

# A Configurable BNN ASIC using a Network of Programmable Threshold Logic Standard Cells

Ankit Wagle\*, Sunil Khatri†, Sarma Vrudhula\*

\* School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe AZ

† Dept. of Electrical and Computer Engineering, Texas A&M University, College Station TX

**Abstract**—This paper presents TULIP, a new architecture for a binary neural network (BNN) that uses an optimal schedule for executing the operations of an arbitrary BNN. It was constructed with the goal of maximizing energy efficiency per classification. At the top-level, TULIP consists of a collection of *unique* processing elements (TULIP-PEs) that are organized in a SIMD fashion. Each TULIP-PE consists of a small network of *binary neurons*, and a small amount of local memory per neuron. The unique aspect of the binary neuron is that it is implemented as a *mixed-signal* circuit that natively performs the inner-product and thresholding operation of an artificial binary neuron. Moreover, the binary neuron, which is implemented as a single CMOS standard cell, is reconfigurable, and with a change in a single parameter, can implement all standard operations involved in a BNN. We present novel algorithms for mapping arbitrary nodes of a BNN onto the TULIP-PEs. TULIP was implemented as an ASIC in TSMC 40nm-LP technology. To provide a fair comparison, a recently reported BNN that employs a conventional MAC-based arithmetic processor was also implemented in the same technology. The results show that TULIP is consistently 3X more energy-efficient than the conventional design, without any penalty in performance, area, or accuracy.

**Index Terms**—Threshold logic, BNN, reconfigurable, high-performance, area-efficient, energy-efficient, high-throughput

## I. INTRODUCTION

Convolutional (Deep) Neural Networks (CNNs or DNNs) have become a dominant algorithmic framework in machine learning due to their remarkable success in many diverse applications [1]–[5], etc., and even performing better than humans in some situations [6]. DNNs are now being applied to domains that require compute-intensive operations performed on very large data sets, using models with millions of parameters [7]. Consequently, extensive ongoing efforts are being made to improve their performance and energy efficiency.

Regardless of the hardware platform (CPU-GPU, FPGA, or ASIC) on which DNNs are deployed, the biggest challenge toward improving their performance and energy efficiency has been the on-chip storage requirement. Cost and yield considerations limit the feasible on-chip storage to be one to two orders of magnitude smaller than what is required by many of the popular DNN models, forcing most of the parameters for even moderate size DNNs to be stored in off-chip DRAMs. This results in a large energy ( $> 200X$ ) and delay ( $> 10X$ ) penalties [8]. This has accelerated efforts to drastically reduce the DRAM storage requirements and the associated access delays. Some well-known methods include weight and synapse pruning, quantization (i.e. reducing bit widths of inputs and

weight), weight sharing, Huffman coding, and approximate arithmetic, to name a few.

Quantization remains the most effective technique to reduce memory requirements and computation latency. An extreme form of quantization is to replace the weights and data by binary values, which results in drastic reductions in both storage requirements and computational latency. The resulting networks, known as binary neural networks (BNNs) [9] have been shown to have nearly the same accuracy as DNNs on some small networks (MNIST, SVHN, and CIFAR10) [9], and similar accuracy to that of larger networks (AlexNet, GoogLeNet, ResNet) [10].

BNNs provide a good tradeoff between reduced energy consumption and improved performance against classification accuracy. As a result, they have generated sustained interest in the machine learning community, among researchers in VLSI architecture, circuits, and CAD, and leading FPGA companies (i.e., Xilinx and Intel) [11], [12].

A DNN is a directed acyclic graph (DAG), in which the nodes represent operations such as matrix-vector products, thresholding applied to inner-products, computation of the maximum of vectors, etc. In BNNs, such computations can be implemented almost entirely with binary operations. This makes FPGAs a particularly practical choice for implementing BNNs. Dedicated modules for each operation can be added to the design based on layer-specific requirements. These modules can be pipelined to maximize the throughput of the design. This approach amounts to mapping the nodes of the DAG, layer by layer, to corresponding modules on the FPGA. Often, the entire BNN can be mapped onto the FPGA. Examples of this design strategy, referred to as a *dataflow* architecture, can be found in [11], [13]–[15].

ASIC implementations of BNNs take a different approach. In order to execute any BNN, their basic computational engine consists of a collection of processing elements (PEs), which are comprised of dedicated circuits to perform the operations specific to neural networks such as convolution, max-pooling, RELU, etc. Implementing a BNN on an ASIC next requires scheduling the execution of the nodes of the DAG on the PEs, while optimizing the intermediate storage and accesses to external memory. This approach, referred to as a *loopback* architecture, is the basis of many recent designs, for which examples may be found in [16]–[18].

In this paper, we describe TULIP, a new ASIC architecture to realize BNNs, designed with the aim of maximizing their energy efficiency. Although TULIP falls under the category of a loopback architecture mentioned above, its processing element

\*The research was supported in part by NSF PFI award 1701241.

(TULIP-PE) is radically different from the existing BNN accelerators, which leads to new algorithms to map BNNs onto TULIP. The key features of TULIP and the contributions of this paper are summarized below.

- 1) TULIP is a scalable SIMD machine, consisting of a collection of concurrently executing TULIP-PEs.
- 2) In addition to the design of TULIP, this paper describes a new approach to map any BNN (any number of nodes and nodes with arbitrary fanin) onto TULIP.
- 3) The architecture of TULIP-PE is radically different compared to the PEs in other BNN accelerators. It consists of a small, fully connected network of *binary neurons* each with a small, fixed fanin. A binary neuron is implemented as a *mixed-signal* circuit that natively computes the inner-product and threshold operation of a neuron. The mixed-signal binary neuron is implemented as a single *standard cell*, that is just a little larger than a conventional flipflop. Moreover, the mixed-signal binary neuron is easily configured to perform *all* the primitive operations required in a BNN. By suitably applying control inputs, a TULIP-PE can be configured to perform all the operations required in a BNN, namely the accumulation of partial sums, comparison, max-pooling, and RELU operations. Hence, exactly one such cell is needed to implement all necessary primitive functions in a BNN.
- 4) Because the binary neurons within a TULIP-PE have a fixed fanin, the function of a binary neuron with an arbitrarily large fanin has to be decomposed into a sequence of operations that have to be scheduled on the TULIP-PE. A novel scheduling algorithm for this purpose is described.
- 5) Due to the small area and delay of a single TULIP-PE, several of these can be used within the same area that is occupied by a conventional MAC, and they can be operated in parallel. This, combined with the uniformity of the computation at the individual node and network levels, leads to significant improvement in energy efficiency, without sacrificing the area or performance.

