sciendo

JAISCR, 2021, Vol. 11, No. 2, pp. 99 – 110
10.2478/jaiscr-2021-0007

HARDWARE ROUGH SET PROCESSOR PARALLEL ARCHITECTURE IN FPGA FOR FINDING CORE IN BIG DATASETS

Maciej Kopczyński, Tomasz Grześ

Faculty of Computer Science, Bialystok University of Technology,

E-mail: t.grzes@pb.edu.pl

Submitted: 5th May 2020; Accepted: 5th November 2020

Abstract

This paper presents FPGA and softcore CPU based solution for large datasets parallel core calculation using rough set methods. Architectures shown in this paper have been tested on two real datasets running presented solutions inside FPGA unit. Tested datasets had 1 000 to 10 000 000 objects. The same operations were performed in software implementation. Obtained results show the big acceleration in computation time using hardware supporting core generation in comparison to pure software implementation.
Keywords: rough sets, FPGA, core attributes, parallel architecture

1 Introduction

The biggest problem associated with creating optimal decision support systems is the time needed to process input data, with particular emphasis on large data sets, the so-called big data. Big data is a term referring to large, diverse and variable data sets, whose processing and analysis is difficult but at the same time valuable, because it can lead to the acquisition of new knowledge. In practical applications, the concept of a large data set is relative and means a situation when the set cannot be processed using simple and commonly available methods [17]. Depending on the purpose of the chosen method and the complexity of the algorithm, this may mean a calculated size in megabytes, gigabytes or terabytes.

One of the examples of such huge datasets are data read from sensors – like from production line. Every part of such line has tens of sensors providing data all the time, which creates huge stream of readings. Amount of obtained data creates huge datasets that has to be processed for e.g. estimating future failures and thus avoiding downtime.

There were many methods created for processing such datasets and focusing on knowledge extraction and minimizing amount of data that has to be processed. One of them are rough sets theory allowing to easily limit number of attributes (columns) in input datasets to these, that are provide the valuable information. In these methods two definitions exist – reduct and core. Reducts are sets of columns that provide useful information considering removing redundant information, while cores are sets of attributes that cannot be removed from initial set under any circumstances. In other words, the attributes existing in the core must also exist in all the reducts.

The rough sets theory developed in the eighties of the twentieth century by Prof. Z. Pawlak is an useful tool for data analysis. Therefore a lot of rough sets algorithms were implemented in scientific and commercial tools for data processing. But data processing efficiency problem is arising with

the increase of the amount of data. Huge datasets are commonly referred to as Big Data.

Some of the solutions developed and oriented for data-type software implementations and algorithms with smaller computational complexity are shown below. An example of such an approach is research done in [1]. The authors presented an algorithm for feature extraction for a specific type of data with linear complexity. An additional software approach to dealing with big datasets involves parallel processing. One of the methods is related to the implementation of algorithms using MapReduce. The parallel technique for computing rough set approximations is presented in [28]. Algorithms for core and reduct calculation based on the distributed programming model of MapReduce can be found in [3]. Other types of approaches to fast processing data sets were shown in [4], where Binarized Neural Networks (BNN) were presented and are based on the optimization of computational complexity and memory demand. In [6], a framework for integrating hardware and software optimizations for sparse CNNs is shown. It is worth adding, that [7] presented an AlphaGo Policy Network using DCNN accelerator on FPGA.

Another solution for processing big datasets are hardware implementations of existing algorithms. This type of approach can be accomplished using FPGA units. Field Programmable Gate Arrays (FPGAs) are group of digital integrated circuits. Functionality of these units can be programmed by engineer at any time. This feature gives the possibility of evaluating any boolean function. That's why they can be used for supporting rough sets calculations, especially if algorithms of polynomial complexity and ability of processing parts of data in parallel are considered.

At the moment there are some hardware implementations of specific rough set methods. The idea of sample processor generating decision rules from decision tables was described in [21]. In [14] the authors presented the architecture of rough set processor based on cellular networks described in [16]. In [8] a concept of hardware device capable of minimizing the large logic functions created on the basis of discernibility matrix was developed. A more detailed summary of the existing ideas and hardware implementations of rough set methods can be found in [9] and in [27]. It should be noted, that [19] introduced efficient heuristics for computing from data table basic tools (like lower and upper approximations, positive regions, reducts) for rough set methods. Their heuristics are feasible for large data tables but are mainly focused on software solutions, while our proposition is focused on the hardware approach. In [18] implementation of the LEM2 algorithm on FPGA is presented.

Previous authors' research results focused on the subject of hardware implementations of rough sets methods can be found in previously published papers: the idea of the processor for rough sets methods [25], hardware-supported reduct calculation [10], the first approach to core generation using FPGA based solution [11], hardware unit for processing datasets consisting of millions of objects [12], a two-stage algorithm for finding reduct [5] as well as finding minimal reduct with two FPGA devices [2].

This paper is a new proposition for parallel hardware supported core calculation for big datasets and introduces modification of methods described in [11] and [13]. According to [11], the main focus was put on small datasets (up to 107 objects) and all operations were implemented as combinational logic, which gave huge acceleration in calculation speed, but was very limited to the amount of data, that could be processed. In [13], the target was to perform a sequential operation on core calculation based on the idea presented in [11], but this approach was limited to a single core.

The paper is organized as follows. In Section 2 some information about the notion of core and datasets used during research is provided. It includes also the pseudocode of the algorithms. Section 3 focuses on description of hardware solutions, while Section 4 is devoted to the experimental results.

