National Technical University of Athens

School of Electrical and Computer Engineering
department of information and computing technology computer and information systems

Accelerating the training of collaborative filtering algorithms for recommender systems using the HLS programming model for FPGAs

DIPLOMA THESIS

Georgios Nanos

Supervisor: Pneumatatos Dionysios
Professor NTUA

COMPUTER SYSTEMS LABORATORY

Athens, October 2021

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Information and Computer Technology Systems
Computer Systems Laboratory

Accelerating the training of collaborative filtering algorithms for recommender systems using the HLS programming model for FPGAs

DIPLOMA THESIS

Georgios Nanos

Supervisor: Pneumatatos Dionysios
Professor NTUA

Approved by the three-member committee of examiners on 4th October, 2021.

.....	.....	.....
Pneumatatos Dionysios	Soudris Dimitrios	Goumas Georgios
Professor NTUA	Professor NTUA	Associate Professor NTUA

Athens, October 2021

.....
NANOS GEORGIOS
Graduate Electrical Engineer
and Computer Engineer NTUA

Abstract

The rapid increase of the development that takes place on the field of Machine Learning last decade, has led to a big increment in the number of based streaming services and platforms that are being used by a very big majority of people. Nowadays, recommendation algorithms are at the core of such platforms as they provide their customers with personalized suggestions in order to minimize the amount of time spent to find anything that suits them. Systems that use those kinds of algorithms are referred as recommendation systems and have a wide appeal and usage in most applications in the market. By making use of algorithms designed for recommending items, recommendation systems utilise the previous choices of a user as well as his profile to find out the best item that matches his preferences, offering a better experience to its members. In the advent of big data, typical processor units have to operate on large chunks of data, resulting in lower performance of systems. Such approaches are time and energy wasteful as recommendation algorithms need to operate the same computational function on huge amount of data, requiring systems to offer high parallelization. An alternate solution, that may meet those needs is the implementation of such algorithms with high demands in reconfigurable architectures that have emerged last decade. FPGAs are a remarkable choice in order to achieve high performance with low energy consumption.

This thesis focuses in the research of recommendation systems that use collaborative filtering method, in order to design and accelerate the critical paths by implementing them for FPGAs, using HLS tools.

Keywords — Machine Learning, Recommendation Systems, Collaborative Filtering, Prediction Algorithm, FPGA, HLS, accelerator

Acknowledgements

With the completion of my thesis I would like to thank Mr.Pneimatatos Dionysios for the opportunity that gave me the opportunity to work on my thesis at the Computer Systems Laboratory.

I would like to thank Panagiotis Miliadis for the advice, ideas and guidance he offered me during my thesis. His support was crucial for its completion.

Finally, a big thank you to my family for always being by my side, supporting me to achieve my dreams.

Nanos Georgios
October 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Nowadays, the increase in the volume of data, particularly due to the boom in machine learning, has led to the development of applications that require exceptional performance and high computational requirements not found in everyday computers. It is therefore imperative to design such applications on more sophisticated systems to make them more efficient to run. In order to increase performance in terms of both execution speed and power consumption, special purpose embedded systems are used. Such a system is the reconfigurable FPGA (Field Programmable Gate Array) architecture that allows the development of high-intensity applications for specialized applications through a hardware description language. Historically, designing applications for FPGAs has been done through the use of Hardware Definition Language (HDL), which has most often been a deterrent to application programming. However, in recent years, High Level Synthesis-HLS tools have been developed that convert code from a high-level language to a hardware description language (HDL) making FPGAs quite attractive for accelerating applications and algorithms.

1.2 Subject of Diploma Thesis

In the context of this thesis, accelerators are developed for the optimization of collaborative filtering systems using HLS tools. In order to develop predictions and make a proposal to platforms using collaborative filtering recommendation systems, only information about users' scores on the items is involved and used, without the system being interested in additional data such as gender, ethnicity, age, etc. [Wan+20] Specifically, to generate the prediction results, collaborative filtering recommendation systems produce a weight matrix between users (User Based approach) or items (Item Based approach). This symmetric weight matrix represents the similarity between users or items in the given platform. Then it is a parameter and input for extracting the predictions and recommendations for each user in the prediction phase. For the weight calculation phase, which is the training part of the system, the Jaccard, Cosine, CosineIR, Euclidean and Pearson algorithms were used to estimate the similarity of the lines of a matrix by calculating the distance. The time profile analysis showed that the training part of the system is the most time consuming and needs optimization. The implemented accelerators aim to increase the performance of the system by speeding up the training. Comparisons of the accelerators developed for the five algorithms for the two approaches are made with off-the-shelf libraries on both CPU and GPU where FPGA is highlighted as an efficient solution both in terms of time and energy.

1.3 Organization of the Thesis

This thesis is divided into 4 technical chapters. A brief presentation of the contents of each chapter follows:

- Chapter 2 provides a brief introduction to machine learning and formulates some initial definitions and concepts. Recommendation systems are then defined with particular emphasis on collaborative filtering and the main stages of such a system, that of training and that of prediction, are presented. Next, the computational algorithms used in the training stage are defined before basic concepts of architecture and implementation techniques in the FPGA part are mentioned.
- Chapter 3 presents the initial implementations of the collaborative filtering recommendation systems and performs the temporal evaluation which shows which part of the system needs to be accelerated.
- Chapter 4 describes the initial accelerators that were implemented and defines the optimizations applied on them, emphasizing design techniques and the reasoning process followed until the final form of the accelerators.
- Chapter 5 focuses on evaluating the results and drawing some conclusions from the comparison of FPGA accelerators with CPU and GPU.
- Chapter 6 draws together the final conclusions of the thesis and lists some future extensions and modifications resulting from this thesis.

Chapter 2

Theoretical Background

2.1 Machine Learning

According to inductive learning, humans understand their environment through observation by the process of induction. Based on external stimuli, he creates mental models and is able to relate his experiences to them and react to any change that occurs dynamically, creating new structures and patterns. The result of the process is the formation of new causal models which are based on older ones and depend to a greater extent on the existing knowledge about a problem.

The exponential growth of data combined with increased computing power has led scientists to attempt to create complex computing systems capable of mimicking the way the human brain works to an extraordinary degree. Features such as inference and decision making or detection and estimation are now features that are extracted by machine learning algorithms, which take as input a large amount of information and achieve better results than conventional algorithms, and predict future trends more effectively by reacting like the human brain. Developments in pattern recognition and Artificial Intelligence have led to the creation of a separate branch of computer science. Machine learning enables computers to acquire data in order to draw conclusions without the need for explicit programming [Sim18]. The main goal in this discipline is to dynamically change computing systems by exposing them to new data and knowledge in order to drive the discovery of complex patterns in the flow of data with the ultimate goal of performing human tasks.

Scientific Fields of Machine Learning

Machine Learning is a subfield of Computer Science and Artificial Intelligence, since its purpose is to program "machines" to act in a dynamic way appropriate to the circumstances. In addition, it is an interdisciplinary field linked to disciplines such as statistics, information theory, game theory and optimisation. In the field of machine learning, computer tools and capabilities are used to perform tasks large-scale, data-intensive, computationally heavy tasks that far exceed human perception.

Machine Learning Algorithms Categories

The complexity of a system is inextricably linked to the types of inputs contained in the data set according to which the model is trained. An increase in the types of input to a system is associated with an increase in the complexity of the model, so models using larger datasets are most often more complex. It is therefore necessary to classify machine learning algorithms into the following categories [Sah20]:

- Supervised Learning: the computer system searches the model of matching inputs to the corresponding outputs according to a desired output for for each desired input.
- Unsupervised Learning: in unsupervised learning, the algorithm is given no input and is asked to identify the structure of the input data by searching for an approximate model.
- Reinforcement Learning: an algorithm tries to accomplish a specific goal through dynamic relationships it develops with external factors.

According to the desired output the categories are:

- Classification: each input is assigned to a class.
- Regression: a real function assigns a real number to each input making the algorithm a supervised learning task.
- Clustering: belongs to the category of unsupervised learning since a set of inputs must be divided into groups that are initially unknown.

2.1.1 Recommendation systems

The digital revolution in the 21st century has directly resulted in a rapid increase in the volume of information on the internet, leading to unexpected problems in its management. Information Overload has led to the average user not having immediate access to search information. Consequently, the issue of the urgent need to create computing systems capable of searching, classifying and clustering the available web information was raised. The initial tools developed were not interested in users' preferences alone and did not personalise the

information, but returned data common to all users. The development of innovative methods for searching and returning information to the user, taking into account the user's needs and preferences, led to the development of Recommendation Systems. Recommendation Systems (RS) filter the data and produce unique search output for each user based on their needs, considering behaviours and personal details such as gender, ethnicity, age etc., from an extremely wide range of information. Recommendation Systems are commonly found in web applications, such as Netflix, YouTube and Spotify, where ratings are used on items such as movies, videos and music in this case, or articles books news etc., by the same or similar users, forming a profile for each user through mathematical algorithms, which dynamically interacts with all other user profiles on each platform. Therefore, a multicomponent neural system is created, which predicts and recommends to the user an object that may be useful to them and that fits their preference profile, based on the generated similarity graph.

In essence, SS involve software tools in the form of ordered lists, which are called upon to produce personalised recommendations for each individual user based on preferences or other parameters, actively collecting various types of data which are separated and related:

- The Objects, i.e. the items proposed to the users of the platform. Their main characteristics are their value, complexity and usefulness.
- The Users. Aiming to personalize the proposals, each SA exploits a set of information about the users of the platform. This data is categorized in a variety of ways and the choice of which to use depends on the recommendation algorithm used in the system. From the behavioral data of the users, the associations between them are derived, indicating the similarity between them. A SA uses such knowledge to extract its recommendation from similar and trustworthy users.
- Transactions and relationships between users and objects. A transaction refers to the interaction between the user and the platform. In essence, the system enters a recorded information requested by the user. These transactions are data concerning essential information generated during human-computer interaction, which is stored and used as input to the proposal generation algorithms. The most prevalent form of transaction data collected by the SA is explicit rating and reporting in which, the user is asked to report their opinion about an object on the platform on a rating scale, usually numeric or unique.

The importance and usefulness of the suggestion schemes is evident both at the user and provider level and leads to a series of positive actions involving both. As far as the provider is concerned, the PSs lead to an increase in the number of products to be sold, the sale of differentiated items, and greater customer loyalty. As far as users are concerned, SAs primarily improve their experience of using a platform or service and at the same time, by finding the most suitable items for them, they gain a sense of understanding on the part of the provider.

Approaches

Various and interesting approaches have been developed for Recommendation Systems, by extracting different user data and assessing a variety of factors. The three main approaches involve [Gee+18]:

1. Collaborative Filtering - CF. This technique is based solely on the user's preference history for certain objects. It involves automatic prediction about a user's interests by collecting data and information from many different members with shared preferences. Predictions of his ratings on these are derived by taking into account similar users or similar objects or a combination of these.
2. Content-based Filtering- CBF. In addition to user scores on certain items, this approach requires specialized information material describing the users or items to form a data list. Content-based techniques match their service material with user characteristics. Specific filtering techniques ignore the contributions of other users, are not interested in matching the active user with the profiles of others, and base their systems' predictions on the user's personal information (e.g., ethnicity, age, gender, etc.) by combining it with further data about the objects (e.g., title, description, year, author, director, etc.).
3. Hybrid Filtering - HF. It combines multiple recommendation system techniques to extract recommendations leading to more accurate results.

Cold Start

This is an issue that is often problematic in the design of collaborative filtering recommendation systems. It concerns the problem that arises when new users enter the SA in the platform data list or new objects are added to the corresponding list. The issue that arises is that the algorithm without feedback can neither correctly predict the preference of the newly entered users nor has the ability to evaluate new items to be purchased, causing imbalances in the reliability of the SA and ultimately leading to less accurate recommendations.

Some ways of dealing with the cold start problem are:

- Ask the new user to rate certain items for the initial building of their profile.
- Ask users to explicitly state their preference for certain items and rate them.
- To suggest items based on the collected demographic and other data along with the corresponding interactions in order for the SAs to suggest items based on ratings provided by other related users with similar demographic or different type of information.

Data Sparsity

The problem of sparsity is one of the biggest problems facing the SA. Data sparsity has a big impact on the quality of the sentences [Al-+17]. The number of objects available on the web and entering platforms and services is growing exponentially. At the same time, most active users of a platform do not perform enough ratings on the objects to be evaluated. As a result, the user-object matrix 2.1.1 is sparse and contains many missing ratings that are requested to be found. Consequently, the prediction algorithm which is responsible for the performance of the system receives little feedback and does not have sufficient hardware to drive the system to accurate recommendations.

2.1.2 Cooperative Filtering

SAs using collaborative filtering techniques mostly produce better approximations than Content-based Filtering (CBF) techniques and do not require any knowledge about the objects. In collaborative filtering there are two main designs approaching the model from different perspectives. There is the model-based approach and the memory-based approach.[Mus+17] The model-based design uses the feedback to generate predictive models as a parameter to make recommendations for other objects. In memory-based methods, users' evaluations of objects are used directly to generate new recommendations. Collaborative filtering recommends items by recognizing behaviors of other users or objects with similar preferences. Two similar memory-based collaborative filtering techniques have been developed:

- USER-BASED
- ITEM-BASED

Collecting data, the SA uses the scores of active users as an evaluation parameter to guide it to correct estimates for all the upgradable items for each user's profile to finally recommend one or more items. Initially, systems using collaborative filtering method to overcome the scaling issue develop an offline item or user similarity matrix of items or users. This matrix contains values indicating the degree of similarity of all users or all objects. According to the USER-BASED method, a quadratic weight matrix between users is developed through mathematical algorithms, indicating the similarity relationship between them, depending on the scores they have registered themselves. Similarly, the ITEM-BASED method generates a quadratic matrix of weights between objects, which defines the similarity between them. Then, aiming to evaluate an item i for a user u, the system looks up the USER-BASED user similarity table to check whether the most similar users to the active user u have evaluated the item i. Those of the most similar users who have evaluated the item i are then counted through mathematical algorithms in deriving the final score prediction.

Similarly, when the ITEM-BASED method is followed to evaluate an item i of the active user u, the system is directed to the item weight table. The most similar items to item i are then searched in the item similarity matrix. It is then checked whether the most similar items to i have been rated by user u. Those of the most similar items to i have been rated by user u are collected are taken into consideration and using their weights

the final rating prediction of item i is performed. Therefore, the SA recommends products that are similar online according to the users' purchase history.

In conclusion, the basic idea in this case is to first build a profile around each user according to their history. The registered ratings play an important role since, users who have rated items in a certain way in the past are likely to give similar ratings to new items in the future.

2.1.3 Learning Stage

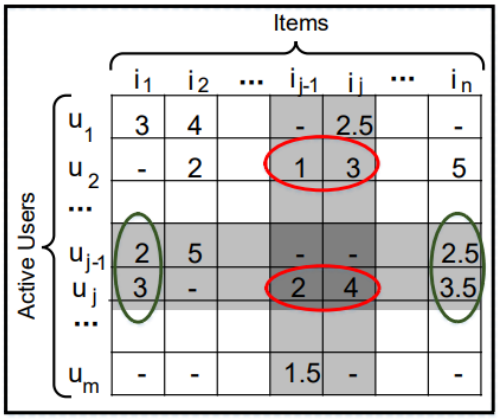


Figure 2.1.1: User-objects matrix

In general, all filtering techniques consist of two phases of training or learning and prediction. The initial dataset is stored in the user-object matrix as shown in the figure 2.1.1. According to the scheme 2.1.1 each row represents the ratings by an active user on all items, while each column shows all ratings of a particular item by different users. The matrix element i^{th} and j^{th} represents user i's evaluation of item j. If the user has not evaluated an item then the corresponding cell of the matrix remains empty. Collaborative filtering is divided into two stages, training (or learning) and prediction [Wan+20]. The training phase is used to develop the relationship of similarities between users or objects. Through the metric relations 2.1.1-2.1.5 the quadratic weight matrix is generated, which is then used as input to the prediction phase to derive the final score predictions [Wan+20].

Jaccard

$$w_{uvJaccard} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} \tag{2.1.1}$$

Cosine

$$w_{uvCosine} = \frac{|N(u) \cap N(v)|}{|\sqrt{N(u) \times N(v)|}} \tag{2.1.2}$$

CosineIR

$$w_{uvCosineIR} = \frac{\sum\limits_{i \in I} r_{ui} \times r_{vi}}{\sqrt{\sum\limits_{i \in I} {r_{ui}}^2 \times \sum\limits_{i \in I} {r_{vi}}^2}} \tag{2.1.3}$$

Euclidean

$$w_{uvEuclidean} = \frac{1}{\sqrt{\sum\limits_{i \in I} (r_{ui} - r_{vi})^2}} \tag{2.1.4}$$

Pearson

$$w_{uvPearson} = \frac{\sum_{i \in I} (r_{ui} - \bar{r_u}) \times (r_{vi} - \bar{r_v})}{\sqrt{\sum_{i \in I} (r_{ui} - \bar{r_u})^2 \times \sum_{i \in I} (r_{vi} - \bar{r_v})^2}} \tag{2.1.5}$$

Regarding the USER-BASED approach to equations 2.1.1-2.1.5 the w_{uv} represents the similarity between users $user_u$ and $user_v$. In 2.1.1 and 2.1.2, N(u) and N(v) represent the sets of items evaluated by user $user_u$ and $user_v$ respectively. In the 2.1.3-2.1.5 equations, r_{ui} and r_{vi} represent the rating of item i by $user_u$ and $user_v$, while I represents the set of items jointly rated by $user_u$ and $user_v$. Finally, $/barr_u$ and $/barr_u$ refer to the average score of the users $user_u$ and $user_v$.

Similarly, in the ITEM-BASED method in the 2.1.1-2.1.5 equations, w_{uv} represents the degree of similarity between objects $item_u$ and $item_v$. Unlike the USER-BASED approach, in this case in 2.1.3-2.1.5 r_{ui} and r_{vi} are defined as the score of object u $item_u$ and the score of object v $item_v$ by user $user_i$. Set I refers to the set of users who have rated both $item_u$ and $item_v$, while $\bar{r_u}$ and $\bar{r_u}$ refer to the average rating of the items $item_u$ and $item_v$.

