Revisiting Deep Learning Parallelism: Fine-Grained Inference Engine Utilizing Online Arithmetic

Ameer M.S. Abdelhadi
Department of Electrical and Electronic Engineering
Imperial College London
London SW7 2AZ, United Kingdom
a.abdelhadi@imperial.ac.uk

Lesley Shannon
School of Engineering Science
Simon Fraser University
Burnaby BC V5A 1S6, Canada
lshannon@ensc.sfu.ca

***Abstract*—Modern Deep Neural Networks (DNNs) exhibit incredible performance on a variety of complex tasks, such as recognition, classification, and natural language processing. Adapting to ever-increasing workloads, deep learning algorithms have become extremely compute- and memory-intensive, making them infeasible for deployment on compact, embedded platforms with power and cost budget limitation. Common methods to minimize and accelerate deep learning involve pruning, quantization and compression of the neural model. While these techniques show a dramatic model reduction, in several cases they incur an accuracy degradation. Moreover, methods involving custom hardware still suffer from large silicon footprint and high power consumption due to massive computations and external memory accesses. In this paper, we revisit the parallelism of neural inference engines. In a departure from the conventional coarse-grained neuron-level parallelism, we propose a synapse-level parallelism by performing highly parallel fine-grained neural computations. Our method employs online Most Significant Digit (MSD) first digit-serial arithmetic to enable early termination of the computation. Using online MSDF bit-serial arithmetic for DNN inference (1) enables early termination of ineffectual computations, (2) enables mixed-precision operations (3) allows higher frequencies without compromising latency, and (4) alleviates the infamous weights memory bottleneck. The proposed technique is efficiently implemented on FPGAs due to their concurrent fine-grained nature, and the availability of on-chip distributed SRAM blocks. compared to other bit-serial methods, our Fine-Grained Inference Engine (FGIE) improves energy efficiency by $\times$1.8 while having similar performance gains.**

***Index Terms*—machine learning, deep learning, deep neural networks (DNNs), deep neural networks inference engine, deep neural networks acceleration, fixed-point deep neural networks**

I. Introduction

Since the beginning of the current century, great achievements in machine learning have been possible due to significant advancements in big data processing. In particular, deep learning—bio-inspired artificial neural networks—has become the *de facto* standard for a variety of machine learning applications including, but not limited to, image classification [1], [2], image detection [3], [4], video classification [5], [6], speech recognition [7], [8], speech synthesis [9], [10], and language modelling [11], [12].

This research has been funded by the National Sciences and Engineering Research Council of Canada (NSERC) Chair for Women in Science and Engineering Grant (British Columbia and Yukon) PDF Funding. This research has also been funded by the Computing Hardware for Emerging Intelligent Sensory Applications (COHESA) project. COHESA is financed under the NSERC Strategic Networks grant number NETGP485577-15.

As depicted in Figure 1, Deep Neural Networks (DNNs) consist of several layers of processing neurons where data is transferred between layers via weighted interconnects. This structure enables learning a higher-level abstracted feature representation of a complex multi-dimensional data when trained with large datasets [13]. However, modern applications compel an ever-increasing workloads, which adversely requires more complex neural networks with enormous parameters to better learn high-level features of the trained dataset and improve prediction accuracy [14]. Moreover, the majority of deep learning applications require real-time, low-latency processing. Due to their complexity, these applications require massive computing, hence they are rarely found in consumers' devices.

State-of-the-art DNNs consist of tens to hundreds of computation layers connected with hundreds of millions of weighted synapsis. For instance, AlexNet—an image classification networks based on Convolutional Neural Networks (CNNs)—requires 60 million parameters [1], Deep Residual Networks—an image classifier from Microsoft Research— requires up to 200 million parameters [15], in addition to other attempts to train 1 billion parameters on a cluster of High-Performance Computing (HPC) machines [14], [16], [17].

The extreme depth of modern DNN models incurs an increasing demand on compute- and memory- intensive processing platforms, accommodating the extensive amount of parameters, and performing the training or inference of the model within a reasonable time. Using general-purpose computing systems to train or merely infer a DNN model is highly inefficient since the model is stored in an external memory. The complete model—possibly containing hundreds of millions of parameters—must be entirely modified in case of training, or entirely fetched (for each input data instance) in case of inference, then processed massively on area- and power-hungry multiply-accumulate (MAC) units.

Training of DNN models is still the bottleneck of computation due to dramatically increasing amounts of training passes. Training can span over several days with modern massive computing systems [14], [16], [17]. While training of the model is required only once before deploying the model into the target platform, the inference of the model is required for every input data instance. The majority of applications require real-time and low-latency processing, thus effectively inferencing the model is the actual bottleneck.

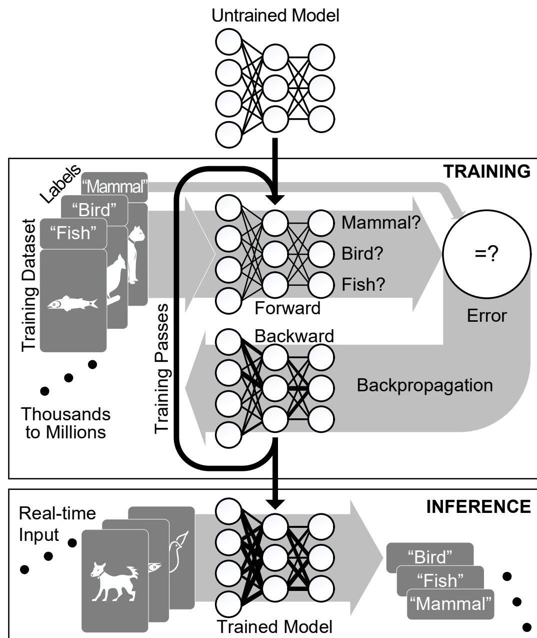


Fig. 1: Deep learning abstraction.

Recently, Graphics Processing Units (GPUs)—highly-parallel processing arrays—are exploited to accelerate DNN computation and satisfy the real-time low-latency requirement [18]. While these highly-parallel arrays successfully accelerate the training and the inference of DNN models [14], their integration into embedded, mobile, and robotic systems—in which the majority of machine learning algorithms are applicable—is power and cost-prohibitive. Alternatively, Special-purpose hardware devices are leveraged by the leading technology vendors (*e.g.*, Google's Tensor Processing Unit (TPU) [19]) to accelerate training and inference while limiting power dissipation. Typically, these devices are designed to handle the majority of deep learning applications, which adversely reduces their efficiency. Furthermore, the fabrication of special-purpose devices requires more engineering effort and incurs longer time-to-market and higher costs.

