R \approx P \cdot Q^T \tag{1}$$

MF has demonstrated its superiority against other CF implementations [44–46]. MF provides both accurate predictions (the estimated rating) and recommendations (the set of top N items more relevant to an user). MF is also highly scalable: once the model is learned, predictions can be computed by a simple dot product of two K-dimensional vectors. However, this model is outdated when new ratings, users or items are incorporated to the RS. The learning process must be repeated periodically in order to incorporate them to the model. Consequently, this training process has a high computational cost.

The training process requires to find the optimal values of the matrices P and Q that minimize the error in the predictions provided by the model. To learn the factor vectors the system minimizes the regularized squared error of the known rating set:

$$\min_{p_u, q_i} \sum_{(u,i)\in\kappa} (r_{u,i} - \vec{q}_i^T \cdot \vec{p}_u)^2 + \lambda(||q_i||^2 + ||p_u||^2) \tag{2}$$

Where κ is the set of (u, i) pairs for with the rating $r_{u,i}$ is known, p_u is the latent factors vector of the user u, q_i is the latent factors vector of the item i and λ is a regularization hyper-parameter to avoid overfitting.

The model can be trained using SGD. It loops over all the existing ratings ($r_{u,i}$) until convergence. For each training case (ie. each known $r_{u,i}$) SGD updates the values of the matrices P and Q according to Eqs. (3) and (4) respectively.

$$\vec{p}_u \leftarrow \vec{p}_u + \gamma \cdot (e_{u,i} \cdot \vec{p}_u - \lambda \cdot \vec{q}_i) \tag{3}$$

$$\vec{q}_i \leftarrow \vec{q}_i + \gamma \cdot (e_{u,i} \cdot \vec{q}_i - \lambda \cdot \vec{p}_u) \tag{4}$$

Where γ is an hyper-parameter that controls the learning speed and $e_{u,i}$ is the error in the prediction of the rating of the user u to the item i:

$$e_{u,i} = r_{u,i} - \vec{q}_i^T \cdot \vec{p}_u \tag{5}$$

The training of the model usually spends a huge amount of time until convergence. To speed up the training process, the updates of the factors vectors of each user and item can be parallelized. However, using Eqs. (3) and (4) of SGD, parallelization cannot be performed. To update each $\vec{p}_u$ the value of each $\vec{q}_i$ is required, and vice versa. To solve this problem *alternating least squares* (ALS) technique is applied [47]. ALS rotates between fixing the $\vec{q}_i$ and fixing the $\vec{p}_u$. In ALS, the system computes each $\vec{q}_i$ independently of the other item factors and computes each $\vec{p}_u$ independently of the other user factors. This gives rise to potentially massive parallelization of the algorithm.

Once the model is trained, predictions can be performed computing the dot product of the factors vector of the target user and the factors vector of the target item. The prediction of the rating of the user u to the item i can be computed according to Eq. (6). Recommendations to each user can be obtained from the set of T unrated items with the highest predictions ($\hat{r}_{u,i}$).

$$\hat{r}_{u,i} = \vec{q}_i^T \cdot \vec{p}_u \tag{6}$$

Algorithm 1 contains the pseudo code for MF. It receives as input the rating matrix R, the number of latent factors K, and the hyper-parameters to control the learning process λ and γ. It returns the latent factors matrices P and Q learned from the rating matrix. Convergence criteria is usually defined as a fixed number of iterations.

```
input : R, K, λ, γ
output: P, Q
Create a random matrix P with U rows and K columns
Create a random matrix Q with I rows and K columns
repeat
    for each user u do // This loop can me parallelized for each user
        for each item i rated by user u do
            error = R[u][i] - dotProduct(P[u], Q[i])
            for each factor k do
                P[u][k] += γ · (error · P[u][k] − λ · Q[i][k])
            end
        end
    end
    for each item i do // This loop can be parallelized for each item
        for each user u that has rated the item i do
            error = R[u][i] - dotProduct(P[u], Q[i])
            for each factor k do
                Q[i][k] += γ · (error · Q[i][k] − λ · P[u][k])
            end
        end
    end
until convergence
return P, Q
```

Algorithm 1: Matrix Factorization algorithm.

4. On-chip implementation of a complete recommender system

In this section we detail two implementations of the RS on a reconfigurable hardware platform. The first implementation (FPGA-RS1) consists simply in installing the required source codes of the RS on an embedded operating system and next running the executable files on an FPGA embedded microprocessor. This fast implementation allows us to obtain a first approach to measure the

Table 1
Datasets used to test the on-chip recommender system.

Dataset	Kaggle	Movielens-100K	Movielens-1M	Netflix-100M
Ratings	100,000	100,000	1,000,000	100,000,000
Users	700	943	6000	480,188
Items	9000	1682	4000	17,691

performance in computing time and energy consumption terms, as well as check for accurate results. The second implementation (FPGA-RS2) consists in designing a parallel RS by using high level synthesis programming, pursuing an improved performance with regard to the previous approach.

4.1. Datasets

The two on-chip implementations of the RS were tested using four state-of-the-art datasets of different characteristics, widely considered for this purpose: The Movies Dataset (Kaggle) [48], Movielens-100K, Movielens-1M [49], and Netflix-100M [50]. These datasets gather the activity of many users when rating movies with scores from 1 to 5, where each user rates at least 20 movies. Table 1 shows the main features of these datasets.

