An FPGA implementation of restricted Boltzmann machine

1st Yichao Zhu
Electrical, Computer and Biomedical Engineering
The University of Rhode Island
Kingston, United States
yichao_zhu@uri.edu

***Abstract*—In the rapidly evolving landscape of the Internet of Things (IoT), the demand for cloud computing has surged consistently. To counteract the heavy burden on cloud resources and combat undesirable network latency, edge computing—a decentralized paradigm—has gained substantial traction among academic and industrial researchers. However, constrained by cost and varied application contexts, the hardware resources of edge devices often prove inadequate. This limitation poses a significant challenge to seamless data transmission to edge computing nodes. The main objective of this project is to find a solution to effectively transmit a substantial volume of sensor data to edge computing nodes. To achieve this, a data reduction strategy is proposed. This approach leverages the Restricted Boltzmann Machine (RBM) model to extract crucial features from sensor data, thereby reducing transmitted data volume. The RBM, a probabilistic graphical model, excels in both data dimensionality reduction and reconstruction. By transmitting data in a condensed form and performing high-dimensional data reconstruction on edge computing nodes, the transmission burden is significantly alleviated. To facilitate RBM availability on edge devices, this project suggests an FPGA-based implementation. The experimentation is conducted on the Xilinx PYNQ-Z2 FPGA board, utilizing the xc7z020-clg400-1 FPGA device. This platform supports Linux installation and seamlessly integrates the FPGA as a system-mounted device. The implementation is applied within an Advanced Driver Assistance System (ADAS) context, effectively reducing radar sensor data size and addressing ADAS computer interface bandwidth limitations. In comparison to traditional software platforms, the FPGA implementation achieves faster throughput of up to 143 times, while maintaining accuracy with minimal discernible compromise.**

***Index Terms*—Edge computing, data reduction, RBM, FPGA**

I. Introduction

Amidst the swift evolution of an intelligent society and the perpetual refinement of human needs, intelligence has permeated various industries and the everyday fabric of society. The advent of the Internet of Things (IoT) stands as a cornerstone of this transformative trend, where an array of smart devices connect to cloud servers. However, the cost associated with data transmission and processing on cloud servers remains a considerable challenge. The inherent limitations of physical infrastructure unavoidably lead to substantial network latency [1]. In response to these challenges, a novel decentralized computing model known as edge computing has gained significant traction and recognition among researchers in both academia and industry. Edge computing is distinguished by its attributes of high bandwidth, ultra-low latency, and real-time data processing. This paradigm shift involves processing data at edge nodes rather than transmitting it to distant cloud servers. By doing so, this architectural approach offers geographical proximity, resulting in accelerated service response times and an alleviated burden on cloud workloads. However, the practical implementation of edge computing is not without its complexities. Limited hardware resources remain a challenge for edge devices, hindering their ability to execute intricate algorithms for autonomous data processing. Furthermore, the transmission of data to computing nodes demanding high bandwidth presents an additional strain on the hardware [2].

In the realm of edge computing applications, it's customary for sensor raw data acquired through edge devices to encompass multidimensional information, often in substantial volumes. Transmitting such extensive data directly from sensors to computing nodes imposes a significant burden on interface bandwidth. This, in turn, gives rise to noteworthy additional costs and emerges as a bottleneck within the edge computing system. Dimensionality reduction technology assumes a pivotal role as a technology aimed at diminishing the size of sensor data while upholding crucial data information. Traditional signal processing algorithms such as the high correlation filter, Principal Component Analysis, and General Discriminant Analysis prove invaluable in achieving this dimensionality reduction [3]. As deep learning advances, data dimensionality methods grounded in neural networks, such as Vanilla Autoencoders [4], have been introduced and yielded exceptional results. However, as edge devices undertake the task of dimensionalizing data, the scarcity of computing resources emerges as a fresh bottleneck. This underscores the need for a hardware-friendly algorithm. To this end, the present project presents a solution for data dimensionality: the implementation of the FPGA Restricted Boltzmann Machine.

The RBM is an energy-based probabilistic graphical model known for its capabilities in unsupervised learning, data generation, and dimensionality reduction [5]. In this project, RBM assumes a dual role as both an encoder and a data generator. Its functions encompass extracting inherent features from raw data and encoding high-dimensional information into a more compact form. Through suitable training, RBM can generate reconstruction data from low-dimensional input, where the reconstructed data closely approximates the original data.

With its straightforward structure, often likened to a two-layer fully connected neural network, RBM proves to be a suitable choice for performing data reduction on raw input. Despite its hardware-friendly nature, the computational complexity of RBM can still pose challenges for edge devices. Taking into account key factors like cost, power consumption, flexibility, size, and performance, FPGA emerges as an optimal solution [6]. By strategically placing an FPGA between sensors and edge computing nodes, RBM can be efficiently implemented at a minimal cost, thereby facilitating the seamless transformation of high-dimensional to low-dimensional data.

An application of substantial raw data within the realm of edge computing is evident in autonomous vehicles that are equipped with sensors and Advanced Driver-Assistance System (ADAS) computers [7]. In this specific application, the sensor employed is the Millimeter wave (mmWave) radar [8], while the ADAS computer serves as the edge computing node. By emitting high-frequency electromagnetic waves, the mmWave radar captures echoes rebounding from objects, thus amassing a wealth of data. All radar data is subsequently transmitted to the ADAS computer, which engages in intricate data processing to derive crucial measurements, including distance, velocity, and angle of the detected objects. Due to the constrained interface bandwidth of the ADAS computer, the sheer volume of raw data generated by the radar poses a challenge, specifically in terms of the efficiency of data transmission between the radar and the computer. Positioning an FPGA with an integrated RBM hardware implementation between the radar and the ADAS computer serves as a strategic solution for addressing the bandwidth challenge. This approach facilitates data preprocessing before it reaches the computer, effectively alleviating the bandwidth limitations, all while maintaining a cost-effective solution

The main contributions of this paper cover the following points:

1. Propose a data reduction approach employing RBM to alleviate the data transmission burden on edge devices.

2. Conduct analysis and architectural design for the RBM FPGA, followed by its implementation on xc7z020-clg400-1.

3. Validate the role of the RBM FPGA implementation in the ADAS application scenario and compare it with a software-based solution.