The paper is organized as follows: Section II describes a *generic* architecture of a binary neuron, which is commonly referred to as a *threshold gate*. Here, only the key characteristics of such an element are described and the details of the circuit design are omitted. There are several recent publications describing the architecture of a threshold logic gate [19]–[21], any one of which would be suitable for TULIP. Section III shows how the function of an arbitrarily large binary neuron can be efficiently decomposed into a computation tree consisting of smaller binary neurons that are mapped to the PEs of TULIP. Section IV describes how the novel PE is constructed and how it can be reconfigured to perform the various operations of the BNN. Section V compares the throughput, power, and area of TULIP against the state of the art approaches. Finally, Section VI concludes this paper.

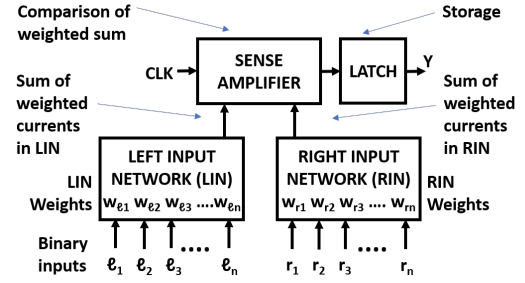


Fig. 1: Threshold Logic Gate (Neuron) Architecture

## II. BACKGROUND TO BINARY NEURONS

A Boolean function  $f(x_1, x_2, \dots, x_n)$  is called a threshold function if there exist weights  $w_i$  for  $i = 1, 2, \dots, n$  and a threshold  $T^1$  such that

$$f(x_1, x_2, \dots, x_n) = 1 \Leftrightarrow \sum_{i=1}^n w_i x_i \geq T, \quad (1)$$

where  $\sum$  denotes the arithmetic sum. Thus a threshold function can be represented as  $(W, T) = [w_1, w_2, \dots, w_n; T]$ . An example of a threshold function is  $f(a, b, c, d) = ab \vee ac \vee ad \vee bcd$ , with  $[w_1, w_2, w_3, w_4; T] = [2, 1, 1, 1; 3]$ . Threshold logic was first introduced by McCulloch and Pitts [23] in 1943 as a simple model of an artificial neuron. Since then, there has been an extensive body of work exploring the many theoretical and practical aspects of threshold logic [22]. The recent resurgence of interest in neural networks has rekindled interest in threshold logic and its circuit realizations. A binary neuron is a threshold logic gate, and is therefore a circuit that realizes a threshold function. Although there exist conventional static CMOS logic implementations of threshold functions, we do not use them in our work as they are inefficient in performance, power, and area. Instead, the binary neuron we consider in this paper is a mixed-signal implementation in which the defining inequality (Eq. 1) is evaluated by directly comparing some electrical quantity such as charge, voltage or current [19], [24]. Interest in binary neurons continues to grow with new architectures incorporating RRAMs, STT-MTJs, and flash transistors, demonstrating substantial improvements in performance, power, and area compared to their CMOS equivalents [20], [21].

Figure 1 shows an abstract block diagram of a circuit that serves as a template for nearly all the recent mixed-signal implementations of binary neurons [21], [24]. It consists of four components: a left and right input network (LIN and RIN respectively), a sense amplifier, and a latch. The key principle under which it operates is as follows. The outputs of the sense amplifier are differential digital signals, with (1, 0) and (0, 1) setting and resetting the latch respectively. The latch state remains unchanged when its inputs are (0, 0) or (1, 1). The weights  $w_i$  that define the threshold function (Eq. 1) are realized in ways that vary among different implementations [19]–[21], but the common feature of all implementations is that they determine the charge, voltage or current of the LIN

<sup>1</sup> W.L.O.G. the weights  $w_i$  and threshold  $T$  can be integers [22].

and RIN once the inputs are applied. That is, LIN and RIN are designed so that the charge, voltage, or current of the path that  $x_i$  controls will be proportional to  $w_i$ . The inputs  $(x_1, x_2, \dots, x_n)$  of a threshold function are mapped to the inputs of the LIN ( $\ell_1, \ell_2, \dots, \ell_n$ ) and RIN ( $r_1, r_2, \dots, r_n$ ) in *such* a way that for every on-set (off-set) minterm, the charge, voltage or current of the LIN (RIN) reliably exceeds that of the RIN (LIN) causing the sense amplifier to set (reset) the latch. Ensuring that the inputs to the LIN and RIN are applied at a clock edge turns the circuit into a multi-input, edge-triggered flipflop, that computes the Boolean threshold function.