4.2. FPGA-RS1: RS on embedded microprocessor and operating system

We chose the low-energy system Zedboard Zynq-7000 prototyping board for this first implementation approach. This low-cost and popular board includes a *system-on-chip* (SoC) Xilinx Zynq XC7Z020 and all the elements required to design any computing system based on Linux, Windows, and Android *operating systems* (OS), among others. The core of the board is an ARM processing architecture attached to the SoC and programmable logic. Furthermore, other elements provide enough features to interact with the user's needs: HDMI and VGA video interfaces, audio input/output, Ethernet connector, slot for SD cards, and USB interfaces (OTG for handling peripherals, JTAG for programming Zynq from PC, and UART for serial port communications). The processing potentiality is conditioned by 512MB of DDR3 memory and two oscillators to generate clock signals of 100 MHz and 33.3 MHz.

We installed an embedded Linux OS (Linaro distribution) on the board in order to allow running the RS on the FPGA. This OS is launched from a separated partition in the SD card, so the changes made by the program are written in that partition. The advantage of using the Linaro distribution is that we can work with the board in the same way that we usually handle the microprocessor of a personal computer. Therefore, the C codes of the RS executed on both ZedBoard and CPU are exactly the same. Once installed Linaro on the board, we access it from our PC through a secure shell client; next, we transfer the source code files, compile them, and run the file with the machine code just as in a PC.

The circuit in the FPGA was designed with Xilinx Vivado 2017, which includes the required support for a Zynq-based processing system.

4.3. FPGA-RS2: parallel RS design

The second implementation improves a lot the performance obtained by the first approach. In this case, the RS was parallelized by considering *high-level synthesis* (HLS) [51]. Specifically, we considered Vivado HLS [52] for this work. HLS allows us to design circuits quickly by parallelizing code in C automatically for FPGA-based implementations. However, we can also modify the design manually by including different optimization directives to build a circuit with higher precision without the need to modify the C code. Therefore, we included some directives to unroll loops and functions, and divide arrays to perform parallel operations.

Fig. 1 shows the basic strategy to run certain tasks in parallel. According to Algorithm 1, after initializing the algorithm, two consecutive loops were parallelized to update the corresponding factorized matrices for each user/item. These loops are sequentially performed several times.

We measured the elapsed time in the system designed using HLS by specifying the same FPGA device and the required operational frequency. Once the code had been synthesized, HLS tells us whether the given frequency can be supported by the FPGA device as well as the number of clock cycles used by the hardware. Hence, we calculated the elapsed time.

4.4. Performance results and comparison

The two FPGA implementations and a simple program for CPUs for comparison purposes, consider the same settings to run the RS. For example, we have considered 150 iterations according to Algorithm 1. As a sample of performance results obtained by this algorithm after applying the four datasets, Table 2 shows the time elapsed in each iteration and the prediction error measured as *root mean square error* (RMSE) for the FPGA-RS2 implementation (all the hardware implementations will give the same RMSE results).

The number of iterations in Algorithm 1 is the main cause of the elapsed time in all the implementations, slowing down the model generation. Tables 3 and 4 show the timing and power results respectively of both FPGA implementations and a simple software solution performed on CPU, after applying the four datasets. The timing and power consumption reports shown in both tables are clearer when we read "speedup" and "power reduction" instead of "time taken" and "power consumed", respectively.

For the RS performed on CPU, we have considered an Intel i7-950 CPU. It is important to highlight that this CPU provides a clock

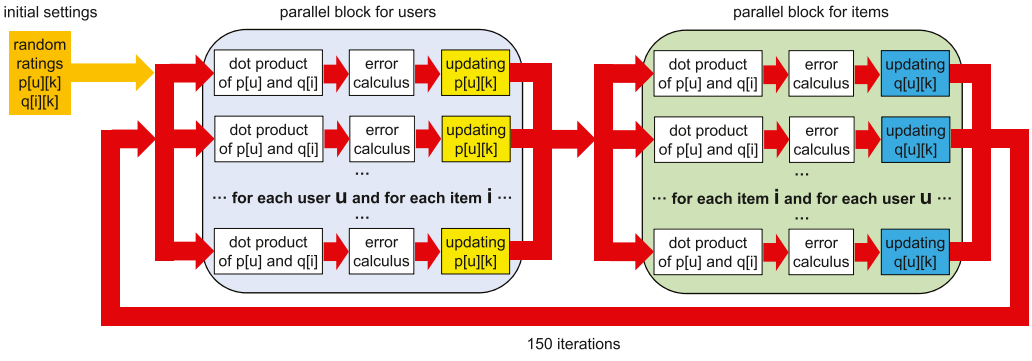

Fig. 1. Basic strategy to design the parallel recommender system.

Table 2
Performance results of the recommender system.

Dataset	Kaggle	Movielens-100K	Movielens-1M	Netflix-100M
Time (s) by iteration	7.53	5.54	19.56	657.66
RMSE (training dataset)	0.9266	0.9286	0.9491	0.9547
RMSE (test dataset)	0.8396	0.8489	0.8664	0.8992

Table 3
Timing results (s).

	Kaggle	Movielens-100K	Movielens-1M	Netflix- 100M
CPU	76.12	33.62	113.41	96,381.80
FPGA-RS1	5057.81	3279.54	11,580.68	525,742.73
FPGA-RS2	1129.70	831.04	2934.57	98,649.80
Speedup FPGA-RS2 vs FPGA-RS1	× 5	× 4	× 4	× 5

Table 4
Power results.

	Kaggle	Movielens-100K	Movielens-1M	Netflix- 100M
CPU	8.21	7.33	12.31	32.21
FPGA-RS1	0.95	0.82	1.64	3.03
FPGA-RS2	0.95	0.82	1.64	3.03
Power reduction FPGA vs CPU	88%	89%	87%	91%

frequency of 3 GHz, whereas the maximum frequency provided by the FPGA implementations to run the RS safely was 667 MHz.

As expected, the embedded FPGA implementations are slower than CPU, although this result does not devalue the advantages of implementing RS on embedded devices based on FPGAs, as we pointed out previously in Section 1. These results indicate also that the parallel RS design outperforms the simple embedded implementation.

