

A CNN/MLP NEURAL PROCESSING ENGINE, POWERED BY
NOVEL TEMPORAL-CARRY-DEFERRING MACS

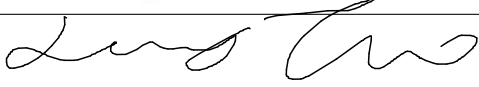
by

Ali Mirzaeian
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Engineering

Committee:



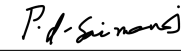
Dr. Avesta Sasan, Dissertation Director



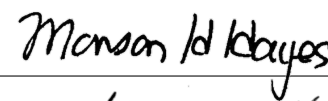
Dr. Zhi Tian, Committee Member



Dr. Liang Zhao, Committee Member



Dr. Sai Manoj Pudukotai Dinakarrao,
Committee Member



Dr. Monson Hayes, Department Chair



Dr. Kenneth S. Ball, Dean,
The Volgenau School of Engineering

Date: _____

Summer 2021
George Mason University
Fairfax, VA

Neuromorphic Hardware Design for Executing Deep Neural Networks on Low Power and
Limited Resource Infrastructures

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor
of Philosophy at George Mason University

By

Ali Mirzaeian
Master of Science
Iran University of Science and Technology (IUST), 2015
Bachelor of Science
Isfahan University of Technology (IUT), 2012

Director: Dr. Avesta Sasan, Professor
Department of Electrical and Computer Engineering (ECE)

Summer 2021
George Mason University
Fairfax, VA

Copyright © 2021 by Ali Mirzaeian
All Rights Reserved

Dedication

I dedicate this dissertation to my beloved family.

Acknowledgments

I would like to thank Dr. Avesta Sasan who made this work possible by positive support and great understanding.

Table of Contents

	Page
List of Tables	vii
List of Figures	viii
Abstract	0
1 Introduction	1
2 TCD-NPE: A Re-configurable and Efficient Neural Processing Engine, Powered by Novel Temporal-Carry-Deferring MACs	3
2.1 Introduction	3
2.2 Related Work	4
2.3 Our Proposed MLP Processing Engine	5
2.3.1 Temporal Carry Deferring MAC (TCD-MAC)	5
2.3.2 TCD-NPE: Our Proposed MLP Neural Processing Engine	9
2.4 Results	19
2.4.1 Evaluation and Comparison Framework	19
2.4.2 TCD-MAC PPA Assessment	20
2.4.3 TCD-NPE Evaluation	21
2.5 Conclusion	25
3 NESTA: Hamming Weight Compression-Based Neural Processing Engine	26
3.1 Introduction and Background	26
3.2 NESTA: Proposed Processing Engine	29
3.2.1 Motivation 1: Temporal Carry	29
3.2.2 Motivation 2: Compression and Expansion	30
3.2.3 NESTA: Our Proposed Solution	32
3.2.4 NESTA: Putting it all together	37
3.2.5 Supported Data Flows	39
3.3 Results	42
3.3.1 Evaluation and Comparison Framework	42
3.3.2 PPA efficiency: NESTA v.s. MAC9s	43
3.3.3 PPA efficiency: NESTA v.s. MACs	46

3.3.4	NESTA for Efficient CNN Processing	48
3.4	Conclusion	48
4	TCD-MAC++: An Enhanced Version of Temporal Carry Deferring MAC . . .	49
4.1	Introduction	49
4.1.1	Temporal Carry Deferring MAC++ (TCD-MAC++)	51
4.1.2	Shorter Critical Path	52
4.1.3	Early Free-Run Stopping (EFRS)	54
4.2	Results	55
4.2.1	TCD-MAC++ PPA Assessment	56
4.2.2	EFRS Evaluation	59
4.3	Conclusion	60
5	Future Works	61
5.1	Introduction	61
5.2	Future Work	61
	Bibliography	64

List of Tables

Table	Page
2.1 PPA comparison between various MACs and TCD-MAC.	20
2.2 Percentage improvement in throughput and energy when using a TCD-MAC (as opposed to a conventional MAC) to process an stream of 1, 10, 100 and 1000 multiplication and addition operations.	21
2.3 TCD-NPE implementation details and PPA results. In this table, we have only reported the leakage power. The dynamic power is activity dependent. The breakdown of energy consumption for processing different benchmarks is reported in Fig. 2.10	22
2.4 MLP benchmarks used in this work [1].	24
3.1 Depth and complexity of some of the existing and modern CNN solutions for object detection.	27
3.2 Comparing the efficiency of HWC-Adder(s) vs Adder tree constructed using Brent-Kung (BK) and Kogge-Stone (KS).	32
3.3 PPA comparison between various MAC flavors and NESTA-V1 and NESTA.	47
3.4 Percentage improvement in Throughput(left) & energy consumption(right) when using NESTA to process 1K of different convolution size.	47

List of Figures

Figure		Page
2.1	Comparing the architecture of A) a typical MAC, versus B) a simplified 2-input version of TCD-MAC. In all variables in form of D_m^i , the subscript (m) captures the bit position values, and postscript (i) capture the cycle (iteration). For example, A^i, B^i are the input data in the i^{th} iteration (corresponding to the i^{th} cycle) of the multiply accumulate operation. The b_m^i, a_m^i , and p_m^i are accordingly the m^{th} significant bits of inputs A, B , and partial sum at the i^{th} cycle (iteration). The division of CPA into GEN and PCPA is also shown in this figure. Note that the <i>PCPA</i> is only executed at the last cycle.	6
2.2	TCD-MAC cycle time is computed by excluding the PCPA. In the last cycle of computation, the TCD-MAC activates the PCPA to propagate the unconsumed carry bits.	8
2.3	TCD-NPE overall architecture. The Mapper algorithm is executed externally, and the sequence of events is loaded into the controller for governing the OS data and control flow.	10
2.4	The logic implementation of Quantization (Left) and Relu Activation (right) for signed fixed-point 16bit values	11
2.5	Assuming a 6×3 PE-array of TCD-MACs, the NPE(K, N) could be configured such that $(K, N) \in \{(1,18), (2,9), (3,6), (6,3)\}$. This figure illustrate the number of rolls, and utilization when each of NPE(K,N) configurations is used to run a $\Gamma(3,I,9)$. model. Each roll is executed I times.	13
2.6	An example execution of algorithm 1 when processing $\Gamma(5, I, 7)$ model using a TCD-MAC with a 6×3 PE-array. (A): the complete computational Tree from CreateTree procedure, (B): binary execution tree obtained from BFS scheduling, (C): the sequence of scheduled events to compute the model based on binary execution tree.	14

2.7	The arrangement of data in W-mem and FM-mem when our proposed TCD-NPE is used in NPE(K,N)=(2,64) configuration mode to process $B = 2$ batches of a hidden layer of an MLP model as defined by $\Gamma(B, I, H) = (2, 200, 100)$	16
2.8	An example of LDN for managing the connection between a (6×3) -PE-array's NoC and memory. (A).left: LDN for writing from NoC data bus to FM-mem. (A).right: LDN for reading from FM-mem to NoC bus. (B): LDN for reading from W-mem into NoC filter bus. The FM-mem in this case, is divided into 6 partitions, supporting the simultaneous process of 6 batches at a time.	19
2.9	Four possible data flow for processing an MLP model. (A): NLR data flow using conventional MACs to form a systolic array. (B): RNA data flow resulted from unrolling the MLP model and mapping the computation tree to conventional MACs (each used as either multiplier or adder) as described in [2]. (C) The OS data flow using conventional MAC. (D): The OS dataflow using TCD-MAC.	23
2.10	Comparison of TCD-NPE with an NPE constructed using conventional MACs that uses the OS, NLR, or RNA data flow. top): Execution time for various MLP benchmarks. Bottom): Energy consumption for various MLP benchmarks.	25
3.1	Computing one CONV layer using input <i>Ifmap</i> /image and filters to produce the output <i>Ofmaps</i>	28
3.2	comparing the architecture of A) a typical MAC, versus B) a simplified 2-input version of NESTA. In all variables in the form of D_m^i , the subscript (m) captures the bit position values, and postscript (i) capture the cycle (iteration). For example, A^i, B^i are the input data in the i^{th} iteration (corresponding to the i^{th} cycle) of the multiply accumulate operation. The b_m^i, a_m^i , and p_m^i are accordingly the m^{th} significant bits of inputs A, B , and partial sum at the i^{th} cycle (iteration). The division of CPA into GEN and PCPA is also shown in this figure. Note that the <i>PCPA</i> is only executed at the last cycle.	29

- 3.3 An Adder tree for 9 16-bit-width entries (left), Hamming Weight Adder (HWC-Adder) of 9 16-bit-wide entries (right). In the HWC-Adder compressor hierarchy (CEL) the complete compressors are colored blue, while compressors with available capacity are white. For building the improved version of HWC-Adder (HWC-Adder*), 2 bits from each compressor in CEL-1 are differed to a compressor in the same bit position in CEL-2, increasing the number of complete compressors and reducing the critical path delay in CEL-1 as reported in table 3.2 31
- 3.4 In NESTA carry bits that are generated in GEN section of the CPA do not propagate into the carry chain. Instead, they are captured by CB registers. In the next cycle, the carry bits (of the previous cycle that are stored in CB registers) are fed to the hamming weight compressors at that bit position, temporally deferring their impact to the next cycle. The compression unit, in each cycle consumes the bit values from 9 new input (W, I) pairs, the Carry bits of the previous cycle (stored in CB registers) and the partial sum stored in S registers. The consumption of bit values in S registers implement the accumulation function. In the last round of computation, instead of capturing the carry bits in CB registers, they are fed to the PCPA (Partial CPA) to propagate through the carry chain and generate the correct convolution results. 34
- 3.5 NESTA cycle time is computed by excluding the execution time of PCPA. In the last cycle of computation of convolution, the NESTA activates the PCPA and captured the correct sum after 2 cycles of execution. 38
- 3.6 NESTA Row Stationary (RS) data flow for executing 3×3 convolution across multiple channels (right) and 5×5 convolution across multiple channels (left). A similar concept can be used to support all other convolutions sizes. . . . 39
- 3.7 A 9-input MAC, which is identified as (Multiplier Choice, Adder Choice). MACs constructed with similar structure are used for PPA and PDP comparison with our proposed NESTA PE solution. 43
- 3.8 Comparing the processing time of a NESTA and a MAC9 for convolutions with (A) 1x1, (B) 3x3, (C) 5x5, and (D) 11x11 kernel size when the convolution expands over multiple channels. As illustrated NESTA for larger convolutions or deeper channels becomes very more efficient. 44

3.9	Breakdown of delay and energy consumption of each layer of VGG [3] when processed by a single MAC, a single MAC9 or a single NESTA core. A linear increase in the number of cores linearly reduces the processing time.	45
3.10	Breakdown of delay and energy consumption of each layer of AlexNet [4] when processed by a Neural engine composed of MACs, MAC9s or NESTA cores.	45
3.11	Area, Delay, Power, and PDP comparison between NESTA and MAC9s constructed using fast adders and multipliers. The star identifies the best MAC9 in each category.	46
4.1	Comparing of three available architecture for processing a stream of MAC operations with operands A^i, B^i in the i^{th} iteration (corresponding to the i^{th} cycle) of the multiply-accumulate operation. A-1) an abstract view of a typical MAC, B-1) an abstract view of a Fused MAC, C-1) an abstract view of a TCD-MAC. A-2, B-2, C-2 are the activated components at each cycle of the mac architecture, and A-3, B-3, C-3 are the dataflows of each one of the MAC architecture for processing operands A^i, B^i	50
4.2	An abstract view of the architectures TCD-MAC and TCD-MAC++ for performing MAC operation on a stream of operands A^i and B^i	52
4.3	Three versions of TCD-MAC++ for calculating mac operations with 4-bit width signed operands. 1) TCD-MAC++ $_{\alpha}$: deferring carry bits from the first layer of expansion layer. 2) TCD-MAC++ $_{\beta}$: deferring the carry bits from the second layer of expansion layer 3) TCD-MAC++ $_{\gamma}$: deferring the carry bits from the third layer of expansion layer (available mac operation with a higher bit-width operands)	53
4.4	The architecture of EFRS for terminating the operation based on the desired precision Pr_0 to Pr_{35} that are obtained from the difference between ORU^i and ORU^{i+1} of two successive iterations i and $i+1$ of TCD-MAC++.	55
4.5	PPA comparison between various MACs and TCD-MAC, and different versions of TCD-MAC++.	56
4.6	Percentage improvement in throughput and energy when using a TCD-MAC++ (as opposed to a conventional MAC) to process an input stream size ranging from 1 to 1000.	58

4.7	The number of needed EFRS of a stream of 1k mac operations of a batch of 1k operands which is performed by three versions of TCD-MAC++. Red dash-line shows the maximum number of needed free run in order to generate the accurate results.	59
5.1	An abstract view of the architecture TCD-MAC/TCD-MAC++ for performing MAC operation on a stream of operands A^i and B^i	62

Abstract

NEUROMORPHIC HARDWARE DESIGN FOR EXECUTING DEEP NEURAL NETWORKS ON LOW POWER AND LIMITED RESOURCE INFRASTRUCTURES

Ali Mirzaeian, PhD

George Mason University, 2021

Dissertation Director: Dr. Avesta Sasan

The applications of machine learning algorithms are innumerable and cover nearly every domain of modern technology. During this rapid growth of this area, more and more companies have expressed a desire to utilize machine learning techniques in smaller devices, such as cell phones or smart Internet of Things (IoT) instruments. However, as machine learning has so far required a power source with more capacity and higher efficiency than a conventional battery. Therefore, introducing neural network accelerators with low energy demands and low latency for executing machine learning techniques has drawn lots of attention in both the academia and industry.

In this work, we first propose the design of Temporal-Carry-deferring MAC (TCD-MAC) and illustrate how our proposed solution can gain significant energy and performance benefit when utilized to process a stream of input data. We then propose using the TCD-MAC to build a reconfigurable, high speed, and low power Neural Processing Engine (TCD-NPE). Furthermore, we expand the idea of TCD-MAC to present NESTA, which is a specialized Neural engine that reformats Convolutions into 3×3 batches and uses a hierarchy of Hamming Weight Compressors to process each batch.

Chapter 1: Introduction

In recent years, machine learning has provided a foundation for rapid technological advancement and massive economic growth. The global value of the machine learning market is estimated at over \$7 billion, and this figure is predicted to become much larger in the coming decade. Machine learning is one of the most exciting frontiers in computer engineering. By training computer algorithms to conduct tasks that normally require human involvement, engineers have saved countless industries both time and money.

Around the globe, machine learning is widely employed in industries like medicine and finance. To take medicine as an example, machine learning is being employed to personalize health care to suit specific patients and to diagnose illnesses. Machine learning has become more ubiquitous as engineers refine the available technology. However, the progress of machine learning and the expansion of its use in the medical industry, as well as many other industries, depends in large part on the progress of efficient computing technologies.

Machine learning is becoming a more and more computationally expensive operation. At the same time, more and more companies have expressed a desire to utilize machine learning techniques in smaller devices, such as cell phones or smart Internet of Things (IoT) instruments. The size restrictions of remote and wireless devices have presented a barrier to this initiative, however, as machine learning has so far required a power source with more capacity and higher efficiency than a conventional battery.

On the hardware platform side, the GPU solutions have rapidly evolved over the past decade and are considered a prominent mean of training and executing DNN models. Although GPU has been a real energizer for this research domain, it is not an ideal solution for efficient learning, and it is shown that development and deployment of hardware solutions dedicated to processing the learning models can significantly outperform GPU solutions. This has lead to the development of Tensor Processing Units (TPU) [5], Field

Programmable Gate Array (FPGA) accelerator solutions [6], and many variants of dedicated ASIC solutions [7–10].

Today, there exist many different flavors of ASIC neural processing engines. The common theme between these architectures is the usage of a large number of simple Processing Elements (PEs) to exploit the inherent parallelism in DNN models. Compare to a regular CPU with a capable Arithmetical Logic Unit (ALU), the PE of these dedicated ASIC solutions is stripped down to a simple Multiplication and Accumulation (MAC) unit. However, many PEs are used to either form a specialized data flow [8], or tiled into a configurable NoC for parallel processing DNNs [10,12,13]. The observable trend in the evolution of these solutions starting from DianNao [7], to DaDianNao [8], to ShiDianNao [9], to Eyris [10] (to name a few) is the optimization of data flow to increase the re-use of information read from memory, and to reduce the data movement (in NOC and to/from memory).

Common between previously named ASIC solutions, is designing for data reuse at NOC level but ignoring the possible optimization of the PE’s MAC unit. A conventional MAC operates on two input values at a time, computes the multiplication result, adds it to its previously accumulated sum, and output a new and *correct* accumulated sum. When working with streams of input data, this process takes place for every input pair taken from the stream. But in many applications, we are not interested in the correct value of intermediate partial sums, and we are only interested in the correct final result.

The first design question that we answer is if we can design a faster and more efficient MAC if we remove the requirement of generating a correct intermediate sum when working on a stream of input data. This question led us to the design of a novel building block to improve the speed of machine learning techniques. We called this basic block Temporal Carry Deferring MAC (TCD-MAC). Later on, we introduced TCD-MAC++ as an extended version of TCD-MAC. We employed these basic blocks for building a Multi-Layer Perceptron (MLP) processing engine and also a Convolutional Neural Network (CNN) engine.

Chapter 2: TCD-NPE: A Re-configurable and Efficient Neural Processing Engine, Powered by Novel Temporal-Carry-Deferring MACs

2.1 Introduction

Deep neural networks (DNNs) has attracted a lot of attention over the past few years, and researchers have made tremendous progress in developing deeper and more accurate models for a wide range of learning-related applications [3, 4, 14–21]. The desire to bring these complex models to resource-constrained hardware platforms such as Embedded, Mobile and IoT devices has motivated many researchers to investigate various means of improving the DNN models’ complexity and computing platform’s efficiency [22, 23]. In terms of model efficiency, researchers have explored different techniques including quantization of weights and features [24, 25], formulating compressed and compact model architectures [25–31], increasing model sparsity and pruning [25, 32], binarization [24, 33], and other model-centered alternatives.

In this chapter, we propose the design of Temporally-deferring-Carry MAC (TCD-MAC), and use the TCD-MAC to build a reconfigurable, high speed, and low power MLP Neural Processing Engine (NPE). We illustrated that TCD-MAC can produce an approximate-yet-correctable result for intermediate operations, and could correct the output in the last state of stream operation to generate the correct output. We then build a Re-configurable and specialized MLP Processing Engine using a farm of TCD-MACs (used as PEs) supported by a reconfigurable global buffer (memory) and illustrate its superior performance and lower energy consumption when compared with the state of the art ASIC

NPU solutions. To remove the data flow dependency from the picture, we used our proposed NPE to process various Fully Connected Multi-Layer Perceptrons (MLP) to simplify and reduce the number of data flow possibilities and to focus our attention on the impact of PE in the efficiency of the resulting accelerator.

2.2 Related Work

The work in [10], categorizes the possible data flows into four major categories: 1) No Local Reuse (NLR) where neither the PE (MAC) output nor filter weight is stored in the PE. Examples of accelerator solutions using NLR data flow include [7, 8, 35]. 2) Output Stationary (OS) where the filter and weight values are input in each cycle, but the MAC output is locally stored. Examples of accelerator solutions using OS data flow include [9, 36–38]. 3) Weight Stationery (WS) where the filter values are locally stored, but the MAC result is passed on. Examples of accelerators using WS data flow include [39–41], and 4) Row Stationary (RS and its variant RS+) where some of the reusable MAC outputs and filter weights remain within a local group of PE to reduce data movement for computing the next round of computation. An example of accelerator using RS is [10].