2 Introductory Information

2.1 The Notion of Core in the Rough Set Theory

Let $DT = (U, A \cup \{d\})$ be a decision table, where U is a set of objects, A is a set of condition attributes and d is a decision attribute. In decision table some of the condition attributes from A may be superfluous (redundant in other words).

This means that their removal cannot worsen the classification. The set $C \subseteq A$ of all indispensable condition attributes is called the core. None of its elements can be removed without affecting the classification power of all condition attributes. In order to compute the core we can use discernibility matrix $[DM(x,y)]_{x,y \in U}$, where $DM(x,y) = \{a \in A : a(x) \neq a(y) \text{ and } d(x) \neq d(y)\}$.

The core is the set of all single element entries of the discernibility matrix, i.e. $CORE = \bigcup_{x,y \in U, cardinality(DM(x,y))=1} DM(x,y)$. A much more detailed description of the concept of the core can be found, for example, in the article [22] or in the book [24].

Algorithm presented in following subsections are based on available knowledge, but were substantially redesigned to provide possibility of their implementation in hardware units. Biggest effort was put on parallelism and identification of most computation-time consuming parts.

2.2 Principle of Core Calculation

One of the most common algorithm for core calculation using discernibility matrix is presented as pseudocode in this subsection. The biggest limitation of this algorithm using direct implementation, mainly for big datasets, is need for storing discernibility matrix *DM* as two dimensional array with size of $|U|x|U|$. Hardware version of core calculation algorithm called CORE-PHIDM (described in Section 2.3) uses principles of this approach to core creation.

Algorithm 1 Definition-based Core Calculation Algorithm

```
INPUT: discernibility matrix DM
OUTPUT: core C
1: C ← ∅
2: for x ∈ U do
3:    for y ∈ U do
4:       if |DM(x,y)| = 1 then
5:          C ← C ∪ DM(x,y)
6:       end if
7:    end for
8: end for
```

Input to the definition-based algorithm is discernibility matrix *DM* and output is core *C*. Core is initialized as empty set in line 1. Two loops in lines 2 and 3 iterate over all objects (denoted as U) in discernibility matrix. Condition instruction in line 4 checks if matrix cell contains only one attribute. If so, then this attribute is added to the core *C*.

2.3 Algorithm CORE-PHIDM for Hardware Supported Core Calculation With No Discernibility Matrix

The main concept of CORE-PHIDM (**CORE** **P**arallel **H**ardware **I**ndirect **D**iscernibility **M**atrix) algorithm is based on a property of singleton, which is cell from discernibility matrix consisting of the only one attribute. This property tells that any singleton cannot be removed without affecting the classification power. As it was mentioned, biggest problem with direct use of discernibility matrix based algorithms when dealing with big datasets is amount of memory needed to store the matrix. Such solution cannot be run in hardware because of FPGA resources limitations.

This is the reason why we proposed the hardware implementable algorithm – CORE-PHIDM. Main idea is to divide the entire dataset into parts stored in multiple independent memory units providing information for hardware modules. These parts are subsequently processed by the units. For clarity of presentation, pseudocode of the algorithm shows the most simple configuration of CORE-PHIDM block: one instance of *subCORE* module with two memory blocks denoted as RAM_{cmn} (common RAM connected to all modules) and RAM_1 (first RAM_n for single and only *subCORE* module). Details related to hardware implementation and naming is described in Section 3. Pseudocode for the algorithm is given below:

Algorithm 2 CORE-PHIDM Algorithm

INPUT: decision table $DT = (U, A \cup \{d\})$, two natural number $n, m > 0$
OUTPUT: core $C \subseteq A$

```
 1: C ← ∅
 2: for cnt1 ← 0 to m − 1 do
 3:    RAM_cmn ← {x ∈ U : x_{cnt1·n} to x_{(cnt1+1)·n−1}}
 4:    for cnt2 ← cnt1 to m − 1 do
 5:       RAM1 ← {x ∈ U : x_{cnt2·n} to x_{(cnt2+1)·n−1}}
 6:       for x ∈ RAM_cmn do
 7:          for y ∈ RAM1 do
 8:             if d(x) ≠ d(y) then
 9:                count ← 0
10:                for a ∈ A do
11:                   if a(x) ≠ a(y) then
12:                      count ← count + 1
13:                      candidate ← a
14:                   end if
15:                end for
16:                if count = 1 and candidate ∉ C
                   then
17:                   C ← C ∪ {candidate}
18:                end if
19:             end if
20:          end for
21:       end for
22:    end for
23: end for
```

Input to the algorithm CORE-PHIDM is decision table *DT*, and output is core *C*. In the first step core *C* is initialized as empty set. Two loops in lines 2 and 4 are responsible for choosing parts of input decision table. Decision table *DT* is divided into *m* parts, where each of them have the size of *n* objects. Lines 3 and 5 are responsible for loading chosen parts of dataset into RAM memories implemented in FPGA. Two loops in lines 6 and 7 take subsequent objects from decision table parts for comparison. Line 8 performs the comparison between decision attribute's value of two objects *x* and *y*. If these two objects belong to different decision classes, the rest of the algorithm is processed. *count* variable, responsible for storing the number of differences on condition attributes values between objects *x* and *y* is set to 0 in line 9. Loop in line 10 iterates over set of condition attributes *A*. Values of *a* condition attribute is compared between objects *x* and *y* in line 11. In case of difference, the *count* variable is incremented and *a* attribute is stored in *candidate* variable. When the attribute loop finishes, attribute in *candidate* variable is added to the core if *count* variable is equal to 1 and this attribute is not in core (lines 16 to 18).