II. Restricted Boltzmann Machine

RBM operates as a probabilistic energy-based model, finding utility in the realms of generative and unsupervised learning [9]. Similar to neural networks, it's adept at extracting features from input data, effecting a mapping of high-dimensional data to its lower-dimensional counterpart. Compared to traditional fitting-based neural networks, RBM is built upon a foundation of probability. RBM consists of two layers: a visible layer and a hidden layer, distinct from the conventional input and output layers. The connections within the model signify probabilistic relationships, and the units in both the visible and hidden layers can be acquired through sampling based on probabilistic computation [10]. The structure diagram is depicted in Fig. 1.

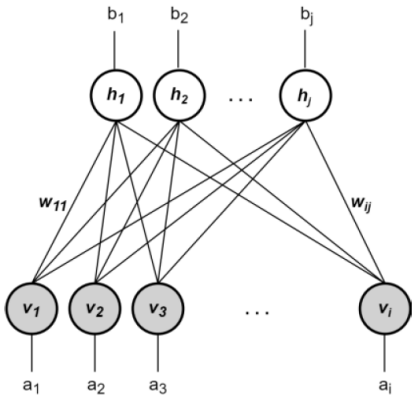


Fig. 1. RBM structure

Illustrated in the diagram, RBM exhibits an analogous structure to that of a fully connected neural network [11]. However, their computational rules differ. The computation process of RBM can be partitioned into two phases: the positive phase and the negative phase. In the positive phase, visible units can be utilized to compute the probability of hidden units, whereas, in the negative phase, hidden units can be employed to compute the probability of visible units in the opposite direction. In the classical binary RBM model, both the visible and hidden units take on binary values, specifically 0 or 1. To execute RBM effectively, the initial requirement is the definition of its energy function.

$$E(v,h) = -\sum_{i=1}^{n} a_i v_i - \sum_{j=1}^{m} b_j h_j - \sum_{i=1}^{n}\sum_{j=1}^{m} v_i h_j w_{ij} \quad (1)$$

Where n and m represent the quantities of units in the visible layer and hidden layer, respectively. The variables v_i and h_j denote the binary values of visible unit i and hidden unit j, while a_i and b_j denote their biases, and w_{ij} signifies the weight connecting them.

Being a probabilistic model, each combination of visible and hidden unit vectors corresponds to a probability. This function is as follows:

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)} \quad (2)$$

Where the symbol Z, referred to as the "partition function," represents the summation of all energies obtained from every conceivable pairing of visible and hidden vectors:

$$Z = \sum_{v,h} e^{-E(v,h)} \quad (3)$$

If our focus is solely on the probabilities of v or h, the probability function can be expressed as follows:

$$\begin{aligned} p(v) &= \frac{1}{Z}\sum_{h} e^{-E(v,h)} \\ p(h) &= \frac{1}{Z}\sum_{v} e^{-E(v,h)} \end{aligned} \tag{4}$$

By utilizing (2), (3), and (4), we can get the needed probability functions. Employing the conditional probability formula and drawing from the principles of the Bernoulli distribution, we can arrive at the conditional probability function as follows:

$$\begin{aligned} p(h_j{=}1|v) &= \sigma(\sum_{i=1}^{m} v_i w_{ij} + b_j) \\ p(v_i{=}1|h) &= \sigma(\sum_{j=1}^{n} h_j w_{ij} + a_i) \end{aligned} \tag{5}$$

In the above expressions, the function σ denotes the Sigmoid function, which is defined as follows:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{6}$$

The functions presented in (5) generate probability vectors for the visible or hidden units being equal to 1, based on the information provided by the other vector. This enables us to obtain vector values through sampling from the Bernoulli distribution. The positive phase is employed to transform high-dimensional input visible data into low-dimensional hidden data, followed by using the low-dimensional hidden data to reconstruct the visible data. When the RBM is effectively trained, the reconstructed data closely resembles the original input visible data. In the context of sensor data reduction, the sensor data values can be interpreted as the RBM visible vectors. Within an acceptable accuracy range, a higher data reduction rate is achieved by utilizing shorter hidden vectors [12].

In numerous real-world applications, data is seldom limited to binary values, prompting necessary modifications to the RBM. The Gaussian Bernoulli Restricted Boltzmann Machine (GBRBM), emerges as an enhanced rendition of the RBM. Analogous to the RBM, the GBRBM remains rooted in probability and employs sampling. However, unlike the RBM, the GBRBM's visible units encompass continuous values rather than binary ones, adhering to the Gaussian distribution instead of the Bernoulli distribution. Importantly, the structure of the hidden layer remains unchanged. The energy and conditional probability functions of the GBRBM are as follows [13]:

$$\begin{aligned} E(v,h) &= \sum_{i=1}^{m}\frac{(v_i-a_i)^2}{2\sigma^2} - \sum_{j=1}^{n} h_j b_j - \sum_{i=1}^{m}\sum_{j=1}^{n}\frac{v_i}{\sigma_i} h_j w_{ij} \\ p(h_j=1|v) &= sigmoid(\sum_{i=1}^{m}\frac{v_i}{\sigma_i} w_{ij} + b_j) \\ p(v_i=v|h) &= \mathcal{N}(\sigma_i \sum_{j=1}^{n} h_j w_{ij} + a_i) \end{aligned} \tag{7}$$

Where σ_i represents the variance of the corresponding visible unit, and $\mathcal{N}$ denotes the Gaussian density function. The visible input data in GBRBM is standardized to possess a unit variance and a mean of zero. With effective training, the model is capable of reconstructing the input visible data.

