POLITECNICO DI TORINO
Repository ISTITUZIONALE

Logic-in-Memory Computation: Is It Worth it? A Binary Neural Network Case Study

Original
Logic-in-Memory Computation: Is It Worth it? A Binary Neural Network Case Study / Coluccio, A., Vacca, M., Turvani, G.. - In: JOURNAL OF LOW POWER ELECTRONICS AND APPLICATIONS. - ISSN 2079-9268. - 10:1(2020), p. 7. [10.3390/jlpea10010007]

Availability:
This version is available at: 11583/2797402 since: 2020-02-25T13:59:56Z

Publisher:
MDPI

Published
DOI:10.3390/jlpea10010007

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

10 October 2026

Journal of
Low Power Electronics and Applications

MDPI

Article

Logic-in-Memory Computation: Is It Worth it? A Binary Neural Network Case Study

Andrea Coluccio *, Marco Vacca and Giovanna Turvani

Department of electronics and telecommunication (DET), Politecnico di Torino, Corso Castelfidardo 39, 10129 Torino, Italy; marco.vacca@polito.it (M.V.); giovanna.turvani@polito.it (G.T.)

* Correspondence: andrea.coluccio@polito.it

Received: 23 December 2019; Accepted: 4 February 2020; Published: 22 February 2020

Abstract: Recently, the Logic-in-Memory (LiM) concept has been widely studied in the literature. This paradigm represents one of the most efficient ways to solve the limitations of a Von Neumann's architecture: by placing simple logic circuits inside or near a memory element, it is possible to obtain a local computation without the need to fetch data from the main memory. Although this concept introduces a lot of advantages from a theoretical point of view, its implementation could introduce an increasing complexity overhead of the memory itself, leading to a more sophisticated design flow. As a case study, Binary Neural Networks (BNNs) have been chosen. BNNs binarize both weights and inputs, transforming multiply-and-accumulate into a simpler bitwise logical operation while maintaining high accuracy, making them well-suited for a LiM implementation. In this paper, we present two circuits implementing a BNN model in CMOS technology. The first one, called Out-Of-Memory (OOM) architecture, is implemented following a standard Von Neumann structure. The same architecture was redesigned to adapt the critical part of the algorithm for a modified memory, which is also capable of executing logic calculations. By comparing both OOM and LiM architectures we aim to evaluate if Logic-in-Memory paradigm is worth it. The results highlight that LiM architectures have a clear advantage over Von Neumann architectures, allowing a reduction in energy consumption while increasing the overall speed of the circuit.

Keywords: Logic-in-Memory (LiM); Von Neumann's bottleneck; memory-wall

1. Introduction

Nowadays Logic-in-Memory (LiM) architectures are widely studied in order to solve the memory-wall problem, which is a bottleneck due to the communication between processing units and memories. A LiM implementation consists of very small computational units placed near a memory element. This enables a distributed computation instead of a classical Von-Neumann one. The big advantage of this design procedure is the reduction of the Von Neumann bottlenecks (such as fetching latency and the wasted power due to the communication between CPU-Memory), which enables also a very fast and energy-efficient structure. From a theoretical perspective, they bring many advantages, but are they worth it?

To answer to this important question, we have to consider implementing a LiM architecture by modifying the original structure of the memory, creating a structure that merges computation and memory Figure 1. As a natural consequence, the overall complexity of the customized design flow increases. To explore the features of a LiM implementation more in depth, two architectures have been designed, an Out-Of-Memory (OOM) that follows a classical Von Neumann approach and the derived LiM novel alternative. The performance obtained in both cases is subsequently compared.

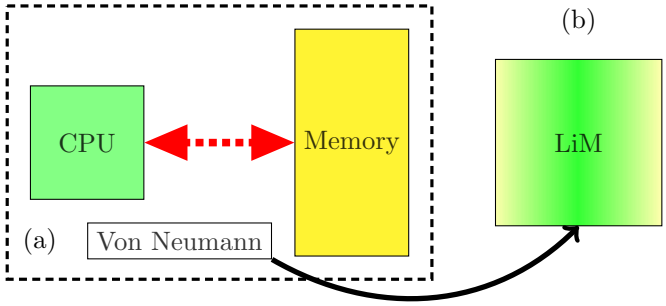

Figure 1. Von Neumann's classical architecture (**a**) composed of CPU and memory. Logic-in-Memory (LiM) novel architecture (**b**) that merges computation and memory.

As a case study, a memory-intensive application, a Neural Network (NN), was chosen, since it is a good candidate to demonstrate the benefits of a LiM architecture. NNs are used to perform very complex tasks such as speech and image recognition in a very efficient and accurate way. Convolutional Neural Network (CNN) and Multi-Layer Perceptron (MLP) models are employed and both can achieve very high accuracy. In literature, many CNNs have been proposed: they can be distinguished by their task, complexity and achieved accuracy. Considering image classification applications, the most common CNNs are LeNet-5[1] and AlexNet [2]. LeNet-5 is a very small network which is able to achieve a TOP-1 error rate of 0.35% on the Modified National Institute of Standards and Technology dataset (MNIST dataset). AlexNet is a more complex structure largely used for recognizing RGB images, which achieves a TOP-5 error rate of 16.4% on the ImageNet dataset. Also GoogLeNet [3], VGG-Net [4] and ResNet-152 [5] can be used on the same dataset and achieve 6.67%, 7.3% and 3.6% TOP-5 error rates respectively. In general, these models require a lot of computational resources implying very high energy consumption, thus making them inoperable in low energy contexts like embedded applications.

In this work, a binarized NN is chosen. Binary Neural Network (BNN) approximations have been proposed in several works like BinaryConnect (BC) [6], Binary-Weight Network (BWN) and XNOR-Net [7], in order to reduce the computational complexity by changing the weight-inputs precision, by means of a binarization process. Weights, and eventually inputs, are approximated with only two values (-1,+1), that can be represented on a single bit, '0' indicates -1 and '1' indicates +1. The chosen approximation for this work is the XNOR-Net [7]. The XNOR-Net reaches high accuracy rates compared to the original floating-point model and is particularly well suited for a LiM solution, since the binary multiplication can be performed by a simple XNOR gate. While a specific Neural Network model was chosen, the architectures were developed with reconfigurability in mind, meaning that most NNs can be implemented by the hardware. Our goal is to demonstrate the effectiveness of a LiM design, so our contributions in this work can be summarized as:

1. Realization of a reconfigurable OOM architecture implementing the XNOR-Net model and proposal of a possible design approach for a LiM alternative.
2. Identification of strong and weak points of a LiM solution with many NN models of different sizes and complexities.
3. Detailed performance evaluations with 45nm @ 1.1V CMOS technology. The estimations are performed with a synthesis and **.vcd**-based post place and route simulations for two models: a CNN and an MLP network respectively. The tools involved in this step are Synopsys Design Compiler for the synthesis, Mentor Modelsim for the simulation and Cadence Innovus for the place and route phase.
4. Generalized performance estimations for both architectures by means of parametric sweeps obtained from several synthesis processes with 45nm @ 1.1V technology. The aim is to compare the implementations with several different parameters and to identify the main differences. Discussion of the obtained results are provided and a qualitative comparison between the implementations are reported in Subsubsection 6.3.1.

5. A state-of-the-art comparison between our LiM and the Content addressable memory based implementation proposed in [8]. In our designs, memories are implemented as register files and each memory cell is a flip flop, since we didn't have the possibility to implement a custom memory. Consequently, the performance values obtained represent an overestimation of a real case. To determine how a real memory model impacts the results obtained, parameters from [8] are taken into account. [8] implements a XNOR-Net LiM design with 65nm CMOS technology, so in our synthesis procedure we used CMOS 65nm technology @ 1.0V to have a fair comparison.
6. Conclusions and discussions for future work.

The rest of the paper is organized as follows: Section 2 gives a brief explanation on what a LiM architecture is and recalls a useful classification from [9]. Section 3 discusses briefly NN background, giving an overview on what its main components are; binary approximations are compared and explained in more details. Section 4 reports the detailed design flow adopted for both OOM and LiM architectures and makes an initial qualitative comparison between them. In Section 6, performance evaluations are reported, firstly taking two NN models as a case study and then by performing parametric sweeps. Lastly, Section 7 presents conclusions and future work.

2. LiM Background

A Quick Overview

LiM concept is widely discussed in the literature and a lot of different approaches have been adopted. In [9] an interesting classification of the various types of LiM paradigms is presented. Four possible typologies can be found.

1. Computation near Memory [9] where part of the computing blocks are moved in the memory proximity proposing solutions such as WIDE-IO2, which is a 3D stacked DRAM memory [10] with a logical layer placed at the bottom of the stack. Data are moved from the DRAM layers to the logical one employing Through-Silicon Vias (TSVs) and the result is then written back to one of the available DRAM layers. 3D stacked DRAM combined with TSVs allow to shorten the paths' lengths that data have to travel to reach the computational core, reducing the Von Neumann bottlenecks and improving efficiency.
2. Computation in Memory [9] paradigm is used in solutions with resistive arrays, based on technologies such as Magnetic Tunnel Junction (MTJ) devices [11]. MTJ is a component that can have two discrete resistance values, according to the direction of the magnetizations of its ferromagnets: if they are parallel, the MTJ is in low resistance state (R_P) meaning more current flowing through it, otherwise, they are in antiparallel configuration (R_{AP}) with highest resistance. These resistance states can be mapped in a logic fashion as logic '0' if they are antiparallel, logic '1' otherwise. By arranging multiple MTJs in a matrix configuration, both memory and logic operations can be performed analogically. Several works use MTJ devices. In [12], Generative Adversarial Network (GAN) implementation has been proposed. This Neural Network consists of a discriminator (D), that works as a detective in the training process, and a generator (G) as a deceiver in a semi-supervised fashion. In these networks, training is a critical issue so hardware accelerators are demanded. [12] improves the so-called adversarial training process by using an array made of MTJs which simplifies the calculation of multiply-accumulate operations with ternary weights ($W \in \{-1, 0, 1\}$), transforming them into bulk In-Memory additions and subtractions. This work achieves remarkable results in term of efficiency and processing speed with respect to GPUs and ASICs. In [13], authors have developed a MTJ-based convolution accelerator in which the memory array is capable of performing bulk AND operations. They have included a small external logic which is in charge of computing the accumulations. Based on a similar working principle, Resistive Random-Access-Memory (RRAM) [14] are devices in which the logic data is encoded in two or multiple resistive states. Differently from MTJs,

resistance is determined by the conductivity of a conduction path that can be broken (high resistance state) or reformed (low resistance state). Sometimes it is used in a 1 transistor 1 RRAM (1T1R) configuration, to avoid unwanted or sneak current paths. In [15], authors have presented a memristor-based implementation of a BNN able to achieve both high accuracy on MNIST and IRIS dataset and low power consumption. In some others, improvements in memristor architectures have been proposed that enable multiple bits per cell. [16] has exploited the frequency dependence of GeSeSn-W memristor devices to obtain multiple conductance values representing different weights. In [17], the memory array has been modified, including up to 4 memristors arranged in parallel in the same cell, in order to have multiple resistance values and so higher precision weights. Based on a similar approach to [12], a GAN training accelerator has been discussed in [18] which is able to efficiently perform approximated add/sub operations in a memristor array, achieving both speed-up and high energy efficiency.

3. Computation with Memory [9] concept consists of memory arrays that intrinsically perform calculations. Possible examples can be Content Addressable Memories (CAM) and Look-up tables.
4. Logic-in-Memory [9] is the concept that we are analyzing in this work, in which small computational units are placed inside or near a memory cell, to perform distributed computation.

As can be deducted, LiM is a widely studied and heterogeneous topic, and it is becoming increasingly important over the years. A lot of works presented in literature implement an application-specific LiM solution. The discussed emerging technologies are very promising, especially in Neural Networks applications, because of their high efficiency to compute multiply-accumulate operations [16]. In our work, we concentrated on CMOS technology because, while it is not the best available, RRAM and MTJ devices are still under development. As future task, we will focus our attention on them once these circuits are optimized.

3. Neural Networks: An Introduction

3.1. Neuron's Model

A NN is a computational model that is able to perform very complex tasks. It is composed of "neurons", which are the basic building blocks. By organizing them in an interconnected network, the NN can take decisions and learn when these decisions are wrong [19].

In Figure 2 a neuron structure example is depicted. As it is possible to see, it is made of two main parts which are *net*, which is in charge of weighted sum computation, and $f(net)$, which is an activation function applied to the neuron's output. In general, *net* expression can be written as:

$$net = \sum_{i=0}^{N} X_i \times W_i + Bias \tag{1}$$

where X_i is the input value, W_i is the corresponding weight and *Bias* is an additive term. Neurons' weights and biases can be adjusted to achieve the desired output with a procedure called training.

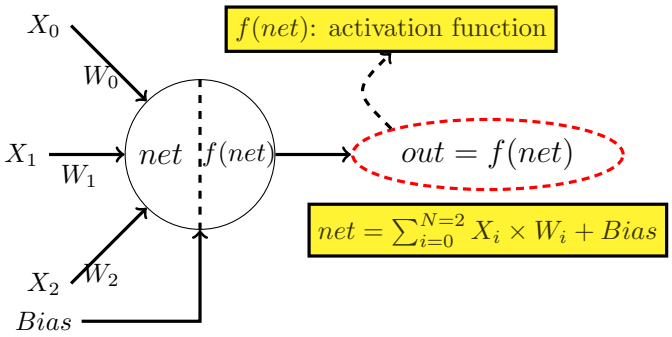

Figure 2. Schematic of a neuron, representing its structure. Three inputs example [19].

In Figure 2, it is indicated another part which is the activation function $f(net)$. Usually, this is a nonlinear function. The most important activation functions are Rectified Linear Unit (ReLU), hyperbolic tangent (tanh) and sigmoid function, which are discussed in great details in [20].

3.2. Neural Network's Structure

Usually, NNs are made up by layers, which are composed of a set of arranged neurons. The most common structures are Convolutional Neural Network and Multi-Layer Perceptron.

In Figure 3 it is reported the LeNet5 CNN as example. The network is composed of 2 convolutional, 2 pooling and 3 fully connected (FC) layers. Each of them is discussed in detail:

- Convolutional layers perform the convolution operation of the input feature map (IFMAP) with a set of weights called kernel. An example of a convolution computation is depicted in Figure 4. The parameter taken into account are the kernel's weights, the input feature map and the stride. After the first convolution is finished, the kernel window is moved by a step equal to stride, and a new convolution can start. In this example, the convolution computation match perfectly the neuron's equation reported in Equation (1), in fact after a convolutional layer is usually used an activation function to normalize the results. In the LeNet 5 CNN [1] example in Figure 3, all the convolutional layers have the same 5×5 kernel sizes. The first one produces six output feature maps (OFMAPs), meaning that the same IFMAP has been convolved with six different kernels. The second convolutional layer instead produces 16 OFMAPs, starting from 6 IFMAPs: for each input, there are 16 kernels that produce 16 outputs, so 16 from the first IFMAP, 16 for the second IFMAP and so on. This implies a total number of OFMAPs equals to

$$\#OFMAPs = 6 \times 16 \tag{2}$$

  To obtain 16 OFMAPs indicated by LeNet 5 scheme, the obtained OFMAPs of each layer are added together.

