technische universität
dortmund

Design and Evaluation of Accelerator Organizations for Binarized Neural Networks

Bachelorarbeit

Somar Iskif

5. September 2022

Supervisors:
Prof. Dr. Jian-Jia Chen
Mikail Yayla, M.Sc.

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
http://ls12-www.cs.tu-dortmund.de

Contents

Chapter 1

INTRODUCTION

The neural network (NN) is a computer model that simulates how the human brain works. In general, in these models, neurons are arranged into layers of nodes, and layers are interconnected through weighted links. Recently, there have been advances in training techniques for deep learning paradigms. Deep learning can learn more complex features so that neural network models can be used for high-dimensional problems. Neural networks are mainly made up of layers, where each layer has several nodes and links. The network is trained by providing specific examples of input and output in the form of inputs mapped to the outputs. This mapping is called "labelling" or "encoding". A feature vector is an output from a single node. The node then sends a weighted input to the next layer of nodes. The next layer of nodes processes the input, and their output is passed on to that layer. It goes on until it reaches the last layer, where the output is estimated as a function of all previous layers of nodes. At this point, the network has been trained such that each node in a layer gives an output with a characteristic or level of precision and quality. The weights of the links in the network are adjusted according to how well the network performs in predicting each output. Neural networks and deep learning were introduced to the business world and have evolved into a new field of services that is rapidly growing. That happens because they can process inputs with large dimensions, thus providing us with amazing results. A neural network can accurately calculate features and predictions that outperform standard statistical machine learning methods. Neural networks can learn everything from face recognition, pattern recognition, and speech recognition. These capabilities are based on their ability to predict outcomes based on interactions of multiple variables. They are also used for pattern classification, clustering, and speech and language processing. Neural networks can detect new patterns that are impossible with traditional algorithms because of their ability to model complex relationships between different variables.

Neural networks are used in all forms of AI. They are used to process language by generating examples and new words, as well as to generate stories. They can also use them to recognize images and can be viewed in the form of machine learning's programming code.

Neural networks are being used to revolutionize the world of medicine by providing better models for analysis and prediction in complex systems. They have already been used to predict cancer, as well as to develop new drugs for cancer. They can also be used to identify details about a patient's medical history, such as how their disease might progress and how long they might live. Medical diagnostic software will soon be able to predict a person's illness based on their medical history and symptoms. We are entering a new era of artificial intelligence, largely thanks to the innovations in the field of neural networks, which are changing the way how we consider the world. These models show us that they can solve many problems with a combination of sophisticated mathematics and deep learning techniques. Deep learning will continue to increase in sophistication and use as it is applied to more complex problems, while neural networks continue to develop new capabilities with every passing year.

1.1 Motivation

Training a neural network is the process of feeding it some data and the correct answer for that data. The neural network then tries to predict the correct answer for future data based on what it has learned from its training. For example, if we train a neural network to recognize objects in images, we might start by training it to recognize dogs. The network will then see thousands of images of dogs and eventually find a way to generalize the dog image we started with in the first place. However, this approach has limitations. If we flip the picture upside down or rotate it 180°, it will assume that these are images of dogs and make a very wrong prediction. We can get around this by re-training the neural network again using a new input data set, but these re-training cycles are expensive in terms of time and resources, such as energy.

The binary neural network (BNN) architecture addresses these issues and enables AI researchers to design and train neural networks at a larger scale. "Research has shown that convolutional neural networks contain significant redundancy, and high classification accuracy can be obtained even when weights and activations are reduced from floating point to binary values [7]." Binarized neural networks are neural networks that use one bit to represent a number, which is zero or one, instead of floating point or integer numbers. This advantage is that it is much more energy efficient and requires much less hardware power. Due to the binary representation, it will be easier and faster to multiply, but on the other hand, the binarized neural network is not 100 % accurate.

Due to their affordable hardware and reasonable accuracy, binary convolution neural networks (BCNNs) have received much attention for embedded applications. We build a crossbar accelerator design to process the binary convolution neural networks (BCNNs). The crossbar-based binary neural network accelerator (CBNNA) is a non-traditional and efficient architecture for a machine-learning algorithm. It uses the computational complex-

ity of matrix multiplies to make efficient use of memory. In crossbar-based binary neural networks, both interconnection topology and channel widths significantly affect computation efficiency. By adjusting the connection weights and channel widths appropriately and by increasing the number of neurons to be small enough compared to the size of their interconnections, it is possible to have an arbitrary number of neurons to be represented in a single crossbar matrix. We will address a significant issue with crossbar-based binary neural network accelerators: how do we divide the workload and map computations to the crossbar array when a crossbar array is too small to hold the weights of one convolution layer at once?
Results from the simulation show that using a different mapping strategy could increase energy efficiency, such that write and read power requirements can be lowered. We specifically tackle two different mapping strategies, strided move (SM) and vertical move (VM), as solutions to this issue for crossbar accelerators and explore the differences between these algorithms according to their efficiency in different hardware demands. We can consider a strided move to be a horizontal movement, moving left and right in our input data set to process it. We can also think of the vertical move as moving up and down into the input data set. Currently, no work investigates the differences between SM and VM for FPGAs (field programmable gate array).

Contributions of this thesis : We examine the impact of the mapping strategies strided move and vertical move on the performance of HW-designs on currently available FPGAs.

1.2 Related Work

I was part of the team that implemented the processing logic for the binary neural network accelerator (BNNA) on the Fachprojekt in the summer semester of 2021 [5]. So In the Fachprojekt, a binary neural network accelerator was implemented on a small scale.
The implementation was done in "VHDL" (very high speed integrated circuit (VHSIC) hardware description language). The modules were first separately implemented and then assembled into a small-scale ”BNNA” [6].
Researchers have widely researched how to build crossbar-based accelerators at different design levels for conventional neural networks (floating-point). Architectures for inference and online training have been proposed [3], [2].

Chapter 2

BASICS

Binarized neural networks were introduced for developing neural networks (NNs). BNN is a neural network with binary activations and weights. Binary weights and activations are utilized to compute parameter gradients at training time. BNN promises to improve power efficiency significantly during the forward pass by considerably reducing memory space and accesses and switching over to bitwise operations for most arithmetic operations.

Each neural network workload can be converted to this notation, where α is the number of rows and β is the number of columns in the weight matrix $W_{(\alpha,\beta)}$, and γ is the number of rows, and δ is the number of columns in the input matrix $I_{(\gamma,\delta)}$ as shown in Fig. 2.1. Larger alpha, beta, gamma, and delta numbers result in large matrices.

$$W_{(\alpha,\beta)} = \overset{\longrightarrow \beta \longrightarrow}{\begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,\beta} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,\beta} \\ \vdots & \vdots & \ddots & \vdots \\ w_{\alpha,1} & w_{\alpha,2} & \cdots & w_{\alpha,\beta} \end{pmatrix}} \Big\downarrow \alpha \qquad I_{(\gamma,\delta)} = \overset{\longrightarrow \delta \longrightarrow}{\begin{pmatrix} i_{1,1} & i_{1,2} & \cdots & i_{1,\delta} \\ i_{2,1} & i_{2,2} & \cdots & i_{2,\delta} \\ \vdots & \vdots & \ddots & \vdots \\ i_{\gamma,1} & i_{\gamma,2} & \cdots & i_{\gamma,\delta} \end{pmatrix}} \Big\downarrow \gamma$$

Figure 2.1: neural network workload Notations

To calculate the scalar multiplication between the weights matrix and the input matrix, the number of columns in the weight matrix must equal the number of rows in the input matrix $\beta = \gamma$. When this equation is true, we can multiply each row of the weights matrix with all columns from the input matrix to get the scalar product. Internally, the BNN represents the -1 in the neural network as a 0 and the 1 as itself and, therefore, as a 1. So, the BNN works with zero instead of -1, which has the advantage that this coding simplifies the multiplication.

