Efficient FPGA Implementation of Binarized Neural Networks Based on Generalized Parallel Counter Tree

Takahiro TANIGAWA † Mugi NODA † Nagisa ISHIURA ††

† Graduate School of Science and Technology †† School of Engineering
Kwansei Gakuin University
1 Gakuen Uegahara, Sanda, Hyogo, 669-1330, JAPAN

Abstract—Binarized neural networks (BNN) allow compact hardware implementation by binarizing weight values and neuron activations. The critical path delay of a combinational circuit implementing a BNN neuron may be curbed by adopting a Wallace tree of full-adders. However, in FPGA implementation, a 3-input full-adder does not make full use of LUTs of more than 5 inputs. This paper proposes the use of a GPC (generalized parallel counter) based compressor tree in FPGA implementation of a BNN neuron to reduce both the delay and size of the resulting circuit. We further enhance the efficiency of the circuit by reducing the comparison of the pop-count and threshould into reference to the carry signal from the compressor tree. The critical path delay and the slice count of our BNN neuron, implemented on a Xilinx Artinx-7 FPGA, were smaller by 15.1% and 11.1%, respectively, compared to those of the circuit produced by regular logic synthesis, at number of inputs 1024.

I. Introduction

Neural networks are a powerful tool for image and audio recognition, and they are being used in a wide range of applications.

With the recent development of network technologies, various data are collected from mobile or edge devices. Data obtained on the edge side is often transferred to the server side for processing, for computational load of inference in neural network is high. However, from the perspective of communication data volume and security, attempts have been made to shift this processing to the edge side. To achieve processing on edge devices with limited computational resources and strict power consumption constraints, research is being conducted on implementing part or all of the inference processing of neural networks in hardware [1].

A popular approach to reduce resources in the hardware implementation of neural networks is to represent data with fixed-point numbers and to reduce the number of bits. The values may be constrained to binary, and then the neural network is referred to as a Binarized Neural Network (BNN) [2]. The BNN allows multiplication operations to be calculated with a single logic gate, leading to a significant reduction in hardware resources. Thus, research on efficient hardware implementation of BNNs has been conducted in recent years [3].

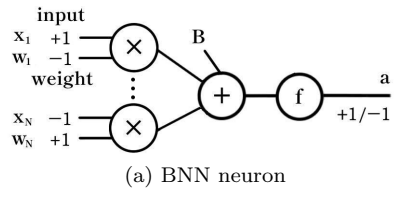
In the hardware implementation of BNN neurons, it can be constructed using XNOR gates to calculate products, a pop counter (parallel counter) to count the number of ones in the result, and a comparator to compare with a threshold. When considering a combinational circuit implementation, XNOR operations can be performed in parallel, so the comparison with the pop counter and threshold becomes the bottleneck in the critical path delay.

A BNN neuron can be constructed using XNOR gates to calculate products, a pop-counter (parallel counter) to count the number of ones in the products, and a comparator to compare the count with a threshold. In the case of combinational circuit implementation, XNOR operations are performed in parallel, so the pop-count and comparison become the bottleneck in the critical path delay.

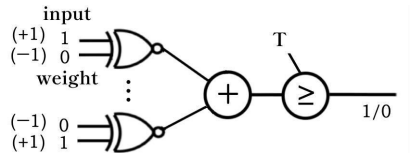
The pop-count can be attributed to multiple-input addition, which can be achieved by constructing Wallace trees [4] or Dadda trees [5] with full adders to reduce the critical path delay. However, when implementing on FPGA devices of the lookup table (LUT) type, a 3-input full adder may not be an ideal building block because the number of inputs to the LUT is typically around 5 to 7. As a measure to fully utilize the LUTs, the use of expanded full-adders, such as 6-input/3-output adders, or further extended generalized parallel counters (GPCs), has been proposed.