On the other hand, reconfigurable devices—such as Field-Programmable Gate Array (FPGAs)— exhibit a balance of cost, flexibility, performance, and power consumption. Compared to GPUs, reconfigurable devices offer higher operations/Watts, more efficiently implement irregular structures (*e.g.*, pruned networks), and fine-grained computations (*e.g.*, quantized networks) [20]. As oppose to custom-design special-purpose hardware, reconfigurable devices can be recompiled with different deep learning architectures and optimizations, hence they offer a balance between the flexibility of GPUs and the efficiency of custom-hardware.

In this paper, we develop an efficient non-conventional deep learning inference engine, especially applicable for FPGA devices. Whereas traditional inference methods evaluate a single synapse of several neurons in parallel coarse-grained computations, the proposed approach evaluates several synapses each within several neurons in parallel fine-grained computation. We call the former technique *coarse-grained bit-parallel neuron-level parallelism*, while we call the latter *fine-grained bit-serial synapse-level parallelism.*

TABLE I: List of Notations and Abbreviations

l	Number of layers
n^i	Number of neurons in layer i
a^i	Input activation vector of the i-th layer
a^i_j	The j-th element of the previous vector
b^i	Bias vector of the i-th layer
W^i	Weights matrix of the i-th layer
$w^i_{j,k}$	The weight of the k-th synapse in the j-th neuron in the i-th layer
$x\langle i\rangle$	Single bit selection: the i-th bit of a binary vector b
s^i	Synapse parallelism. The number of synapses that can be processed in parallel by a single FGIE tile in the i-th layer
p^i	Neuron parallelism. The number of neurons that can be processed in parallel by FGIE tiles in the i-th layer.
q^i	Width of the fixed-point representation (in bits) of the i-th layer.
B^i	The number of BRAMs required for the i-th layer.

In a departure for conventional bit-serial inference methods, our Fine-Grained Inference Engine (FGIE), exploits online Most Significant Digit First (MDSF) arithmetic. To the best of our knowledge, this is the first attempt to utilize online arithmetic for neural computation. Thanks to online arithmetic, FGIE achieves $\times 1.8$ improvement in energy efficiency compared to the best of other bit-serial approaches. These efficiency gains were possible since online MSDF arithmetic process most significant data first and allows to terminate the computation once the processed data is sufficient to determine the computation outcome. FGIE also supports layer-wise mixed-precision operations, which reduces the number of computations without reducing the model accuracy. FGIE also allows high frequencies without compromising latency, due to the fine-grained operations. Also, the memory bandwidth is dramatically reduced since weights are read serially.

Notation and abbreviations used for the rest of the paper are listed in Table I. The paper is organized as follows. Section II reviews deep neural networks, optimization methods, bit-serial inference, and online arithmetic. Section III describes our fine-grained synapse-level-parallel inference approach. In Section IV our experimental framework is presented, results are discussed, and other state-of-the-art approaches are compared. Finally, Section V concludes the paper with future suggestions.

II. Background and Preliminaries

This section reviews the basics of DNN inference and online arithmetic. A formalization of multilayer perceptrons is given in Subsection II-A. Optimizing deep learning models is discussed in Subsection II-B Bit-serial inference is reviewed in Subsection II-C. Finally, preliminaries of online arithmetic and redundant number system are provided in Subsection II-D.

A. Deep Neural Networks

Figure 2 shows a multilayer perceptron (MLP). MLP is a feedforward fully-connected (FC) artificial neural network. Computation of a single layer is described as the following matrix-vector multiplication (MxV)

$$a^{i+1} = \sigma\left(W^i a^i + b^i\right), \tag{1}$$

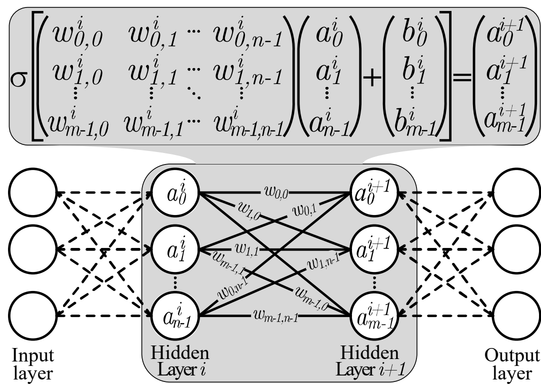


Fig. 2: A single layer of fully-connected network shown as MxV operation.

and element-wise for index k as

$$a_k^{i+1} = \sigma\left(\sum_{j=0}^{n_i-1} w_{k,j}^i a_j^i + b_j^i\right), \tag{2}$$

where a^i is the input activation vector of layer i, a^{i+1} is the output activation vector of layer i, which is also fed as the input activation of the next layer $i+1$, b^i is the bias vector of layer i, W^i is the weights matrix of layer i, and σ is a non-linear activation function. The Rectified Linear Unit (ReLU) is typically used for more efficient computation and better training via backpropagation [21],

$$\text{ReLU}(x) \coloneqq x^+ = \max(0, x). \tag{3}$$

Fully-connected layers typically incur massive computation and storage requirements. The growth of the weights matrix is quadratic to the activation vector size. For instance, AlexNet consists of two fully-connected layers, FC6 and FC7, each with activation vectors of 4k elements [1]. Their associated weights matrix is therefore 4k×4k=16M weights. A single-precision floating-point representation of the weights thus requires a storage of 512Mb.

B. Optimizing Deep Learning Models

Although oversized and over-parametrized DNN models yield higher accuracy while training large datasets, compact models can be efficiently used for inference without substantial degradation of the model accuracy.

Since the majority of power in DNNs is dissipated on memory accesses [22], reducing the number of parameters by pruning the network will result in a reduced model with less storage requirement, accelerated computation, and less power consumption [23]. Quantization is another approach used to compress the network by reducing the accuracy of the parameters [23]–[25]. In extreme cases, binary or ternary synaptic weights are used [26].

While these methods successfully reduce the complexity of DNN models, it's challenging to map these irregular structures to GPUs, the current mainstream DNN platforms [20]. Special-purpose hardware has been developed to implement the aforementioned compression schemes [22], [23], [27]–[29]. However, this special-purpose hardware is tailored to a specific compression scheme and is only suitable for limited DNN models. Furthermore, these accelerators still suffer from common drawbacks of special-purpose hardware, namely, high fabrication cost, engineering effort, and long time-to-market.

TABLE II: Example of Inference Architectures and Their Classification.