2.1 System Model

One processing element is often insufficient or too slow because neural networks have a lot to process. Because of this, there are crossbars that can process the calculations of binary neural networks, where they can calculate several neurons at once. A crossbar looks like this; there are XNORs gates, columns, and interface circuits. Each column processes the workload of one neuron, which is one row in the weights matrix.

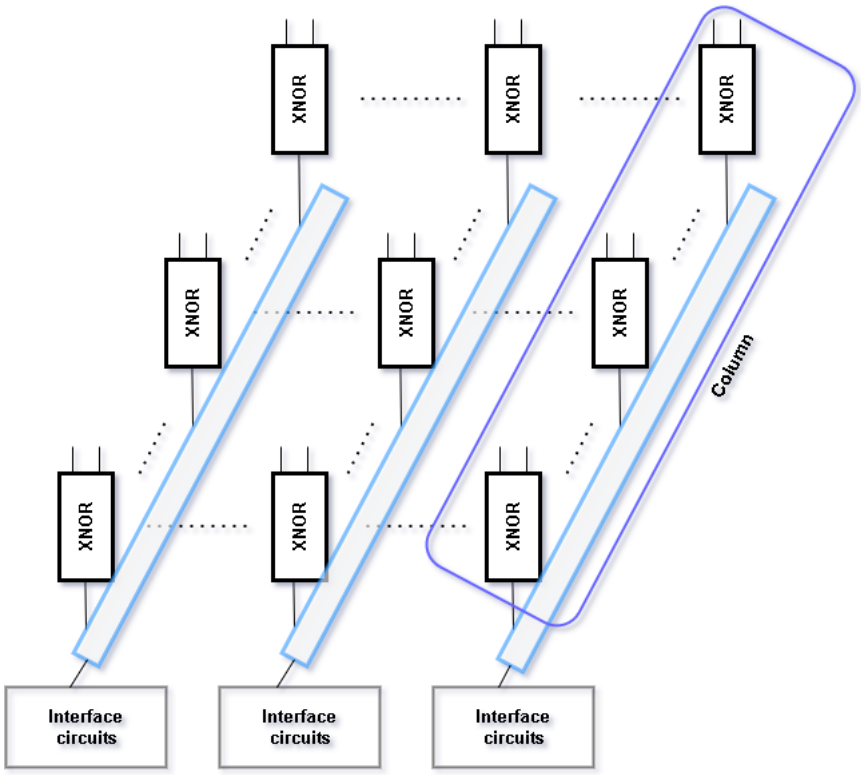

Figure 2.2: crossbar Design

As we can see in Fig. 2.2, the diagonal dots mean that there can be multiple XNOR gates in a column. The horizontal dots indicate that there can be many columns in the crossbar to process the workload of many neurons at once. So in a crossbar, we have m columns and n XNOR gates, and each column is connected to an interface circuit. Therefore, the crossbar can calculate n bits, meaning n bits-vector of the m neurons in each step. An interface circuit consists of the following components popcount, accumulator (adder and register), and comparator.

Fig. 2.2 is the general overview of the design. This is already relatively complex; we cannot just solve it directly; we cannot just start implementing it. We really have to start one step at a time, as the requirements, which parts do we need to implement the design. For one column, we had to calculate XNOR; we had to calculate popcount, comparator, and then the loop. Those are the crossbar parts with the register in between to create the

crossbar accelerator for binarized neural networks step by step.
At first, we started to implement the column and the interface circuit components, which looks like in Fig. 2.3 as one component.

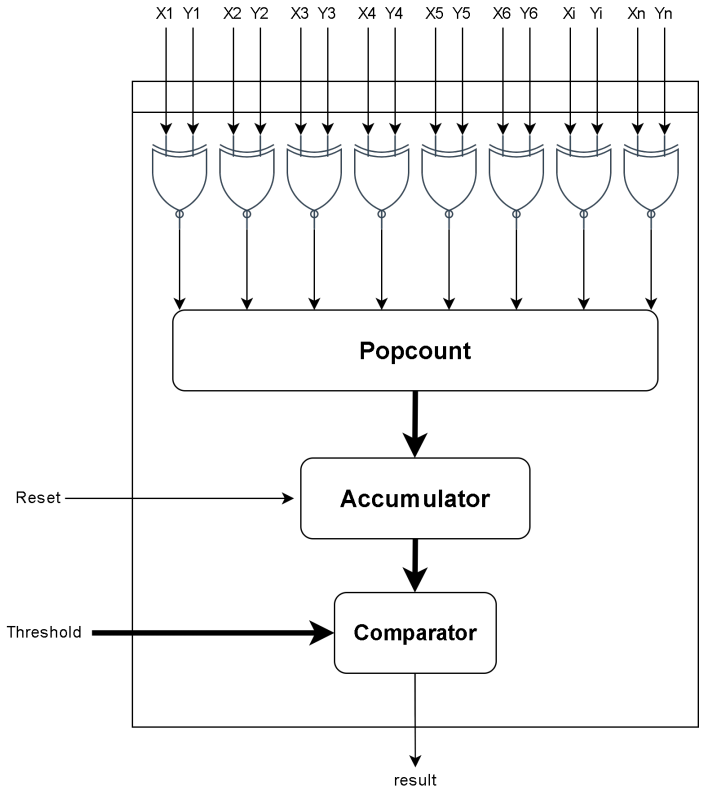

Figure 2.3: column and Interface Circuit

The first step in which we implemented the column is the XNOR gate, the only component it has. We can use an XNOR between two bits instead of the multiplication operation. As you can see below we have an example where to multiply $(-1) * (-1)$ represented by 0's, the equivalent in real values is $(-1) * (-1) = 1$, in the other hand $(0 \oplus 0)$ returns 1. As we can easily realize, the results are correct for all cases.

$$\begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix} \times \begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix} \Leftrightarrow XNOR(\begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

To multiply two vectors, we can connect several XNOR gates in parallel; for example, x XNORs gates can multiply two vectors of size x.

Now the outputs of the XNOR gates have to be calculated. This process is taken care of by the popcount component, as in Fig. 2.4. The popcount component counts the 1's in a bit stream, to be more specific, in the vector, which is the result of the XNOR's gates in each column, and then outputs the result.

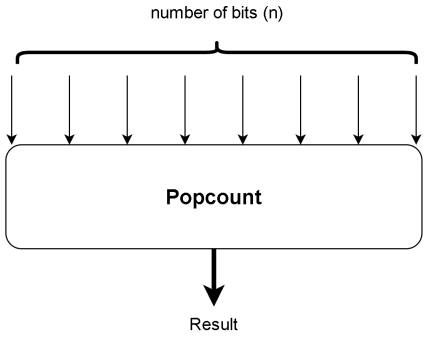

Figure 2.4: Popcount

We realized that since each column can only handle n bits per vector at a time, larger vectors need to be split up, so we needed a register Fig. 2.5 to add the results from the popcount component (partial sums).