  These considerations bring to the following formula for a convolutional layer, derived from [21]:

$$y_o(j,i) = Bias_o + \sum_{c_{in}=0}^{\#C_{in}-1} \sum_{k=0}^{W_y-1} \sum_{p=0}^{W_x-1} k_{o,c_{in}}(k,p) \times X_{o,c_{in}}(j \times stride + k, i \times stride + p) \tag{3}$$

  where i, j are the indexes for the OFMAP corresponding pixel, c_{in} is the input channel index, $\#C_{in}$ the total number of input channels, W_x, W_y are the kernel's matrix size indicating number of rows and columns respectively, o subscript refers to the OFMAP considered and p, k are the kernel's indexes.
- Pooling layers have a similar behavior to convolutional layers. In the literature, different kind of poolings are used such as average or max pooling [22]. They perform the maximum (or the average) of the selected input pixels and returns only one value, performing the so-called

subsampling operation. Pooling, and more specifically max pooling, is widely used to reduce the size and the complexity of the CNN. In Figure 3, the kernel size is 2×2 for all the cases.

- FC layers are MLP subnetworks included in the CNN to perform the classification operation. They are made of layers of fully interconnected neurons, as shown in Figure 3.

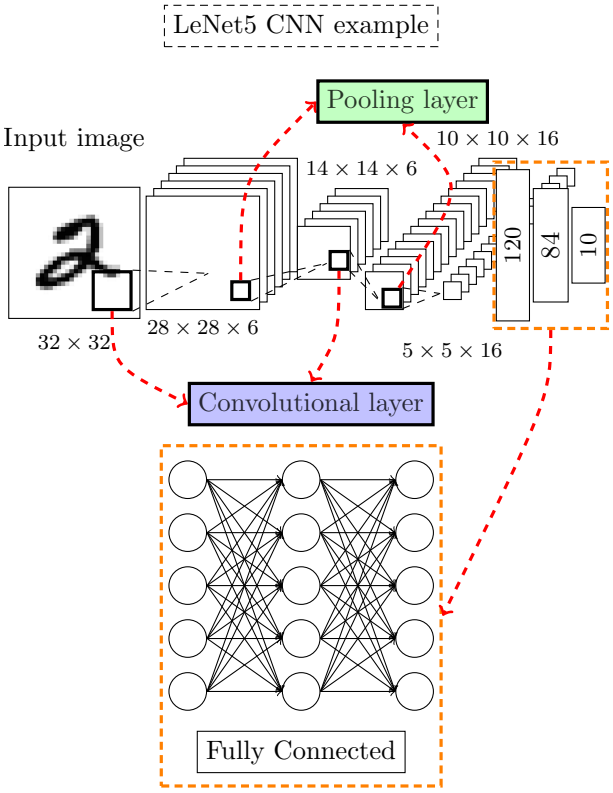

Figure 3. Structure of LeNet 5 Convolutional Neural Network (CNN) [1], composed of 2 convolutional, 2 pooling and 3 fully connected layers and their sizes are indicated in the model.

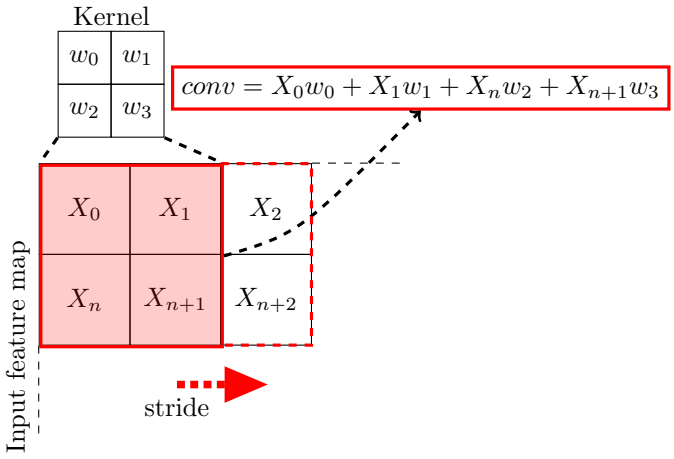

Figure 4. Convolution computation example with a 2×2 kernel.

There are also normalization layers (not reported in Figure 4). One of the most used is the Batch Normalization (BatchNorm) [23] that is very useful in BNNs to recover a portion of the accuracy lost from the binarization [24]. BatchNorm equation is reported from [23]:

$$\tilde{X} = \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}} \times \gamma + \beta \tag{4}$$

where μ, σ are the batch mean and variance, while γ, β are correction values. These four variables are trainable, meaning that during training procedure they are modified in order to increase the accuracy. ϵ is usually added to the variance to avoid 0 division if the variance is 0. ϵ is a very small number, so the following approximation for non-zero variance can be made:

$$\tilde{X} \approx \frac{X-\mu}{\sigma} \times \gamma + \beta = X \times \frac{\gamma}{\sigma} + \left(-\frac{\mu \times \gamma}{\sigma} + \beta \right) = X \times A + B \tag{5}$$

3.3. Binary Approximation

Since NN are very complex models, they can be very power hungry and implementing them on low energy budget systems, like in embedded application, can be challenging [25]. For this reason, a BNN approximation is chosen, trying to reach a good trade-off between complexity and accuracy. In [7] is presented an interesting comparison between some BNN approximations, introducing also XNOR-Net. The values are recalled in Figure 5. In the plot, TOP5 is intended as the accuracy classification rate to hit one out of five most probable classes. In the plot, TOP5 is intended as the accuracy rate to hit 1 out of 5 most probable classes. The BNNs accuracy are compared with the original floating-point implementation (FP) of AlexNet neural network [2].

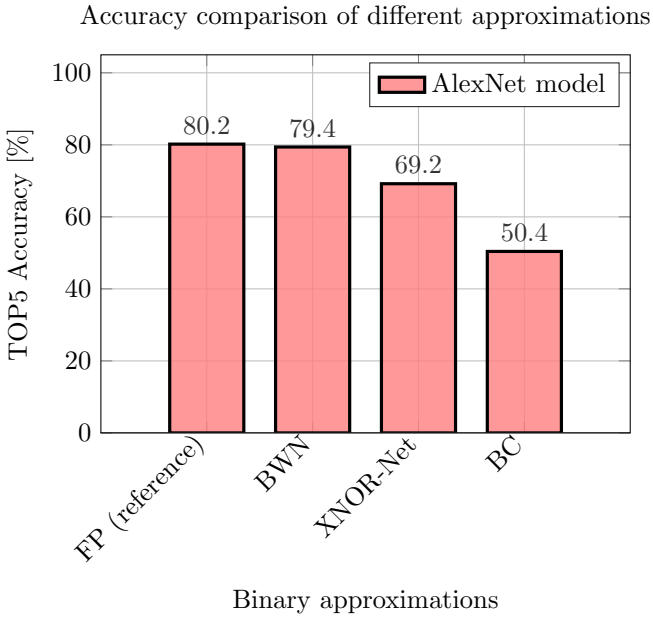

Figure 5. TOP5 accuracy comparison between different binary approximations [7].

In the considered approximation, all the weights are in binary format, meaning that $w \approx w_b \in \{-1, 1\}$ where w_b is the binary weight value. The binarization techniques are now briefly summarized from [7].

- BWN [7] binarizes only weights of the NN, keeping at full precision the activations and the inputs. By binarizing only weights, the convolution operation can be performed only with adds and subtractions, avoiding multiplication as reported in Equation (6) [7].

$$Conv_{out,BWN} = X * w + Bias \approx \alpha(X * w_b) + Bias \tag{6}$$

  An extra factor α is multiplied to the convolution result, in order to compensate precision losses [7]:

$$\alpha = \frac{\sum_{i=0}^{N} \|w_i\|}{N} \tag{7}$$

where w_i is the considered full precision weight and N is the number of weights. BWN is a very good alternative useful to reduce CNN's complexity. However it requires full precision inputs and activations.

- XNOR-Net [7] binarizes both weights and inputs. The convolution result is obtained by performing the binary convolution and multiplying by a correction factor α (the same in Equation (7)) and a matrix $\mathbf{K}$. $\mathbf{K}$ is defined in Equation (8).

$$\mathbf{K} = \overbrace{\frac{\sum_{c_{in}=0}^{\#C_{in}-1} |X(:,:,c_{in})|}{\#C_{in}}}^{\text{First term}} * \overbrace{\begin{bmatrix} \frac{1}{W_x \times W_y} & \frac{1}{W_x \times W_y} & \cdots \\ \frac{1}{W_x \times W_y} & \frac{1}{W_x \times W_y} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}}^{\text{Second term}} \tag{8}$$

In Equation (8), the first term indicates the absolute punctual sum of the multiple IFMAPs divided by the number of input channels, thus the number of IFMAPs. The second term is a regular matrix of $W_x \times W_y$ size, which contains $\frac{1}{W_x W_y}$ in all positions. Finally, the XNOR-Net convolution can be rewritten as [7]:

$$Conv_{out,XNOR-Net} \approx (X_b \circledast w_b) \cdot \mathbf{K} \times \alpha \tag{9}$$

where X_b is the binarized input, $\circledast$ is the binary convolution, $\cdot$ is punctual multiplication and $\times$ is a simple product. In [7] the binary convolution is performed considering the XNOR pop-counting of binary inputs/weights. XNOR truth table matches to the multiplication if -1 is mapped to logic '0' and +1 is logic '1'. Pop-counting computes the difference between the number of 1s and the number of 0s of the input sample.

- BC [6] binarizes both inputs and weights, without applying any correction factor to the final convolutional equation. This implies less recognition accuracy as shown in Figure 5. Taking into account all the considerations on the binarization techniques, we chose XNOR-Net [7] as reference model since it represents a very good trade-off between accuracy and complexity.

3.4. NN Implementations Based on LiM Concept

The LiM approach is often applied in NNs' implementations. Some of them are considering the binary approximations, choosing an implementation based on emerging technologies. Some works [12,13,26,27] are based on MTJ technology while [15–18,28,29] have used RRAM. In each of these works the resistive element is used to perform simple logical operations based on current sensing technique. In [26,27,30,31] several Binary Convolutional Neural Networks (BCNNs) implementations are discussed: they achieve very good results in terms of energy and power, thanks to the intrinsic low power nature of the MTJ and RRAM devices. [28] proposes a BNN design based on SRAM array. The logic parts perform the computations and are disposed below the memory array. The memory parts enable to store the required parameters for the NN computation (like weights and biases) and the logic parts compute the results for the next layer, forming an alternation between memory-logic. This architecture achieves very good performance in terms of energy and speed, thanks to its pipelined-like structure. In [29], the NN has been mapped in a Wide-IO2 DRAM, using TSVs as high speed communication link obtaining remarkable results in terms of execution time.

4. OOM and LiM Architectures

Here we discuss the adopted processing flow more in detail. The goal of a LiM architecture is to move part of the computation inside a memory array, which already contains the needed logical elements to complete the calculations. Using as a case study the XNOR-Net, we can derive that the

main part of the algorithm is the calculation of the XNOR products combined with pop-counting to determine the result of the binary convolution. The adopted design flow is the following:

- Design of a classical architecture, called OOM, capable of implementing the XNOR-Net model;
- Derive a LiM alternative, defining what are the building blocks inside a memory cell;
- Qualitative comparison of the architectures, describing the advantages and disadvantages of both of them;
- Performance estimation and comparison by means of synthesis and place and route procedures of two NN models;
- Performance estimation of different NN models.

4.1. OOM Architecture Design

4.1.1. Single Input/Multiple Output Channels Design

By looking at Equation (9) and Figure 6, the core part of the OOM architecture is composed of a set of XNOR gates and a pop-counter. In Figure 6, a tiny example of a 2×2 convolution is reported: since the dimension of the kernel is 4, the total number of XNOR gates required are 4, because each of them performs a multiplication. In general, their number must be at least equal to $W_x \times W_y$. NNs' kernel sizes depend on the model chosen, for example in AlexNet the maximum kernel size is 11×11 [2] or in LeNet5 is 5×5 [1]. The flexibility of the hardware circuit depends strictly on how big the kernel size is considered, so a worst-case analysis must be taken into account. To best of our knowledge, kernel sizes higher than 11×11 are very seldom, since accuracy usually decreases with bigger filter sizes. Binarized inputs/weights are fed directly to XNOR inputs from a memory implemented as a register file (RF) in the design, named Binary Input RF in Figure 7. In Binary Input RF, each row contains all the input elements required for a convolutional window computation, implying a bitwidth size equals to $W_x \times W_y$ bits. Regarding the number of rows, they have to be at least equal to the total number of convolutional windows D_{out}^2 required, which is also the dimension of the OFMAP. D_{out} can be computed considering kernel, IFMAP sizes (D_{in}) and the stride.

$$D_{out} = \frac{D_{in} - W_x}{stride} + 1 \tag{10}$$

Also in this case, the number of memory rows D_{out}^2 has to consider the maximum OFMAP size of the NN model considered. When a different OFMAP has to be computed, the weight set is simply switched by using a multiplexer. The total number of multiplexer is equal to the maximum number of OFMAPs, called number of output channels ($\#C_{out}$) of the NN. In Equation (10), it is indicated only W_x, since usually the kernels are regular matrices with $W_x = W_y$.

Regarding the pop-counting computation, handling many parallel inputs requires too many hardware resources. For this reason, the outputs of the XNOR gates are multiplexed and only one of them is processed per clock cycle. A pop-counter can be simply implemented with an adder, a NOT gate and a register as shown in Figure 8. Together with the pop-counting circuit, the main computational part has been called XNOR-Pop Unit, as shown in Figure 7.

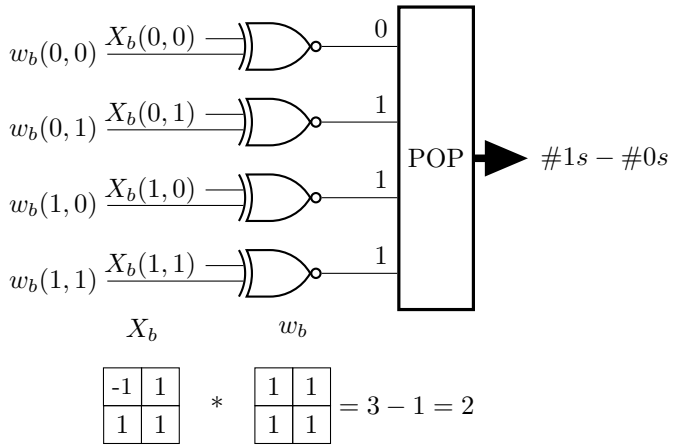

Figure 6. Example of Binary convolution based on XNOR-Pop procedure.