Architecture	Granularity	Activations	Weights	Arithmetic	Platform
DaDianNao [30]	coarse	parallel	parallel	parallel	ASIC
NeuFlow [31]	coarse	parallel	parallel	parallel	ASIC\FPGA
Stripes [34]	mixed	serial	parallel	LSB first	ASIC
Pragmatic [35]	mixed	serial	parallel	LSB first	ASIC
Loom [36]	fine	serial	serial	LSB first	ASIC
BISMO [37]	fine	serial	serial	LSB first	FPGA
Moss *et al.* [38]	fine	mixed	mixed	LSB first	FPGA
FGIE (proposed)	fine	serial	serial	MSB first	FPGA

C. Bit-serial Inference Engines

As shown in Table II, DNN inference engines can be categorized into three classes based on the granularity of the computation: coarse-grained bit-parallel, mixed-granularity, and fine-grained bit-serial. Bit-parallel ASIC-based DNN inference engines, such as Google's Tensor Processing Unit [19], DaDianNao [30], and NeuFlow [31] are designed to support a specific number format with constant precision across all network layers. However, parallel processing of may introduce a redundancy in the computation model for two reasons. First, parallel arithmetic compels specifying the number format at design-time, whereas the optimal number precision may be different for each network layer [32]. Second, The number values may have ineffectual computations, such as multiplying by zero. Cnvlutin [33] for instance, is an attempt to eliminate ineffectual computation and enhance the computation model. Conversely, bit-serial inference engines inherently support dynamic precision computation, allowing a run-time dynamic mixed-precision configuration. Stripes [34] and Pragmatic [35] adopt a mixed-granularity architecture. While the activations are received in bit-serial manner, the weights are read as a bit-parallel data. Unlike Stripes, Pragmatic allows skipping ineffectual zero bits of the weights. Loom [36], on the other hand, processes both weights and activations in a bit-serial manner. BISMO [37] and the work of Moss *et al.*provide a multiply–accumulate (MAC) for reconfigurable devices. While BISMO is bit-serial, the work of Moss *et al.* dynamically switch granularity and precision in run-time.

D. Online Arithmetic and Redundant Number System

Online arithmetic is an unconventional approach widely used in digital signal processing [39]. Online arithmetic algorithms perform a digit-serial Most Significant Digit First (MSDF) computation. The result is generating serially, Most Significant Digit (MSD) first, while consuming the inputs, also starting from the most significant digits. Generating the output's MSD based on partial knowledge of the inputs is achieved by employing a Redundant Number System (RNS).

Given an N-digit radix r number in the range $(-1, 1)$, $X = \sum_{i=1}^{N} x_i r^{-i}$, a conventional number representation allows only r possible values for each digit, namely, $x_i \in \{0, 1, \ldots, r-1\}$. This is the smallest digit set to represent radix r where each number will have a unique representation, thus

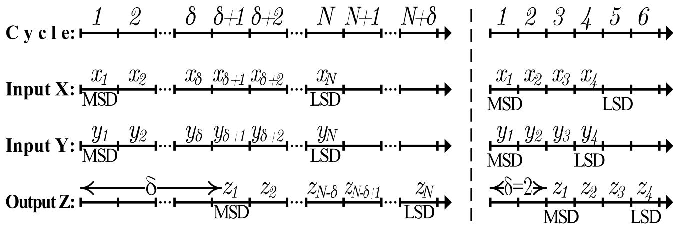


Fig. 3: (left) Timing of the online digit-serial MSDF arithmetic. Outputs are generated starting from the MSD while consuming inputs starting from the MSD. (right) An example with $N = 4$ and $\delta = 2$

this representation is non-redundant. Conversely, a redundant number system allows more than r possible values in the digit set, creating several representations for a single value. The redundancy allows fixing the result of previously computed digits by revising the subsequent digits. For instance, while the conventional (non-redundant) addition operation propagates carry from LSD to MSD, using a redundant number system can absorb the carry and eliminate the curry propagation.

Major redundant representation systems are signed-digit (SD) [40] and carry-save (CS) [41]. While in a conventional non-redundant number system of radix r the digit set $\{0, 1, \cdots, r-1\}$ is used, the SD representation represents each digit with the symmetric redundant set $\{-a, \ldots, 0, \ldots, a\}$, where $a \in \{\lceil r/2 \rceil, \ldots r-1\}$.

As described in Figure 3, outputs and inputs are processed both in MSDF manner. Generating the MSD digit of the output requires receiving δ MSD digits of the inputs. In general, generating an output digit of significance i, where the MSD is of significance 1, requires seeing $i + \delta$ digits of the inputs. $\delta \in \mathbb{N}^+$ is the *online delay*; a precision-independent constant. Assuming N-digits radix r fixed-point representation in the range (-1,1), the inputs and outputs in Figure 3 at cycle $c \in \{1, \ldots, N + \delta\}$ are

$$X[c] = \sum_{i=1}^{c} x_i r^{-i};\ Y[c] = \sum_{i=1}^{c} y_i r^{-i};\ Z[c] = \sum_{i=1}^{c-\delta} z_i r^{-i}. \quad (4)$$

III. FGIE: Fine-Grained Inference Engine via Synapse-Level Parallelism

We now present the proposed architecture. Subsection III-A motivates and explains the key idea for this work. A detailed classification of bit-serial inference is provided in Subsection III-B. The functionality and design consideration of our method is described in Subsection III-C.

A. Motivation and Key Idea

Modern DNN architectures are compute- and memory-intensive, however, these models are typically over-parameterized, which adversely incurs massive redundancy. As described earlier in Subsection II-B, this redundancy can be alleviated by reducing the number of synapses via pruning [23] or by reducing the model precision via quantization [23]–[26], however, both of these techniques reduce the model accuracy and yield an irregular computation structure. Compression schemes are applied to reduce storage [22], [23], [27]–[29], nevertheless, the overhead of decompression is cost-prohibitive. DNNs with bit-serial arithmetic [34]–[36], [38], [42] is another way to reduce computation redundancy. These methods break parallel operations (*e.g.*, multiply-add) into fine-grained bit-serial operations, which enables the elimination of neutral computations.

TABLE III: Percentage of Silent[a] Neurons and the Average Number of Bits Required to Determine if a Neuron is Silent.

Network	Precision bit/layer	Size neuron/layer	Silent neurons	Determine if silent [bit/layer]
AlexNet [1]	10-9-9	4096×4096×1000	52.5%	4.67-4.34-4.33
GoogLeNet [45]	7	1024	58.7%	3.39
VGGS [46]	10-9-9	4096×4096×1000	56.4%	4.62-4.27-4.25
VGGM [46]	10-8-8	4096×4096×1000	57.0%	4.61-3.83-3.85
VGG19 [46]	10-9-9	4096×4096×1000	56.8%	4.61-4.27-4.26

[a] Silent: Not firing. Not passing activation threshold.