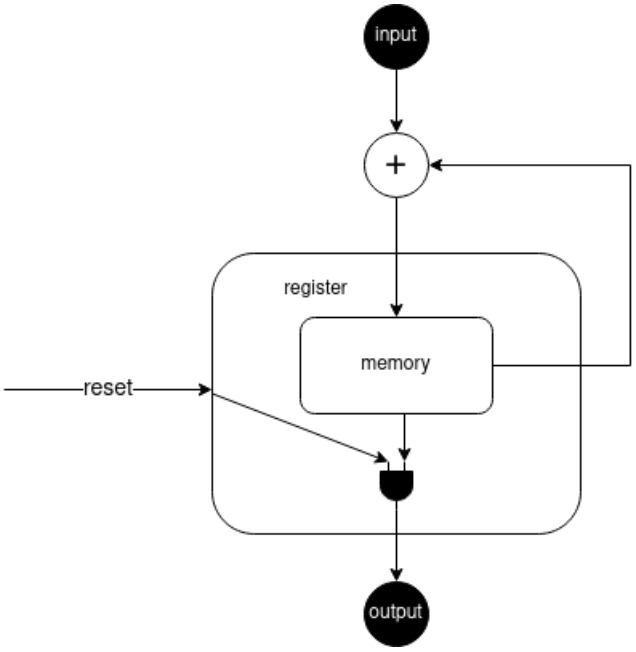

Figure 2.5: Accumulator

The intermediate register sums up the partial results of the popcount module until the crossbar's column processes the whole vector. Then the register outputs the result and resets its memory to zero.

Now to the last component in our processing element, the comparator. As in Fig. 2.6, the comparator compares the threshold with the final output of the register.

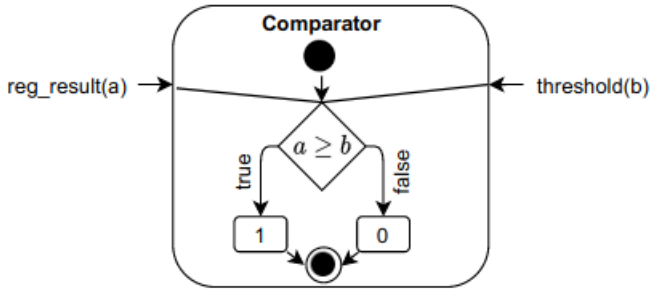

Figure 2.6: Comparator

If the output from the register is greater than or equal to the threshold, then the comparator outputs 1. Otherwise, it outputs 0. We just have the design of the column and interface circuit implemented, and now we can make several instances of them to have the core design of our crossbar. The implementation was done in "VHDL". The modules were first separately implemented and then assembled into a large-scale crossbar binary neural network accelerator (CBNNA).

2.2 Problem

As we mentioned before, if we have larger numbers for alpha, beta, gamma, and delta, it will result in large matrices, which means there is more data to process in neural networks if they get larger. We look at one convolution layer and one crossbar array. The crossbar array size must be at least $(\alpha \times \beta)$ to process the convolution layer in $\mathcal{O}(const)$ time. The workloads or inputs (i.e., the convolution computations) and the weights must be partitioned if the crossbar array size is less than the minimum size specified above [1]. The partitioned workloads and weights must then be mapped to the crossbar. However, how do we divide the workload and map calculations to the crossbar array when a crossbar array is too small to carry the weights of one convolution layer?
Two issues need to be resolved.

- The crossbar array needs to be reprogrammed for various computations after partitioning, even for inference. If we often reprogram the crossbar array, it is quite time-consuming and energy-consuming because programming is done column by column.
- Registers are needed to hold intermediate results (which take up a lot of area overhead).

Therefore, optimizing the mapping process in terms of execution time, power, and register area is important.

According to Fig. 2.7, when the available crossbar size is $(m \times n)$, we describe the issue from the matrix perspective. The input matrix is used to create an unrolled input matrix, a virtual one. It is made up of δ blocks, each of which is $(\alpha \times \beta)$ in size. Each block corresponds to a single convolution window on the initial input matrix. The convolution window is molded into a column, and all the columns in a block are identically replicated, making all the columns in a block the same. On the α output channels, each block generates α points at the same location. If the crossbar array's size equals or exceeds $(\alpha \times \beta)$, we simply compute column-wise inner products between each block and the crossbar array to get the outputs. If not, we must divide the blocks and weights into tiles that are the same size as the available crossbar array. By computing column-wise inner products between the tile and the available crossbar array, the computation of a single tile can be completed in $\mathcal{O}(const)$ time, producing a partial sum of length N. It is necessary that the partial sums generated by the tiles in the same column in one block need to be accumulated.

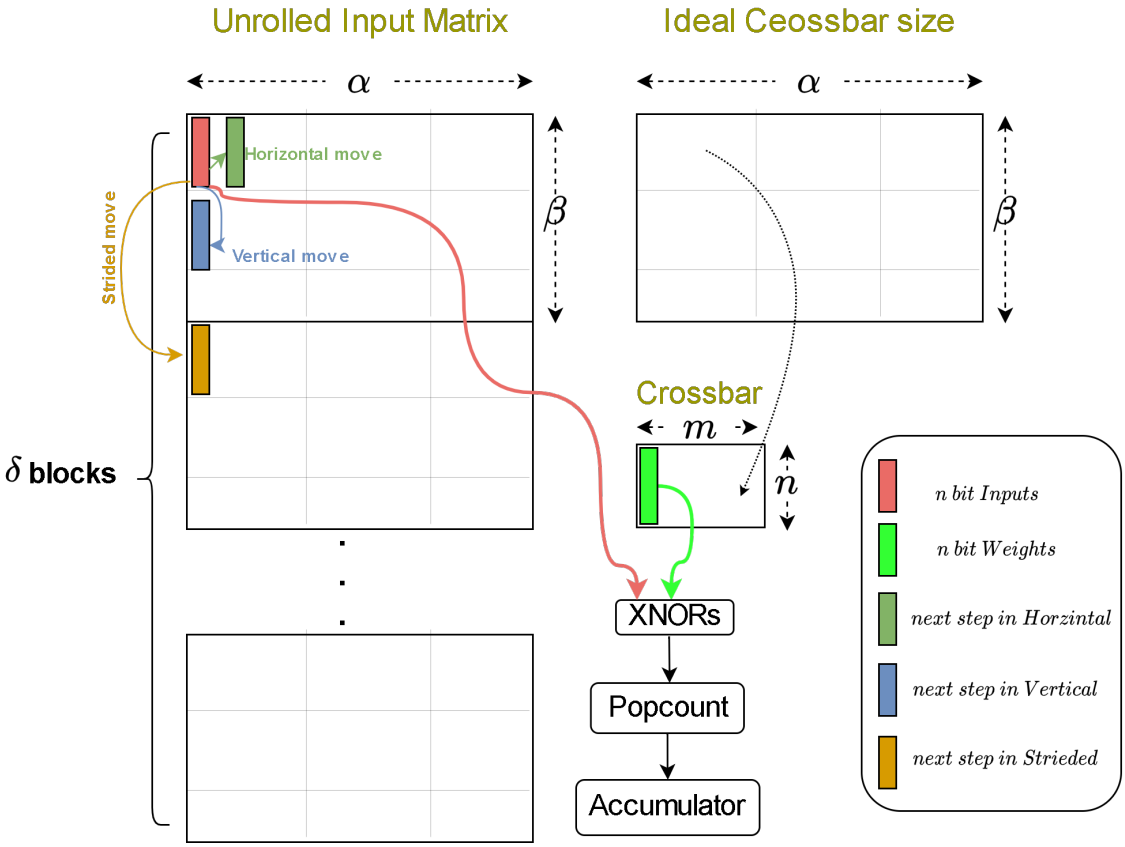

Figure 2.7: Illustration of workload partitioning and mapping [1]

2.3 Solution

The computing order significantly influences execution time, energy use, register area, and the number of reprogramming operations. After one tile in Fig. 2.7 has been computed, we have three options for the following tile: the tile to the right, the tile below, or the tile in the same spot in the following block. The terms “horizontal move”, “vertical move,” and “strided move,” respectively, are used to describe the three approaches. Then, by calculating the needed execution time, energy, and space, we examine the best calculation sequence [1].