Whether it's RBM or GRBM, their positive phase incorporates a vector-matrix multiplication followed by a Sigmoid computation. This lends itself to the utilization of an RBM FPGA implementation for a diverse array of RBMs. As the dimensions increase, the computational complexity of the vector-matrix multiplication escalates swiftly. The overall count of necessary multiplications and additions is as follows:

$$N = (2m^2+1)n \tag{8}$$

Where, m denotes the length of the input vector, and n signifies the length of the output vector. In this project, the input vector's maximum length is 512, and the output vector's maximum length is 256, resulting in a substantial 67,109,120 times of computation. This represents a demanding computational workload for edge devices, particularly considering that sensor data is continuously fed into the system. As a result, a hardware acceleration solution is imperative.

III. ADAS APPLICATION

The autonomous driving system serves as a prime example of an edge computing application. To navigate obstacles effectively, vehicles must possess the capability to perceive their surroundings. This necessitates the integration of sensors, such as radars, within the vehicle. These radars emit electromagnetic waves, capture the reflecting waves, and subsequently relay this data to an ADAS computer. Subsequently, the computer processes the information to estimate the distance, velocity, and angle of the objects present in the vicinity [14].

Fig .2 illustrates the logical diagram of the proposed architecture for the ADAS system in this project. The left segment of the figure depicts the edge computing platform, comprising the radar sensor boards and the FPGA RBM implementation, while the right portion represents the ADAS computer equipped with the RBM reconstruction algorithm. On the radar board, the radar emits signals via the Tx antenna and captures reflection signals through the Rx antennas. To attain high-dimensional information, multiple receiving antennas are employed. These Rx signals are combined with Tx signals, transitioning into Intermediate Frequency (IF) signals, which are then sampled using an ADC. Following this, the signals undergo orthogonal modulation, transforming into I/Q complex signals. Given the substantial volume of radar data and the constraints imposed by the interface bandwidth of the ADAS computer, the data undergoes dimensionality reduction through FPGA RBM processing. Eventually, the reduced data is reconstructed within the ADAS computer, subsequently enabling the ADAS computer to make informed decisions. The parameters for the radar board are detailed in TABLE I.

The ADC sampling rate can reach up to 10 M/s, according to the table, and there are multiple ADC channels dedicated

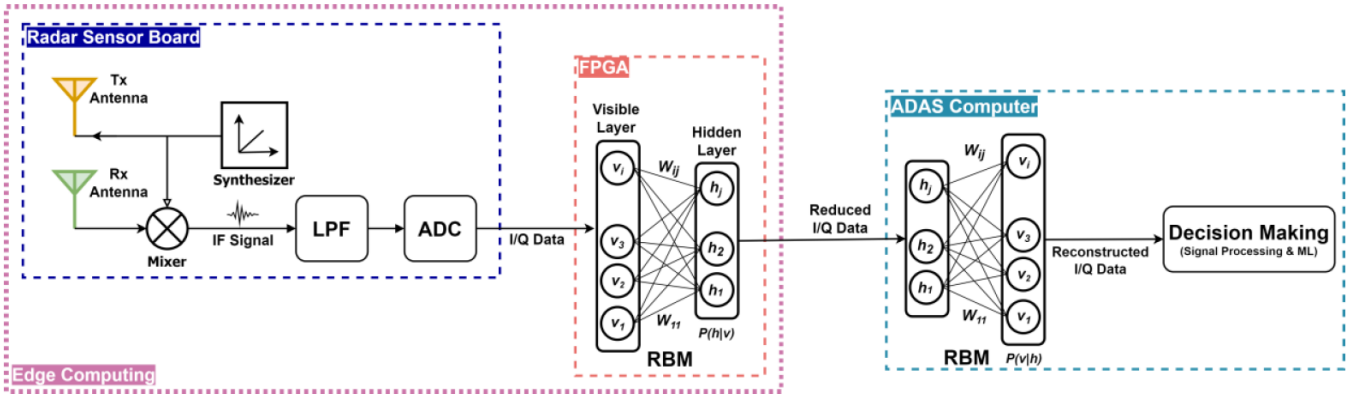


Fig. 2. ADAS system structure

TABLE I
RADER SENSOR CONFIGURATION PARAMETERS

Parameter	Value
Start Frequency (GHz)	77
Frequency Slope(MHz/μs)	29.982
Idle Time (μs)	100
Tx Start Time (μs)	6
ADC Start Time (μs)	6
ADC Samples	512/256/128/64
Sample Rate (ksps)	10000
Ramp End Time (μs)	60
Number of Chirp Loops	128

to the various Rx antennas. Given that the radar operates continuously, it generates a continuous stream of data. In the absence of data reduction, the radar's output would place a substantial load on the interface connecting the radar board and the ADAS computer. RBM offers a solution for data reduction; however, it demands significant computational power and real-time streaming data processing capabilities from edge devices. Leveraging the inherent patterns within radar signals, we can use them to train the RBM. Subsequently, the RBM's weights and biases can be embedded in an FPGA, enabling the utilization of these parameters for RBM calculations. To cater to diverse applications, the RBM FPGA implementation can incorporate a degree of flexibility, allowing it to perform calculations for RBMs of varying sizes.

