NESTA: Hamming Weight Compression-Based Neural Proc. Engine

Ali Mirzaeian
ECE Department
George Mason University
Fairfax, Virginia, 22030
e-mail: amirzaei@gmu.edu

Houman Homayoun
ECE Department
George Mason University
Fairfax, Virginia, 22030
e-mail: hhomanyou@gmu.edu

Avesta Sasan
ECE Department
George Mason University
Fairfax, Virginia, 22030
email: asasan@gmu.edu

Abstract—**In this paper, we present NESTA, a specialized Neural engine that significantly accelerates the computation of convolution layers in a deep convolutional neural network, while reducing the computational energy. NESTA reformats Convolutions into 3×3 batches and uses a hierarchy of Hamming Weight Compressors to process each batch. Besides, when processing the convolution across multiple channels, NESTA, rather than computing the precise result of a convolution per channel, quickly computes an approximation of its partial sum, and a residual value such that if added to the approximate partial sum, generates the accurate output. Then, instead of immediately adding the residual, it uses (consumes) the residual when processing the next batch in the hamming weight compressors with available capacity. This mechanism shortens the critical path by avoiding the need to propagate carry signals during each round of computation and speeds up the convolution of each channel. In the last stage of computation, when the partial sum of the last channel is computed, NESTA terminates by adding the residual bits to the approximate output to generate a correct result.**

Convolutional Neural Network, DNN Accelerator, MAC

I. Introduction and Background

Deep learning models that deploy Convolutional Neural Networks (CNN) for feature extraction have become increasingly popular in recent years [1]. 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 [2], [3], [4], [5], [6], [7], [8], object detection, and object localization[9], [10]. This, as illustrated in Table I, is made possible by using deep and complex neural networks expressed using specialized frameworks such as Caffe [11], PyTorch [12] and Tensorflow [13], 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 [14]. 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 [15].

TABLE I: Depth and complexity of some of the existing and modern CNN solutions for object detection.

	AlexNet[2]	VGG[3]	GoogLeNet[4]	Resnet[5]
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

The generality of a processing engine significantly affects the throughput and energy efficiency of neural processing hardware[16][17]. 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[14], Diannao[18], Dadiannao[19], or Shidiannao[20] are deployed. The major difference in the performance of ASIC accelerator solutions, previously proposed in [21], [22], [14], [18], [19], [20], [16], [17], [23], [24], 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.

In this paper, 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.

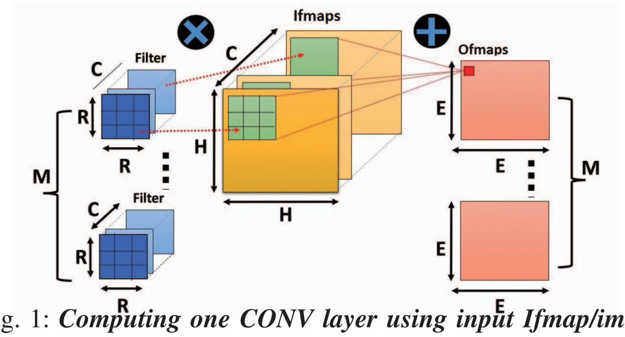


Fig. 1: ***Computing one CONV layer using input Ifmap/image and filters to produce the output (Ofmaps)***

II. NESTA: Proposed Processing Engine

Before describing our proposed solution, we first explain the concept of *temporal carry* in a miniaturized solution in section II-A, then we explain the concept of *compression and expansion*