The OS and NLR are generic data flow and could be applied to any DNN, while the WS and RS only apply to Convolutional Neural Networks (CNN) to promote the reuse of filter weights. Hence, the type of applicable data reuse (output and/or weight) depends on the model being processed. The Multi-Layer Perceptrons (MLP) is a sub-class of NNs that has extensively used for modeling complex and hard to develop functions [42]. An MLP has a feed-forward structure, and is comprised of three types of layers: (1) An input layer for feeding the information to the model, 2) one or more hidden layer(s) for extracting features, and (3) an output layer that produces the desired output which could be regression, classification, function estimation, etc. Unfortunately, when it comes to MLPs, or when processing Fully Connected (FC) layers, unlike CNNs, no filter weight could be reused. In these models the viable data flows are the OS and NLR. The only possible solution

for using the WS solution in processing MLPs is the case of multi-batch processing that may benefit from weight reuse. Another related work is the NPE proposed in [2]. This solution, denoted as RNA, is a special case of NLR, where data flow is controlled through NoC connectivity between different PEs; RNA breaks the MLP model into multi-layer loops that are successively mapped to the accelerator PEs, and uses the PEs as either a multiplier or an adder, dynamically forming a systolic array.

In the result section of this paper, We demonstrate that the OS solutions are in general more efficient than NLR solutions. We further illustrate that our proposed TCD-MAC, when used in the context of our proposed NPE, outperform state of the art accelerators that rely on (fastest and most efficient) conventional MAC solutions.

2.3 Our Proposed MLP Processing Engine

Before describing our proposed NPE solution, we first describe the concept of *temporal carry* and illustrate how this concept can be utilized to build a Temporal Carry deferring Multiplication and Accumulation (TCD-MAC) unit. Then, we describe, how an array of TCD-MAC are used to design a re-configurable and high-speed MLP processing engine, and how the sequence of operations in such NPE is scheduled to compute multiple batches of MLP models.

2.3.1 Temporal Carry Deferring MAC (TCD-MAC)

Suppose two vectors A and B each have N M-bit values, and the goal is to compute their dot product, $\sum_{i=0}^{N-1} (A_i * B_i)$ (similar to what is done during the activation process of each neuron in a NN). This could be achieved using a single Multiply-Accumulate (MAC) unit, by working on 2 inputs at a time for N rounds. Fig. 2.1(A-top) shows the general view of a typical MAC architecture that is comprised of a multiplier and an adder (with 4-bit input width), while Fig. 2.1(A-bottom) provides a more detailed view of this architecture. The partial products (M partial product for M-bits) are first generated in Data Reshape Unit

(DRU). Then the hamming weight compressors (HWC) in the Compression and Expansion Layer (CEL) transform the addition of M partial products into a single addition of two larger binaries, the addition of which in an adder generates the multiplication result.

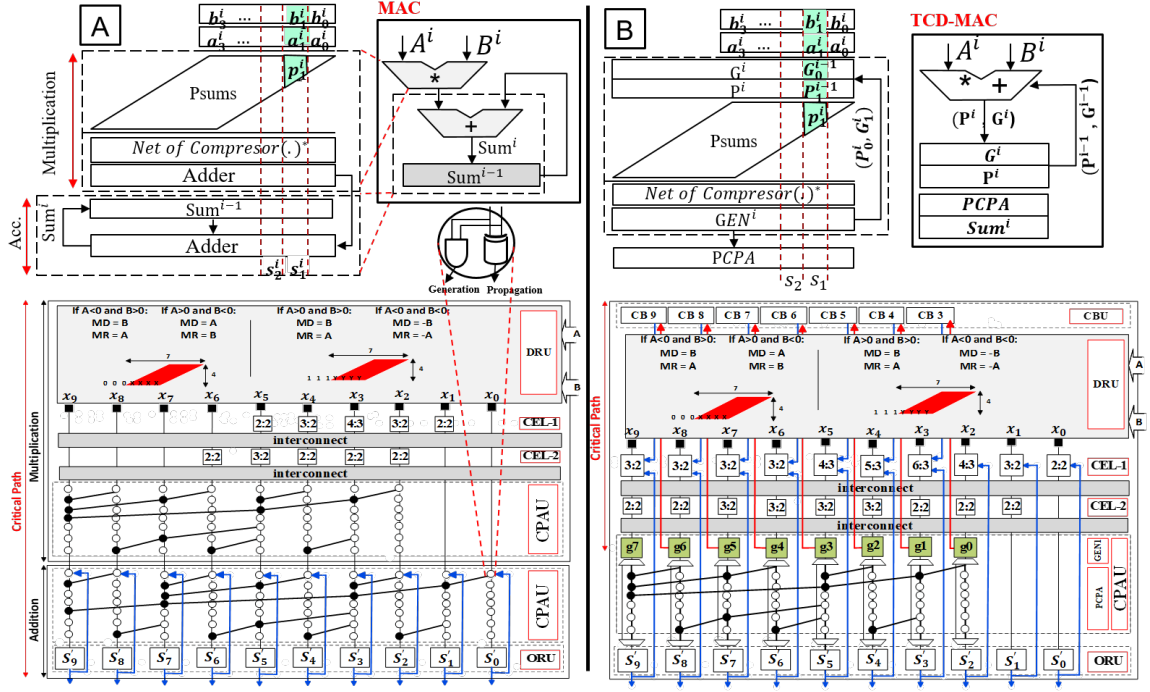


Figure 2.1: Comparing the architecture of A) a typical MAC, versus B) a simplified 2-input version of TCD-MAC. In all variables in form of D_m^i , the subscript (m) captures the bit position values, and postscript (i) capture the cycle (iteration). For example, A^i, B^i are the input data in the i^{th} iteration (corresponding to the i^{th} cycle) of the multiply accumulate operation. The b_m^i, a_m^i , and p_m^i are accordingly the m^{th} significant bits of inputs A, B , and partial sum at the i^{th} cycle (iteration). The division of CPA into GEN and PCPA is also shown in this figure. Note that the PCPA is only executed at the last cycle.

The building block of the CEL unit are the HWC. A HWC, denoted by $C_{HW}(m:n)$, is a combinational logic that implements the Hamming Weight (HW) function for m input-bits (of the same bit-significance value) and generates an n -bit binary output. The output n of HWC is related to its input m by: $n = \lceil \log_2^m \rceil$. For example “011010”, “111000”, and “000111” could be the input to a $C_{HW}(6:3)$, and all three inputs generate the same Hamming weight value represented by “011”. A Completed HWC function $CC_{HW}(m:n)$ is

defined as a C_{HW} function, in which m is $2^n - 1$ (e.g., CC(3:2) or CC(7:3)). Each HWC takes a column of m input bits (of the same significance value) and generates its n -bit hamming weight. In the CEL unit, the output n -bits of each HWC is fed (according to its bit significance values) as an input to the proper $C_{HW}(s)$ in the next-layer CEL. This process is repeated until each column contains no more than 2-bits, which is a proper input size for a simple adder. In Fig. 2.1 it is assumed that a Carry Propagation Adder Unit (CPAU) is used. The result is then added to the previously accumulated value in the output register in the second adder to generate a new accumulated sum. Note that in conventional MAC, the carry (propagation) bits in the CPAUs are spatially propagated through the carry chain which constitutes the critical timing path for both adder and multiplier.

Fig.2.1.B shows our proposed TCD-MAC. In this solution, only a single CPAU is used. Furthermore, the CPAU is broken into two distinct segments 1) The GENERation (GEN) and Partial CPA (PCPA). The Gen is the first layer of CPA logic that produces the Generate (G_i^c) and Propagate (P_i^c) signals for each bit position i at cycle c . The TCD-MAC relies on the assumption that we only need to correctly compute the final result of multiplication and accumulation over an array of inputs (e.g. $\sum_{i=0}^{N-1} (A_i * B_i)$), while relaxing the requirement for generating correct intermediate sums. This relaxed specification is applicable when a MAC is used to compute a Neuron value in a DNN. Benefiting from this relaxed requirement, the TCD-MAC skips the computation of PCPA, and injects (defers) the G_i^c and P_i^c generated in cycle c , to the CEL unit in cycle $c + 1$. Using this approach, the propagation of carry-bit in the long carry chain (in PCPA) is skipped, and without loss of accuracy, the impact of the carry bit is injected to the correct bit position in the next cycle of computation. We refer to this process as temporal (in time) carry propagation. The Temporally carried G_i^c is stored in a new set of registers denoted as Carry Buffer Unit (CBU), while the P_i^c in each cycle is stored in the output register Unit (ORU). Note that CBU bits can be injected to any of the $C_{HW}(m : n)$ in any of the CEL layers in the same bit position. However, it is desired to inject the CB bits to a $C_{HW}(m : n)$ that is incomplete to avoid an increase in

the size and critical path delay of the CEL.

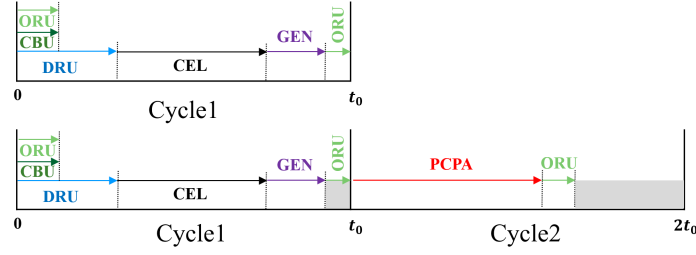


Figure 2.2: TCD-MAC cycle time is computed by excluding the PCPA. In the last cycle of computation, the TCD-MAC activates the PCPA to propagate the unconsumed carry bits.

Assuming that a TCD-MAC works on an array of N input pairs, the temporal carry injection is done $N-1$ times. In the last round, however, the PCPA should be executed. As illustrated in Fig. 2.2, in this approach, the cycle time of the TCD-MAC could be reduced to that excluding the PCPA, allowing the computation over PCPA to take place in an extra cycle. The one extra cycle allows the unconsumed carry bits to be propagated in PCPA carry chain, forcing the TCD-MAC to generate the correct output. Using this technique we shortened the cycle time of TCD-MAC for a large number of cycles. The saving obtained from shorter cycles over a large number of cycles significantly outweighs the penalty of one extra cycle.

To support signed inputs, in TCD-MAC we pre-process the input data. For a partial product $p = a \times b$, if one value (a or b) is negative, it is used as the multiplier. With this arrangement, we treat the generated partial sums as positive values and later correct this assumption by adding the two's complement of the multiplicand during the last step of generating the partial sum. Following example clarify this concept: let's suppose that a is a positive and b is a negative b -bit binary. The multiplication $b \times a$ can be reformulated as:

$$b \times a = (-2^7 + \sum_{i=0}^6 x_i 2^i) \times a = -2^7 a + (\sum_{i=0}^6 x_i 2^i) \times a \quad (2.1)$$

The term -2^7a is the two's complement of multiplicand which is left-shifted by 7 bits, and the term $(\sum_{i=0}^6 x_i 2^i) \times a$ is only accumulating shifted version of the multiplicand.

2.3.2 TCD-NPE: Our Proposed MLP Neural Processing Engine

TCD-NPE is a configurable neural processing engine which is composed of a 2-D array of TCD-MACs. The TCD-MAC array is connected to a global buffer using a configurable Network on Chip (NOC) that supports various forms of data flow as described in section 4.1. However, for simplicity, we limit our discussion to supporting OS and NLR data flows for executing MLPs. This choice is made to help us focus on the performance and energy impact of utilizing TCD-MACs in designing an efficient NPE without complicating the discussion with the support of many different data flows.

Figure 2.3 captures the overall TCD-NPE architecture. It is composed of 1) Processing Element (PE) array which is a tiled array of TCD-MACs, 2) Local Distribution Networks (LDN) that manages the PE-array connectivity to memories, 3) Two global buffers, one for storing the filter weights and one for storing the feature maps, and 4) The Mapper-and-controller unit which translates the MLP model into a supported data and control flow. The functionality and design of each of these units are described next:

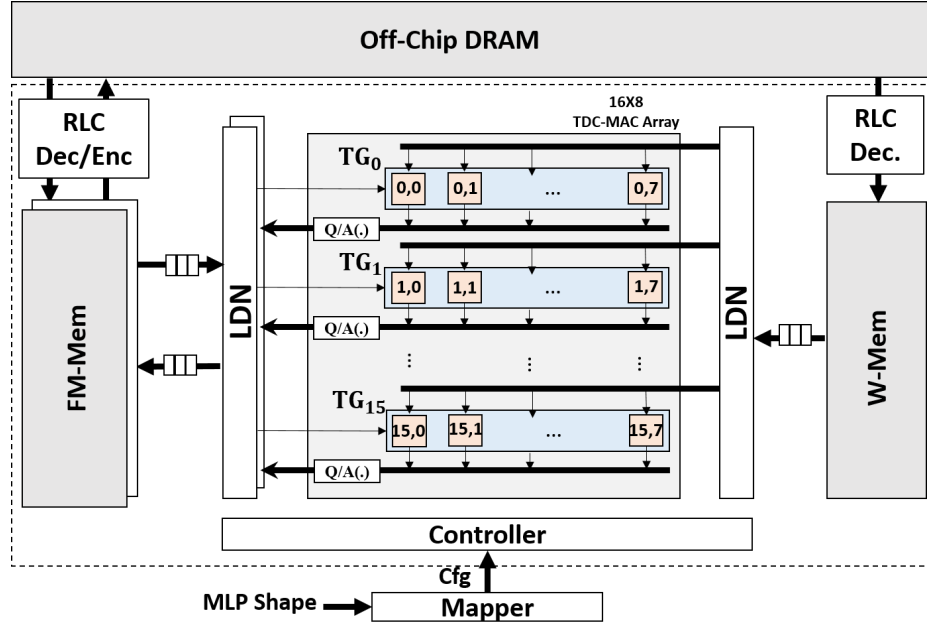


Figure 2.3: TCD-NPE overall architecture. The Mapper algorithm is executed externally, and the sequence of events is loaded into the controller for governing the OS data and control flow.

PE Array

The PE-array is the computational engine of our proposed TCD-NPE. Each PE in this tiled array is a TCD-MAC. Each TCD-MAC could be operated in two modes: 1) Carry Deferring Mode (CDM), or 2) Carry Propagation Mode (CPM). According to the discussion in section 2.3.1, when working with an input stream of size N , the TCD-MAC is operated in the CDM model for N cycles (computing approximate sum), and in the CPM mode in the last cycle to generate the correct output. This is in line with OS data flow as described in section 2.2. Note that the TCD-MAC in this PE-array could be operated in CPM mode in every cycle allowing the same PE-array architecture to also support the NLR. After computing the raw neuron value (prior to activation), the TCD-MAC writes the computed sum into the NOC bus. The Neuron value is then passed to the quantization and activation unit before being written back to the global buffer. Fig. 3.5 captures the logic implementation for quantization (to 16 bits) and Relu[4] activation in this unit.

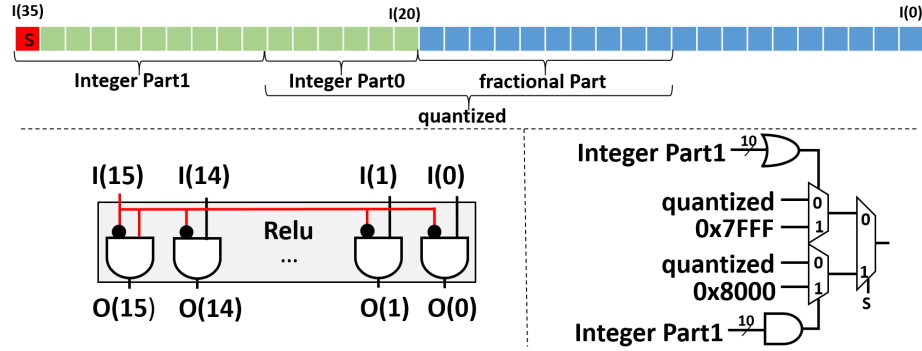


Figure 2.4: The logic implementation of Quantization (Left) and Relu Activation (right) for signed fixed-point 16bit values

Consider two layers of an MLP where the input layer contains M feature-values (neurons) and the second layer contains N Neurons. To compute the value of N Neurons, we need to utilize N TCD-MACs (each for $M+1$ cycles). If the number of available TCD-MACs is smaller than N , the computation of the neurons in the second layer should be unrolled to multiple rolls (rounds). If the number of available TCD-MACs is larger than neurons in the second layer (for small models), we can simultaneously process multiple batches (of the model) to increase the NPE utilization. Note that the size of the input layer (M) will not affect the number of needed TCD-MACs, but dictates how many cycles ($M+1$) are needed for the computation of each neuron.

When mapping a batch of MLP to the PE-array, we should decide how the computation is unrolled and how many batches (K), and how many output neurons (N) should be mapped to the PE-array in each roll. The optimal choice would result in the least number of rolls and the maximum utilization of the NPE. To illustrate the trade-offs in choosing the value of (K, N) let us consider a PE-array of size 18, which is arranged in 6 rows and 3 columns of TCD-MACs (similar to that in Fig. 2.3). We refer to each row of TCD-MACs as a TCD-MAC Group (TG). In our implementation, to reduce NOC complexity, the TG groups work on computing neurons in the same batch, while different TG groups could be assigned to work on the same or different batches. The architecture in Fig. 2.3 has 6 TG groups. Let

us use $\text{NPE}(K, N)$ to denote the choice of using the PE-array to compute N neuron values in K batches where $N \times K = 18$. In our example PE-array the following selections of K and N are supported: $(K, N) \in (1, 18), (2, 9), (3, 6), (6, 3)$. The $(9, 2)$ and $(18, 1)$ configuration are not supported as the value of N in this configurations is smaller than TG size = 3.

Fig. 2.5.left shows an abstract view of TCD-NPE and describe how the weights and input features (from one or more batches) are fed to the TCD-NPE for different choices of K and N . As an example 2.5.(left).A shows that input features from one batch are broadcasted between all TGs, while the weights are unicasted to each TCD-MAC. Let us represent the input scenario of processing B batches of U neurons in a hidden or output layer of an MLP model with I input features using $\Gamma(B, I, U)$. Fig. 2.5.(right) shows the NPE status when a $\Gamma(3, I, 9)$ model (3 batches of a hidden layer with 9 neurons in a hidden layer each fed from I input neurons) is executed using each of 4 different $\text{NPE}(K, N)$ choices. For example Fig. 2.5.(right).top shows that using configuration $\text{NPE}(1, 18)$, we process one batch with 18 neurons at a time. In this example, when using this configuration, the NPE is underutilized (50%) as there exist only 9 neurons in each batch. Following a similar argument, the $\text{NPE}(6, 3)$ arrangement also have 50% utilization. However the arrangement $\text{NPE}(2, 9)$, and $\text{NPE}(3, 6)$ reach 75% utilization (100% for the roll, and 50% for the second roll), hence either $\text{NPE}(2, 9)$ or $\text{NPE}(3, 6)$ arrangement is optimal for the $\Gamma(3, I, 9)$ problem as they produce the least number of rolls. Note that the value of I in $\Gamma(3, I, 9)$ denotes the number of input features which dictate the number of cycles that the $\text{NPE}(K, N)$ should be executed.