On the other hand, this work is inspired by the low rate of firing neurons in biological systems, namely these systems are sparsely active where the majority of neurons are "silent" [43]. An attempt to apply this biological finding on artificial DNNs also reveals that the majority of neurons are not firing, namely, not passing the activation threshold where a neuron transfers its output to the next level. For instance, Table III reveals that 56.3% of the fully-connected layer neurons are silent when deploying the ImageNet dataset [44] on several CNN models. These silent neurons do not transfer their output to the next level. If these silent neurons can be early detected, the computation can be stopped, and consequently less power will be consumed.

Table III shows that an average of 4.6 MSBs (from each 10-bit synapse) are sufficient to detect if a neuron is firing. If ReLU is used as an activation function, firing neurons will have positive values. To perform early detection of silent neurons, MSBs of the neural outputs should be generated and processed first. Thus, an MSDF online arithmetic is required for neural computation. Benefits of using online logic arithmetic are four-fold. First, the computation can be stopped as early as we detect that the neuron is not passing the activation threshold, thus more computations per Watts can be performed. Second, online fine-grained operations allow higher frequencies without compromising latency. Third, bit-serial operations allow layer-wise mixed-precision operations. Finally, bit-serial communication demands less bandwidth of the synaptic weight memory. While other bit-serial methods perform traditional LSB first arithmetic [34]–[36], [38], [42], to the best of our knowledge, our technique is the first to utilize MSDF online arithmetic for DNN inference.

B. A classification of of DNN inference parallelism

A classification of DNN inference parallelism is given in Figure 4. This figure is a high-level overview of three inference architectures, based on the computation granularity. For each architecture, we also describe the parallelism of the MxV operation—the core computation of DNN inference. As depicted in Figure 4 (a), a traditional bit-parallel inference engine evaluate a single synapse of several neurons in parallel coarse-grained computations. The MxV parallelism map shows that each activation input is processed for all neurons in parallel.

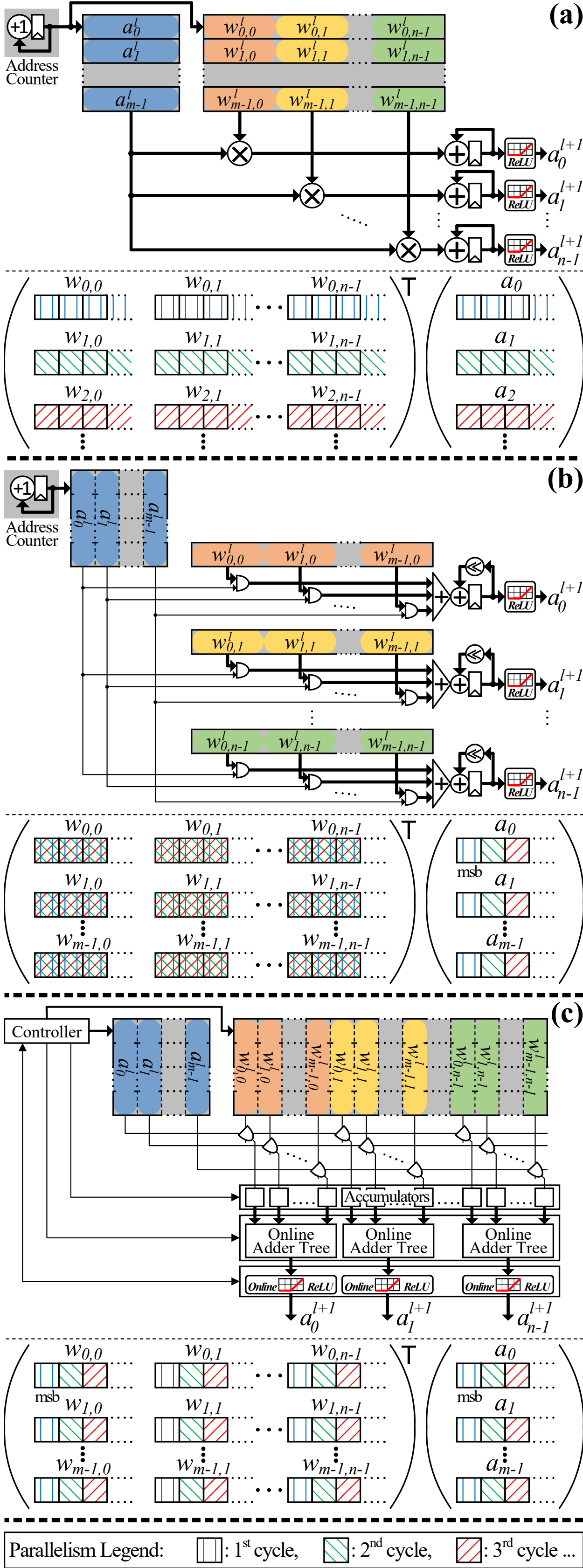


Fig. 4: Classification of neural inference parallelism: (a) Coarse-grained neuron-level parallelism (*e.g.*, DaDianNao [30]), (b) mixed-granularity parallelism (*e.g.*, Stripes [34] and Pragmatic [35], and (c) the proposed Fine-Grained Inference Engine (FGIE) with synapse-level parallelism.

A mixed-granularity bit-serial architecture is shown in Figure 4 (b). This class of inference breaks a multiply-add operation into a series of add-shift operations. The parallelism map of this model shows that several same-significance bits, one bit from each activation input, are processed together. These bits are AND'ed with the corresponding weights, then added together with the shifted and accumulated output. This operation yields the dot product of the input activations and the corresponding weights. This model requires high memory bandwidth to fetch and process synaptic weights in parallel. Conversely, the proposed approach evaluates several synapses, each within several neurons in parallel fine-grained computation as illustrated in Figure 4 (c).

In a departure from the conventional coarse-grained neuron-level parallelism, we propose a synapse-level parallelism by performing highly parallel time-multiplexed fine-grained neural computations. As depicted in the parallelism map of this approach, most significant bits are processed first, and most significant bits of the output are generated consequently. The fine-grained operations are performed online (MSB-first), which allows detecting at early computation stages if the neuron is firing. In case the neuron output is not firing, the computation can be stopped for energy saving. More detail on the functionality of this approach is provided next.

C. The Functionality of our Fine-Grained Inference Engine

A high-level overview of our proposed architecture is illustrated in Figure 4 (c). The multiply-add operation is fragmented into fine-grained operations. Both weights and activations are stored vertically in the BRAMs and are received as a bit-serial stream. The multiply-add operation of multiple synapses is described in Figure 5. Weight and activation bits that generate the MSB of the sum are read and processed first. The example given in Figure 5 describes a multiply operation with 4-bit operands. In this specific example, the third bit is the MSB that we want to generate first. All weight and activation bits that contribute to this summation are bits with indices that sum up to 3, thus the index pairs $\{(0,3),(1,2),(2,1),(3,0)\}$. For each synapse, these bits are serially AND'ed and the result is accumulated. Figure 5 shows the temporal order of the fine-grained operations that are required to generate the sum, starting from the MSB. Register "S" accumulates all values of same significance and passes the result to an online adder tree, followed by an online ReLU. Since the online ReLU is fed with MSBs first, it can detect early if the output passes the activation threshold.