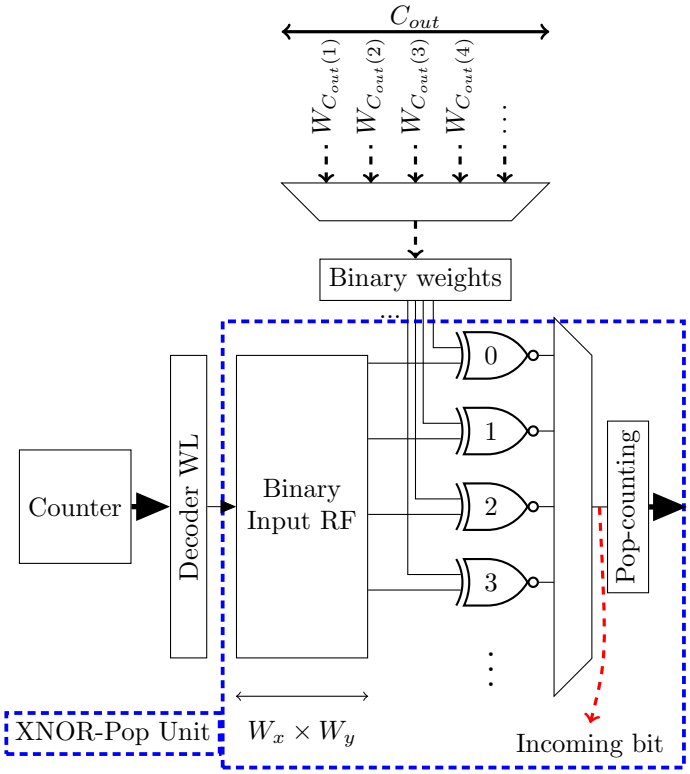

Figure 7. Out-Of-Memory (OOM) main computational part. Each Binary Input RF's row holds the binary inputs required for a convolutional window computation, while weights are provided by an external memory. The outputs of the XNOR gates have been multiplexed to reduce the computational overhead of the pop-counting part.

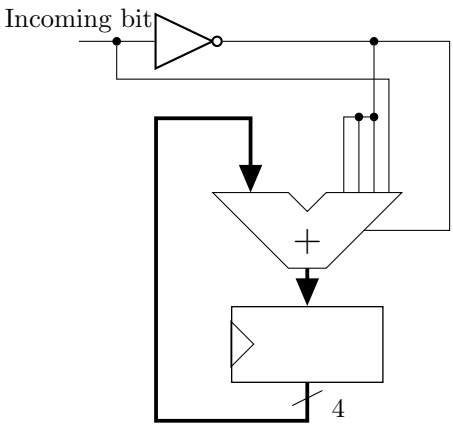

Figure 8. Four bits example of a pop-counter circuit.

4.1.2. Multiple Input Channels Design

Many CNNs have multiple IFMAPs in input. Each convolutional window must be computed separately and, in the end, summed together to get the resulting OFMAP. This can be obtained by increasing the level of parallelism of the architecture, having multiple XNOR-Pop Units working at the same time. As can be seen in Figure 9, a C_{in} number of XNOR-Pop Units are required and a final accumulation circuit computes the sum of the single channels. XNOR-Pop Units are multiplexed to reduce the hardware complexity for bigger networks.

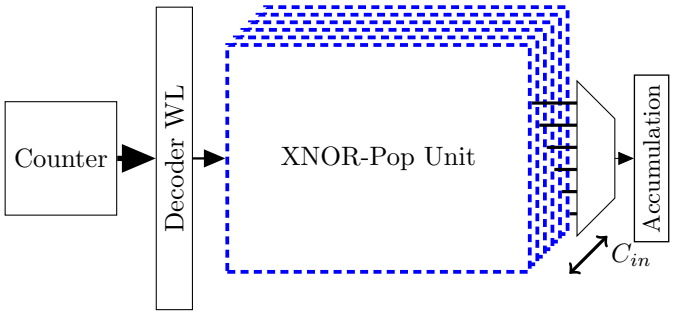

Figure 9. Multiple input channels OOM design.

4.1.3. FC Layer Integration

Until this point, the convolution algorithm has been mapped on the hardware architecture described so far. To implement the fully connected layer the same circuit can be reused by simply inverting weights and inputs sources. To better understand this concept, the example reported in Figure 10 is considere. The weights values for each input neuron are w_0^0, w_1^0, w_2^0 for X_0, w_0^1, w_1^1, w_2^1 for X_1 and w_0^2, w_1^2, w_2^2 for X_2. The output O_0 can be computed considering Equation (11).

$$O_0 = \text{pop-count}(\overline{X_0 \oplus w_0^0}, \overline{X_1 \oplus w_0^1}, \overline{X_2 \oplus w_0^2}) \tag{11}$$

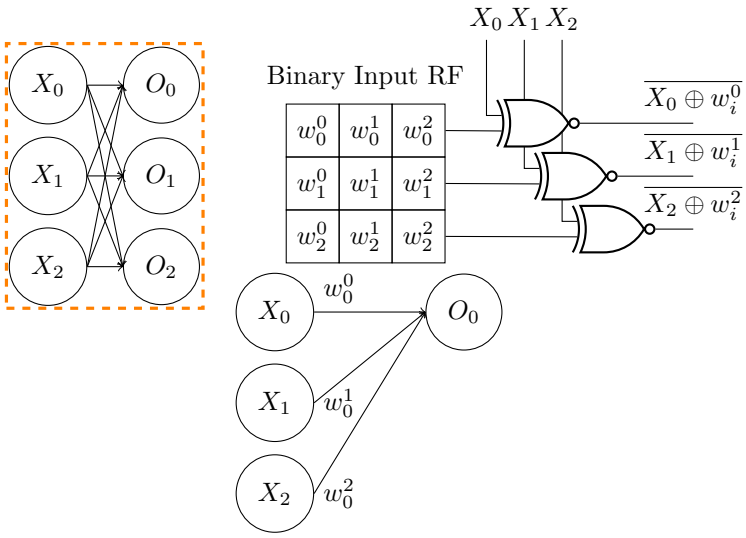

Figure 10. Example of a 3-3 FC network mapping.

As depicted in Figure 10, the Binary Input Register File (RF) contain the binary weights instead of the inputs, in fact by addressing each line the multiplication of the weights with inputs is performed, and then pop-counted.

The size of the Binary Input RF is also bounded to the FC network's characteristics, so the relations of width-height of the Binary Input RF are the following:

$$\begin{cases} \text{Memory size}_x = max(W_x \times W_y, \#\text{input neurons}_{FC}) \\ \text{Memory size}_y = max(D_{out}, \#\text{output neurons}_{FC}) \end{cases} \tag{12}$$

Although this is the straight forward way to map an FC algorithm on the architecture, this can be very complex with a high number of input neurons. Considering LeNet5 [1] depicted in Figure 3, the first FC layer has 120 output neurons, that can be acceptable, but for more sophisticated algorithms like AlexNet [2], which has 4096 input neurons, makes this kind of scheduling very inefficient. A generic output neuron's equation O_i is given by

$$O_i = \sum_{j=0}^{4095} X_j \times w_i^j + Bias = X_0 \times w_i^0 + X_1 \times w_i^1 + ... + X_{4095} \times w_i^{4095} + Bias \tag{13}$$

this sum can be computed by performing fewer number of adds per each clock cycle. The partial result is stored and added in each clock cycle. The algorithm steps become:

$$\begin{aligned} &\text{Store temp}(0) = 0 \\ &\text{Store temp}(1) = \sum_{j=0}^{n} X_j \times w_i^j + \text{Store temp}(0) \\ &\text{Store temp}(2) = \sum_{j=n+1}^{2n} X_j \times w_i^j + \text{Store temp}(1) \\ &\qquad ... \end{aligned} \tag{14}$$

where n is the total number of considered terms for each summation and Store temp holds temporary additions partial results.

Figure 11 shows an example of serialization of 2 input neurons per cycle, meaning that only a subset of weights (highlighted by the dashed lines in the figure) are stored inside the Binary Input RF.

The partial result is computed and then temporarily stored in each algorithmic step. In Equation (14), n is 2 and consequently the Memory size values can be rewritten as

$$\begin{cases} \text{Memory size}_x = max(W_x \times W_y, n) \\ \text{Memory size}_y = max(D_{out}, \#\text{output neurons}_{FC}) \end{cases} \quad (15)$$

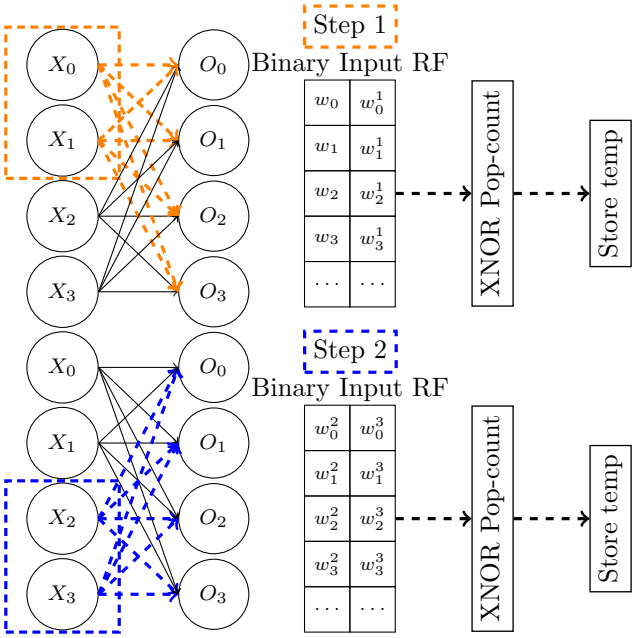

Figure 11. Example of serialization of the FC computation.

4.1.4. OOM Convolution-FC Unit

A scheme of the complete OOM architecture is now provided in Figure 12 and each element's functionality is reported in details.

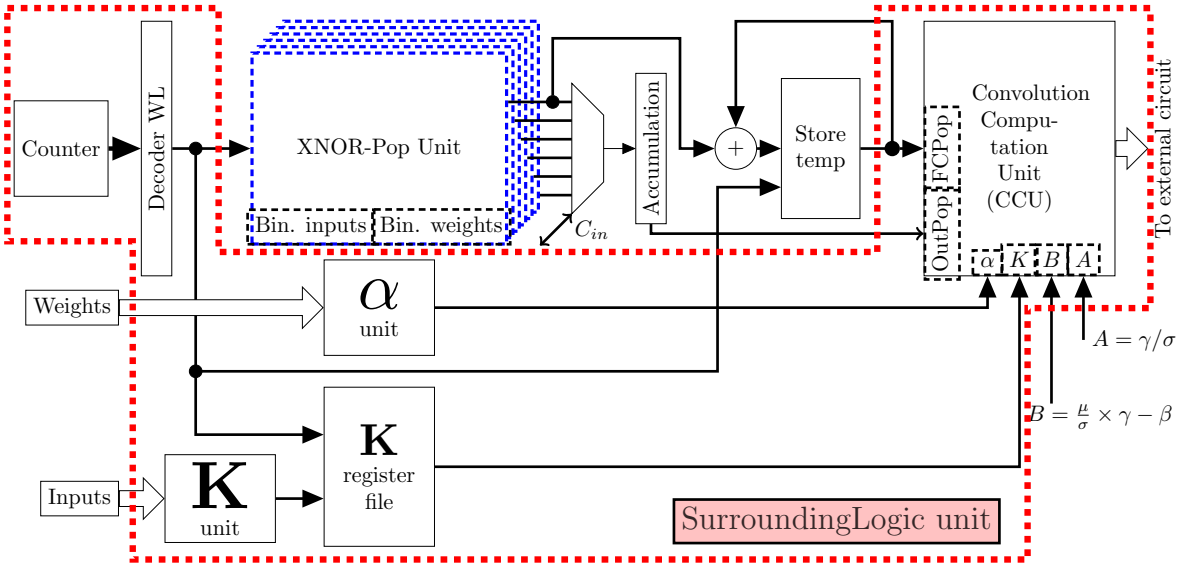

Figure 12. Complete OOM architecture. The thicker red dashed line frames the units which are the main components of the SurroundingLogic unit. Inputs are provided to the SurroundingLogic unit from the external world. Outputs are processed and saved outside in the testbench.

- XNOR-Pop Unit is the block described before, which takes Binarized weights/inputs and computes the resulting binary convolution;
- Store temp is a register file that holds the partial FC values, resulting from the scheduling described in Subsubsection 4.1.3. As the XNOR-Pop Unit's Binary Input RF, which is depicted in Figure 7, it is addressed by the same counter, since only one neuron is processed per time. Only the output coming from the first XNOR-Pop Unit is taken, because FC requires only one input channel to be executed;
- **K** and α units are in charge of computing **K** and α values, as required by XNOR-Net convolution approximation expressed in Equation (9). Since **K** are matricial values, a register file has been inserted in the design to hold them;
- Convolution Computation Unit (CCU) performs the final calculation to provide the convolutional result, which is the formula reported in Equation (9). Moreover, it applies BatchNorm, if required by the algorithm: its coefficients are computed offline and provided by the external testbench.

4.2. LiM Architecture

4.2.1. XNOR-Pop LiM Unit

The design driving concept of a LiM architecture is to increase as much as possible the level of parallelism. Starting from the OOM standard implementation, we designed two LiM arrays that perform XNOR bitwise and pop-counting operations. Since we didn't have the possibility to implement a custom memory, we used as memory element a flip flop and a static CMOS based logical part.

These choices imply a higher power and area estimation in the synthesis phase, that will be discussed more in details in Section 6. Regarding the XNOR part, the idea is to put a XNOR gate inside each memory cell and to perform the binary product between the content of the cell and an external binary input. An example is depicted in Figure 13, in which is shown how a simple 2×2 convolution is mapped inside a LiM array. In order to perform the bitwise multiplication between the binary input and the corresponding weight, as we can see from the example in Figure 13 the highlighted portion of IFMAP has to be convolved with the kernel in the following way:

$$\text{Incoming bit}_0 = \text{pop-count}(\overline{X_0 \oplus w_0}, \overline{X_1 \oplus w_1}, \overline{X_4 \oplus w_3}, \overline{X_5 \oplus w_3}) \tag{16}$$

Since one of the XNOR inputs is hardwired to an external connection, it is sufficient to store inside the memory array the input required to perform the convolution. The same for the following row line: the convolution is performed with the same kernel, so each memory row corresponds to a convolutional window.

Regarding pop-counting procedure, in order to reduce the complexity of the memory cell, we can simplify the pop-count equation in the following way:

$$\text{pop-count} = \#1s - \#0s = 2 \times \#1s - \text{length(word)} \tag{17}$$

where length(word) is intended as the size of the array entering in the pop-counter, which is 4 in Figure 13. A ones counter is simply made of half adders (HA) connected as depicted in Figure 14, so in the pop-counting part there will be a HA for each memory cell. Figure 15 provides an overview of the entire LiM implementation. It is possible to distinguish between LiM XNOR part and the LiM ones counter whose detailed architectures are depicted in Figures 13 and 14 respectively. Together, with the multiplexer depicted in Figure 15, they form the LiM XNOR-Pop unit.