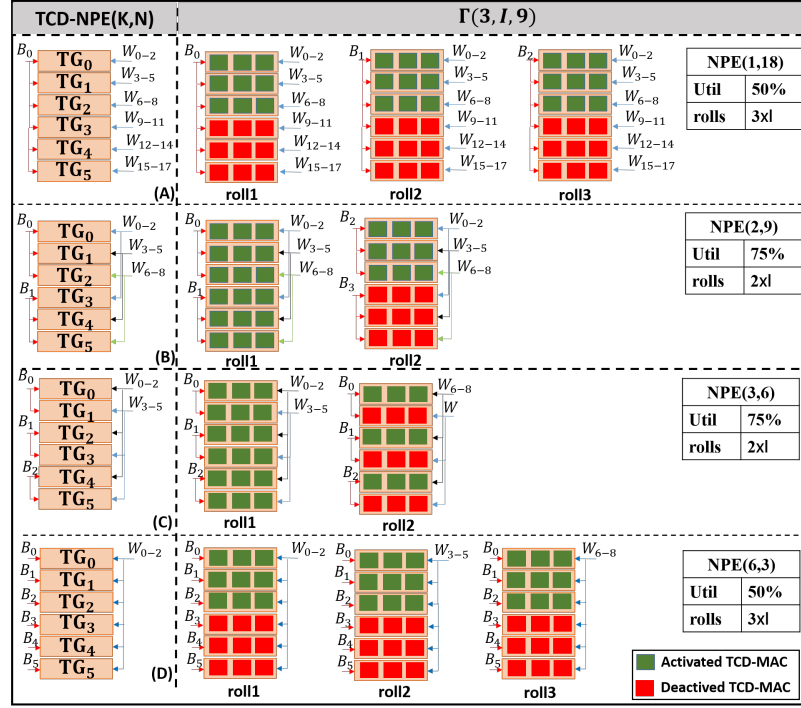


Figure 2.5: Assuming a 6×3 PE-array of TCD-MACs, the NPE(K, N) could be configured such that $(K, N) \in \{(1,18), (2,9), (3,6), (6,3)\}$. This figure illustrate the number of rolls, and utilization when each of NPE(K,N) configurations is used to run a $\Gamma(3, I, 9)$ model. Each roll is executed I times.

Mapping Unit

An MLP has one or more hidden layers and could be presented using $\text{Model}(I - H_1 - H_2 - \dots - H_N - O)$, in which I is the number of input features, H_i is the number of Neurons in the hidden layer i , and O is the number of output layer neurons. The role of the mapping unit is to find the best unrolling scenario for mapping the sequence of problems $\Gamma(B, I, H_1)$, $\Gamma(B, H_1, H_2)$, ..., $\Gamma(B, H_{N-1}, H_N)$, and $\Gamma(B, H_N, O)$ into minimum number of NPE(K,N) computational rounds.

Algorithm 1 describes the mapper function for unrolling a multi-batch multi-layer MLP problem. In this Algorithm, B is the batch size that could fit in the NPE's feature-memory (if larger, we can unroll the B into $N \times B^*$ computation round, where B^* is the number of batches that fit in the memory). $M[L]$ is the MLP layer size information, where $M[i]$ is

the number of nodes in layer i (with $i = 0$ being Input, and $i = N + 1$ being Output, and all others are hidden layers). The algorithm schedules a sequence of NPE(K, N) events to compute each MLP layer across all batches.

Algorithm 1 Schedule NPE(K,N) rolls (events) to execute B batches of $M(L) = \text{MLP}(I, H_1, \dots, H_N, O)$.

```

procedure PRACTICALCFGFINDER(Model  $M[L]$ , BatchSize  $B$ )
  for ( $l = 1$ ;  $\text{size}(M)$ ;  $l++$ ) do
     $\text{Tree}_{\text{head}} = \text{CreateTree}(B, M[l])$ 
     $\text{ExecTree} \leftarrow$  Shallowest binary tree (least rolls) from  $\text{Tree}_{\text{head}}$ 
    Schedule  $\leftarrow$  Schedule computational events by using BFS
                      on  $\text{ExecTree}$  to report NPE(K,N) and  $r$  at each node.
  return Schedule

procedure CREATETREE( $B, \Theta$ )
   $C[i] \leftarrow$  find each  $(K_i, N_i) | K_i, N_i \in \mathbb{N}, \& K_i < B$ 
                                 $\& \text{size}(\text{NPE}) = K_i \times N_i$ 
  for ( $i = 0$ ;  $i < \text{size}(C)$ ;  $i++$ ) do
     $M_B = \min(B, C[i][1])$ .  $\triangleright C[i][1] = K_i$ 
     $M_\Theta = \min(\Theta, C[i][2])$ .  $\triangleright C[i][2] = N_i$ 
     $\psi = (M_B, M_\Theta)$   $\triangleright \psi$ : NPE's (K,N) configuration
     $r = \lfloor B/M_B \rfloor \times \lfloor \Theta/M_\Theta \rfloor$   $\triangleright r$ : # of rolls with NPE( $M_B, M_\Theta$ )
    if ( $B \% M_B \neq 0$ ) then
       $\text{Node}_B \leftarrow \text{CreateTree}(B \% M_B, \Theta)$ 
    if ( $K \% M_\Theta \neq 0$ ) then
       $\text{Node}_\Theta \leftarrow \text{CreateTree}(B - B \% M_B, K \% M_\Theta)$ 
    Node  $\leftarrow \text{createNode}(r, \psi, \text{Node}_B, \text{Node}_\Theta)$ 
  return Node

```

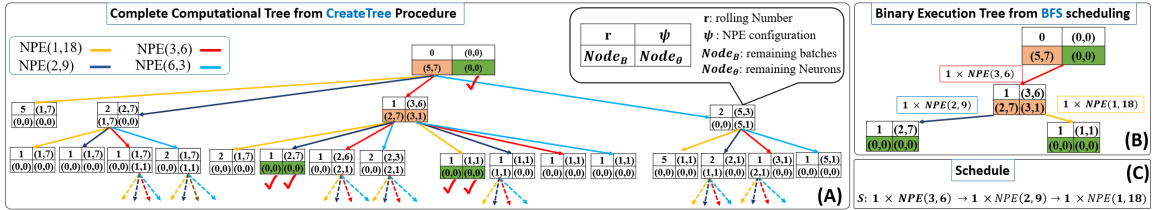


Figure 2.6: An example execution of algorithm 1 when processing $\Gamma(5, I, 7)$ model using a TCD-MAC with a 6×3 PE-array. (A): the complete computational Tree from CreateTree procedure, (B): binary execution tree obtained from BFS scheduling, (C): the sequence of scheduled events to compute the model based on binary execution tree.

To schedule the sequence of events, the Alg. 1 first generates the expanded computational tree of the NPE using *CreateTree* procedure. This procedure first finds all possible ways that NPE could be segmented for processing N neurons of K batches, where $K \leq B$ and stores them into configuration database C . Then for each of configurations of $NPE(K, N)$, it derives how many rounds (r) of $NPE(K, N)$ computations could be executed. Then it computes a) the number of remaining batches (with no computation) and b) the number of missing neurons in partially computed batches. It, then, creates a tree-node, with 4 major fields 1) the load-configuration $\Psi(K_i^*, N_i^*)$ that is used to partially compute the model using the selected $NPE(K_i, N_i)$ such that $(K_i^* \leq K_i) \& (N_i^* \leq N_i)$, 2) the number of rounds (rolls) r taken with computational configuration Ψ to reach that node, 3) a pointer to a new problem $Node_B$ that specifies the number of remaining batches (with no computation), and 4) a pointer to a new problem $Node_\Theta$ for partially computed batches. Then the *CreateTree* procedure is recursively called on each of the $Node_B$ and $Node_\Theta$ until the batches left, and partial computation left in a (leaf) node is zero. At this point, the procedure returns. After computing the computational tree, the mapper extracts the best execution tree by finding a binary tree with the least number of rolls (where all leaf nodes have zero computation left). The number of rolls is computed by summing up the r field of all computational nodes. Finally, the mapper uses a Breath First Search (BFS) on the Execution Tree ($ExecTree$) and report the sequence of $r \times NPE(K, N)$ for processing the entire binary execution tree. The reported sequence is the optimal execution schedule. Fig. 2.6 provides an example for executing 5 batches of a hidden MLP layer with 7 neurons. As illustrated the computation-tree (Fig. 2.6.A) is first generated, and then the optimal binary execution tree (Fig. 2.6.B) resulting in the minimum number of rolls is extracted. Fig. 2.6.C captures the result of scheduling step where BFS search schedule the sequence of $r \times NPE(K, N)$ events.

Controller

The controller is an FSM that receives the "Schedule" from Mapper and generated the appropriate control signals to control the proper OS data flow for executing the scheduled

sequence of events.

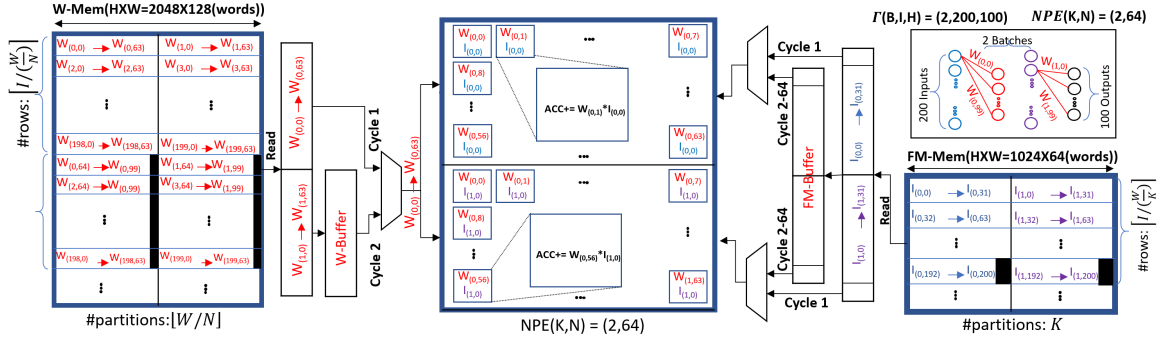


Figure 2.7: The arrangement of data in W-mem and FM-mem when our proposed TCD-NPE is used in $NPE(K,N)=(2,64)$ configuration mode to process $B = 2$ batches of a hidden layer of an MLP model as defined by $\Gamma(B, I, H) = (2, 200, 100)$.

Memory Architecture

The NPE global memory is divided into feature-map memory (FM-Mem), and Filter Weight memory (W-Mem). The FM-Mem consist of two memories with ping-pong style of access, where the input features are read from one memory, and output neurons for the next NN layer, are written to the other memory. When working with multiple batches (B), the input features from the largest number of fitting batches (B^*) is read into feature memory. For simplicity, we have assumed that the feature map is large enough to hold the features (neurons) in the largest layer of at least one MLP (usually the input) layer. Note that the NPE still can be used if this assumption is violated, however, now some of the computed neuron values have to be transferred back and forth between main memory (DRAM) and the FM-Mem for lack of space. The filter memory is a single memory that is filled with the filter weights for the layer of interest. The transfer of data from main memory (DRAM)

to the W-Mem and FM-Mem is regulated using Run Length Coding (RLC) compression to reduce data transfer size and energy.

The data arrangement of features and weights inside the FM-Mem and W-Mem is shown in Fig. 2.7. The data storage philosophy is to sequentially store the data (weight and input features) needed by NPE (according to its configuration) in consecutive cycles in a single row. This data reshaping solution allows us to reduce the number of memory accesses by reading one row at a time into a buffer, and then consuming the data in the buffer in the next few cycles. We explain this data arrangement concept using the example shown in Fig. 2.7.

Fig. 2.7 shows the arrangement of data when we use our proposed TCD-NPE in NPE(K,N)=(2,64) configuration to process $B = 2$ batches of a hidden layer of an MLP model as defined by $\Gamma(B, I, H) = (2, 200, 100)$. Note that the PE array size, in this case is 16×8 which is divided into two 8×8 arrays for processing each of 2 batches. The W-Mem, shown in left, is filled by storing the first $N=64$ weights of each outgoing edge from input Neurons (features) to each of the neurons in the hidden layer. Considering that the width of W-Mem is 256 bytes, and each weight is 2 bytes, the width of W-Mem (W_{W-mem}) is 128 words. Hence, we can store 64 weights of the outgoing edge from each 2 input neurons in one row. The memory-write process is repeated for $\lceil (I/(W_{W-mem}/N)) \rceil = 100$ rows, and then the next $N = 64$ weights of outgoing edges from each input neuron are written (in this case we only have 36 weights left, as there exist a total of 100 outgoing edges from each input neuron, 64 of which is previously stored) in the next $\lceil (I/(W_{W-mem}/N)) \rceil = 100$ rows. At processing time, by using the NPE(2,64) configuration, the TCD-NPE consumes $N = 64$ weights in each cycle. Hence, with one read from W-Mem, it receives the weights needed for $W_{W-mem}/N = 128/64 = 2$ cycles, reducing the number of memory accesses by half.

The FM memory, on the other hand, is divided into $B = 2$ segments. Assuming that the width of FM memory is $W_{FM-mem} = 64$ words, each segment can store $W_{FM-mem}/B = 64/2 = 32$ input features. The memory, as shown in Fig. 2.7, is filled by writing the input

features of each batch into subsequent rows of each virtually segmented memory. Note that both FM-Mem and W-Mem should be word writable to support writing to a section of a row without changing the value of other memory bits in the same row. The input features from each batch is written to the $\lceil (I/(W_{FM-mem}/B)) \rceil = \lceil (200/(64/2)) \rceil = 7$ rows. At processing time, using the NPE(2,64) configuration, the TCD-NPE in one access (Reading one row) will receive W_F/B input features from B different batches and store them in a buffer. In each subsequent cycle, it consumes one input from each batch, hence, the arrangement of data and sequential read of data into a buffer will reduce the number of memory accesses by a factor of $W_{FM-mem}/B = 64/2 = 32$.

Local Distribution Network (LDN)

The Local Distribution Networks (LDN) interface the read/write buffers and the Network on Chip (NOC). They manage the desired multi- or uni-casting scenarios required for distributing the filter values and feature values across TGs. Figure 2.8 illustrate an example of LDNs in an NPE constructed using 6×3 array of TCD-MACs. As illustrated in this example, the LDNs are used for 1) reading/writing from/to buffers of FM-mem while supporting the desired multi-/uni-casting configuration (generated by controller) to support the selected NPE(K, N) configuration (Fig.2.8.A) and 2) reading from W-mem buffer and multi-/uni-casting the result into TGs (Fig.2.8.B). Note that the LDN in Fig. 2.8 is specific to NPE of size 6×3 . For other array sizes, a similar LDN should be constructed.

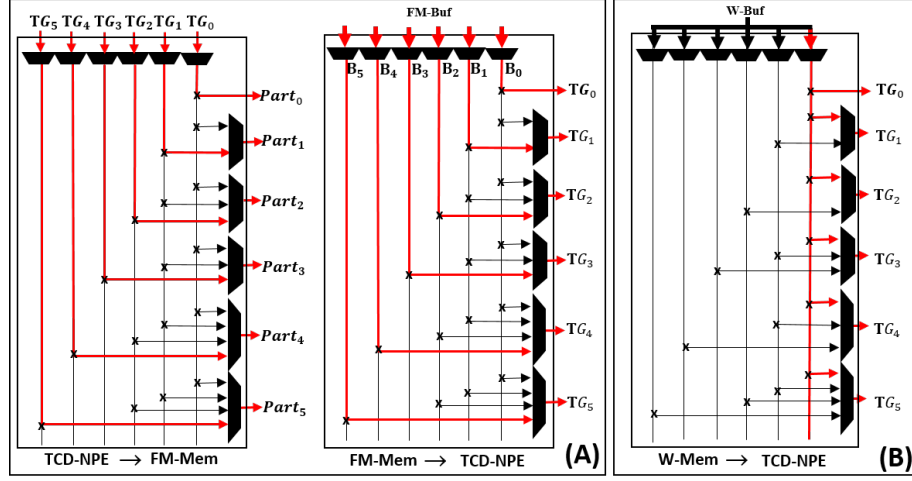


Figure 2.8: An example of LDN for managing the connection between a (6×3) -PE-array's NoC and memory. (A).left: LDN for writing from NoC data bus to FM-mem. (A).right: LDN for reading from FM-mem to NoC bus. (B): LDN for reading from W-mem into NoC filter bus. The FM-mem in this case, is divided into 6 partitions, supporting the simultaneous process of 6 batches at a time.

2.4 Results

In this section, we first evaluate the Power, Performance, and Area (PPA) gain of using TCD-MAC, and then evaluate the impact of using the TCD-MAC in our proposed TCD-NPE. The TCD-MAC and all MACs evaluated in this section operate on signed 16-bit fixed-point inputs.

2.4.1 Evaluation and Comparison Framework

The PPA metrics are extracted from the post-layout simulation of each design. Each MAC is designed in VHDL, synthesized using Synopsis Design Compiler [43] using 32nm standard cell libraries, and is subjected to physical design (targeting max frequency) by using the Synopsys reference flow in IC Compiler [44]. The area and delay metrics are reported using Synopsys Primetime [45]. The reported power is the averaged power across 20K cycles of simulation with random input data that is fed to Prime timePX [45] in FSDB format. The general structure of MACs used for comparison is captured in Fig. 2.1. We have

Table 2.1: PPA comparison between various MACs and TCD-MAC.

MAC Type	Area(μm^2)	Power(μw)	Delay(ns)	PDP(pJ)
(BRx2, KS)	8357	467	2.85	13.31
(BRx2, BK)	8122	394	3.3	13
(BRx8, BK)	7281	383	3.14	12.03
(BRx4, BK)	6437	347	3.35	11.62
(WAL, KS)	7171	346	3.04	10.52
(WAL, BK)	6520	334	3.13	10.45
(BRx4, KS)	6551	393	2.47	9.71
(BRx8, KS)	7342	354	2.63	9.31
TCD-MAC	5004	320	1.57	5.02

compared our solution to a wide array of MACs. In these MACs, for multiplication, we used Booth-Radix-N (BRx2, BRx4, BRx8) and Wallace implementations. For addition we have used Brent-Kung (BK) and Kogge-Stone (KS) adders. Each MAC is identified by the tuple (Multiplier choice, Adder choice).

2.4.2 TCD-MAC PPA Assessment

Table 2.1 captures the PPA comparison of the TCD-MAC against a popular set of conventional MAC configurations. As reported, the TCD-MAC has a smaller overall area, power and delay compare to all reported MACs. Using TCD-MAC provide 23% to 40% reduction in area, 4% to 31% improvement in power, and an impressive 46% to 62% improvement in PDP when compared to other reported conventional MACs.

Note that this improvement comes with the limitation that the TCD-MAC takes one extra cycle to generate the correct output when working on a stream of data. However, the power and delay saving of TCD-MAC significantly outweigh the delay and power for one extra computational cycle. To illustrate this, the throughput and energy improvement of using a TCD-MAC for processing different sizes of input streams (1, 10, 100, 1000) is compared against selected conventional MACs and is reported in Table 4.6. As illustrated, when using the TCD-MAC for processing an array of inputs, the power and delay savings

quickly outweigh the delay and power of the added cycle as input stream size increases.

Table 2.2: Percentage improvement in throughput and energy when using a TCD-MAC (as opposed to a conventional MAC) to process an stream of 1, 10, 100 and 1000 multiplication and addition operations.