The GPC is a parallel counter that allows both inputs with a weight of 1 and inputs with a weight of 2^k. Literatures [6, 7] propose extremely efficient ways of constructing addition trees for FPGA implementation, targeting multiple-input additions in multiplication operations and the like.

This paper presents an efficient FPGA implementation of a BNN neuron utilizing GPCs. The use of GPC tree to construct the pop-counter is proposed to minimize its critical path delay and the circuit size. Furthermore, comparison with the threshold is replaced by a test of the most



(a) BNN neuron



(b) Logic circuit implementation

Fig. 1. BNN neuron and its logic circuit implementation

significant carry output by switching the threshold with a bias, which further reduces the delay and circuit size.

We have implemented BNN neurons on Xilinx's Artix-7 FPGAs based on this method. Compared to neurons generated by simple logic synthesis, our implementation achieved a 11.1% reduction in slice usage and an 15.1% reduction in delay time when the number of inputs to the neuron is 1024.

II. FPGA Implementation of Binarized Neural Networks

The activation value a of a neural network neuron can be expressed as follows, as shown in Fig. 1a :

$$a = f(B + \sum_{i=1}^{N} x_i \cdot w_i),$$

where N is the number of neuron inputs, x_i represents the i-th input, w_i is the weight associated with the i-th input, B is the bias, and f is the activation function.

In the case of a binarized neural network (BNN), the values of a, x_i, and w_i are constrained to be either +1 or -1, and $f(x) = 1$ if $(x \geq 0)$ and $f(x) = -1$ otherwise.

Now, if we encode the values +1 and -1 as 0 and 1, respectively, we can replace the multiplication of inputs and weights with the Exclusive-NOR (XNOR) operation. As a result, a BNN neuron can be constructed with XNOR gates, an adder (pop counter) to count the number of ones in the output, and a comparison with a threshold T, as shown in Figure 1b.

In this paper, we consider implementing a BNN neuron as a combinational circuit, shown in Fig. 1b , on an FPGA.

A. Lookup Tables (LUTs) and Carry Chains in FPGAs

Lookup table (LUT) type FPGAs utilize configurable components (LUTs), flip-flops, and programmable interconnections to implement custom logic functions. LUTs store truth tables in memories to implement logic functions. The typical number of inputs to LUTs ranges from 5 to 7. In some FPGAs, a k-input/1-output LUT may be used to implement a $k-1$-input/2-output logic gate that share common input lines.

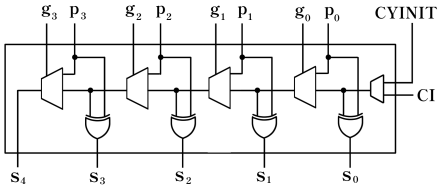


Fig. 2. Example implementaion of carry chain

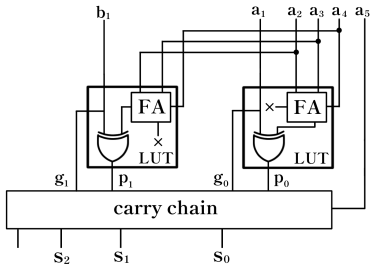


Fig. 3. FPGA-implementation of GPC(1,5;3) [8]

Furthermore, some FPGAs have built-in carry chains which are used for efficient implementation of adders and subtractors. The carry chain accepts the carry generate and carry propagate signals for each digit of two binary numbers as inputs and computes the sum for the binary numbers. An example of a 4-digit carry chain circuit is shown in Figure 2. g_i and p_i represent the carry generate and propagate signals, respectively, while s_i denotes the resultant sum.

A module called a *slice* or a logic block contains several LUTs, carry chains, and flip-flops. Logic circuits are realized using these slices and programmable interconnections.

B. Generalized Parallel Counter (GPC)-Based Addition Tree

Wallace tree [4] and Dadda tree [5] are recognized as established logic circuit designs for multi-input addtion, with both utilizing full-adders as their basic building blocks.