978-1-7281-4123-7/20/$31.00 © 2020 IEEE

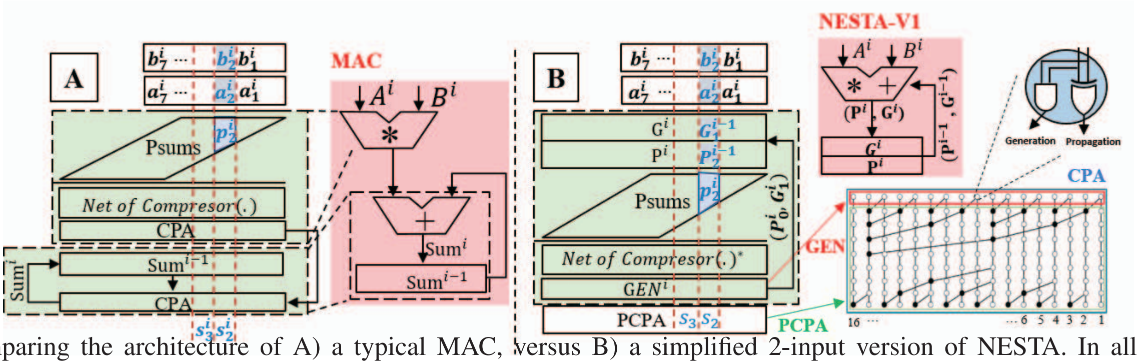


Fig. 2: comparing the architecture of A) a typical MAC, versus B) a simplified 2-input version of NESTA. In all variables in the form of D_m^i, the subscript (m) captures the bit position values, and postscript (i) capture the cycle (iteration). For example, A^i, B^i are the input data in the i^{th} 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.

in section II-B. Finally, in section II-C, we use these concepts to construct and describe our proposed solution.

A. 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. 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, 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. 2(A-right) captures how CPA has been broken into GEN (which is highlighted in red), and PCPA (Partial CPA).

Fig.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.

B. 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.(left), is using an adder-tree, while each add operator could be implemented using a fast adder such as carry-look-ahead [25] (CLA), Brent-Kung [26] (BK) or Kogge-Stone [27] (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 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.

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

Fig. 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.

Table II 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 II. The resulting improvements in the HWC-Adder(s) are the result of 1) using larger HW compressors (as opposed to C(2:2)

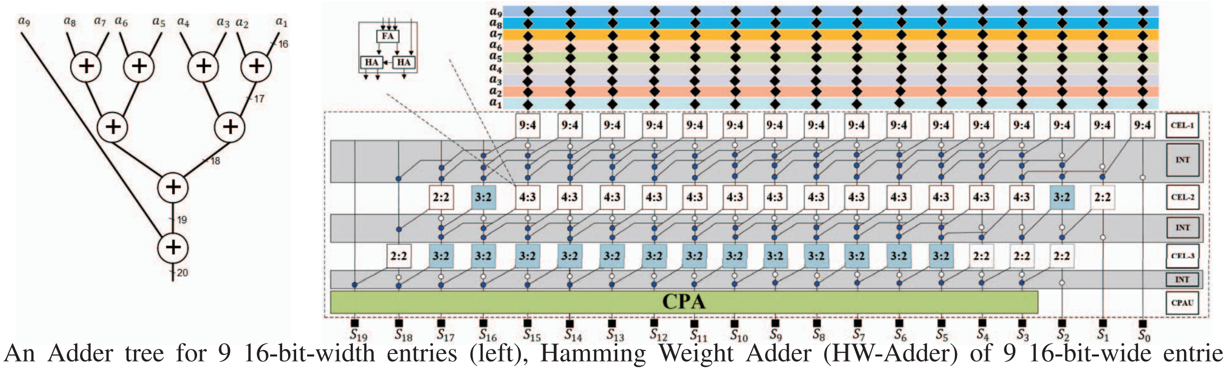


Fig. 3: An Adder tree for 9 16-bit-width entries (left), Hamming Weight Adder (HW-Adder) of 9 16-bit-wide entries (right). In the HWC-Adder compressor hierarchy (CEL) the complete compressors are colored blue, while compressors with available capacity are white. For building the improved version of HWC-Adder (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 II

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

Adder Type	Area(um^2)	Delay(ns)	Power(uW)	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

and C(3:2) in Brent-Kung), and 2) maximizing the number of complete compressors, thus reducing the hardware deficiency.

C. 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 nth round of execution, NESTA performs the following operation:

$$NESTA_{ACC}(n) = NESTA_{ACC}(n-1) + \sum_{i=9n}^{9n+9} I_i \times W_i \quad (2)$$

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 II-A. 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 II-A, 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.

Fig. 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:

1) Data Reshape Unit (DRU): The DRU, as illustrated in Fig. 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.