Mac Type	Throughput improvement(%)				Energy Improvement(%)			
	1	10	100	1000	1	10	100	1000
(BRX2, KS)	25	59	62	63	-10	40	45	45
(BRX2, BK)	23	58	62	62	5	48	52	53
(BRX8, BK)	17	55	58	59	0	45	50	50
(BRX4, BK)	14	53	57	57	7	49	53	54
(WAL, KS)	5	48	52	53	-3	44	48	49
(WAL, BK)	4	48	52	52	0	45	50	50
(BRX4, KS)	-3	44	48	49	-27	31	36	37
(BRX8, KS)	-7	41	46	47	-19	35	40	41

2.4.3 TCD-NPE Evaluation

In this section, we describe the result of our TCD-NPE implementation as described in section 2.3.2. Table 2.3-top summarizes the characteristics of TCD-NPE implemented, the result of which is reported and discussed in this section. For physical implementation, we have divided the TCD-NPE into two voltage domains, one for memories, and one for the PE array. This allows us to scale down the voltage of memories as they had considerably shorter cycle time compared to that of PE elements. This choice also reduced the energy consumption of memories and highlighted the saving resulted from the choice of MAC in the PE-array. Note that the scaling of the memory voltage could be even more aggressive than what implemented in our solution; In several prior work [46–50], it was shown that it is possible to significantly reduce the read/write/retention power consumption of a memory unit by aggressively scaling it supplied voltage while deploying architectural fault tolerance techniques and solutions to mitigate the increase in the memory write/read/retention failure

rate. On top of that, learning solutions are also approximate in nature, and inherently less sensitive to small disturbance to their input features. This inherent resiliency could be used to deploy fault tolerant techniques to only protect against bit errors in most significant bits of input feature map, resulting in reduced complexity of deployed fault tolerance scheme.

Table 2.3-bottom captures the overall PPA of the implemented TCD-NPE extracted from our post layout simulation results which are reported for a Typical Process, at 85C° temperature, when the PE-array and memory elements voltages are set according to Table 2.3.

Table 2.3: TCD-NPE implementation details and PPA results. In this table, we have only reported the leakage power. The dynamic power is activity dependent. The breakdown of energy consumption for processing different benchmarks is reported in Fig. 2.10

Feature	Detail	Feature	Detail
PE-array	16 × 8	Processing Element	TCD-MAC
Input Data Format	Signed 16-bit fixed-point	Data Flow	OS
W-mem size	512 KByte	Activation Units	Relu
FM-mem Size	2 × 64 KByte	PE-array voltage	0.95V
Mapper	Off-chip using Alg. 1	Mem voltage	0.70V
Area	3.54 mm ²	Max Frequency	636 MHz
PE-array Area	0.724 mm ²	Memory Area	2.5 mm ²
Overall Leak. Power	75.5 mW	Memory Leak. Power	51.7 mW
PE-array Leak. Power	6.4 mW	Others Leak. Power	17 mW

To compare the effectiveness of TCD-NPE, we compared its performance with a similar NPE which is composed of conventional MACS. According to the discussion in section 2.2, we limit our evaluation to the processing of MLP models. Hence, the only viable data flows are OS and NLR. The TCD-MAC only supports OS, however, by replacing a TCD-MAC with a conventional MAC, we can also compare our solution against OS and NLR. We compare 4 possible data flows that are illustrated in Fig. 2.9. In this Fig. The case (A) is NLR data flow (supported only by conventional MAC) for computing the Neuron values

by forming a systolic array withing the PE-array. The case (B) An NLR data flow variant according to [2] when the computation tree is unrolled and mapped to the PEs, forcing the PE to either act as an adder or multiplier. The case (C) is the OS data flow realized by using conventional MAC. And, finally, the case (D) is the OS data flow implemented using TCD-NPE.

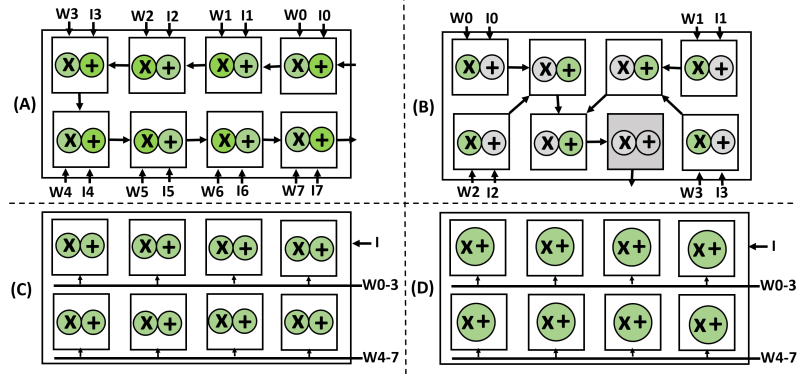


Figure 2.9: Four possible data flow for processing an MLP model. (A): NLR data flow using conventional MACs to form a systolic array. (B): RNA data flow resulted from unrolling the MLP model and mapping the computation tree to conventional MACs (each used as either multiplier or adder) as described in [2]. (C) The OS data flow using conventional MAC. (D): The OS dataflow using TCD-MAC.

For OS dataflows, we have used the algorithm 1 to schedule the sequence of computational rounds. We have compared the efficiency of each of four data flows (described in Fig. 2.9) on a selection of popular MLP benchmarks characteristic of which is described in Table. 2.4.

Table 2.4: MLP benchmarks used in this work [1].

Applications	Dataset	Topology
Digit Recognition	MNIST	784:700:10
Census Data Analysis	Adult	14:48:2
FFT	Mibench data	8:140:2
Data Analysis	Wine	13:10:3
object Classification	Iris	4:10:5:3
Classification	poker Hands	10:85:50:10
Classification	Fashion MNIST	728:256:128:100:10

As illustrated in Fig. 2.10.left, the execution time of the TCD-NPE is almost half of an NPE that uses a conventional MAC in either OS or NLR data flow, and significantly smaller than the RNA data flow (an NLR variant) that was proposed in [2]. Fig. 2.4.right captures the energy consumption of the TCD-NPE and compares that with a similar NPE constructed using conventional MACs. For each benchmark, the energy consumption is broken into 1) computation energy of PE-array, 2) the leakage of the PE-array, 3) the leakage of the memory, and 4) the dynamic energy of memory (and buffer combined). Note that the voltage of the memory is scaled to a lower voltage, as described in table 2.3. This choice was made as the cycle time of the PE's was significantly shorter than the memory cycle times. The scaling of the memory voltage increased its associated cycle time to one cycle, however, significantly reduced its dynamic and leakage power, making the PE-array energy consumption the largest energy consumer. In addition, note that by sequentially shaping the data in the memories, and usage of buffers, we significantly reduced the number of required memory accesses, resulting in a significant reduction in the dynamic power consumption of the memories. As illustrated, the TCD-NPE not only produces the fastest solution but also produces the least energy-consuming solutions across all NPE configurations, all data flows and all simulated benchmarks.

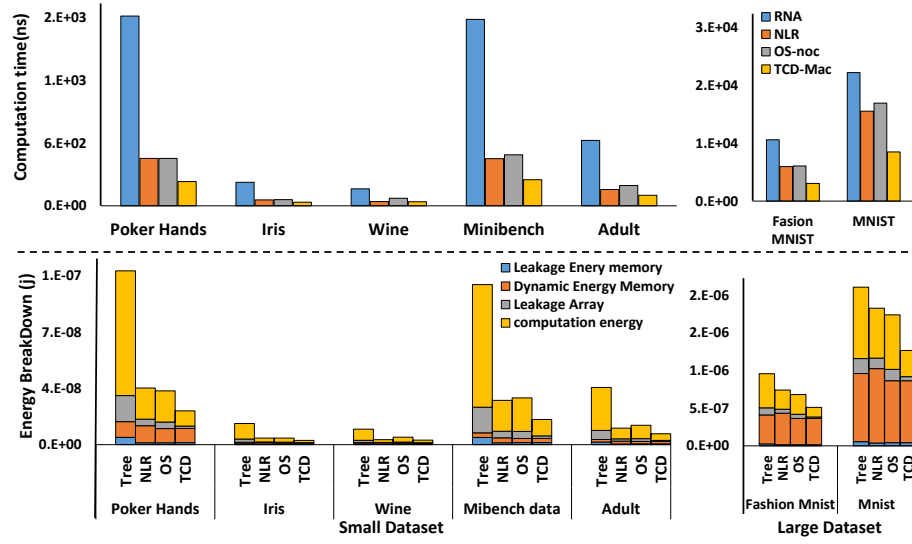


Figure 2.10: Comparison of TCD-NPE with an NPE constructed using conventional MACs that uses the OS, NLR, or RNA data flow. top): Execution time for various MLP benchmarks. Bottom): Energy consumption for various MLP benchmarks.

2.5 Conclusion

In this chapter, we introduced the concept of temporal carry bits and used the concept to design a novel MAC for efficient stream processing (TCD-MAC). We further proposed the design of a Neural Processing Engine (TCD-NPE) that is architected using an array of TCD-MACs as its processing element. We, further, proposed a novel scheduler that schedules the sequence of events to process an MLP model in the least number of computational rounds in the proposed TCD-NPE. We reported that the TCD-NPE significantly outperform similar neural processing solutions that are constructed using conventional MACs in terms of both energy consumption and execution time (performance).

Chapter 3: NESTA: Hamming Weight Compression-Based Neural Processing Engine

3.1 Introduction and Background

Deep learning models that deploy Convolutional Neural Networks (CNN) for feature extraction have become increasingly popular in recent years [51]. The popularity of these learning solutions stems from their ability to achieve unprecedented accuracy, surpassing that of human’s ability, for various tasks such as object and scene recognition [3,4,27,28,31,52–55], object detection, and object localization[56,57]. This, as illustrated in Table 3.1, is made possible by using deep and complex neural networks expressed using specialized frameworks such as Caffe [58], PyTorch [59] and Tensorflow [5], and trained and executed in acceptable time by Graphical Processing Units (GPU).

Although innovation in parallel computing has enabled us to train and execute such complex models, the applicability of such models remains limited due to their computational and storage requirements. These state of the art CNNs require up to hundreds of megabytes for a model and partial result storage and 30k-600k operations per input pixel [10]. The high computational complexity of these models, in turn, poses energy (power) and throughput (delay) challenges to the underlying hardware. Typically, in such learning solutions the majority (over 90%) of computational complexity is for processing the convolution (CONV) layers [60].

The generality of a processing engine significantly affects the throughput and energy efficiency of neural processing hardware[39][9]. The more general the hardware, the less efficient (in terms of delay and power) the computation becomes. The least attractive solutions are generated by running CNNs on general-purpose CPUs. Utilizing more specialized

hardware such as GPUs and FPGAs provide a significant improvement in the efficiency of computation, while most efficient computing, with an order(s) of magnitude improvement in performance and power consumption, is reported when specialized ASIC accelerators such as Eyeriss[10], Diannao[7], Dadiannao[8], or Shidiannao[37] are deployed. The major difference in the performance of ASIC accelerator solutions, previously proposed in [7–10, 37, 39, 63–66], is on the type of data flow implemented for maximizing data reuse (weight, partial sum, and activation value) and minimizing memory access. This is when the neural Processing Elements (PE), that implement the multiply-accumulate (MAC) function, remain non-optimized in these accelerator solutions.

Table 3.1: Depth and complexity of some of the existing and modern CNN solutions for object detection.

	AlexNet[4]	VGG[3]	GoogLeNet[52]	Resnet[53]
Top5 Accuracy	80.2%	89.6%	89.9%	96.3%
layers	8	19	22	152
FLOPS	729M	19.6G	1.5G	11.3G
FLOPS in 3×3 CONV	118M	19.5G	1.18G	6.7G

We claim that the architecture of PEs in an ASIC DNN accelerator could significantly improve when the computational model, data locality, and data reuse concepts are used to architect a DNN/CNN specific PE. We propose NESTA as a PE that is designed based on these principles. To reduce data movement, and minimize the generation of partial sums, NESTA consumes 9 values of the convolution at a time (equal to the size of a 3×3 convolution) until all filter-image pairs of a convolution across all channels are consumed. To significantly speed up the computation and reduce energy consumption, NESTA does not use adders or multipliers. Instead, it converts the convolution into a sequence of N compression and one final addition. The add operation transforms the compressed and accumulated

result into a correct partial sum.

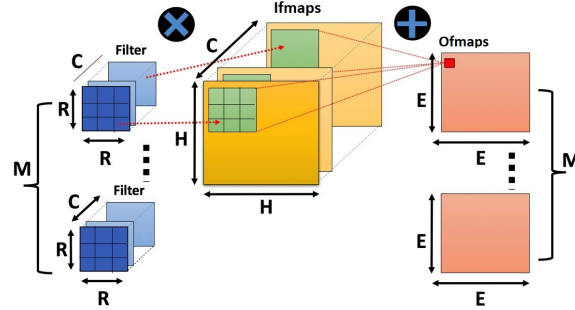


Figure 3.1: *Computing one CONV layer using input Ifmap/image and filters to produce the output (Ofmaps)*

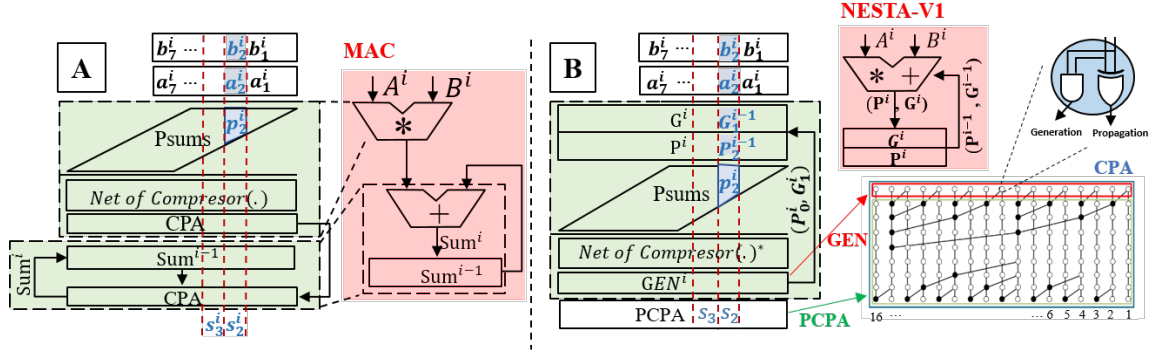


Figure 3.2: comparing the architecture of A) a typical MAC, versus B) a simplified 2-input version of NESTA. In all variables in the form of D_m^i , the subscript (m) captures the bit position values, and postscript (i) capture the cycle (iteration). For example, A^i, B^i are the input data in the i^{th} iteration (corresponding to the i^{th} cycle) of the multiply accumulate operation. The b_m^i, a_m^i , and p_m^i are accordingly the m^{th} significant bits of inputs A, B , and partial sum at the i^{th} cycle (iteration). The division of CPA into GEN and PCPA is also shown in this figure. Note that the PCPA is only executed at the last cycle.

3.2 NESTA: Proposed Processing Engine

Before describing our proposed solution, we first explain the concept of *temporal carry* in a miniaturized solution in section 3.2.1, then we explain the concept of *compression and expansion* in section 3.2.2. Finally, in section 3.2.3, we use these concepts to construct and describe our proposed solution.

3.2.1 Motivation 1: Temporal Carry

Suppose two vectors A and B each have N 8-bit values, and the goal is to compute their dot product, $\sum_{i=0}^{N-1} (A_i * B_i)$ (similar to what is done during the activation process of each neuron in a NN). This could be accomplished using a single Multiply-Accumulate (MAC) unit and working on 2 inputs at a time for N rounds. Fig. 3.2(A-right) shows the General view of a typical MAC architecture that comprised of two parts multiplication and addition. We have assumed that a Carry Propagation Adder (CPA) is used as adder unit for reducing the MAC delay. More detailed view of this architecture, 3.2(A-left), reveals that for generating

the final result, the CPA will be executed $2N$ times, N times for producing the results of N multiplications and N times for accumulating the result of multiplications. These CPAs are located at the critical path of this architecture so eliminating them lead to a performance gain. Fig. 3.2(A-right) captures how CPA has been broken into GEN (which is highlighted in red), and PCPA (Partial CPA).

Fig.3.2(B-right) shows a simplified version of our proposed solution, NESTA-V1. As illustrated NESTA-V1, 1) intertwines the multiplication and addition, and 2) reduces the delay of CPA by only using the GEN section of the CPA. The GEN section only produces the first level generate G^i , and propagate P^i signals, after which NESTA-V1 feedback each P^i and G^i to its compressor network for inclusion in the cycle computation. We can consider this as the process of *generating a temporal carry signal, as opposed to a spatial carry signal* which is used in typical MACs. This is made possible, considering that we do not need the output of individual multiplications, and our target is to compute the correct $\sum_{i=0}^{N-1} (A_i * B_i)$. Hence, in NESTA-V1 for $N-1$ times, only the GEN section of CPA is executed, while for the last iteration the complete CPA is executed (including PCPA) to avoid generating further temporal carry bits.

3.2.2 Motivation 2: Compression and Expansion

Lets consider an application that requires hardware acceleration for computing the following expression: $p = \sum_{i=1}^9 a_i$, in which $a_i(s)$ are 16-bit unsigned numbers. One natural solution, as illustrated in Fig. 3.3.(left), is using an adder-tree, while each add operator could be implemented using a fast adder such as carry-look-ahead [69] (CLA), Brent-Kung [70] (BK) or Kogge-Stone [71] (KS) adder. Regardless of the choice of the adder, the resulting adder tree is not the most efficient. The adder power delay product (PDP) could significantly improve if a multi-input adder is reconstructed using Hamming Weight (HW) compressors. For this purpose, we reformulate the computation of p as shown in Equation 3.1, by rearranging the values into 16 arrays, where each array is composed of 9 bits with equal significance value.

With this formulation, we can use a hierarchy of Hamming Weight compressor to perform the addition.

$$p = \sum_{i=0}^{15} \sum_{j=1}^9 (2^i \& a_j) \quad (3.1)$$

Fig. 3.3-(right) captures the structure of the proposed HW compression Adder (HWC-Adder), which is composed of 4 stages. In each of the first 3 stages, the HW compressors $C(m:n)$ take a stack of m bit values of the same significance (shown vertically) and computes its HW value (of size n) which is expanded vertically. Aligning the bit values of the same significance generates a smaller stack of bit values at each bit position as input to the next level of compressors. We refer to each of these stages (stages 1 to 3) as Compression and Expansion Layer (CEL). In the last stage, every bit-column contains no more than 2 bits. In this stage, a 2-input addition generates the final results.