It should be noted, that CORE-PHIDM module in complex configurations, where more than one *subCORE* block is used, loads more subsequent parts of *DT* into the RAM_n memories (line 5). Similarly, comparisons between decision attribute's values and condition attributes' values for objects take place between object selected from RAM_{cmn} and objects selected from each RAM_n block of existing *subCORE* modules (lines 8 to 19).

2.4 Example of Core Calculation for Definition-based Algorithm

Table 1 presents exemplary decision table consisting of 4 condition attributes and one decision attribute. Dataset describes decision related to concrete mixing on basis of thresholded and descriptive readings from 4 types of sensors: outlook, temperature value, humidity and wind strength.

Table 1. Exemplary dataset consisting of 4 condition and 1 decision attribute

ID	outlook	temp	humidity	windy	mixing
1	sunnny	high	high	no	no
2	sunnny	high	high	yes	no
3	cloudy	high	high	no	yes
4	sunnny	low	high	no	yes
5	sunnny	high	normal	no	yes
6	sunnny	high	normal	yes	no
7	cloudy	high	normal	yes	yes
8	sunnny	low	high	no	no
9	sunnny	high	normal	no	yes
10	sunnny	low	normal	no	yes
11	sunnny	low	normal	yes	yes
12	cloudy	low	high	yes	yes

Before core is calculated, discernibility matrix has to be created first. Using formulas presented in Section 2.1 for calculating discernibility matrix *DM*, this type of matrix was created for Table 1 and os presented in Table 2. Matrix is symmetrical to it's diagonal, so only lower triangle is calculated. Additionally, to limit size of the table, discernibility matrix *DM* was calculated for only 6 first objects from dataset. The result is shown in Table 2.

According to definition-based core calculation algorithm described in Section 2.4, example of core calculation for discernibility matrix *DM* shown in Table 2 is presented below. In the beginning, core

C is empty. Two main loops make the first iteration passing through all cells of the discernibility matrix. The condition of adding the attribute to the core is checked for each of them. The first such case occurs for the cell at the intersection of objects 1 and 3. This cell contains only one attribute - $\{o\}$. This attribute does not exist in the core, so it is added to it, so: $C = \{o\}$. After completing the remaining algorithm steps, the final core $C = \{o,t,h,w\}$, what corresponds to $C = \{outlook, temp, humidity, windy\}$.

Table 2. Discernibility matrix for 6 first objects from exemplary dataset

ID	1	2	3	4	5	6
1	$\emptyset$					
2	$\emptyset$	$\emptyset$				
3	$\{o\}$	$\{o,w\}$	$\emptyset$			
4	$\{t\}$	$\{t,w\}$	$\emptyset$	$\emptyset$		
5	$\{h\}$	$\{h,w\}$	$\emptyset$	$\emptyset$	$\emptyset$	
6	$\emptyset$	$\emptyset$	$\{o,h,w\}$	$\{t,h,w\}$	$\{w\}$	$\emptyset$

2.5 Data to Conduct Experimental Research

In this paper, we present the results of the conducted experiments using two datasets: Poker Hand Dataset (created by Robert Cattral and Franz Oppacher) and data about children with insulin-dependent diabetes mellitus (type 1) (created by Jaroslaw Stepaniuk).

First dataset was obtained from UCI Machine Learning Repository [15]. Each of 1 000 000 records is an example of a hand consisting of five playing cards drawn from a standard deck of 52. Each card is described using two attributes (suit and rank), for a total of 10 predictive attributes. There is one decision attribute that describes the "Poker Hand". Decision attribute describes 10 possible combinations of cards in descending probability in the dataset: nothing in hand, one pair, two pairs, three of a kind, straight, flush, full house, four of a kind, straight flush, royal flush

Insulin-dependent diabetes mellitus is a chronic disease of the body's metabolism characterized by an inability to produce enough insulin to process carbohydrates, fat, and protein efficiently. Treatment requires injections of insulin. Twelve condition attributes, which include the results of physical and laboratory examinations and one decision attribute (microalbuminuria) describe the database used in our experiments. The data collection so far consists of 107 cases. The database is shown at the end of the paper [23]. A detailed analysis of the above data (only with the use of software systems) is in chapter 6 of the book [24].

The Poker Hand database was used for creating smaller datasets consisting of 1 000 to 500 000 of objects by selecting given number of rows of original dataset maintaining decision class distribution. Diabetes database was used for generating bigger datasets consisting of 1 000 to 1 000 000 of objects. New datasets were created by multiplying the rows of original dataset.

Created datasets had to be transformed to binary version. Numerical values were discretized and each attributes' value was encoded using four bits for both datasets. Every single object was described on 44 bits for Poker Hand and 52 bits for Diabetes. To fit to memory boundaries in both cases, objects descriptions had to be extended to 64 bits words filling unused attributes with binary 0's.

3 Hardware Implementation

Block diagram of the hardware implementation of the algorithm CORE-PHIDM is shown on Figure 1. System consists of the following blocks:

– RAM_{cmn} – a part of the decision table that is compared with the parts of the decision table stored in the other *subCORE* Generator Blocks;

– *subCORE* Generator Block – a block (described later in this section) that performs comparison between two parts of the decision table.

With every iteration of loop 2-23 (see CORE-PHIDM algorithm in Section 2.3) RAM_{cmn} is filled with a part of the decision table (in line 3). Every *subCORE* also contains the parts of the decision table. *subCORE* calculates the value of the sub-core in a way described in lines 7-19. Because there can be more than one instance of the *subCORE*, loop 7-19 can be multiplied and executed in parallel for faster calculations.