However, in the context of FPGA implementation, a traditional 3-input/2-output full adder fails to utilize the potential of LUTs, which typically offer 5 to 7 inputs. As a consequence, the adoption of extended full-adders, such as 6-input/3-output adders, and more advanced Generalized Parallel Counters (GPC), has been suggested. These strategies are designed to enhance FPGA circuit efficiency in terms of both size and critical path delay [6] [7] [8] [9] [10].

Unlike the full adder or the 6-input/3-output adder, the GPC accepts inputs of weight 2^k as well as 1 and calcu-

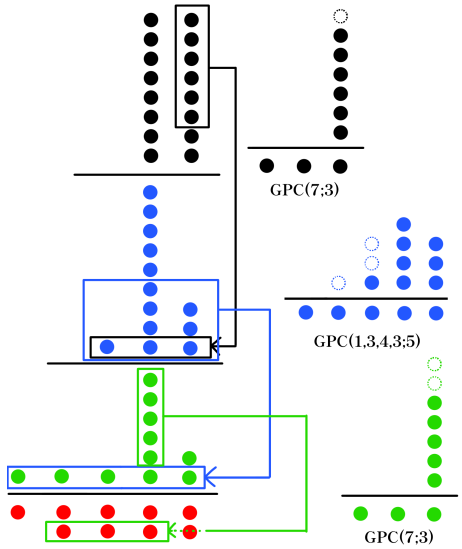


Fig. 4. GPC-based adder tree

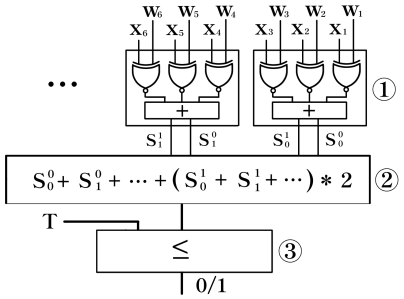


Fig. 5. FPGA-implementation of BNN neuron

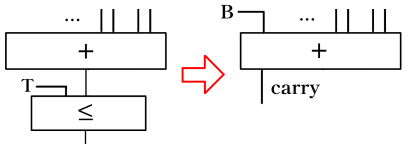


Fig. 6. Replace threshold comparison with carry reference

lates the sum of the inputs. For example, GPC(1,5;3) performs addition of a 1-bit input of weight 2^1 and 5-bit inputs of weight 2^0, and outputs the result in 3 bits. GPC(1,5;3) can be constructed as shown in Fig. 3 , where b_1 is the weight 2^1 input and a_1 through a_5 are weight 2^0 inputs, and (s_2, s_1, s_0) are resulting outputs. It may be accomodated in a single slice with two LUTs and a carry chain.

An example of the GPC-based addition tree is shown in Fig. 4 . Each dot represents a single bit, and the dots enclosed in rectangles represent the inputs and outputs of GPCs. Two types of GPCs, GPC(7;3) and GPC(1,3,4,3;5), are used in this design.

The total number of bits are reduced through GPCs to yield two binary numbers. The final sum is computed by adding the two numbers using a row adder whose details will be shown later. The GPCs reduce the total number of bits until they become two binary numbers. The final sum will be computed by adding these two numbers, whose details will be presented later.

In [6] and [7], methods were introduced for constructing optimal addition trees utilizing GPCs, targeting multiple-input addition in binary multiplication and related applications. They aim to generate circuit configurations with minimal levels, along with the minimal LUT counts or slice counts, for given size and quantity of binary numbers.

III. Efficient FPGA Implementation of BNNs Using GPCs

This paper presents an efficient FPGA-targeted design of a BNN neuron.

It proposes the use of optimal GPC construction for the implementation of a population counter. It also proposes optimation of the circuit design for threshold comparison.