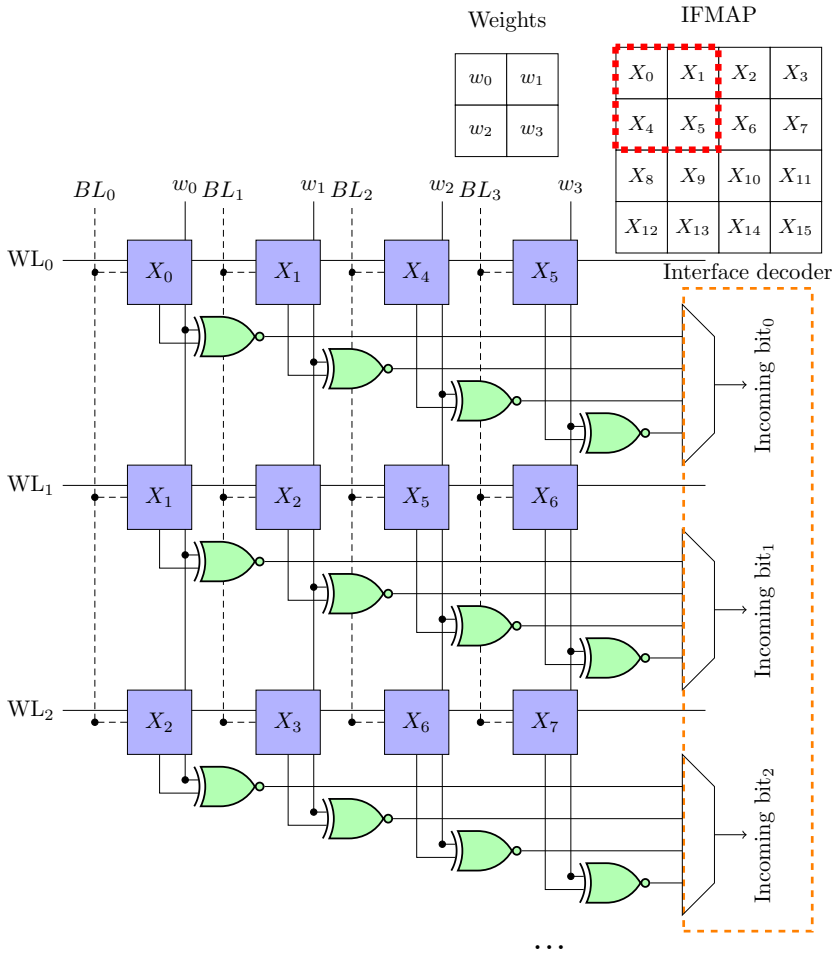

Figure 13. XNOR part of the XNOR-Pop Unit LiM implementation: example of 2×2 kernel and 4×4 IFMAP sizes with stride 1.

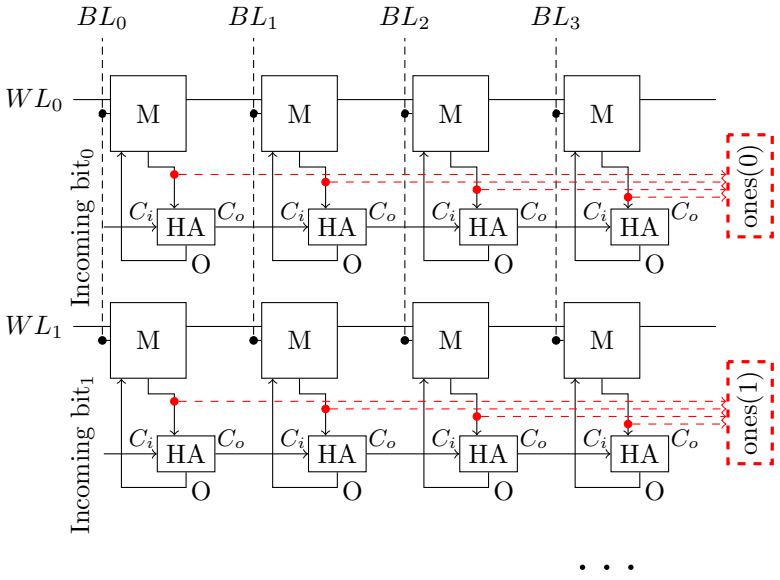

Figure 14. Example of a 4 bits ones counter LiM implementation.

4.2.2. LiM convolution-FC Unit

From the previous considerations, the entire LiM architecture can be designed as shown in Figure 15.

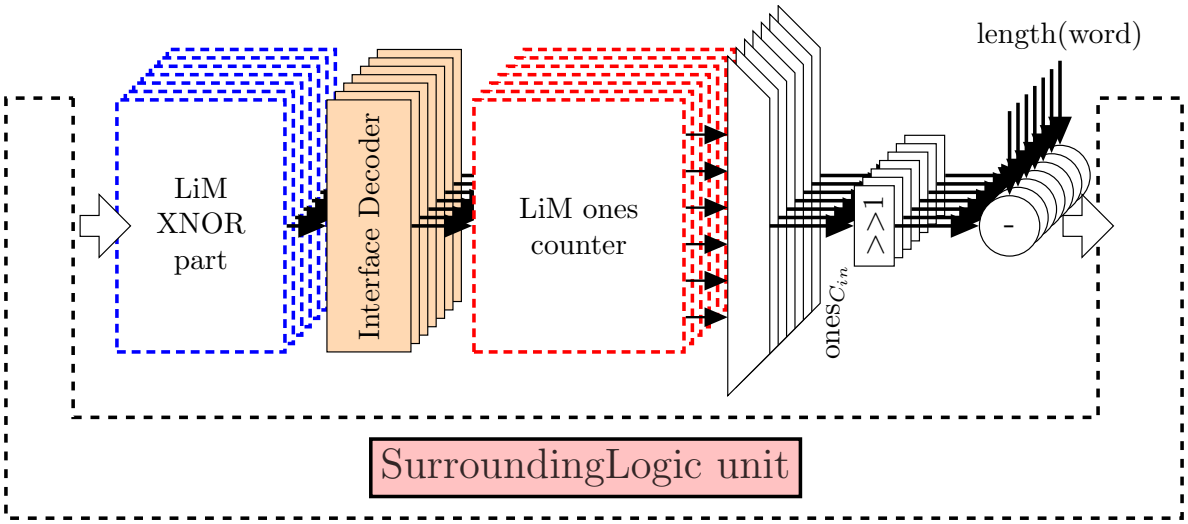

Figure 15. LiM entire architecture. The main blocks of the LiM implementation are the LiM XNOR part, interface decoder, LiM one-counter, and shifters-subtractors for the pop-counting computation, that are replicated for C_{in} number of times. The surrounding logic is the same as the OOM case reported in Figure 12.

The Surrounding logic unit remains the same, since the interface has been kept between OOM-LiM XNOR-Pop units. The other units are replicated $\#C_{in}$ times, depending on the total number of input channels required by the algorithm. The LiM alternative can achieve a higher level of parallelism, because XNOR-ones counter parts can perform the operations in parallel. In Figure 15, there are also "$<<$ 1" blocks: they perform the shift by 1 position, corresponding to multiplication by 2.

4.3. Top-Level Entity

The top-level entity contains both the Convolution-FC, Pooling circuits. Pooling is simply made of a multiplexed comparator that takes the maximum out of $W_x \times W_y$ number of inputs. The top-level entity contains also an Interface, which is in charge of dispatching the inputs coming from the testbench and to provide the results of Pooling/Convolution-FC to the outside. The top-level entity can be schematized in Figure 16.

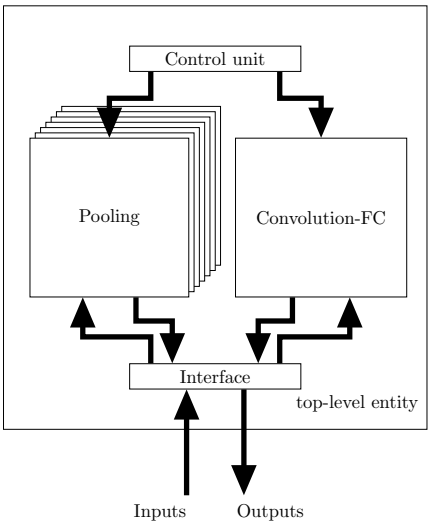

Figure 16. Top-level entity of both LiM and OOM architectures.

5. Qualitative Comparison OOM-LiM Architectures

In order to make a qualitative comparison, algorithm execution time was considered as a benchmark parameter. We can distinguish between convolution, fully connected and max pooling execution times, since they are completely different. The computation is based on a CNN, since it generally contains all of those layers. The CNN's parameters are not specified, since we are doing a parametric estimation.

5.1. OOM Execution Time

5.1.1. Pooling Layer

In our analysis we start from Pooling layer. As said in Subsection 4.3, Pooling is made of a simple multiplexed comparator. The input scanning ends when all them have been considered, so after an entire pooling window content is evaluated. This value is multiplied by the total number of pixels of the resulting OFMAP obtaining

$$\text{Pool}_{time} \approx D^2_{out(pool)} \times (W_{x(pool)} \times W_{y(pool)}) \times t_{ck} = D^2_{out(pool)} \times (W^2_{x(pool)}) \times t_{ck} \quad (18)$$

where $D^2_{out(pool)}$ is the pooled OFMAP size. The worst-case filter dimension is set to W^2_x for both convolutional and pooling.

5.1.2. Convolutional Layer

At the beginning of the convolution algorithm, the binary inputs are precharged inside the Binary Input RF and **K** matrix is computed in the meanwhile, meaning that for each input set are required W^2_x clock cycles. Since an entire OFMAP has a number of pixels equals to $D^2_{out(conv)}$, the total number of cycles required in this step are $D^2_{out(conv)} \times W^2_x$ clock cycles. After that, convolution is performed: considering Figure 7, an entire convolutional window is computed when all the XNOR outputs have been scanned. The number of XNOR gates is equal to the Binary Input RF's word length, which is W^2_x. By multiplying the time required by a convolutional window computation with the total number of convolutional windows $D^2_{out(conv)}$, we get the total convolution time which is $\text{Convolution}_{time,OOM}$. The last contribution set is the BatchNorm, that can applied after each convolutional window and α

computation together with results' storing. Each of them takes only 1 clock cycle. We can derive the equation for the convolutional layer execution time with 1 input/output feature map as follows.

$$\text{Convolution}_{time,OOM} \approx \left(\overbrace{D^2_{out(conv)} \times W^2_x}^{\text{Store inputs \& } \mathbf{K} \text{ computation}} + \overbrace{D^2_{out(conv)} \times (W^2_x + 1)}^{\text{Convolution \& BatchNorm}} + \overbrace{(1+1)}^{\alpha \text{ - Store results}} \right) \times t_{ck} \quad (19)$$

When multiple output channels are considered, the convolution windows computation has to be repeated for each of the OFMAP:

$$\begin{aligned} \text{Convolution}_{time,OOM} \approx & \overbrace{D^2_{out(conv)} \times W^2_x}^{\text{Store inputs \& } \mathbf{K} \text{ computation}} \times t_{ck} + \\ & + C_{out} \times \left(\overbrace{D^2_{out(conv)} \times (W^2_x + 1)}^{\text{Convolution \& BatchNorm}} + \overbrace{(1+1)}^{\alpha \text{ - Store results}} \right) \times t_{ck} \end{aligned} \quad (20)$$

the last situation is the multiple input/output channels case. Since the convolution operation is parallelized, the convolutional windows coming from each XNOR-Pop Unit is added in a serial fashion. This means that to achieve the final convolution value, each contribution has to be added together before executing the BatchNorm. The final $\text{Convolution}_{time}$ expression is the following.

$$\begin{aligned} \text{Convolution}_{time,OOM} \approx & \overbrace{D^2_{out(conv)} \times W^2_x}^{\text{Store inputs \& } \mathbf{K} \text{ computation}} \times t_{ck} + \\ & + C_{out} \times \left(\overbrace{D^2_{out(conv)} \times (W^2_x + 1 + C_{in})}^{\text{Convolution \& BatchNorm multiple } C_{in}} + \overbrace{(1+1)}^{\alpha \text{ - Store results}} \right) \times t_{ck} \end{aligned} \quad (21)$$

5.1.3. FC Layer

For the FC computation, we have to consider the scheduling explained in Subsubsection 4.1.3. As the convolution case, the algorithm starts precharging the inputs inside the array, taking $D_{out(FC)}$ clock cycles, where $D_{out(FC)}$ is the total number of output neurons. Since the dimension of the Binary Input RF is Memory size_x, only Memory size_x input neurons are considered per time, so as performed for the convolutional layer, the time required for a FC output is equal to $D_{out(FC)} \times \text{Memory size}_x$ that has to be added to the previous contribution. FC results have to be stored, and this can be made by scanning the content of Store temp register (depicted in Figure 12), taking $D_{out(FC)}$ clock cycles. The execution time for a single step of the FC scheduling is given by:

$$\text{FC}_{time,OOM} \approx \left(\overbrace{D_{out(FC)}}^{\text{Store inputs}} + \overbrace{D_{out(FC)} \times \text{Memory size}_x}^{\text{FC output computation}} + \overbrace{D_{out(FC)}}^{\text{Store temp scanning}} \right) \times t_{ck} \quad (22)$$

this partial result has to be repeated by the total number of iterations (n_{iter}) required to calculate the FC layer. The final FC execution time expression is:

$$\text{FC}_{time,OOM} \approx \left[n_{iter} \times \left(\overbrace{D_{out(FC)}}^{\text{Store inputs}} + \overbrace{D_{out(FC)} \times \text{Memory size}_x}^{\text{FC output computation}} \right) + \overbrace{D_{out(FC)}}^{\text{Store temp scanning}} \right] \times t_{ck} \quad (23)$$

5.2. LiM Execution Time

Similarly to the OOM case, Pooling, Convolution and FC execution times are provided and explained. Since Pooling layer is the same in both cases, it is not analyzed in this part.