MSDF online neural arithmetic A redundant number system is used to prevent any carry propagation. This is in contrary to ripple-carry arithmetic, where carry digits flow from the least-significant digits up to the most significant digits. Our architecture employs the Signed-Digit (SD) redundant number representation [40]. While in a common non-redundant number system of radix r the digit set $\{0, 1, \cdots, r-1\}$ is used, the redundant SD number system uses the digit set $\{-a, \cdots, -1, 0, 1, \cdots, a\}$, where $a \in \{\lceil r/2 \rceil, \cdots, r-1\}$. Due to this redundancy, the output MSD can be calculated

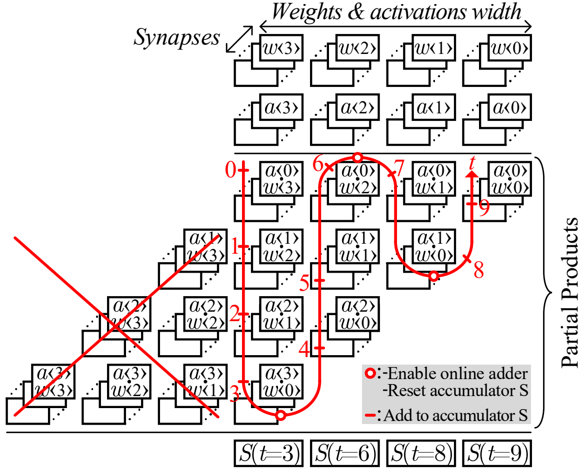


Fig. 5: Scheduling of digit-serial MSD-first multiply-add.

using partial digits of the inputs. Since each number can be represented in multiple ways, the value of the output can be revised when calculating the less-significant digits.

The MSDF online adder. The online adder is depicted is Figure 6. A digit-parallel online adder is shown in Figure 6 (middle). The core of this online adder is the redundant half-adder (rHA) unit. The rHA unit receives two number, x and y, of radix r and SD representation of $\{-a, \cdots, -1, 0, 1, \cdots, a\}$, and computes the sum, s and the curry out c using the same representation, namely,

$$\text{(c,s)=}\begin{cases}(\ \ 1, x+y-r) & if \quad x+y \geq \ \ a \\ (-1, x+y+r) & if \quad x+y \leq -a. \\ (\ \ 0, x+y \quad\ \ ) & otherwise\end{cases} \tag{5}$$

The main purpose of the rHA unit is to return the sum into the same redundant representation if the sum overflows the representation range. A bit-parallel redundant adder is depicted in Figure 6 (middle). This adder is implemented using the rHA unit and two-digit redundant adder. The indices of the inputs, x and y, and the output sum, z, represents the arrival time of the serial bits, namely, bits with lower indices are received first, thus are more significant. The online bit-serial adder is illustrated in Figure 6 (middle) and is designed by temporarily folding the bit-parallel redundant adder. The online adder is used to construct an online addition tree as depicted in Figure 4 (c), to sum the accumulated same-significant digits of all synapses (most significant first).

The MSDF online ReLU. As shown in (3), a bit-parallel ReLU passes positive inputs only. If the input is negative, the ReLU produces zero. a bit-parallel ReLU simply checks the sign bit of the inputs and generates the output accordingly. On the other hand, an MSDF bit-serial ReLU detects early (before seeing all input digits) if the input is negative and subsequently outputs zero, sends a "stop compute" signal backwards. The input of our bit-serial online ReLU is represented using the same SD redundant number system as of the online address. This representation allows our online ReLU to detect if its input is negative as soon as a non-zero digit is received.

The online ReLU receives its input serially. The digits are accumulated as they received to convert the redundant SD

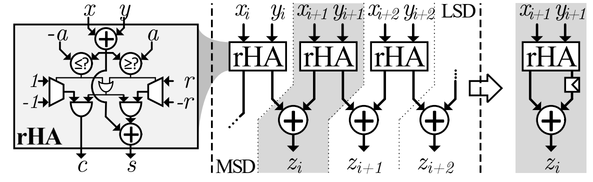


Fig. 6: (left) Redundant half-adder. (middle) Parallel redundant adder. (left) Online MSD-first adder.

representation into binary. The online ReLU detects that the output activation is zero if the current digit is negative while all previously received digits are zeros. If this condition is satisfied, the ReLU signals the neuron to stop computation and resets the accumulator, which consequently outputs zero to the next layer, otherwise the ReLU will pass the accumulated value to the next layer.

Scalability of the FGIE architecture. Massively parallel architectures as in Figure 4 support payloads of a limited size since processing resources are limited. To enable scalable architectures where any payload can be deployed, time-multiplexing of the processing resources is required. Although FGIE is designed as a synapse-parallel architecture, we limit the number of parallel synapses that are processed by a single FGIE tile, and time-multiplex each tile to enable scalability.

As depicted in Figure 8 (top), an FGIE tile is a basic Processing Element (PE), that is able to process s synapses of a single neuron in parallel. The FGIE tile receives s same-significant weight bits, and s same-significant activation bits, scheduled as in Figure 5. These bits are AND'ed to generate a single entry in the partial products, then all bits in the same column of the Partial products are accumulated. The scheduler generates the MSBs of the multiply result first (leftmost column of the partial products in Figure 5). The generated digits are then fed serially to an online adder tree (Figure 6), followed by an online ReLU (Figure 7).

As illustrated in Figure 8 (bottom), the FGIE tile is replicated p^i in layer i. This is the neural parallelism, namely the number of neurons that are computed simultaneously. Following the neural model, all FGIE tiles in the same layer are connected to the same activation memory, whereas each FGIE tile is connected to a dedicated weights memory.

Memory packing. While bit-parallel coarse-grained architectures arrange batches of weights and activations each in a memory line, our FGIE architecture packs weights and activations differently. The FGIE engine processes data serially based on bit significance, accordingly, bits of same significance from multiple synapses are stored in a single memory line. This allows for all bits of the same significance to be fetched in a single cycle. Figure 9 describes the packing of

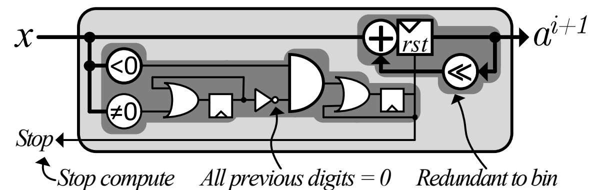
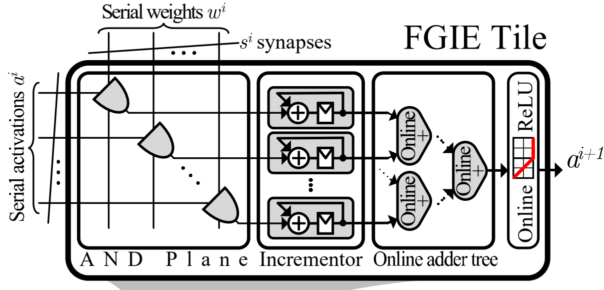


Fig. 7: Online ReLU. Detects negative inputs, stops compute and resets output.