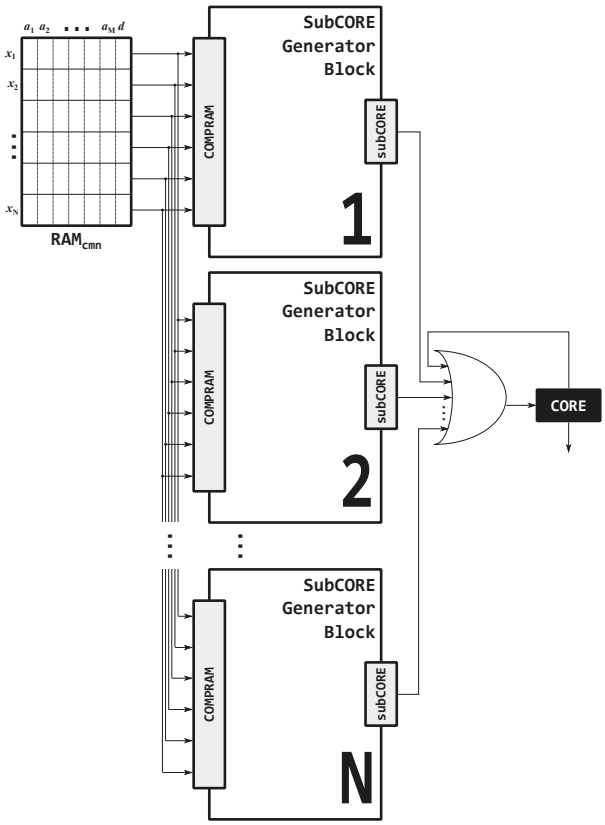

Figure 1. Main system architecture

Block diagram of the *subCORE* is shown in Figure 2. Block consists of the following components:

- *CB* – **c**omparator **b**lock, that performs comparison between two objects from decision table; when attributes have different values the corresponding bit is equal 1, otherwise 0 (line 11);
- *SD* – **s**ingleton **d**etector passes from input to output only singletons; when the value on the input is not singleton, the output consists of all 0's (line 16);
- *OR* gates cascade – every output of *SD* is *OR*'ed and the result is *subCORE* (line 17);
- RAM_n – RAM for storing the part of the decision table (line 5);
- MUX_n – multiplexer controlled by *Control Logic*, that selects the following object for comparison (line 7);
- *Control Logic* – block that generates the sequence for selecting the following objects from RAM (line 7).

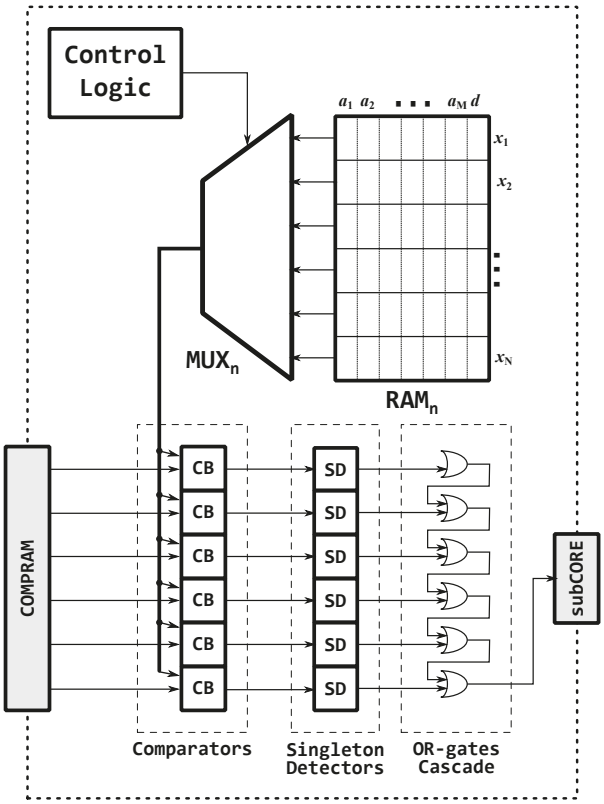

Figure 2. subCORE Generator Block block diagram

Data from RAM_{cmn}, consisting the objects from decision table is passed to the *CB*'s. Second input of every *CB* is connected to MUX_n, that selects the following object from RAM_n. All *CB*'s outputs are filtered by the *SD*, and then *OR*'ed together to obtain the sub-core.

3.1 Example of Operation

For the clarity of presentation, only the most important data flow operations are shown. Hardware processing unit require dataset to be transformed into binary version. Dataset shown in Table 1 after transformation is presented in Table 3.

For clarity of presentation and with correspondce to CORE-PHIDM algorithm described in Section 2.3 example shows most simple configuration of CORE-PHIDM block: one instance of *subCORE* module with two memory blocks denoted as RAM_{cmn} (common RAM connected to all modules) and RAM_1 (RAM for first *subCORE* module).

Values of the CORE-PHIDM block inputs are:

- RAM_{cmn} = 00111 01111 10110 10101 10011 01011 11010 00101 10011 10001 11001 11100,
- RAM_1 = 00111 01111 10110 10101 10011 01011 11010 00101 10011 10001 11001 11100,

what corresponds to data in exemplary binary decision table. Most significant bit (MSB) describes decision attribute, while LSB correlates to *outlook* attribute.