Finally, it is important to know the energy impact of running the RS implementations. We think that power consumption is an important indicator because it allows us to design energy-aware embedded circuits that minimize operational costs when performing many predictions over time in computing-intensive environments, such as RS. The power consumption of the RS in the CPU implementation was measured by using *Powerstat* tool under Linux Ubuntu OS, whereas *Power Analyzer* tool in Xilinx Vivado provided the total on-chip power of the two FPGA implementations. In this case, the power reduction in both FPGA implementations regarding the CPU was very significant (around 90%). Even this reduction is more significant when we consider the largest dataset.

The timing results obtained by the FPGA implementations of the whole RS led us to tackle the acceleration of particular tasks. This is the case of the prediction phase.

5. Accelerating the prediction phase: proposal and alternatives

The design of any accelerator system based on FPGAs must cover several issues, from the basic core at the bottom level to the communication architecture at the upper level, by solving problems related to scalability, memory, and bandwidth. Therefore, the first and mandatory step is to carefully design the basic core: If its performance is not adequate, the remaining steps are not completed to design a more complex architecture. Therefore, we focus on this first step. We designed a custom *prediction parallel circuit* (PPC) for accelerating the prediction according to the model described in Section 3. This core replicates small operators for matrix multiplications, from the perspective of fine-grained parallelization, to allow for parallel predictions. As we obtained a good speedup, different architectures for the entire RS process can be further explored, including a configurable PPC. In this sense, we explore an approach that includes the PPC in training and generates the prediction model driven by an FPGA-embedded processor.

Our proposal contributes to a better understanding of some aspects of the problem. We tackle the strong component of the floating-point arithmetic in matrix multiplication using a large set of optimized floating-point adders and multipliers. These elements work in a coordinated way to implement parallel operations. We also used custom datasets composed of small matrices for experimental purposes. Thus, if our proposal achieves good speedups, we can delegate the solution for larger matrices to future research that can use the same methodology. Moreover, we think this approach may be interesting when the RS works with the practical requirements and prediction models of fixed size. We also seek significant speedups on common computing devices, such as CPUs and GPUs. Furthermore, the proposed circuit consumes less power, which is useful in intensive computing scenarios.

5.1. Design of the prediction parallel circuit

Once the RS generates the prediction model, many predictions can be requested in real time. A hardware accelerator can perform several predictions in parallel from the values in P and Q. Hence, our effort is oriented to design a fast on-chip architecture that can perform parallel predictions by multiplying the corresponding rows and columns of P and Q in parallel as well. PPC is designed according to two levels of parallelism.

- At the bottom level, each prediction $\hat{r}_{u,i}$ is calculated in two stages from the corresponding data of the pair $(p_{u,k}, q_{i,k})$. Row u in P is multiplied by column i in Q by using K floating-point parallel multipliers, and the results are added in parallel by using $K/2$ floating-point adders. All individual predictions are calculated in parallel and at the same time.
- At the top level, we replicate a certain number of prediction elements (*NPE*). This number mainly depends on the area available in the FPGA device, which is strongly related to the number of latent factors. These elements operate in parallel to provide the results of the predictions at the same time. Hence, the

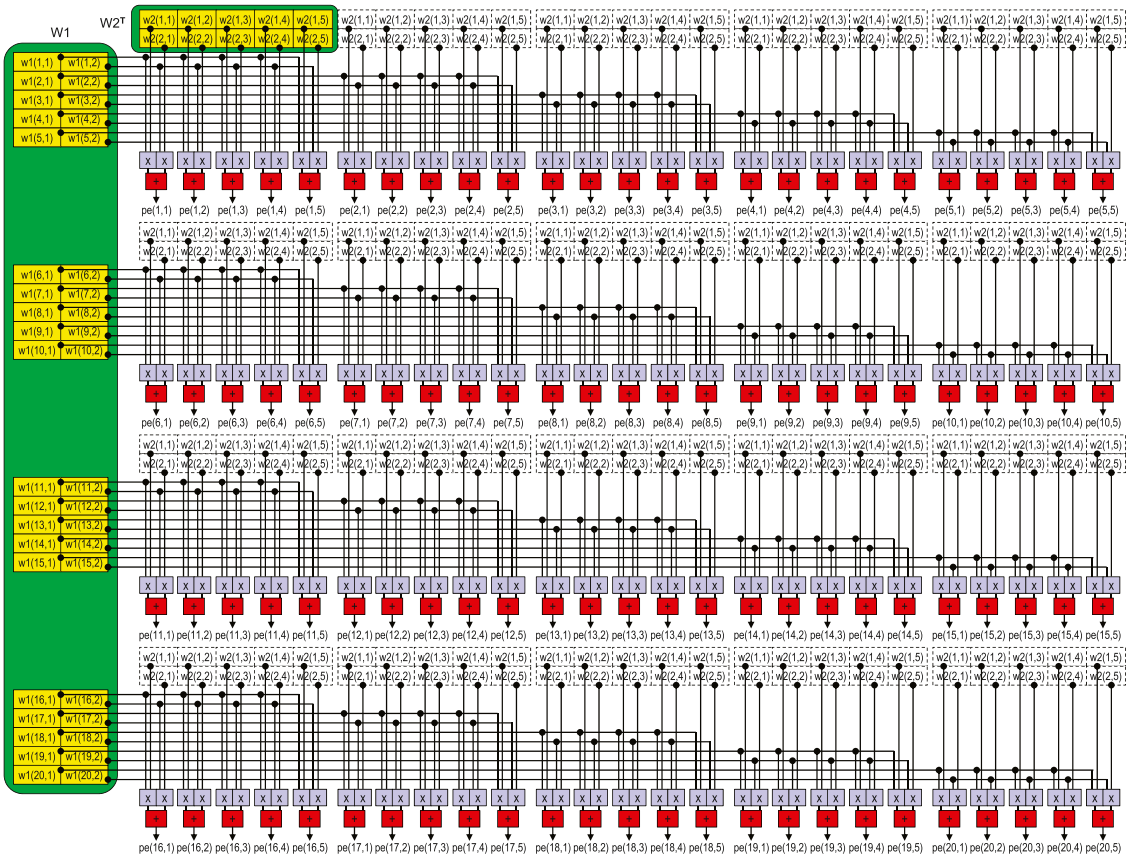

Fig. 2. PPC architecture with the two levels of parallelism for dataset 20 × 5.