In most or almost all cases, β, the number of weights is much larger than n, the number of XNORs gates, leading to the problem that the crossbar array is not large enough to hold the weights of one convolution layer. So, how can we partition the workload and map the Computations to the crossbar array? A solution is to partition the computations and accumulate intermediate results. Intermediate results are also accumulated (partial sums). Most important is that we need to map the workload with proper techniques. As said earlier, many methods exist to partition and map the calculations.

Practically speaking, the issue of insufficient hardware resources always occurs because hardware resources are always restricted, while convolution layer sizes might be arbitrary and huge.

Chapter 3

METHODS

The strided, vertical and horizontal movement methods are different neural network structures that have been used to implement many deep learning and artificial intelligence algorithms. Each method significantly impacts the hardware performance and latency and resource and power consumption.
In horizontal and vertical movements, every move requires reprogramming the crossbar array. However, in the horizontal move, we need a lot more registers to store intermediate results than in the vertical move, so we chose vertical over horizontal. In the strided move, reprogramming is not necessary for the same-place tiles in different blocks like in the horizontal move, this will save a lot of time and energy while programming, and therefore we choose strided move over horizontal. Because of these two reasons, we leave horizontal move out of our study. The intermediate results from the preceding tile must be saved in its exclusive registers whenever we move to the next tile using a strided move, which significantly increases the area overhead.
We will discuss the difference between these two methods, strided and vertical, and compare the performance of the two structures. Learning about strided and vertical move architectures is a good way to start understanding artificial intelligence with deep learning models.

Strides can be considered as a factor by which we are sweeping or stepping through a set of input nodes for a particular layer. This pattern keeps the model from over-examining each node in the network, thereby decreasing its computational cost. Vertical moves can be thought of as a factor by which we are zooming through each node in the network for a particular layer. This pattern is used to reduce the number of nodes that are being examined by a network, thereby increasing its computational efficiency.

3.1 Vertical Move

In the vertical move, as we can see in Fig. 3.1, we take n bits vector of m neurons from the weight matrix with the first suitable part from the input matrix (first red rectangle in both matrices). After the calculation of this part is finished, we move to the next part of the weights matrix, as well as to the part below in the same column of the input matrix (blue rectangle), and so on, until the whole column is calculated. Just then, we start over again from the first part of the weights matrix with the first part of the next column of the input matrix (second red rectangle) until all columns in the input matrix are finished. Now we would have calculated the first m neurons from the weights matrix with all the columns from the input matrix. We can now continue in the same way but with the following m neurons in the weights matrix (purple rectangle) again with the first part of the input matrix (first red rectangle), and so on until all α neurons have been calculated with all δ columns of the input matrix.

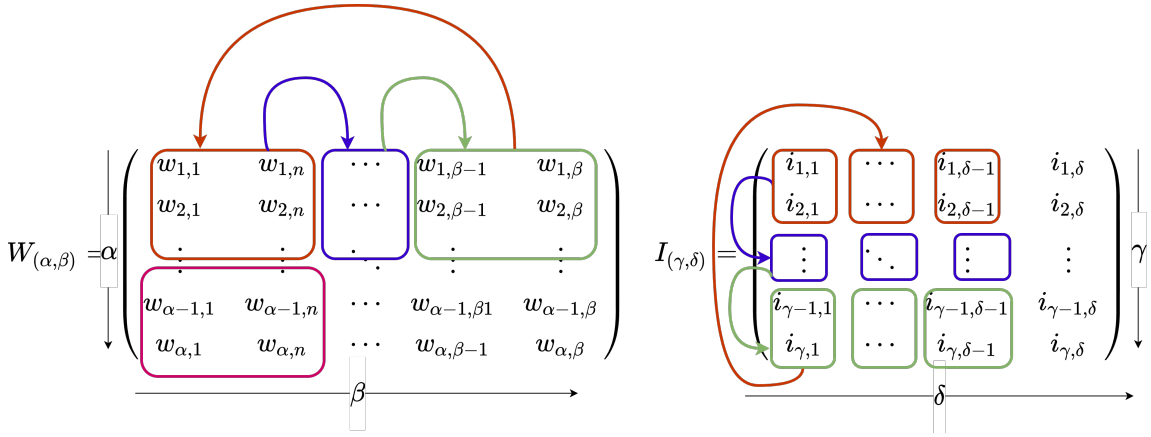

Figure 3.1: vertical move

With this method, the crossbar array has to be reprogrammed with new weights for each move, which has the disadvantage of increasing energy as reprogramming consumes a lot of it and also increases latency and execution time as it waits for the new weights to be loaded on each move. On the other hand, we only have to store the results of the last part with the next results in the same column until it is done, which reduces the register area in this method, and therefore we only need one register per crossbar's column.

Example For Vertical Move

In Fig. 3.2, we can see a small example how the vertical move works, in the example we have $\alpha = 2$, $\beta = \gamma = 6$ and $\delta = 2$. The threshold matrix should have the same rows as α so that each neuron from the weights matrix has its threshold; 2 rows (in decimal).

We use a crossbar array, which is $(Column \times XNORs) = (m \times n) = (2 \times 2)$. That means two columns (left and right). So two neurons at once and two bits vector at a time.