### III. BINARY NEURAL NETWORK USING BINARY NEURONS

A threshold function with a large number of inputs needs to be decomposed into a network (directed acyclic graph or DAG) of threshold functions with bounded fanin, each of which can be directly realized by a binary neuron. Figure 2(a) depicts such a network, in which each layer (level in the DAG) consists of a collection of threshold functions  $f_{ij}$ , where  $i$  is the index of the layer and  $j$  is the index of the function within any layer  $i$ . The conventional approach taken by all recently reported BNN architectures is to *accumulate* the partial sums (i.e. the LHS of inequality 1) using standard digital circuits using multiply and accumulate operations are performed. The final thresholding operation using a conventional binary comparator. This approach does not exploit the underlying special nature of the functions being computed, namely, the fact they are threshold functions. Another possible disadvantage of this approach is that it may use arithmetic operators of maximum width, regardless of how small the results of the partial sums are.<sup>2</sup>

There are two basic approaches to decompose a given threshold function into a network of *bounded fanin* threshold functions, several heuristic approaches [25], [26] view the threshold function as any other logic function, and use existing logic synthesis tools to perform a technology-independent re-synthesis into a traditional logic network. This logic network is searched for subgraphs that are bounded-fanin threshold functions. An exact and more elegant algorithm for this is presented in [27]. It directly constructs a network of bounded-fanin threshold functions, in which each function performs thresholding on partial sums. Unfortunately, both these approaches result in extremely large networks.

The architecture of TULIP combines both the above described approaches in a novel way. Figure 2 depicts the design flow and the main components of TULIP. First, a BNN is expressed as a network of threshold functions  $f_{ij}$  (Figure 2(a)). Next, the LHS sum of each threshold function is decomposed into a tree of adders (Figure 2(b)) of bounded size, and each such adder is realized by the repeated use of one configurable binary neuron (Figure 2(b), see insets). This eliminates the waste incurred by conventional methods of accumulation that use operators of max-width. In Figure 2(b) the labels inside the node show the order in which that node is executed

on a TULIP-PE for a threshold function with 1023 inputs. Note that although accumulation can be implemented by using conventional adders of varying sizes, the key difference with TULIP is that *all* the operations that arise in a BNN (addition, accumulation, comparison, and max-pooling) are implemented by the same, single configurable binary neuron in TULIP.

The main processing element in TULIP (TULIP-PE) consists of a complete network of 4 configurable binary neurons (as shown in Figure 2(c)). The operations in the adder tree, as well as all the other operations in a BNN, are scheduled to be executed on a TULIP-PE so as to minimize the storage required for intermediate results. Each full-adder<sup>3</sup> is implemented as a cascade of two binary neurons (Figure 2(b), left inset). Larger width adders are implemented using a cascade of full adders (Figure 2(b) right inset).

Finally, the top-level structure of TULIP consists of a number of PEs along with image and kernel buffers (Figure 6). TULIP is scalable, i.e., the throughput can simply be increased linearly by adding PEs and using larger image and kernel buffers, without changing the scheduling algorithm.

### IV. TULIP IMPLEMENTATION

TULIP involves the co-design and co-optimization of novel hardware and scheduler optimizations that together perform the operations of the BNN. In this section, the hardware architecture of the TULIP-PE is discussed first. Then the scheduling algorithm needed to perform various operations such as addition, comparison, etc. is discussed. Finally, the top-level architecture is described, which uses an array of TULIP-PEs to realize the entire BNN.

#### A. Hardware architecture of TULIP-PE

A TULIP-PE (Figure 2(c)) has 4 fully connected neurons, referred to as  $N1, \dots, N4$ , each with 16-bit local registers. Each neuron of the TULIP-PE is shown in Figure 3. Inter-neuron communication is implemented using multiplexers as shown in Figure 3. Each neuron has four inputs  $a, b, c$ , and  $d$ , with weights 2, 1, 1, and 1 respectively and a threshold  $T$  that is modified using digital control signals. The number of neurons in each TULIP-PE is determined based on the computational requirements. The minimum number of neurons needed to perform addition, comparison, maxpooling, and RELU was found to be four, and was hence chosen for this paper. All 4 neurons of a TULIP-PE share their inputs  $b$  and  $c$ . This is done so that the neuron can fetch data from its local register, and broadcast it to all other neurons. The local registers are constructed using latches. As opposed to global registers, the local registers allow the neurons to access temporarily stored data faster, and also reduce the power consumption per read/write access.

<sup>3</sup>This can be changed to implement a two-bit or three-bit carry-lookahead addition. Doing so would simply require a binary neuron with a different set of weights, and could increase the throughput at the expense of a small increase in area and power. We plan to address this in future work.

<sup>2</sup>In general, adders of varying width may be utilized.