Table 3. Exemplary binary dataset consisting of 4 condition and 1 decision attribute

ID	outlook	temp	humidity	windy	mixing
1	1	1	1	0	0
2	1	1	1	1	0
3	0	1	1	0	1
4	1	0	1	0	1
5	1	1	0	0	1
6	1	1	0	1	0
7	0	1	0	1	1
8	1	0	1	0	0
9	1	1	0	0	1
10	1	0	0	0	1
11	1	0	0	1	1
12	0	0	1	1	1

Muliplexer MUX_n chooses ID_1 object from RAM_{cmn} represented by 00111 in the first clock cycle. This object is compared with all remaining objects in RAM_1 by comparators *CB*. Calculated value corresponds to the first column of discernibility matrix created for the dataset. Binary word created by comparators is 0000 0000 0001 0010 0100 0000 1101 0000 0100 0110 1110 1011. Each of 4-bit subsequences is passed to given OR gate connected in cascade. Parallelly, whole word created by comparators is routed to the *SD* block that detects 4-bits long subsequences containing only one logical 1. Result created by *SD* is 001110001000. Each bit of created value controls single OR gate in cascade. MSB is connected to the first OR gate. Values of inputs and outputs of every gate are:

- $IN_{1CB} = 0000$; $IN_{1PREV} = 0000$; $IN_{1SD} = 0$; $OUT_1 = 0000$,
- $IN_{2CB} = 0000$; $IN_{2PREV} = 0000$; $IN_{2SD} = 0$; $OUT_2 = 0000$,
- $IN_{3CB} = 0001$; $IN_{3PREV} = 0000$; $IN_{3SD} = 1$; $OUT_3 = 0001$,
- $IN_{4CB} = 0010$; $IN_{4PREV} = 0001$; $IN_{4SD} = 1$; $OUT_4 = 0011$,
- $IN_{5CB} = 0100$; $IN_{5PREV} = 0011$; $IN_{5SD} = 1$; $OUT_5 = 0111$,
- $IN_{6CB} = 0000$; $IN_{6PREV} = 0111$; $IN_{6SD} = 0$; $OUT_6 = 0111$,
- $IN_{7CB} = 1101$; $IN_{7PREV} = 0111$; $IN_{7SD} = 0$; $OUT_7 = 0111$,
- $IN_{8CB} = 0000$; $IN_{8PREV} = 0111$; $IN_{8SD} = 0$; $OUT_8 = 0111$,
- $IN_{9CB} = 0100$; $IN_{9PREV} = 0111$; $IN_{9SD} = 1$; $OUT_9 = 0111$,
- $IN_{10CB} = 0110$; $IN_{10PREV} = 0111$; $IN_{10SD} = 0$; $OUT_{10} = 0111$,
- $IN_{11CB} = 1110$; $IN_{11PREV} = 0111$; $IN_{11SD} = 0$; $OUT_{11} = 0111$,
- $IN_{12CB} = 1011$; $IN_{12PREV} = 0111$; $IN_{12SD} = 0$; $OUT_{12} = 0111$,

IN_{CB} input corresponds to value calculated by given comparator from *CB* block. IN_{PREV} input is connected with OUT output of preceding gate. Output of last gate is passed to *TEMP* register.

In next clock cycles CORE-PHIDM module processes rest of the objects in RAM_{cmn} by comparing them with remaining part of dataset in RAM_1. Every time previous value of *CORE* output is merged with it's current value and the *subCORE* by OR operation.

Finally created core is $C = 1111$, what corresponds to $C = \{outlook, temp, humidity, windy\}$.

4 Experimental Results

Software version of CORE-PHIDM algorithm described in Section 2.3 was implemented in C language. It should be noted, that implementation used only one PC CPU core to create basis for comparision between PC and FPGA-based solution. The results of the software execution were obtained using a PC equipped with an 8 GB RAM and 4-core Intel Core i7 3632QM with maximum 3.2 GHz in

Turbo mode clock speed running Windows 10 operational system. The source code of application was compiled using the GNU GCC 9.2 compiler.

Quartus II 13.1 was used for design, compilation, synthesis and verifying simulation of the hardware implementation in VHDL language. Synthesized hardware blocks were downloaded and run on TeraSIC DE-3 equipped with Stratix III EP3SL150F1152C2N FPGA chip. FPGA clock running at 50 MHz for the sequential parts of the project was derived from development board oscillator. Implemented CORE-PHIDM algorithm is presented in Section 2.3.

Table 4. Comparison of execution time between hardware and software implementation of CORE-PHIDM algorithm for both datasets using 1 instance of *subCORE* module

Objects	t_H	t_S	$C = \frac{t_S}{t_H}$	$C_{clk} = 64\frac{t_S}{t_H}$
—	[s]	[s]	—	—
Poker Hand dataset				
1k	0.003	0.033	10.875	695.992
2.5k	0.013	0.143	11.119	711.632
5k	0.055	0.603	10.951	700.876
10k	0.207	2.410	11.623	743.851
25k	1.225	14.721	12.015	768.940
50k	4.710	58.726	12.469	798.043
100k	21.737	237.942	10.946	700.572
250k	130.947	1 515.449	11.573	740.674
500k	506.225	6 092.916	12.036	770.302
1M	1 850.523	24 313.094	13.138	840.864
Diabetes dataset				
1k	0.003	0.018	5.911	378.325
2.5k	0.013	0.078	6.044	386.827
5k	0.055	0.328	5.953	380.980
10k	0.207	1.31	6.318	404.340
25k	1.225	8.002	6.531	417.978
50k	4.710	34.216	7.265	464.970
100k	21.737	135.309	6.225	398.390
250k	130.947	861.781	6.581	421.194
500k	506.225	3 464.821	6.844	438.043
1M	1 850.523	13 825.976	7.471	478.169

NIOS II softcore processor, as well as most parts of embedded system were instantiated using Qsys 13.1 tool. Software for NIOS II was implemented in C language using NIOS II Software Build Tools for Eclipse IDE.