2) 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 singed 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 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:

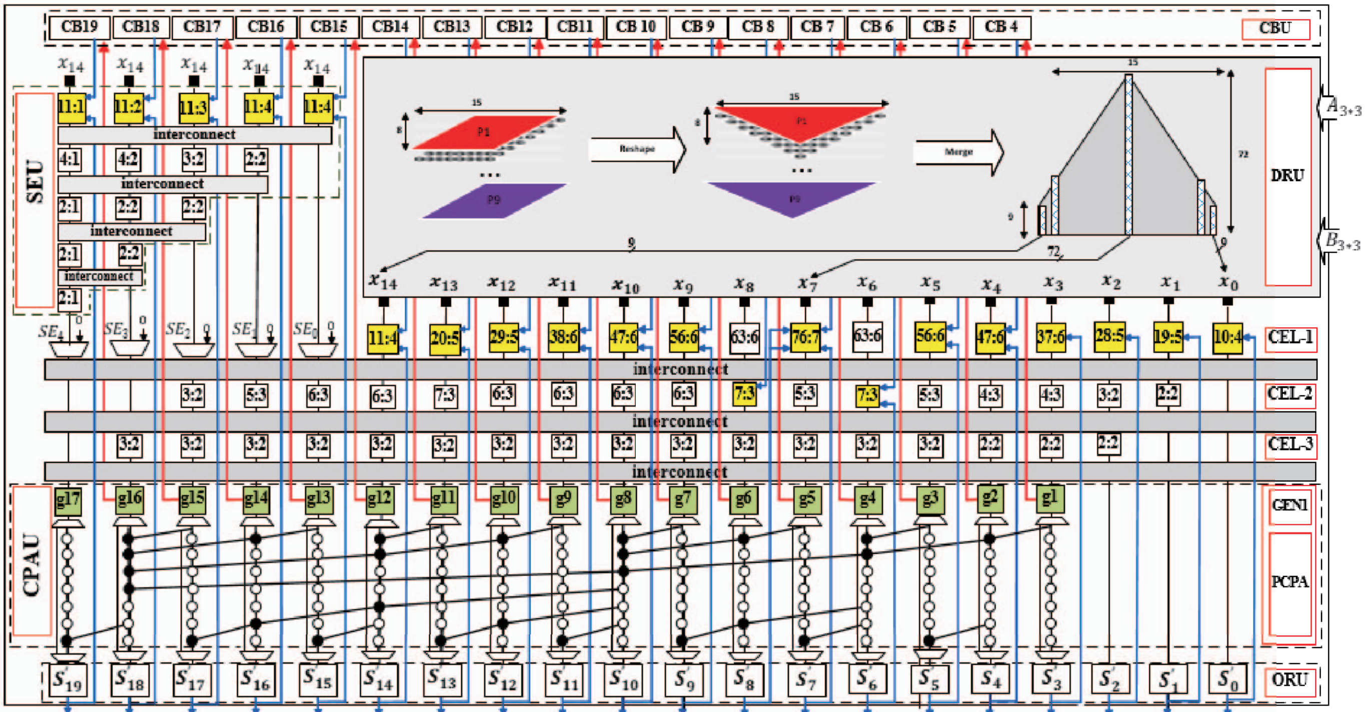

CB19
CB18
CB17
CB16
CB15
CB14
CB13
CB12
CB11
CB 10
CB 9
CB 8
CB 7
CB 6
CB 5
CB 4
CBU
SEU
interconnect
Reshape
Merge
P1
P9
DRU
A_{3*3}
B_{3*3}
x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
CEL-1
CEL-2
CEL-3
CPAU
g17 g16 g15 g14 g13 g12 g11 g10 g9 g8 g7 g6 g5 g4 g3 g2 g1
GEN1
PCPA
ORU
S'_{19} S'_{18} S'_{17} S'_{16} S'_{15} S'_{14} S'_{13} S'_{12} S'_{11} S'_{10} S'_9 S'_8 S'_7 S'_6 S'_5 S'_4 S'_3 S'_2 S'_1 S'_0

Fig. 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.

$$b \times a = (-2^7 + \sum_{i=0}^{6} x_i 2^i) \times a = -2^7 a + (\sum_{i=0}^{6} x_i 2^i) \times a \quad (3)$$

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.

3) Compression and Expansion layers (CEL): The input to ith 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 II-B, 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.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.

4) Carry Propagation Adder Unit(CPAU): Similar to HWC-Adder, described in section II-B, 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.

5) 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.

6) 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.

D. 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. 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.

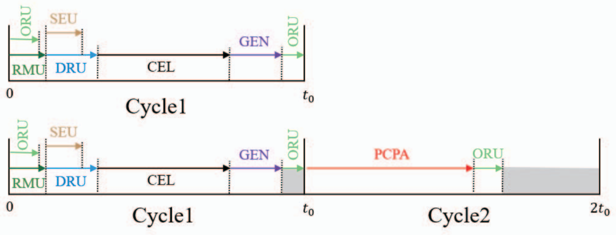


Fig. 5: NESTA cycle time is computed by excluding the execution time of PCPA. In the last cycle of computation of convolution, the NESTA activates the PCPA and captured the correct sum after 2 cycles of execution.