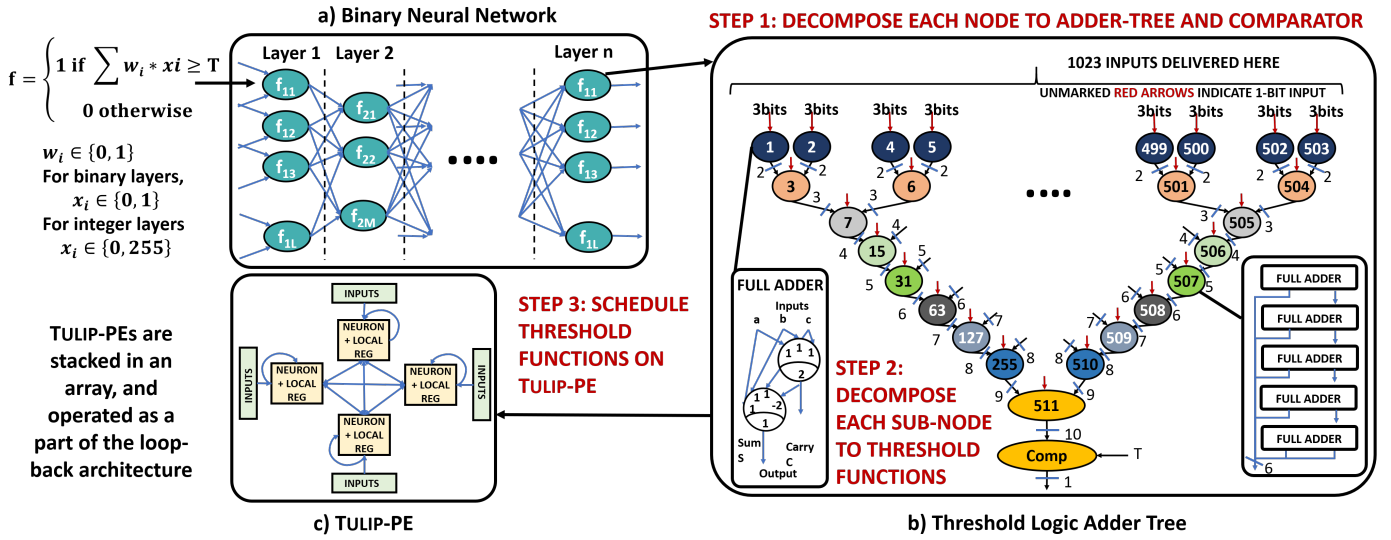


Fig. 2: TULIP Flow: Each node of a BNN is decomposed into an adder-tree. Each sub-node of an adder-tree is decomposed into a network of two-level threshold functions. The decomposed network is scheduled using reverse post-order schedule (Indicated using node numbers; Unmarked red arrows indicate 1-bit input), on a TULIP-PE built using a cluster of four hardware neurons.

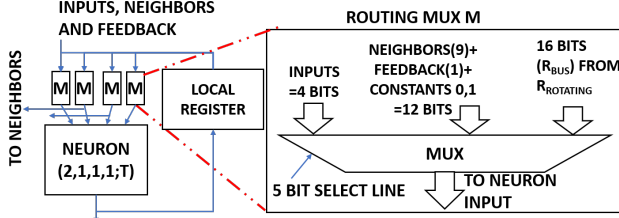


Fig. 3: The hardware neuron and its connections.

### B. Decomposition and scheduling of an adder tree

In this section, we describe how a threshold function  $f_{ij}$  in the BNN (Figure 2(a)) is computed on a single TULIP-PE (Figure 2 (b)). The node  $f_{ij}$  computes the predicate  $S \geq T$ , where  $S = \sum_i w_i x_i$ . The adder tree shown in Figure 2(b) is a binary decomposition of the  $S$  into partial sums, with the leaf nodes (shown at the top) computing the sum of three inputs. The computation of partial sums uses a *reverse post order* (RPO) scheme, which schedules the computation of a sum at a given node after both the sums associated with the left and right subtrees rooted at its left and right nodes have been computed. Therefore the number of bits required for the output of a node is one more than the number of bits of its inputs. In Figure 2(b), the numeric label shown inside a node indicates its position in the RPO. The key property of the RPO is that it minimizes the maximum amount of storage required to store the intermediate results.

Consider the  $N$ -input adder tree shown in Figure 2b. The adder-tree has  $\lfloor \log_2(N) \rfloor$  levels, assuming that the leaf nodes are at level 0. Let  $v$  be a node at level  $i$  in the adder tree, and  $v_l$  and  $v_r$  be its left and right subtrees (both at level  $i-1$ ). Let  $m_i$  denote the maximum storage used for all computations up to and including a node at level  $i$ . Since the node at level  $i$  corresponds to an  $i+1$ -input adder, the storage required for the output of a node at level  $i$  is  $i+2$ . Since, the adder tree is balanced, without the loss of generality, we can assume  $v_l$

is scheduled before  $v_r$ . To compute  $v$ , it is only required to store the output of  $v_l$ , which requires  $i+1$  bits of storage. The maximum storage used to compute  $v_l$  is  $m_{i-1}$ . Hence  $m_i = i+1+m_{i-1}$ , with  $m_0 = 2$ . Therefore,  $m_i = (i^2+3i)/2+2$ . As the highest level is  $\lfloor \log_2 N \rfloor - 1$ , the maximum required storage will be  $(\lfloor \log_2(N) \rfloor^2 + \lfloor \log_2(N) \rfloor)/2 + 1$ . Therefore, an adder-tree has a storage requirement complexity of  $O(\log_2^2(N))$ .

### C. Addition and Accumulation Operation

For a node  $p$  in the adder tree, assume neurons  $N1$  and  $N4$  broadcast two operands from  $R_1$  and  $R_4$ , using the threshold function shown in Figure 4(a) bottom-right inset. Then,  $N2$  and  $N3$  will be used to generate the sum and carry bits of  $p$ , over multiple cycles, using the threshold function shown in Figure 4(a) top-right inset. Since the sum bits are computed on  $N2$ , the final result of  $p$  will be stored in the local register of  $N2$ , i.e.  $R_2$ . Figure 4(a) demonstrates the schedule for 4-bit addition (see node 15 of the adder-tree in Figure 2(b)) using two 4-bit operands  $x$  and  $y$ , i.e.  $\{x_3, x_2, x_1, x_0\}$  and  $\{y_3, y_2, y_1, y_0\}$ . The final result of  $x + y$  is stored in  $R_2$ .

Now, consider nodes  $p$ ,  $q$ , and  $r$  in the adder-tree, as shown in Figure 4(b).  $r$  sums the results of  $p$  and  $q$ . Since the result of  $p$  is stored in  $R_2$ , the result of  $q$  is stored in  $R_3$  to allow simultaneous reading of operands while computing  $r$ .  $r$  reads  $R_2$  and  $R_3$  to generate its sum bits on  $N1$ , and carry on  $N4$ . The memory used by the results of  $p$  and  $q$  can now be freed. Each addition operation stores its result to a specific memory location in the local registers so that the data in the memory is not prematurely overwritten during RPO scheduling.