IV. FPGA RBM ARCHITECTURE

To realize RBM on an FPGA, the entire RBM can be subdivided into multiple smaller kernels. This division aids in hardware routing. Given the application's nature, the input data for the RBM comprises continuous data from multiple ADC channels, necessitating efficient data stream processing and high throughput. For seamless real-time data processing, the FPGA implementation should feature a fully pipelined structure with an Initial Interval (II) value set to 1. The pipeline structure subdivides a complex task into smaller tasks that can be executed simultaneously. II refers to the interval between consecutive start times of a task. Fig. 3 visually represents the concept of pipeline and II. When the implementation achieves pipeline operation and II is set to 1, it can receive and process data during every FPGA clock cycle [15].

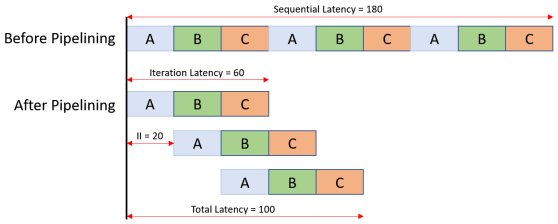


Fig. 3. Pipelining and II

To optimize development efficiency, this project departs from the conventional hardware description languages like VHDL or Verilog. Instead, it adopts Vitis HLS (High-Level Synthesis), a novel development tool from Xilinx [16]. Vitis HLS employs C-like code for FPGA programming. The HLS tool translates C++ code into Register-Transfer Level (RTL) description, circumventing the need for intricate interface programming and streamlining the syntax intricacies. The synthesis outcome is depicted in Table. II, signifying that the implementation is poised to attain an 80M clock frequency with II=1.

A. Overall architecture

The RBM FPGA implementation consists of several kernels. As described in sections III and IV, the task assigned to the FPGA is computing the positive phase of the RBM. The computation of the RBM positive phase entails two stages: vector-matrix multiplication computation and Sigmoid function computation, which serve as the primary computational kernels. Since the input data originates from multiple ADC channels, a reordering kernel is required. To enhance efficiency, it is important to minimize data transmission between the FPGA and DDR whenever feasible, necessitating a kernel for memory weight and bias. In order to orchestrate all these kernels, a controller and signal splitting kernels are essential components.

The comprehensive RBM hardware architecture is depicted in Fig. 4. The RBM implementation doesn't establish a direct connection with the host computer; instead, they communicate through a DDR. Because the data formats between the host computer and FPGA differ, an RBM interface kernel manages the task of data format conversion. Elaboration on each kernel is presented in the subsequent subsections.

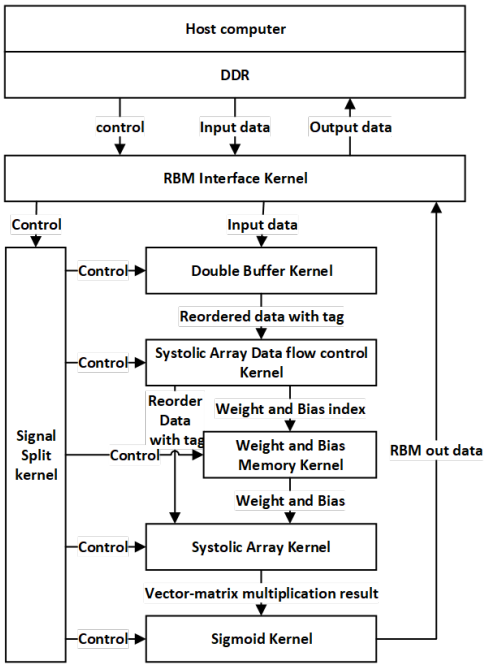


Fig. 4. FPGA RBM structure

B. Data precision

To optimize hardware resource utilization, careful consideration must be given to data type and precision. While a general-purpose computing platform often employs floating-point data types, performing float-based computations in hardware is less efficient, requiring more hardware resources compared to fixed-point data computation. Given the FPGA's hardware performance and the dynamic range of RBM parameters, it is feasible to implement fixed-point quantization for all computations.

The FPGA chip employed in this project is the xc7z020-clg400-1 model. This FPGA features an integrated hardware DSP core, DSP48E1, capable of executing a multiplication operation in a single cycle, as depicted in Fig .5. DSP48E1 supports a maximum multiplication precision of 18 bits by 25 bits. By fully leveraging DSP48E1's capabilities, a pre-addition of 25bits can be performed prior to multiplication, followed by an addition of 48 bits post multiplication, all without incurring additional hardware costs. The computation equation is outlined below:

$$output = C\pm(B(D\pm A)+C_{IN}) \tag{9}$$

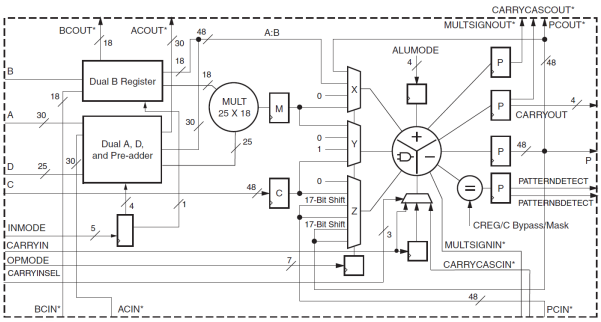


Fig. 5. DSP48E1's structure

The ADC output data from the radar board is in the form of 16-bit integer data. Consequently, the RBM's visible units employ a 16-bit integer data type. Upon analyzing the parameters of the RBM employed in this project, we observe that the absolute values of the weights are distributed within the range of [2.19e-7, 1.23], while the absolute values of the biases are distributed within the range of [1.39e-2, 26.13]. To maximize the utilization of DSP48E1, the weight can be configured as a 25-bit signed fixed-point integer type, with 3 bits in the integer part and 22 bits in the fractional part. The bias solely engages in additional operations and can be designated as a 48-bit type with a 22-bit fractional part, matching that of the weight. Considering the maximum input vector length in this project is 512, a 9-bit length, the highest potential outcome of vector-matrix multiplication is a 50-bit signed fixed-point number. This number consists of a 28-bit integer part and a 22-bit fractional part. For optimal hardware efficiency, the last two bits can be truncated, yielding a 48-bit data type with a 28-bit integer part.