E. 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 [28], [29], [30], [31], [32], 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. 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 [14][33][34]. 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 [14] 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 [18]. 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 [35][36]. IS and OS dataflows try to minimize the movement of Ifmaps and Psums respectively, examples of which include [17][37][20]. The RS dataflow combines the WS and the OS dataflows to achieve greater efficiency. Eyeriss is an example of RS implementation described in [14][38].

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. 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. 1, 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. 1, 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 1 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. 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. 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

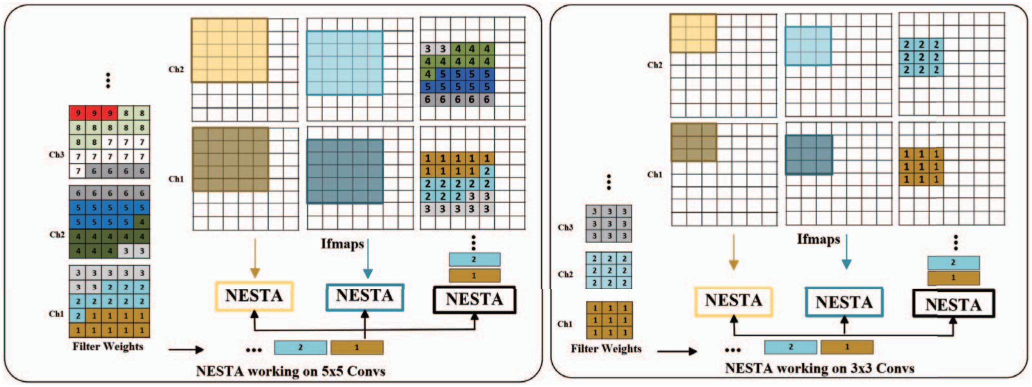


Fig. 6: NESTA Row Stationary (RS) data flow for executing 3×3 convolution across multiple channels (right) and 5×5 convolution across multiple channels (left). A similar concept can be used to support all other convolutions sizes.

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. 6.

As described in section II-D, 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.

III. 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.

A. 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 [39] 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 [40]. The area and delay metrics are reported using Synopsys Primetime [41]. The reported power is then averaged across 20K cycles of simulation with random input data fed to PrimetimePX [41] 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. 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 II-B. 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.

B. 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.

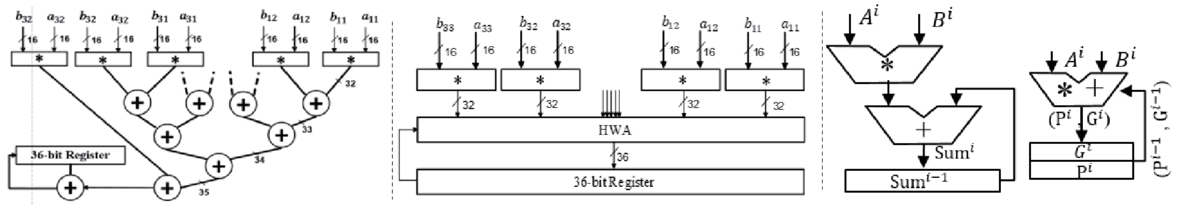
Performance: In terms of delay, the delay of NESTA is better than all other MAC9 flavors. For example, the delay of NESTA is 23.7% and 11.3% better than (BRX2, (Tree, BK)) and (BRX4, (HWA, KS)) as the slowest and fastest MAC9s in Fig. 11. However, the reduction in the delay comes with a catch; When NESTA 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.8. The larger the number of input channels, the smaller the overhead of one extra cycle for the final addition. As illustrated in Fig. 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.

Area: Figure 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.

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. 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.