The adder-tree used in this paper handles up to 10-bit addition on a TULIP-PE. However, this range can be further extended by configuring the TULIP-PE for accumulation. Numbers can be added to an accumulated term stored in the local registers using a multi-cycle addition operation. Figure 4(c) shows the addition of an input number  $p$  with

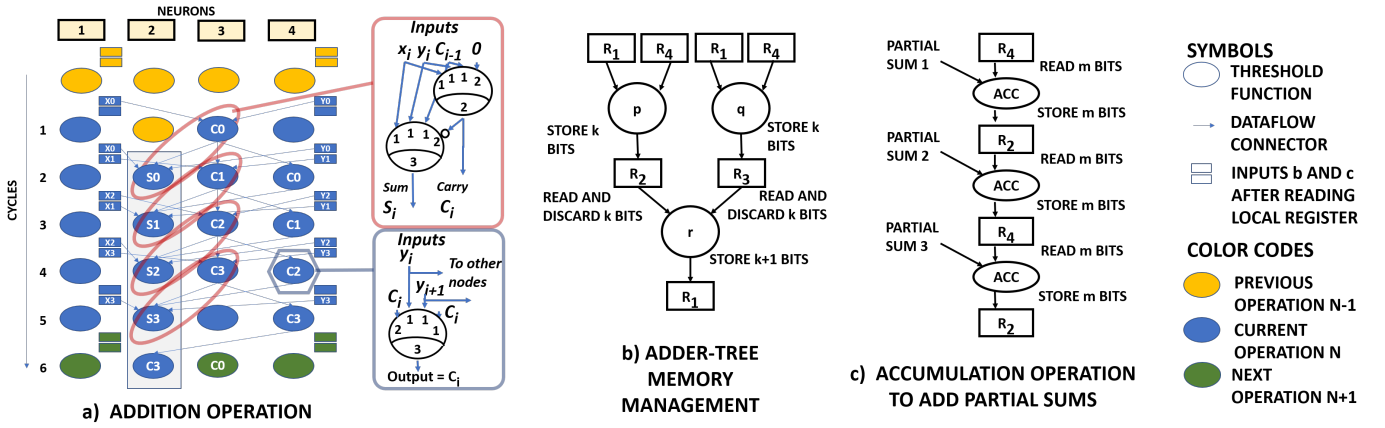


Fig. 4: Adder, Adder-tree and Accumulator Schedule

the accumulated term  $q$ . Since the same local register cannot provide the operands and store the results simultaneously, the storage of  $q$  is alternated between the  $R_2$  and  $R_4$ , for each new accumulation.

#### D. Comparison, Batch Normalization, Maxpooling, RELU Operation

**Comparison:** A multi-cycle sequential comparator is implemented using 3-input threshold functions, as shown in Figure 5(a). To the best of our knowledge, this is the first implementation of a sequential comparator that uses 3-input neurons. Two  $n$ -bit numbers  $x$  and  $y$  that need to be compared are serially delivered from LSB to MSB to the comparator that returns the value of the predicate ( $x > y$ ). In the first cycle, the LSBs of both numbers are compared. In the  $i^{th}$  cycle of the comparison, if  $x_i > y_i$ , then the output is 1, and if  $x_i < y_i$ , then the output is 0. If  $x_i = y_i$ , then the result of the  $(i-1)^{th}$  cycle is retained. The inset in Figure 5(a) shows the logic for bitwise comparison. At the end of  $n$  cycles, the output is 1 if  $x > y$ , and 0 otherwise. The schedule of a 4-bit comparison is shown in Figure 5(a). The 4-bit inputs  $x$  and  $y$  are streamed to the comparator either through the local registers or through the input channels.

**Batch Normalization:** This operation performs biasing of an input value in BNNs. For BNNs, it is realized by subtracting the value of bias from the threshold  $T$  of the binary neuron, as described in [28]. Therefore, batch normalization in TULIP is implemented using the comparison operation.

**Maxpooling:** In a BNN, this operation is an OR operation on a pooling window of layer outputs. This can be implemented using the threshold gate shown in Figure 5(b). Each of the neurons implement one four-input OR function, without the need for local registers. The schedule for this operation requires a single cycle as shown in Figure 5(b).

**RELU:** This implementation of RELU in TULIP is also an extension of the comparator schedule shown above. In RELU, if the input value is greater than threshold  $T$ , then the output gets the value of the input, otherwise, it is 0. This is achieved by ANDing the result of the input value with the comparator's result, using a 2-input threshold function [1,1;2].

#### E. Top Level View of the Architecture

The top-level TULIP architecture is shown in Figure 6. It was designed to deliver high energy efficiency per operation while matching the throughput for the state-of-the-art implementations. This architecture consists of four major types of components: an image buffer, a kernel buffer, one or more processing units, and a controller. The kernel buffer is a shift-register which stores the weights of the BNN. Weights are populated on-chip before the inputs are loaded. The image buffer is a two-stage standard cell memory (SCM) named L2 and L1. Its use reduces off-chip communication (the SCM schedule is based on the technique presented in [17]). In this architecture, 32 input feature maps (IFMs)<sup>4</sup> are loaded on-chip into L2 on a pixel-by-pixel basis. Once L2 is loaded with IFMs, L1 starts fetching the window of IFM pixels needed for the convolution operation, on a window-by-window basis. This window of input pixels is broadcasted to all the processing units present in the design. The processing units are responsible for performing the convolution. These units also receive the appropriate weights from the kernel buffer.

A processing unit only triggers after necessary inputs and weights are received. The inputs and weights are multiplied using XNOR gates, to generate product terms. The processing unit has two components for accumulating the product terms: a MAC unit and eight TULIP-PEs. A TULIP-PE is used to handle an output feature map (OFM) of the binary layers. Although the TULIP-PEs are capable of handling the integer layers as well, it would result in reduced throughput. This is because the TULIP-PEs require several cycles for integer operations, which becomes progressively worse as the size of the operands increase. Hence, MACs are used for integer layers.