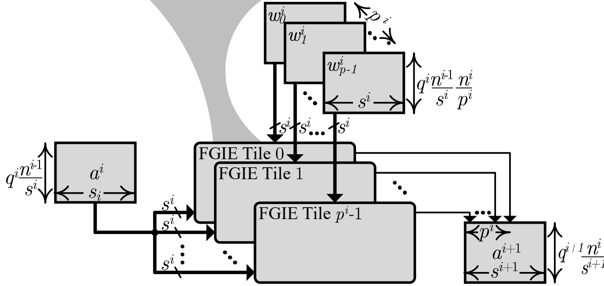


Fig. 8: (top) An FGIE tile. Processes s synapses in parallel. (bottom) An FGIE layer. Computes p neurons with s synapses each.

weights and activations in BRAMs. Bits of same significance are stored in blocks, each block contains n^{i-1}/s^i lines, where each line packs s^i bits. This is because an FGIE tile in layer i can process s^i bits simultaneously. Given that the fixed-points representation has q^i bits in layer i, the memory depth is thereby $q^i(n^{i-1}/s^i)$. Since an FGIE layer is time-multiplexed n^i/p^i times, it requires n^i/p^i times deeper memory.

To estimate the number of memory blocks needed to accommodate weights and activations, the packing function $\text{pack}(\text{BRAM}, d, w)$ tells how many memory blocks of a specific type are required to store a $d \times w$ data block. This is a device-dependent parameter and is based on the size and configurability of the memory block. The number of memory blocks required by level i is therefore

$$\boldsymbol{B}^i = \overbrace{\text{pack}(\text{BRAM}, q^i \frac{n^{i-1}}{s^i}, s^i)}^{\text{activation BRAMs for a single tile}} + \overbrace{\text{pack}(\text{BRAM}, q^i \frac{n^{i-1}}{s^i} \frac{n^i}{p^i}, s^i)}^{\text{weight BRAMs for a single tile}} p^i. \quad (6)$$

For instance, Intel's M20K blocks can be configured into several RAM depth and data width configurations [47]. The total amount of utilized SRAM bits can be either 16Kbits, or 20Kbits. Assuming that the RAM packing process minimizes the number of blocks cascaded in depth to avoid additional

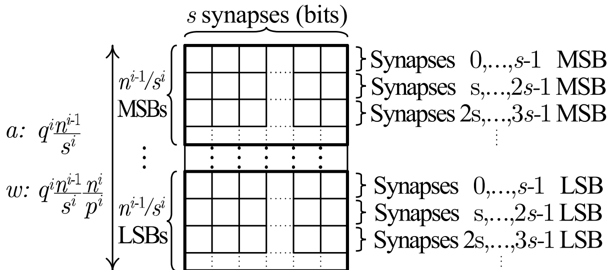


Fig. 9: Memory packing. Groups of s same-significance synapse bits are packed into the same memory line. Activations block requires $q^i n^{i-1}/s^i$ lines, while weights block require n^i/p^i times more.

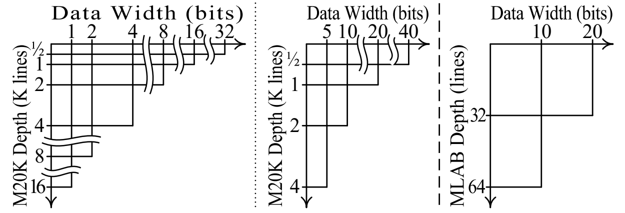


Fig. 10: Intel's FPGA on-chip memory (left) M20K 20Kb configuration (middle) M20K 16Kb configuration (right) MLAB 640-bit configuration.

address decoding, each 16K lines will be packed into single bit-wide blocks, and the remainder will be packed into the minimal required configuration. An estimation of the number of packed M20K blocks required to construct a RAM with a specific depth, d, and data width, w, is

$$\text{pack(M20K}, d, w) = \left\lfloor \frac{d}{16\text{k}} \right\rfloor w + \begin{cases} w & d \,\%\, 16\text{k} > 8\text{k} \\ \lceil w/2 \rceil & 8\text{k} \geq d \,\%\, 16\text{k} > 4\text{k} \\ \lceil w/5 \rceil & 4\text{k} \geq d \,\%\, 16\text{k} > 2\text{k} \\ \lceil w/10 \rceil & 2\text{k} \geq d \,\%\, 16\text{k} > 1\text{k} \\ \lceil w/20 \rceil & 1\text{k} \geq d \,\%\, 16\text{k} > \frac{1}{2}\text{k} \\ \lceil w/40 \rceil & \frac{1}{2}\text{k} \geq d \,\%\, 16\text{k} > 0 \\ 0 & otherwise \end{cases}. \quad (7)$$

On the other hand, Intel's Stratix 10 and Arria 10 devices provides full accessibility to LUT configuration memory as SRAM blocks with decoded addresses called MLAB [47] (also known as LUTRAM). However, the LUT configuration memory can be used either for LUT configuration or as part of the MLAB. For example, each ALM of a Stratix 10 device can accommodate a 6-input LUT, hence 64 configuration bits. Each 10 ALM (a single LAB) creates a simple dual-ported 64×10 or 32×20 MLAB block. MLABs can be also utilized to store activation bits. If the required memory is shallow, MLABs provice higher memory utilization compared to M20K. An estimation of the number of packed MLAB blocks required to construct a RAM with a specific depth, d, and data width, w, is

$$\text{pack(MLAB}, d, w) = \left\lfloor \frac{d}{64} \right\rfloor \left\lceil \frac{w}{10} \right\rceil + \begin{cases} \lceil w/10 \rceil & d \,\%\, 64 > 32 \\ \lceil w/20 \rceil & 32 \geq d \,\%\, 64 > 0 \\ 0 & otherwise \end{cases}. \quad (8)$$

To avoid BRAM underutilization, the depth of the required memory block $q^i \frac{n^{i-1}}{s^i} \frac{n^i}{p^i}$ is required to be at least the shallowest configuration of the BRAM, this is, for instance, 512 lines for M20K blocks, and 32 lines for MLAB blocks.