numbers of floating-point multipliers (*NM*) and adders (*NA*) required for the overall circuit are $NPE \times K$ and $NPE \times K/2$, respectively.

Fig. 2 shows a hardware implementations of this accelerator circuit for a simple test dataset of size s 20 × 5. It includes both levels of parallelism: vector multiplication and parallel prediction. The inputs to the multipliers come from *P* and *Q*, whereas those to the adders from the outputs of the multipliers. Matrix multiplication is fully parallelized, whereas the sums in the prediction calculation combine parallel and sequential stages.

We designed different elements of PPC by programming codes in *VHSIC Hardware Description Language* (VHDL) [53]. This hardware description language is efficient, especially when programming at the register-transfer level. We also measured the speedup of the FPGA devices taking advantage of the post-placement and routing simulation tools provided by the implementation software.

We implemented and tested the accelerator circuit on a high-performance FPGA device: Virtex6 xc6vlx550t (40 nm CMOS depth, 550 k logic cells, 864 DSP slices, and 22,752kB RAM blocks). The results were compared to those obtained by a contemporary CPU: Intel i7-2600 (32 nm CMOS depth, 3.4 GHz). Four features were important for identifying the performance of the FPGA device: process technology (CMOS depth), number of logic cells (in connection with the area available to fit the circuit), number of internal DSP slices (allows the floating-point arithmetic operators to increase their speed), and the number of memory blocks (useful to handle data).

We considered two alternatives to the PPC based on state-of-the-art hardware prototyping technologies: high-level synthesis and GPU devices. These alternatives helped us test the comparative performance of the PPC in terms of computing time and power consumption. The performance of PPC and these two approaches was also compared with that of a typical CPU.

- *High level synthesis* approach. Using this technology previously presented, we designed a circuit that implements the same computational tasks as the PPC by exploiting parallelism. The advantage of this approach is the ease of programming with HLS to scale to architectures for managing larger datasets or considering more than two latent factors. On this matter, a PPC programmed by VHDL is less flexible.
- *Graphics processing units* (GPU) approach. Modern GPUs are hardware platforms popular for *high-performance computing* (HPC) applications in many fields. They exploit massively parallel operations on chip, including the floating-point arithmetic. These features have motivated us to check whether GPUs [16] can surpass the performance of FPGA devices for this problem, following the trend of the competition between FPGAs and GPUs for floating-point arithmetic operations. To accomplish this goal, we designed a parallel code that executed the operations described in (6) and implemented the GPU according to the PPC architecture. For a realistic comparison with the PPC, we performed the same parallel tasks and measured only the execution time of the arithmetic operations. The time spent sending and receiving data from the matrices was not measured to compare similar operations. For this implementation, we used the GeForce TitanX GPU board for the experiments. It has 3,072 CUDA cores running at 1000 MHz clock frequency, and 12GB of GDDR5 memory at 7 Gbps frequency with maximum bandwidth of 336.5 GB/s. It consumes 250 W of power. To programme the code for matrix operations, we used the GCC 4.9.3 C compiler and OpenCL 1.2 library [54].

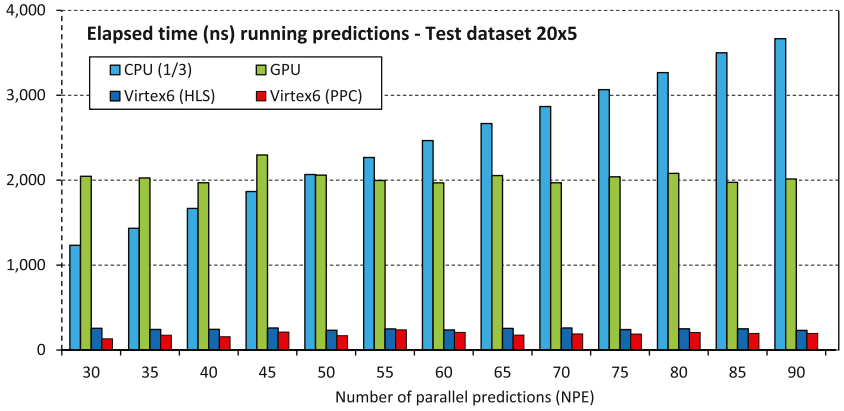

Fig. 3. Time needed to perform different number of predictions (*NPE*) in the test dataset, for four implementations: PPC, HLS, GPU and CPU. Note that time of CPU is divided by 3 in order to reduce the bar height.

5.2. Performance results

Next we present the experimental results, mainly the speedup achieved by the PPC in comparison with other approaches. The best implementation practices drawn from the PPC, useful to improve overall performance, are also explained. We highlight three considerations stemming from the analysis. First, the area occupied was given by several indicators (registers, look-up tables, and occupied slices) that allowed us to calculate the number of predictions we could perform in parallel on the same device. Also, performance in terms of computing time was calculated from the maximum frequency corresponding to the minimum clock cycle. Power consumption was calculated in the placement and routing phases.