TABLE II
RBM SYNTHESIS REPORTS

Modules	Latency (cycles)	Interval	Pipelined	BRAM	DSP	FF	LUT
RBM	42	1	dataflow	103	67	35621	28320
control_split	1	1	yes	0	0	13	64
rbm_size_split	1	1	yes	0	0	31	112
double_buffer	2	1	yes	4	0	472	725
data_flow_control	22	1	yes	0	0	18760	6493
weight_bias_memory	2	1	yes	66	0	1253	268
systolic_array	6	1	yes	0	64	11806	17894
sigmoid	6	1	yes	12	3	901	941

C. Double buffer

Figure 6 illustrates the data format of the ADAS radar board's output. The sensor data, acquired from multiple ADC channels, is streamed out continuously, with data from different ADC channels interleaved in sequential order. Since the data is processed in vector units, reordering the data according to their respective channels could significantly simplify the subsequent data processing structure.

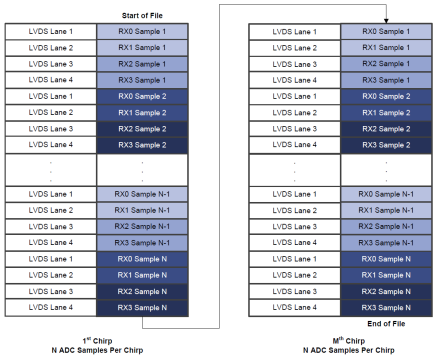


Fig. 6. Data format

Figure 7 depicts the double buffer structure utilized in this project. This configuration consists of two buffers, each encompassing several FIFOs corresponding to specific ADC channels. The two buffers operate alternately and do not simultaneously input or output data. When one buffer becomes full, it switches to the other buffer to receive incoming data. Simultaneously, the filled buffer continuously outputs internal data to the subsequent kernel until it is fully emptied. It remains in a standby state until the other buffer is filled once again, allowing it to receive data anew. The buffer's fullness criterion is based on the currently visible vector length of the RBM. For streamlined follow-up operations, each input data, upon transfer to the downstream structure, is augmented with a tag in addition to the data itself. This tag consists of two bits, one denoting the first data flag and the other the last data flag. These flags serve to identify the initial and final data points within a vector. The double buffer kernel effectively achieves the tasks of reordering input ADC data without encountering stalls and marking the first and last data points in a vector, thus facilitating subsequent data processing.

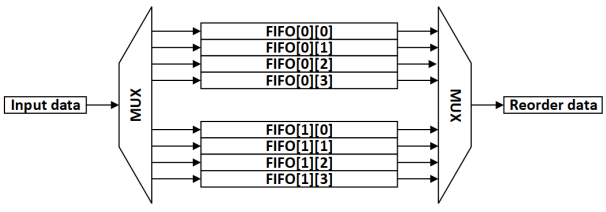


Fig. 7. Double buffer

D. Data flow controller and signal split

The data interchange between the FPGA and the host computer is in a stream format, encompassing the control signals as well, given that a single stream can only be connected to one kernel, a signal split kernel is employed to distribute the signals among all the kernels.

The data flow controller is positioned between the double and computation kernels. Apart from relocating the output data from the double buffer to the computation kernels, it can also generate padding data and transfer it to the computational kernel. Padding data doesn't participate in the computation; instead, it serves the purpose of flushing out any remaining data in the computation kernel, preventing valid data from becoming stuck. The data that traverses the data flow controller will be affixed with a flag, a padding flag, which indicates whether the data is padding data. Moreover, the data controller will read the data tag, keep track of the data index, and emit corresponding weight and bias indices to the weight and bias memory kernel. This enables the weight and bias memory kernel to correctly furnish the computational kernel with the appropriate weight and bias values.

E. Weight and bias memeory

The weight and bias memory kernel receives weight and bias inputs, stores them within the FPGA's on-chip memory blocks, and outputs them as required. It's important to note that FPGA RAM blocks have limited access interfaces, with each interface capable of performing either read or write operations at a single instance. In the context of this project, the vector-matrix computation kernel is implemented as a systolic array, capable of performing a number of computations between input data and weight elements simultaneously, up to the count of processing elements (PE). In order to obtain multiple weights within a single clock cycle, the weight memory is partitioned along the columns. This partitioning increases the number of available interfaces, allowing the memory to output a sufficient amount of data in a single clock cycle.

F. Systolic array

The vector-matrix multiplication holds paramount importance in the RBM positive phase computation. In this project, we prioritize throughput over latency due to the continuous stream of input data. The systolic array structure suits this purpose [17]. The systolic array mainly comprises a multitude of Processing Elements (PE), an input data delay line, and output registers. During operation, in each clock cycle, input data is fed into the data delay line's head. All data in the delay line shifts by one position from the head to the tail. Following this shift, a multiplication computation occurs between input data and weights in every PE, provided that the input data in the corresponding position is valid. Each PE corresponds to a vector dot product involving input data and a weight column vector. With data continuously flowing through the delay line, multiplications occur persistently. For a PE, a vector dot product initiates upon the arrival of the first data in the vector and concludes upon the arrival of the last data in the vector. As an entire input data vector traverses the delay line, a vector-matrix multiplication concludes. Each PE outputs its computation result to a register, which then incorporates the