2.1.4 Prediction Stage

Once the training stage is completed, the recommendation system moves on to the prediction stage. The prediction phase is used to calculate the score for each empty element of the 2.1.1 matrix. The USER-BASED collaborative filtering computes the empty positions of the matrix 2.1.1 via Eq:

$$P_{ui} = \frac{\sum_{u \in S(u,K) \cap N(i)} w_{uv} \times r_{vi}}{\sum_{u \in S(u,K) \cap N(i)} |w_{uv}|} \tag{2.1.6}$$

The prediction P_{ui} of the system for user u $user_u$ and item i $item_i$ assumes the computation of the set S(u,K), which is the set of K-most similar neighborhoods to user u $user_u$. N(i) refers to the set of users who have rated the item i $item_i$ to be evaluated. The term w_{uv} is the similarity of users u $user_u$ and v $user_v$ computed during the training stage of the system and finally, r_{vi} is the rating of the neighboring user v $user_v$ belonging to the set of neighboring users of user u $user_u$.

In addition, the ITEM-BASED approach calculates the empty positions of the user-object matrix from the equation:

$$P_{ui} = \frac{\sum_{j \in S(i,K) \cap N(u)} w_{ij} \times r_{uj}}{\sum_{j \in S(i,K) \cap N(u)} |w_{ij}|} \tag{2.1.7}$$

Like the USER-BASED method, the ITEM-BASED prediction P_{ui} of the system for user u $user_u$ and item i $item_i$ requires the computation of S(i,K), which in this case involves the computation of the set of K most similar-grounded objects to item i $item_i$, instead of previously computed users. The set N(u) refers to the set of objects evaluated by user u $user_u$ while the parameter w_{uv} represents the similarity between object i $item_i$ and j $item_j$. Finally, the term r_{uj} represents the evaluation of object j $item_j$ by user u $user_u$.

Once the prediction algorithm has completed all calculations that fall short of the user-object matrix 2.1.1, the highest scoring predictions developed can be proposed to users.

2.2 FPGA

2.2.1 Introduction to the FPGA

The FPGA (Field Programmable Gate Array) is a digital integrated circuit that contains programmable logic blocks that are interconnected with also programmable connections. The first reprogrammable logic

device was created in 1984 by Xilinx. In parallel, Altera developed a programmable device, which evolved into the EP1200 which was the first high-density programmable logic device [Tri15]. The most important aspect in an FPGA, which is given the term Field Programmable in its name where it denotes programming in the field and is due to its particular flexibility and fast implementation of designs, is that unlike other ICs where their functionality does not change after manufacturing requiring re-manufacturing on silicon an FPGA allows reprogramming instead of being limited to any predefined hardware function, greatly reducing design time and cost.

2.2.2 Architecture of the FPGA

FPGAs consist mainly of programmable SRAM interfaces and multiple arrays of Configurable Logic Blocks (CLBs)[GK19].

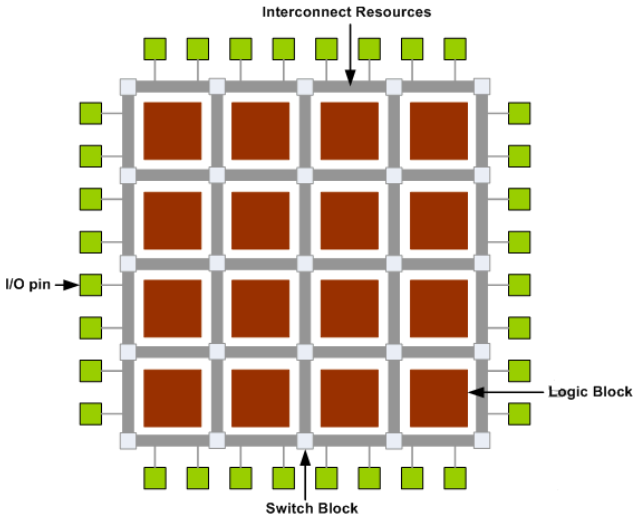


Figure 2.2.1: Basic FPGA Structure

Their programming is mainly based on the SRAM cells they contain. The SRAM bits are distributed throughout the FPGA and program the various blocks and internal interconnects, essentially forming a sliding register, where a chain or sequence of chains of bits slides from the input of the sliding register to the output as shown in the figure 2.2.2.

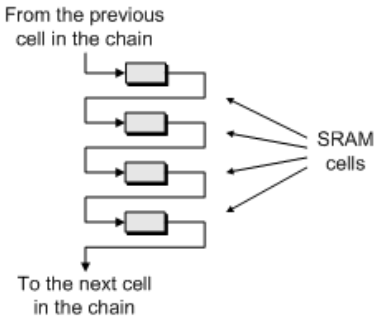


Figure 2.2.2: Simplified SRAM cell chain

A CLB consists of 4 slices and each slice consists of 2 logic cells, which is the elementary programming unit and is responsible for executing basic logic functions of the form $y = aOR(bANDc)$. The basic form of the logic cells that make up the CLB is shown in the figure 2.2.3.

As shown in the figure 2.2.3, a CLB consists of a 4-input LUT (Look Up Table), which plays a key role in the performance of the FPGA [AR04], a multiplexer and a flip-flop with positive or negative edge triggering

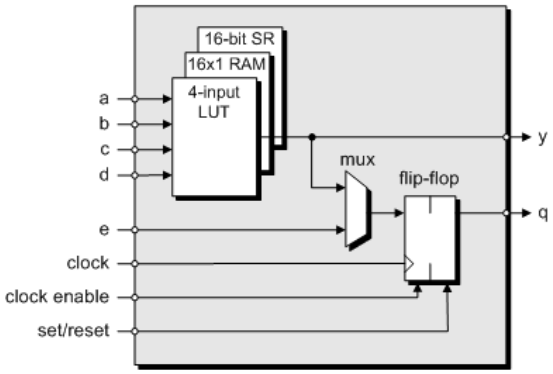


Figure 2.2.3: Simplified logical cell format

or latching capability. A 4-input combinational function, a 16-position sliding register or a 16-bit distributed memory can be implemented by the LUT. Logic cells within a slice communicate rapidly, slices within a CLB communicate a little slower, while communication between CLBs can be noticeably slower by making long, slow interconnects, especially if they are completely remote. FPGAs in addition to CLBs and LUTs have multipliers and organized BlockRAMs (BRAMs) which are hardwired, can be used in design and programming as stand-alone units, interconnected with CLBs but cannot be rearranged. Access to the memory can be achieved on each operating cycle and its interconnection via 2 dual-port ports achieves parallel access to two different areas, on the same clock cycle.

In addition, IOB Input Output Blocks (IOB Input Output Blocks) are contained at the edges of the FPGA which as shown in the figure 2.2.1 are organized in banks which are used with a specific separate I/O pattern. The support of tristate logic (input, output or input and output) lies in the design. The IOBs are responsible for the FPGA's communication with the host and with external devices.

The DSP (Digital Signal Processor) blocks take on a high computational load, making up the fundamental building block ALU (Arithmetic Logic Unit), the digital circuitry that performs arithmetic and logic calculations at very high operating frequencies by combining adders, subtractors and multipliers.

The Interconnection Network handles the interconnection between the CLBs and the I/O blocks, as well as the connection of the CLBs to each other.

Routing Channels are responsible for the vertical or horizontal interconnection of the Logic Blocks.

Various clock frequencies and/or phase differences can be generated by programming Digital Clock Managers (DCMs) which generate various clock frequencies from the clock input from the external crystal.

2.2.3 FPGA and CPU

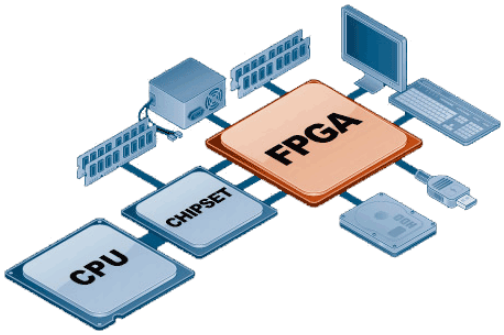


Figure 2.2.4: FPGA and CPU

One of the major advantages of FPGA is the high levels of parallelism achieved in process execution. A number

of applications and systems covering a fairly wide range of scientific fields benefit from being implemented in FPGA due to the parallel architecture offered. The most common FPGA applications involve image and video processing, Artificial Intelligence (AI), electronics, wired and wireless communications. Most of these areas are changing very rapidly as application requirements evolve and new protocols and standards are adopted. FPGA is a particular alternative as it allows manufacturers to design systems that can be updated when necessary, reconfiguring it to support new algorithms that are created or altering existing ones if the needs of the effector change. This flexibility is difficult or even impossible to achieve with ASIC (Application Specific Integrated Circuit) [KR07]. The need to manage and process more data more effectively and efficiently highlights several weaknesses of general purpose processors and creates challenges for new computing resources. CPUs are devices which, in general, in order to perform functions or algorithms, execute a sequence of sequential instructions serially. What makes FPGAs a particularly attractive solution over traditional processors is that they can be configured as parallel processing devices. An entire algorithm can be executed in one clock cycle 2.2.7 or in far fewer cycles than a processor needs. In addition, it is possible to create multiple Compute Units leading to extremely higher performance speeds than can be achieved even with a multi-core processor. Unlike common general purpose processors, FPGAs have independent processing nodes (PEs) assigned to a dedicated part of the chip and can work autonomously without any other influence from other logic blocks. In an FPGA o compiler adapts the available computing resources to the needs and requirements of the application at hand. 'When an application is running on an FPGA o compiler uses the computational structures and available resources according to the needs of the same application. Therefore a change in the application or design implies a change in the circuitry implemented in the FPGA. In contrast, the compiler of a processor is responsible for adapting the application in question to its computational resources. In a general-purpose processor, the application itself is adapted to the available computational resources of the processor through the compiler. In addition, several times the clock frequency of a system translates into higher execution speed. However, when the comparison comes to a processor and an FPGA, this does not seem to be the case, since typical clock frequencies of a processor are 2 GHz while a typical clock frequency value of a classical FPGA is around 400 MHz [Cho13]. Considering the above, it seems that FPGAs can outperform the classical processor in speed by taking advantage of their architecture and achieving highly parallelizable application executions. A typical processor follows certain steps to execute an instruction [Iva02] 2.2.5.

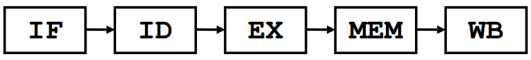


Figure 2.2.5: Command execution steps

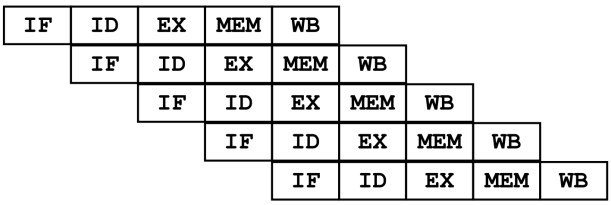


Figure 2.2.6: Processor execution command channeling

Several times, in program execution, there are data and instruction dependencies in a program flow that make the above instruction pipelining impossible, leading processors to alternatives such as stall and forwarding to avoid hazards. The above execution steps of 2.2.5 instructions are not performed on an FPGA. When compiling kernels in an FPGA, they are placed on top of it circuit-wise, data dependencies are analyzed, executions of neighboring instructions are grouped together, and the process is parallelized as much as possible. Independent kernel operations are executed in parallel while interdependent operations are executed by instruction pipelining. In non-independent instructions, the big advantage lies in the fact that, unlike the classical processor architecture, instruction pipelining can be achieved by forwarding data from one stage to another without requiring a stall. Therefore, the conclusion is that FPGAs are significantly superior in terms

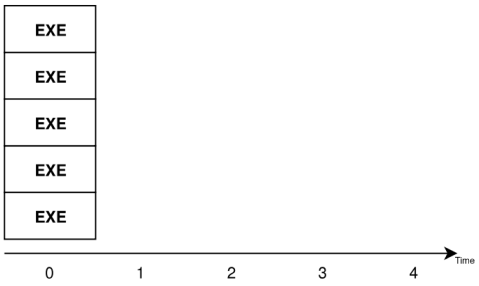


Figure 2.2.7: Simultaneous execution of instructions on the FPGA

of performance, processing data much faster compared to CPUs, while achieving low power consumption. Therefore, FPGAs have grown explosively in the last decade, in terms of computing performance, as they can achieve true parallel execution in highly complex operations better than CPUs. Along with the evolution of FPGAs and their capabilities, due to Moore's law potentially becoming obsolete in a few years, the performance gap between FPGAs and CPUs is increasing.

2.2.4 FPGA and GPU

Usually after evaluating and benchmarking FPGAs with processors, comes the comparison with GPUs. Generally, FPGAs have excellent speeds on fixed-point algorithms to take advantage of their advantages, while graphics cards excel at speeding up algorithms based on floating-point operations [AMY09]. When comparing the performance of the two devices, it is necessary to take into account the energy consumption of both devices in order to get a more complete picture of the trade-offs offered by their different architectures. Typically, FPGAs are significantly superior to GPUs in terms of energy efficiency, since they are able to offer huge processing capabilities and successfully manage intense computing loads without being a wasteful energy choice, fully reducing the need for thermal management and protection, which is a necessary condition for smooth operation in the execution of complex processes on graphics cards. Then a disadvantage of GPUs is that they do not provide the ability to predict latency, which is a very useful tool when designing applications on the FPGA. Therefore, to achieve software acceleration FPGAs are cost effective in terms of energy savings and GPUs are cost effective and are a better choice within the hardware cost but in the cost of development and implementation of the solution FPGAs prevail.

2.2.5 Programming of FPGAs

Designing on an FPGA was for a reasonable time one of its disadvantages and one of the main differences between it and the general purpose processor. The programming of an FPGA involves the precise configuration of the routing channels in order to achieve communication between the desired CLBs in the most efficient way to implement an application. Traditionally, FPGA programming and configuration is achieved through the use of Hardware Definition Language-HDLs, similar to that used for an Application Specific Integrated Circuit (ASIC), and is intended to perform a computationally intensive task [RTT99]. However, the completion of this process as well as the code of this programming mode is quite difficult to debug is very demanding and time-consuming, unlike the classical computing platforms and therefore, it was a deterrent for an engineer who mainly deals with high-level languages such as C/C++. This fact, in recent years, has led to the development of new optimized methodologies and the creation of High Level Synthesis-HLS tools, through which code is converted from a high-level language such as C/C++ to a hardware description language (HDL) such as VHDL or Verilog. High-level synthesis tools developed on platforms such as Xilinx's SDSoC (Software Defined System On Chip) offer advantages such as [Xil21]:

- Developing applications using a high-level language requires a lot of less time.
- Ability for Design Space Exploration. An efficient synthesis system is able to produce several designs and solutions for the same specification in a reasonable amount of time. This allows the designer to explore the various trade-offs between cost, speed and power consumption and generated circuit area

and thus, take an existing design and produce an implementation with the same functionality according to his needs at a given time.

- Finding errors and verifying the operation of an application is faster in a high-level language level and logical errors are more easily caught compared to a hardware description language (HDL).
- High-performance architectures and algorithm accelerators can be more easily developed using optimization directives.
- Making IC (Intergrated Circuits) technology available to more people. As more and more design expertise moves into the synthesis system, it becomes easier for a non-expert to program a chip that will have a given set of specifications.
- Using high-level synthesis, portable code is developed, which can be used to program different FPGAs just by changing some parameters in the high-level synthesis tool.

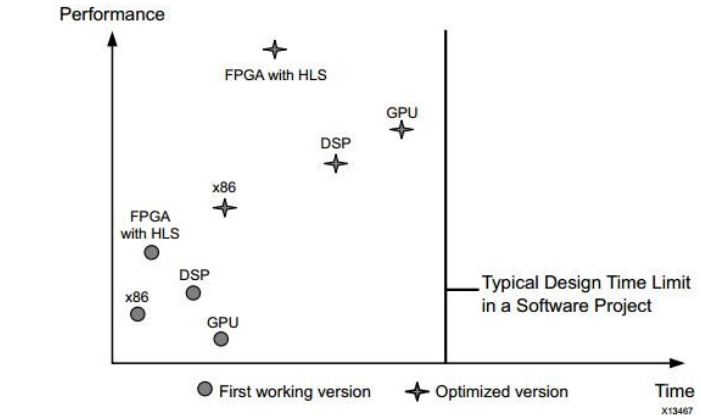


Figure 2.2.8: Design time and application performance using the Vivado HLS compiler

The software and hardware disciplines are coming closer together through high-level synthesis tools offering advantages and allowing software engineers to easily use FPGAs as hardware accelerators for faster execution of computationally demanding tasks, while hardware engineers are able to improve their efficiency by using a high-level language in the context of the application they are developing.

2.2.6 The SDSoc Environment

Hardware programming is made faster and more cost-effective through the Xilinx SDSoC (Software Defined System On Chip) tool, which offers a high-level programming language (C, C++) for the development of FGPA effects, which allows the exploitation of the advantages of the hardware without requiring special expertise on it. At the same time, SDSoC using C/C++ enables further performance improvements of SoC and MPSoC hardware and software devices, and offers an analytical report (profiling) at the architecture level for the implementation being designed.

Specifically, the functions it provides concern:

- A brief account of the Performane Estimation and the consumption of available resources (Area Estimation).
- Report on the automated runtime organization of memory, bus and Initation Interval.
- Running C/C++ applications on a fully functional Zynq SoC and MPSoC system.
- Performance acceleration at the programmable logic level generated by the ARM processor and FPGA bitstream.
- Information about the communication between PS (Processing System) and PL (Programmable Logic).

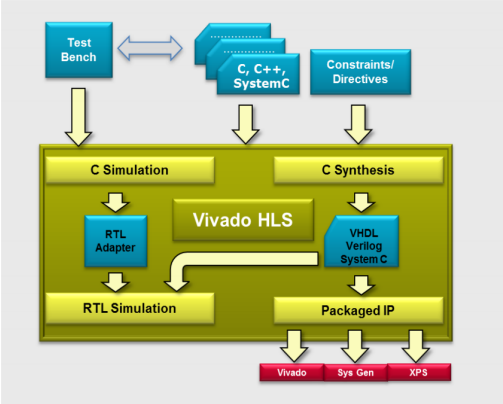


Figure 2.2.9: High Level Synthesis by using the Vivado HLS

Tools such as SDSoc provide the user with the ability to influence the generated architecture through a system of optimization directives and pragmas. As shown in Fig. 2.2.9, in addition to the system specification and application acceleration directives, a test bench is developed , which is used for simulation and testing of the system, making it particularly useful in correcting logical errors (C simulation) and at the implementation level (RTL simulation) to address any bugs resulting from the tool's synthesis process.

The SDSoc (Sofware-Defined System On Chip) environment is an Eclipse-based integrated development environment (IDE) for heterogeneous embedded systems [SDS15]. SDSoc offers a C/C++ environment, which facilitates the implementation of applications. It includes an integrated system aimed at optimizing them, which involves an automated Software Acceleration system. Initially the code of the system application is written by the designer in a high-level language (C/C++), with the developer specifying a convoluted platform from those available in the tool as well as a subset of functions such as frequency of operation, top-level function and others.