External memory support. Modern high-end FPGA devices integrate several hard on-chip DRAM controllers together with their configurable fabric. Although the latency and bandwidth of external memories are inferior to on-chip SRAM memory blocks, they are capable of storing a large amount of data, not possible to store on-chip. In case the on-chip memory is not sufficient to store the entire weights matrices, we propose to store a portion of the weights matrices in the external memory. Since the multiplication operation is fragmented, every weight bit is reused with all q activation bits, thereby we need to fetch a weight bit every q cycles. The wights are cached on-chip for reuse. For every layer i, the external memory is now required

TABLE IV: Relative Performance/Area and OPS/Watt Compared to DaDianNao [30] (base), Stripes [34], Loom [36], BISMO [37], and Moss *et al.* [38].

Network	Precision bit/layer	Size neuron/layer	Stripes [34] Perf/Area	Stripes [34] OPS/Watt	Loom [36] Perf/Area	Loom [36] OPS/Watt	BISMO [37] Perf/Area	BISMO [37] OPS/Watt	Moss *et al.* [38] Perf/Area	Moss *et al.* [38] OPS/Watt	FGIE (proposed) Perf/Area	FGIE (proposed) OPS/Watt
AlexNet [1]	10-9-9	4096×4096×1000	0.79	0.84	1.23	1.39	1.25	1.57	1.32	1.63	1.17	2.98
GoogLeNet [45]	7	1024	0.81	0.82	1.66	1.79	1.72	1.81	1.80	1.92	1.38	3.18
VGGS [46]	10-9-9	4096×4096×1000	0.79	0.83	1.19	1.34	1.24	1.52	1.35	1.61	1.14	2.97
VGGM [46]	10-8-8	4096×4096×1000	0.80	0.84	1.22	1.32	1.27	1.55	1.38	1.63	1.21	2.99
VGG19 [46]	10-9-9	4096×4096×1000	0.79	0.84	1.20	1.37	1.24	1.49	1.32	1.60	1.15	2.95
Geometric Mean			0.80	0.83	1.29	1.43	1.33	1.58	1.42	1.67	1.21	3.01

to transfer s^i bits for each one of the p^i FGIE tiles every q^i cycles. A total of

$$\sum_{i=0}^{l-1} s^i p^i / q^i \quad \text{bits/cycle.} \tag{9}$$

The weighted are packed in the same manner as described before, namely packed by bit-significance. Each external memory transfer cycle will bring a block of bits of same-significance.

For instance, Intel Stratix 10 MX devices integrate two HBM2 devices in a single package, enabling a memory bandwidth of 512 GBps. At a typical core frequency of 500 MHz, this interface provides 8388 bits/cycle. This bandwidth can support an FGIE of $(l, s, p, q) = (8, 128, 64, 8)$, for instance

IV. Experimental Results

To evaluate our proposed FIGE architecture and compare it to earlier techniques, fully parameterized Verilog modules have been developed. All different FGIE instances are synthesized, placed, and routed using Intel Quartus Prime 19.1 [48] targeting Intel's Arria10 GX1150 `10AX115U1F45E1SG` FPGA device [47] while enabling timing-driven synthesis, retiming, and other optimizations. This is the highest speed grade device with 427k ALMs and 2,713 M20K BRAMs (53Mb). Half of the ALMs can be used as MLABs, where each MLAB replaces 10 ALMs. This is a total of 20,774 MLAB blocks (12.68Mb). Timing and power measurements were obtained using Intel Quartus Prime Timing Analyzer and Power Analyzer, respectively. Power analysis is data-driven, namely activity factors are retrieved from actual inference instances. The network performance is obtained via Verilog cycle-accurate simulation. The FGIE is evaluated assuming all activations are stored on-chip, and weights are stored either on-chip or off-chip.

Table V shows different implementations of the FGIE architecture on Arria10 GX1150 device. The BRAM consumption is in agreement with (6). F_{max} measurements show that our architecture enables the device to run at high frequencies due to the fine-granularity nature of the computations. This is in agreement with the 500MHz fabric operation frequency of Arria10 GX devices, as reported by Intel [47].

Table IV reports the relative performance per area and energy efficiency of different bit-serial methods, including Stripes [34], Loom [36], BISMO [37], and Moss *et al.* [38], all relative to DaDianNao [30]. Similar to the inference model in Figure 4 (a), DaDianNao is a bit-parallel accelerator with 16-bit fixed-point weight and activations. DaDianNao can process several synapses of several neurons in parallel. Performance per area is measured as the ratio of the execution time and the consumed area. OPS/Watt is energy efficiency. All numbers reported in Table IV are relative to DaDianNao [30].

TABLE V: Resources Consumption of Multiple FGIE Instances.

s	p	Precision bit/layer	Size neuron/layer	ALMs	M20Ks	F_{max} (MHz)
64	32	8-9-10	1024×1024×1024	83,811	1,784	413
64	64	8-9-10	1024×1024×1024	140,816	1,835	387
128	16	8-9-10	1024×1024×1024	77,709	1,838	425
128	32	8-9-10	1024×1024×1024	138,343	1,852	394
256	16	8-9-10	1024×1024×1024	139,641	1,671	377

In Table IV we evaluate the fully-connected layer of 5 popular CNN architectures: AlexNet [1], GoogLeNet [45], VGGS, VGGM, and VGG19 [46]. The precisions profile per layer weights and activations is derived to prevent any loss of the top-1 accuracy. This precision varies from 7 to 10 bits. This follows the method of Judd *et al.* for fair comparison.

Stripes [34] does not employ mixed-weight precisions for fully-connected layers, thus its relative performance and energy efficiency is inferior to the other methods. The other methods (Loom [36], BISO [37], and the work of Moss *et al.* [38]), are all fine-grained bit-serial architectures. Their relative mean performance/area is between ×1.29 and ×1.42, and energy efficiency is between ×1.44 and ×1.67. Compared to other bit-serial methods, the online arithmetic overhead of FGIE incurs a minor reduction in the relative performance/area. However, the energy efficiency of FGIE is ×1.8 the best of other bit-serial methods thanks to the online arithmetic partial computation.

V. Conclusions and Future Work

In this paper, we propose a novel architecture of an efficient non-conventional deep learning inference engine, especially applicable for FPGA devices. Our design efficiently utilizes online MSDF arithmetic to stop computation and avoid ineffectual computations. This technique successfully improves energy efficiency by ×1.8 compared to other bit-serial methods, enables mixed-precision operations, supports high frequencies, and alleviates the memory bottleneck.

As future work, we are planning to extend the online computation to the complete network, not only inside layers. This allows generating the network output MSD-first. If the number of output MSD's is enough to make output discrimination (*e.g.*, classification with enough confidence) the computation will be stopped. Furthermore, FGIE architecture can be optimized for ASIC devices.