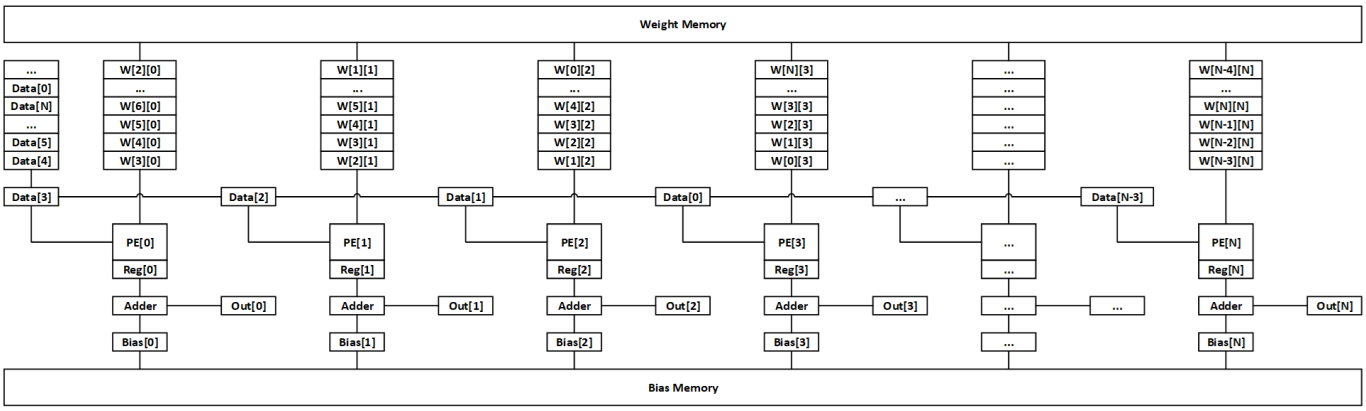


Fig. 8. Systolic array

bias. The resultant output data is subsequently transmitted to the next stage. The systolic structure is depicted in Fig .8.

Fig. 9 illustrates the structure of a PE. The primary function of the PE in this project is to oversee multiplication and accumulation operations. Upon the activation of the data valid flag, the PE reads the data and computes the product of the input data and the weight. This product is then accumulated with previous results to yield the outcome of the vector dot product. With the activation of the first data flag, the PE's computation register resets. Conversely, upon the activation of the last data flag, the value stored in the PE's computation register is outputted. This value corresponds to the result of a dot product calculation.

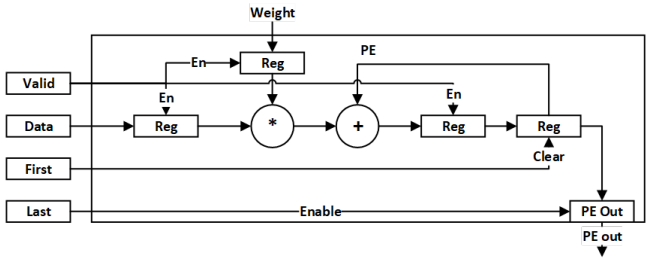


Fig. 9. PE's structure

G. Sigmoid

The Sigmoid function is expressed in (6). This equation comprises exponential and division operations, both of which pose challenges in hardware implementation. An exponential function can be approximated using a Taylor series expansion for calculation, and division operation can be realized in hardware through methods like long division. However, these techniques can incur significant hardware costs. The Sigmoid function exhibits distinctive characteristics, as depicted in Fig .10(a). It features symmetry around the point [0, 0.5], and as the input value moves farther from the origin, it tends toward either one or zero. This allows us to approximate the Sigmoid function as one if the input value surpasses a certain threshold. Owing to its symmetric nature, we can compute the Sigmoid for the positive values of the independent variable. When the independent variable is negative, we can determine its value using the relationship: $sigmoid(x) = 1 - sigmoid(-x)$.

Leveraging the symmetry and convergence properties of the Sigmoid function, we can substantially reduce the computational workload. However, the challenges posed by exponential and division computations still remain. One effective approach is to utilize piecewise linear approximation as an alternative to directly calculating the Sigmoid function. Piecewise linear approximation is a numerical technique that transforms complex non-linear curve computations into simpler linear computations [18]. As illustrated in Fig. 10(b), the Sigmoid curve is divided into numerous small intervals. Within each interval, the Sigmoid curve is approximated bu a straight line that shares the same start and end points as the original curve. This enables us to obtain an approximate Sigmoid value through basic multiplication and addition operations. If the intervals are small enough, the approximation error tends to be zero. The equation for the piecewise Sigmoid can be expressed as:

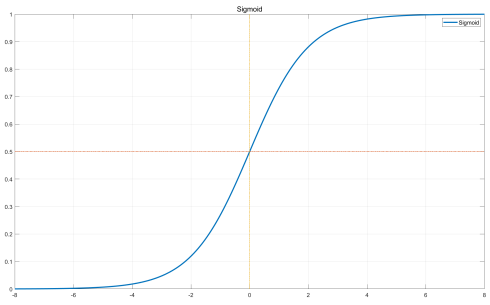
$$sigmoid(x) = \begin{cases} a_i x + b_i, \ x_i \leq x < x_{i+1} \\ 1, \ x > x_{imax} \\ 1 - sigmoid(-x), \ x < 0 \end{cases} \tag{10}$$

Where i represents the i-th interval, and a_i and b_i denote the slope and intercept of the linear approximation within that interval.