Timing results were obtained using LeCroy waveSurfer 104MXs-B (1 GHz bandwidth, 10 GS/s) oscilloscope. For longer times, hardware time measurement units instantiated inside FPGA were used.

It should be noticed, that PCs clock is $\frac{clk_{PC}}{clk_{FPGA}} = 64$ times faster than development boards clock source.

Table 5. Comparison of execution time between hardware and software implementation of CORE-PHIDM algorithm for both datasets using 2 instances of *subCORE* module

Objects	t_H	t_S	$C = \frac{t_S}{t_H}$	$C_{clk} = 64\frac{t_S}{t_H}$
—	[s]	[s]	—	—
Poker Hand dataset				
1k	0.002	0.033	17.770	1 137.250
2.5k	0.008	0.143	18.169	1 162.806
5k	0.0340	0.603	17.894	1 145.231
10k	0.127	2.410	18.991	1 215.453
25k	0.750	14.721	19.632	1 256.448
50k	2.882	58.726	20.375	1 304.003
100k	13.303	237.942	17.886	1 144.734
250k	80.139	1 515.449	18.910	1 210.261
500k	309.807	6 092.916	19.667	1 258.674
1M	1 132.511	24 313.094	21.468	1 373.971
Diabetes dataset				
1k	0.002	0.018	9.659	618.183
2.5k	0.008	0.078	9.876	632.075
5k	0.034	0.328	9.727	622.521
10k	0.127	1.31	10.323	660.692
25k	0.750	8.002	10.672	682.977
50k	2.882	34.216	11.871	759.761
100k	13.303	135.309	10.171	650.969
250k	80.139	861.781	10.754	688.232
500k	309.807	3 464.821	11.184	715.762
1M	1 132.511	13 825.976	12.208	781.328

All calculations were performed using datasets described in Section 2.5 with sizes between 1 000 and 1 000 000 objects. Data was preprocessed on PC in terms of binary transformation and discretization for all cases.

Table 4 presents results of the time elapsed for hardware (t_H) and software (t_S) solution using indirect row-by-row discernibility matrix calculation (algorithm CORE-PHIDM described in Section 2.3) for both datasets.

Hardware based solution in first case uses 1 instance of *subCORE* generator block. Parts of dataset are stored in RAM_{cmn} and RAM_1.

Last two columns in following tables describe the speed-up factor without (C) and with (C_{clk}) taking clock speed factor between PC and FPGA into consideration. Abbreviations in objects number are: $k = 10^3$, $M = 10^6$.

Table 5 shows results for time elapsed for hardware (t_H) and software (t_S) solution using indirect row-by-row discernibility matrix calculation for both datasets, where hardware part, in this case, uses 2 instances of *subCORE* module. Parts of dataset are stored in RAM_{cmn}, RAM_1 and RAM_2.

Table 6 presents results for time elapsed for hardware (t_H) and software (t_S) solution using indirect row-by-row discernibility matrix calculation for both datasets, where hardware part in this case uses 4 instances of *subCORE* module. Parts of dataset were stored in RAM_{cmn}, RAM_1, RAM_2, RAM_3 and RAM_4.

Table 6. Comparison of execution time between hardware and software implementation of CORE-PHIDM algorithm for both datasets using 4 instances of *subCORE* module

Objects	t_H	t_S	$C = \frac{t_S}{t_H}$	$C_{clk} = 64\frac{t_S}{t_H}$
—	[s]	[s]	—	—
Poker Hand dataset				
1k	0.001	0.033	30.741	1 967.443
2.5k	0.005	0.143	31.432	2 011.655
5k	0.019	0.603	30.957	1 981.249
10k	0.073	2.410	32.855	2 102.733
25k	0.433	14.721	33.963	2 173.656
50k	1.666	58.726	35.249	2 255.924
100k	7.690	237.942	30.944	1 980.390
250k	46.323	1 515.449	32.715	2 093.752
500k	179.079	6 092.916	34.024	2 177.506
1M	654.631	24 313.094	37.140	2 376.970
Diabetes dataset				
1k	0.001	0.018	16.710	1 069.457
2.5k	0.005	0.078	17.086	1 093.490
5k	0.019	0.328	16.828	1 076.962
10k	0.073	1.31	17.859	1 142.998
25k	0.433	8.002	18.462	1 181.550
50k	1.666	34.216	20.537	1 314.387
100k	7.690	135.309	17.596	1 126.176
250k	46.323	861.781	18.604	1 190.641
500k	179.079	3 464.821	19.348	1 238.269
1M	654.631	13 825.976	21.120	1 351.697

FPGA resources utilization is fixed for given hardware core calculation configuration and is independent of the input dataset size. Datasets are divided into parts which are processed by the module. Different modules configurations for CORE-PHIDM use:

– 21 562 Logical Elements (LE) with 1 instance of *subCORE* module,

– 33 234 Logical Elements (LE) with 2 instances of *subCORE* module,

– 45 668 Logical Elements (LE) with 4 instances of *subCORE* module.

There are 113 600 Logical Elements (LE) available in Startix III FPGA. These numbers also include resources consumed by NIOS II processor.

Figure 3 presents a graphs showing the relationship between the number of objects and execution time for hardware and software solutions using different configurations of *subCORE* modules for CORE-PHIDM algorithm. Both axes have the logarithmic scale.