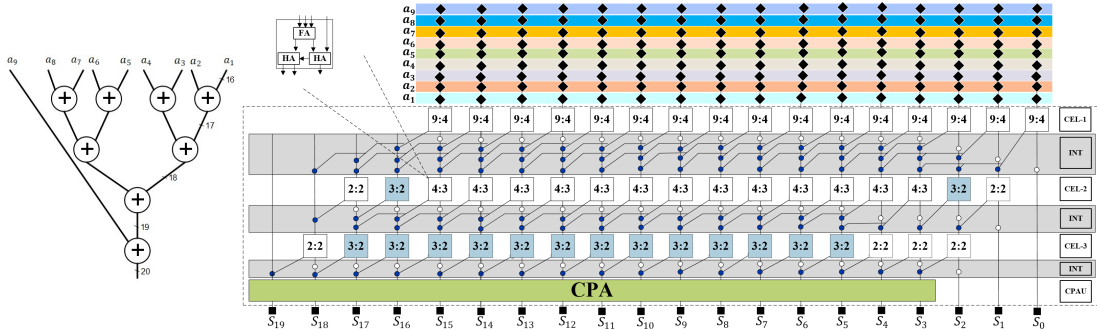


Figure 3.3: An Adder tree for 9 16-bit-width entries (left), Hamming Weight Adder (HW-Adder) of 9 16-bit-width entries (right). In the HWC-Adder compressor hierarchy (CEL) the complete compressors are colored blue, while compressors with available capacity are white. For building the improved version of HWC-Adder (HWC-Adder*), 2 bits from each compressor in CEL-1 are diffired to a compressor in the same bit position in CEL-2, increasing the number of complete compressors and reducing the critical path delay in CEL-1 as reported in table 3.2

Table 3.2 compares the PPA and PDP of an adder tree constructed using Brent-Kung and Kogge-Stone adders, and that of HWC-Adder. As illustrated the energy consumption of the HWC-Adder is 50.2% and 39.8% lower than that of the BK and KS adder-trees respectively. At the same time, the delay of HWC-Adder is 8.3% and 9.8% lower than that of the KS and BK adder-trees respectively. The delay of HWC-Adder architecture could be further improved, if instead of incomplete C(9:4) HW compressors in the first CEL, we use complete CC(7:3) compressors, passing the unconsumed bits (2 bits) to the next hierarchy layer, in which the C(4:3) incomplete compressors are converted to C(6:3). This transformation shortens the critical path (reduces the number of logic levels) in stage CEL-1 and reduces the area, without increasing the number of logic levels in CEL-2, hence, producing a faster implementation. The PDP of the resulting HWC-Adder* is captured in the table 3.2. The resulting improvements in the HWC-Adder(s) are the result of 1) using larger HW compressors (as opposed to C(2:2) and C(3:2) in Brent-Kung), and 2) maximizing the number of complete compressors, thus reducing the hardware deficiency.

Table 3.2: Comparing the efficiency of HWC-Adder(s) vs Adder tree constructed using Brent-Kung (BK) and Kogge-Stone (KS).

Adder Type	Area(μm^2)	Delay(ns)	Power(μW)	PDP(fj)
Adder tree (BK)	4723	2.66	0.555	1.48
Adder tree (KS)	5135	2.60	0.686	1.78
HWC-Adder	4738	2.40	0.369	0.88
HWC-Adder*	4428	2.35	0.368	0.86

3.2.3 NESTA: Our Proposed Solution

Our proposed solution, NESTA, is a specialized neural processing engine designed for executing learning models in which filter-weights, input-data, and applied biases are expressed

in fixed-point format. NESTA combines 9 multiplications and 9 additions into one batch-operation for gaining energy and performance benefits. Let's assume $NESTA_{ACC}$ is the current accumulated value, while I and W represent the input values and filter weights respectively. In its n^{th} round of execution, NESTA performs the following operation:

$$NESTA_{ACC}(n) = NESTA_{ACC}(n-1) + \sum_{i=9n}^{9n+9} I_i \times W_i \quad (3.2)$$

To improve efficiency, NESTA does not use adders and multipliers. Instead, it uses a sequence of hamming weight compressions followed by a single add operation. Furthermore, in each cycle c , after consuming 9 input-pairs (weight and input), instead of computing the correct accumulated sum, NESTA quickly computes an approximate partial sum $S'[c]$ and a carry $C[c]$ such that $S[c] = S'[c] + C[c]$. The $S'[c]$ is the collection of generated bits (Gi) and $C[c]$ is the collection of propagated (Pi) bits produced by GEN unit of CPA. Note that the division of CPA into GEN and PCPA was described in section 3.2.1. The $S'[c]$ is saved in the output registers, while the $C[c]$ are stored in Carry Buffer Unit (CBU) registers. In the next cycle, both $S'[c]$ and $C[c]$ are used as additional inputs (along with 9 new inputs and weights) to the CEL unit. Saving the carry (propagate) values (Ps) in CBU and using them in the next iteration reflects the temporal carry concept that was described in section 3.2.1, while the reuse of S' in the next round implements the accumulation function of NESTA.

In the last cycle, when working on the last batch of inputs, NESTA computes the correct $S[c]$ by using the PCPA to consume the remaining carry bits and by performing the complete addition $S[c] = S'[c] + C[c]$. Note that the add operation generates a correct partial sum whenever executed. But, to avoid the delay of the add operation, NESTA postpones it until the last cycle. For example, when processing a 11×11 convolution across 10 channels, to compute each value in Ofmap, 1210 ($11 \times 11 \times 10$) MAC operations are needed. To compute this convolution, NESTA is used 135 times $\lceil 1210/9 \rceil$, followed by one single add

operation at the end to generate the correct output.

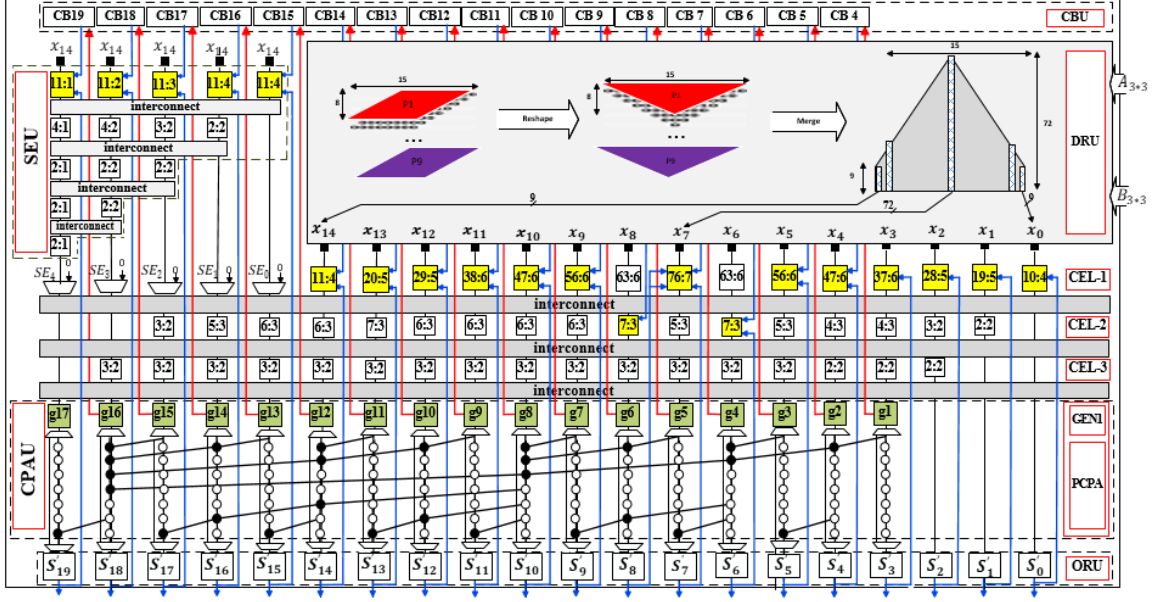


Figure 3.4: In NESTA carry bits that are generated in GEN section of the CPA do not propagate into the carry chain. Instead, they are captured by CB registers. In the next cycle, the carry bits (of the previous cycle that are stored in CB registers) are fed to the hamming weight compressors at that bit position, temporally deferring their impact to the next cycle. The compression unit, in each cycle consumes the bit values from 9 new input (W, I) pairs, the Carry bits of the previous cycle (stored in CB registers) and the partial sum stored in S registers. The consumption of bit values in S registers implement the accumulation function. In the last round of computation, instead of capturing the carry bits in CB registers, they are fed to the PCPA (Partial CPA) to propagate through the carry chain and generate the correct convolution results.

Fig. 3.4 captures the NESTA architecture. It is comprised of 6 units: 1) Data Reshaping Unit (DRU), 2) Sign Expansion Unit (SEU), 3) Compression and Expansion Layers (CEL), 4) Adder Unit (AU), 5) Carry Buffer Unit(CBU), and 6) Output Register Unit(ORU). Each of these units is described next:

Data Reshape Unit (DRU)

The DRU, as illustrated in Fig. 3.4-(DRU), receives 9 pair of multiplicands and multipliers (W and I), converts each multiplication to a sequence of additions by ANDing each bit value of multiplier with the multiplicand and shifting the resulting binary by the appropriate amount, and returns bit-aligned version of the resulted partial products.

Sign Extension Unit:(SEU)

The SEU is responsible for producing the sign bits SE_0 to SE_4 . The inputs to SEU is sign bit (X_{14}). The result of a multiplying and adding 9, 8-bit values is at most 20-bits. Hence, we need to sign-extend each one of the 15-bit partial sums (for supporting larger the architecture is accordingly modified). To support signed inputs, we also need to slightly change the input data representation. For a partial product $p = a \times b$, if one values a or b is negative, we need to make sure that the negative number is used as the multiplier and the positive one as the multiplicand. With this arrangement, we treat the generated partial sums as positive values and make a correction for this assumption by adding the two's complement of the multiplicand during the last step of generating the partial sum. This feature is built into the architecture using a simple 1-bit sign detection unit, and by adding multiplexers to the output of input registers to capture the sign bits. Note that multiplexers are only needed for the last 5-bits as shown in figure 3.4-(SEU). Following example clarify this concept: let's suppose that a is a positive and b is a negative b-bit binary. The multiplication $b \times a$ can be reformulated as:

$$b \times a = (-2^7 + \sum_{i=0}^6 x_i 2^i) \times a = -2^7 a + (\sum_{i=0}^6 x_i 2^i) \times a \quad (3.3)$$

The term $-2^7 a$ is the two's complement of multiplicand which is shifted to the left by 7 bits, and the term $(\sum_{i=0}^6 x_i 2^i) \times a$ is only accumulating shifted version of the multiplicand. Note that some of the output bits generated by SEU compressor extend beyond 20 required

bits. These sign bits are safely ignored. Finally, the multiplexers switch at the output of SEU is used to allow NESTA to switch between signed and unsigned modes of operation.

Compression and Expansion Layers (CEL)

The input to i th bit of CEL unit in cycle n is the 1) bit-aligned partial sums (at the output of DRU) in position i 2) the temporary sum generated by GEN unit of NESTA at time $c-1$ at bit position i , and 3) the Propagate (carry) value generated by GEN unit of NESTA at time $c-1$ at bit position $i-1$. Following the concept of HWC-Adder, described in section 3.2.2, the CEL is constructed using a network of Hamming Weight Compressors (HWC). A HWC function $C_{HW}(m:n)$ is defined as the Hamming Weight (HW) of m input-bits (of the same bit-significance value) which is represented by an n -bit binary number, where n is related to m by: $n = \lfloor \log_2^m \rfloor + 1$. For example "011010", "111000", and "000111" could be the input to a $C_{HW}(6:3)$, and all three inputs generate the same Hamming weight value represented by "011". A Completed HWC function $CC_{HW}(m:n)$ is defined as a C_{HW} function, in which m is $2^n - 1$ (e.g., $CC(3:2)$ or $CC(15:4)$). As illustrated in Fig.3.4, each HWC takes a column of m input bits (of the same significance value) and generate its n -bit hamming weight. The resulting n bits is then horizontally distributed as input to $C_{HW}(s)$ in the next-layer CEL. This process is repeated until each column contains no more than 2-bits.

Carry Propagation Adder Unit(CPAU)

Similar to HWC-Adder, described in section 3.2.2, the CPA is divided into GEN and PCPA. If NESTA is executed n times, the PCPA is skipped $n-1$ times and is only executed in the last iteration. GEN is the first logic level of CPA executing the generate and propagate functions to produce temporary sum/generate G and carry/propagate P which are used as input in the next cycle.

Carry Buffer Unit (CBU)

The CBU is a set of registers that store the propagate/carry bits generated by GEN at each cycle, and provide this value to CEL unit in the next cycle. Note that CB bits can be injected to any of the $C_{HW}(m : n)$ in any of the CEL layers in that bit position. Hence, it is desired to inject the CB bits to an incomplete $C_{HW}(m : n)$ to avoid an increase in the critical path delay of CEL.

Output Register Unit (ORU)

The ORU capture the output of GEN in the first $n-1$ cycles or PCPA in the last cycle of operation. Hence, in the first $n - 1$ cycle, NESTA stores the Generate (G) output of GEN unit and feeds this value back to the CEL unit in the next cycle. In the last cycle, it stores the sum generated by PCPA.

3.2.4 NESTA: Putting it all together

NESTA receives 9 pair of Ws and Is. The DRU generate the partial products and bit-align them as input to the CEL unit. The CEL unit at each round of computation consumes 1) bit values generated by DRU, 2) generate (temporary sum) values stored at S registers, and 3) propagate (carry) bits in CB registers. This is when the SEU assures that the sign bits are properly generated. For the first n cycles, only the GEN unit of CPA is executed. This allows NESTA to skip the delay of the carry chain of the PCPA. To be efficient, the clock period of NESTA is reduced to exclude the time needed for the execution of PCPA. The timing paths in PCPA are defined as multi-cycle paths (2 cycle paths). Hence, the execution of the last cycle of NESTA takes 2 cycles, see Fig. 3.5. In the last round of execution, the PCPA unit is activated, allowing the addition of stored values in S registers and CB registers to take place for producing the correct and final SUM. Considering that the number of channels in each layer of modern CNNs is fairly large (128 to 512) the savings in the result of shortening NESTA cycle time (by excluding PCPA) accumulated over large number of cycles (of NESTA execution) is far larger than one additional cycle needed at

the end to execute the PCPA for producing the correct final sum.

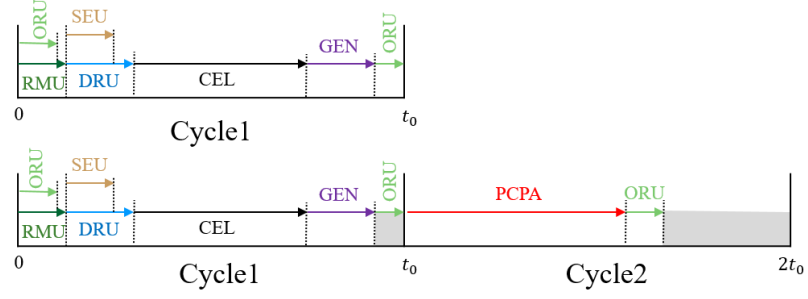


Figure 3.5: NESTA cycle time is computed by excluding the execution time of PCPA. In the last cycle of computation of convolution, the NESTA activates the PCPA and captured the correct sum after 2 cycles of execution.

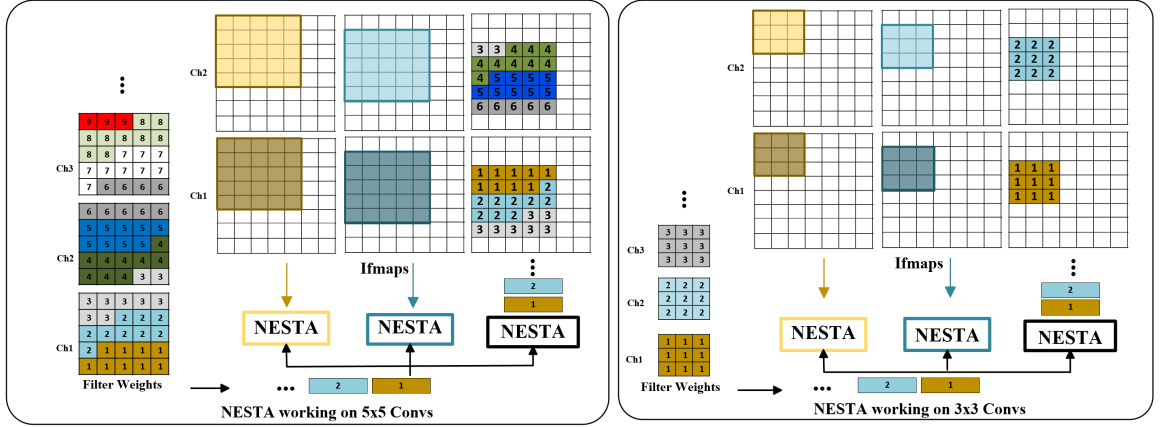


Figure 3.6: NESTA Row Stationary (RS) data flow for executing 3×3 convolution across multiple channels (right) and 5×5 convolution across multiple channels (left). A similar concept can be used to support all other convolutions sizes.

3.2.5 Supported Data Flows

A considerable portion of the power consumed in a neural processing engine is related to storage, read and write from its memory subsystem. The extent of power consumed in the memory subsystem is a function of 1) the read/write/retention power of each memory element, and 2) the frequency of access to each memory. In several prior work [46–50], it was shown that it is possible to significantly reduce the read/write/retention power consumption of a memory unit by aggressively scaling its supplied voltage while deploying architectural fault tolerance techniques and solutions to mitigate the increase in the memory write/read/retention failure rate. The frequency of access to the memories, on the other hand, can not be controlled from an architectural perspective as it is a dataflow optimization problem.

Memory access pattern of a model which is being executed on a neural engine significantly impacts its energy efficiency and performance. Accessing data in off-chip DRAM consumes around 200X more energy and takes around 20X longer compared to accessing data in on-chip SRAM memories [10][72][73]. Hence, for a modern Deep Neural Network

with a large number of operations and parameters, designing a dataflow that minimizes the access to off-chip DRAM and maximizes the data reuse (while data is on-chip) can go a long way in improving its energy efficiency and performance. Related to neural processing engines, several dataflows has been studied in the literature. The work in [10] divides the DNN dataflows into 5 major categories: 1) No Local Reuse(NLR), 2) Weight Stationary(WS), 3) Input Stationary(IS), 4) Output Stationary(OS), and 5) Row Stationary(RS, RS+). These data flows differ on the way they reuse input frame maps (Ifmaps), partial sums (Psums), and filter weights. The NLR does not have any reuse at the PE level and requires the largest number of transaction with a global buffer. Diannao is an example of NLR based accelerator described in [7]. The WS dataflow stores the filter weights within the PEs. The goal is to minimize the re-fetching of filter weights by limit their movement. Examples of WS implementation include [74][75]. IS and OS dataflows try to minimize the movement of Ifmaps and Psums respectively, examples of which include [9][36][37]. The RS dataflow combines the WS and the OS dataflows to achieve greater efficiency. Eyeriss is an example of RS implementation described in [10][76].

Another way to understand the differences between these dataflows is through the study of the algorithm governing the computation of the convolution in each data flow. Let us consider the convolution in Fig. 3.1 with M filters (each with size $C \times R \times R$), repeated in a batch of B images with each image being of size $C \times H \times H$. To process this CONV, as shown in Alg. 2, seven nested loops are required. Because each one of the loops is independent of the others, changing the order of each these loops can produce a new dataflow. Each dataflow promotes a different form of data reuse. It should be noted that it is possible that one permutation of these nested loops to be applicable to more that one dataflow. For example in the Alg. 2, execution line order 1-2-3-4-5-6-7-8, NLR, WS, and RS have the same representation, however, depending on the underlying NOC different data access patterns can be designed. Os and IS dataflows also can be obtained if the execution's line of the seven loops changes to 1-2-4-5-3-6-7.