SDSoc offers the possibility of cross-compilation with integration of C/C++ programs in PL (Programmable Logic) and ARM processors of the FPGA. During compilation, the tool develops and combines Hardware and Software code which has the ability to keep up with the program's prevailing semantics and achieve high synchronization between Hardware and Software processes while allowing pipeline at runtime and communication between different Compute Units. To achieve high performance in any application, the tool has the ability to invoke multiple Hardware functions as well as multiple Compute Units.

At the same time, to arrive at the desired bitstream, SDSoC invokes the Vivado Design Suite tools and runs the hardware functions together with Vivado High-Level Synthesis (HLS) in PL (Programmable Logic), where it produces a complete hardware system based on the selected platform including DMA (Direct Memory Access), interconnects, hardware buffers, IP intergrator, IP library, and other tools to develop the final bitstream. The SDSoc IDE (Integrated Development Environment) supports a software development paradigm with compiling, debugging as well as fast performance estimation. Figure 2.2.10 shows the system flow. It includes iteration in steps up to generated system achieves proper performance.

In addition, in the context of developing an application there are certain limitations in the capabilities of the SDSoC compiler that the designer has to deal with and solve. The tool compilers cannot compile into a hardware description language (HDL) any structure or process supported by high-level languages such as C/C++. In general there are two categories of such structures, those that are not supported at all and those that are partially supported. The unsupported structures are:

- Dynamic objects: Anything that is to be compiled into a hardware description language is required to be of known size at compile time and bound to the FPGA's BRAM. Therefore, calls such as malloc(), alloc(), free(), new and delete are not supported.
- System calls: there is no operating system on FPGAs. Vivado HLS automatically ignores the most common system calls (abort(), atexit(), exit(), fprintf(), printf(), putchar() and puts() without generating any error.

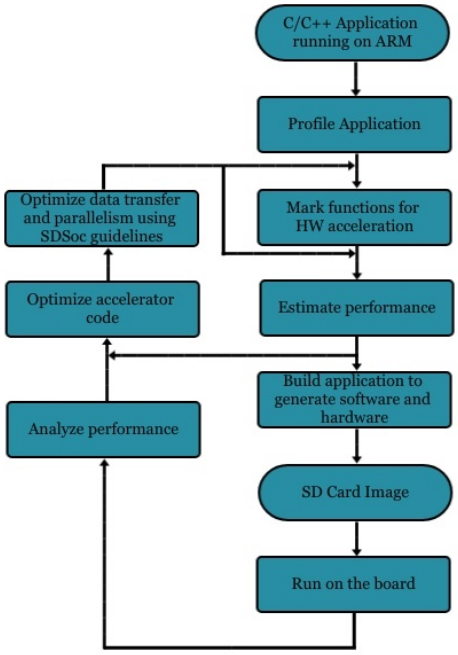


Figure 2.2.10: User Design Flow SDSoc

The partially supported ones are:

- Indicators: Pointer arrays are not supported and many address operations (pointer arithmetic) are not composable leading to compile-time errors.
- Memory functions: Functions such as memcpy() and memset() are supported but only when fixed values are given as their arguments during programming.

When the SDSoC tool's compiler generates the desired bitstream, it also generates useful files consisting of information about the designed implementation. The three main metrics that must be taken into account to provide feedback and continuous design and implementation improvement are:

- Latency: It is the execution time of the design, from input to output of the system
- Area: This is information about the available and consumed resources of the design mentioned in the paragraph 2.2.2
- Initiation Interval II: It is the number of cycles needed by the hardware function designed into the FPGA before it can accept new input

The above main parameters are involved in the design and trade-offs are made between them in the context of optimizing and accelerating an application as shown in the diagram 2.2.11.

2.2.7 Optimizations of Application Code

RTL synthesis is controlled by a set of directives called pragmas [Xil21], which are defined during source code design either directly or in the form of compiler directives, or in a separate file. HLS performs some default optimizations that parallelize as much as possible certain implementation processes, however in order to achieve optimal performance and blazing speeds the design must be designed around certain processes that are conducive to FPGA programming, in combination of course with the efficient use of pragma instructions that interfere with the architecture.

The Vivado HLS tool enables some optimizations on the functions implemented in the hardware. These optimizations are called directives and guide the compiler in determining how the application functions work:

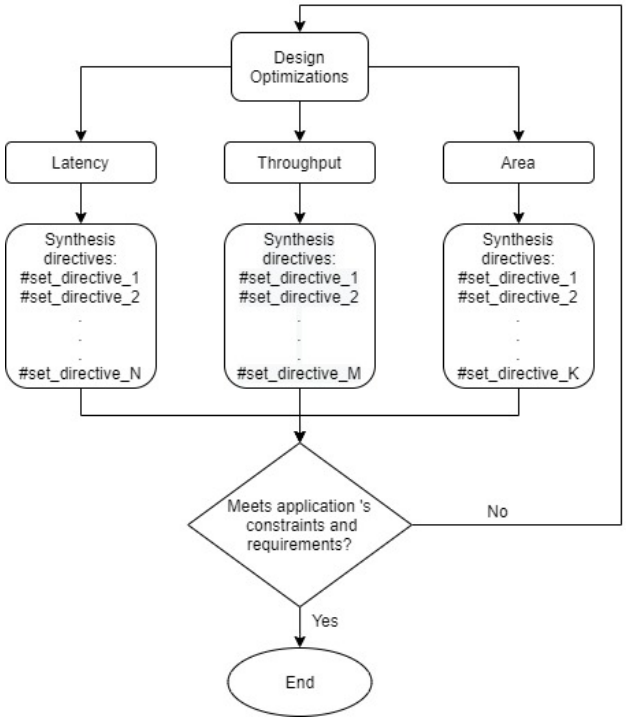


Figure 2.2.11: Design Optimizations and trade-offs

- Latency: The possibility is given to set a maximum and a maximum threshold on latency for the hardware function.

- Instatiate: It concerns optimizations of local functions.

- Interface: create an interface protocol that enables access to the input and output ports of the final RTL design.

- Inline: There is overhead in clock cycles for input and output functions, so a hierarchy of the RTL design is lifted, raising one level of the individual computational part applied to improve latency and throughput. The inline transformation can replace a function call by creating a copy of the function's internals within the hardware function. In this way, the function that used the inline instruction that has used the inline no longer appears as a separate level of hierarchy.

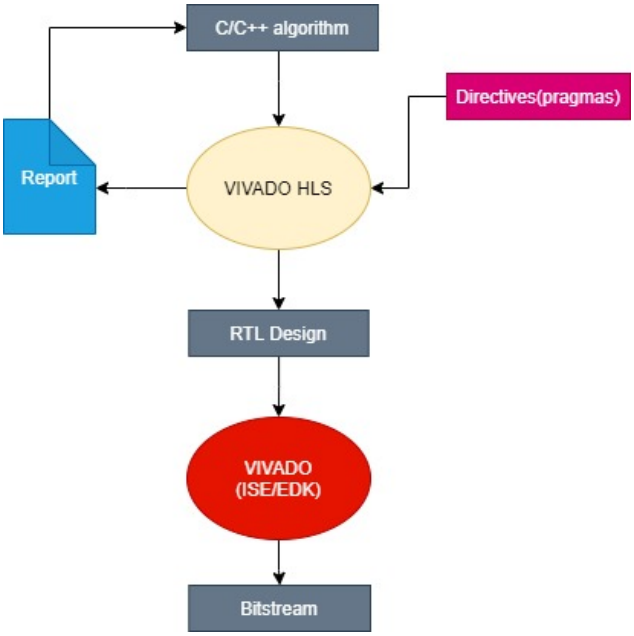


Figure 2.2.12: Design Flow of HLS

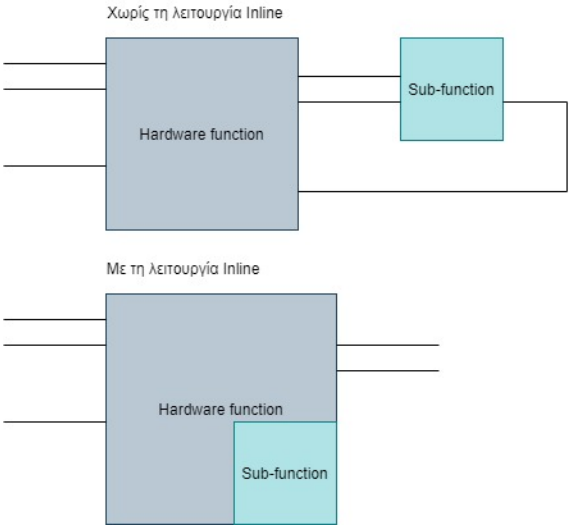


Figure 2.2.13: Inline Implementation

- Pipeline: allows the parallel execution of the individual functions of the function. It attempts to reduce the start interval of the body of a loop, so that new inputs are received before the previous ones are completed and processed, significantly improving throughput.

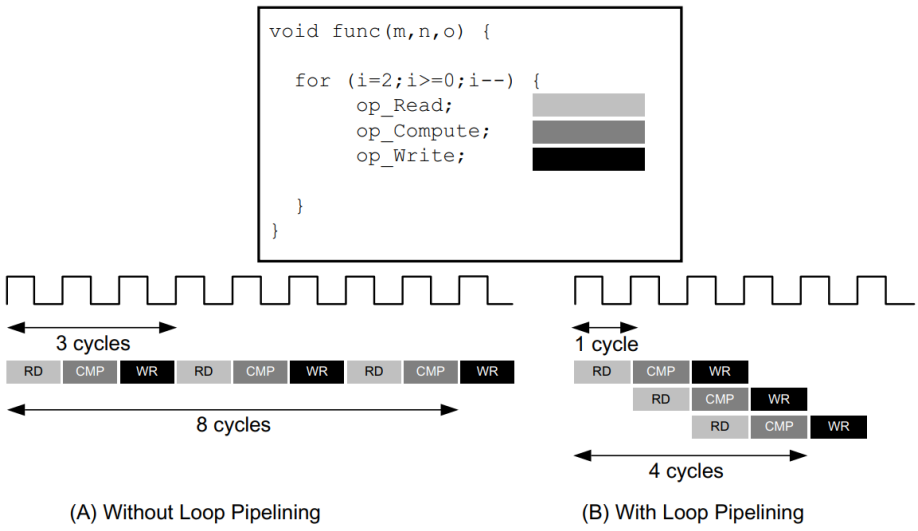


Figure 2.2.14: Pipeline Implementation

- Dataflow: Increases the performance of the application by seeking to achieve pipelining at the task level of the algorithm. This directive is not used to tunnel instruction execution, but rather to tunnel entire computational tasks. This transformation is a sequential function description and creates an architecture that has the ability to run certain functions in parallel. It usually increases the available hardware resources considerably however it improves the throughput and latency of the design.

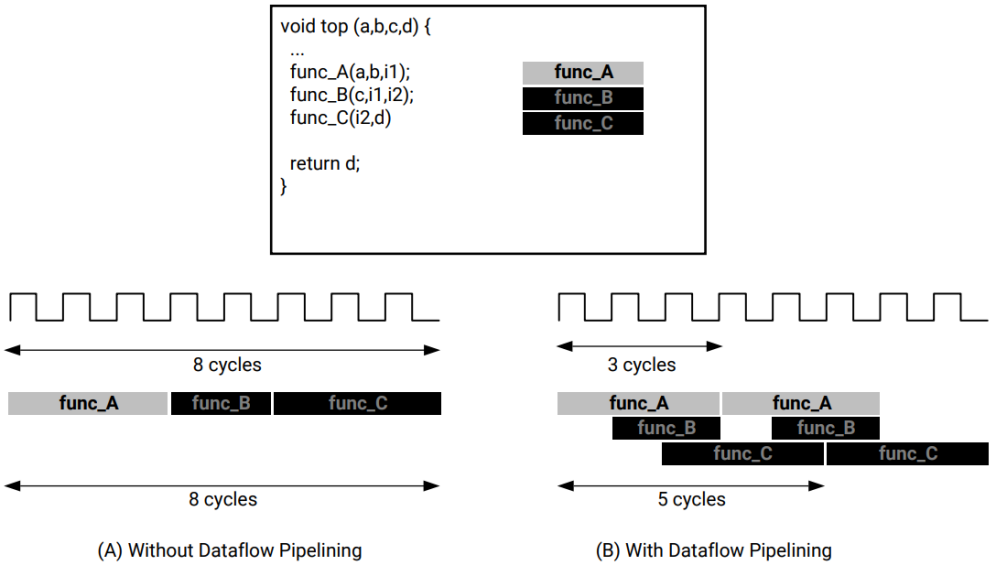


Figure 2.2.15: Dataflow Implementation

During the development and optimization of the application algorithm, the hardware functions designed in C/C++ often need to be accelerated due to delays created either by a set of iterative loops that make up the structure of the algorithm, or by the format of the data in the FPGA's BRAM memory (e.g. tables) or by the transfer of data from the PS to the PL. iterative loops in hls and their hierarchy, the format of the data handled by the application and the optimal data transfer has a great impact on both performance and the commitment of available resources. Most of the time the designer is forced to focus on a specific optimization direction (performance, power consumption, resource utilization, circuit area) using the following pragmas

appropriately [Xil21]:

- Array Map: To reduce the use of BRAM (Block Ram) in the FPGA the tool merges multiple tables defined in the design into one larger one. In most libraries RAMS models have predefined sizes. Therefore when each small table binds a separate block of memory, space is wasted unnecessarily and the final design consumes FPGA resources that could be used otherwise. If the implementation consists of many small tables, it is useful to map them as a single table before defining them in the hardware function to reduce the hardware resources consumed by the application. Two modes of mapping are supported, horizontal, which performs a concatenation of the small tables, and vertical, which defines a new table with more bits and concatenates the initial words of the tables.

- Array Partition: it partitions an array into smaller subarrays, in the ways (block, cyclic, complete) shown in the figure 2.2.16.

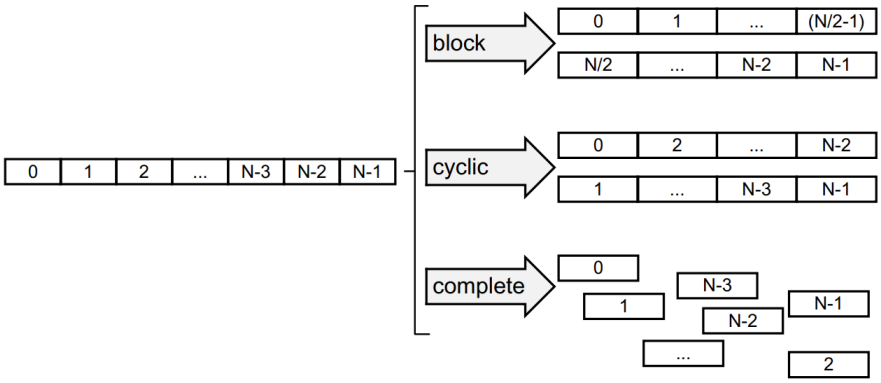


Figure 2.2.16: Array Partition Implementation

Most of the time, memories have a limited number of I/O, read and write ports and thus, limit the load/stores performed in a memory intensive algorithm. Therefore, in this way, the hardware accelerator can read multiple data simultaneously and is not limited to only the two ports of a BRAM memory, avoiding memory bottlenecks.

- Array Reshape : This transformation merges array cells at the bit level, increasing the length of the memory word. It is a handy instruction in transferring multiple data, from PS to PL, combining array partitioning with the vertical array mapping transformation.

- Unroll : Performs a loop unwrap, creating multiple copies of a loop to reduce the number of iterations. In this way, when there are no data dependencies between iterations, new instructions are executed in parallel from different parts of the FPGA. The ability to partially or fully unfold iterative loops can increase performance explosively.

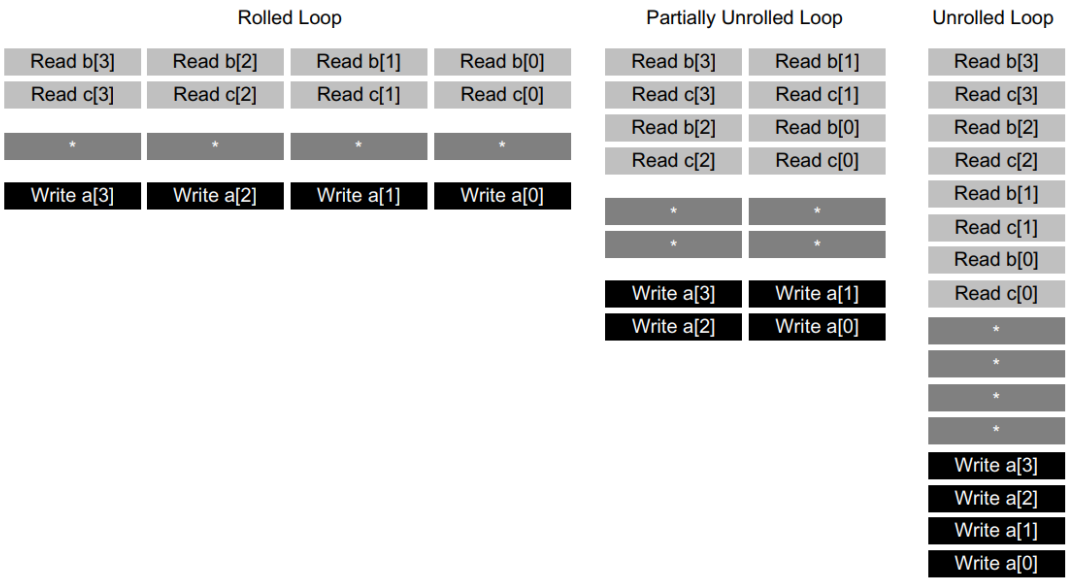


Figure 2.2.17: Implementation of Loop Unrolling

- Merging: The existence of several consecutive loops can sometimes create delays and unnecessary clock cycles. Merging of consecutive loops contributes to reduce latency by allowing further performance improvement.

- Data Access Pattern: Defines the way in which data is accessed and transferred from the PS to the PL.

- Data Mover: The Data Mover is responsible for the transfer of data between the PS (Processing System) and the Accelerators as well as between the Accelerators. The Data Motion Network is made up of the following elements :

  1. To Hardware Interface of the accelerator (A)

  2. The data movers between the PS and the accelerator or between the accelerators (B)

  3. The system memory ports on the PS (C)

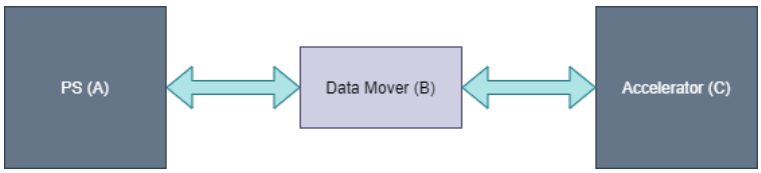


Figure 2.2.18: Data transfer between PS and PL

Of course, depending on the design, the speed requirements and the size of the data to be transferred, a different intermediary is selected for data transfer. For optimal performance the interface defined should be based on the properties and size of the data being transferred as much as possible.