The overall circuit configuration for the BNN neuron assumed in this paper is shown in Fig. 5 . At first, the count of 1's in the outcome of XNOR operations between K pairs of inputs and weights are computed using LUTs, with K being set to 3 in the figure (①). The resulting output is represented in binary format, represented as (s^1, s^0). Then, the sum of these outputs is calculated (②). If the sum equals or exceeds the threshold T T, the output is set to 1; otherwise, it is set to 0 (③).

In this paper, we do not rely on logic synthesis for ② but instead, construct a GPC tree so as to reduce circuit size and critical path delay. Furthermore, the result of ③ is determined only by referencing the carry signal from the MSB of the addition result, by incorporating a bias of $2^n - T$ in the addition, where n is the number of the bits in the sum.

A. Implementation of Pop-counter Using GPC Tree

The configuration of the addition circuit in this paper is illustrated in Fig. 7 . The inputs to the addition tree are N/k binary numbers, with their individual bits represented by dots (the figure depicts the senario for $k = 3$, where each binary number consists of two bits). A GPC tree is employed to reduce the number of bits iteratively, until they converge to two binary numbers. These two binary numbers are then added together to produce the final result. The adder used in the final step is referred to as a "row adder."

The construction of the GPC (Generalized Pop Counters) tree follows the methodology proposed in references [6], [7], [8], [9], and [10]. Given our assumption of implementing BNN neurons as combinational circuits in this paper, our goal is to identify the GPC tree configuration that minimizes the number of slices among the ones with the fewest stages. We formalize this as an optimization

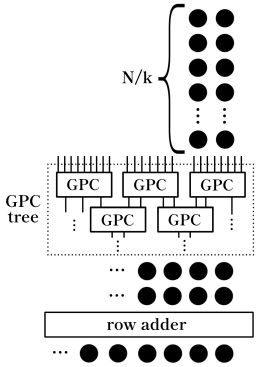


Fig. 7. GPC-based adder tree

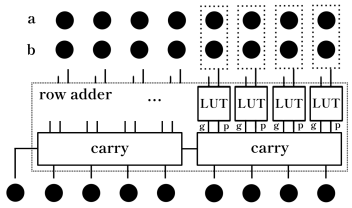


Fig. 8. FPGA-Implementation of row adder

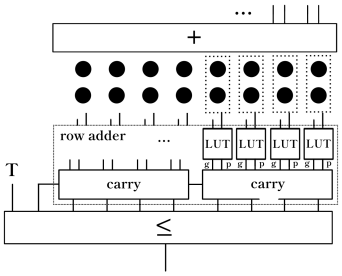
problem using integer linear programming.

The row adder in the last step in Fig. 7 can be efficiently implemented using FPGA carry chains. Fig. 8 shows an example design of a row adder employing a 4-digit carry chain. Let us denote i-th digits of the two binary numbers to be added as a_i and b_i. The *carry generation signal* $g_i = a_i \cdot b_i$ and the *carry propagation signal* $p_i = a_i \oplus b_i$ can be computed using a single 2-input/2-output LUT. These signals are then fed into the carry chain to generate the sum of the two binary numbers.

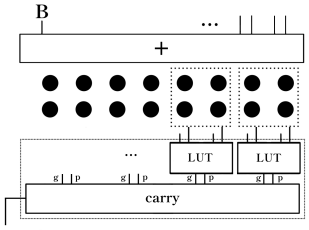
B. Threshold Comparison

Fig. 9 (a) illustrates a straightforward configuration of the comparison circuit along with the row adder. The output from the row adder is compared with a separate comparator. In this paper, we attempt to eliminate the comparator. The proposed circuit configuration is shown in Fig. 9 (b). Instead of comparing the sum against the threshould T, we integrate bias $B = 2^n - T$ into addition (where n is the number of bits of the sum) and test only the carry signal from the MSB.