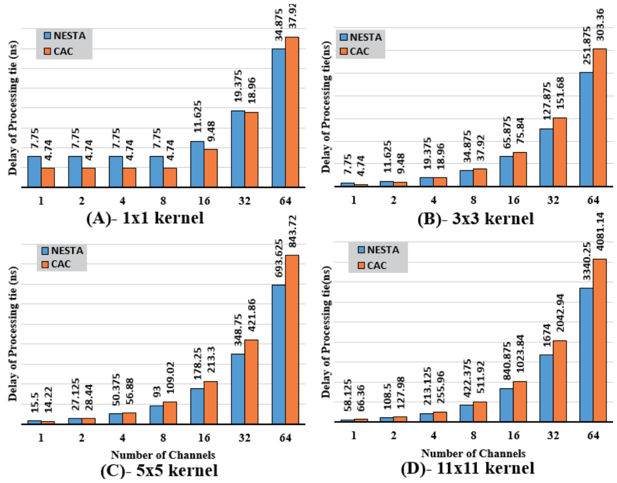
C. PPA efficiency: NESTA v.s. MACs

Table III 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



b_{32} a_{32} b_{32} a_{32} b_{31} a_{31} b_{12} a_{12} b_{11} a_{11}
16
*
+
32
33
34
35
36-bit Register
b_{33} a_{33} b_{32} a_{32} b_{12} a_{12} b_{11} a_{11}
HWA
36
36-bitRegister
A^i B^i
Sum^i
Sum^{i-1}
$(P^i$ $G^i)$
G^i
P^i
(P^{i-1}, G^{i-1})

Fig. 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.



(A)- 1x1 kernel

(B)- 3x3 kernel

(C)- 5x5 kernel

(D)- 11x11 kernel

Fig. 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.

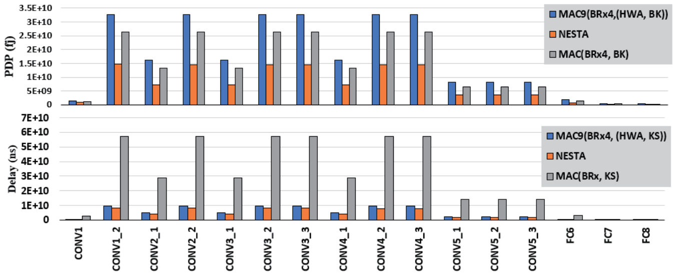


Fig. 9: Breakdown of delay and energy consumption of each layer of VGG [3] when processed by a single MAC, a single MAC9 or a single NESTA core. A linear increase in the number of cores linearly reduces the processing time.

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 IV 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.

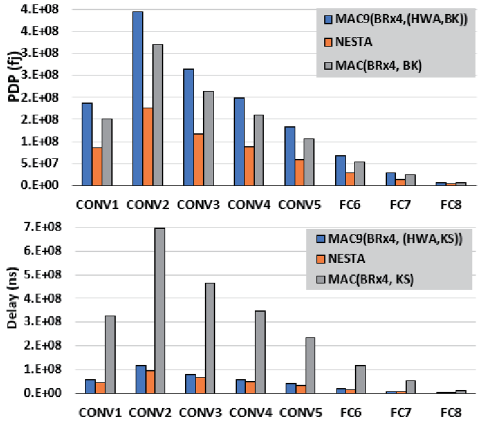


Fig. 10: Breakdown of delay and energy consumption of each layer of AlexNet [2] when processed by a Neural engine composed of MACs, MAC9s or NESTA cores.

TABLE III: 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 IV: 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

D. 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[2] and VGG[3]. In this paper, we only investigate the energy consumed for the processing the information and would address the saving

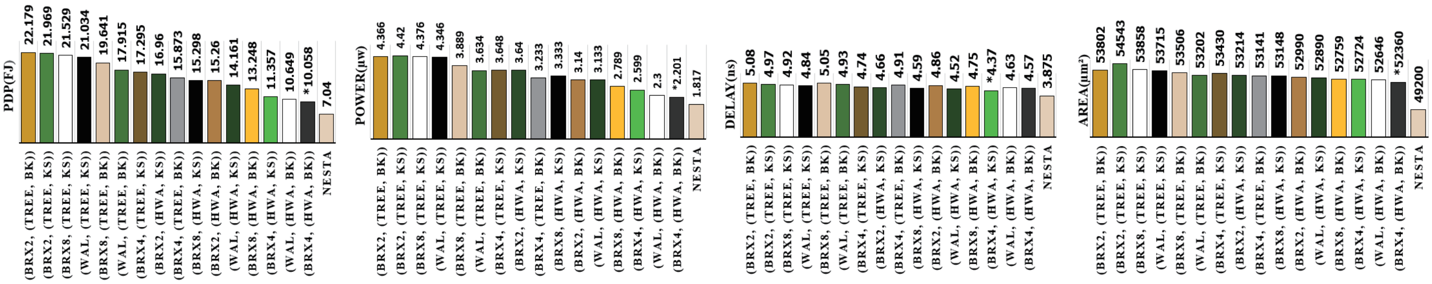


Fig. 11: Area, Delay, Power, and PDP comparison between NESTA and MAC9s constructed using fast adders and multipliers. The star identifies the best MAC9 in each category.