Fig. 3 shows the time needed to perform different number of predictions (*NPE*) according to (6). The figure shows the times taken for four implementations: PPC, HLS, GPU, and CPU.

From this figure, we can arrive at three conclusions. First, only in the CPU implementation did the time needed for computation increase with *NPE*. The reason is simple: The elapsed time in the PPC, HLS, and GPU considered parallel *NPE* predictions, whereas we used a loop of sequential predictions for the CPU. The FPGA implementations (PPC and HLS) took much less time than the GPU and PPC, even though the GPU performed predictions in parallel too. This fact, together with the high power consumption of the GPU, confirmed that the FPGA is the best option to accelerate predictions of the recommender system. Between the two FPGA implementations, the PPC performed slightly better than HLS.

Table 5 shows the PPC speedups. Speedups on the CPU were up to $\times$ 60, showing that the more parallel predictions we considered, the better the speedup. With regard to the GPU, the PPC obtained speedups of around $\times$ 10 in most cases. Thus, the PPC yielded slightly better performance than the HLS (except for in only one case, when $NPE = 100$). Of course, the computing time of all the approaches measured the same number of predictions.

Table 5
PPC speedups with regard to other implementations (CPU, GPU, and HLS) according to different number of predictions (*NPE*) for test dataset 20 $\times$ 5.

NPE	PPC vs CPU	PPC vs GPU	PPC vs HLS
30	$\times$ 28.2	$\times$ 15.6	$\times$ 2.0
35	$\times$ 24.6	$\times$ 11.6	$\times$ 1.4
40	$\times$ 32.1	$\times$ 12.6	$\times$ 1.6
45	$\times$ 26.5	$\times$ 10.9	$\times$ 1.2
50	$\times$ 36.7	$\times$ 12.2	$\times$ 1.4
55	$\times$ 28.7	$\times$ 8.4	$\times$ 1.2
60	$\times$ 35.6	$\times$ 9.5	$\times$ 1.1
65	$\times$ 45.5	$\times$ 11.7	$\times$ 1.4
70	$\times$ 45.5	$\times$ 10.4	$\times$ 1.4
75	$\times$ 48.9	$\times$ 10.9	$\times$ 1.3
80	$\times$ 47.6	$\times$ 10.1	$\times$ 1.2
85	$\times$ 53.8	$\times$ 10.1	$\times$ 1.3
90	$\times$ 56.4	$\times$ 10.3	$\times$ 1.2
95	$\times$ 60.3	$\times$ 10.4	$\times$ 1.0
100	$\times$ 55.4	$\times$ 9.2	$\times$ 0.7

5.3. Implementation keys

We considered three implementation keys: synthesis options, operation frequency, and power consumption.

We tested two possibilities to synthesize floating-point operators: internal *digital signal processors* (DSPs) and logic blocks. If we consider DSPs, the performance can be improved, but the limited number of DSPs forces us to consider digital logic if we want more parallel operators. Nevertheless, more logic blocks involve increasing the area required for the FPGA; hence, the trade-off between the number and performance of parallel units must be evaluated in each case.

The best results of the PPC considered different synthesis options and implementations for the arithmetic operators. The different synthesis options were grouped into three optimization profiles: default (DEF), performance in terms of time with physical synthesis (TPP), and performance in terms of time without I/O block packing (TPN). Other synthesis profiles were discarded because of their poor results. Each synthesis of the CPC was repeated up to six times (according to the three synthesis profiles and the two possibilities, of using DSPs or logic blocks for the floating-point operators) to obtain the highest clock frequency.

Fig. 4 shows the operational frequency of the PPC with regard to the number of parallel predictions for the three synthesis profiles, and by considering whether the floating-point arithmetic operators include internal DSPs or logic blocks (Logic). The curves follow a non-linear trend to reduce frequency when the number of predictions increased. Among other effects, this implied that an increase in PPC speedup compared with the CPU was neither linear nor unlimited because more logic elements appeared when more floating-point arithmetic operators were replicated, rendering the datapath denser. Consequently, the routing effect had a negative influence. Nevertheless, the reduction in operational frequency was less pronounced when more predictions were performed. However, smaller frequencies do not imply a worse speedup, as a higher number of parallel predictions makes up for higher clock cycles. Finally, Fig. 4 considers the DEF profile in case we use DSPs for floating-point operators. Otherwise, we can opt for a TPN profile to yield an acceptable result in a majority of cases.

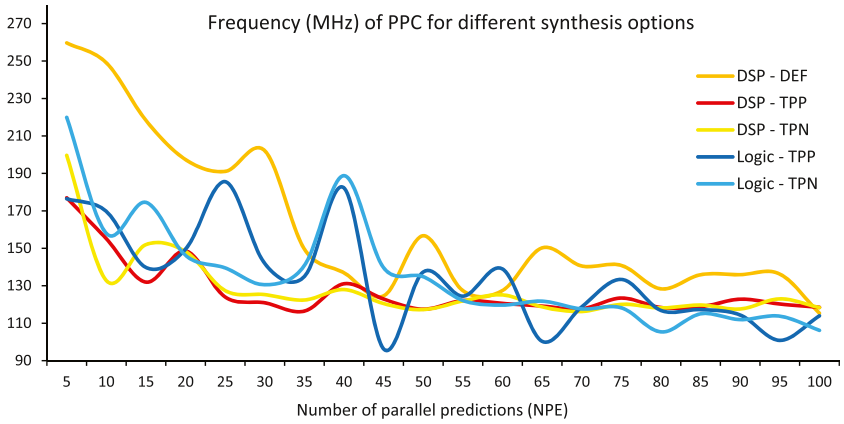

Fig. 4. Frequency of the PPC for the three synthesis profiles.