5.2.1. Convolutional Layer

As already done in OOM, the array has to be precharged taking $D^2_{out(conv)}$ clock cycles. After that, all the XNOR gates inside the XNOR part work together at the same time, and the Interface Decoder, which is depicted in Figure 13, takes one by one each XNOR result and provide it to the ones counter. When all XNORs' output have been scanned after W_x^2 clock cycles, the ones counter results are stored inside the LiM ones counter reported in Figure 15. At this point, all the LiM ones counter values must be fetched for each input channel, requiring $C_{in} \times D^2_{out}$ clock cycles to perform the residual part of the algorithm. The final formula for the LiM convolution execution time is

$$\text{Convolution}_{time,LiM} \approx \overbrace{D^2_{out(conv)} \times W_x^2}^{\text{Store inputs \& } \mathbf{K} \text{ computation}} \times t_{ck} + \\ + C_{out} \times \left[\overbrace{W_x^2 + D^2_{out(conv)} \times (1 + \#C_{in})}^{\text{Convolution \& BatchNorm multiple } C_{in}} + \overbrace{(1+1)}^{\alpha \text{ - Store results}} \right] \times t_{ck} \tag{24}$$

5.2.2. FC Layer

Similarly to the OOM case, we have scheduled the algorithm to reduce the complexity. After $D_{out(FC)}$ clock cycles required to store the values inside the LiM array, an entire FC step is computed in Memory size$_x$ clock cycles and the final results are scanned from the LiM ones counter in $D_{out(FC)}$ cycles. In LiM architecture, the Store temp register file is not required since the pop-count values are already stored in the LiM part. By iterating the entire algorithm n_{iter} times, we get the final FC execution time:

$$\text{FC}_{time,LiM} \approx \left[n_{iter} \times \left(\overbrace{D_{out(FC)}}^{\text{Store inputs}} + \overbrace{\text{Memory size}_x}^{\text{FC output computation}} \right) + \overbrace{D_{out(FC)}}^{\text{Store temp scanning}} \right] \times t_{ck} \tag{25}$$

5.3. Comparison Results

The results obtained by performing the ratio between OOM/LiM execution times are now provided. The previous part, and in particular Subsections 5.1 and 5.2 take into account an approximate computation of the execution time, since the overheads of idle/dummy states were not considered for sake of simplicity. In this part, we show the real estimations that consider all the contributions. Considering Figure 17, it is possible to see how delay ratio (obtained as execution time OOM/execution time LiM) changes in different cases. A series of sweeps were made, considering the most important variables, in particular $\#C_{in}, \#C_{out}, W_x, D_{in}, D_{out(fc)}, n_{iter}$. On the vertical axes, there is Delay ratio in all plots. Some of the estimations were performed considering the convolution timing equations reported in Equations (21) and (24). These plots are are labelled with "Convolution computation" flag in Figure 17. The remaining one consider FC delay expressions reported in Equations (23) and (25).

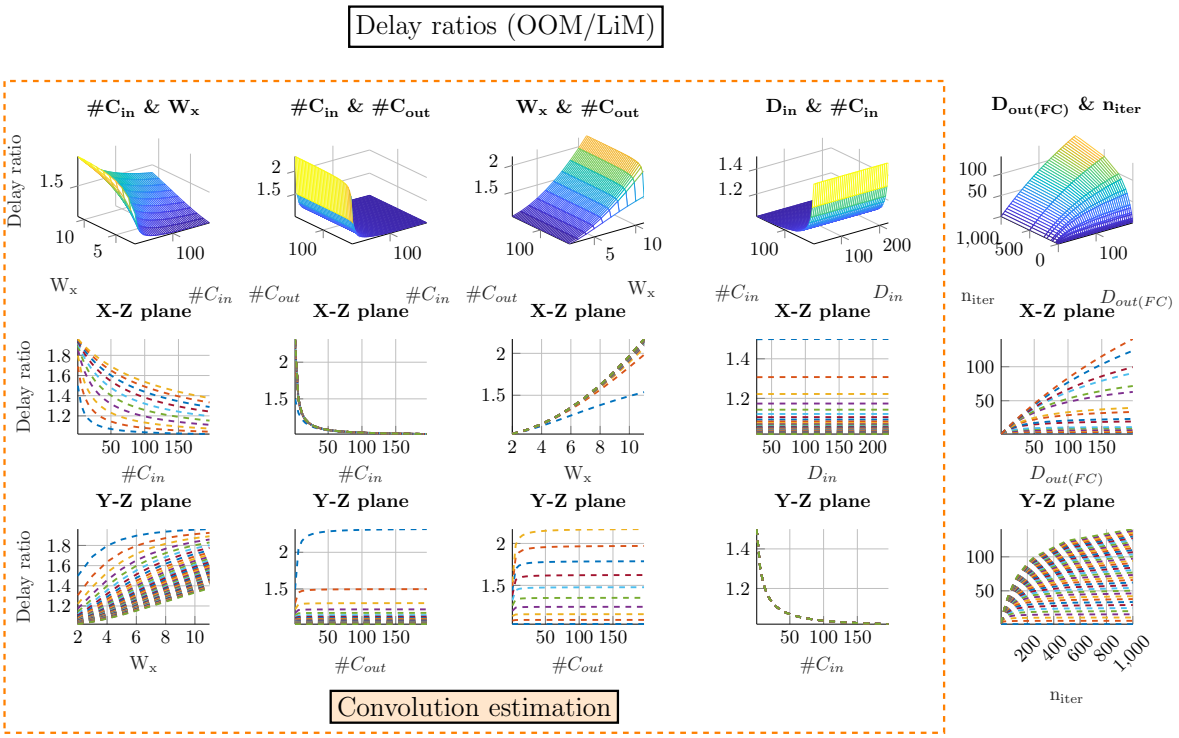

Figure 17. Delay ratio obtained as OOM/LiM for different parameters, in order to see how the two architectures behave for different cases.

- Delay ratio vs #C$_{in}$ & W$_x$: the Delay ratio with respect to #C_{in} has a decreasing trend because, as shown in Figure 15, the Interface Decoder, the multiplexers placed after the LiM ones counter and the serial accumulation of the values of each channel represent a bottleneck for LiM architecture. As a result a higher execution time for higher values of #C_{in} is observed. In general, for high values of W_x, the Delay ratio increases, because of the parallelization in LiM architecture.
- Delay ratio vs #C$_{in}$ & #C$_{out}$: the trend for #C_{in} is the same as the previous case. For high values of #C_{out}, we can expect a very good Delay ratio efficiency, because LiM already has the values stored inside the array and it is sufficient to change the weights set by simply selecting it, following the same principle of the OOM case depicted in Figure 7.
- Delay ratio vs W$_x$ & #C$_{out}$: in general, by increasing both #C_{out} and W_x we have a higher Delay ratio. By looking to X-Z plane, it is possible to see that for higher #C_{out} the curve becomes steeper. It is a very good trend for very deep NNs, because usually output channels and filter sizes are high.
- Delay ratio vs D$_{in}$ & #C$_{in}$: D_{in} is the IFMAP size, which indirectly determines the OFMAP size as reported in Equation (10). High values of D_{in} imply much more complex NN but the Delay ratio remains almost constant, showing that LiM architecture latency is not degraded by the IFMAP's complexity.
- Delay ratio vs D$_{out(FC)}$ & n$_{iter}$: this last plot set reports an FC layer estimation. In this case the formula for the FC execution time of OOM and LiM is considered. As it can be seen, a higher $D_{out(FC)}$ could be beneficial for a LiM architecture, which has a small increasing trend, because the LiM array performs all the computations in parallel, so there is no need to fetch each data from the memory, compute the result and store inside the Store temp register file as in the OOM case. The predominant variable is n_{iter}, because by looking at Figure 7, the OOM architecture has the important drawback that everytime an FC step terminates, the entire Binary Input RF has to be scanned to perform the FC computation, requiring $n_{iter} \times D_{out(FC)} \times$ Memory size$_x$ clock cycles. If the number of output neurons is huge ($D_{out(FC)}$),$n_{iter} \times D_{out(FC)} \times$ Memory size$_x$ becomes very large compared to the LiM case.

From these considerations, it is evident that LiM architecture introduces a gain in terms of execution time, because by increasing the level of parallelism in the architecture, multiple operations can be performed at the same time. The LiM bottlenecks are the Interface Decoder and the multiplexers depicted in Figure 15, that introduce both higher delay and power consumption, but they are required to interface the design blocks.

6. Perfomance Evaluation

In this part, the evaluation steps will be explained. In this work the memories were implemented as register files and each memory cell is a flip flop, so the results obtained are an overestimation (especially for the LiM case). The real performance values can be obtained with a more precise memory model. The performance evaluation is made of three parts:

1. For both OOM and LiM implementations, two NN models were chosen and used as cases of study. These models were implemented, trained and validated by Keras framework [32] and a Matlab script respectively. Then, the architectures were synthesized with Synopsys Design Compiler with 45nm CMOS technology @ 1.1V, providing the values of power, area, Critical Path Delay (CPD), execution time and energy consumption. Regarding the power consumption, two kind of estimations are provided: the first is very straight forward and consists of a report power from Synopsys with worst case scenario of switching activity equals to 1 in all the nodes. The second, a post place&route power estimation with Cadence Innovus, using backannotation with `.vcd` file provided by Modelsim, in order to evaluate the effect of both switching activity and interconnections.
2. Parametric sweeps are performed in order to evaluate the trend of the performance parameters in different cases. Power, Area, CPD and Energy ratios are computed between the OOM and LiM values, that are particularly useful to determine the main contributions of both architectures. To perform such procedure, a series of scripts are used to perform several synthesis processes with Synopsys Design Compiler and, everytime a synthesis ends, the performance values are stored in external files. Also in this case, the technology used is 45nm CMOS @ 1.1V.
3. An analysis of the differences between our LiM, where memory elements are flip flops, and a LiM circuit with a custom memory is performed. In [8], a very similar XNOR-Net implementation has been implemented with a CAM memory-based XNOR-Pop procedure. Some useful results are provided, since authors have implemented a modified memory array with 65nm CMOS technology. For this reason, a synthesis with 65nm CMOS technology @ 1.0V is performed, trying to use the same metrics as [8] to evaluate how a more real memory model can influence the results obtained.

6.1. Two NN Models Examined

6.1.1. Fashion-MNIST CNN Results

The first NN model is able to classify with an accuracy of 81% a Fashion-MNIST image [33], which is a greyscale picture of 28×28 pixels that can belong to one of 10 different categories such as T-shits, trousers, pullovers, dresses, coats, sandals, shirts, sneakers, bags and ankle boots. The NN model is reported in Table 1. The parameters listed in Table 1 give an indication on the dimensions of the hardware implementations, such as the dimensions Memory size_x and Memory size_y of the Binary Input RF/LiM arrays in order to perform an ad-hoc synthesis optimized for that NN model. Binary Input RF, LiM XNOR part and LiM ones counter have Memory $\text{size}_y = 24 \times 24 = 576$ rows with a bitwidth of Memory $\text{size}_x = 32$ bits to host both convolution and FC algorithms. Since there are a maximum number of input channels equals to 6 as reported in Table 1, the XNOR-Pop Unit for both OOM and LiM has been replicated 6 times.

Table 1. Fashion-MNIST CNN under test parameters.

Layer Number	Type	IFMAP Size	Kernel Size	C_{in}	C_{out}	Stride
1	Convolutional	$28 \times 28 \times 1$	5×5	1	6	1
2	Max Pooling	$24 \times 24 \times 6$	2×2	6	6	1
3	Convolutional	$12 \times 12 \times 6$	5×5	6	6	1
4	Max Pooling	$8 \times 8 \times 6$	2×2	6	6	1
5	FC	96	-	-	-	-
6	BatchNorm	-	-	-	-	-
7	FC	120	-	-	-	-
8	BatchNorm	-	-	-	-	-
9	FC	84	-	-	-	-
10	BatchNorm	-	-	-	-	-
11	FC	10	-	-	-	-
12	BatchNorm	-	-	-	-	-
13	ReLU	-	-	-	-	-

The number of bits used to perform the extra calculations required (such as multiplications, BatchNorm etc) are 18 expressed in fixed point format. After performing the synthesis with Synopsys Design Compiler, the results obtained for area, CPD and power are reported in Table 2.

Table 2. Synthesis results for the Fashion-MNIST CNN model.

Type	Area [mm^2]	Power [mW]	CPD [ns]
LiM	1.68	254.50	4.11
OOM	1.10	193.30	4.14

A preliminary analysis of the results listed in Table 2, highlights a higher area and power consumption in LiM with respect to the OOM alternative, since the LiM implementation is highly parallellized and, consequently, a higher number of logic elements are required. The CPD is slightly higher in the OOM case because of a more complicated FC scheduling handling circuit (depicted in Figure 12), which requires the Store temp register file. From these results, a simple comparison between power, area and CPD is not enough to determine which is the best architecture between the two proposed ones. For this reason we estimate also the execution time and the energy estimation obtained as Power $\times$ Execution time, as shown in Table 3.

Table 3. Power-Execution time-Energy results for Fashion-MNIST CNN model.

Type	Power [mW]	Execution Time [ms]	Energy [µJ]
LiM	254.50	0.21	53.44
OOM	193.30	0.92	178.41

From Table 3, we can derive two very important values which are the Energy ratio and Delay ratio as follows

$$\text{Delay ratio} = \frac{\text{Execution time}_{OOM}}{\text{Execution time}_{LiM}} = \frac{0.92\,\text{ms}}{0.21\,\text{ms}} \simeq 4.38 \tag{26}$$

$$\text{Energy ratio} = \frac{\text{Energy}_{OOM}}{\text{Energy}_{LiM}} = \frac{178.41\,\mu\text{J}}{53.44\,\mu\text{J}} \simeq 3.34 \tag{27}$$

Energy and delay ratio give very important indications on LiM strong points: it consumes less energy and it's faster, although has a higher power value. From an energetic point of view, LiM architecture is more efficient for that particular NN model. In Figure 18 and Table 4 are reported the results obtained from a post place&route estimation with `.vcd` backannotation. Considering both switching activity

and interconnections, the resulting power of the LiM architecture is increased by $\sim 22\%$, bringing a lower energy ratio, but still promoting LiM as an energy efficient architecture.

$$\text{Energy ratio}_{\text{post place and route}} = \frac{130.91\,\mu\text{J}}{68.9\,\mu\text{J}} \simeq 1.9 \tag{28}$$

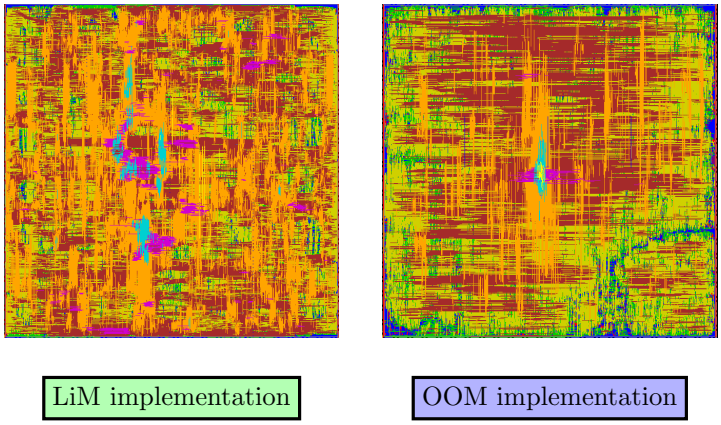

Figure 18. Snapshot of the chips obtained after a post place and route procedure for Fashion-MNIST CNN.

Table 4. Post place and route estimation of Fashion-MNIST CNN implementation.

Type	Area [mm^2]	Power [mW]	CPD [ns]	Execution Time [ms]	Energy [µJ]
LiM	1.70	328.3	4.11	0.21	68.9
OOM	1.07	142.3	4.14	0.92	130.91

6.2. MNIST-MLP Network

Another NN model is evaluated, in order to verify the behavior of both LiM and OOM architectures with different computational models. The realized MLP network is made of a set of FC layers organized as 784-196-196-10 neurons and it is able to achieve up to $\sim 90\%$ of accuracy on MNIST dataset. Further details on MLP structure are presented in Table 5. Dropout layers indicated in Table 5 are useful in training procedure, since they prevent the network from overfitting by simply "turning off" neurons with a given probability [34]. As already done in Subsubsection 6.1.1, the results that will be presented are the ones obtained by the synthesis and the estimated value of energy, based on the execution time. The chosen dimensions of the implementation are Memory size$_x = 14$, $\#C_{in} = 1$ while the memory arrays have 196 rows because the maximum number of output neurons ($D_{out(FC)}$) is 196.