SDSoC Data Mover	Vivado IP Data Mover	Accelerator IP Port Types	Transfer Size	Contiguous Memory Only
axi_lite	processing_system7	register, axilite		No
axi_dma_simple	axi_dma	bram, ap_fifo, axis	< 8 MB	Yes
axi_dma_sg	axi_dma	bram, ap_fifo, axis		No (but recommended)
axi_fifo	axi_fifo_mm_s	bram, ap_fifo, axis	(≤ 300 B)	No
zero_copy	accelerator IP	aximm master		Yes

Figure 2.2.19: Properties of the SDSoC data mover

- Streaming: In FPGA when data access is sequential and used once (FIFO) it is very useful to read in streams. By default, arrays are implemented as memory elements, unless the complete partitioning reduces them to individual registers via the array partition complete instruction mentioned above. In the tool, it is possible to use the FIFO interface, instead of RAM, by specifying the array as streaming.
- Tripcount: During design it is possible that the calculation of latency cannot always be defined by the compiler because, in a loop, the number of iterations may be a function of some other variable. In such cases, to enable the upper and lower bounds of latency to be defined, the programmer has the option of setting the lower and higher value of this parametric variable.

Design of Computer Units

When designing an application ideal performance can be achieved by applying multiple calls to the hardware accelerator being implemented. It is possible to configure the application to run by pipelining these calls. Further increase in parallelism is achieved by mechanisms that create multiple Compute Units of the accelerator with the "resource" transformation [Xil21]. Each new Compute Unit deployed doubles the resources occupied in the FPGA. By combining the "async-wait" mechanism it becomes possible to achieve high levels of synchronization of the compute units created. Of course, as in any multi-threaded model, enough attention needs to be paid to the details of the synchronization and the sharing of data across the different Compute Units to avoid deadlocks. Also, it is particularly important to consider the issue of load balancing in the management of multiple compute units. In computer science, the term load balancing is a subject of research in the field of parallel systems and refers to the process of distributing a set of tasks and data across computing units in order to make their overall processing more efficient. Load balancing can optimize response time and avoid uneven loading of some compute nodes while other compute nodes remain idle. Load balancing and aims to distribute the workload of the computational nodes generated, so as not to overload any of them with extra tasks and delay the system response. As mentioned above, due to the small BRAM memory available in FPGAs, the management and transfer of data from the host to the various Compute Units needs special attention so that each Compute Unit has access to a specific bandwidth of the host memory when reading and transferring data simultaneously, each in its own BRAM and contributes equally to the system performance.

Chapter 3

Algorithm Implementation and Temporal Evaluation

3.1 Introduction

First, the implementation of the recommendation systems for the various algorithms for calculating the similarity coefficients was carried out. The recommendation systems were designed based on the USER-BASED approach 2.1.6 and based on the ITEM-BASED approach 2.1.7 for five different approaches to compute the similarity w_{uv} which are Jaccard, Cosine, CosineIR, Euclidean and Pearson 2.1.1-2.1.5. In the first stage the writing of the algorithms was done in C/C++ for classical processors and then the transfer to FPGA was performed with some changes in the structure of the original code that were necessary to take full advantage of the FPGA benefits. Then, the analysis of the time profile for the different stages of the algorithms was performed to make it clear which part of the overall system needs acceleration.

The MovieLens dataset [HK15] was also chosen to evaluate the performance of the implementations. The dataset contains user ratings on movies on a scale of 1-5. The dataset has the format userID::movieID::rating::timestamp and more specifically, it contains in the first column the user id, in the second column the movie id, in the third column the rating 1-5 of the user with userID on the movie with movieID, and the fourth column the timestamp of the specific rating, information that we do not exploit in this design. Useful information, is that each user has rated at least 20 movies. In particular, we made use of Movielens-100k and Movielens-1m containing 100,000 and 1,000,000 user ratings on movies respectively.

3.2 Agorithm functions

The basic functions developed to calculate the similarity between users are: (The code structure for calculating the similarity between objects is similar for the five algorithms):

Jaccard

```
float sim_Jaccard[N][N];
int set_of_U, set_of_V, set_of_UV;

for(int i=0;i<num_user;i++){
  for(int j=i+1;j<num_user;j++){
        set_of_U=0;
        set_of_V=0;
        set_of_UV=0;
        for(int k=0;k<num_item;k++){
            if(data[i][k]!=0)
                set_of_U++;
            if (data[j][k]!=0)
                set_of_V++;
            if((data[i][k]!=0)&&(data[j][k]!=0))
                set_of_UV++;
        }
        if(set_of_UV==1) //if only one item is rated by 2 users then their similarity is
    zero
            set_of_UV=0;
        sim_Jaccard[i][j]=set_of_UV/fabs(set_of_U+set_of_V-set_of_UV);
        sim_Jaccard[j][i]=sim_Jaccard[i][j]; //(i,j)=(j,i)
    }
}
```

Listing 3.1: Calculating Jaccard Similarity between users

Cosine

```
float set_of_U, set_of_V, set_of_UV, sim_Cosine[N][N];

for(int i=0;i<num_user;i++){
    for(int j=i+1;j<num_user;j++){
        set_of_U=0;
        set_of_V=0;
        set_of_UV=0; //variable to check if only 1 item is common between 2 users
        for(int k=0;k<num_item;k++){
            if(data[i][k]!=0)
                set_of_U++;
            if (data[j][k]!=0)
```

```
                set_of_V++;
            if((data[i][k]!=0)&&(data[j][k]!=0))
                set_of_UV++;
        }
        if(set_of_UV==1) //if only one item is rated by 2 users then their similarity is zero
            set_of_UV=0;
        sim_Cosine[i][j]=set_of_UV/sqrt(set_of_U*set_of_V);
        sim_Cosine[j][i]=sim_Cosine[i][j]; //(i,j)=(j,i)
    }
}
```

Listing 3.2: Calculating Cosine Similarity between users

CosineIR

```
float product_of_ratings_U_V, sum_of_squares_U, sum_of_squares_V, sim_CosineIR[N][N];
int set_of_UV;
for(int i=0;i<num_user;i++){
    for(int j=i+1;j<num_user;j++){
        set_of_UV=0; //variable to check if only 1 item is common between 2 users
        product_of_ratings_U_V=0;
        sum_of_squares_U=0;
        sum_of_squares_V=0;
        for(int k=0;k<num_item;k++){
            if((data[i][k]!=0)&&(data[j][k]!=0)){
                set_of_UV++;
                product_of_ratings_U_V+=data[i][k]*data[j][k];
                sum_of_squares_U+=data[i][k]*data[i][k];
                sum_of_squares_V+=data[j][k]*data[j][k];
            }
        }
        if(set_of_UV==1)
            product_of_ratings_U_V=0; //if only one item is common between 2 users then their similarity is zero
        sim_CosineIR[i][j]=product_of_ratings_U_V/sqrt(sum_of_squares_U*sum_of_squares_V);
        sim_CosineIR[j][i]=sim_CosineIR[i][j]; //(i,j)=(j,i)
    }
}
```

Listing 3.3: CosineIR Similarity calculation between users

Euclidean

```
float sum_of_diff_ratings, Euclidean_sim_one_or_zero, sim_Euclidean[N][N];
int set_of_UV;

for(int i=0;i<num_user;i++){
    for(int j=i+1;j<num_user;j++){
        set_of_UV=0; // variable to check if only 1 item is common between 2 users
        sum_of_diff_ratings=0;
        Euclidean_sim_one_or_zero=1;
        for(int k=0;k<num_item;k++){
            if((data[i][k]!=0)&&(data[j][k]!=0)){
                set_of_UV++;
                sum_of_diff_ratings+=(data[i][k]-data[j][k])*(data[i][k]-data[j][k]);
            }
        }
        if(set_of_UV==0)
            Euclidean_sim_one_or_zero=0;
        sim_Euclidean[i][j]=Euclidean_sim_one_or_zero/sqrt(sum_of_diff_ratings);
        sim_Euclidean[j][i]=sim_Euclidean[i][j]; //(i,j)=(j,i)
    }
}
```

Listing 3.4: Calculation of Euclidean Similarity between users

Pearson

```
//calculate mean values for Pearson similarity
float sum_of_ratings_of_each_user, num_ratings_of_each_user, float mean_ratings[N];

for(int i=0;i<num_user;i++){
    sum_of_ratings_of_each_user=0;
    num_ratings_of_each_user=0;
    for(int j=0;j<num_item;j++){
        if(data[i][j]!=0){
            sum_of_ratings_of_each_user+=data[i][j];
            num_ratings_of_each_user++;
        }
    }
    mean_ratings[i]=sum_of_ratings_of_each_user/num_ratings_of_each_user;
}

float product_of_ratings_U_V, sum_of_squares_U, sum_of_squares_V, float sim_Pearson[N][N];
int set_of_UV;

for(int i=0;i<num_user;i++){
    for(int j=i+1;j<num_user;j++){
        set_of_UV=0; //variable to check if only 1 item is common between 2 users
        product_of_ratings_U_V=0;
        sum_of_squares_U=0;
        sum_of_squares_V=0;
        for(int k=0;k<num_item;k++){
            if((data[i][k]!=0)&&(data[j][k]!=0)){
                set_of_UV++;
                product_of_ratings_U_V+=(data[i][k]-mean_ratings[i])*(data[j][k]-
    mean_ratings[j]);
                sum_of_squares_U+=(data[i][k]-mean_ratings[i])*(data[i][k]-mean_ratings[i]);
                sum_of_squares_V+=(data[j][k]-mean_ratings[j])*(data[j][k]-mean_ratings[j]);
            }
        }
        if(set_of_UV==1)
            product_of_ratings_U_V=0; //if only one item is common between 2 users then
    their similarity is zero
        sim_Pearson[i][j]=product_of_ratings_U_V/sqrt(sum_of_squares_U*sum_of_squares_V);
        sim_Pearson[j][i]=sim_Pearson[i][j]; //(i,j)=(j,i)
    }
}
```

Listing 3.5: Pearson Similarity calculation between users

According to the algorithm, then having computed the similarity matrix w_{uv} containing the weights between users (or objects), the K most similar users of each user (or the K most similar objects of each object) are computed by the Knn algorithm [Guo+04].

Calculation function of Knn

```
float user_similarity;
int sim_k_users[N][k_max];
for(int i=0;i<num_user;i++){
    for(int k=0;k<k_max;k++){
        user_similarity=0;
        for(int j=0;j<num_user;j++){
            if(sim_user[i][j]>user_similarity){
                user_similarity=sim_user[i][j];
                sim_k_users[i][k]=j;
            }
        }
        sim_user[i][sim_k_users[i][k]]=0;
    }
}
```

Listing 3.6: Calculation of K most similar users

At this point, note that k_{max} is the maximum number of neighbors of the Knn algorithm and in this case it refers to the number of users/objects that are most similar to each user/object. This number was set during the design and the analysis of the temporal profile was performed for different values of k_{max}. Empirically

and from other studies, the root of the number of users or objects in the dataset was set as the final value, depending on the approach (USER-BASED/ITEM-BASED) [Zho+17].

Then the weight matrix of the K most similar users is an input for calculating the prediction of the recommendation system.

Forecast function of the system

```
float numerator_of_prediction_function, denominator_of_prediction_function;

for(int i=0;i<num_user;i++){
    for(int j=0;j<num_item;j++){
        numerator_of_prediction_function=0;
        denominator_of_prediction_function=0;
        if(data[i][j]==0){ //if the item has not been rated
            for(int k=0;k<k_max_users;k++){ // for all the similar k-neighbors the user has, check if they have rated the item
                if(data[sim_k_users[i][k]][j]!=0){ //if the similar neighbor has rated the movie that i-user wants to predict
                    numerator_of_prediction_function+=data[sim_k_users[i][k]][j]*sim_user[i][sim_k_users[i][k]]; //numerator
                    denominator_of_prediction_function+=sim_user[i][sim_k_users[i][k]]; //denominator
                }
            }
            data_filled[i][j]=numerator_of_prediction_function/denominator_of_prediction_function; //fill in the data table
        }
    }
}
```

Listing 3.7: Calculation of the forecast

The other functions that will be discussed later in the time profile are not directly related to the calculations but serve to design the system. The getDataFile function reads through a struct the data set which is of the form userID::movieID::rating::timestamp.

```
void getDataFile(FILE *file, char* delimiter, entryData* D){
  char    *line = NULL;
  size_t  len   = 0;
  ssize_t read;
  int     k     = 0;
  printf("Reading Dataset\n");
  while(1){
                read=getline(&line,&len,file);
                if(read==-1)break;
                else if(line[0]=='#'||line[0]=='\0'||!strcmp(line,"")||!strcmp(line," ")||line[0]=='\n'){
                        continue;
                }
    D[k].rowUser = atoi(strtok(line,delimiter));
    D[k].colItem = atoi(strtok(NULL,delimiter));
    D[k].rating  = atof(strtok(NULL,delimiter));
    k++;
  }
  free(line);
}
```

Listing 3.8: Reading the data set

The findNumberofEntries function calculates the number of scores contained in the dataset.

```
int findNumberofEntries(FILE* file){
  char    *line = NULL;
  size_t  len = 0;
  ssize_t read;
  int     k = 0;
  printf("Reading Dataset\n");
  while(1){
```

```
    read=getline(&line,&len,file);
    if(read==-1)break;
    else if(line[0]=='#'||line[0]=='\0'||!strcmp(line,"")||!strcmp(line," ")||line[0]=='\n')
    {
      continue;
    }
    k++;
  }
  printf("Dataset Entries are: %d\n",k);
  free(line);
  return k;
}
```

Listing 3.9: Calculating the number of evaluations in the dataset

3.3 Time Profile Illustration

For the implemented recommendation systems, for the different similarity computation functions w_{uv}, the time profile analysis was performed to make clear which part of the system needs optimization and is necessary to be accelerated. For different values of k_{max}, the most similar users/objects computation function, the execution times of the system were analyzed to make clear which part of the algorithm is the critical part. As part of the temporal analysis, the gprof tool was used from which information about the percentage of the total execution time consumed by each function was obtained.

% time	cumulative seconds	self seconds	% time	cumulative seconds	self seconds	functions
Movielens 100k			Movilens 1m			
89.54	7.13	7.13	97.34	591.94	591.94	Jaccard_Similarity
6.41	7.64	0.51	1.36	600.23	8.29	Prediction_algorithm
1.55	7.59	0.12	1.06	606.69	6.46	Knn_algorithm
89.10	7.05	7.05	97.22	589.81	589.81	Cosine_Similarity
6.83	7.59	0.54	1.45	598.62	8.81	Prediction_algorithm
2.02	7.75	0.16	1.07	605.14	6.52	Knn_algorithm
80.27	3.48	3.48	94.87	287.04	287.04	CosineIR_Similarity
10.84	3.95	0.47	2.35	294.16	7.12	Prediction_algorithm
3.69	4.11	0.16	2.14	300.62	6.47	Knn_algorithm
82.67	3.46	3.46	94.99	286.24	286.24	Euclidean_Similarity
10.75	3.91	0.45	2.19	292.84	6.60	Prediction_algorithm
3.82	4.07	0.16	2.15	299.32	6.48	Knn_algorithm
81.10	3.50	3.50	94.77	275.19	275.19	Pearson_Similarity
10.66	3.96	0.46	2.46	282.35	7.16	Prediction_algorithm
3.94	4.13	0.17	2.14	288.56	6.22	Knn_algorithm

Table 3.1: k_{max} 50

% time	cumulative seconds	self seconds	% time	cumulative seconds	self seconds	functions
Movielens 100k			Movilens 1m			
82.74	7.11	7.11	94.60	590.43	590.43	Jaccard_Similarity
12.00	8.14	1.03	3.10	609.80	19.37	Prediction_algorithm
3.73	8.46	0.32	2.04	622.56	12.76	Knn_algorithm
81.59	7.06	7.06	94.36	562.78	562.78	Cosine_Similarity
12.61	8.15	1.09	3.29	582.41	19.63	Prediction_algorithm
3.70	8.47	0.32	2.05	594.66	12.25	Knn_algorithm
71.07	3.50	3.50	90.04	289.67	289.67	CosineIR_Similarity
18.68	4.42	0.92	5.28	306.67	17.00	Prediction_algorithm
6.50	4.74	0.32	4.03	319.62	12.95	Knn_algorithm
71.06	3.45	3.45	90.39	275.11	275.11	Euclidean_Similarity
18.95	4.37	0.92	4.94	290.15	15.04	Prediction_algorithm
6.39	4.68	0.31	4.00	302.33	12.18	Knn_algorithm
70.55	3.51	3.51	89.82	275.37	275.37	Pearson_Similarity
19.30	4.47	0.96	5.54	292.37	17	Prediction_algorithm
6.43	4.79	0.32	3.99	304.59	12.22	Knn_algorithm

Table 3.2: k_{max} 100

% time	cumulative seconds	self seconds	% time	cumulative seconds	self seconds	functions
Movielens 100k			Movilens 1m			
77.99	7.59	7.59	92.68	633.91	633.91	Jaccard_Similarity
16.15	9.16	1.57	4.34	663.57	29.66	Prediction_algorithm
4.94	9.64	0.48	2.79	682.62	19.05	Knn_algorithm
76.13	7.06	7.06	91.96	568.09	568.09	Cosine_Similarity
17.60	8.69	1.63	4.83	597.90	29.81	Prediction_algorithm
5.08	9.16	0.47	2.95	616.14	18.24	Knn_algorithm
62.43	3.48	3.48	85.85	284.94	284.94	CosineIR_Similarity
25.83	4.93	1.44	7.86	311.03	26.10	Prediction_algorithm
8.43	5.40	0.47	5.73	330.05	19.02	Knn_algorithm
62.73	3.44	3.44	86.37	283.83	283.83	Euclidean_Similarity
24.98	4.82	1.37	7.26	307.69	23.86	Prediction_algorithm
8.75	5.30	0.48	5.76	326.63	18.94	Knn_algorithm
63.32	3.53	3.53	85.72	277.71	277.71	Pearson_Similarity
26.73	5.02	1.49	8.04	303.77	26.06	Prediction_algorithm
8.43	5.50	0.47	5.63	322.00	18.23	Knn_algorithm