The controllers used in the MAC units are simple counters. However, for the TULIP-PEs, a reconfigurable sequence generator is used. This sequence generator follows the RPO schedule, and controls the local registers and the multiplexers of the TULIP-PEs. The control signals are broadcast to all the processing units. The design of the controller is simple and has a negligible impact on the area and power of the overall TULIP architecture. The TULIP architecture also incorporates

<sup>4</sup>Memory can be scaled to store fewer or more IFMs

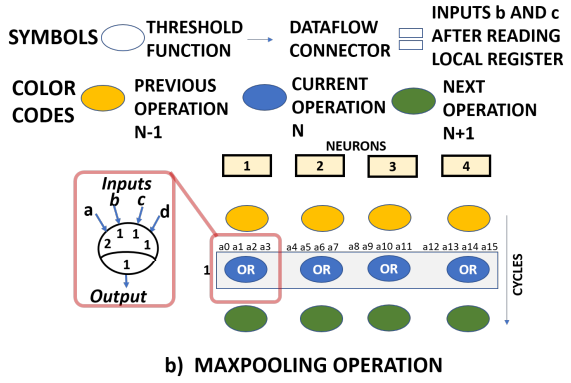
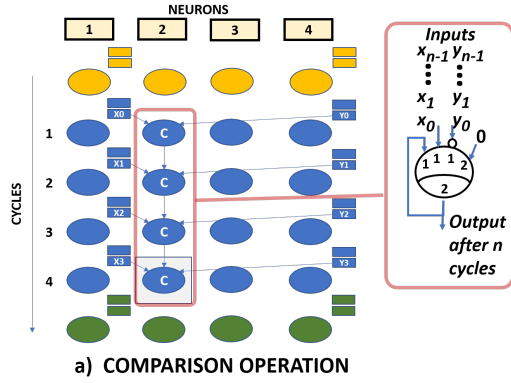


Fig. 5: Comparator and Maxpooling Schedule

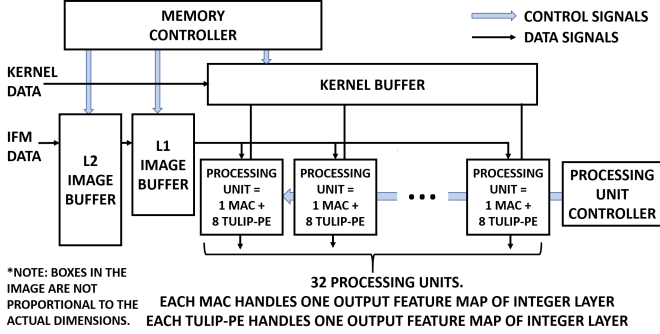


Fig. 6: TULIP Top Level Architecture: Controller configures the processing units. Memory channels the input pixels and weights through image and kernel buffers. The output of the processing units is collected in the output buffers before sending back to the memory.

	Hardware Neuron [21]	Logical Equivalent	X Improve
Area ( $\mu m^2$ )	15.6	27	1.8X
Power ( $\mu W$ )	4.46	6.72	1.5X
Worst Delay (ps)	384	697	1.8X

TABLE I: Hardware neuron versus standard cell neuron

a clock gating strategy whenever a part of the design is not used. The necessary clock gating signals are also generated by the controller.

Although the TULIP architecture locks its configuration to a specific set of components for delivering weights and inputs, it can easily be tailored for a given application. For example, if a BNN does not have integer layers, then the MAC units can be removed, and the multi-bit input buffers can be trimmed to 1-bit input buffers. Various weight and input distribution techniques, such as the one presented in [29] can also be used, by stacking the processing units in a 2-D arrangement instead of a 1-D configuration.

## V. EXPERIMENTAL RESULTS

### A. Experimental Setup

The TULIP architecture was built based on the hardware neuron described in [21]. The neuron was re-implemented in a 40nm technology, programmed to [2,1,1,1;T], and characterized across corners ( $SS$  0.81V 125°C,  $TT$  0.9V 25°C and  $FF$  0.99V 0°C). The value of T is switched during run-

Single PE Metrics	YodaNN MAC (B)	TULIP-PE (T)	Ratio(X) (B/T)
Area( $\mu m^2$ )	3.54E+04	1.53E+03	23.18
Power(mW)	7.17	0.12	59.75
Cycles	17	441	0.038
Time period(ns)	2300	2300	1
Time(ns)	39	1014	0.038

TABLE II: Comparison of fully reconfigurable MAC unit [17] with a TULIP-PE, for a 288 input neuron (Kernel =3x3)

time by changing the appropriate control signals of the neuron. Table I demonstrates that this hardware neuron is substantially better than its conventional CMOS standard cell equivalent in terms of area, power, and delay. This is significant since TULIP uses this neuron for all operations (computation of partial sums, comparison, RELU, and maxpool). TULIP was synthesized and placed using TSMC 40nm-LP standard cells with Cadence Genus<sup>®</sup> and Innovus<sup>®</sup> (Figure 7). The VCD file generated using real BNN workloads was used for power analysis, to model switching activity accurately.

We compare TULIP with a recent BNN design named YodaNN [17] which was designed in 65nm UMC technology. To make a fair comparison, we implemented the entire YodaNN design in the same technology as TULIP(40nm-LP from TSMC), and synthesized, placed and routed both the designs. Both the TULIP and YodaNN were designed for up to 12-bit inputs, with binary weights. Therefore, for YodaNN, we added clock gating for 11/12 input bits when binary layers are evaluated. There are other ASIC architectures available, such as XNORBIN [16], which use more advanced memory techniques to improve energy efficiency. However, these architectures do not support integer layers and are therefore not suitable for comparison. Although [17] does not report the throughput and energy efficiency for fully connected layers, we estimate the throughput and power by performing an element-wise matrix multiplication using the MAC units present in their architecture.