Table 5. MNIST MLP under test parameters.

Layer Number	Type	IFMAP Size	Kernel Size	C_{in}	C_{out}	Stride
1	Dropout	$28 \times 28 \times 1$	-	-	-	-
2	FC	784	-	-	-	-
3	BatchNorm	196	-	-	-	-
4	ReLU	196	-	-	-	-
5	Dropout	196	-	-	-	-
6	FC	196	-	-	-	-
7	BatchNorm	196	-	-	-	-
8	ReLU	196	-	-	-	-
9	Dropout	196	-	-	-	-
10	FC	196	-	-	-	-
11	BatchNorm	10	-	-	-	-

In Table 6, the architectures have a similar power consumption, because in OOM it is required a Store temp register file that has 196 rows. The hardware complexity of the LiM implementation is not so different from the OOM's one, because the memory arrays have a very small size of 196×14. Since it is an MLP network, n_{iter} parameter becomes very important, because it gives an indication on how many times the FC scheduling has to be executed for each layer: n_{iter} changes for each FC layer and can be obtained as $n_{iter} = D_{in(FC)}/\text{Memory size}_x$. From the energy and execution time results in Table 6, it is evident that OOM is not competitive with respect to the LiM version. This is due to the much more inefficient FC handling, since the whole Binary Input RF has to be scanned, while the LiM version performs all the calculations directly inside the array.

$$\text{Delay ratio} = \frac{1.62\,\text{ms}}{0.132\,\text{ms}} \simeq 12.27 \tag{29}$$

$$\text{Energy ratio} = \frac{23.20\,\mu\text{J}}{1.99\,\mu\text{J}} \simeq 11.7 \tag{30}$$

After performing post place and route estimation, the results obtained are reported in Figure 19 and in Table 7.

Table 6. Perfomance parameters of MLP implementation.

Type	Area [mm^2]	Power [mW]	CPD [ns]	Execution Time [ms]	Energy [µJ]
LiM	0.11	15.10	4.22	0.132	1.99
OOM	0.09	14.32	4.32	1.62	23.20

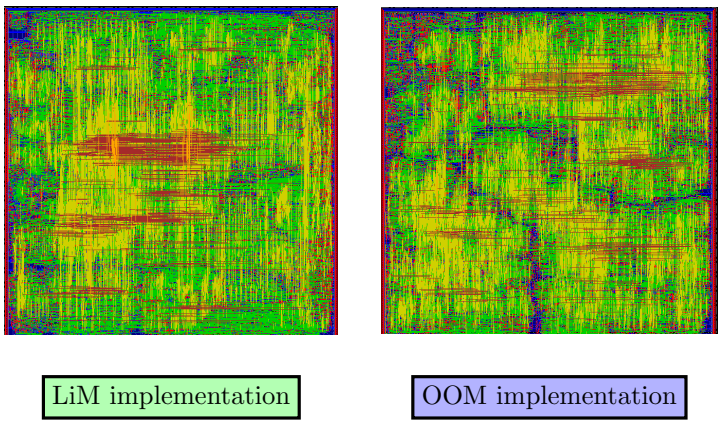

Figure 19. Snapshot of the chips obtained after a post place and route procedure for MLP NN.

Table 7. Post place and route estimation of MLP implementation.

Type	Area [mm^2]	Power [mW]	CPD [ns]	Execution Time [ms]	Energy [µJ]
LiM	0.1033	13.06	4.22	0.132	1.72
OOM	0.086	10.68	4.32	1.62	17.30

In Table 7, the power results are slightly lower than the synthesis ones, since the `.vcd` file and the switching activity information have relevant roles, giving a more precise power estimation, instead of the worst case reported in Table 6. The energy ratio results to be equal to $\sim 10\times$ compared to the previous one equal to $\sim 11.7\times$ provided by the synthesis.

6.3. Parametric Sweeps

The meaningful parameters of the designs such as $\#C_{in}$ and memory arrays dimensions (Memory size$_x$,Memory size$_y$) were varied to determine the differences between the two architectures in terms of performance. Two parameters are chosen per time and a sweep is executed on them, while the remaining are kept constant. For sake of clarity, from now on the following substitution is considered:

$$\begin{cases} H = \text{Memory size}_y^2 \\ W_a = \text{Memory size}_x \end{cases} \tag{31}$$

In Figures 20 and 21 are depicted the Power, Area and CPD for different values of $\#C_{in}$, W_a and $\sqrt{H}$. By increasing the W_a, power and area increase almost quadratically since W_a directly influences the bitwidth of the memories. Also, the trends depending by $\sqrt{H}$ behave quadratically, meaning that a the memory complexities influence a lot the performance of both architectures. In general, the power and area for LiM case are slightly higher than the OOM ones, since the total number of logic elements required by the LiM implementation is greater than OOM. CPD remains almost the same, even for more complex implementations. To better understand the differences of the obtained parameters for both architectures, a ratio was computed for all the cases: the results obtained are reported in Figure 22, where in general for an increasing size of $\#C_{in}$, W_a and $\sqrt{H}$ the power and area ratios decreases, confirming the bigger grade of complexity of the LiM. Another useful estimation can be performed on the energy ratios for the various cases.

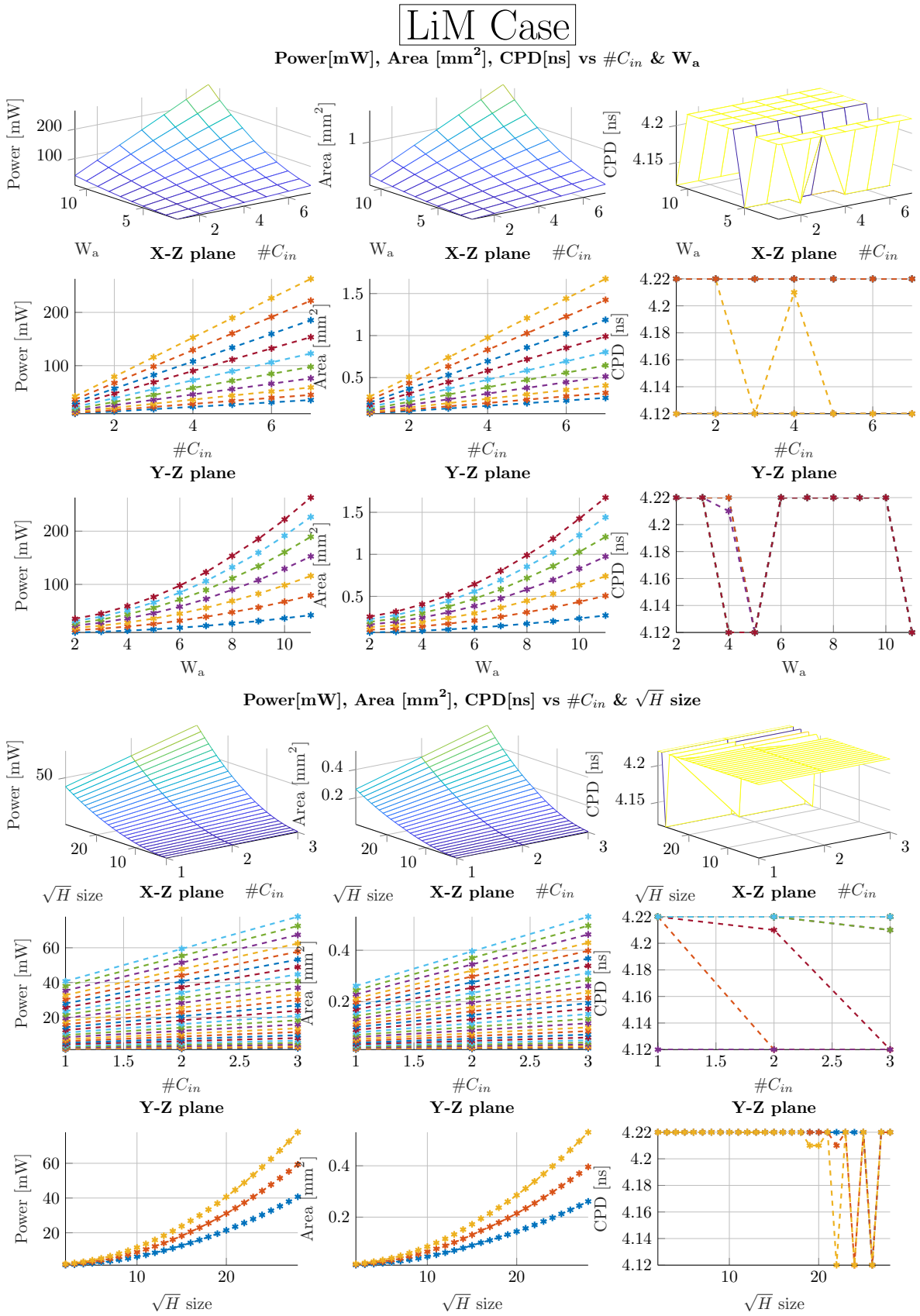

Figure 20. Power, Area and CPD results for different values of #C_{in}, W_a and $\sqrt{H}$ considering LiM implementation.

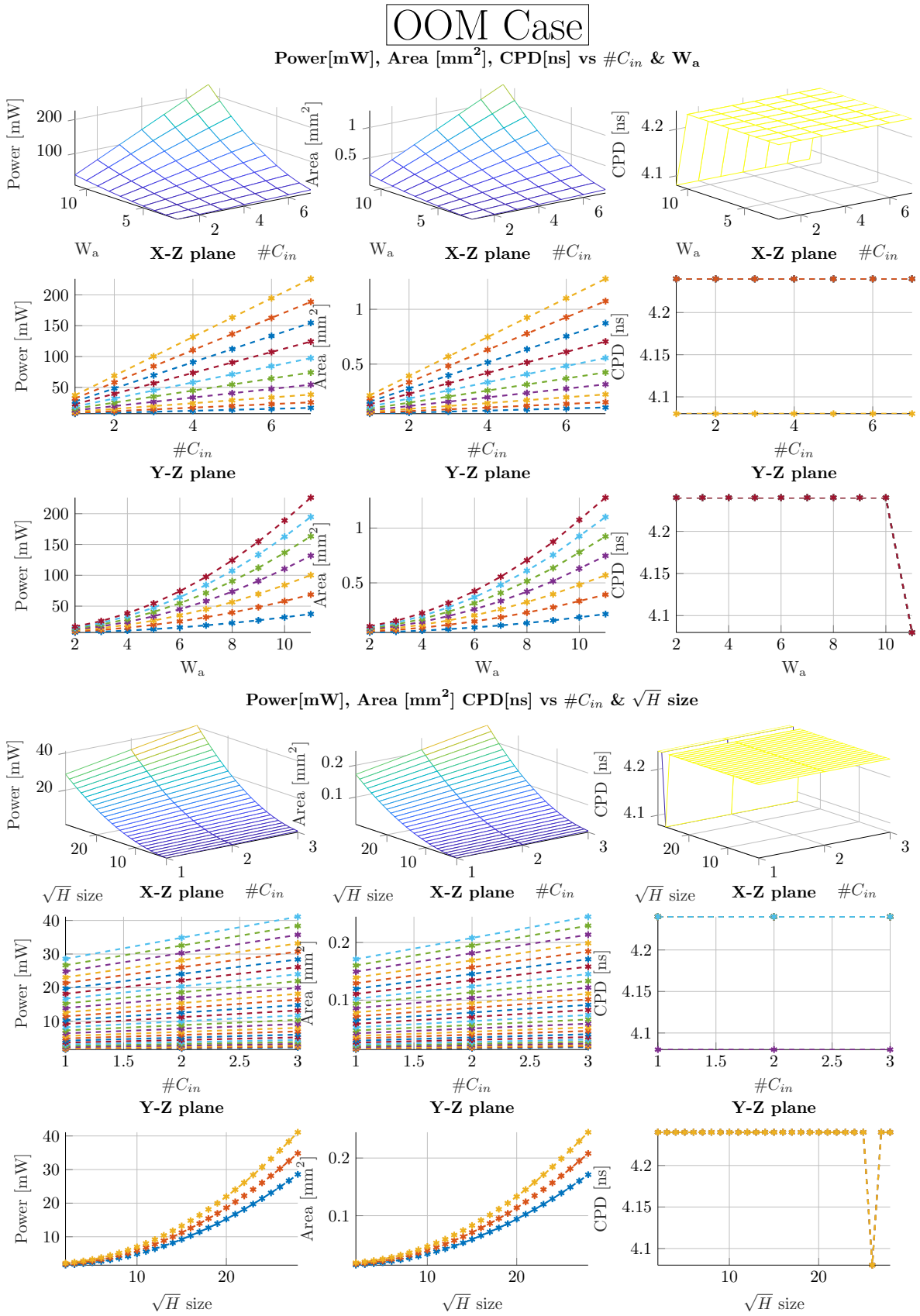

Figure 21. Power, Area and CPD results for different values of #C_{in}, W_a and $\sqrt{H}$ considering OOM implementation.

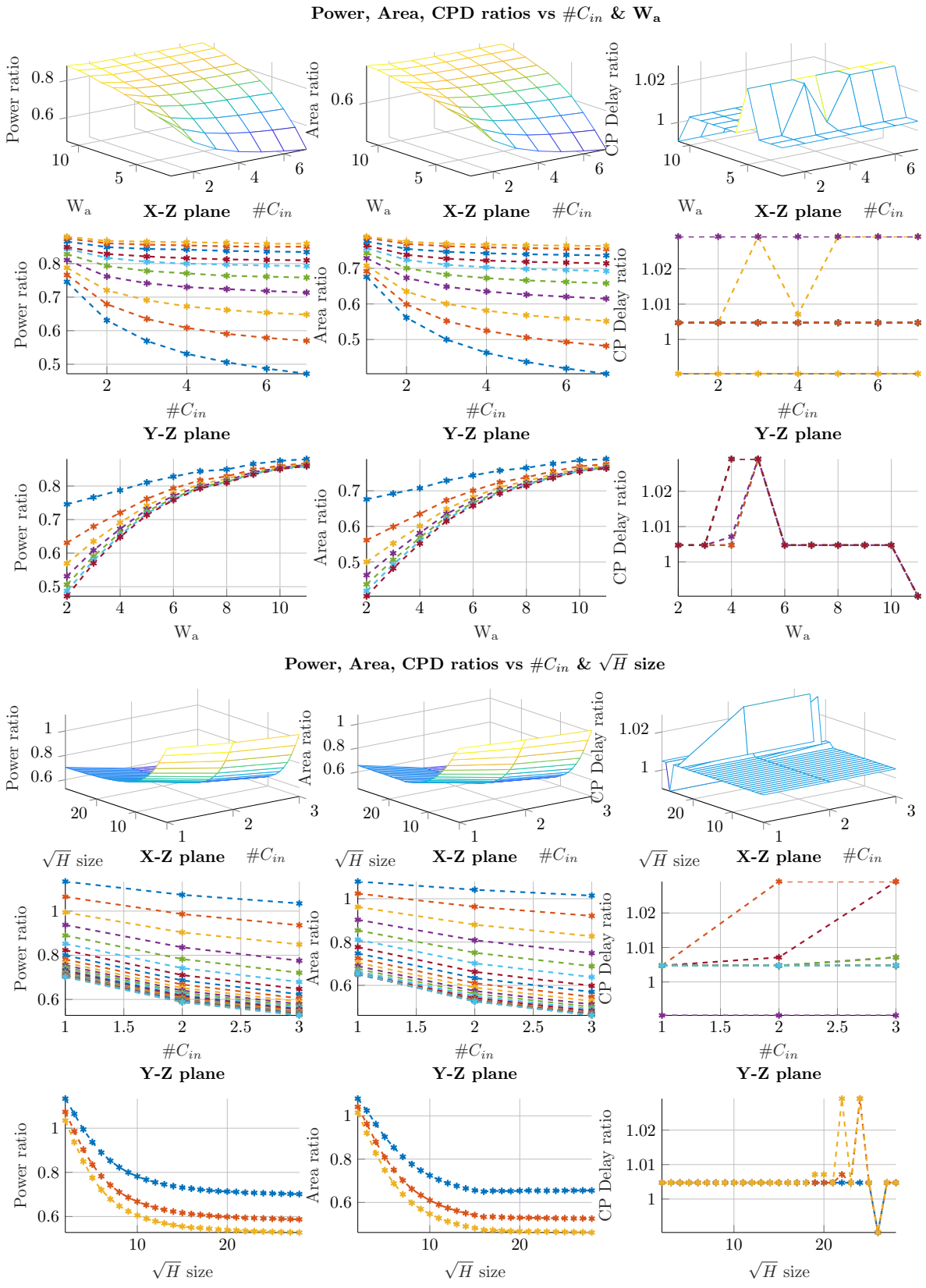

Figure 22. Power, Area and CPD ratios with respect to #C_{in}, W_a and $\sqrt{H}$.