Algorithm 2 seven nested loops for calculating an Ofmap. B, M, C, H, R are Batch-size, Number of Filters, Channel size, Height/weight of an ifmap, and filter size respectively.

```

1: for ( $b = 0; b < B; b++$ ) do
2:   for ( $u = 0; u < M; u++$ ) do
3:     for ( $c = 0; c < C; c++$ ) do
4:       for ( $h = 0; h < H; h++ = S$ ) do
5:         for ( $w = 0; w < H; w++ = S$ ) do
6:           for ( $i = 0; i < R; i++$ ) do
7:             for ( $j = 0; j < R; j++$ ) do
8:               ofmap[b][u][h][w] += ifmap[b][c][h+i][w+j]*
9:               filter[u][c][i][j]
```

NESTA could be used to implement any of these dataflows. However, in this work (for lack of space), we only describe how NESTA dataflow could be designed to model the RS dataflow and will address the implementation of other dataflows for our future work. Fig. 3.6 capture the RS dataflow used to compute 3×3 (right) and 5×5 (left) convolution across many channels. To capture the data reuse and communications between NESTA cores (assuming that many NESTA cores are packed into a SOC to build a many-core accelerator), we have used three NESTAs to construct each of scenarios illustrated in Fig. 3.6. Since NESTA accept 9 inputs at a time, it can perform a 3×3 convolution in one cycle, or a 5×5 convolution in 3 cycles. The data is reshaped in the accelerator’s global buffer and is streamed to the NESTA cores. Depending on the number of available NESTA cores we can compute the value of different neurons in parallel to promote higher data reuse. In this case we can either 1) compute the neurons in different OFmaps by loading different weights to each group of NESTA and share the ifmap weights (not shown in this figure), or 2) compute the neuron values in the same Ofmap by sharing the weights across different NESTA cores and stream different (partially overlapping) ifmap values to each group as shown in Fig. 3.6.

As described in section 3.2.4, in its last round computations(when working on convolution across multiple channels), NESTA switches to its two-cycle operation mode and activates the PCPA that would take 2 cycles to generate the correct final sum. Note that in deep channels, or for large convolutions, the cost of one extra cycle is negligible compared to the gain of removing the PCPA from the critical path in all computational cycles.

3.3 Results

In this section, we evaluate the NESTA in terms of Power, Performance, and Area (PPA). NESTA and all MACs cited in this section support 16-bit signed fixed-point inputs.

3.3.1 Evaluation and Comparison Framework

The PPA metrics are extracted from the post-layout simulation of each design. Each MAC or MAC9 is designed in VHDL, synthesized using Synopsis Design Compiler [43] using 32nm standard cell libraries, and is subjected to physical design (targeting max frequency) by using the reference flow provided by Synopsys and by using IC Compiler [44]. The area and delay metrics are reported using Synopsys Primetime [45]. The reported power is then averaged across 20K cycles of simulation with random input data fed to PrimetimePX [45] in FSDB format. To build a fair comparison, in addition to simple 2-input MACs, we constructed multiple flavors of 9-input MACs (MAC9s) using various high-speed adders and multipliers to compute the convolution in one shot. The general structure of MACs and MAC9s used for comparison is captured in Fig. 3.7. Each MAC9 is constructed using 9 multipliers, the output of which is fed to a 10-input adder tree (9 inputs from multiplier and 1 from output register) to compute a 3×3 convolution in one shot. For multiplication, we used Booth-Radix-N (BRx2, BRx4, BRx8), and Wallace multipliers. For addition, we used Brent-Kung (BK) and Kogge-Stone (KS) adders. In addition, we considered a hybrid approach, where the multipliers are fed to an HWC-Adder which was discussed in section 3.2.2. Each 2-input MAC is identified by (Multiplier choice, Adder choice) and each 9-input MAC9 is identified by (Multiplier Choice, (Adder Arrangement, Adder Choice)). For example (BRx2, (tree, Brent-Kung)) is a MAC9 constructed by using 9 BRx2 multipliers followed by an adder tree constructed from Brent-Kung Adders. Similarly, a (BRx2, (HWC-Adder, Brent-Kung)) uses the same multiplier, but replace the adder tree with an HWC-Adder that uses a single Brent-Kung adder.

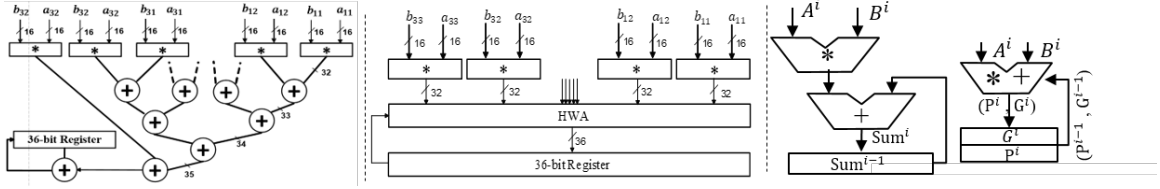


Figure 3.7: A 9-input MAC, which is identified as (Multiplier Choice, Adder Choice). MACs constructed with similar structure are used for PPA and PDP comparison with our proposed NESTA PE solution.

3.3.2 PPA efficiency: NESTA v.s. MAC9s

Power: The power consumption of NESTA is considerably less than other MAC9 flavors. When comparing NESTA with various flavors of MAC9, the power consumption is reduced by 17.4% to 58.9% when compared to (BRX4, (HWA, BK)) and (BRX2, (Tree, KS)) representing the MAC9s with lowest and highest power consumption, respectively.

Performance: In terms of delay, the delay of NESTA is better than all other MAC9 flavors. For example, the delay of NESTA is 23.7% and 11.3% better than (BRX2, (Tree, BK)) and (BRX4, (HWA, KS)) as the slowest and fastest MAC9s in Fig. 3.11. However, the reduction in the delay comes with a catch; When NESTA process the last batch of inputs of the last channel, it has to take one extra cycle to perform the final addition. Hence, energy efficiency becomes a function of the number of processed input batches. This tradeoff is illustrated in Fig.3.8. The larger the number of input channels, the smaller the overhead of one extra cycle for the final addition. As illustrated in Fig. 3.8, NESTA becomes more efficient if the number of processed input batches is greater than 64, 8, 2, 1 for kernel size 1x1, 3x3, 5x5, 11x11 respectively.

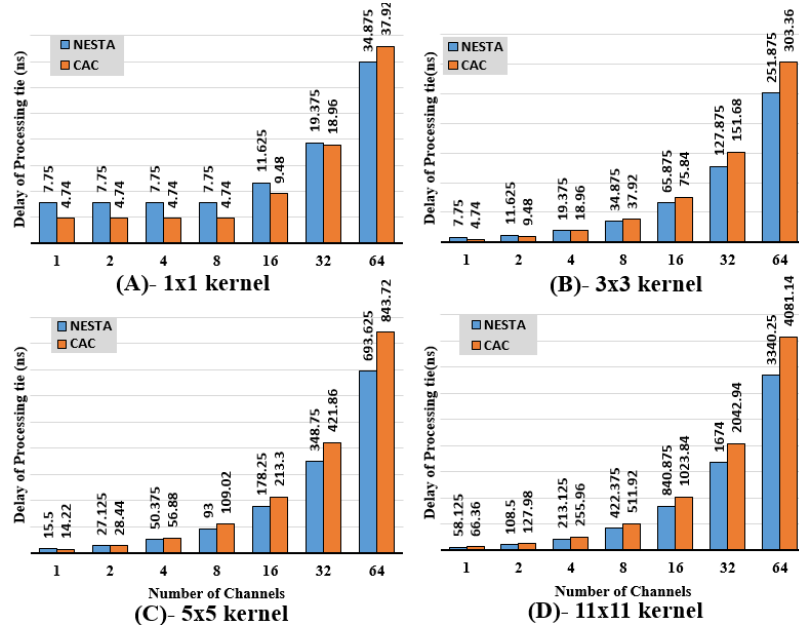


Figure 3.8: Comparing the processing time of a NESTA and a MAC9 for convolutions with (A) 1x1, (B) 3x3, (C) 5x5, and (D) 11x11 kernel size when the convolution expands over multiple channels. As illustrated NESTA for larger convolutions or deeper channels becomes very more efficient.

Area: Figure 3.11 captures the PPA comparison of NESTA with various flavors of MAC9s. As illustrated, NESTA is implemented in a smaller area. The area saving is between 6% to 9% when NESTA is compared with (BRX4, (HWA, BK)) and (BRX2, (Tree, KS)), which are the smallest and largest MAC9s, respectively.

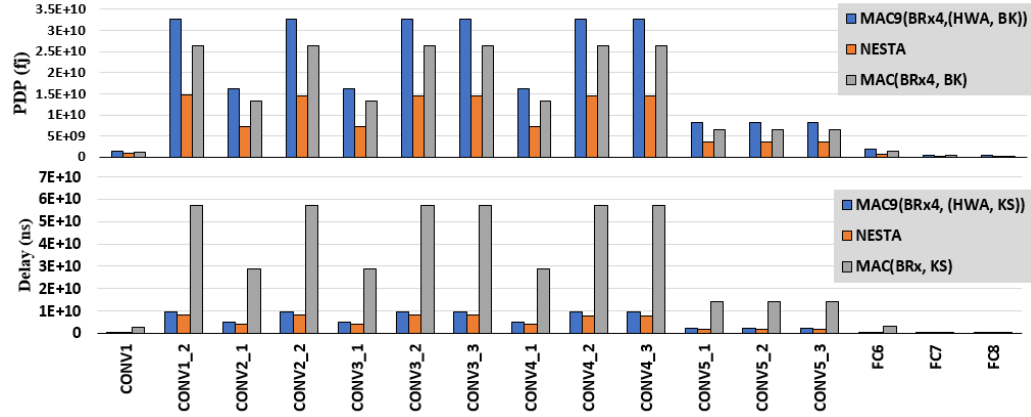


Figure 3.9: Breakdown of delay and energy consumption of each layer of VGG [3] when processed by a single MAC, a single MAC9 or a single NESTA core. A linear increase in the number of cores linearly reduces the processing time.

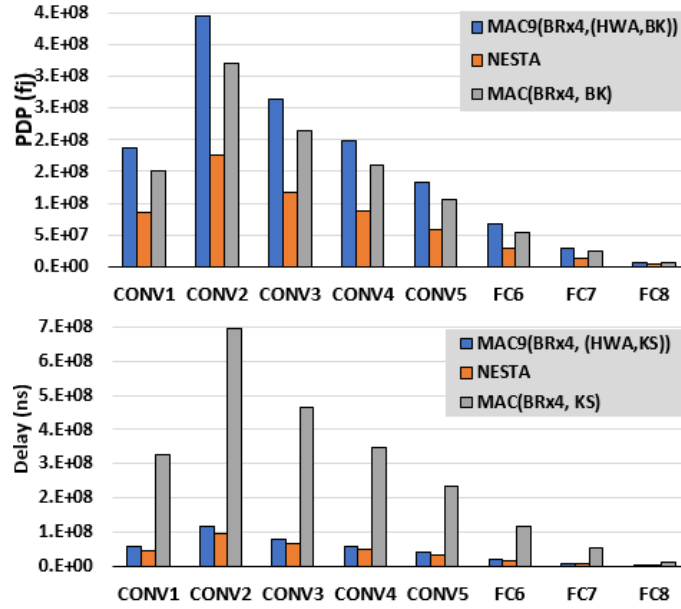


Figure 3.10: Breakdown of delay and energy consumption of each layer of AlexNet [4] when processed by a Neural engine composed of MACs, MAC9s or NESTA cores.

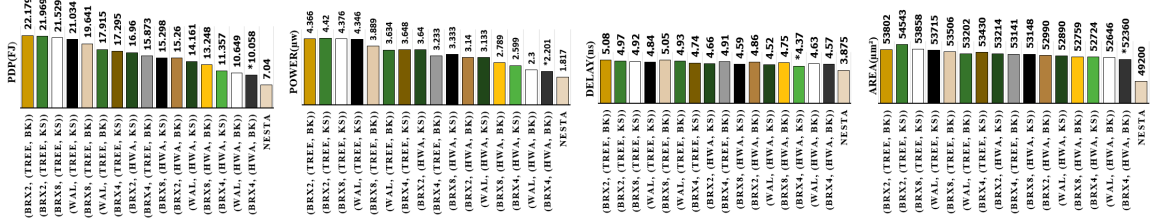


Figure 3.11: Area, Delay, Power, and PDP comparison between NESTA and MAC9s constructed using fast adders and multipliers. The star identifies the best MAC9 in each category.

PDP: Considering that NESTA has lower delay and power consumption compared to other MAC9s, the PDP savings for NESTA is even more significant. According to Fig. 3.11, NESTA reduces the PDP by 30% to 67% when compared to (BRX4, (HWA, BK)) and (BRX2, (Tree, KS)) that have the lowest and highest PDP respectively.

3.3.3 PPA efficiency: NESTA v.s. MACs

Table 3.3 captures the PPA metrics of various 2-input MACs and 9-input NESTA. Each single MAC has a smaller area, power, and delay compared to NESTA, however, in terms of functionally, one NESTA is equivalent to 9 MACs. Hence, For a fair comparison between NESTA and selected MACs, we compare their energy efficiency and throughput when fixing the area. For this comparison, we assume a NN accelerator assigns a fixed silicon area for instantiating 9-input NESTAs or 2-input MACs and report the improvement in throughput and energy with this constraint. Table 4.6 captures our comparison results. As illustrated, NESTA in terms of throughput (delay of processing normalized to the unit area) and energy efficiency (processing a large number of convolutions) is substantially more efficient than all MAC solutions studied. By using NESTA as the PE solution in an accelerator, the throughput improves between 1% to 37%, correspond to (Brx4, BK) and (BRx2, KS) respectively, and energy efficiency improves 33% to 78% when compared with NESTA-V1 and (BRx2, BK) which represent the best and worst MACs in terms of energy efficiency.

Table 3.3: PPA comparison between various MAC flavors and NESTA-V1 and NESTA.

MAC Type	Area(μm^2)	Power(μw)	Delay(ns)	PDP(fJ)
(BRx2, KS)	9394	0.612	3.57	2.24
(BRx2, BK)	9227	0.577	3.59	2.13
(BRx8, KS)	8123	0.523	3.5	1.88
(BRx8, BK)	7929	0.509	3.55	1.86
(WAL, KS)	7024	0.533	3.46	1.84
(WAL, BK)	7876	0.566	3.21	1.81
(BRx4, KS)	6899	0.480	3.10	1.48
(BRx4, BK)	6775	0.452	3.172	1.43
NESTA-V1	6825	0.442	2.914	1.287
NESTA	49200	1.817	3.875	7.04

Table 3.4: Percentage improvement in Throughput(left) & energy consumption(right) when using NESTA to process 1K of different convolution size.

MAC Type	3X3	5X5	7X7	11X11	3X3	5X5	7X7	11X11
(BRx2, KS)	37	37	37	37	65	62	62	64
(BRx2, BK)	36	36	36	36	78	76	76	77
(BRx8, KS)	26	26	26	26	58	55	54	57
(BRx8, BK)	25	25	25	25	58	55	54	56
(WAL, KS)	13	13	13	13	57	54	53	56
(WAL, BK)	16	16	16	16	57	53	52	55
(BRx4, KS)	1	1	1	1	47	43	42	45
(BRx4, BK)	1	1	1	1	45	41	40	43
NETSA-V1	30	30	30	30	39	34	33	37

3.3.4 NESTA for Efficient CNN Processing

In this section, we study the performance and energy consumption of a Neural Processing solution that uses 9-input NESTA, MAC9s, or 2-input MACs to process Alexnet[4] and VGG[3]. We only investigate the energy consumed for the processing the information and would address the saving (due to data reuse in NESTA) dataflow related power saving in the future work. Fig. 3.9 and Fig. 3.10 capture the delay and energy consumed for processing each layer (including CONVs and FCs layers) of Alexnet [4] and VGG [3] respectively. This is when the choice of processing engine is varied between MACs, MAC9s and NESTA cores. In each figure, NESTA is compared with the best choice of MAC or MAC9 for energy or delay according to the results of section 4.2.1 and 3.3.2. As illustrated, MAC9 solutions are faster than MAC's but consume more power. However, NESTA outperforms both MAC9 and MAC solutions in terms of both power and delay (and PDP) when processing each layer of AlexNet or VGG.

3.4 Conclusion

In this chapter, we introduced NESTA, a novel processing engine for efficient processing of Convolutional Neural Networks. NESTA benefits from 1) its ability to generate temporal carry bits that could be passed to be included in the next round of computation without affecting the overall results, and 2) the utilization of a hierarchy of compressors to efficiently compute 9 multiplication and additions at the same time. When computing the convolution across multiple channels and/or larger convolution window sizes, NESTA generates an approximate sum (S') and a temporal carry (P) in each cycle. In the last cycle, when processing the last convolution, NESTA takes an additional cycle and add the remaining carriers to the approximate sum to generate the correct output. Our post-layout simulation results report 30% to 67% reduction in power delay product (PDP) when NESTA is compared with various flavors of 9-input MAC units, and 33% to 78% reduction in PDP when compared with Neural processing engines constructed from various MAC flavors.

Chapter 4: TCD-MAC++: An Enhanced Version of Temporal Carry Deferring MAC

4.1 Introduction

In the previous chapters, we illustrated a detailed description of the components of a conventional MAC, Fused MAC, and also TCD-MAC. Fig. 4.1 briefs the data flow of the studied MAC in previous chapters. Fig. 4.1(A-top) shows the general view of a typical MAC architecture that is comprised of a multiplier and an adder (with 4-bit input width), while Fig. 4.1(A-bottom) provides a more detailed view of this architecture. Fig. 4.1(B) illustrates the architecture of a Fused MAC in which one of the CPAU is eliminated and its task is offloaded to the first Compression and Expansion Layer. Finally, Fig. 4.1(C) illustrates the proposed TCD-MAC architecture.

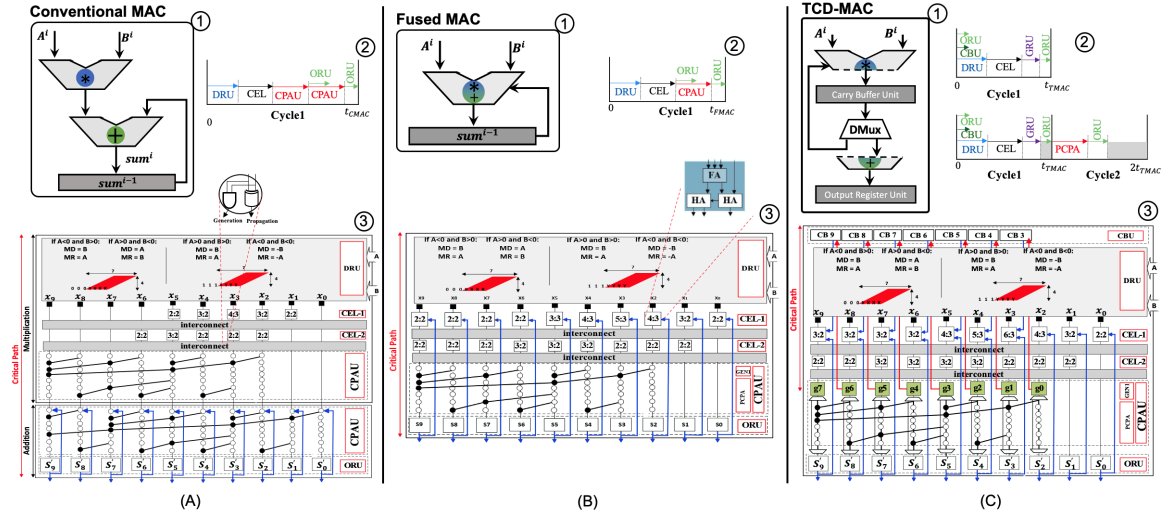


Figure 4.1: Comparing of three available architecture for processing a stream of MAC operations with operands A^i, B^i in the i^{th} iteration (corresponding to the i^{th} cycle) of the multiply-accumulate operation. A-1) an abstract view of a typical MAC, B-1) an abstract view of a Fused MAC, C-1) an abstract view of a TCD-MAC. A-2, B-2, C-2 are the activated components at each cycle of the mac architecture, and A-3, B-3, C-3 are the dataflows of each one of the MAC architecture for processing operands A^i, B^i .