(due to data reuse in NESTA) dataflow related power saving in the future work. Fig. 9 and Fig. 10 capture the delay and energy consumed for processing each layer (including CONVs and FCs layers) of Alexnet [2] 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 III-C and III-B. 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.

IV. Acknowledgement

This work was supported by the National Science Foundation (NSF) through Computer Systems Research (CSR) program under NSF award number 1718538.

V. Conclusion

In this paper, 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.

References

[1] G. H. Yann LeCun, Yoshua Bengio, "Deep learning," *Nature*, vol. 521, no. 7553, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1097–1105.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] 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.

[5] 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.

[6] 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.

[7] K. Neshatpour, H. Homayoun, and A. Sasan, "Icnn: The iterative convolutional neural network," in *ACM Transactions on Embedded Computing Systems (TECS)*. ACM, 2019.

[8] K. Neshatpour, F. Behnia, H. Homayoun, and A. Sasan, "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.

[9] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "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.

[10] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[11] Y. Jia and et al., "Caffe: Convolutional architecture for fast feature embedding," in *Proc.of the 22nd ACM international conf. on Multimedia*. ACM, 2014, pp. 675–678.

[12] N. Ketkar, "Introduction to pytorch," in *Deep Learning with Python*. Springer, 2017, pp. 195–208.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[14] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. of the 43rd International Symposium 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

[15] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conf. on artificial neural networks*. Springer, 2014, pp. 281–290.

[16] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International conf. on*. IEEE, 2009, pp. 53–60.

[17] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[18] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[19] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proc.of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[20] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International conf. on*. IEEE, 2013, pp. 13–19.

[21] 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.

[22] 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.

[23] 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.

[24] 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

[25] O. Kwon, E. E. Swartzlander Jr, and K. Nowka, "A fast hybrid carry-lookahead/carry-select adder design," in *Proc.of the 11th Great Lakes symposium on VLSI*. ACM, 2001, pp. 149–152.

[26] R. P. Brent and H.-T. Kung, "A regular layout for parallel adders," *IEEE transactions on Computers*, no. 3, pp. 260–264, 1982.

[27] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE transactions on computers*, vol. 100, no. 8, pp. 786–793, 1973.

[28] A. Sasan, K. Amiri, H. Homayoun, A. M. Eltawil, and F. J. Kurdahi, "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.

[29] A. Sasan, H. Homayoun, A. Eltawil, and F. Kurdahi, "A fault tolerant cache architecture for sub 500mv operation: resizable data composer cache (rdc-cache)," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2009, pp. 251–260.

[30] A. Sasan, H. Homayoun, A. M. Eltawil, and F. Kurdahi, "Inquisitive defect cache: A means of combating manufacturing induced process variation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 9, pp. 1597–1609, Sep. 2011.

[31] A. Sasan, H. Homayoun, K. Amiri, A. Eltawil, and F. Kudahi, "History amp; variation trained cache (hvt-cache): A process variation aware and fine grain voltage scalable cache with active access history monitoring," in *Thirteenth International Symposium on Quality Electronic Design (ISQED)*, March 2012, pp. 498–505.

[32] A. Sasan, H. Homayoun, A. Eltawil, and F. Kurdahi, "Process variation aware sram/cache for aggressive voltage-frequency scaling," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 911–916.

[33] 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.

[34] 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.

[35] 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

[36] V. Gokhale and et al., "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings 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

[37] S. Gupta, A. Agrawal, , and P. Narayanan, "Deep learning with limited numerical precision," in *ICML*, 2015, pp. 1737–1746.

[38] Y. Chen, J. S. Emer, 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

[39] (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

[40] Synopsys. (2018) Ic compiler icc. [Online; accessed April 17, 2018]. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/physical-implementation/ic-compiler.html

[41] (2018) Synopsys primetime. [Online; accessed April 17, 2018]. [Online]. Available: synopsys.com/implementation-and-signoff/signoff/primetime.html