Table 3.3: k_{max} 150

% time	cumulative seconds	self seconds	% time	cumulative seconds	self seconds	functions
Movielens 100k			Movilens 1m			
72.60	7.32	7.32	90.68	634.30	634.30	Jaccard_Similarity
19.66	9.30	1.98	5.52	672.88	38.58	Prediction_algorithm
6.06	9.91	0.61	3.61	698.15	25.27	Knn_algorithm
70.61	6.88	6.88	89.68	591.81	591.81	Cosine_Similarity
21.38	8.96	2.08	6.21	632.81	41.00	Prediction_algorithm
6.27	9.57	0.61	3.84	658.16	25.35	Knn_algorithm
56.46	3.39	3.39	82.51	285.86	285.86	CosineIR_Similarity
30.65	5.24	1.84	9.58	319.05	33.19	Prediction_algorithm
10.16	5.85	0.61	7.30	344.35	25.30	Knn_algorithm
57.45	3.38	3.38	82.95	284.63	284.63	Euclidean_Similarity
29.40	5.12	1.73	9.13	315.97	31.33	Prediction_algorithm
10.20	5.72	0.60	7.37	341.25	25.29	Knn_algorithm
54.54	3.46	3.46	82.51	280.30	280.30	Pearson_Similarity
32.00	5.50	2.03	9.76	313.44	33.14	Prediction_algorithm
9.62	6.11	0.61	7.13	337.67	24.23	Knn_algorithm

Table 3.4: k_{max} 200

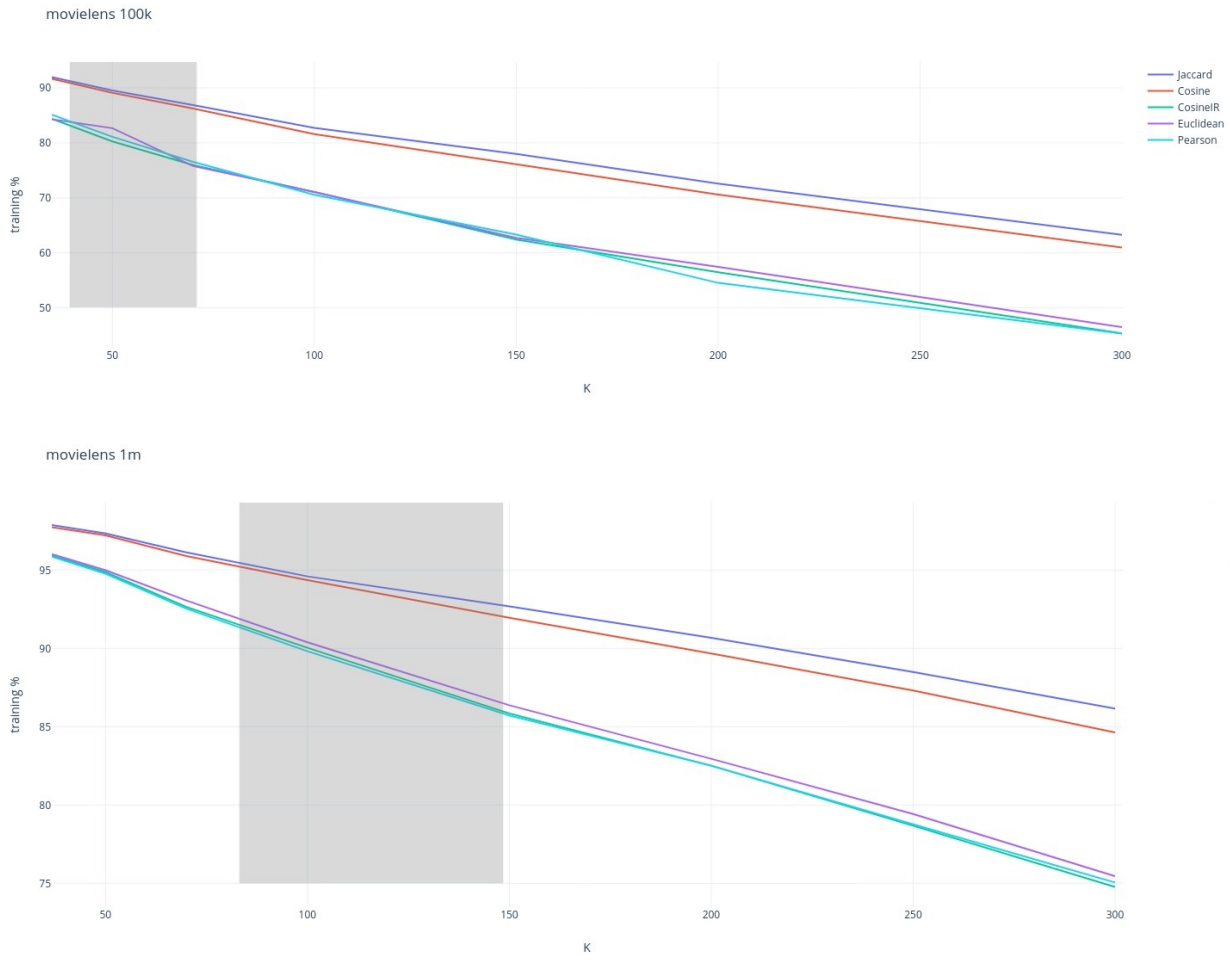


Figure 3.3.1: Performance of the system training algorithm as a function of K

Tables 3.1-3.4 show the percentage time consumed by the system algorithms for the various computation functions, where weights between users or objects are computed, for the datasets movielens 100k and movielens 1million, as a function of various values of K of the Knn algorithm. The three main functions that monopolize the system have been placed in the tables, and obviously there are several more which are not time-consuming and therefore not mentioned. As can be seen from the above tables, the percentages occupied by the training part of the system are overwhelmingly larger than the algorithms for predicting and computing the K most neighboring users or objects. Of course, as can be seen more clearly in the graph 3.3.1, the increase in K implies an increase in the execution time of the prediction algorithms, since as is logical the prediction function takes into account more similar users or similar objects to extract the final prediction, resulting in an increase in the percentage of the total execution time it consumes. The choice of the ideal K is a trade-off between accuracy and speed [Zho+17]. The gray boxed diagram 3.3.1 shows the range of the best possible values that can be given to K. For significant increases in K, system training is the overwhelming time consumption of the algorithm as a whole. The difference becomes even more apparent in the movielens 1m dataset, which contains a significantly larger number of users versus objects, as is usually the case in large platforms using recommender systems. From the above temporal analysis it becomes clear that in order to improve the recommendation system in terms of time and increase the speed of extracting the final predictions, it is necessary to speed up the training part since, as it turned out, this overloads the overall system response.

Chapter 4

Accelerating Algorithms

4.1 Introduction

To optimize the training of the system, the directives offered by Xilinx mentioned above were used. At each step of the implementation and design, certain stages and intermediate runtimes will be presented and commented on in order to make clear the rationale followed for the development of the final accelerator. The SDSoc tool generates together with the desired bitstream analytical reports for the accelerator placed on the FPGA about the resources it binds. At the same time, the trade-off of performance versus resource consumption in the FPGA is continuously taken into account so that through feedback the designed accelerator has maximum performance.

To begin with, developing and designing accelerators in C/C++ using HLS directives is a process that differs significantly from others that require the use of hardware languages such as VHDL or Verilog, but it involves certain challenges that are useful to take into account to a large extent. Primarily, FPGAs contain small and fast BRAMs, therefore the structure of most algorithms needs to be changed to avoid multiple calls and data transfers from PS to PL. In particular, in the specific case where the goal is to speed up system training and compute weights between users or objects, the reasoning path of the algorithm changes radically. In particular, in the USER-BASED representation the rows of the 2.1.1 represent the users while the columns represent the objects. To compute the similarity relation of one user to another, regardless of the mathematical function used, it is required that the accelerator checks and compares all the elements of two rows, i.e., all the objects of the two users. The implementation of the functions given in chapter 3 for a classical processor differs significantly from the design on an FPGA where, one has to take into account the limitation of the available BRAM memory and to design the data transfer as appropriately as possible which costs in performance. In an embedded system, the data is stored on the host and by multiple transfers come to the FPGA. Therefore, it becomes clear that the computation function no longer has the structure it used to have, and there is a need to properly separate the data in the 2.1.1 matrix to avoid unnecessary read/writes to and from the host. In addition, the calculation of now the weights of all users to each other becomes more complex and the amount of data coming to the PL from the PS must be appropriately sized, since the ideal size of the BRAM where the data will be stored must be defined during design, so as not to waste unnecessary FPGA resources that can be used elsewhere and have a greater impact on maximizing the performance of the system. Another challenge that one has to face when designing an accelerator is that the size of the tables is angular at compile time, so it is not possible to unwind the loops completely since the number of users or objects and in fact the number of loop iterations is unknown to the accelerator. Regarding the PS and PL interface which is responsible for the data transfer it is necessary to study the advantages and disadvantages offered by each data mover provided by Xilinx. For communication between the DMAs and the accelerator the STREAM protocol was used for most reads, while burst reads were performed to a lesser extent.

4.2 Acceleration of Critical Functions

First, the program running on the host and calling the accelerator is presented.

```
1  //ALLOCATE HARDWARE BUFFERS------------------------
2  float   *data           = (float*)sds_alloc(data_matrix_size_bytes);
3  float   *similarity     = (float*)sds_alloc(sim_matrix_size_bytes);
4
5  //HARDWARE FUNCTION-------------------------
6
7  #ifdef __SDSOC__
8      power_monitoring_start(logfilename_PS_AND_PL2,2);
9      total_accLearn_StartStamp = sds_clock_counter();
10 #endif
11
12 toplevelHW_learn(data, similarity, numUsers, numItems, 0, numUsers);
13
14 #ifdef __SDSOC__
15     power_monitoring_stop(2);
16     total_accLearn_FinishStamp = sds_clock_counter();
17     total_accLearn = total_accLearn_FinishStamp - total_accLearn_StartStamp;
18 #endif
```

Listing 4.1: Calling the accelerator from the host

The host calls the accelerator via the top-level hardware function. The accelerator is then responsible for reading the data and determining when the transfer from the host to it will take place.

```
void toplevelHW(float* data, float* similarity, int numUsers, int numItems, int indx, int
    lmtindx){

    stream_data_fifo _data;
#pragma HLS stream variable=_data depth=ITEM_SIZE

    int _user0[ITEM_SIZE], _sim0[USER_SIZE], _sum0, Item_tiles = numItems/ITEM_SIZE, jjj0,
    isfull0=0, flg = Item_tiles-1;

    for(int i=indx;i<lmtindx-1;i++){
        for(int k=0;k<Item_tiles;k++){
            jjj0=i+1;
            isfull0=0;
            pickUser(data, _user0, i, k, numItems);
            for(int j=i+1;j<numUsers;j++){
                if(isfull0==USER_SIZE){
                    simisfull(similarity, _sim0, i, jjj0, numUsers, k, flg);
                    isfull0=0;
                    jjj0+=USER_SIZE;
                }
                pickNextUser(data, _data, j, k, numItems);
                compute_sim(_user0, _data, _sum0);
                store(_sim0, _sum0, isfull0);
                isfull0++;
            }
            simisfull(similarity, _sim0, i, jjj0, numUsers, k, flg);
        }
    }
}
```

Listing 4.2: top-level accelerator function

Through the pickUser function, each user is selected to be compared with all others via burst read. All other users that are under comparison with the one selected via pickUser are transferred via pickNextUser, using streaming with fifo, to make the transfer from PS to PL fast. This design difference in the way the two functions transfer data has to do with the fact that pickNextUser is responsible for hundreds of times more data transfers than pickUser therefore it needs faster transfers than pickUser, while using streaming in both functions at different points of nested loops of course, was deprecated by the SDSoc tool.

```
void pickUser(float* data, int _user0[ITEM_SIZE], int i, int k,int numItems){
    pickUser: for(int h=0;h<ITEM_SIZE;h++){
#pragma HLS PIPELINE
        _user0[h] = data[i*numItems+k*ITEM_SIZE+h];
    }
}
```

Listing 4.3: Burst reads via the pickUser function

```
void pickUser(float* data, int _user0[ITEM_SIZE], int i, int k, int numItems){
    pickNextUser: for(int h=0;h<ITEM_SIZE;h++){
#pragma HLS PIPELINE
        _data << data[j*numItems+k*ITEM_SIZE+h];
    }
}
```

Listing 4.4: Streaming via the pickNextUser function

Then the compute_sim calculation function takes place which is responsible for calculating the weights between users or objects. The design was done in such a way that to produce the calculations with one of Jaccard, Cosine, CosineIR, Euclidean, Pearson it is sufficient to convert only the compute_sim calculation function. Then, once the compute_sim function calculates the similarity coefficient of the lines (users or objects) under comparison, the result is stored via the store function in the BRAM of the accelerator, while the simisfull function checks the capacity of the BRAM and fifo reserved to perform write backs if necessary to free space and refresh them with the new calculations.

4.3 Algorithm Optimizations

Then, to further optimize the performance of the accelerator as a design choice, it was chosen to compare more than one row of the 2.1.1 matrix with each other. In essence, since all users or objects have to be compared with all others, it is possible to increase those that are checked simultaneously in order to reduce as much as possible the data transfers to and from the FPGA and the total number of loop iterations in the toplevelHW function. Of course, when the number of matrix rows being compared simultaneously is increased, the resources consumed by the application on the FPGA are also increased since more data is transferred in each computation. As an example, checking four users/objects that are compared simultaneously with all others leads to the following changes in the structure:

```
void toplevelHW_learn(float* data, float* similarity, int numUsers, int numItems, int indx,
    int lmtindx){
#pragma HLS INLINE
#pragma HLS stream variable=_data depth=ITEM_SIZE
  stream_data_fifo _data;
    int _user0[ITEM_SIZE], _user1[ITEM_SIZE], _user2[ITEM_SIZE], _user3[ITEM_SIZE];
    int _sim0[USER_SIZE], _sim1[USER_SIZE], _sim2[USER_SIZE], _sim3[USER_SIZE];
    int _sum0=0, _sum1=0, _sum2=0, _sum3=0;
    int Item_tiles = numItems/ITEM_SIZE, jjj0, isfull0=0, flg = Item_tiles-1, ii=indx;

    for(int i=indx;i<lmtindx-4;i+=4){
        ii+=4;
        for(int k=0;k<Item_tiles;k++){
            jjj0=i+1;
            pickUser(data, _user0, _user1, _user2, _user3, i, k, numItems);
            isfull0=0;
            for(int j=i+1;j<numUsers;j++){
                if(isfull0==USER_SIZE){
                    simisfull(similarity, _sim0, i, jjj0, numUsers, k, flg);
                    simisfull(similarity, _sim1, i+1, jjj0, numUsers, k, flg);
                    simisfull(similarity, _sim2, i+2, jjj0, numUsers, k, flg);
                    simisfull(similarity, _sim3, i+3, jjj0, numUsers, k, flg);
                    isfull0=0;
                    jjj0+=USER_SIZE;
                }
                pickNextUser(data, _data, j, k, numItems);
                compute_sim(_user0, _user1, _user2, _user3, _data, _sum0, _sum1, _sum2,
    _sum3);
                store(_sim0, _sim1, _sim2, _sim3, _sum0, _sum1, _sum2, _sum3, isfull0);
                isfull0++;
            }
            simisfull(similarity, _sim0, i, jjj0, numUsers, k, flg);
            simisfull(similarity, _sim1, i+1, jjj0, numUsers, k, flg);
            simisfull(similarity, _sim2, i+2, jjj0, numUsers, k, flg);
            simisfull(similarity, _sim3, i+3, jjj0, numUsers, k, flg);
        }
    }

    for(int i=ii;i<lmtindx;i++){
        for(int k=0;k<Item_tiles;k++){
            jjj0=i+1;
            pick1User(data, _user0, i, k, numItems);
            isfull0=0;
            for(int j=i+1;j<numUsers;j++){
                if(isfull0==USER_SIZE){
                    simisfull(similarity, _sim0, i, jjj0, numUsers, k, flg);
                    isfull0=0;
                    jjj0+=USER_SIZE;
                }
                pickNextUser(data, _data, j, k, numItems);
                compute_Remaining_sim(_user0, _data, _sum0);
                store_Remaining_sim(_sim0, _sum0, isfull0);
                isfull0++;
            }
            simisfull(similarity, _sim0, i, jjj0, numUsers, k, flg);
        }
    }
```

```
}
```

Listing 4.5: Control four users simultaneously

As shown, pickUser now transfers four users at once, while simultaneously comparing them in compute_sim with the users transferred via fifo by the pickNextUser function. This avoids several unnecessary comparisons in compute_sim, reduces the number of data transfers of the 2.1.1 table from host memory to the FPGA, and reduces the total number of loop iterations. Of course, the above design led to an increase in the resources channeled to the FPGA, but performance increased significantly relative to the extra resources committed. Subsequently, the number of users controlled simultaneously was further increased to 4, 8, 10, 20, 40, 60. The analysis of the trade-off of resource consumption and time performance showed that comparing and computing weights of above users simultaneously has a positive effect and can be paid in resources for the performance it offers.

```
void pickUser(float* data, int _user0[ITEM_SIZE], int i, int k,int numItems){
        #pragma HLS INLINE
    pickUser: for(int h=0;h<ITEM_SIZE;h++){
#pragma HLS PIPELINE
        _user0[h] = data[i*numItems+k*ITEM_SIZE+h];
        _user1[h] = data[(i+1)*numItems+k*ITEM_SIZE+h];
        _user2[h] = data[(i+2)*numItems+k*ITEM_SIZE+h];
        _user3[h] = data[(i+3)*numItems+k*ITEM_SIZE+h];
    }
}
```

Listing 4.6: Data transfer for four users

The calculation function is also converted since all users are now compared to all users by quads.

```
void compute_sim(int _user0[ITEM_SIZE], int _user1[ITEM_SIZE], int _user2[ITEM_SIZE], int
    _user3[ITEM_SIZE], stream_data_fifo &_data, int &_sum0, int &_sum1, int &_sum2, int &
    _sum3, int N0, int N1, int N2, int N3, int Ni){
    int temp_user0, temp_user1, temp_user2, temp_user3, temp_data;

    compute_sim: for(int i=0;i<ITEM_SIZE;i++){
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=2
        temp_user0 = _user0[i];
        temp_user1 = _user1[i];
        temp_user2 = _user2[i];
        temp_user3 = _user3[i];
        _data >> temp_data;
        _sum0 += (temp_user0!=0) ? (temp_data!=0) ? 1 : 0 : 0;
        _sum1 += (temp_user1!=0) ? (temp_data!=0) ? 1 : 0 : 0;
        _sum2 += (temp_user2!=0) ? (temp_data!=0) ? 1 : 0 : 0;
        _sum3 += (temp_user3!=0) ? (temp_data!=0) ? 1 : 0 : 0;
  }
    _sum0 += (N0+Ni)*10000;
    _sum1 += (N1+Ni)*10000;
    _sum2 += (N2+Ni)*10000;
    _sum3 += (N3+Ni)*10000;
}
```

Listing 4.7: Function for calculating weights by blocks using the Jaccard algorithm

By analogy, in the reasoning process followed, the number of users transferred and compared with each other at the same time was increased in order to analyse the advantages offered by this optimisation and to what extent the system benefits from the increase in users in relation to resource consumption. Figures 4.3.2 show the speed up compared to the original accelerator in the USER-BASED approach, where now more users are compared transferred and computed at the same time.