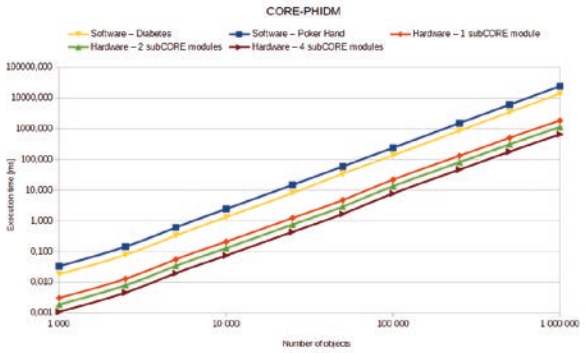

Figure 3. Relationship between number of objects and calculation time for hardware and software implementation of CORE-PHIDM for different *subCORE* module configurations

Presented results show big increase in the speed of data processing for all presented solutions. Hardware module execution time compared to the software implementation using row-by-row discernibility matrix calculation (algorithm CORE-PHIDM) is 5 (1 instance of *subCORE* module) to 37 (4 instances of *subCORE* module) times faster. If we take clock speed difference between PC and FPGA under consideration, these results are much better – average speed-up factor is 378 (1 instance) to 2 376 (4 instances). Speed-up factor is almost constant for given configuration of *subCORE* module and is similar for all sizes of processed datasets.

Hardware processing times for given algorithm for both datasets are the same – it doesn't matter what is the width in bits of single object from dataset, unless it fits in assumed memory boundary. Hardware processing unit takes the same time to finish the calculation for every object size, because

it always performs the same type of operation. This is true for all configurations of presented hardware core calculation modules.

It should be noted, that average speed-up factor related to number of instances of core calculation module for CORE-PHIDM is not linear and is equal to:

- 1.634 for configuration of 2 instances of core calculation module,
- 2.827 for configuration of 4 instances of core calculation module.

Decreasing speed-up factor is realted to NIOS II processor overhead needed for binary data copying to RAM_n memories what needs increasing time for this operation.

Let comparison of attribute value between two objects or retrieving the element from discernibility matrix be an elementary operation. k denotes number of conditional attributes, n is the number of objects in decision table and p is number of *subCORE* module instances. Computational complexity of software implementation for the core calculation using CORE-PHIDM algorithm is $\Theta(kn^2)$. Using hardware implementation, complexity of core calculation is $\Theta(\frac{n^2}{p})$. The k is missing in hardware implementation, because presented solutions perform comparison between all attributes in $\Theta(1)$ - all attributes values between two objects are compared in single clock cycle. Additionally, core module performs comparisons between many objects at time. In most cases $k << n$, while $p << k$, so we can say, that computational complexity for software and hardware implementations are the same.

5 Conclusions and Future Research

Hardware parallel implementation of a core calculation algorithm gives us a big acceleration in comparison to a software solution. This approach is the main direction of using scalable decision systems in solutions demanding results of calculations in a short time.

Core hardware calculation units were not optimized for performance. Processing time can be substantially reduced by increasing FPGA clock frequency and by introducing triggering on both clock edges. The hardware solution presented in this paper is easily scalable. Multiplying core calculation blocks improved the processing speed. Currently, 4 parallel units occupy only about 50% of a pretty small FPGA.

Further research will focus on checking different sizes of modules, optimizing the modules for performance as well as optimizing the data transfer between the decision table and core calculation units.

The type of processed data must also be taken into consideration. The presented solution is suitable for consistent datasets. This approach doesn't handle databases with missing values properly. In recent years increasing the popularity of systems dealing with incomplete information is gaining popularity. The algorithm and its implementation should be modified for processing this type of data. Examples of such an approach for software solutions can be found in e.g. [26].

Acknowledgments

The work was supported by the grant W/WI/2/2020, WZ/WI-IIT/2/2020 from Bialystok University of Technology and funded with resources for research by the Ministry of Science and Higher Education in Poland.

References

[1] Borowik, G.; Jankowski, J.; Kowalski K. Fast algorithm for feature extraction. Proceedings of SPIE 9662, In Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments, Proceedings of SPIE 2015, pp. 1110-1117.

[2] Choromański, M.; Grześ, T.; Hońko, P. Two FPGA Devices in the Problem of Finding Minimal Reducts. In Lecture Notes in Computer Science, Publisher: Springer, 2019, Vol. 11703, pp. 410-420.

[3] Czołombitko, M.; Stepaniuk, J. Generating core based on discernibility measure and MapReduce. Proceedings of the Pattern recognition and machine intelligence: 6th International conference, PReMI 2015, Warsaw, Poland, June 30-July 3, 2015, Lecture Notes in Computer Science, vol. 9124, pp. 367-376.

[4] T. Geng et al., O3BNN-R: An Out-of-Order Architecture for High-Performance and Regularized BNN Inference, in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 1, 1 Jan. 2021, doi: 10.1109/TPDS.2020.3013637, pp. 199-213.

[5] Grześ T., Kopczyński M. Hardware Implementation on Field Programmable Gate Array of Two-Stage Algorithm for Rough Set Reduct Generation. In Lecture Notes in Computer Science, Publisher: Springer, 2019, Vol. 11499, pp. 495-506.

[6] Y. Liang, L. Lu and J. Xie, OMNI: A Framework for Integrating Hardware and Software Optimizations for Sparse CNNs in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, doi: 10.1109/TCAD.2020.3023903.

[7] Z. -N. Li, C. Zhu, Y. -L. Gao, Z. -K. Wang and J. Wang, AlphaGo Policy Network: A DCNN Accelerator on FPGA in IEEE Access, doi: 10.1109/ACCESS.2020.3023739.