This approach eliminates the requirement to compute all the individual digits of the sum using the row adder. Instead, it necessitates the generation of generator and propagator signals only for the carry bit originating from the most significant bit (MSB). To achieve this, we can utilize the generator and propagator signals from two adjacent digits (or, alternatively, three adjacent digits) combined. This strategy effectively reduces the necessary number of Look-Up Tables (LUTs), carry chains, and the overall carry propagation delay during the calculation process.

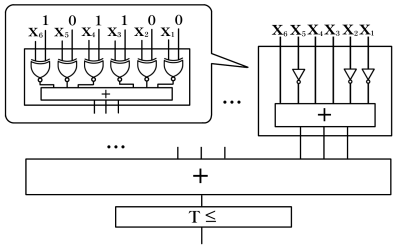


(a) Conventional threshold comparison.

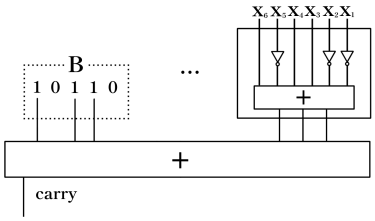


(b) Threshold comparison by carry referencing

Fig. 9. Threshold comparison



(a) Basic configuration



(b) Optimized configuration with our approach

Fig. 10. Parameter-embedded BNN neuron

C. Application to Parameter-Embedded Neuron

When performing repeated inference using the same weights and thresholds, it's possible to embed the parameters of the neuron directly into the circuit rather than supplying them from external memory, which lead to reduction both in circuit size and execution cycles.

The basic configuration of a parameter-embedded neuron for BNN is shown in Fig. 10 (a). The XNOR operation between w_i and x_i is simplified: it becomes x_i when

TABLE I
List of GPCs

GPC [cite]			
(1;1)		(1,4,0,6;5)	[9]
(3;2)		(1,3,2,5;5)	[6]
(7;3)	[9]	(1,3,4,3;5)	[6]
(1,5;3)	[9]	(2,1,3,5;5)	[10]
(2,3;3)		(1,3,5;4)	[6]
(6,2,3;5)	[9]	(2,2,3;4)	
(6,0,6;5)	[9]	(2,0,7;4)	
(6,1,5;5)	[9]	(2,1,5;4)	
(1,4,1,5;5)	[9]		

```
module BASELINE_NEURON_8(
  input wire[7:0]in,
  input wire[7:0]weight,
  input wire[3:0]threshold,
  output wire ans);

  wire [2:0]dst0;
  wire [2:0]dst1;
  wire [3:0] sum;
  LUT6 #(.INIT(64'h9669699669969669))input6_0_lower(
   .O(dst0[0]), .I0(in[0]), .I1(weight[0]), .I2(in[1]),
   .I3(weight[1]), .I4(in[2]), .I5(weight[2]));
  LUT6 #(.INIT(64'h9669699669969669))input6_1_lower(
   .O(dst0[1]), .I0(in[3]), .I1(weight[3]), .I2(in[4]),
   .I3(weight[4]), .I4(in[5]), .I5(weight[5]));
  LUT6 #(.INIT(64'h9669699669969669))input6_2_lower(
   .O(dst0[2]), .I0(in[6]), .I1(weight[6]), .I2(in[7]),
   .I3(weight[7]), .I4(1'h0), .I5(1'h1));
  LUT6 #(.INIT(64'hf99f90099009f99f))input6_0_upper(
   .O(dst1[0]), .I0(in[0]), .I1(weight[0]), .I2(in[1]),
   .I3(weight[1]), .I4(in[2]), .I5(weight[2]));
  LUT6 #(.INIT(64'hf99f90099009f99f))input6_1_upper(
   .O(dst1[1]), .I0(in[3]), .I1(weight[3]), .I2(in[4]),
   .I3(weight[4]), .I4(in[5]), .I5(weight[5]));
  LUT6 #(.INIT(64'hf99f90099009f99f))input6_2_upper(
   .O(dst1[2]), .I0(in[6]), .I1(weight[6]), .I2(in[7]),
   .I3(weight[7]), .I4(1'h0), .I5(1'h1));

  assign sum = dst0[0] + dst0[1] + dst0[2] +
               dst1[0]*2 + dst1[1]*2 + dst1[2]*2;
  assign ans = (sum>=threshold);

endmodule
```