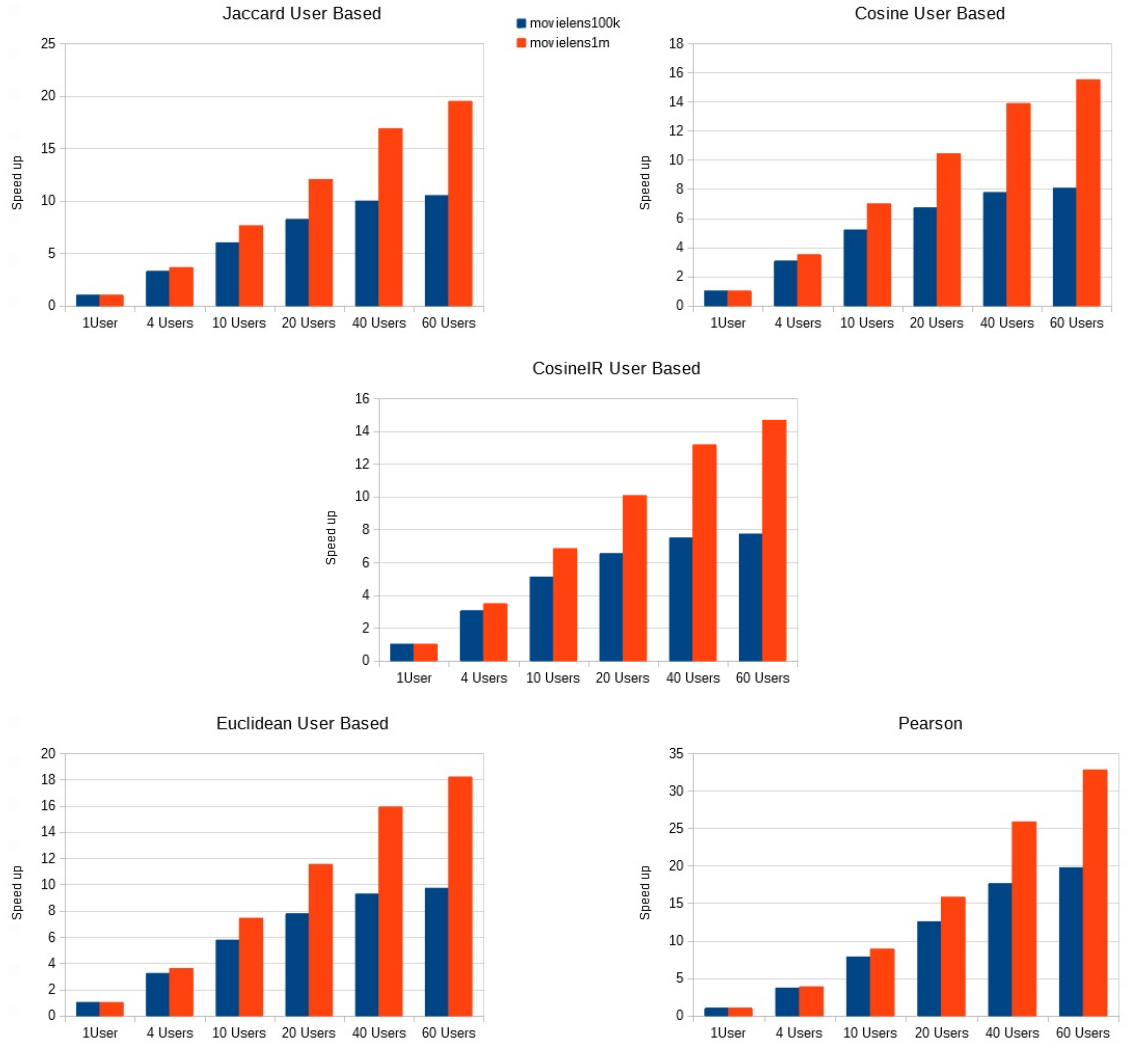


Figure 4.3.1: The acceleration compared to the original accelerator due to an increase in users

As observed above, the speed up offered by the simultaneous testing and computation of multiple users has a positive response on the system providing from 15 to 33 times reduction in the overall runtime in the USER-BASED approach. Also the difference becomes even clearer in movielens1m, where the speedup due to this optimization compared to the original accelerator is increased dramatically. Of course to get a global picture of what is happening it is necessary to analyze the cost in space and resources in the FPGA, so that the ideal user threshold becomes clear, which if compared simultaneously benefits the system.

Jaccard	BRAM	DSP	FF	LUTs
1 User	0%	1%	1%	3%
4 Users	1%	1%	1%	3%
10 Users	3%	1%	1%	4%
20 Users	5%	1%	1%	5%
40 Users	11%	2%	1%	7%
60 Users	16%	3%	3%	16%

Cosine	BRAM	DSP	FF	LUTs
1 User	0%	1%	2%	5%
4 Users	1%	0%	2%	5%
10 Users	3%	0%	2%	6%
20 Users	6%	0%	2%	6%
40 Users	11%	0%	3%	7%
60 Users	16%	0%	3%	12%

CosineIR	BRAM	DSP	FF	LUTs
1 User	1%	1%	2%	5%
4 Users	2%	2%	3%	6%
10 Users	5%	3%	3%	8%
20 Users	9%	6%	4%	11%
40 Users	18%	10%	5%	16%
60 Users	27%	15%	6%	23%

Euclidean	BRAM	DSP	FF	LUTs
1 User	0%	1%	3%	3%
4 Users	1%	1%	1%	4%
10 Users	3%	2%	1%	4%
20 Users	5%	3%	2%	6%
40 Users	11%	5%	2%	8%
60 Users	16%	15%	6%	18%

Pearson	BRAM	DSP	FF	LUTs
1 User	0%	1%	3%	5%
4 Users	0%	1%	3%	6%
10 Users	3%	2%	4%	8%
20 Users	7%	2%	5%	11%
40 Users	13%	4%	6%	18%
60 Users	27%	5%	8%	25%

Table 4.1: FPGA resources based on users

As can be seen from the speedup tables, comparing four users simultaneously instead of one-to-one quadruples the performance on movielens1m while causing at least 3 times shorter response time on movielens100k. Additionally, it is evident that as the number of users doubles, the response time decreases sharply, down to 40 users. Thereafter, there is a decrease in runtime of a smaller order than the initial one. At the same time it is shown that, the performance of the accelerator for 60 users is for all similarity algorithms greatly increased compared to the original accelerator design. From the resource consumption tables it can be seen that for comparison up to 20 users, the resource utilization is quite low, indicating that there is room for improvement. For 40 and 60 users the resource utilization becomes more noticeable especially in the LUTs part, where it seems to be the metric that increases significantly as users increase. Globally, the speed up provided by the 40 user control ranges from 13 to 26 times while the 60 user control ranges from 15 to 33 times. On the other hand, designing with 60 users significantly increases resources compared to designing with 40 users, without coming with the same order of speedup, which does not necessarily make it the most ideal solution to the problem.

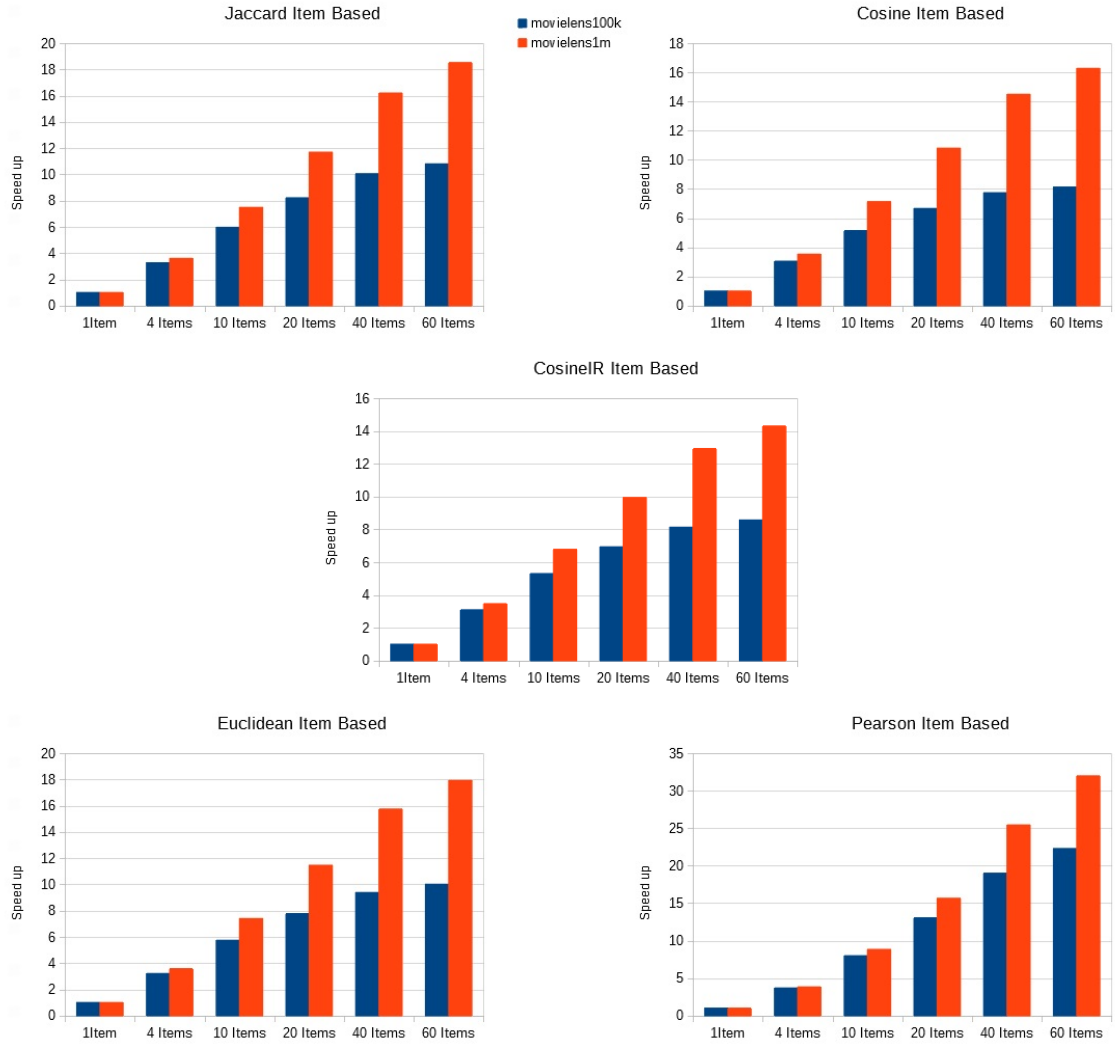


Figure 4.3.2: The acceleration relative to the original accelerator due to an increase in objects

Similarly, with the USER-BASED approach, in ITEM-BASED the number of items checked simultaneously is increased. Expectedly, the acceleration behavior with increasing number of items is similar to that of USER-BASED. Although the implementation times are completely different (as will be shown below) the speed up with increasing objects relative to the initial implementation shows similar increases to the USER-BASED approach. As before the advantages in speed up are shown by testing four objects alone. The gap between the designs for 40 and 60 objects is reduced compared to the USER-BASED version, and the difference is minimal in the movielens100k dataset. In the ITEM-BASED approach, it is more clearly shown that it is preferable to design a kernel with 40 objects instead of 60, since the increase in system resources is not accompanied by a similar increase in execution speed, so it is more appropriate to use the additional resources differently in order to contribute more to the overall performance. It is worth noting that the above designs were also tested on CPU versions of the system, but did not offer any improvement. In particular, optimal resolution on CPU is a design with 4 users or objects, while further increase of users or objects had a whole and larger negative impact on overall performance.

4.4 Multiple Computing Units

The analysis then focused on different ways of utilising the remaining available resources as further increases in users or objects did not contribute significantly to performance. Specifically, the design focused on creating

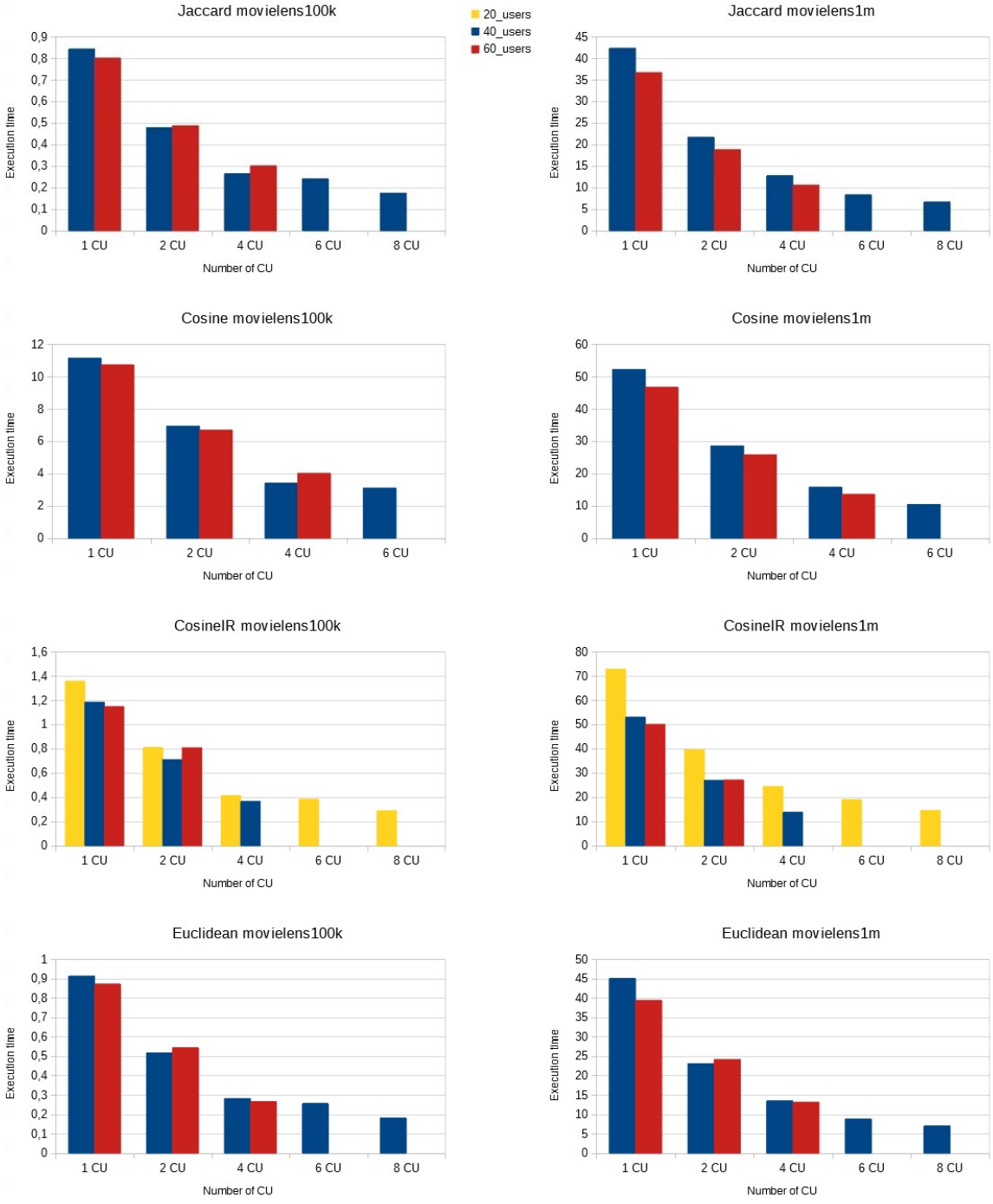
multiple computing units, copies of the original that work in parallel and independently of each other. The deployment of multiple Compute Units (CUs) takes place on the host and calls the designed accelerators in parallel and asynchronously.

```
//HARDWARE FUNCTION------------------------

#ifdef __SDSOC__
    power_monitoring_start(logfilename_PS_AND_PL2,2);
    total_accLearn_StartStamp = sds_clock_counter();
#endif

#pragma SDS resource(1)
#pragma SDS async(1);
    toplevelHW_learn(data, similarity, numUsers, numItems, 0, limit0);

#pragma SDS resource(2)
#pragma SDS async(2);
    toplevelHW_learn(data, similarity, numUsers, numItems, limit0, limit1);

#pragma SDS resource(3)
#pragma SDS async(3);
    toplevelHW_learn(data, similarity, numUsers, numItems, limit1, limit2);

#pragma SDS resource(4)
#pragma SDS async(4);
    toplevelHW_learn(data, similarity, numUsers, numItems, limit2, limit3);

#pragma SDS wait(1)
#pragma SDS wait(2)
#pragma SDS wait(3)
#pragma SDS wait(4)

#ifdef __SDSOC__
    total_accLearn_FinishStamp = sds_clock_counter();
    power_monitoring_stop(2);
    total_accLearn = total_accLearn_FinishStamp - total_accLearn_StartStamp;
#endif
```

Listing 4.8: Deploy four copies of the accelerator on the host

The design aims to create copies of the previously developed accelerator, which also control 20, 40 or 60 users (or objects) simultaneously as mentioned in the section 4.3, and perform the computations in parallel with each other. Through the directives (pragmas), pragma SDS resource(i), pragma SDS async(i) the SDSoc tool makes it possible to develop copies of an accelerator before calling it on the host, while with the directive pragma SDS wait(i), the synchronization and waiting of the hardware function can be manually controlled. Also, the hardware accelerators deployed are exact copies, therefore they occupy exactly the same resources on the FPGA, significantly increasing the space channeled. At the same time, now each copy of the accelerator works independently of the other and as shown is fed with different inputs. Consequently, now each accelerator has a range of users (or objects in the ITEM-BASED approach) for which it takes care of calculating their weight factor with all other users. Therefore, the copies are responsible for different boundaries of the matrix 2.1.1. The analysis continued by observing the resources and requirements of each application by checking the available resources in combination with optimizations via Xilinx directives. At the same time, the number of replicas that can be deployed on the FPGA is inextricably linked to the number of users being controlled simultaneously as discussed in the 4.3 section. A design that calculates the similarity factor for 20 rows of the 2.1.1 matrix results in a different number of compute modules that can be deployed on the FPGA than a design that generates the similarity factor for 40 or 60 rows at a time. The question therefore arises in the context of the analysis, of increasing the number of users controlled simultaneously as a function of the number of copies that can be placed according to the available resources of the FPGA. According to the table 4.1 the limitation in creating additional compute units is the LUTs. If the design exceeds the available LUTs of the FPGA then the SDSoC tool does not allow the design and informs about the overrun. For example from the table 4.1 o Pearson algorithm with 60 users/objects canals 25% of the available LUTs and up to 3 copies of this accelerator can be placed on the FPGA in a shuffle. On the other hand, the Pearson algorithm with 40 users/objects consumes 18% of the LUTs and therefore allows the design of at least four computing units, which may perform less individually, but together may outperform three

copies with 60 users/objects. Subsequently, therefore, it was examined which solution combination is the most appropriate, and performs best by creating multiple copies, depending of course on each algorithm, controlling for speed and resource consumption.