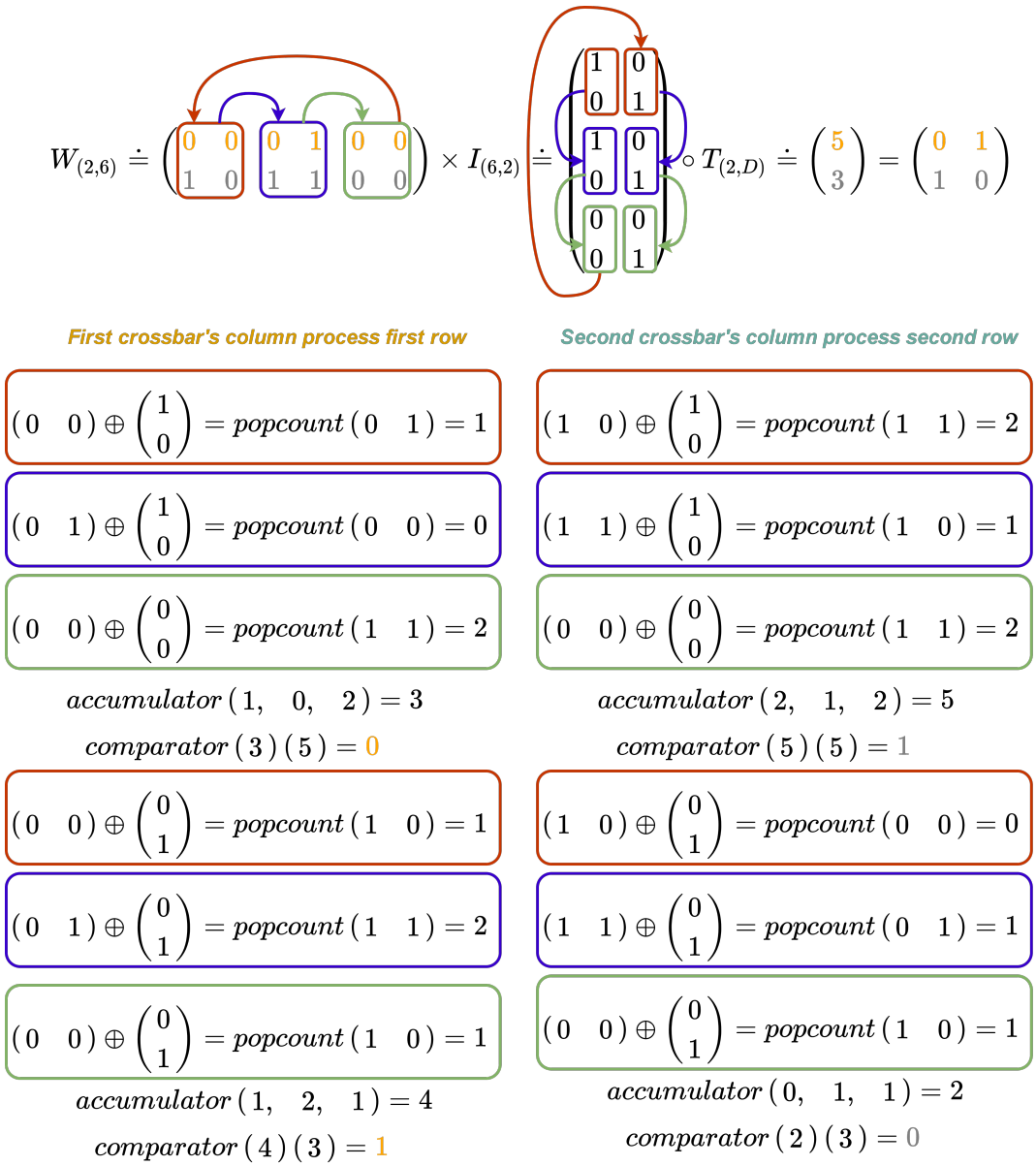

Figure 3.2: vertical move example

As we can notice, in the vertical move, we only need one register for each crossbar's column because we only process a column from the input matrix after the previous one is calculated. Therefore, we only need one register to accumulate the partial sums for that column until it is done. After resetting its value, we use the same register to add the partial sums for the next input's column for the same neuron and, therefore, for the same crossbar's column.

3.2 Strided Move

In strided move as in Fig. 3.3, we take n bits vector of m neurons from the weight matrix with the first appropriate part from the input matrix (first red rectangle in both matrices). After the computation of this part is finished, we move to the part at the same location in the next column in the input matrix (second red rectangle). In contrast, for the weights matrix, we keep reusing the same weights until we are finished with the first parts of all δ columns. Only then can we load the next part of the weights matrix in the crossbar array and calculate them with the second part of the first column of the input matrix (first blue rectangle). This process repeats itself until all *delta* columns are done. We move to the following m neurons with the same strategy.

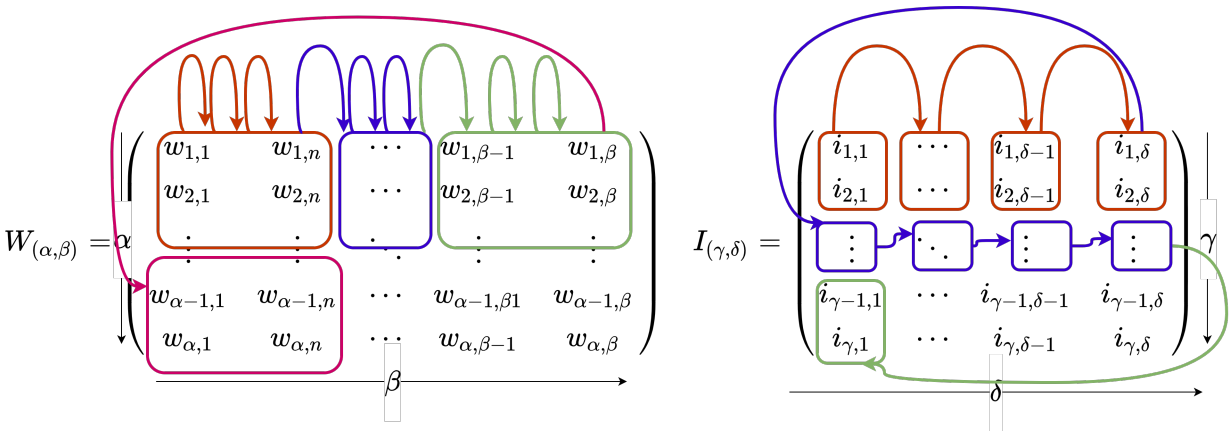

Figure 3.3: strided move

In this way, there is no need to reprogram the crossbar array with every move, which gives us the advantage that the energy is reduced because reprogramming the crossbar with weights consumes much energy. As a result, we can reuse the weights stored in the crossbar array, reducing the latency and making it faster than when we wait to load the weights again in each move. Nevertheless, we need more register area for this because we need to store the results from the previous parts and accumulate them with the next corresponding results. That is the disadvantage of this method, the increasing register area. For strided move, we need δ registers for each column.

Example For Strided Move

In Fig. 3.4, we use the same example as before but for the strided move.
We use the same matrices values and sizes, as well as the same configurations for the crossbar array $(Column \times XNORs) = (m \times n) = (2 \times 2)$.