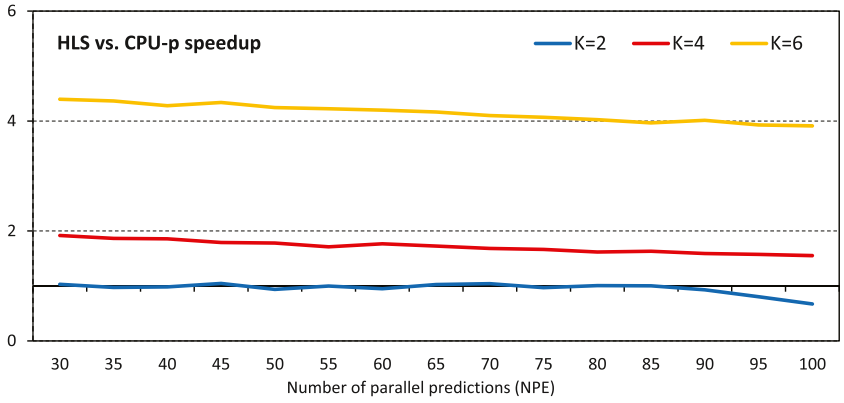

Fig. 5. HLS speedup with regard to CPU-p (parallelized CPU) according to different number of latent factors (K) and predictions (NPE) for the test dataset

With regard to energy issues, the total power consumption and thermal distribution in the FPGA device were measured after the placement and routing phase of the implementation process by means of the *Xilinx XPower Analyzer* tool. For the test dataset, 30 parallel predictions, floating-point arithmetic operators implemented with DSPs, and the default synthesis profile, we obtained a power consumption of 6.4 watts for the PPC, whereas the processor consumed approximately 95 watts. This means that the FPGA reduces power consumption by at least 93% compared with the CPU.

5.4. Scalability study

In this experiment, we studied the effect of increasing the number of latent factors in the FPGA speedup. The greater the number of latent factors, the greater the size of matrices P and Q. Consequently, the numbers of addition and multiplication operations involved in the prediction calculations increase considerably. The main consequence in implementing hardware is a significant increase in the FPGA logic resources needed to replicate more parallel units at both levels: matrix multiplications parallelized and parallel predictions. Moreover, the greater number of floating-point arithmetic operators significantly affects the available area of the FPGA. Nevertheless, real-world recommender systems can consider more than two latent factors, hence the need to evaluate the FPGA speedup in these cases.

For this purpose, we tested FPGA speedup for $K =$ two, four, and six latent factors. As a new hardware design of the PPC for an increased K becomes much more complex, and bearing in mind that the performance of the PPC and HLS was similar, we take advantage of the programming flexibility that HLS provides to explore K scalability. Moreover, to make a more realistic comparison, we programmed the same CPU with *OpenMP* to perform parallel predictions with four cores. We call this new parallelized design CPUp. Finally, we considered the same test dataset, knowing that it implied greater computational effort.

Fig. 5 shows the HLS speedup with regard to the parallelized CPU (CPUp) according to different numbers of latent factors (K) and predictions (NPE). The analysis of this figure generates two main conclusions. First, the FPGA solution improved in performance in terms of time when K increased, which is promising for the application of a reconfigurable hardware design to practical cases. Second, there was a slight decrease in speedup if we considered a greater number of parallel predictions. The reason for this behavior has been pointed out before: A greater the number of logic elements required to replicate prediction units implies a denser datapath that slightly decreases operational frequency, as shown in Fig. 4.

6. Conclusions

This paper explores the application of reconfigurable computing technology based on FPGAs to design and implement on-chip solutions for recommender systems. On the one hand, we designed a parallelized recommender system to be hosted on an embedded platform. This solution provides a first approach for running off-line and light embedded collaborative filtering applications when using highly-portable and low-energy computing environments. This approach was successfully tested considering state-of-the-art datasets. On the other hand, we have designed a circuit that can supply simultaneous predictions based on matrix multiplication. The design performs fast parallel predictions using a

model previously obtained through basic collaborative filtering algorithm, where many floating-point arithmetic operations are involved. The results show that FPGAs provide high speedups with regard to general-purpose CPUs and high-performance GPUs, not only because of the parallelization of matrix operations and efficient floating-point arithmetic operators, but also thanks to concurrent predictions. Furthermore, the low power consumption of FPGA devices in comparison with CPUs and GPUs facilitates the designing of low-cost computing solutions for embedded applications. This approach may be interesting when the recommender system tackles frameworks where the size of the prediction model is fixed but its contents update quickly. Moreover, the proposal is worthwhile if many predictions are required in real time.

Declaration of Competing Interest

The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

Acknowledgements

This work was partially funded by the Government of Extremadura (Spain) under the project IB16002, and by the ERDF (European Regional Development Fund, EU) and the State Research Agency under the contract TIN2016-76259-P.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.micpro.2020.102997.