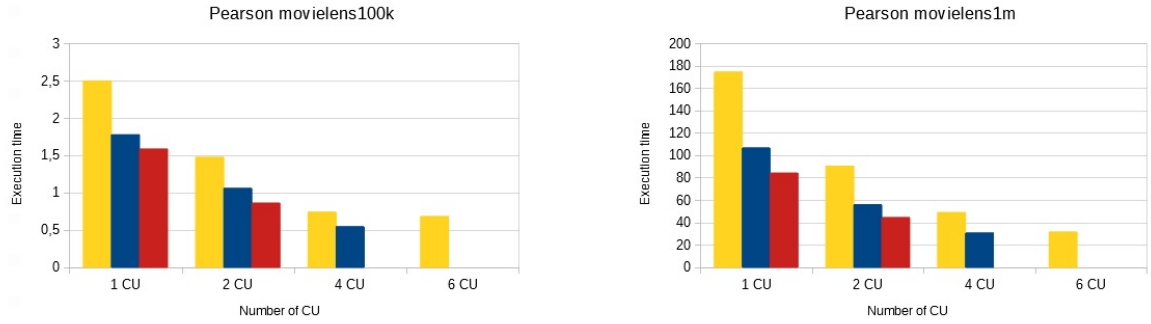
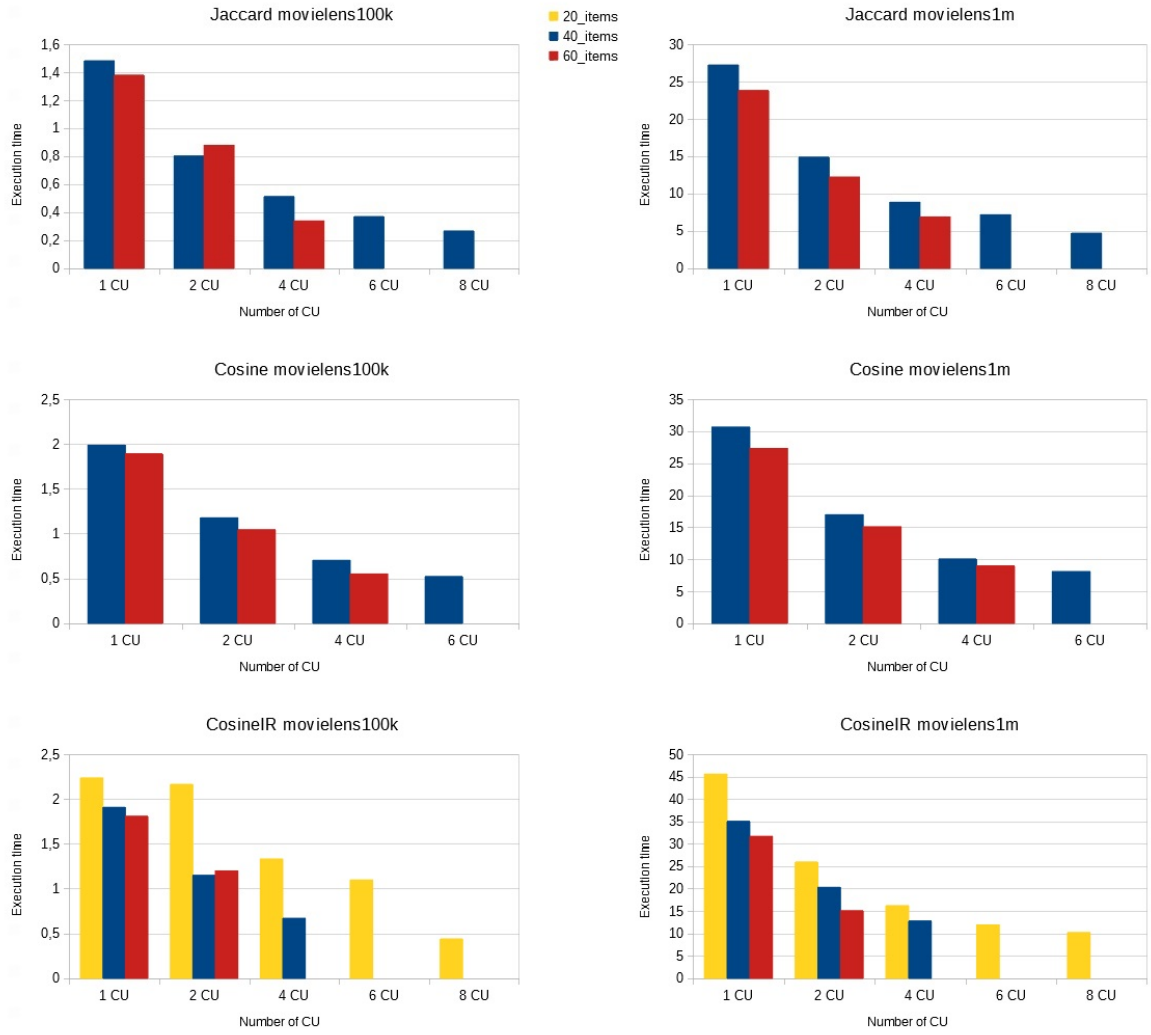


Figure 4.4.1: The improvement in runtime due to an increase in the number of computing units in the User Based approach

Similarly in the ITEM-BASED model the benefits of designing multiple CUs are shown below.



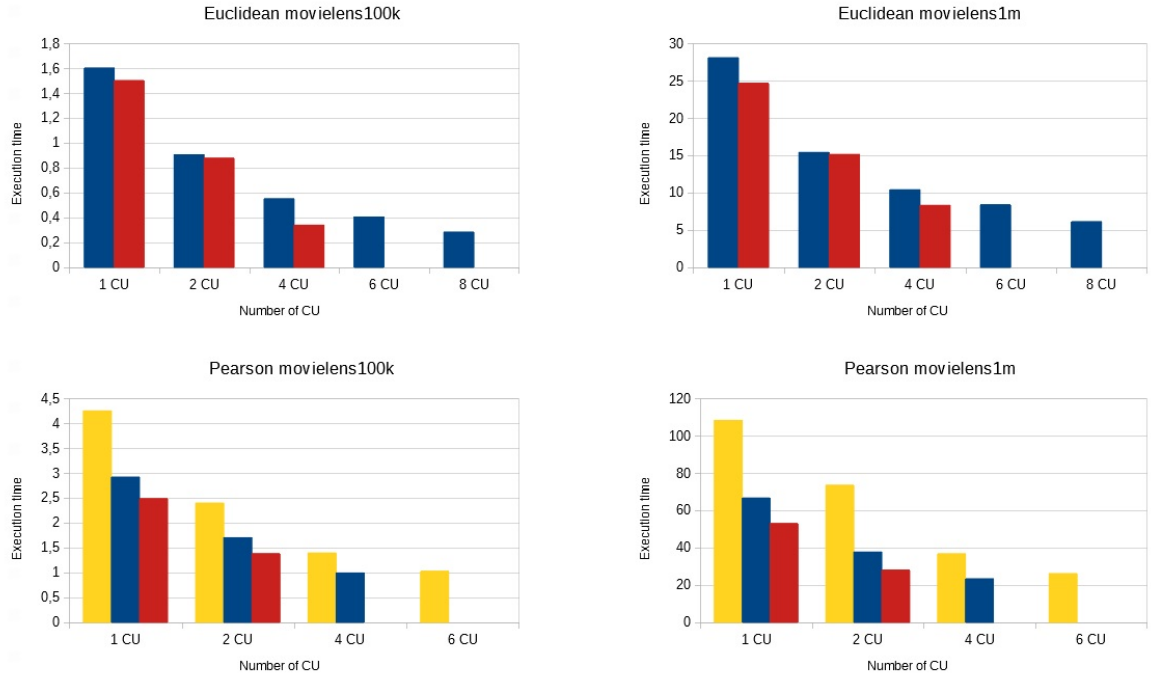


Figure 4.4.2: The improvement in runtime due to an increase in computing units in the Item Based approach

As is logical in more demanding algorithms such as cosineIR and Pearson, the SDSoC tool does not allow the creation of four computational units due to a limitation imposed by the LUTs. In implementations where this limitation is set for 60 users, compute units for 20 users were developed to better illustrate the acceleration over an accelerator offered by compute unit design. Comparing the graphs for both approaches it is easy to see that in many cases the speedup offered by multiple copies of the accelerator is of the same order for both movielens100k and movielens1m for the ITEM-BASED model. In contrast, in the USER-BASED approach for the most part the acceleration achieved in movielens1m is consistently higher than that in movielens100k. This is because in most datasets, there are many more users than objects (as is the case in real platforms), so when the ITEM-BASED model is followed, each compute unit receives a smaller input range and performs fewer iterations. Of course, this advantage is offset by the considerably more data transfers that need to be performed compared to the USER-BASED approach, since for each object users are transferred from the host, hence several more data reads are performed on the FPGA. Simultaneously, as expected doubling of the accelerator copies leads to a doubling of the acceleration over the original accelerator.

4.4.1 Load Balancing of the Computing Units

Load Balancing is a key technique for ensuring the desired performance and proper operation of demanding applications that work with multiple computing units and execute processes in parallel. The inputs to each copy of the accelerator are derived by studying the performance of the system as a whole. In general, the matrix containing the similarity coefficients 4.2 of users (or objects) is symmetric.

	U0	U1	U2	...	Un
U0	0	w(0,1)	w(0,2)	...	w(0,n)
U1	w(1,0)	0	w(1,2)	...	w(1,n)
U2	w(2,0)	w(2,1)	0	...	w(2,n)
.					
.	...	...	...	...	...
.					
Un	w(n,0)	w(n,1)	...	...	0

Table 4.2: Shape of the weight matrix

Therefore, if N is the number of users in a system, then the structure of the algorithm is such that at each iteration user i starts its comparison with i+1 and ends at user N, since the comparison of i with all previous users has already been performed at a previous stage due to the symmetry of the matrix 4.2. Therefore, if a system following the USER-BASED approach has N users, the algorithm running on one accelerator only, performs $N-1+N-2+N-3+...+1$ comparisons which is equal to $\frac{N*(N-1)}{2}$. As far as a hardware accelerator is concerned, this fact does not particularly affect the performance, but in the context of designing multiple copies of the accelerator, each containing different inputs, it is necessary to take into account this peculiarity of the algorithm, otherwise the system does not benefit from their creation. Consequently, each copy of the accelerator has a much longer response time than the next one designed. Therefore, to improve the execution speed of the systemit is particularly important that each replica accepts a significantly smaller input range than its immediate successor. After analysis it was found that, ideal input to each computing unit for each design option is shown in the table 4.3.

	1st CU	2nd CU	3rd CU	4th CU	5th CU	6th CU	7th CU	8th CU
2 CU	30%	70%						
4 CU	13%	17%	20%	50%				
6 CU	8%	8%	12%	12%	20%	40%		
8 CU	5%	5%	7%	7%	10%	12%	14%	40%

Table 4.3: User or object input rates for each CU depending on the design

Chapter 5

Evaluation of Performance and Resource Consumption

5.1 Final FPGA Resource Utilization and Operating Frequency

The final consumption of available FPGA resources for each algorithm.

			Resources per 1 CU			
	Configuration	Total CUs	BRAM	DSP	FF	LUTs
Jaccard	40 Users/Items	8	11%	2%	1%	7%
Cosine	40 Users/Items	6	11%	0%	3%	7%
CosineIR	20 Users/Items	8	9%	6%	4%	11%
Euclidean	40 Users/Items	8	11%	5%	2%	8%
Pearson	40 Users/Items	4	13%	4%	6%	18%

	BRAM	DSP	FF	LUTs
Jaccard	88%	16%	8%	56%
Cosine	66%	0%	18%	54%
CosineIR	72%	48%	32%	88%
Euclidean	88%	40%	16%	64%
Pearson	52%	16%	24%	72%

Table 5.1: Final FPGA resource consumption for each similarity calculation algorithm

The SoC (System on Chip) FPGA Zynq UltraScale + MPSoC ZCU102 [Xil19] was used to run the developed accelerators. On this particular board for the beginning, the code written in C++ was tested together with the accelerators in order to perform the proper operation test. A quad-core ARM processor is shown in the Processing Subsystem of the board cooperating with the PL section. The available memory is 4GB DDR4 64-bit SODIMM. Additionally, the operating frequency of the ZCU102 ranges between 75 and 600MHz. The SDSoC tool allows the operating frequency to be set at the compile stage with the -clkid n command. Depending on the design and available resources SDSoC may or may not allow the frequency to be set at any level. The tool according to design requirements defined during the development of the accelerators allowed the operating frequency to be set up to 200 MHz.

	BRAM	DSP	FF	LUTs
Total	1824	2520	548160	274080

Table 5.2: The total available FPGA resources

As can be seen from the above tables of available space consumption in the FPGA, the resource constraints come from either the available BRAM memory or the LUTs. As shown in the configuration diagrams 4.1, the increase in BRAM committed is accompanied by a significant improvement in system performance, because it greatly reduces the number of data transfers from PS to PL and vice versa that are necessary to be done due to data dependencies and reduces the number of comparisons, and therefore the number of iterations of the internal loops that are performed. Then, the second design constraint is given by the available LUTs of the system which increase significantly in more complex algorithms such as Pearson in which, while BRAM is at low levels (52%) the LUTs have reached a point where they do not allow the addition of an additional computational unit. In the relevant references of the SDSoC tool the Initiation Interval (II) is reported to be achieved in the inner loop where the compute function compute_sim 4.5 is 1, as defined during the design of the implementation, which means that new data can enter the function for computation every subsequent clock cycle.

5.2 Final performance evaluation compared to processor

To perform the final evaluation of the performance of the accelerator implemented on the FPGA compared to a processor, ready-made libraries were used which perform the same functions of Jaccard, Cosine, CosineIR, Euclidean and Pearson algorithms in an optimized way on a given matrix. We used the scipy library [Vir+20] which is a free open source Python library with wide use in scientific computing.

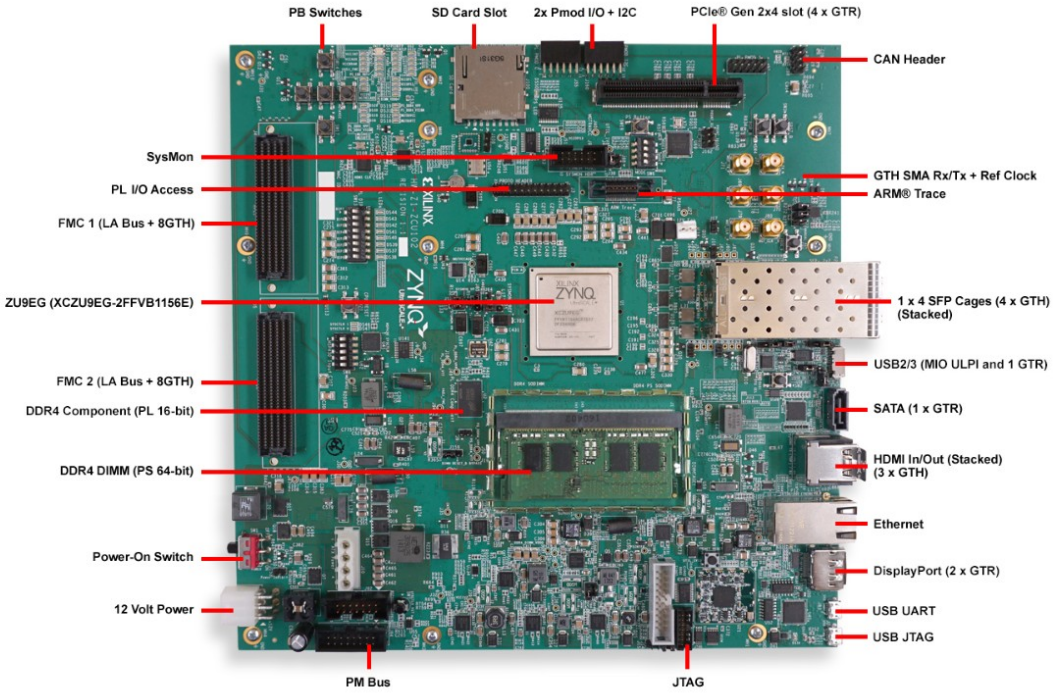


Figure 5.1.1: zcu102 Evaluation Board [Xil19]

The speedups achieved by hardware accelerators deployed on FPGA compared to scipy libraries running on CPU:

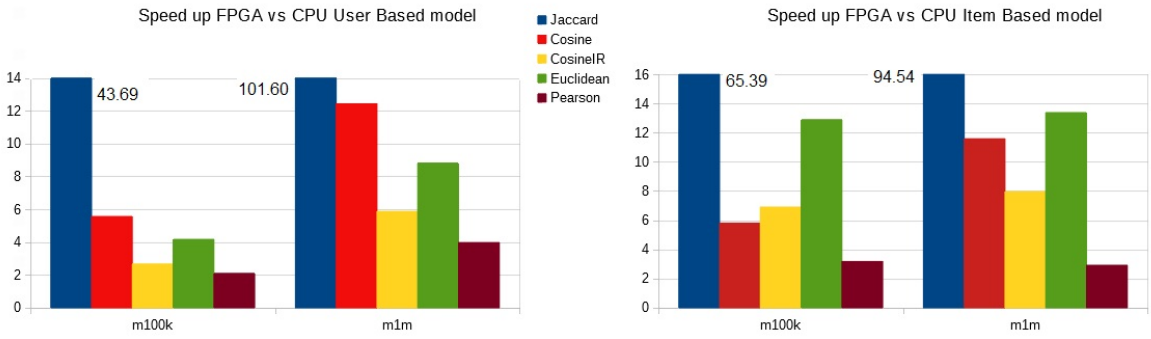


Figure 5.2.1: The speedups of Jaccard, Cosine, CosineIR, Euclidean and Pearson algorithms on FPGA versus CPU

For the comparison of CPU accelerators, a system with an Intel(R) Core(TM) i5-5200U 2.20GHz processor with 8GB DDR3 available memory and NVIDIA GeForce 920M GPU was used. As shown in the diagram, the hardware accelerator for the Jaccard algorithm significantly outperforms the scipy library. The differences are even more significant for movielens 1m where up to 100 times faster execution speed is achieved. Therefore, this shows that in FPGA the execution of operations with integer numbers has excellent performance. What confirms the above reasoning is Pearson's performance. The lowest speedup seems to be achieved in Pearson's algorithm, and it is the only one that has better response in the USER-BASED version and not in the ITEM-BASED version. This is due to the complexity of this particular algorithm, where it is required to calculate

the averages of users or objects. Therefore, between the two approaches the best performance is found in USER-BASED where the averages are computed for users which are significantly less than objects which leads to less averaging and therefore less workload for the accelerator. Additionally the Pearson algorithm is the only one where the multiplications are done with floats instead of integers which makes it for the FPGA quite a bit heavier in the computation part leading to a lower system response. Regarding the other algorithms it is obvious that the Item Based model manages to achieve higher speedup than the USER-BASED which can be very useful when choosing one of the two approximations.