TCD-MAC has two modes of operations 1) Carry Deferring Mode (CDM), in which the partial macs are calculated and the temporal carries are stored in CBU for use in the next round of operations. 2) Carry Propagation Mode (CPM), in which no temporal carry is generated and the carry bits in CBU are consumed to generate the result using a carry propagation adder (generating spatial carries). In the architecture of TCD-MAC, GEN which was the first layer of CPAU was the interface that specifies the boundary of components that are activated in CDM or CPM.

In this chapter, our previous work is extended, introducing TCD-MAC++, by adding the following functionalities and added features to our proposed solution:

- Building faster variants of TCD-MAC, by folding (unrolling) the first hamming weight layer, instead of propagating the data through a hierarchal tree of hamming weight compressors.
- Adding an additional mode of operation to the TCD-MAC, allowing the TCD-MAC

to be implemented in smaller area and power.

- Adding Early Free-Run Stopping (EFRS) indicator for showing the end of operation based on the desired precision i.e., EFRS allows the TCD-NPE to run in approximate mode through early termination.

4.1.1 Temporal Carry Deferring MAC++ (TCD-MAC++)

Fig 4.2 shows an abstract view of TCD-MAC and TCD-MAC++ architectures for performing MAC operation on a stream of operands A^i and B^i . Unlike the similarities between the data flow of TCD-MAC and TCD-MAC++, TCD-MAC++ has a larger number of registers in CBU, and also uses a larger set of Hamming Weight Compressors (HWC). The reason behind these differences is that at architectures of TCD-MAC++ some of the components in TCD-MAC have been truncated and their tasks have been offloaded to the other remaining components. Consequently, a larger set of HWC and CBU registers needed to be used to address the imposed extra calculations. Shortening the critical path of a TCD-MAC and introducing the concept of Early Free-Run Stopping (EFRS) are two major enhancements that have been included in TCD-MAC++.

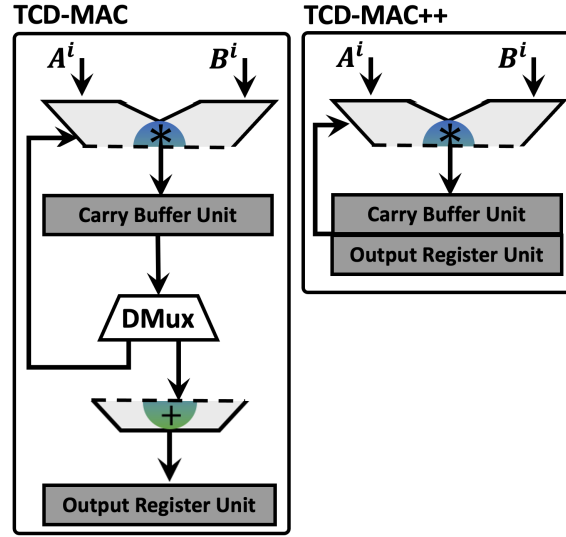


Figure 4.2: An abstract view of the architectures TCD-MAC and TCD-MAC++ for performing MAC operation on a stream of operands A^i and B^i .

4.1.2 Shorter Critical Path

In the architecture of TCD-MAC, GEN which was the first layer of CPAU was the interface that specifies the boundary of components that are activated in CDM or CPM. In TCD-MAC++ architecture, three other candidates for the interface GEN has been investigated. These three candidates have been shown in Fig. 4.3 under the names TCD-MAC++ $_{\alpha}$, TCD-MAC++ $_{\beta}$, TCD-MAC++ $_{\gamma}$.

In TCD-MAC++ $_{\alpha}$ the GEN interface is located right after the first CEL and similarly in TCD-MAC++ $_{\beta}$ and TCD-MAC++ $_{\gamma}$ the GEN interface is located after the second and third CEL, respectively. Between these three architectures, TCD-MAC++ $_{\alpha}$ has the lowest critical path at each round of operation and at the same time has the larger number of needed registers in the CBU. Similarly, TCD-MAC++ $_{\gamma}$ has the largest critical path and the lowest number of registers in CBU. The critical path and number of CBU's registers in

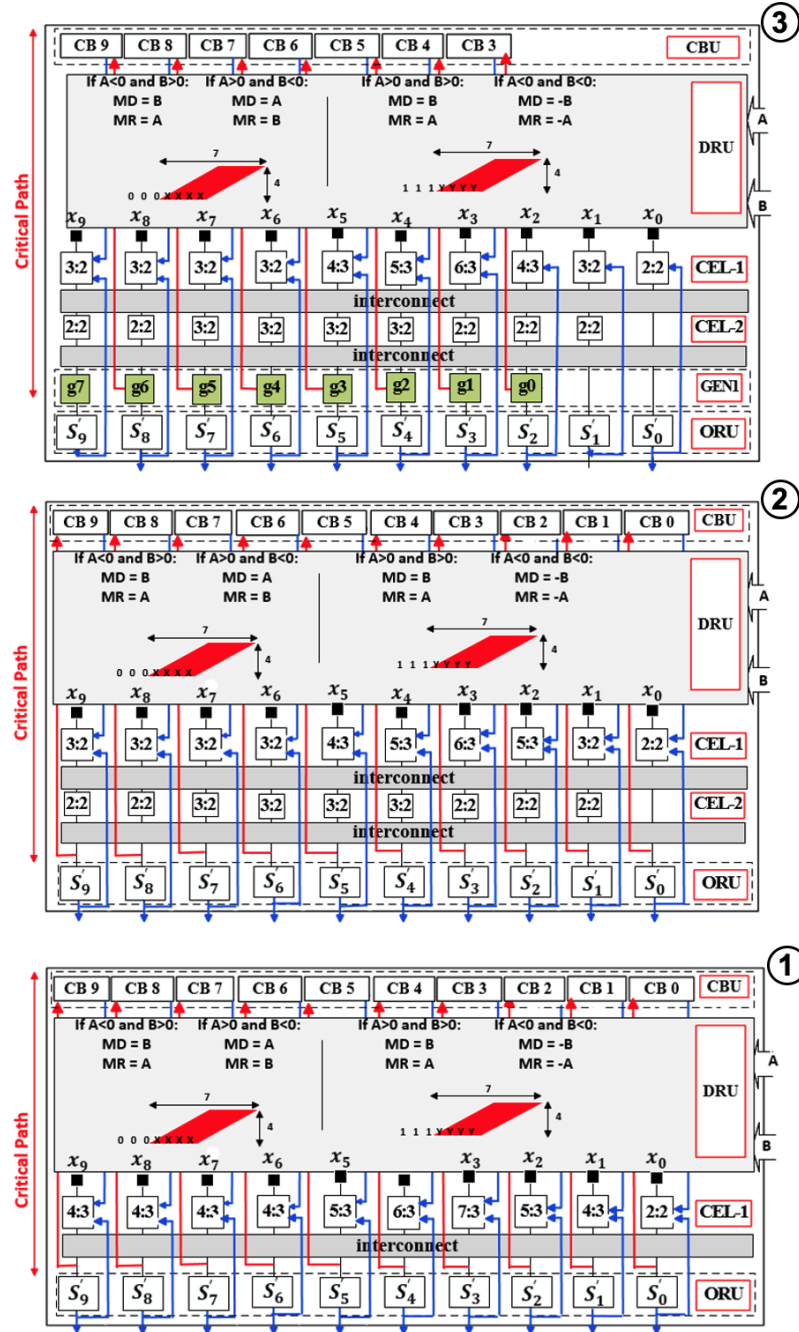


Figure 4.3: Three versions of TCD-MAC++ for calculating mac operations with 4-bit width signed operands. 1) TCD-MAC++_α: deferring carry bits from the first layer of expansion layer. 2) TCD-MAC++_β: deferring the carry bits from the second layer of expansion layer 3) TCD-MAC++_γ: deferring the carry bits from the third layer of expansion layer (available mac operation with a higher bit-width operands)

the TCD-MAC++ $_{\beta}$ is in between two other architectures.

4.1.3 Early Free-Run Stopping (EFRS)

Unlike the TCD-MAC that the last mode of operation was needed to be CPM, in TCD-MAC++ the result can be generated by running only in CDM. This enhancement makes the architectures of TCD-MAC++ much faster than the TCD-MAC. This performance is at the expense of having more free running at the final rounds of operations. When TCD-MAC++ operates in free-running, the input operands are zero, and only the values in the CBU and ORU are involved in calculations. This free-running continues to a point that the final result is generated. For each version of TCD-MAC++ the maximum number of free-running can be obtained. The maximum number of free runs for TCD-MAC++ $_{\alpha}$, TCD-MAC++ $_{\beta}$, and TCD-MAC++ $_{\gamma}$ are 36, 35, and 34 iteration, respectively. However, the results show that the calculations can be terminated much sooner than reaching the maximum number of free-running.

Figure 4.4 illustrates the used architecture for detecting the end of operations. This feature is embedded inside TCD-MAC++ and is called Early Free-Run Stopping (EFRS). In EFRS the values Pr_0 to Pr_{35} are precision indicators. Each of these indicators corresponding to a different range of bit significance from one another. For example, In Fig. 4.4 the green paths depict a scenario that the value of Pr_j is '1', which indicates the current ORU and the previous one have the same value in their bit significance positions higher than j . Noted that $Pr_{j=0}$ and $Pr_{j=35}$ have the highest and lowest precision, respectively.

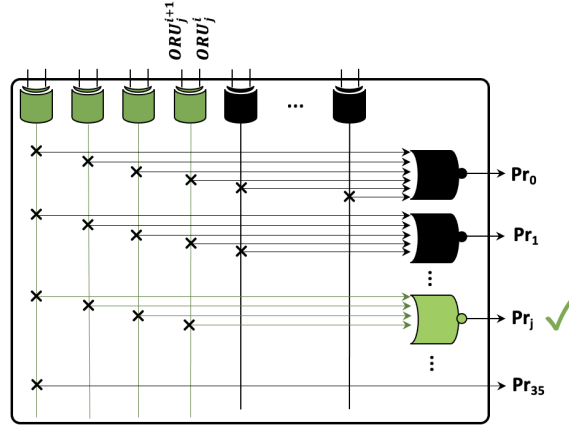


Figure 4.4: The architecture of EFRS for terminating the operation based on the desired precision Pr_0 to Pr_{35} that are obtained from the difference between ORU^i and ORU^{i+1} of two successive iterations i and $i+1$ of TCD-MAC++.

4.2 Results

In this section, we evaluate the Power, Performance, and Area (PPA) gain of using three versions of TCD-MAC++, and compare it with the conventional MACs and also our previously offered TCD-MAC architecture.

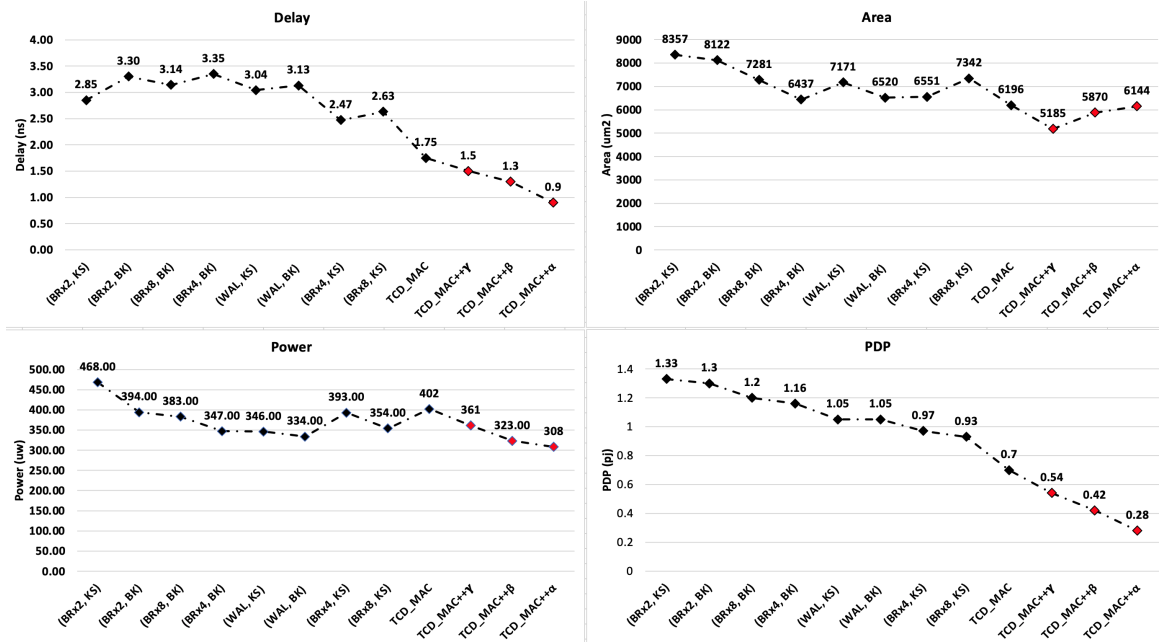


Figure 4.5: PPA comparison between various MACs and TCD-MAC, and different versions of TCD-MAC++.

4.2.1 TCD-MAC++ PPA Assessment

Figure 4.5 captures the PPA comparison of the variation of TCD-MAC++ against a popular set of conventional MAC configurations. As reported, all the versions of TCD-MAC++ have lower latency and consumption energy compare to all reported MACs. More Precisely, TCD-MAC++ $_{\alpha}$ is 48.5% faster than a TCD-MAC and 63.5% faster than the fastest conventional MAC i.e., (BRx4, KS). TCD-MAC++ $_{\alpha}$ consumes 60% less energy than a TCD-MAC and 69.3% less energy than the conventional MAC with the lowest energy i.e., (BRx8, KS). This improvement comes with the limitation that the variations of the TCD-MAC++ take extra cycles (free-run) to generate the correct output when working on a stream of data. However, the power and delay saving of TCD-MAC++ significantly outweigh the delay and power for extra computational cycles.

To illustrate this, the throughput and energy improvement of using a TCD-MAC++ $_{\alpha}$ for processing different sizes of input streams is compared against selected conventional

MACs and is reported in Figure 4.6. In this experiment, we used different input stream sizes ranging from 1 to 1000. The results show a poor performance from both aspects of latency and energy when the input stream size is small. More specifically, when the stream size is less than 10, TCD-MAC++ $_{\alpha}$ performs worse than the TCD-MAC and the other conventional MAC. But when the stream size is more than 10 TCD-MAC++ $_{\alpha}$ outperforms all the variations of TCD-MAC and also conventional MACs. Similarly, TCD-MAC++ $_{\alpha}$ outperforms other MAC units when the stream size is larger than 7. It worth mentioning that TCD-MAC and also TCD-MAC++ have been designed for Neural Networks algorithm and more specifically Deep Neural Networks and these models dealing with input streams of large size.

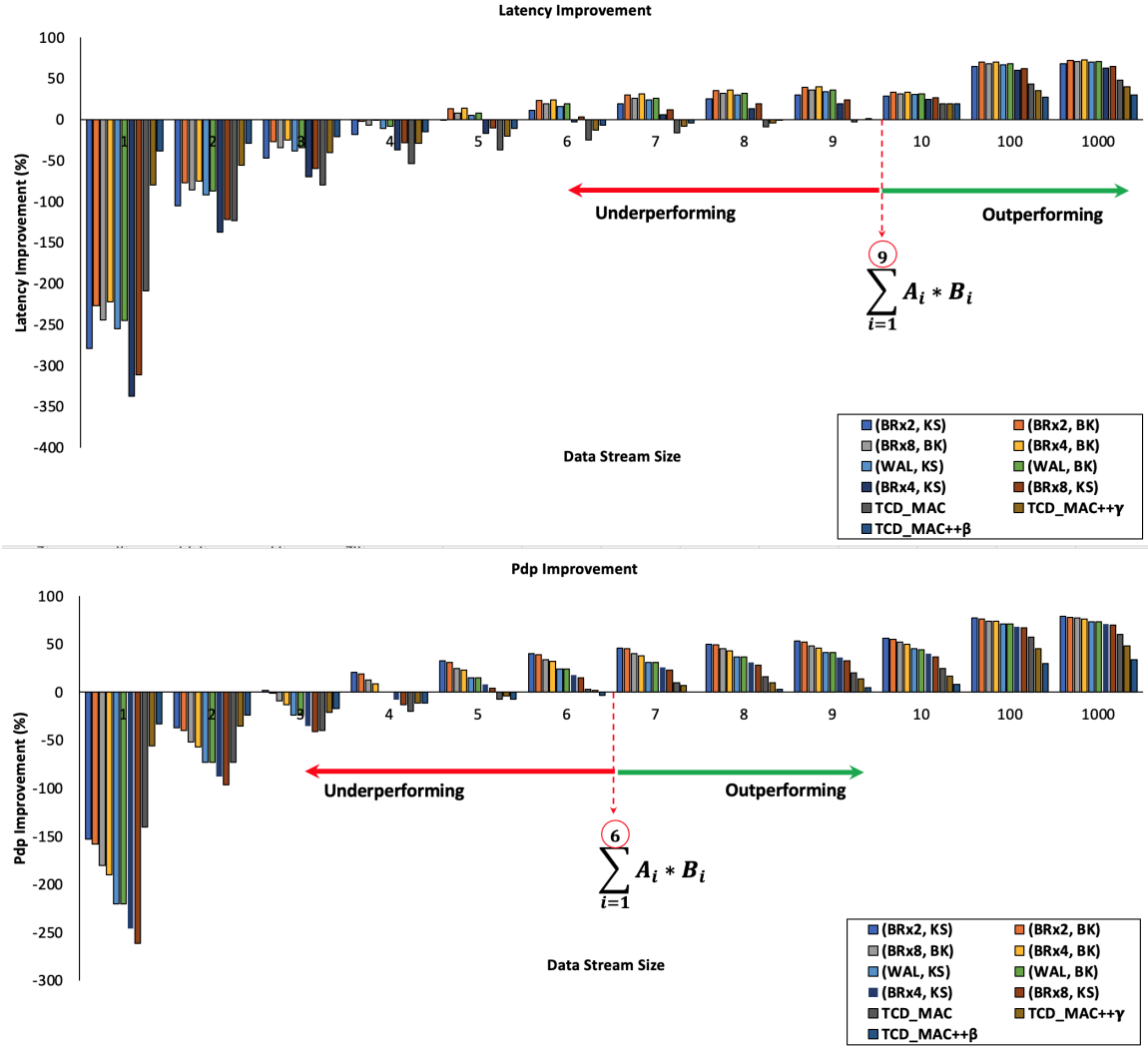


Figure 4.6: Percentage improvement in throughput and energy when using a TCD-MAC++ (as opposed to a conventional MAC) to process an input stream size ranging from 1 to 1000.

4.2.2 EFRS Evaluation

In this section, we show the role of EFRS to decrease the number of extra free runs to generate the accurate results of MAC operation on a stream of operands. The maximum number of EFRS for each of the three versions of the TCD-MAC++ shown in Fig. 4.7. For example, the maximum EFRS for TCD-MAC++ $_{\alpha}$ is 36, which means if TCD-MAC++ operates in the CDM, it needs at most 36 extra runs with zero inputs to generate the final results. However, in most cases, there is no need to wait for that many iterations to have accurate results. Fig. 4.7 shows an experiment of 1k MAC operations of batches of size 1k operands and the histogram of needed EFRS for each design has been shown. This experiment shows that TCD-MAC++ $_{\alpha}$, TCD-MAC++ $_{\beta}$, and TCD-MAC++ $_{\gamma}$ need at most 15, 7, 5 EFRS to generate accurate results, respectively.