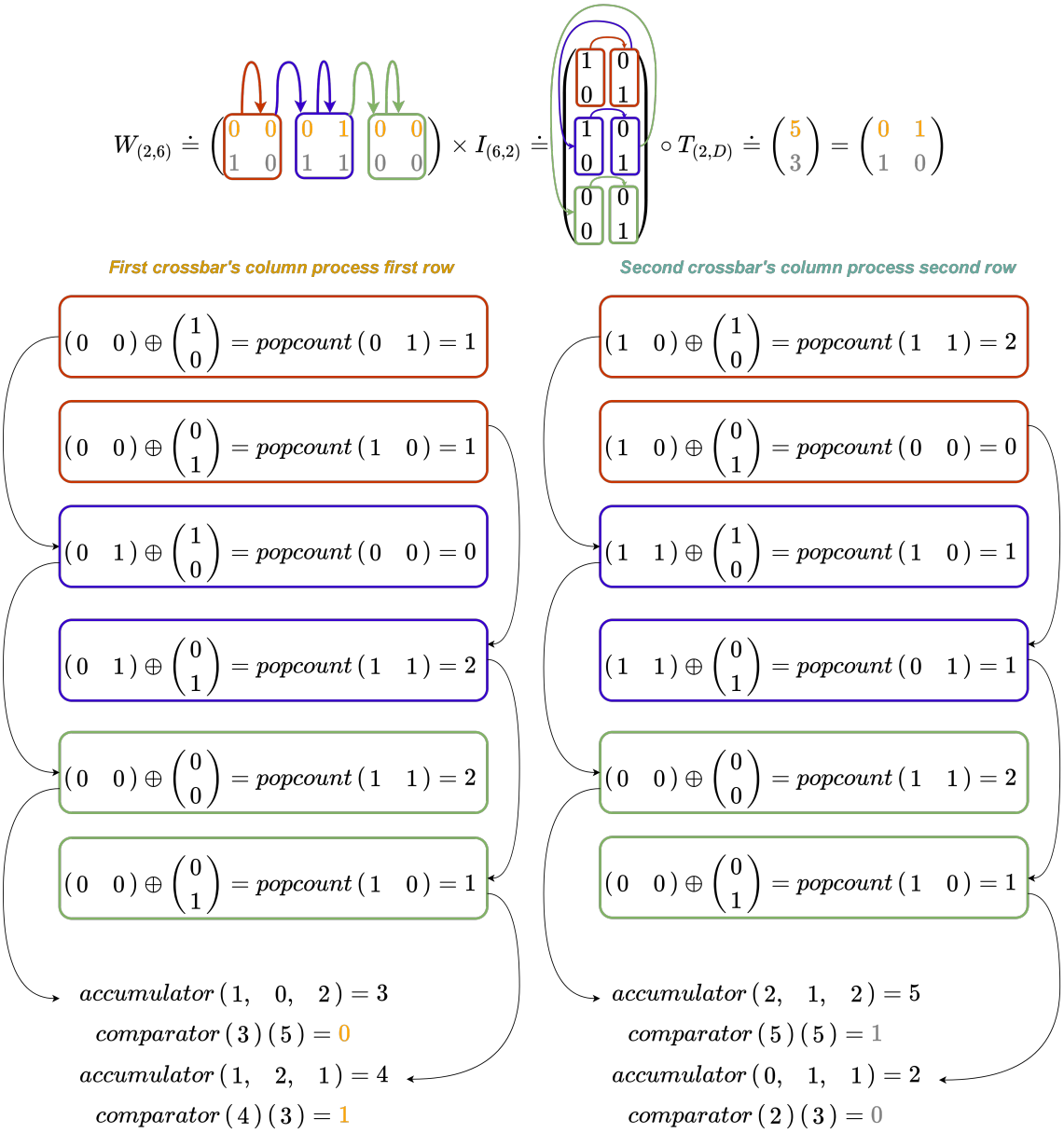

Figure 3.4: strided move example

From Fig. 3.4, we can understand how the strided move differs from the vertical move. In strided move, we need to wait until all input columns are calculated, then, we can start to accumulate the results, and therefore we need δ registers for each crossbar's column to store the partial sums for δ input's column. In this example, we need $\delta = 2$ registers for each crossbar's column.

As we can see, the results in both methods are the same and correct.

3.3 Evaluation Plan

We will present a comparison of two mapping methods for the crossbar binary neural networks. Asymptotic characteristics such as computational complexity, time complexity, and memory complexity are investigated. Some numerical results on these analyzing measures are also presented to illustrate their computing performance.

The aim of this study is for the following methods (1) Vertical move and (2) strided move:

- Measure and compare execution times of different Configurations.
- Compare Latency, energy, and register area ("LUTs" (look-up table) and "FFs" (flip-flops).
- Find out the numerical and analytical consequences of the differences in this comparison for the CBNNs.
- Analyze the performance of these methods, also for modified versions in FPGAs.
- Evaluate benefits and drawbacks of using these strategies for CBNNs.
- Find out what is better for FPGA, vertical or strided move?

Chapter 4

EVALUATION

All results for energy and register area (LUTs and flip-flops) are taken from The VIVADO design Suite [11] after Synthesis [8] and implementation [10] steps, using the (Xilinx ZCU104 evaluation Board) [12], with a clock cycle of $10ns$ and therefore a frequency of $100Hz$ as the constraints [9], without the "ROMs" (read-only memory), which have the weights, input, and the threshold matrices, and without the "BUSs" components (which transfer the data from the ROMs to the crossbar array) because ROMs and BUSs are independent modules which do not belong directly to the binarized neural network accelerator.

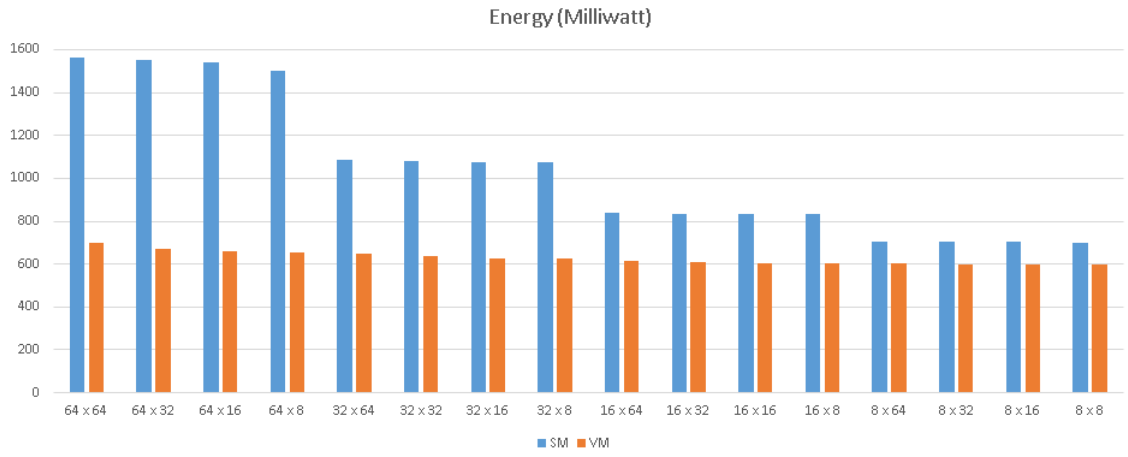

Figure 4.1: energy

In Fig. 4.1, we can see the energy required for the different Configuration in both strided move and vertical move, where for example, (64×32) stands for $(m \times n)$ that the crossbar has $m = 64$ column, and each column has $n = 32$ XNOR gates. We notice that the crossbar consumes more energy when using the strided move than when we use the vertical move. It is right that in the strided move no need to reprogram the crossbar with weights in each move, and that should reduce the energy against the vertical move where with every move, we should reload the crossbar array with new weights. Still, the

reason for this unexpected result is that more registers for strided move (each column needs δ register) means that we need much more energy than in vertical move for all these registers. In contrast, we only need one register per column in the vertical move. The energy needed for these δ registers makes the reduced energy through not reprogramming the crossbar has only a small impact on the total energy. If we look at other configurations of the crossbar, we can also see that the more column in the crossbar we use, the greater the energy we need for the crossbar. The same is for XNORs gates too, but it does not have as much effect on the energy as for the crossbar's columns.