Those values are obtained as $\text{Energy}_{OOM}/\text{Energy}_{LiM}$, as shown in Figure 23, decrease for higher memory dimensions, since power of the LiM architecture starts to assume a predominant contribution in the energy equation. It is important to keep in mind that these are very pessimistic estimations,

and they can be improved by employing more realistic memory cells. The pessimistic case, which is reported in Figure 23 in #C_{in} & $\sqrt{H}$ size plot, is to have a very long ($\sqrt{H}$ big) and narrow (W_a very small) memory structure, which is replicated a lot of times (#C_{in} big): these set of conditions describes an improbable situation, because the driving force for a memory design is to have a regular squared shape array. The last energy estimation reported in Figure 23 flagged by FC #C_{in} & $\sqrt{H}$ size, takes into account an FC algorithm mapped on the implementations considering the worst case of big $\sqrt{H}$ and #C_{in}. By varying $\sqrt{H}$, the trend for the energy ratio is increasing, meaning that the more complex is the FC algorithm the lower is the energy for the LiM implementation compared to OOM one.

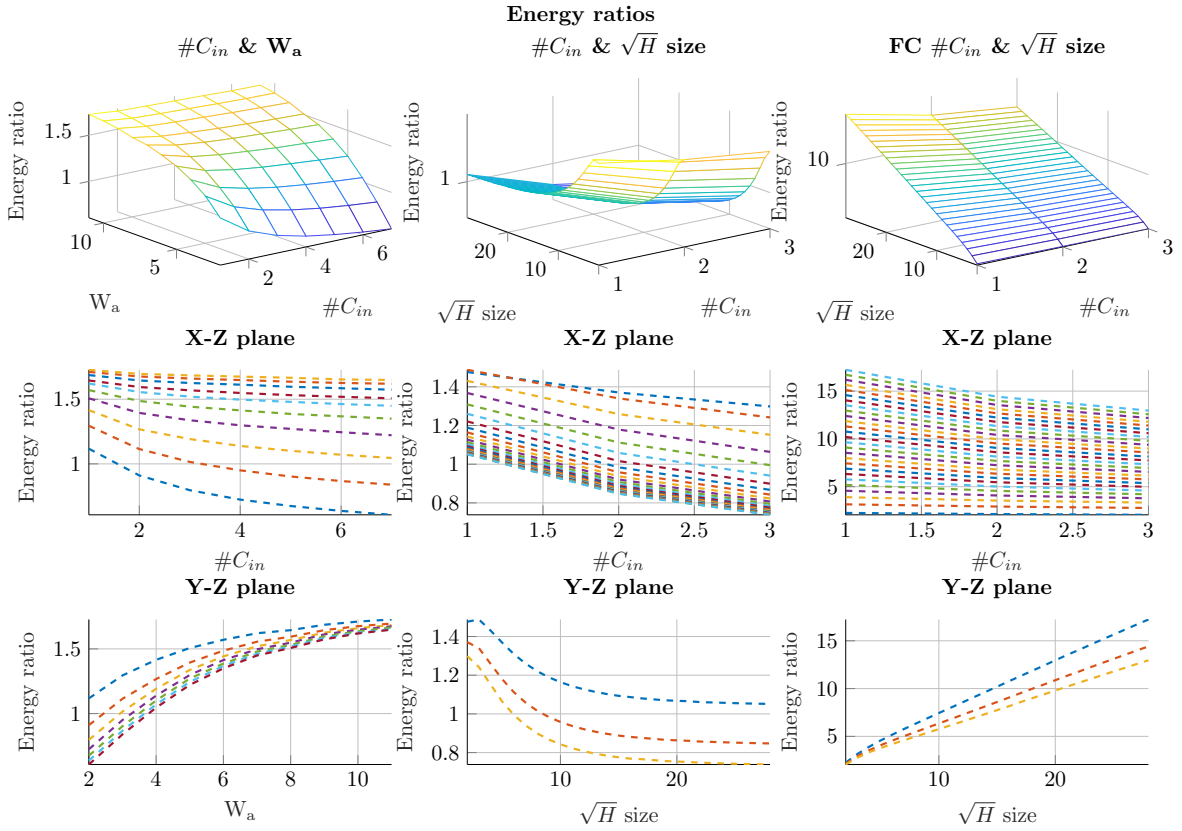

Figure 23. Energy ratio values obtained by varying #C_{in}, W_a and $\sqrt{H}$.

6.3.1. Qualitative Estimation

To give a definitive answer on which architecture performs better, a qualitative estimation is performed, considering the mean values of all the cases explained before.

A ratio obtained as OOM/LiM between each parameter is proposed, which clarify the main points of both implementations. As shown in Figure 24, the values of area and power ratios are below 1, meaning that in general the LiM architecture behaves worse than OOM for the motivations explained before. On the other hand, execution time and energy ratios are equal to $\sim 6\times$ and $\sim 4\times$ respectively, implying that a very good improvement can be achieved by the LiM implementation on these two quantities. These trends confirm our expectations on LiM and further improvements can be achieved by having a more precise LiM array model.

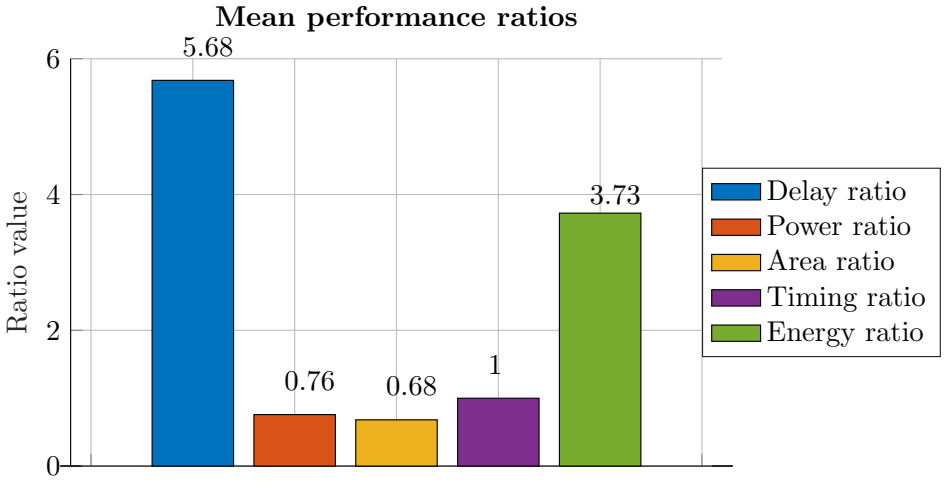

Figure 24. Mean performance ratios obtained as an average of all the cases analyzed from the previously discussed results.

6.3.2. LiM Array Estimation: Impact on Perfomance

In order to estimate the performance of the LiM array, several synthesis estimations were performed with different arrays dimensions. Taking into account the system's structure depicted in Figure 15, the LiM values of power and area are compared with the ones obtained from the same process applied only to the SurroundingLogic unit, in order to understand what are the main performance contributions. In Figure 25 are shown the performance values obtained by sweeping both $\sqrt{H}, W_a$, while $\#C_{in}$ is kept equal to 1, in order to estimate how the array sizes impact the overall performance. As it is possible to see, area and power increases almost quadratically, because of a more complex LiM structure. In Figure 26 it is reported an estimation of the SurroundingLogic unit by varying the same parameters as in Figure 25. The CPD bottleneck is located in the SurroundingLogic unit rather than LiM parts, because of the multipliers/adders employed to perform the final convolution result. Higher values of W_a implies a constant power/area, since there is no correlation between the LiM Memory size$_x$ and the complexity of the SurroundingLogic unit. By increasing $\sqrt{H}$, power and area increase because of the higher complexity required, for example a bigger dimension of the **K** register file (Figure 12). By comparing the performance in terms of power obtained in Figures 25 and 26, it is possible to see that the highest contribution comes from LiM parts, as shown in the breakdown plot depicted in Figure 27. The percentage values are obtained following a rough approach, starting with computing the total power/area, given by the sum of the results obtained in Figures 25 and 26 and by dividing the LiM power/area by the total ones. As it is possible to see, for bigger arrays, LiM parts will assume a predominant contribution on the power/area performance. This behavior recalls the need of employing a more accurate LiM model, instead of the discussed one based on flip flops and static logic gates.

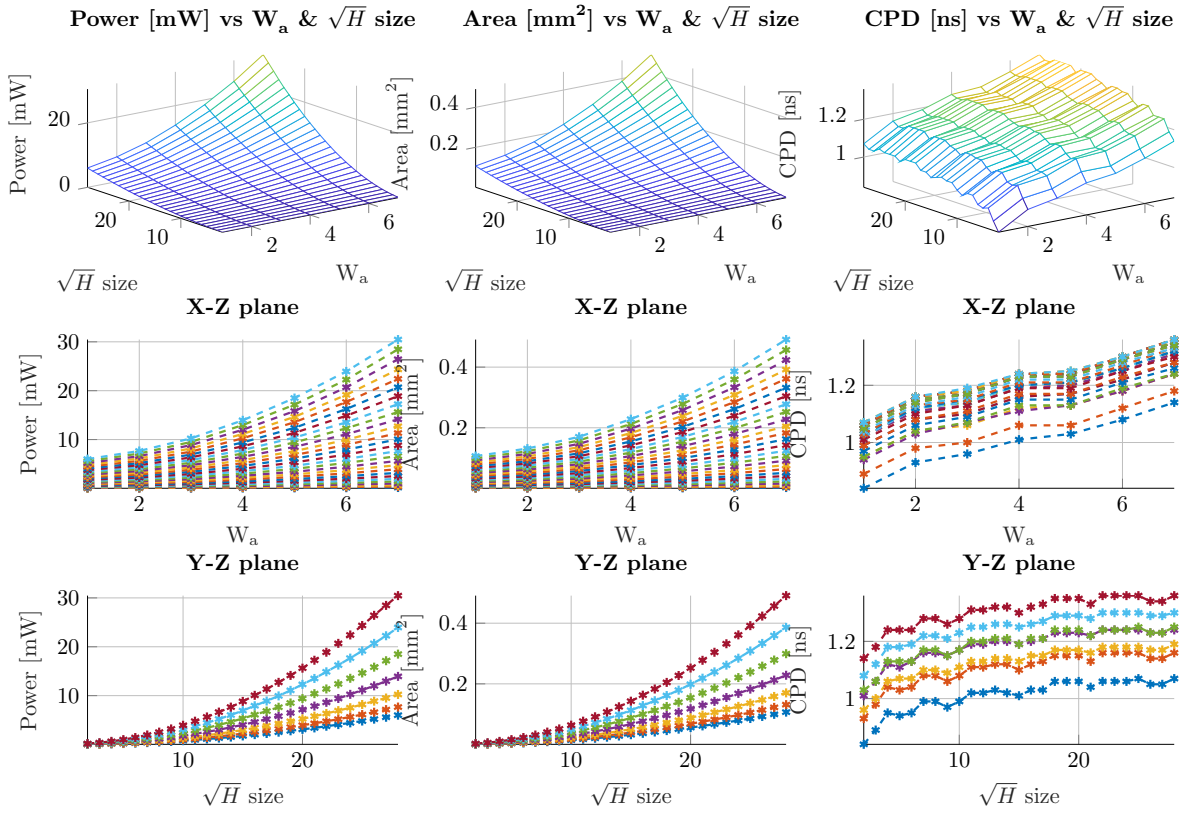

Figure 25. LiM performance estimations by varying W_a and $\sqrt{H}$ sizes. #C_{in} is kept equal to 1.

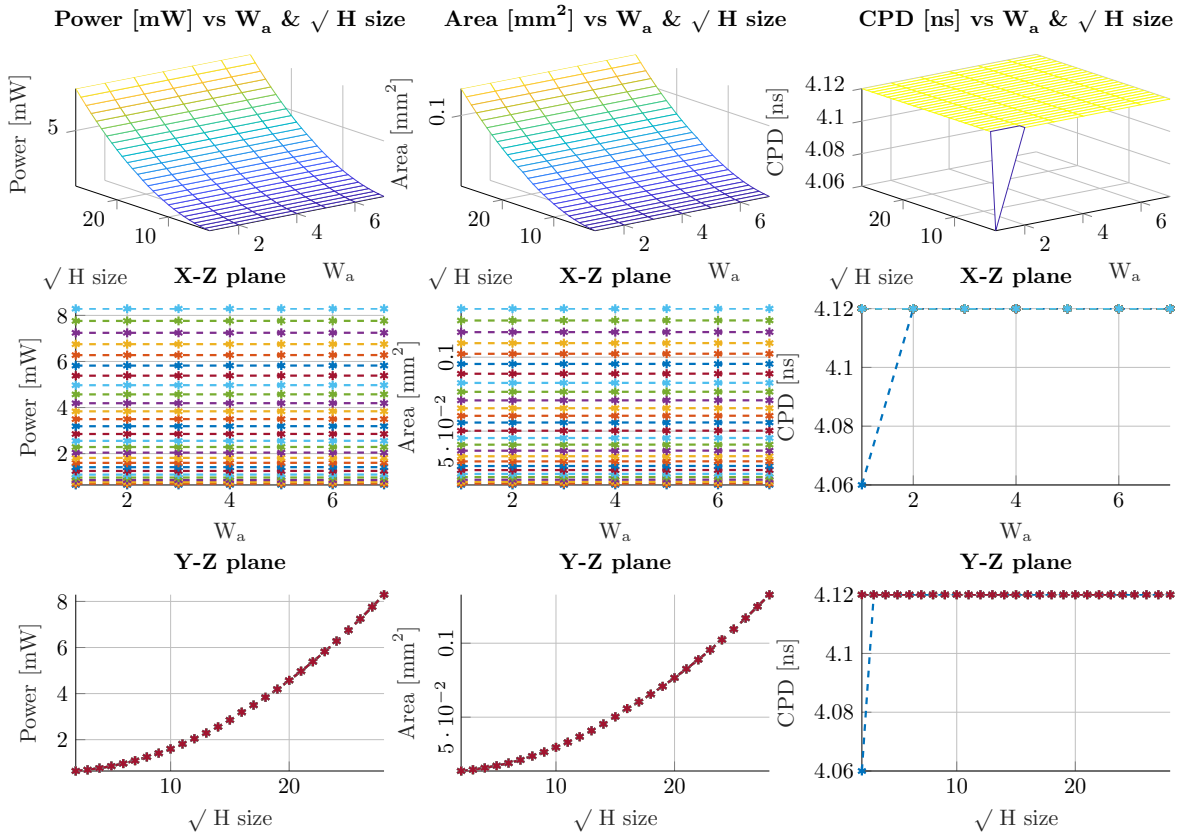

Figure 26. SurroundingLogic unit performance estimations by varying W_a and $\sqrt{H}$ sizes. #C_{in} is kept equal to 1.