References

[1] D. Jannach, M. Zanker, A. Felfernig, G. Friedrich, Recommender Systems. An Introduction, Cambridge University Press, 2011.
[2] J. Bobadilla, F. Ortega, A. Hernando, A. Gutiérrez, Recommender systems survey, Knowl.-Based Syst. 46 (2013) 109–132.
[3] G. Adomavicius, A. Tuzhilin, Context-aware recommender systems, in: Recommender systems handbook, Springer, 2015, pp. 191–226.
[4] N. Thai-Nghe, L. Drumond, T. Horvath, A. Krohn-Grimberghe, A. Nanopoulos, L. Schmidt-Thieme, Factorization techniques for predicting student performance, in: Educational Recommender Systems and Technologies: Practices and Challenges, IGI-Global, 2012, pp. 129–153.
[5] G. Adomavicius, A. Tuzhilin, Toward the next generation of recommender systems: asurvey of the state-of-the-art and possible extensions, IEEE Trans. Knowl. Data Eng. 17 (6) (2005) 734–749.
[6] F. Ricci, L. Rokach, B. Shapira, Introduction to recommender systems handbook, in: Recommender systems handbook, Springer, 2011, pp. 1–35.
[7] J. Bobadilla, F. Serradilla, J. Bernal, A new collaborative filtering metric that improves the behavior of recommender systems, Knowl.-Based Syst. 23 (6) (2010) 520–528.
[8] H.J. Ahn, A new similarity measure for collaborative filtering to alleviate the new user cold-starting problem, Inf. Sci. 178 (1) (2008) 37–51.
[9] J.L. Herlocker, J.A. Konstan, L.G. Terveen, J.T. Riedl, Evaluating collaborative filtering recommender systems, ACM Trans. Inf. Syst. (TOIS) 22 (1) (2004) 5–53.
[10] A. Hernando, J. Bobadilla, F. Ortega, A non negative matrix factorization for collaborative filtering recommender systems based on a bayesian probabilistic model, Knowl.-Based Syst. 97 (2016) 188–202.
[11] P. Paatero, U. Tapper, Positive matrix factorization: a non-negative factor model with optimal utilization of error estimates of data values, Environmetrics 5 (2) (1994) 111–126.
[12] S. Rendle, L. Schmidt-Thieme, Online-updating regularized kernel matrix factorization models for large-scale recommender systems, in: Proceedings of the 2008 ACM conference on Recommender systems, 2008, pp. 251–258.
[13] C. Unsalan, B. Tar, Digital System Design with FPGA: implementation Using Verilog and VHDL, McGraw-Hill, 2017.
[14] R. Tessier, K. Pocek, A. DeHon, Reconfigurable computing architectures, Proc. IEEE 103 (3) (2015) 332–354.
[15] M. Vestias, H. Neto, Trends of cpu, gpu and fpga for high-performance computing, in: 24th International Conference on Field Programmable Logic and Applications, IEEE, 2014, pp. 1–6, doi:10.1109/FPL.2014.6927483.
[16] S. Chey, J. Liz, J. Sheaffery, K. Skadrony, J. Lach, Accelerating compute-intensive applications with gpus and fpgas, in: IEEE Symposium on Application Specific Processors, IEEE, 2008, pp. 101–107.
[17] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, J. sun Seo, Alamo: fpga acceleration of deep learning algorithms with a modularized rtl compiler, Integration 62 (2018) 14–23, doi:10.1016/j.vlsi.2017.12.009.
[18] P.R. Gankidi, J. Thangavelautham, Fpga architecture for deep learning and its application to planetary robotics, in: 2017 IEEE Aerospace Conference, 2017, pp. 1–9, doi:10.1109/AERO.2017.7943929.
[19] J. Canilho, M. Véstias, H. Neto, Multi-core for k-means clustering on fpga, in: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1–4, doi:10.1109/FPL.2016.7577313.
[20] F. Winterstein, S. Bayliss, G.A. Constantinides, Fpga-based k-means clustering using tree-based data structures, in: 2013 23rd International Conference on Field programmable Logic and Applications, 2013, pp. 1–6, doi:10.1109/FPL.2013.6645501.
[21] Z. Lin, C. Lo, P. Chow, K-means implementation on fpga for high-dimensional data using triangle inequality, in: 22nd International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 437–442, doi:10.1109/FPL.2012.6339141.
[22] K. Nagarajan, B. Holland, A.D. George, K.C. Slatton, H. Lam, Accelerating machine-learning algorithms on fpgas using pattern-based decomposition, J. Signal Process. Syst. 62 (1) (2011) 43–63, doi:10.1007/s11265-008-0337-9.
[23] Amazon ec2 f1 instances, 2019.
[24] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, C. Zhang, Fpga-accelerated dense linear machine learning: aprecision-convergence trade-off, in: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 160–167, doi:10.1109/FCCM.2017.39.
[25] L. Bottou, Large-scale machine learning with stochastic gradient descent, in: Proceedings of 19th International Conference on Computational Statistics, Springer, 2010, pp. 177–186.
[26] R. Bordawekar, B. Blainey, C. Apte, Analyzing analytics, SIGMOD Record 42 (4) (2014) 17–28.
[27] X. Ma, C. Wang, Q. Yu, X. Li, X. Zhou, An fpga-based accelerator for neighborhood-based collaborative filtering recommendation algorithms, in: IEEE International Conference on Cluster Computing (CLUSTER 2015), Chicago, IL, USA, 2015, pp. 494–495.
[28] S.M. Qasim, S.A. Abbasi, B. Almashary, A proposed fpga-based parallel architecture for matrix multiplication, in: IEEE Asia Pacific Conference on Circuits and Systems, 2008 (APCCAS 2008), 2008, pp. 1763–1766.
[29] J.W. Jang, S. Choi, V.K. Prasanna, Area and time efficient implementations of matrix multiplication on fpgas, in: Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology, 2002, pp. 93–100.
[30] T.L.Y.Q. Ting Zhang Cheng Xu, M. Nie, An optimized floating-point matrix multiplication on fpga, Inf. Technol. J. 12 (2013) 1832–1838.
[31] N. Dave, K. Fleming, M. King, M. Pellauer, M. Vijayaraghavan, Hardware acceleration of matrix multiplication on a xilinx fpga, in: 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, 2007 (MEMOCODE 2007), 2007, pp. 97–100.
[32] Z. Jovanovic, V. Milutinovic, Fpga accelerator for floating-point matrix multiplication, IET Comput. Digit. Tech. 6 (4) (2012) 249–256.
[33] W. Wu, Y. Shan, X. Chen, Y. Wang, H. Yang, Fpga accelerated parallel sparse matrix factorization for circuit simulations, in: A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, T. El-Ghazawi (Eds.), Reconfigurable Computing: Architectures, Tools and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 302–315.
[34] S. Zhou, R. Kannan, V.K. Prasanna, Accelerating low rank matrix completion on fpga, in: 2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2017, pp. 1–7.
[35] E. Matthews, L. Shannon, A. Fedorova, Shared memory multicore microblaze system with smp linux support, ACM Trans. Reconfigurable Technol. Syst. 9 (4) (2016) 26:1–26:22, doi:10.1145/2870638.
[36] D. Buell, T. El-Ghazawi, K. Gaj, V. Kindratenko, High-performance reconfigurable computing, Computer 40 (3) (2007) 23–27.
[37] V. Sriram, M. Leeser, Fpga supercomputing platforms, architectures, and techniques for accelerating computationally complex algorithms, EURASIP J. Embedded Syst. (2009).
[38] M.C. Herbordt, Y. Gu, T. VanCourt, J. Model, B. Sukhwani, M. Chiu, Computing models for fpga-based accelerators, Comput. Sci. Eng. 10 (6) (2008) 35–45.
[39] T. Guneysu, T. Kasper, M. Novotny, C. Paar, L. Wienbrandt, R. Zimmermann, High-performance cryptanalysis on rivyera and copacobana computing systems, in: High-Performance Computing Using FPGAs, Springer, New York, 2013, pp. 335–366.
[40] H. Li, K. Li, J. An, K. Li, Msgd: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on gpus, IEEE Trans. Parallel Distrib.Syst. 29 (7) (2018) 1530–1544, doi:10.1109/TPDS.2017.2718515.
[41] H. Li, K. Ge Li, J. An, K. Ge Li, An online and scalable model for generalized sparse non-negative matrix factorization in industrial applications on multi-gpu, IEEE Transactions on Industrial Informatics (2019), doi:10.1109/TII.2019.2896634. 1–1
[42] H. Li, K. Li, J. An, W. Zheng, K. Li, An efficient manifold regularized sparse non-negative matrix factorization model for large-scale recommender systems on gpus, Inf. Sci. 496 (2019) 464–484, doi:10.1016/j.ins.2018.07.060.
[43] Y. Koren, R. Bell, C. Volinsky, Matrix factorization techniques for recommender systems, Computer 42 (8) (2009) 30–37.
[44] A. Mnih, R.R. Salakhutdinov, Probabilistic matrix factorization, in: Advances in Neural Information Processing Systems, 2008, pp. 1257–1264.