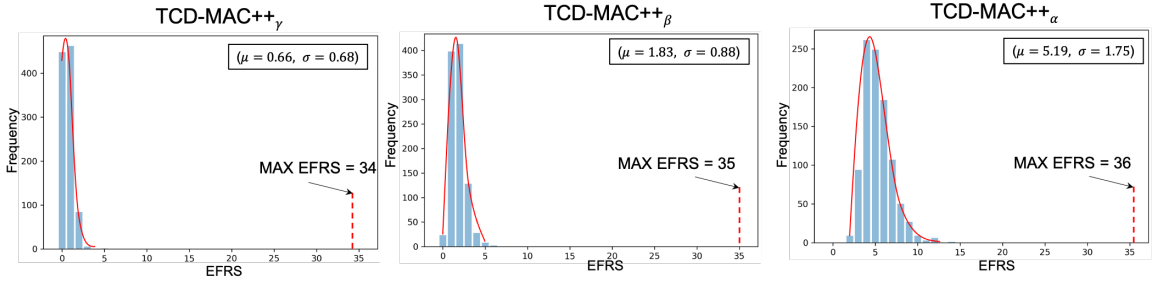


Figure 4.7: The number of needed EFRS of a stream of 1k mac operations of a batch of 1k operands which is performed by three versions of TCD-MAC++. Red dash-line shows the maximum number of needed free run in order to generate the accurate results.

4.3 Conclusion

In this chapter, we introduced an enhanced version of Temporal Carry Deferring MAC (TCD-MAC++) which is well-suited for calculating the dot product of a stream of data. We also introduced the concept of Early Free Running Stop (EFRS), in which, TCD-MAC++ terminates its execution when all of the residual bits in the CBU have been consumed i.e., the accurate result has been generated there is no further input data. We reported that the TCD-MAC++ significantly outperforms similar neural processing solutions that are constructed using conventional MACs in terms of both energy consumption and execution time (performance).

Chapter 5: Future Works

5.1 Introduction

In the previous chapters, we introduced TCD-MAC and different variations of TCD-MAC so-called TCD-MAC $++_{\alpha}$, TCD-MAC $++_{\beta}$, and TCD-MAC $++_{\gamma}$. We also evaluated these variations and showed that they outperform a single MAC when the number of needed computations in a stream of data is large enough. Specifically, we showed TCD-MAC requires one extra free run to generate an accurate result. And also, TCD-MAC $++_{\alpha}$, TCD-MAC $++_{\beta}$, and TCD-MAC $++_{\gamma}$, theoretically require 36, 35, 34 free run at most, respectively.

We showed that for different variations of TCD-MAC $++$, the accurate result is generated with a fewer number of free runs than the maximum free run. In this regard, we introduced Early Free Run Stop (EFRS) that terminates the operation when an accurate result is generated. For different lengths of an input stream, we showed that TCD-MAC $++_{\alpha}$, TCD-MAC $++_{\beta}$, and TCD-MAC $++_{\gamma}$ in practice require 15, 7, 5 EFRS, respectively. Following this chapter, we introduce some of the research directions based on TCD-MAC and TCD-MAC $++$ variations.

5.2 Future Work

TCD-MAC and variations of TCD-MAC $++$ can produce the accurate result of a MAC operation over a stream of data after a different number of free runs from one another. for example, for calculating the dot product of two streams with a size of N , TCD-MAC takes $N+1$ cycle time, T_1 , and TCD-MAC $++_{\alpha}$ requires $N+15$ cycle time, T_2 . And also the energy-per-cycle in the TCD-MAC is higher than TCD-MAC $++_{\alpha}$. Accordingly, when N is

a small number, $N \leq 9$, TCD-MAC can generate the results faster than TCD-MAC++ $_{\alpha}$. As N grows, TCD-MAC++ $_{\alpha}$ can produce the result faster than the TCD-MAC.

Considering structures like Fig. 5.1, we can propose MAC units that support both the TCD-MAC and TCD-MAC++ $_{\alpha}$. Subsequently, based on the size of the input stream of data, either of these MAC structures can be used.

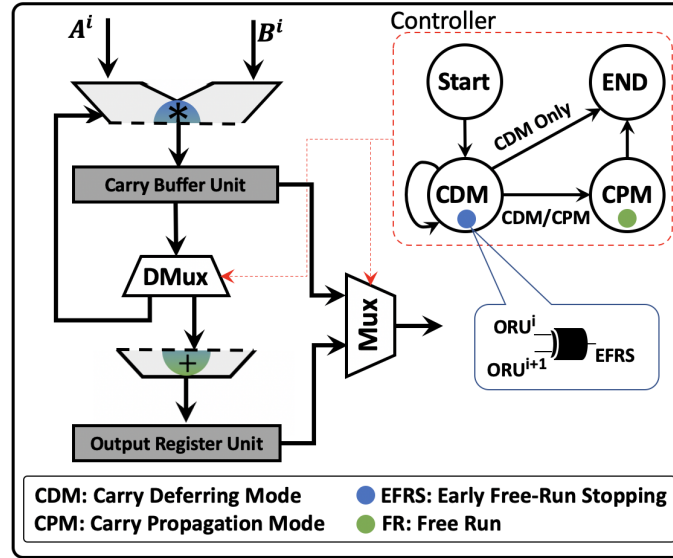


Figure 5.1: An abstract view of the architecture TCD-MAC/TCD-MAC++ for performing MAC operation on a stream of operands A^i and B^i .

In this report, TCD-MAC and TCD-MAC++ are only used for generating accurate results, however, the same structure can also be used for generating approximate results. Which in turn, the delay and energy of computation can be reduced. To generate an approximate result, we can stop the computation before the accurate result is generated. This is simply doable by the architecture we offered in Fig. 4.4. For example, in this figure, the j^{th} bit of precision, pr_j , triggers only based on the bits that their bit significance are higher than j i.e., the j least significant bits are ignored. Subsequently, the EFRS signal

is generated based on only the most significant bits. It should be noted that in neural network accelerators we usually have an array of TCD-MAC/TCD-MAC++. So by adding an approximate-computing feature, each one of mac units can terminate in a different cycle. Accordingly, there is a need to manage when each mac unit starts and finishes. In this report, we only evaluate the accurate version of TCD-MAC/TCD-MAC++ in which all the mac units are synchronized to each other i.e., all the TCD-MAC/TCD-MAC++ start and finish their operation at the same time.

Bibliography

Bibliography

- [1] D. Dua and C. Graff, “UCI machine learning repository,” 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>
- [2] F. Tu and et al., “Rna: A reconfigurable architecture for hardware neural acceleration,” in *Proc.s of the 2015 Design, Automation & Test in Europe Conf. & Exhibition*. EDA Consortium, 2015, pp. 695–700.
- [3] K. Simonyan and et al., “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
- [4] A. Krizhevsky and et al., “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1097–1105.
- [5] M. Abadi, P. Barham, Chen, and et al., “Tensorflow: a system for large-scale machine learning,” in *OSDI*, vol. 16, 2016, pp. 265–283.
- [6] G. Lacey and et al., “Deep learning on fpgas: Past, present, and future,” *arXiv preprint arXiv:1602.04283*, 2016.
- [7] T. Chen and et al., “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.
- [8] Y. Chen, Luo, and et al., “Dadiannao: A machine-learning supercomputer,” in *Proc.of the 47th Annual IEEE/ACM Int. Symp. on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.
- [9] Z. Du and et al., “Shidiannao: Shifting vision processing closer to the sensor,” in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.
- [10] Y.-H. Chen and et al., “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *Proc.of the 43rd Int. Symp. on Computer Architecture*, ser. ISCA ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 367–379. [Online]. Available: <https://doi.org/10.1109/ISCA.2016.40>
- [11] K. Z. Azar, H. M. Kamali, S. Roshanisefat, H. Homayoun, C. P. Sotiriou, and A. Sasan, “Data flow obfuscation: A new paradigm for obfuscating circuits,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 643–656, 2021.
- [12] H. M. Kamali and S. Hessabi, “Adapnoc: A fast and flexible fpga-based noc simulator,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–8.

- [13] H. M. Kamali, K. Z. Azar, and S. Hessabi, "Ducnoc: A high-throughput fpga-based noc simulator using dual-clock lightweight router micro-architecture," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 208–221, 2017.
- [14] Z. Chen, L. Zhang, G. Kolhe, H. M. Kamali, S. Rafatirad, S. M. Pudukotai Dinakarrao, H. Homayoun, C.-T. Lu, and L. Zhao, "Deep graph learning for circuit deobfuscation," *Frontiers in Big Data*, vol. 4, p. 12, 2021.
- [15] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "Nngsat: Neural network guided sat attack on logic locked complex structures," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.
- [16] J. K. H. H. Mirzaeian, Ali and and A. Sasan, "Diverse knowledge distillation (dkd): A solution for improving the robustness of ensemble models against adversarial attacks," in *Thirteenth International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2021.
- [17] F. Behnia et al., "Code-bridged classifier (cbc): A low or negative overhead defense for making a cnn classifier robust against adversarial attacks," in *International Symposium on Quality Electronic Design (ISQED) 2020*, 2020.
- [18] A. Mirzaeian and A. Sasan, Patent 16/944 901.
- [19] A. Mehrabi, A. Manocha, B. C. Lee, and D. J. Sorin, "Prospector: Synthesizing efficient accelerators via statistical learning," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 151–156.
- [20] A. Mehrabi, A. Manocha, B. C. Lee, and D. J. Sorin, "Bayesian optimization for efficient accelerator synthesis," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, Dec. 2021. [Online]. Available: <https://doi.org/10.1145/3427377>
- [21] Z. Rajabi, A. Shehu, and O. Uzuner, "A multi-channel bilstm-cnn model for multilabel emotion classification of informal text," in *2020 IEEE 14th International Conference on Semantic Computing (ICSC)*, 2020, pp. 303–306.
- [22] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
- [23] H. M. Kamali, K. Z. Azar, S. Roshanisehat, A. Vakil, and A. Sasan, "Extru: A lightweight, fast, and secure expirable trust for the internet of things," in *14TH IEEE Dallas Circuits and System Conference (DCAS)*. IEEE, 2020, pp. 1–8.
- [24] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *arXiv preprint arXiv:1511.00363*, 2015.
- [25] S. Han and et al., "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
- [26] A. G. Howard and et al., "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861*, 2017.

- [27] K. Neshatpour, F. Behnia, H. Homayoun, and A. Sasan, "Icnn: An iterative implementation of convolutional neural networks to enable energy and computational complexity aware dynamic approximation," in *Design, Automation & Test in Europe conf. & Exhibition (DATE), 2018*. IEEE, 2018, pp. 551–556.
- [28] K. Neshatpour and et al., "Icnn: The iterative convolutional neural network," in *ACM Transactions on Embedded Computing Systems (TECS)*. ACM, 2019.
- [29] A. Mirzaeian, S. Manoj, A. Vakil, H. Homayoun, and A. Sasan, "Conditional classification: A solution for computational energy reduction," in *Thirteenth International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2021.
- [30] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.
- [31] K. Neshatpour and et al., "Exploiting energy-accuracy trade-off through contextual awareness in multi-stage convolutional neural networks," in *20th International Symposium on Quality Electronic Design (ISQED)*, March 2019, pp. 265–270.
- [32] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5687–5695.
- [33] I. Hubara and et al., "Binarized neural networks," in *Advances in neural information processing systems*, 2016, pp. 4107–4115.
- [34] K. Zamiri Azar, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Threats on logic locking: A decade later," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, 2019, pp. 471–476.
- [35] C. Zhang and et al., "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proc.s of the 2015 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.
- [36] S. Gupta and et al., "Deep learning with limited numerical precision," in *Int. Conf. on Machine Learning*, 2015, pp. 1737–1746.
- [37] M. Peemen, A. A. Setio, and et al., "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st Int. conf. on*. IEEE, 2013, pp. 13–19.
- [38] A. Mirzaeian and et al., "Nesta: Hamming weight compression-based neural proc. engine," *preprint arXiv:1910.00700*, 2020.
- [39] M. Sankaradas, Jakkula, and et al., "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE Int. conf. on*. IEEE, 2009, pp. 53–60.
- [40] S. Chakradhar and et al., "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.

- [41] V. Gokhale and et al., “A 240 g-ops/s mobile coprocessor for deep neural networks,” in *Proc.s of the IEEE Conf. on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.
- [42] H.-D. Block, “The perceptron: A model for brain functioning. i,” *Reviews of Modern Physics*, vol. 34, no. 1, p. 123, 1962.
- [43] (2018) Synopsys design compiler dc. [Online; accessed April 17, 2018]. [Online]. Available: <https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html>
- [44] Synopsys. (2018) Ic compiler icc. [Online; accessed April 17, 2018]. [Online]. Available: <https://www.synopsys.com/implementation-and-signoff/physical-implementation/ic-compiler.html>
- [45] (2018) Synopsys primetime. [Online; accessed April 17, 2018]. [Online]. Available: [synopsys.com/implementation-and-signoff/signoff/primetime.html](https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html)
- [46] A. Sasan and et al., “Variation trained drowsy cache (vtd-cache): A history trained variation aware drowsy cache for fine grain voltage scaling,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 4, pp. 630–642, 2012.
- [47] A. Sasan and et al., “A fault tolerant cache architecture for sub 500mv operation: resizable data composer cache (rdc-cache),” in *Proc. of the 2009 Int. Conf. on Compilers, architecture, and synthesis for embedded systems*. ACM, 2009, pp. 251–260.
- [48] A. Sasan and et al., “Inquisitive defect cache: A means of combating manufacturing induced process variation,” *IEEE Transactions on VLSI Systems*, vol. 19, no. 9, pp. 1597–1609, Sep. 2011.
- [49] A. Sasan and et al., “History amp; variation trained cache (hvt-cache): A process variation aware and fine grain voltage scalable cache with active access history monitoring,” in *Thirteenth Int. Symposium on Quality Electronic Design (ISQED)*, March 2012, pp. 498–505.
- [50] A. Sasan and et al., “Process variation aware sram/cache for aggressive voltage-frequency scaling,” in *2009 Design, Automation Test in Europe Conf. Exhibition*, April 2009, pp. 911–916.
- [51] Y. LeCun and et al., “Deep learning,” *Nature*, vol. 521, no. 7553, 2015.
- [52] C. Szegedy and et al., “Going deeper with convolutions,” in *Proc.of the IEEE conf. on computer vision and pattern recognition*, 2015, pp. 1–9.
- [53] K. He and et al., “Deep residual learning for image recognition,” in *Proc.of the IEEE conf. on computer vision and pattern recognition*, 2016, pp. 770–778.
- [54] H. Mardani Kamali and A. Sasan, “Much-swift: A high-throughput multi-core hw/sw co-design k-means clustering architecture,” in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, 2018, pp. 459–462.

- [55] H. M. Kamali, “Using multi-core hw/sw co-design architecture for accelerating k-means clustering algorithm,” *arXiv preprint arXiv:1807.09250*, 2018.
- [56] R. Girshick and et al., “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proc. of the IEEE conf. on computer vision and pattern recognition*, 2014, pp. 580–587.
- [57] P. Sermanet and et al., “Overfeat: Integrated recognition, localization and detection using convolutional networks,” *arXiv preprint arXiv:1312.6229*, 2013.
- [58] Y. Jia and et al., “Caffe: Convolutional architecture for fast feature embedding,” in *Proc. of the 22nd ACM Int. conf. on Multimedia*. ACM, 2014, pp. 675–678.
- [59] N. Ketkar, “Introduction to pytorch,” in *Deep Learning with Python*. Springer, 2017, pp. 195–208.
- [60] J. Cong and et al., “Minimizing computation in convolutional neural networks,” in *Int. conf. on artificial neural networks*. Springer, 2014, pp. 281–290.
- [61] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, “Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 97–122, 2019.
- [62] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, “Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks,” in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.
- [63] R. Hojabr, K. Givaki, S. Tayaranian, P. Esfahanian, A. Khonsari, D. Rahmati, and M. H. Najafi, “Skippynn: An embedded stochastic-computing accelerator for convolutional neural networks,” in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 132.
- [64] A. Mirzaeian, H. Homayoun, and A. Sasan, “Tcd-npe: A re-configurable and efficient neural processing engine, powered by novel temporal-carry-deferring macs,” *arXiv preprint arXiv:1910.06458*, 2019.
- [65] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, “Accelerating convolutional neural network with fft on embedded hardware,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, pp. 1737–1749, Sep. 2018.
- [66] A. Jafari, M. Hosseini, A. Kulkarni, C. Patel, and T. Mohsenin, “Binmac: Binarized neural network manycore accelerator,” in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, ser. GLSVLSI ’18. New York, NY, USA: ACM, 2018, pp. 443–446. [Online]. Available: <http://doi.acm.org/10.1145/3194554.3194634>
- [67] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, “Interlock: An intercorrelated logic and routing locking,” in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

- [68] S. Roshanisefat, H. M. Kamali, K. Z. Azar, S. M. P. Dinakarrao, N. Karimi, H. Homayoun, and A. Sasan, "Dfssd: Deep faults and shallow state duality, a provably strong obfuscation solution for circuits with restricted access to scan chain," in *2020 IEEE 38th VLSI Test Symposium (VTS)*. IEEE, 2020, pp. 1–6.
- [69] O. Kwon and et al., "A fast hybrid carry-lookahead/carry-select adder design," in *Proc. of the 11th Great Lakes Symp. on VLSI*. ACM, 2001, pp. 149–152.
- [70] R. P. Brent and H.-T. Kung, "A regular layout for parallel adders," *IEEE trans. on Computers*, no. 3, pp. 260–264, 1982.
- [71] P. M. Kogge and et al., "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE trans. on computers*, vol. 100, no. 8, pp. 786–793, 1973.
- [72] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of hardware data prefetchers on server processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, p. 52, 2019.
- [73] M. Bakhshalipour, P. Lotfi-Kamran, A. Mazloumi, F. Samandi, M. Naderan-Tahan, M. Modarressi, and H. Sarbazi-Azad, "Fast data delivery for many-core processors," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1416–1429, 2018.
- [74] S. Chakradhar and et al., "A dynamically configurable coprocessor for convolutional neural networks," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, Jun. 2010. [Online]. Available: <http://doi.acm.org/10.1145/1816038.1815993>
- [75] V. Gokhale and et al., "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proc.s of the 2014 IEEE conf. on Computer Vision and Pattern Recognition Workshops*, ser. CVPRW '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 696–701. [Online]. Available: <http://dx.doi.org/10.1109/CVPRW.2014.106>
- [76] Y. Chen and et al., "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *CoRR*, vol. abs/1807.07928, 2018. [Online]. Available: <http://arxiv.org/abs/1807.07928>

Curriculum Vitae

Ali Mirzaeian received his B.Sc. degree in computer engineering from Isfahan University of Technology (IUT) in Isfahan, Iran in 2013, and his M.Sc. degree in computer engineering from Iran University of Science and Technology (IUST), Tehran, Iran in 2016. He received his Ph.D. at the Electrical and Computer Engineering department at George Mason University. His research focuses on accelerator design for neural networks, neural network security, graph neural network, and reinforcement learning.