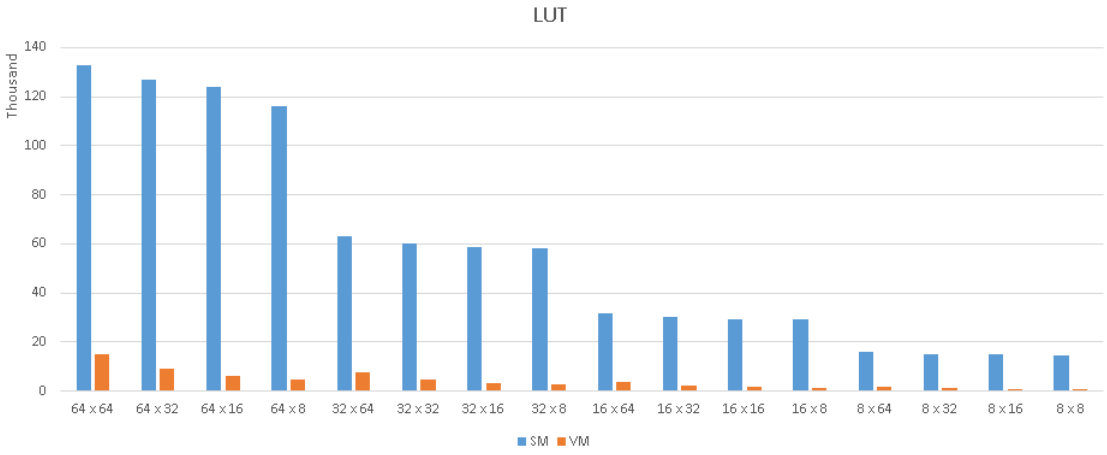

Figure 4.2: register area: LUT

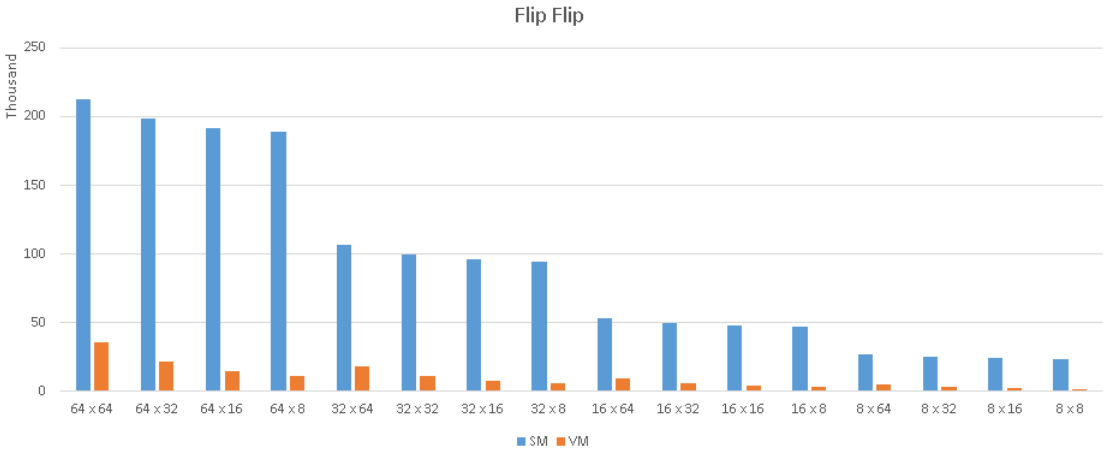

Figure 4.3: register area: flip-flop

From Fig. 4.2 and Fig. 4.3, we can understand how much register area (number of LUTs and flip-flops) the crossbar needs when using the strided move against the vertical move. In the strided move with $(columns \times XNORs) = (64 \times 64)$ it needs about (130×10^3) LUTs and over (200×10^3) FFs, while it is less than (20×10^3) LUTs and less than (50×10^3) FFs in vertical move for the same Configuration. Here it is also similar to the energy; more

column means more register area, and fewer column results in fewer LUTs and FFs. We can also find out that the number of columns used has a bigger impact on the number of LUTs and FFs over the number of XNORs.

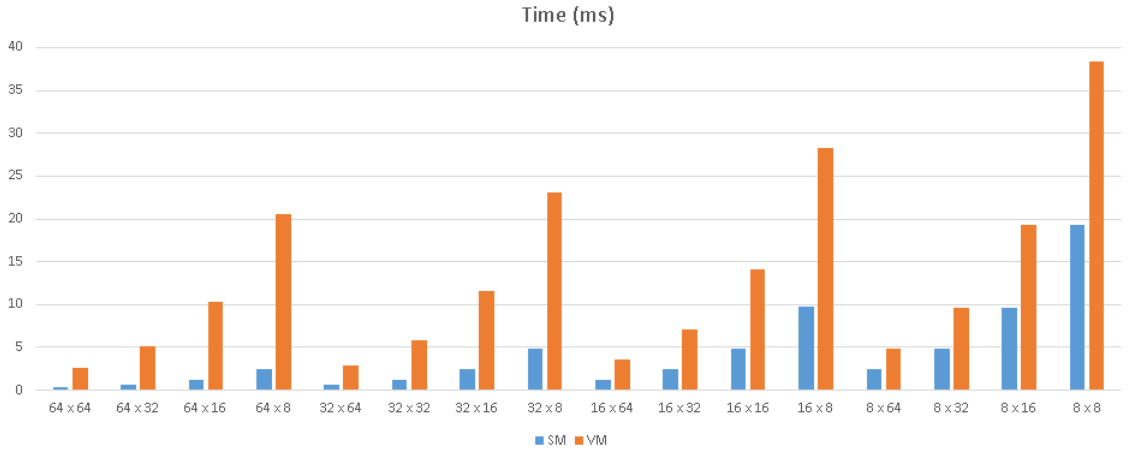

Figure 4.4: execution time

Now let us discuss the execution time for both movements, strided and vertical. As we can see in Fig. 4.4, strided move is much faster than vertical because, in strided move, there is no need to reprogram the crossbar array with every move, which gives us the advantage that we can reuse the weights which are stored in the crossbar array until we are finished with them, it will reduce the latency and make it faster than when we wait to load the weights again in every move like in vertical move. We also can easily observe that the more columns in the crossbar and more XNORs in each column, the faster the crossbar accelerator is, and both the number of columns and number of XNORs used in the crossbar have a massive impact on the execution time in both strided and vertical movements.

Finally, we introduce the impact of the four metrics, execution time, LUTs, FF, and energy, on each other and the dependency between them through the following Pareto Front diagrams. We can see from Fig. 4.5 that in both methods, strided move and vertical move, time and energy are in an inverse proportion; the higher the energy needed for the crossbar for different configurations, the less time needed to finish the calculations, so when we use more column it means more energy, but also means more neurons are getting calculated at the same time, and therefore it is faster. In the vertical move, it seems like the energy values are in a constant interval for the different configurations; it means that in the vertical move, more columns do not have a big impact on the energy but a massive one on the execution time.