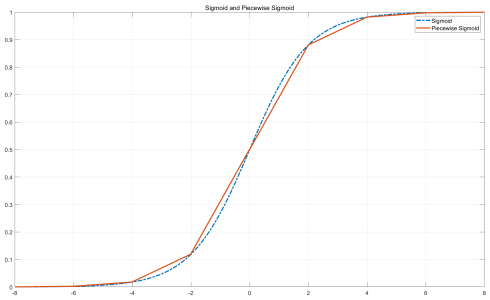
In this project, the Sigmoid function is discretized into 4096 small intervals within the range of $x \in [0, 16]$. The slope and bias values for each interval are precomputed and stored in a look-up table. Since the number of intervals is a power of 2, we can utilize bit shifting to determine the appropriate index for the loop-up table based on the input value.

V. Experiment

This project utilizes the PYNQ-Z2 platform for the implementation and testing of the FPGA RBM. The PYNQ-Z2 is a Xilinx FPGA development board featuring the Xilinx xc7z020-clg400-1 FPGA chip that incorporates two



(a) Sigmoid



(b) Piecewise Sigmoid

Fig. 10. Sigmoid and Piecewise Sigmoid

ARM cores alongside the FPGA. This development board is equipped with essential embedded development resources, including DDR, an SD card slot, a network interface, and more. As as result, it is possible to install a Linux operating system on the board, utilizing its ARM cores as CPU. Xilinx provides Python development packages that facilitate the Linux operating system's interaction with the FPGA resource as if it were a device integrated into the system. Through these Python packages, the bitstream file can be downloaded from Linux to the FPGA, and the interfaces between the ARM cores and the FPGA can be invoked for control and utilization of the FPGA.

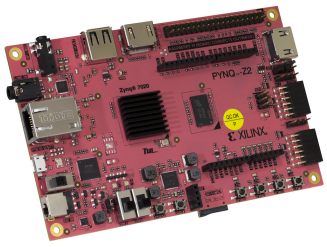


Fig. 11. PYNQ-Z2 board

The RBM model parameters and input data from the ADAS radar sensors are pre-arranged. The FPGA implementation is loaded onto the FPGA section of the PYNQ-Z2 platform, subsequently recognized as a hardware accelerator within the Linux environment. The experimentation involves varying RBM sizes along with their corresponding input data. Python's Numpy package is employed in Linux to execute the RBM algorithm, and the outcomes generated by Numpy serve as the reference results. The experiment entails a comparison between the precision of results, time expenditure, and throughput achieved by the Numpy and FPGA implementations, ultimately yielding the speed-up ratio. The hardware cost of this RBM implementation is depicted in Fig. 12.

Criteria	Guideline	Actual	Status
LUT	70%	34.71%	OK
FD	50%	23.03%	OK
LUTRAM+SRL	25%	34.87%	REVIEW
MUXF7	15%	2.59%	OK
DSP	80%	30.45%	OK
RAMB/FIFO	80%	34.29%	OK
DSP+RAMB+URAM (Avg)	70%	32.37%	OK
BUFGCE* + BUFGCTRL	24	0	OK
DONT_TOUCH (cells/nets)	0	0	OK
MARK_DEBUG (nets)	0	0	OK
Control Sets	998	451	OK
Average Fanout for modules > 100k cells	4	0	OK
Non-FD high fanout nets > 10k loads	0	0	OK
TIMING-6 (No common primary clock between related clocks)	0	0	OK
TIMING-7 (No common node between related clocks)	0	0	OK
TIMING-8 (No common period between related clocks)	0	0	OK
TIMING-14 (LUT on the clock tree)	0	0	OK
TIMING-35 (No common node in paths with the same clock)	0	0	OK
Number of paths above max LUT budgeting (0.575ns)	0	0	OK
Number of paths above max Net budgeting (0.403ns)	0	0	OK

Fig. 12. FPGA hardware resource cost

The accuracy metric is measured using the Mean Absolute Percentage Error (MAPE), and its equation is provided below:

$$error = \frac{\sum_{i=1}^{n} |v_s - v_h|}{\sum_{i=1}^{n} |v_s|} \tag{11}$$

Where v_s and v_h represent the software result and hardware result, respectively. The variable n stands for the length of the vector, and m represents the number of vectors.

Table III and Table IV present the results of the experiments. The vector-matrix multiplication in these experiments was carried out using the numpy.matmul function. All trials were performed using 32,768 input vectors. The term "throughput" denotes the quantity of input data processed per second. With fluctuations in the size of the RBM, the FPGA RBM implementation exhibits relatively consistent throughput. The FPGA implementation showcases a throughput over 143 times greater compared to the Numpy implementation, which employs a highly efficient computational function. The error of the FPGA implementation is low to the magnitude of 10^{-6}, indicating the divided intervals in the piecewise linear approximation are short enough and the data precision setting is suitable. These errors are small enough to avoid affecting the subsequent data reconstruction.

VI. CONCLUSION

This project has a primary objective of addressing the resource and interface limitations inherent in edge devices. It

TABLE III
TEST RESULT

RBM size	Numpy time (s)	FPGA time	error
[128,16]	12.237	0.0856	4.593e-08
[128,32]	12.477	0.0884	1.174e-06
[128,64]	14.330	0.0882	2.040-06
[256,16]	12.430	0.173	2.132e-08
[256,32]	14.482	0.175	1.113e-07
[256,64]	17.310	0.175	3.172e-07
[512,16]	14.872	0.343	2.311e-07
[512,32]	18.028	0.348	2.140e-07
[512,64]	26.196	0.348	2.529e-07

TABLE IV
TEST RESULT ANALYSIS

RBM size	Numpy throughput	FPGA throughput	ratio
[128,16]	0.343M	49.021M	143.021
[128,32]	0.336M	47.453M	141.165
[128,64]	0.293M	47.540M	162.420
[256,16]	0.675M	48.607M	72.0256
[256,32]	0.485M	47.868M	82.637
[256,64]	0.340M	47.973M	98.991
[512,16]	1.128M	48.941M	43.383
[512,32]	0.931M	48.210M	51.804
[512,64]	0.640M	48.206M	75.270

presents a solution that employs the RBM model to reduce data dimensionality and subsequently reconstructs edge computing nodes. To operationalize this solution, the project's main focus lies in devising an FPGA implementation for the RBM model. Ultimately, a comparison is drawn between the software-based RBM and the FPGA-based RBM using the PYNQ-Z2 platform. The results of this comparison demonstrate that the FPGA implementation proposed in this project effectively fulfills its design objective, boasting a remarkably high throughput and achieving an acceleration of up to 143 times.