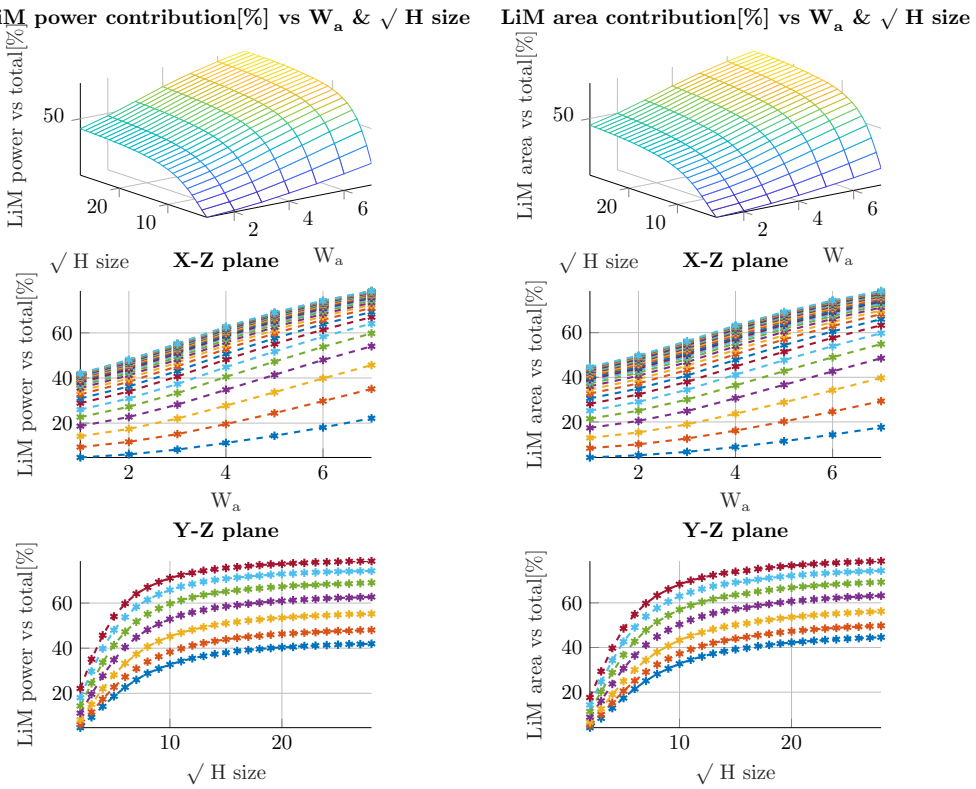

Figure 27. Power and area breakdown of LiM parts.

6.4. A More Detailed LiM Model

Reference [8] proposes a very similar approach, but it performs a Content Addressable Memory (CAM)-based XNOR-Pop procedure, implementing the second convolutional layer of LeNet5 NN model [1], which is depicted in Figure 3. Five arrays are realized and their dimensions are 30×10. They have been implemented with 65nm CMOS technology: the performance results are reported in Table 8. To have a fair comparison with [8], the same conditions have been applied to our LiM design: only the XNOR-Pop part, reported in Figure 15, is synthesized with 65nm technology with a dimension of 30×10 for LiM XNOR part array. To obtain the energy estimation, we started from the power result given by Synopsys and we have mapped the second convolutional layer of LeNet5 CNN, obtaining the corresponding execution time called $\text{Convolution}_{time,\text{II-LeNet5}}$ using the more precise version of Equation (24).

$$\text{Convolution}_{time,\text{II-LeNet5}} = 15852 \times t_{ck} \tag{32}$$

The power obtained by Synopsys is for only 1 LiM array, so the this value has to be multiplied by five:

$$\text{Power}_{\text{5-arrays}} = 0.2473\,\text{mW} \times 5 \approx 1.24\,\text{mW} \tag{33}$$

From the synthesis, CPD for the LiM array is equal to 1.91 ns, so the total energy is:

$$\text{Energy}_{II-LeNet5} = \text{Power}_{\text{5-arrays}} \times \text{Convolution}_{time,\text{II-LeNet5}} \approx 38\,\text{nJ} \tag{34}$$

We can perform a comparison between our less LiM model based on flip flops with the case described in Table 8: the energy ratio between our work and the reference one is about 4.22 while the Bank Area ratio is almost equal to 4.92. This means that, if we design a custom memory, instead of relying on flip flops the performance of our architetcure can be greatly improved. But even considering this fact, the results here presented highlight that LiM architetcures have a huge advantage over traditional Von Neumann circuit, in terms of energy and overall execution speed.

Table 8. CAM-based XNOR-Pop [8] and our LiM architectures performance parameters comparison.

Design	Technology	Bank Size	# of Banks	Bank Area [μm^2]	Energy [nJ]
[8]	65 nm	30×10	5	2456.6	~ 9
This work (LiM)	65 nm	30×10	5	12090.6	~ 38

7. Conclusions and Future Works

In this work LiM and OOM architectures have been designed to demonstrate if a logic-in-memory approach is effectively better than a Von Neumann one in designing architectures for memory-intensive applications. From the results here highlighted, LiM design obtains remarkable results in terms of energy dissipation, because of a higher degree of parallel execution of the algorithm. Since the memory part of our designs was synthesized with Synopsys, the results that we obtained are overestimated, meaning that the energy can be significantly smaller with a proper memory design. We can conclude therefore that Logic-In-Memory architectures are worth it. Even considering the increased complexity of the memory design, they provide significant advantages over Von-Neumann architectures.

As a future work we are designing custom memories, based both and CMOS and eventually on emerging technologies, to further improve our analysis.

Author Contributions: Conceptualization, A.C, M.V. and G.T.; methodology, A.C.; software, A.C.; validation, A.C.; formal analysis, A.C.; investigation, A.C.; resources, A.C.; data curation, A.C.; writing–original draft preparation, A.C.; writing–review and editing, G.T. and M.V.; visualization, M.V.; supervision, M.V.; project administration, M.V. and G.T.

Funding: This research received no external funding

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

MDPI	Multidisciplinary Digital Publishing Institute
LiM	Logic-in-Memory
OOM	Out-Of-Memory
RF	Register File

References

1. LeNet-5, Convolutional Neural Networks. Available online: http://yann.lecun.com/exdb/lenet (accessed 10 January 2020).
2. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*; Neural Information Processing Systems Foundation, Inc.: San Diego, CA, USA, 2012; pp. 1097–1105.
3. Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 1–9.
4. Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv* **2014**, arXiv:1409.1556.
5. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
6. Courbariaux, M.; Bengio, Y.; David, J.P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*; Neural Information Processing Systems Foundation, Inc.: San Diego, CA, USA, 2015; pp.3123–3131.
7. Rastegari, M.; Ordonez, V.; Redmon, J.; Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*; Springer: Berlin, Germany, 2016; pp. 525–542.

8. Choi, W.; Jeong, K.; Choi, K.; Lee, K.; Park, J. Content Addressable Memory Based Binarized Neural Network Accelerator Using Time-domain Signal Processing. In *Proceedings of the 55th Annual Design Automation Conference*; ACM: New York, NY, USA, 2018; DAC '18, pp. 138:1–138:6. doi:10.1145/3195970.3196014.
9. Santoro, G.; Turvani, G.; Graziano, M. New Logic-In-Memory Paradigms: An Architectural and Technological Perspective. *Micromachines* **2019**, *10*, 368.
10. Akin, B.; Franchetti, F.; Hoe, J.C. Data reorganization in memory using 3D-stacked DRAM. *ACM SIGARCH Comput. Architect. News* **2015**, *43*, 131–143. doi:10.1145/2749469.2750397.
11. Durlam, M.; Naji, P.; DeHerrera, M.; Tehrani, S.; Kerszykowski, G.; Kyler, K. Nonvolatile RAM based on magnetic tunnel junction elements. In Proceedings of the 2000 IEEE International Solid-State Circuits Conference, San Francisco, CA, USA, 9 February 2000; pp. 130–131. doi:10.1109/ISSCC.2000.839718.
12. Rakin, A.S.; Angizi, S.; He, Z.; Fan, D. Pim-tgan: A processing-in-memory accelerator for ternary generative adversarial networks. In Proceedings of the 2018 IEEE 36th International Conference on Computer Design (ICCD), Orlando, FL, USA, 7–10 October 2018; pp. 266–273.
13. Roohi, A.; Angizi, S.; Fan, D.; DeMara, R.F. Processing-In-Memory Acceleration of Convolutional Neural Networks for Energy-Efficiency, and Power-Intermittency Resilience. In Proceedings of the 20th International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 6–7 March 2019; pp. 8–13.
14. Wang, H.; Yan, X. Overview of Resistive Random Access Memory (RRAM): Materials, Filament Mechanisms, Performance Optimization, and Prospects. *Phys. Status Solidi (RRL) Rapid Res. Lett.* **2019**, *13*, 1900073, doi:10.1002/pssr.201900073.
15. Krestinskaya, O.; James, A.P. Binary weighted memristive analog deep neural network for near-sensor edge processing. In Proceedings of the 2018 IEEE 18th International Conference on Nanotechnology (IEEE-NANO), Cork, Ireland, 23–26 July 2018; pp. 1–4.
16. Eshraghian, J.K.; Kang, S.M.; Baek, S.; Orchard, G.; Iu, H.H.C.; Lei, W. Analog weights in ReRAM DNN accelerators. In Proceedings of the 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), Hsinchu, Taiwan, 18–20 March 2019; pp. 267–271.
17. Lee, J.; Eshraghian, J.K.; Cho, K.; Eshraghian, K. Adaptive precision cnn accelerator using radix-x parallel connected memristor crossbars. *arXiv* **2019**, arXiv:1906.09395.
18. Roohi, A.; Sheikhfaal, S.; Angizi, S.; Fan, D.; DeMara, R.F. ApGAN: Approximate GAN for Robust Low Energy Learning from Imprecise Components. *IEEE Trans. Comput.* **2019**, doi:10.1109/TC.2019.2949042.
19. Agatonovic-Kustrin, S.; Beresford, R. Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research. *J. Pharm. Biomed. Anal.* **2000**, *22*, 717–727.
20. Nwankpa, C.; Ijomah, W.; Gachagan, A.; Marshall, S. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv* **2018**, arXiv:1811.03378.
21. Wang, Y.; Lin, J.; Wang, Z. An Energy-Efficient Architecture for Binary Weight Convolutional Neural Networks. *IEEE Trans. Very Large Scale Integration (VLSI) Syst.* **2018**, *26*, 280–293. doi:10.1109/TVLSI.2017.2767624.
22. Scherer, D.; Müller, A.; Behnke, S. Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networks*; Springer: Berlin, Germany, 2010; pp. 92–101.
23. Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv* **2015**, arXiv:1502.03167.
24. Sari, E.; Belbahri, M.; Nia, V.P. How Does Batch Normalization Help Binary Training? Available online: http://xxx.lanl.gov/abs/1909.09139 (accessed on 20 December 2019).
25. Whatmough, P.N.; Lee, S.K.; Wei, G.; Brooks, D. Sub-uJ deep neural networks for embedded applications. In Proceedings of the 2017 51st Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, 29 October–1 November 2017; pp. 1912–1915. doi:10.1109/ACSSC.2017.8335697.
26. Pan, Y.; Ouyang, P.; Zhao, Y.; Kang, W.; Yin, S.; Zhang, Y.; Zhao, W.; Wei, S. A Multilevel Cell STT-MRAM-Based Computing In-Memory Accelerator for Binary Convolutional Neural Network. *IEEE Trans. Magnet.* **2018**, *54*, 1–5. doi:10.1109/TMAG.2018.2848625.
27. Fan, D.; Angizi, S. Energy Efficient In-Memory Binary Deep Neural Network Accelerator with Dual-Mode SOT-MRAM. In Proceedings of the 2017 IEEE International Conference on Computer Design (ICCD), Boston, MA, USA, 5–8 November 2017; pp. 609–612. doi:10.1109/ICCD.2017.107.

28. Yonekawa, H.; Sato, S.; Nakahara, H.; Ando, K.; Ueyoshi, K.; Hirose, K.; Orimo, K.; Takamaeda-Yamazaki, S.; Ikebe, M.; Asai, T.; et al. In-memory area-efficient signal streaming processor design for binary neural networks. In Proceedings of the 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS), Boston, MA, USA, 6–9 August 2017; pp. 116–119. doi:10.1109/MWSCAS.2017.8052874.
29. Jiang, L.; Kim, M.; Wen, W.; Wang, D. XNOR-POP: A processing-in-memory architecture for binary Convolutional Neural Networks in Wide-IO2 DRAMs. In Proceedings of the 2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), Taipei, Taiwan, 24–26 July 2017; pp. 1–6. doi:10.1109/ISLPED.2017.8009163.
30. Sun, X.; Yin, S.; Peng, X.; Liu, R.; Seo, J.; Yu, S. XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 1423–1428. doi:10.23919/DATE.2018.8342235.
31. Wang, W.; Li, Y.; Wang, M.; Wang, L.; Liu, Q.; Banerjee, W.; Li, L.; Liu, M. A hardware neural network for handwritten digits recognition using binary RRAM as synaptic weight element. In Proceedings of the 2016 IEEE Silicon Nanoelectronics Workshop (SNW), Dresden, Germany, 19–23 March 2016; pp. 50–51. doi:10.1109/SNW.2016.7577980.
32. Keras. Available online: https://github.com/fchollet/keras (accessed on 20 December 2019).
33. Xiao, H.; Rasul, K.; Vollgraf, R. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* **2017**. arXiv:1708.07747.
34. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).