5.3 Final performance evaluation compared to GPU

To evaluate the performance of the hardware accelerator developed with a GPU, the comparison was performed only for the Euclidean algorithm since no libraries were found for the other algorithms that pairwisely compute the covariance weights between rows or columns of a matrix and can be executed on a GPU. In particular, the PyTorch library [Pas+19] was used which is an open source machine learning library based on the Torch library and is used for applications such as computer vision and natural language processing and can be executed via CUDA on Nvidia graphics cards. For the comparison of GPU accelerators, a system with an Intel(R) Core(R) i7-8750H 2.20GHz processor with 16GB DDR4 available memory and NVIDIA GeForce GTX 1060 GPU was used. The results obtained in terms of performance are shown below.

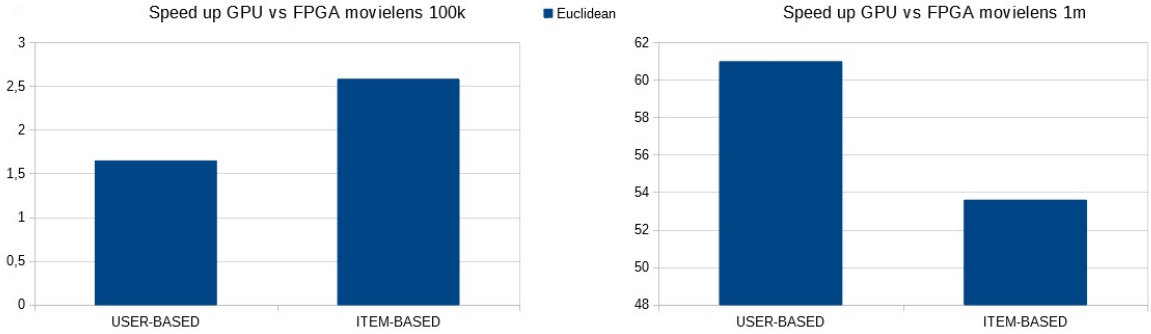


Figure 5.3.1: The acceleration of the GPU's Euclidean algorithm over the FPGA

As shown in 5.3.1 the GPU outperforms the FPGA. In particular, in the 100k movielens dataset we see that the differences are quite small since the GPU achieves about 1.6 and 2.6 times better time in the USER-BASED and ITEM-BASED models respectively. Of course the difference in performance becomes more obvious in movielens 1m where the GPU outperforms by far in the USER-BASED approach which is generally the most time-consuming case. In this case, it is also useful to show the energy consumed on the GPU to get a more complete picture on the matter.

5.4 Evaluation of FPGA power consumption compared to CPU and GPU

Next, the power consumption of the FPGA was compared to the CPU and GPU.

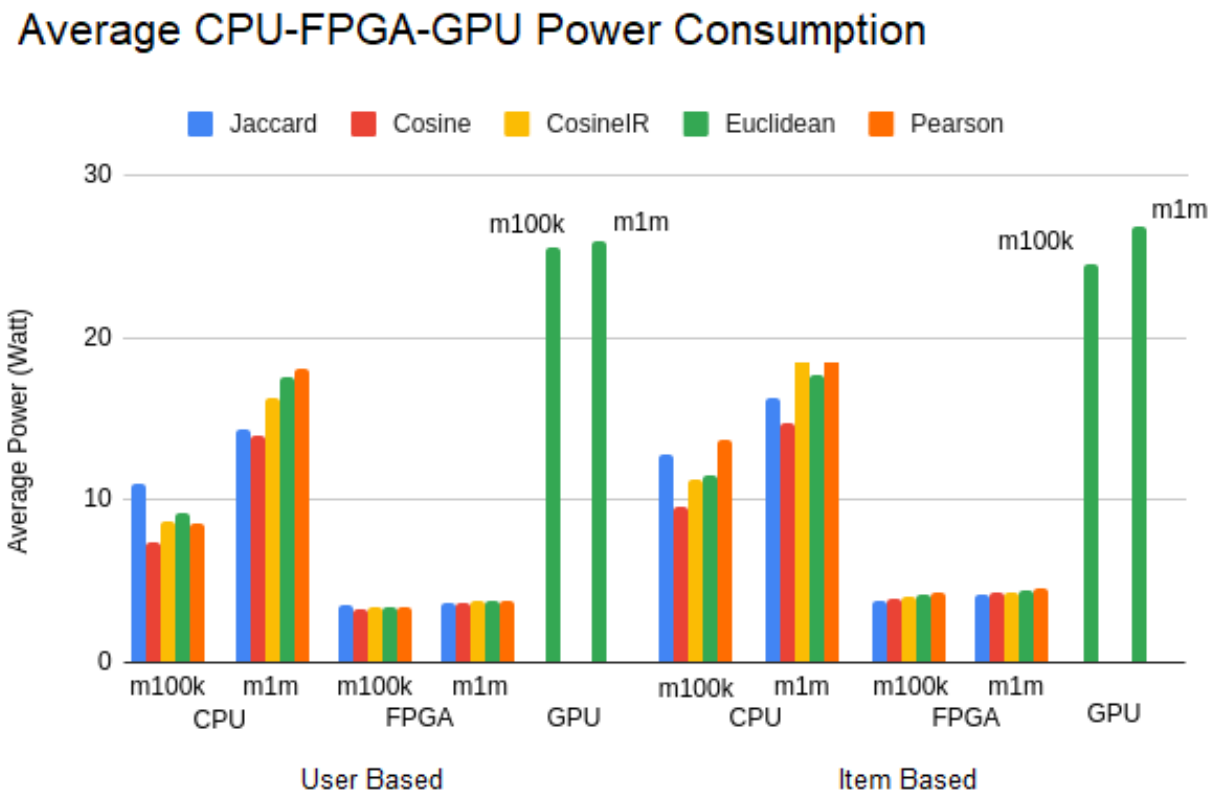


Figure 5.4.1: Average FPGA power consumption on the CPU and GPU

The measurement of power consumption in the SoC 5.1.1 is performed for all the board, both for the FPGA and the ARM processor, using tools provided by Xilinx [Don19]. Regarding the consumption on the GPU, tools provided by nvidia [NVI20] itself were used, and for measuring the power on CPU, the likwid [THW10] tool was used at the socket level. As expected, in terms of power consumption, the FPGA consumes significantly less power than the CPU for all the system training algorithms. FPGAs show quite an improvement in both power consumption and performance over CPU. Pearson's algorithm appears to be the least energy efficient on both FPGA and CPU in most of the instantiations, except for the USER-BASED approach of dataset movielens100k, where Jaccard has the highest consumption when running on CPU. At the same time, as shown the GPU had excellent performance in execution speed but with significantly higher power consumption compared to its FPGA counterpart. This shows that the FPGA is more efficient in metric performance per watt, in the movilenes100k dataset where the efficiencies were very close 5.3.1, as opposed to the consumption being 6.5 times higher on the GPU. On the other hand, the FPGA does not achieve such good results in the metric performance per watt, in the movilens1m dataset since the GPU has this time 7 times higher power consumption but with 61 and 53 times better performance in the USER-BASED and ITEM-BASED approach respectively 5.3.1. Overall, despite the worse execution times compared to the GPU the FPGA realizes satisfactory performance combined with a fairly low power consumption. The comparison with CPU is an easy dichotomy as the accelerators developed manage to outperform the CPU in both time and power efficiency.

Chapter 6

Conclusion

6.1 Summary and Conclusions

As shown by the overall analysis, the implementation of the algorithms on hardware is shown to be highly efficient and beneficial in terms of performance both in terms of performance and energy consumption compared to the corresponding libraries executed on CPU. In this thesis, when designing the accelerators for the algorithms chosen to be optimized, a key issue was the most efficient conversion of the algorithms in order to make the handling of the large amount of data that was asked to be transferred and processed on the FPGA as efficient as possible. In particular, to speed up the implementations developed, the major problem was not so much the complexity of the functions but the more efficient transfer of data from the host to the FPGA. In particular, for the improvement of the algorithms the most important role was played by the appropriate change of their structure with respect to the original software version, so that the movements that have to be made to the FPGA to perform the necessary computations have the least possible impact on the execution speed of the FPGA. The designed accelerators differ in terms of the computation function, following the same way of transferring data to and from the PL. For the transfers to the BRAM memory of the FPGA it was necessary to split the data and pass them serially as sets. Once the data processing and the computation functions in the hardware functions are performed then write backs are performed and each data set is replaced with the next one. The size of the data sets passed to the FPGA is a design choice, according to the resources available on the zcu102. As it emerged during the design, the implementations are directly dependent on the available resources of the FPGA. The accelerated algorithms have several memory-bound dependencies, therefore in order to obtain improved performance on the FPGA, it is necessary to give special attention to the structure of the hardware code, in order to make as few unnecessary transfers as possible, without increasing the complexity and altering the accuracy of the results. In particular, constraints are imposed by the available BRAM and board LUTs. Deploying applications on a larger and improved FPGA, with increased BRAM would lead to further performance gains by passing a larger volume of data and reducing the reads and write-backs performed due to data dependencies.

In addition, if a system follows the collaborative filtering method, it is required to choose a mathematical algorithm that will calculate the weights of the user or object matrix, depending on the method used. The acceleration of the algorithms and the evaluation of the metrics, showed that the ITEM-BASED approach has a larger performance and better acceleration in all algorithms except Pearson. As is usually the case in a platform or service, the datasets used have more objects than users. In both approaches, the size of the BRAM and the streaming fifo bound to the FPGA resources are the same. In the USER-BASED approach the Compute Units receive users as input, and the datasets transferred when the FPGA's BRAM is full are about objects that users rate. Similarly, in the ITEM-BASED model, Compute Units receive objects as input, and the data sets transferred when the BRAM is full are for users who have rated that object. Therefore, according to the percentages in the table 4.3, each Compute Unit in the ITEM-BASED version receives more input and is asked to perform comparisons for more objects than the USER-BASED version for users. This leads, to a larger overhead of each Compute Unit but to considerably fewer unnecessary reads and write-backs that significantly burden system performance, since BRAM and fifo are reused less often in

this case.

Therefore, a platform that will be used for collaborative filtering is useful to take this observation into account in order to achieve maximum efficiency in training the system when extracting weights.

In this thesis, a first attempt was made to implement the training of cooperative filtering algorithms, making use of HLS tools, in order to optimize them. In the first phase, the algorithms of training and prediction of the recommendation system following collaborative filtering were studied and designed. Subsequently, recommendation systems in C++ whose training part follows one of Jaccard, Cosine, CosineIR, Euclidean and Pearson for both USER and ITEM BASED approaches were designed.

A temporal evaluation of the developed systems was then carried out, which showed that the most time-consuming part is the training part, during which the weights of similarities between users or objects are calculated. Once, it was decided which part of the system needed acceleration, the HLS instructions provided by Xilinx and the techniques used in developing accelerators were explored and studied. This was followed by developing the accelerators, optimizing them using various techniques, introducing multiple compute units, and comparing performance and energy consumption with the off-the-shelf scipy and pytorch libraries with CPU and GPU respectively. The evaluation of the results showed that the implemented acceleration kernels outperform the CPU in terms of energy and time. At the same time, the comparison with GPU showed the superiority of GPU in terms of execution speeds, since FPGA achieved worse times than this architecture, but outperformed it in terms of energy consumption.

In conclusion, FPGA is an extremely attractive solution for accelerating similar algorithms, making it a highly advantageous choice.

6.2 Future Extensions

Many options for future extensions are generated by this thesis. A first idea is to use improved FPGAs with sufficient available resource sizes to enable the creation of a hardware accelerator capable of optimizing both the prediction part along with the training part of the system, by deploying one or two cores on the board. At the same time, as a future extension of this study, it could be considered to integrate the FPGA into a stand-alone powered system, which could take as input the users and objects of a service and dynamically extract the results, as it would continuously accept new inputs from registered evaluations and re-train the system to extract new predictions leading to recommendations of greater accuracy. Also, building libraries of Jaccard, Cosine, CosineIR, Euclidean and Pearson algorithms on FPGAs is a future work. Finally, the designs were carried out with float variables which consume considerably less resources in the design than double variables have a better response on the FPGA [Ngu+] but offer less accurate results. As a future extension, it would also be possible to design the above algorithms with double numbers on an improved larger FPGA, which would provide more resources available in the design. In addition to recommender systems, the field of machine learning contains a large number of problems which, due to the amount of data handled, are not optimally executed and could be accelerated on FPGAs.

Chapter 7

Bibliography

[AR04] Ahmed, E. and Rose, J. "The effect of LUT and cluster size on deep-submicron FPGA performance and density". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.3 (2004), pp. 288–298. DOI: 10.1109/TVLSI.2004.824300.

[AMY09] Asano, S., Maruyama, T., and Yamaguchi, Y. "Performance comparison of FPGA, GPU and CPU in image processing". In: *2009 International Conference on Field Programmable Logic and Applications*. 2009, pp. 126–131. DOI: 10.1109/FPL.2009.5272532.

[Al-+17] Al-bashiri, H. et al. "Collaborative Filtering Recommender System: Overview and Challenges". In: *Advanced Science Letters* 23 (Sept. 2017), pp. 9045–9049. DOI: 10.1166/asl.2017.10020.

[Cho13] Chow, P. "Why Put FPGAs in your CPU socket?" In: *2013 International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 3–3. DOI: 10.1109/FPT.2013.6718320.

[Don19] Don Matson, L. B. *Accurate Design Power Measurement Made Easier*. Oct. 2019.

[GK19] Gandhare, S. and Karthikeyan, B. "Survey on FPGA Architecture and Recent Applications". In: *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*. 2019, pp. 1–4. DOI: 10.1109/ViTECoN.2019.8899550.

[Gee+18] Geetha, G. et al. "A Hybrid Approach using Collaborative filtering and Content based Filtering for Recommender System". In: *Journal of Physics: Conference Series* 1000 (Apr. 2018), p. 012101. DOI: 10.1088/1742-6596/1000/1/012101. URL:

[Guo+04] Guo, G. et al. "KNN Model-Based Approach in Classification". In: (Aug. 2004).

[HK15] Harper, F. M. and Konstan, J. A. "The MovieLens Datasets: History and Context". In: *ACM Trans. Interact. Intell. Syst.* 5.4 (Dec. 2015). ISSN: 2160-6455. DOI: 10.1145/2827872. URL:

[Iva02] Ivanov, L. "Modeling and verification of a pipelined CPU". In: Sept. 2002, pp. III–417. ISBN: 0-7803-7523-8. DOI: 10.1109/MWSCAS.2002.1187062.

[KR07] Kuon, I. and Rose, J. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 203–215. DOI: 10.1109/TCAD.2006.884574.

[Mus+17] Mustafa, N. et al. "Collaborative filtering: Techniques and applications". In: *2017 International Conference on Communication, Control, Computing and Electronics Engineering (ICCCCEE)*. 2017, pp. 1–6. DOI: 10.1109/ICCCCEE.2017.7867668.

[Ngu+] Nguyen, T. et al. "FPGA-based HPC accelerators: An evaluation on performance and energy efficiency". In: *Concurrency and Computation: Practice and Experience* n/a.n/a (), e6570. DOI: https://doi.org/10.1002/cpe.6570. eprint: URL:

[NVI20] NVIDIA. *NVML Reference Manual*. July 2020. URL:

[Pas+19] Paszke, A. et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL:

[RTT99] Riesgo, T., Torroja, Y., and Torre, E. de la. "Design methodologies based on hardware description languages". In: *IEEE Transactions on Industrial Electronics* 46.1 (1999), pp. 3–12. DOI: 10.1109/41.744370.

[Sah20] Sah, S. "Machine Learning: A Review of Learning Types". In: (July 2020). DOI: 10.20944/preprints202007.0230.v1.

[SDS15] SDSoC, X. *SDSoC Environment User Guide*. July 2015.

[Sim18] Simeone, O. "A Very Brief Introduction to Machine Learning With Applications to Communication Systems". In: *IEEE Transactions on Cognitive Communications and Networking* 4.4 (2018), pp. 648–664. DOI: 10.1109/TCCN.2018.2881442.

[THW10] Treibig, J., Hager, G., and Wellein, G. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments". In: *2010 39th International Conference on Parallel Processing Workshops*. 2010, pp. 207–216. DOI: 10.1109/ICPPW.2010.38.

[Tri15] Trimberger, S. M. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology". In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331. DOI: 10.1109/JPROC.2015.2392104.

[Vir+20] Virtanen, P. et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[Wan+20] Wang, C. et al. "WooKong: A Ubiquitous Accelerator for Recommendation Algorithms With Custom Instruction Sets on FPGA". In: *IEEE Transactions on Computers* 69.7 (2020), pp. 1071–1082. DOI: 10.1109/TC.2020.2988209.

[Xil19] Xilinx. *ZCU102 Evaluation Board User Guide UG1182*. June 2019.

[Xil21] Xilinx. *Vivado Design Suite User Guide UG902*. May 2021.

[Zho+17] Zhongguo, Y. et al. "A case based method to predict optimal k value for k-NN algorithm". In: *Journal of Intelligent Fuzzy Systems* 33 (Mar. 2017), pp. 1–10. DOI: 10.3233/JIFS-161062.