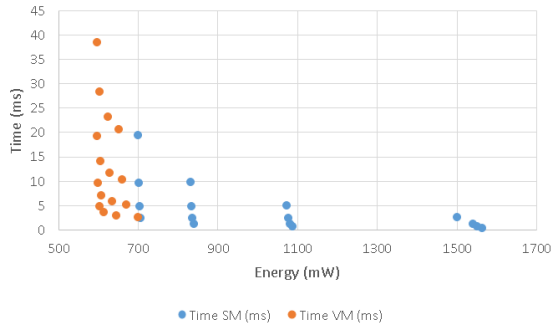

Figure 4.5: energy vs. time

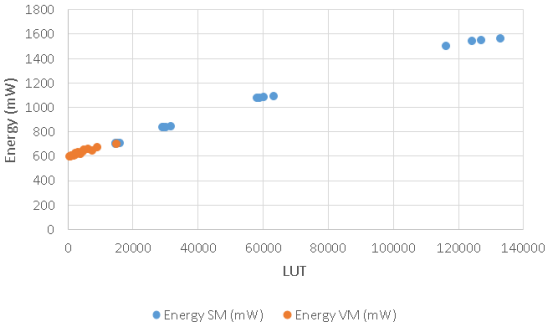

Figure 4.6: energy vs. LUT

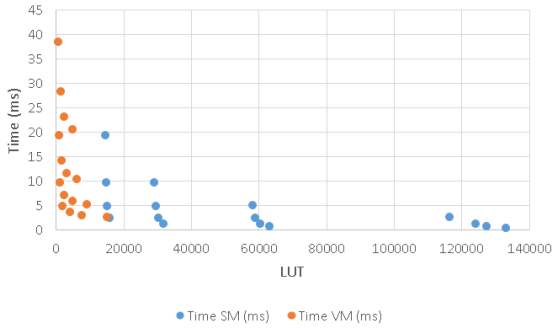

Figure 4.7: time vs. LUT

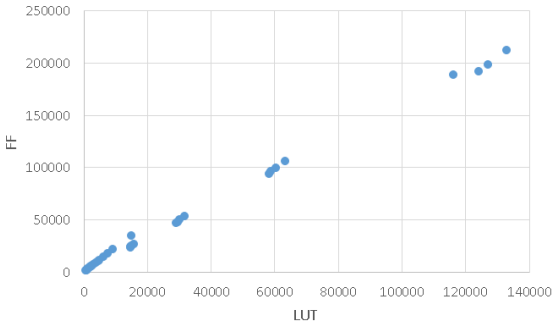

Figure 4.8: FF vs. LUT

From Fig. 4.6, it is easy to understand that in both methods, more LUTs, when we use more columns in the crossbar array, imply more energy and vice versa. The impact of LUTs and energy on each other for each move alone can be seen as a linear function. The discussion for the time versus LUTs is similar to the time versus energy, as we can see in Fig. 4.7. It is obvious from Fig. 4.8 that the LUTs and flip-flops for both strided move and vertical move are in a direct proportion; the higher the LUTs, the more flip-flops there are, and vice versa. It is the same to see it the other way around. The relationship between these two metrics could also be considered as a linear function.

Chapter 5

CONCLUSION

5.1 Summary

A crossbar for binarized neural network accelerator (CBNNA) was implemented throughout this project. We have also considered workload partitioning and mapping. We have offered a general solution based on hardware co-optimization when the crossbar array is not large enough to hold the workload for one convolution layer. This study focuses on designing a binarized crossbar array in two methods, strided move and vertical move, which can achieve low-power or high execution time while maintaining high performance.
The implementation of the crossbar is a difficult task to implement properly, as it requires significant effort, relying on a proper structure for the operation. The implementation was carried out in "VHDL". The code is available on GitHub; please refer to https://github.com/somar0/Crossbar-Design.
The information on the "BNNA" structure was poor, so there was much room for flexibility when implementing the modules. The modules were first separately implemented, and then they were combined into a "CBNNA". We have implemented the Design for both strided move and vertical move for different Configurations $(m \times n) = (column \times XNOR)$ where m and n are in the form of powers of 2^x, the design works for any value for m but for n with max $n = 64$. The implementation was done through the VIVADO tool. The evaluation results are taken from the VIVADO energy, LUT, and flip-flop reports after synthesis and implementation steps for the core design with the controller. The execution time was calculated through the simulation when the signal "finish_all" goes high; it goes high when all results are calculated.

5.2 Results

As a result of our study, we can say that if we are looking for a crossbar binarized neural network accelerator, we need to define the requirements we need for our hardware. Do we need it to be, for example, time-critical, or should it be resource efficient regarding energy or register area?
The choice depends on the demands, hardware requirements, and available resources. We have presented both qualitative analysis and quantitative results and proved that the crossbar, while using the vertical move, can enhance both energy and register area compared with strided move because in strided move, there is high write energy and, of course, more LUTs and flip-flops for the δ registers for each column in the crossbar array (all of them are for the accumulator component), but on the other hand in the strided move, we reuse the weights from the m neurons until we are done from them, because of this, the strided move is considerably faster than the vertical move where in the vertical we need to wait until the new weights are loaded in every single move.
The answer to the question, which is better for FPGA strided or vertical move, is discussible because if the execution time for our hardware is more important than resource usage, then we need to use strided move. If it is the other way around, then we should absolutely go with the vertical move.
In my opinion, if we speak from the FPGA general point of view, we will conclude that because there are many FPGA types and each one of them has its different capacity and available resources, then some of them will not be able to run the strided move because of lack of resources, or it needs more than the maximum energy that it can provide. Therefore, the vertical move is better than the strided move for the FPGA in general.

5.3 Future Work

It would be interesting to optimize the crossbar binary neural network accelerator (CBNNA) to process a full binary neural network and train it, scale the horizontal move on the CBNNA or other calculation principles such as the systolic array, or apply the approximate computing (Stochastic Computing) principle on the crossbar [4], or even to connect several crossbars in parallel to gain faster executing time.

List of Figures

Bibliography

[1] Chen, Xiaoming, Xunzhao Yin, Michael Niemier and Xiaobo Sharon Hu: *Design and optimization of FeFET-based crossbars for binary convolution neural networks.* In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1205–1210, March 2018.

[2] Cheng, Ming, Lixue Xia, Zhenhua Zhu, Yi Cai, Yuan Xie, Yu Wang and Huazhong Yang: *Time: A training-in-memory architecture for memristor-based deep neural networks.* In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[3] Chi, Ping, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang and Yuan Xie: *PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory.* SIGARCH Comput. Archit. News, 44(3):27–39, jun 2016.

[4] Hirtzlin, Tifenn, Bogdan Penkovsky, Marc Bocquet, Jacques-Olivier Klein, Jean-Michel Portal and Damien Querlioz: *Stochastic computing for hardware implementation of binarized neural networks.* IEEE Access, 7:76394–76403, 2019.

[5] Iskif, Woschny, Krekker and Simon: *Binäre Neuronale Netzwerk Beschleuniger.*

[6] Liang, Shuang, Shouyi Yin, Leibo Liu, Wayne Luk and Shaojun Wei: *FP-BNN: Binarized neural network on FPGA*, 2018.

[7] Umuroglu, Yaman, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre and Kees Vissers: *Finn: A framework for fast, scalable binarized neural network inference.* In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 65–74, 2017.

[8] Xilinx, Inc.: *Vivado Design Suite User Guide:Synthesis UG901 (v2012.2)*, July 25, 2012.

[9] Xilinx, Inc.: *Vivado Design Suite User Guide Using Constraints UG903 (v2013.1)*, March 20, 2013.

[10] Xilinx, Inc.: *Vivado Design Suite User Guide Implementation UG904 (v2022.1)*, May 24, 2022.

[11] Xilinx, Inc.: *Vivado Design Suite User Guide Using the Vivado IDE UG893 (v2019.2)*, October 30, 2019.

[12] Xilinx, Inc.: *ZCU104 Evaluation Board User Guide (UG1267)*, October 9, 2018.

Eidesstattliche Versicherung

(Affidavit)

Iskif, Somar

Name, Vorname
(surname, first name)

208875

Matrikelnummer
(student ID number)

☒ Bachelorarbeit
(Bachelor's thesis)

☐ Masterarbeit
(Master's thesis)

Titel
(Title)

Design and Evaluation of Accelerator Organizations for Binarized Neural Networks

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Dortmund, 03.09.2022

Ort, Datum
(place, date)

Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Dortmund, 03.09.2022

Ort, Datum
(place, date)

Unterschrift
(signature)

*Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.