[45] Y. Koren, Factorization meets the neighborhood: a multifaceted collaborative filtering model, in: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2008, pp. 426–434.
[46] F. Ortega, A. Hernando, J. Bobadilla, J.H. Kang, Recommending items to group of users using matrix factorization based collaborative filtering, Inf. Sci. 345 (2016) 313–324.
[47] R.M. Bell, Y. Koren, Scalable collaborative filtering with jointly derived neighborhood interpolation weights, in: Seventh IEEE International Conference on Data Mining (ICDM 2007), 2007, pp. 43–52, doi:10.1109/ICDM.2007.90.
[48] R. Banik, The movies dataset, version 7, 2017.
[49] F.M. Harper, J.A. Konstan, The movielens datasets: history and context, ACM Trans. Interact. Intell. Syst. 5 (4) (2015) 19:1–19:19, doi:10.1145/2827872.
[50] J. Bennett, S. Lanning, et al., The netflix prize, in: Proceedings of KDD cup and workshop, 2007, New York, NY, USA., 2007, p. 35.
[51] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, Z. Zhang, High-level synthesis for fpgas: from prototyping to deployment, IEEE Trans. Comput.-Aided Des.Integr. Circuits Syst. 30 (4) (2011) 473–491, doi:10.1109/TCAD.2011.2110592.
[52] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, F. Morgan, Xilinx vivado high level synthesis: Case studies, in: 25th IET Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014), 2014, pp. 352–356, doi:10.1049/cp.2014.0713.
[53] F. Vahid, R. Lysecky, VHDL for Digital Design, Wiley, 2007.
[54] D. Kirk, W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, 2010.

Francisco Pajuelo-Holguera received the Master in Computer Engineering from the University of Extremadura, Spain, in 2019. He is currently hired researcher working on parallel implementations of algorithms and systems by means of reconfigurable hardware.

Juan A. Gómez-Pulido received the Ph.D. degree in physics, electronics specialty, from the Complutense University, Madrid, Spain, in 1993. He is currently professor of computer organization and design of processors in the Department of Technology of Computers and Communications, University of Extremadura, Spain. He has authored or co-authored 62 ISI journals, tens of book chapters, and more than two hundred peer-reviewed conference proceedings. His main research interests fall within hot topics on machine learning applied to big-data analysis, reconfigurable and embedded computing based on FPGA devices, optimization, and evolutionary computing.

Fernando Ortega received the Master in Artificial Intelligence and the Ph.D. degree in Computer Science from the Universidad Politcnica de Madrid, Spain. Currently, he is assistant professor at the same university. His research interests include information retrieval, natural computing and specially recommender systems. He belongs to the FilmAffinity.com research team working on the collaborative filtering kernel of the web site.

Jose M. Granado-Criado is a professor of computer architecture in the Department of Computer and Communications Technologies, University of Extremadura, Spain. He received his PhD in computer science from the University of Extremadura in 2009. Dr. Granado-Criado's main research interests are in the field of parallel processing and, particularly, in the use of reconfigurable hardware (FPGAs), GPUs, and embedded systems (SoC, MPSoC, etc.) in custom-computing applications, such as cryptography, evolutionary computation, and bioinformatics.