Fig. 11. HDL description of baseline neuron ($N = 8$)

w_i is 1 and $\bar{x}_i$ when w_i is 0. If 6-input LUTs are available, 6 digits of x_i can be processed together to produce a 3-digit sum, which is fed into the paddition tree.

The same methodology of eliminating the comparator can be applied, as illustrated in Figure 10 (b). Moreover, if the bias B contains digits with a value of 0, these digits can be safely disregarded, further reducing the complexity of the addition tree.

IV. Implementation and Experiment

BNN neurons have been designed based on the proposed method in Verilog HDL. The experimental platform was the Xilinx Artix-7 FPGA (xc7a100tcsg324-3). This FPGA is equipped with 6-input/1-output Look-Up Tables (LUTs), which can alternatively function as 5-input/2-output LUTs when input resources are shared. The FPGA contains 15850 slices, each with four LUTs and 4-bit carry chain. Logic synthesis was carried out using Vivado 2023.1 for this experiment.

The set of GPCs utilized in this experiment is detailed in TABLE I . All these GPCs are explicitly designed to require no more than four LUTs and a carry chain, fitting perfectly in a single slice. The configurations of the GPC adder trees, aiming for the lowest levels and minimal slices, were determined through integer programming[1]. We utilized the CPLEX 22.1.0 solver, running on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz. Remarkably, all solutions were found to be optimal and were obtained in less than one second.

The synthesis results are displayed in Table II, with (a) representing the standard scenario where parameters are not embedded, and (b) representing the scenario where parameters are embedded. In both tables, N denotes the number of inputs for the BNN neuron, and "slices" and "delay" represent the number of utilized slices and the critical path delay, respectively. Within each table, "baseline" corresponds to a basic description of the adder tree and threshold comparison. "GPC-tree" signifies the adder tree constructed using GPCs, while "GPC-tree+bias" indicates that the threshold comparison has been replaced with referencing the carry from the MSB. Fig. 11 shows Verilog HDL description of the baseline neuron for $N = 8$. While the addition of three pairs of XNOR operations are explicitly mapped to LUTs, summation and comparison are expressed with standard arithmetic operations. In the "GPC-tree" and "GPC-tree+bias" designs, every GPC is explicitly assigned to a slice to prevent unintended increases in the slice count.

For standard BNNs, a comparison between "baseline" and "GPC-tree" demonstrates that circuit optimization has led to reductions in all circuit sizes. When comparing the performance of "GPC-tree" with that of "GPC-tree+bias," a decrease is observed, except for the case when $N = 512$. This increase can be attributed to the additional cost incurred by the introduction of bias. Nevertheless, it is noteworthy that the delays experienced a consistent decrease across all scenarios. In the case of $N = 1024$, the circuit size and critical path delay were reduced by 11.1% and 15.1%, respectively.

The trends observed for the parameter-embedded BNNs are similar to those of the standard BNNs, but the impact on circuit size is more pronounced in the parameter-embedded BNNs. For instance, with $N = 1024$, the circuit size saw a significant reduction of 20.6%, while the critical path delay decreased by 9.7%.