REFERENCES

[1] Linghe Kong et al. "Edge-Computing-Driven Internet of Things: A Survey". In: *ACM Comput. Surv.* 55.8 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3555308. URL: https://doi.org/10.1145/3555308.

[2] Morghan Hartmann, Umair Sajid Hashmi, and Ali Imran. "Edge computing in smart health care systems: Review, challenges, and research directions". In: *Transactions on Emerging Telecommunications Technologies* 33.3 (2022). e3710 ett.3710, e3710. DOI: https://doi.org/10.1002/ett.3710. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ett.3710. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3710.

[3] Shaeela Ayesha, Muhammad Kashif Hanif, and Ramzan Talib. "Overview and comparative study of dimensionality reduction techniques for high dimensional data". In: *Information Fusion* 59 (2020), pp. 44–58. ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2020.01.005. URL: https://www.sciencedirect.com/science/article/pii/S156625351930377X.

[4] Sebastian Ciobanu. "Vanilla Probabilistic Autoencoder". In: *Proceedings of ISCA 34th International Conference on Computer Applications in Industry and Engineering*. Ed. by Yan Shi et al. Vol. 79. EPiC Series in Computing. EasyChair, 2021, pp. 71–81. DOI: 10.29007/s1mx. URL: https://easychair.org/publications/paper/R2l1.

[5] J. Vrábel, P. Pořízka, and J. Kaiser. "Restricted Boltzmann Machine method for dimensionality reduction of large spectroscopic data". In: *Spectrochimica Acta Part B: Atomic Spectroscopy* 167 (2020), p. 105849. ISSN: 0584-8547. DOI: https://doi.org/10.1016/j.sab.2020.105849. URL: https://www.sciencedirect.com/science/article/pii/S0584854720300410.

[6] Daniel L. Ly and Paul Chow. "A High-Performance FPGA Architecture for Restricted Boltzmann Machines". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '09. Monterey, California, USA: Association for Computing Machinery, 2009, pp. 73–82. ISBN: 9781605584102. DOI: 10.1145/1508128.1508140. URL: https://doi.org/10.1145/1508128.1508140.

[7] Arien P. Sligar. "Machine Learning-Based Radar Perception for Autonomous Vehicles Using Full Physics Simulation". In: *IEEE Access* 8 (2020), pp. 51470–51476. DOI: 10.1109/ACCESS.2020.2977922.

[8] Xinrong Li et al. "Signal Processing for TDM MIMO FMCW Millimeter-Wave Radar Sensors". In: *IEEE Access* 9 (2021), pp. 167959–167971. DOI: 10.1109/ACCESS.2021.3137387.

[9] Geoffrey Hinton. "A practical guide to training restricted Boltzmann machines". In: *Momentum* 9.1 (2010), p. 926.

[10] Nan Zhang et al. "An overview on Restricted Boltzmann Machines". In: *Neurocomputing* 275 (2018), pp. 1186–1199. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.09.065. URL: https://www.sciencedirect.com/science/article/pii/S0925231217315849.

[11] S. Elfwing, E. Uchibe, and K. Doya. "Expected energy-based restricted Boltzmann machine for classification". In: *Neural Networks* 64 (2015). Special Issue on "Deep Learning of Representations", pp. 29–38. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.006. URL: https://www.sciencedirect.com/science/article/pii/S0893608014002160.

[12] Luisa F. Polanía and Rafael I. Plaza. "Compressed sensing ECG using restricted Boltzmann machines". In: *Biomedical Signal Processing and Control* 45 (2018), pp. 237–245. ISSN: 1746-8094. DOI: https://doi.org/10.1016/j.bspc.2018.05.022. URL: https://www.sciencedirect.com/science/article/pii/S1746809418301307.

[13] Nan Wang, Jan Melchior, and Laurenz Wiskott. "An analysis of Gaussian-binary restricted Boltzmann machines for natural images." In: *ESANN*. Citeseer. 2012.

[14] Francisco J. Belmonte et al. “Overview of Embedded Systems to Build Reliable and Safe ADAS and AD Systems”. In: *IEEE Intelligent Transportation Systems Magazine* 13.4 (2021), pp. 239–250. DOI: 10.1109/MITS.2019.2953543.

[15] Welson Sun, Michael J. Wirthlin, and Stephen Neuendorffer. “FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 254–265. DOI: 10.1109/TCAD.2006.887923.

[16] Jieru Zhao et al. “Machine Learning Based Routing Congestion Prediction in FPGA High-Level Synthesis”. In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 1130–1135. DOI: 10.23919/DATE.2019.8714724.

[17] Xuechao Wei et al. “Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC ’17. Austin, TX, USA: Association for Computing Machinery, 2017. ISBN: 9781450349277. DOI: 10.1145/3061639.3062207. URL: https://doi.org/10.1145/3061639.3062207.

[18] Ashkan Hosseinzadeh Namin et al. “Efficient hardware implementation of the hyperbolic tangent sigmoid function”. In: *2009 IEEE International Symposium on Circuits and Systems*. 2009, pp. 2117–2120. DOI: 10.1109/ISCAS.2009.5118213.