### B. Evaluation of TULIP-PE against MAC

In Table II, the 15-bit reconfigurable MAC unit based on the design present in YodaNN [17] is compared against the TULIP-PE module. The MAC unit used in YodaNN is capable of handling 3x3, 5x5 and 7x7 kernel sizes. Note that both

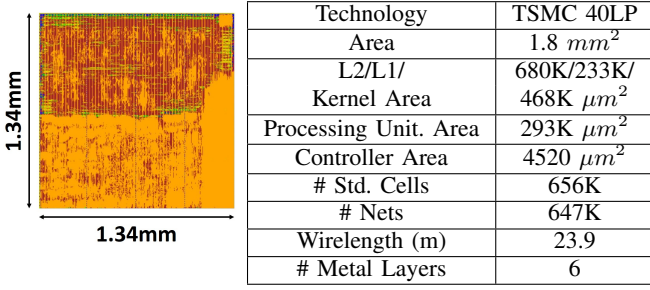


Fig. 7: Layout of TULIP Architecture in TSMC 40nm-LP

the MAC unit and TULIP-PE are capable of handling integer inputs and binary weights. In large BNN architectures such as Alexnet [30], the initial layers are integer layers, while the rest of the layers are binary. YodaNN uses MAC units for all layers while TULIP uses TULIP-PEs for binary layers and simplified MACs (which support only 5x5 and 7x7 kernel windows) for integer layers. Since the computation technique between YodaNN and TULIP differs only for binary layers, the comparison of the MAC and the TULIP is done for the binary layers. That is, both modules perform the weighted sum for binary activations and binary weights of 288 inputs, i.e. 3x3 kernel for 32 IFMs. Based on the Table II, we note that the TULIP-PE is 23.18X smaller than the MAC unit and consumes 60X less power. However, it consumes 27X more time as compared to the MAC unit, since it performs bit-level addition. The power delay product of a TULIP-PE is 2.27X lower than the MAC unit, while at the same time being 23X smaller than the MAC.

The use of an adder-tree based schedule helps the TULIP-PEs deliver a better power-delay product than a conventional MAC unit. Furthermore, since a MAC unit is not capable of operations such as comparison, maxpooling, etc., the data is sent to other parts of the chip for these operations in [17]. However, the TULIP-PE, is capable of preserving the data locality and can perform the comparison and max-pooling operations internally, without the need to move the data to other modules, which saves additional energy.

### C. Evaluation of the TULIP Architecture

The following notation is used for evaluating the TULIP architecture. For 2-D convolution, let  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$  denote the dimensions of the IFMs and OFMs respectively. Let the kernel window size be  $(k \times k)$ .

The number of processing units in TULIP can be scaled to suit the application. However, for the sake of evaluation, TULIP was designed with 32 simplified MAC units and 256 TULIP-PEs, to ensure that the chip area of Tulip matches that of YodaNN. Note that the simplified MAC unit is not reconfigurable, and hence consumes significantly lower area and power than the MAC presented in YodaNN. Therefore, for TULIP, convolution is done in batches of 32 OFMs at a time for integer layers, and 256 OFMs at a time for binary layers. Since the IFMs are re-fetched for each batch of OFMs, they are fetched  $Z = z_2/32$  times for integer layers and  $z_2/256$  times for binary layers. The YodaNN architecture uses 32

Convolution Layers	Parts	YodaNN			TULIP		
		P	Z	P*Z	P	Z	P*Z
1 (Integer)	4	1	3	3	1	3	3
2 (Integer)	1	2	8	16	2	8	16
3 (Binary)	1	4	12	48	8	2	16
4 (Binary)	1	6	12	72	12	2	24
5 (Binary)	1	6	8	48	12	1	12

TABLE III: Effect of input fetch requirements based on Alexnet layers for YodaNN and TULIP. P: Number of times partial products are computed. Z: Number of times inputs are fetched into L2 and L1 buffers for OFM calculation.

fully reconfigurable MAC units, and occupies the same area as TULIP. Therefore, the number of times YodaNN fetches IFMs ( $Z$ ) =  $z_2/32$ . Additionally, when the kernel size is small ( $k \leq 5$ ), the MAC units in both the designs can fetch twice the number of IFMs. Since the TULIP can initiate more OFMs for binary layers, it significantly reduces the number of times an input needs to be fetched. For this paper, both the YodaNN and TULIP architecture load 32 IFMs at a time on-chip. This specification can however be changed to meet the application requirements. If the total IFMs cannot fit on-chip, the OFMs are generated in pieces of P partial results. These partial results are later accumulated on-chip to generate the final OFM. For both the architectures in this paper,  $P = z_1/32$ . The total number of operations is counted by considering addition and multiplication separately. For a 2-D convolution layer, the total multiply and accumulate operations in TULIP are  $2z_1k^2x_2y_2z_2$ , and for comparison of each accumulated sum with T, it is  $x_2y_2z_2$ . For Alexnet, Table III compares the number of times the inputs need to be refetched (Z), and the number of times the P partial products need to be computed for both YodaNN and TULIP. Since both the designs use MAC units for integer layers, there is no difference in both P and Z. However, for binary layers, TULIP demonstrates 3X to 4X improvement in overall input-refetch (indicated by  $P \times Z$ ) as compared to the YodaNN architecture.

Table IV and Table V compare the characteristics of YodaNN with TULIP. Table IV presents the results for the convolution layers and Table V presents the results for the entire BNN. The TULIP architecture outperforms YodaNN in energy efficiency by about 3X for the convolution layers. This is due to the combined use of adder-tree based schedule, coupled with clock gating. The energy efficiency also increases due to better input re-use, which allows the throughput to improve slightly. Considering all layers, TULIP's energy efficiency is 2.4X better than YodaNN. This is because memory consumes significantly more energy than the processing units when executing fully connected layers, which slightly diminishes the energy efficiency achieved in the convolution layers. The results also show that the gains are consistent across different neural networks.

## VI. CONCLUSION

This paper is the first implementation of TULIP, a BNN accelerator that uses current-mode binary neurons, and demonstrates up to 3X improvement in energy efficiency against a

Conv only	BinaryNet		AlexNet	
Dataset	CIFAR10		Imagenet	
	YodaNN	TULIP (X)	YodaNN	TULIP (X)
Op.(MOp)	1017	1017 (1.0)	2050	2050 (1.0)
Perf.(GOp/s)	47.6	49.5 (1.0)	72.9	79.1 (1.1)
Energy(uJ)	472.6	159.1 (3.0)	678.8	224.5 (3.0)
Time(ms)	21.4	20.6 (1.0)	28.1	25.9 (1.1)
En.Eff. (TOP/s/W)	2.2	6.4 (3.0)	3.0	9.1 (3.0)

TABLE IV: Comparison of YodaNN with TULIP architecture for accelerating convolution layers of standard datasets.