[8] Kanasugi, A.; Yokoyama, A. A basic design for rough set processor. In Proceedings of the 15th Annual Conference of Japanese Society for Artificial Intelligence, 2001.

[9] Kopczyński, M.; Stepaniuk, J. Hardware Implementations of Rough Set Methods in Programmable Logic Devices. In Rough Sets and Intelligent Systems – Professor Zdzisław Pawlak in Memoriam, Intelligent Systems Reference Library 43, Heidelberg, Springer; 2013, pp. 309-321.

[10] Kopczyński, M.; Grześ, T.; Stepaniuk, J. FPGA in Rough-Granular Computing: Reduct Generation. In proceedings of the 2014 IEEE/WCI/ACM International Joint Conferences on Web Intelligence Vol.2, Warsaw, IEEE Computer Society, 2014, pp. 364-370.

[11] Kopczyński, M.; Grześ, T.; Stepaniuk, J. Generating core in rough set theory: Design and implementation on FPGA. In Lecture Notes in Computer Science Vol. 8537, Berlin, Springer-Verlag, 2014, pp. 209-216.

[12] Kopczyński, M.; Grześ, T.; Stepaniuk, J. Computation of Cores in Big Datasets: An FPGA Approach. In Lecture Notes in Computer Science Vol.9436, Berlin, Springer-Verlag, 2015, pp. 153-163.

[13] Kopczyński, M.; Grześ, T.; Stepaniuk, J. Core for Large Datasets: Rough Sets on FPGA. Fundamenta Informaticae 2016, 147, pp. 241-259.

[14] Lewis, T.; Perkowski, M.; Jozwiak, L. Learning in Hardware: Architecture and Implementation of an FPGA-Based Rough Set Machine. In proceedings of the Euromicro, 25th Euromicro Conference (EUROMICRO '99), Volume 1; 1999, pp. 13-26.

[15] Lichman, M. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science; 2013.

[16] Muraszkiewicz, M.; Rybiński, H. Towards a Parallel Rough Sets Computer In: Rough Sets, Fuzzy Sets and Knowledge Discovery. Springer-Verlag; 1994, pp. 434-443.

[17] Marz, N.; Warren, J.H. Big Data: Principles and best practices of scalable realtime data systems. Manning Publications Co. Greenwich; 2015.

[18] N. Narsale and V. Agarwal, Implementation Of LEM2 algorithm On FPGA, 2019 3rd International Conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 2019, doi: 10.1109/ICECA.2019.8822143, pp. 1-5.

[19] Nguyen, S. H.; Nguyen H. S. Some Efficient Algorithms for Rough Set Methods. In Sixth International Conference on Information Processing and Management of Uncertainty on Knowledge Based Systems IPMU'1996, volume III, Granada, Spain, July 1-5 1996, pp 1451–1456.

[20] Nguyen, H.S. Approximate Boolean Reasoning: Foundations and Applications in Data Mining. Transactions on Rough Sets V, Lecture Notes in Computer Science Vol. 4100, Berlin, Springer-Verlag; 2006, pp. 334-506.

[21] Pawlak, Z. Elementary rough set granules: Toward a rough set processor. In Rough-Neurocomputing: Techniques for Computing with Words, Cognitive Technologies, Springer-Verlag, Berlin, Germany; 2004, p. 5-14.

[22] Pawlak, Z.; Skowron, A. Rudiments of rough sets. Information Sciences; 2007; 177(1), p. 3-27.

[23] Stepaniuk, J. Knowledge discovery by application of rough set models. In Rough Set Methods and Applications. New Developments in Knowledge Discovery, Information Systems, Physica-Verlag, Heidelberg; 2000, p. 137-233.

[24] Stepaniuk, J. Rough-Granular Computing in Knowledge Discovery and Data Mining. Springer, 2008.

[25] Stepaniuk, J.; Kopczyński, M.; Grześ, T. The First Step Toward Processor for Rough Set Methods. Fundamenta Informaticae 2013; 127, pp. 429-443.

[26] Sun, L.; Xu, J.; Li, Y. A Feature Selection Approach of Inconsistent Decision Systems in Rough Set. Journal of Computers, vol. 9, Academy Publisher; 2014, pp. 1333-1340.

[27] Tiwari, K.S.; Kothari, A.G. Design and Implementation of Rough Set Algorithms on FPGA: A Survey. IJARAI, 2014, 3, no. 9, pp. 14-23.

[28] Zhang, J.; Wong, J.; Pan, Y.; Li, T. A parallel matrix-based method for computing approximations in incomplete information systems. In IEEE Transactions on Knowledge Data Engineering, 2015, 27, p. 326-339.

Maciej Kopczyński is an experienced professional working at the Bialystok University of Technology since 2008. He had finished his Ph.D. in 2017 mainly focusing on hardware implementation of selected AI algorithms in FPGA structures. Maciej conducts classes related to entrepreneurship, programming, web application development, artificial intelligence and algorithms. Maciej is a co-founder of four companies: Photon Entertainment, RiftCat, Anima and MakeIT. He also has extensive experience in working with industry in the aspect of implementing projects in the field of very advanced electronics (mainly RF systems) and complex software solutions.

Tomasz Grześ is an experienced professional working at the Bialystok University of Technology since 1998. He had finished his Ph.D. in 2010 focusing on power minimization in sequential circuits on programmable logic devices. Tomasz conducts classes related to digital systems design, programming, computer networks, as well as embedded and mobile systems. He also has extensive experience in working with the industry in the aspect of implementing projects in the field of advanced control and management systems, firmware development, and specialized digital systems design.