References

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. of the Int. Conf. on Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proc. of the Int. Conf. on Learning Representations (ICLR)*, May 2015.

[3] R. Girshick, "Fast R-CNN," in *Proc. of the IEEE Int. Conf. on Comput. Vision (ICCV)*, 2015, pp. 1440–1448.

[4] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Trans. on Pattern Anal. and Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, June 2017.

[5] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-Scale Video Classification with Convolutional Neural Networks," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recog. (CVPR)*, June 2014.

[6] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning Spatiotemporal Features with 3D Convolutional Networks," in *Proc. of the IEEE Int. Conf. on Comput. Vision (ICCV)*, 2015, pp. 4489–4497.

[7] G. Hinton *et al.*, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[8] D. Amodei *et al.*, "Deep Speech 2: End-to-end Speech Recognition in English and Mandarin," in *Proc. of the Int. Conf. on Mach. Learning (ICML)*, 2016, pp. 173–182.

[9] A. Gibiansky *et al.*, "Deep Voice 2: Multi-Speaker Neural Text-to-Speech," in *Proc. of the Int. Conf. on Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 2962–2970.

[10] A. van den Oord *et al.*, "Parallel WaveNet: Fast high-fidelity speech synthesis," in *Proc. of the Int. Conf. on Mach. Learning (ICML)*, vol. 80, July 2018, pp. 3915–3923.

[11] E. Arisoy, T. Sainath, B. Kingsbury, and B. Ramabhadran, "Deep Neural Network Language Models," in *Proc. of the NAACL-HLT 2012 Workshop*, 2012, pp. 20–28.

[12] X. Li, T. Qin, J. Yang, and T. Liu, "LightRNN: Memory and Computation-efficient Recurrent Neural Networks," in *Proc. of the Int. Conf. on Neural Inf. Process. Syst. (NIPS)*, 2016, pp. 4392–4400.

[13] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[14] A. Coates, B. Huval, T. Wang, D. Wu, A. Ng, and B. Catanzaro, "Deep Learning with COTS HPC Systems," in *Proc. of the Int. Conf. on Mach. Learning (ICML)*, 2013, pp. 1337–1345.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," in *Proc. European Conf. on Comput. Vision (ECCV)*, 2016, pp. 630–645.

[16] J. Dean *et al.*, "Large Scale Distributed Deep Networks," in *Proc. of the Int. Conf. on Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1223–1231.

[17] Q. V. Le *et al.*, "Building High-level Features Using Large Scale Unsupervised Learning," in *Proc. of the Int. Conf. on Mach. Learning (ICML)*, 2012, pp. 507–514.

[18] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recog. (CVPR)*, June 2011, pp. 109–116.

[19] N. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proc. Annu. Int. Symp. on Comput. Archit. (ISCA)*, 2017.

[20] E. Nurvitadhi *et al.*, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" in *Proc. of the ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 5–14.

[21] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. of the Int. Conf. on Artificial Intell. and Statistics (PMLR)*, vol. 15, Apr. 2011, pp. 315–323.

[22] S. Han *et al.*, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 243–254, June 2016.

[23] S. Han, H. Mao, and W. Dally, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding," *CoRR*, vol. arXiv:abs/1510.00149, 2015.

[24] D. Lin, S. Talathi, and V. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," in *Proc. of the Int. Conf. on Mach. Learning (ICML)*, 2016, pp. 2849–2858.

[25] B. Jacob *et al.*, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recog. (CVPR)*, June 2018, pp. 2704–2713.

[26] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural Networks with Few Multiplications," *CoRR*, vol. abs/1510.03009, 2015.

[27] S. Han *et al.*, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proc. Annu. Int. Symp. on Comput. Archit. (ISCA)*, 2016, pp. 243–254.

[28] Y. Choi, M. El-Khamy, and J. Lee, "Universal Deep Neural Network Compression," *CoRR*, vol. arXiv:abs/1802.02271, 2018.

[29] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing Neural Networks with the Hashing Trick," in *Proc. of the Int. Conf. on Mach. Learning (ICML)*, 2015, pp. 2285–2294.

[30] Y. Chen *et al.*, "DaDianNao: A Machine-Learning Supercomputer," in *Proc. of the Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, Dec. 2014, pp. 609–622.

[31] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recog. (CVPR)*, June 2011, pp. 109–116.

[32] S. Khoram and J. Li, "Adaptive quantization of neural networks," in *Proc. of the Int. Conf. on Learning Representations (ICLR)*, May 2018.

[33] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *Proc. Annu. Int. Symp. on Comput. Archit. (ISCA)*, June 2016.

[34] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. of the Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[35] J. Albericio *et al.*, "Bit-pragmatic Deep Neural Network Computing," in *Proc. of the Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, 2017, pp. 382–394.

[36] S. Sharify, A. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks," in *dac*, 2018, pp. 20:1–20:6.

[37] Y. Umuroglu, L. Rasnayake, and M. Själander, "BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing," in *Proc. of the Int. Conf. on Field Programmable Logic and Applications (FPL)*, Aug. 2018, pp. 307–3077.

[38] D. Moss *et al.*, "A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study," in *Proc. of the ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 107–116.

[39] M. D. Ercegovac and T. Lang, "On-line arithmetic for DSP applications," in *Proc. of the Midwest Symp. on Circuits and Syst. (MWSCAS)*, Aug. 1989, pp. 365–368 vol.1.

[40] A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 389–400, Sept. 1961.

[41] T. Kim, W. Jao, and S. Tjiang, "Circuit Optimization Using Carry–Save–Adder Cells," *tcad*, vol. 17, no. 10, pp. 974–984, Oct. 1998.

[42] D. Gudovskiy and L. Rigazio, "ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks," *CoRR*, vol. abs/1706.02393, 2017.

[43] F. Rieke, D. Warland, R. de Ruyter V., and W. Bialek, *Spikes: Exploring the Neural Code*. Cambridge, MA, USA: MIT Press, 1999.

[44] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, p. 211–252, Dec. 2015.

[45] C. Szegedy *et al.*, "Going Deeper with Convolutions," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recog. (CVPR)*, June 2015.

[46] S. Liu and W. Deng, "Very Deep Convolutional Neural Network Based Image Classification Using Small Training Sample Size," in *Proc. IAPR Asian Conf. on Pattern Recog. (ACPR)*, Nov. 2015.

[47] *Intel Arria 10 Core Fabric and General Purpose I/Os Handbook*, Santa Clara, CA, USA, May 2019.

[48] *Intel Quartus Prime Pro Edition Handbook*, Santa Clara, CA, USA, May 2017, version QPP5V1.