All Layers	BinaryNet		AlexNet	
Dataset	CIFAR10		Imagenet	
	YodaNN	TULIP (X)	YodaNN	TULIP (X)
Op.(MOp)	1036	1036 (1.0)	2168	2168 (1.0)
Perf.(GOp/s)	37.7	35.8 (0.9)	12.3	13.1 (1.1)
Energy(uJ)	495.2	183.9 (2.7)	1013.3	427.5 (2.4)
Time(ms)	27.5	28.9 (0.9)	176.8	165.0 (1.1)
En.Eff. (TOP/s/W)	2.1	5.6 (2.7)	2.1	5.1 (2.4)

TABLE V: Comparison of YodaNN with TULIP for accelerating entire BNNs of standard datasets

state of the art BNN hardware accelerator [17], without using the standard low power techniques such as voltage scaling and approximate computing. The TULIP design uses the same area as [17], and slightly improves the throughput. The gains are achieved because TULIP uses an adder-tree based schedule, instead of an accumulator. The gains are further boosted through the use of processing elements (TULIP-PEs) built using a special arrangement of hardware neurons. These TULIP-PEs have extremely low area and power footprint compared to the existing realizations of the same function. As a result, TULIP can deploy an order of magnitude more PEs as compared to a MAC-based architecture for the same chip area.

## REFERENCES

- [1] G. Hinton, L. Deng, D. Yu, G. Dahl, A.R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, and T. Sainath. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29:82–97, November 2012.
- [2] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [3] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.
- [4] K.J. Hunt, D. Sbarbaro, R. Zbikowski, and P. Gawthrop. Neural networks for control systems survey. *Automatica*, 28(6):1083 – 1112, 1992.
- [5] Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.
- [6] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.
- [7] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. 2014.
- [8] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In *Proceedings of the 2017 ACM/SIGDA, FPGA '17*, pages 5–14, New York, NY, USA, 2017. ACM.
- [9] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and . Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. 2016.
- [10] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.
- [11] Yaman Umuroglu, 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, FPGA '17*, pages 65–74, New York, NY, USA, 2017. Association for Computing Machinery.
- [12] A. G. Anderson and C. P. Berg. The high-dimensional geometry of binary neural networks. *CoRR*, abs/1705.07199, 2017.
- [13] Y. Li, Z. Liu, W. Liu, Y. Jiang, Y. Wang, W. L. Goh, H. Yu, and F. Ren. A 34-FPS 698-GOP/s/W Binarized Deep Neural Network-Based Natural Scene Text Interpretation Accelerator for Mobile Edge Computing. *IEEE Transactions on Industrial Electronics*, 66(9):7407–7416, 2019.
- [14] X. Sun, S. Yin, X. Peng, R. Liu, J. Seo, and S. Yu. XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1423–1428, 2018.
- [15] T. Geng, T. Wang, C. Wu, C. Yang, S. L. Song, A. Li, and M. Herbordt. LP-BNN: Ultra-low-Latency BNN Inference with Layer Parallelism. In *2019 IEEE ASAP*, volume 2160-052X, pages 9–16, 2019.
- [16] A. Al Bahou, G. Karunaratne, R. Andri, L. Cavigelli, and L. Benini. XNORBIN: A 95 TOP/s/W hardware accelerator for binary convolutional neural networks. In *2018 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pages 1–3, 2018.
- [17] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP:1–1, March 2017.
- [18] H. Nakahara, H. Yonekawa, T. Sasao, H. Iwamoto, and M. Motomura. A memory-based realization of a binarized deep convolutional neural network. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 277–280, 2016.
- [19] S. Bobba and I. Hajj. Current-Mode Threshold Logic Gates. In *Proc. of ICCD*, pages 235–240, 2000.
- [20] S. N. Mozaffari and S. Tragoudas. Maximizing the number of threshold logic functions using resistive memory. *IEEE TNANO*, 17(5), Sep. 2018.
- [21] A. Wagle, G. Singh, J. Yang, S. Khatri, and S. Vrudhula. Threshold logic in a flash. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 550–558, 2019.
- [22] S. Muroga. *Threshold Logic and its Applications*. Wiley-Interscience New York, 1971.
- [23] W. S. McCulloch and W. Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity*. MIT Press, Cambridge, MA, USA, 1988.
- [24] S. Leshner, B. Krzysztof, and S. Vrudhula. Design of a robust, high performance standard cell threshold logic family for deep sub-micron technology. In *Proceedings of the IEEE International Conference on Microelectronics*, Cairo, Egypt, Dec. 19–22 2010.
- [25] A. Neutzling, J. M. Matos, A. I. Reis, R. P. Ribas, and A. Mishchenko. Threshold logic synthesis based on cut pruning. In *IEEE/ACM ICCAD*, Nov 2015.
- [26] N. Kulkarni, J. Yang, J. S. Seo, and S. Vrudhula. Reducing Power, Leakage, and Area of Standard-Cell ASICs Using Threshold Logic Flip-Flops. *IEEE TVLSI*, 24(9), Sept 2016.
- [27] Viswanath Annampedu and Meghanad D. Wagh. Decomposition of threshold functions into bounded fan-in threshold functions. *Information and Computation*, 227:84–101, 2013.
- [28] Taylor Simons and Dah-Jye Lee. A review of binarized neural networks. *Electronics*, 8(6):661, 2019.
- [29] Y. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE ISCA*, pages 367–379, 2016.
- [30] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. volume 9908, pages 525–542, 10 2016.