V. Conclusion

In this paper, we introduce an efficient method for FPGA implementation of Binary Neural Network (BNN)

[1]The programs used for constructing the adder trees and GPCs are available on GitHub at the following links: `https://github.com/void-hoge/cmpgen` `https://github.com/void-hoge/gpcgen`

TABLE II
SYNTHESIS RESULTS

(a) BNN neuron

	N	32	64	128	256	512	1024
slices	baseline	28	69	177	176	352	677
	GPC-tree	22	40	86	160	307	605
	GPC-tree+bias	21	38	78	151	314	602
delay (ns)	baseline	9.93	13.90	21.17	14.41	15.12	16.71
	GPC-tree	9.64	10.24	11.64	13.32	14.99	15.48
	GPC-tree+bias	8.72	8.86	10.43	12.26	14.12	14.18

(b) BNN neuron with parameter embedding

	N	32	64	128	256	512	1024
slices	baseline	16	42	110	102	231	442
	GPC-tree	15	25	48	95	181	364
	GPC-tree+bias	11	22	47	98	197	351
delay (ns)	baseline	7.60	10.95	16.93	12.64	13.70	15.63
	GPC-tree	8.28	9.21	10.72	10.97	13.21	14.82
	GPC-tree+bias	8.81	8.69	10.06	10.94	12.73	14.12

Synthesizer: Xilinx Vivado (2023.1) Target: Xilinx Artix-7(xc7a100tcsg324-3)

neurons, utilizing Generalized Parallel Counters (GPC) to construct the adder trees. Our approach yields notable advantages, achieving a 11.1% reduction in circuit size and a 15.1% reduction in critical path delay compared to straightforward logic synthesis when handling 1024 inputs.

We envision that this proposed method holds promise not only for BNNs but also for FPGA implementations of neural networks employing fixed-point arithmetic. Furthermore, while our focus in this paper is on parallel combinatorial circuit implementation, processing all inputs simultaneously, there are future prospects to explore its applicability in parallel/serial hybrid implementations or pipelined implementations, where inputs undergo processing in multiple stages.

ACKNOWLEDGEMENTS

Authors would like to express their appreciation to Dr. Hiroyuki Kanbara of ASTEM/RI, Prof. Hiroyuki Tomiyama of Ritsumeikan University, and Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their discussion and valuable advises. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University.

REFERENCES

[1] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Hock, Y. Liew, K. Srivatsan, D. Moss, and S. Subhaschandra: "Can FPGAs beat GPUs in accelerating nextgeneration deep neural networks?", in *Proc. FPGA 2017*, pp. 5–14 (Feb. 2017).

[2] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio: "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1," *Computing Research Repository (CoRR)*, pp. 1–11 (Mar. 2016).

[3] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr: "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," in *Proc. International Conference on Field-Programmable Technology (FPT 2017)*, pp. 77–84 (May. 2017).

[4] S. C. Wallace.: "A suggestion for a fast multiplier," in *IEEE Trans. Electronic Computer*, vol. EC-13, issue 1 (Feb. 1964).

[5] L. Dadda.: "Some schemes for parallel multipliers," in *Associazione Elettrotecnica ed Elettronica Italiana*, vol. 34, pp. 349–356 (May 1965).

[6] Y. Yuan, L. Tu, K. Huang, X. Zhang, T. Zhang, D. Chen, and Z. Wang: "Area optimized synthesis of compressor trees on Xilinx FPGAs using generalized parallel counters," in *IEEE Access*, pp. 134815–134827 (Sept. 2019).

[7] M. Kumm and J. Kappauf: "Advanced compressor tree synthesis for FPGAs," in *IEEE Trans. Computers*, pp. 1078–1091 (Jan. 2018).

[8] M. Kumm and P. Zipf: "Efficient high speed compression trees on Xilinx FPGAs," in *Proc. Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pp. 1–13 (Jan. 2014).

[9] T. B. Preußer: "Generic and universal parallel matrix summation with a flexible compression goal for Xilinx FPGAs," in *Proc. International Conference on Field Programmable Logic and Application*, pp. 1–7 (Sept. 2017).

[10] M. Kumm and P. Zipf: "Pipelined compressor tree optimization using integer linear programming," in*Proc. International Comference on Filed Programmable Logic and Applications (FPL2014)* pp. 1–8 (Sept. 2014).