Bit-pragmatic Deep Neural Network Computing

Article · October 2016

CITATIONS
0

READS
19

5 authors, including:

Jorge Albericio
NVIDIA
21 PUBLICATIONS **33** CITATIONS

SEE PROFILE

Bit-Pragmatic Deep Neural Network Computing

Jorge Albericio, Patrick Judd, Alberto Delmás, Sayeh Sharify, Andreas Moshovos
Department of Electrical and Computer Engineering
University of Toronto
{jorge, juddpatr, a.delmaslascorz, sayeh, moshovos}@ece.utoronto.ca

***Abstract*—We quantify a source of ineffectual computations when processing the multiplications of the convolutional layers in Deep Neural Networks (DNNs) and propose *Pragmatic* (*PRA*), an architecture that exploits it improving performance and energy efficiency. The source of these ineffectual computations is best understood in the context of conventional multipliers which generate internally multiple *terms*, that is, products of the multiplicand and powers of two, which added together produce the final product [1]. At runtime, many of these terms are zero as they are generated when the multiplicand is combined with the zero-bits of the multiplicator. While conventional bit-parallel multipliers calculate all terms in parallel to reduce individual product latency, *PRA* calculates only the non-zero terms using a) on-the-fly conversion of the multiplicator representation into an explicit list of powers of two, and b) bit-parallel multplicand/bit-serial multiplicator processing units.**

***PRA* exploits two sources of ineffectual computations: 1) the aforementioned zero product terms which are the result of the *lack of explicitness* in the multiplicator representation, and 2) the *excess in the representation precision* used for both multiplicants and multiplicators, e.g., [2]. Measurements demonstrate that for the convolutional layers, a straightforward variant of *PRA* improves performance by 2.6x over the DaDiaNao (DaDN) accelerator [3] and by 1.4x over STR [4]. Similarly, *PRA* improves energy efficiency by 28% and 10% on average compared to DaDN and STR. An improved cross lane synchronization scheme boosts performance improvements to 3.1x over DaDN. Finally, *Pragmatic* benefits persist even with an 8-bit quantized representation [5].**

I. Introduction

Deep neural networks (DNNs) have become the state-of-the-art technique in many recognition tasks such as object [6] and speech recognition [7]. While DNN's have high computational demands, they are today practical to deploy given the availability of commodity Graphic Processing Units (GPUs) which can exploit the natural parallelism of DNNs. Yet, the need for even more sophisticated DNNs demands even higher performance and energy efficiency motivating special purpose architectures such as the state-of-the-art DaDianNao (*DaDN*) [3]. With power limiting modern high-performance designs, achieving better energy efficiency is essential can enable further advances [8].

DNNs comprise a pipeline of *layers* where more than 92% of the processing time is spent in convolutional layers [3], which this work targets. These layers perform inner products where *neurons* and *synapses* are multiplied in pairs, and where the resulting products are added to produce a single *output neuron*. A typical convolutional layer performs hundreds of inner products, each accepting hundreds to thousands neuron and synapse pairs.

DNN hardware typically uses either 16-bit fixed-point [3] or quantized 8-bit numbers [5] and bit-parallel compute units. Since the actual precision requirements vary considerably across DNN layers [2], typical DNN hardware ends up processing an excess of bits when processing these inner products [4]. Unless the values processed by a layer need the full value range afforded by the hardware's representation, an excess of bits, some at the most significant bit positions (prefix bits) and some at the least significant positions (suffix bits), need to be set to zero yet do not contribute to the final outcome. With bit-parallel compute units there is no performance benefit in not processing these excess bits.

Recent work, *Stripes* (*STR*) uses serial-parallel multiplication [9] to avoid processing these zero prefix and suffix bits [4] yielding performance and energy benefits. *STR* represents the neurons using pre-specified per layer precisions. Given a neuron n represented in p bits and a synapse s represented in, for example, 16-bits, *STR* processes n bit-serially over p cycles, where in each cycle one bit of n is multiplied by s accumulating the result into a running sum. While *STR* takes p cycles to compute each product, it can ideally improve performance by $16/p$ compared to a 16-bit fixed-point bit-parallel hardware by processing $16\times$ more neurons and synapse pairs in parallel. The abundant parallelism of DNN convolutional layers makes this possible.

While *STR* avoids processing the ineffectual suffix and tail bits of neurons that are due to the one-size-fits-all representation of conventional bit-parallel hardware, it still processes many ineffectual neuron bits: Any time a zero bit is multiplied by a synapse it adds nothing to the final output neuron. These ineffectual bits are introduced by the conventional positional number representation. If these multiplications could be avoided it would take even less time to calculate each product improving energy and performance. Section II shows that in state-of-the-art image classification networks show that 93% and 69% of neuron bit and synapse products are ineffectual when using respectively 16-bit fixed-point and 8-bit quantized representations.

This work presents *Pragmatic* (*PRA*) a DNN accelerator whose goal is to process only the *essential* (non-zero) bits of the input neurons. *PRA* subsumes *STR* not only since a) it avoids processing non-essential bits regardless of their position, but also as b) it obviates the need to determine *a priori* the specific precision requirements per layer. *PRA* employs the following four key techniques: 1) on-the-fly conversion of neurons from a storage representation (e.g., conventional

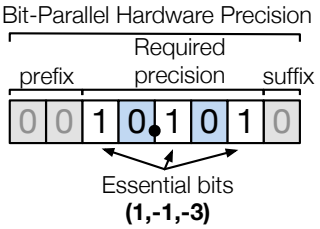


Fig. 1. Sources of ineffectual computation with conventional positional representation and fixed-length hardware precision.

positional number or quantized) into an explicit representation of the essential bits only, 2) bit-serial neuron/bit-parallel synapse processing, an idea borrowed from *STR* but adapted for the aforementioned representation, 3) judicious SIMD (single instruction multiple data) lane grouping to maintain wide memory accesses and to avoid fragmenting and enlarging the multi-MB on-chip synapse memories (Sections V-A4 and V-E), and 4) computation re-arrangement (Section V-D) to reduce datapath area. All evaluated *PRA* variants maintain wide memory accesses and use highly-parallel SIMD-style (single-instruction multiple-data) computational units. *PRA* introduces an additional dimension upon which software can improve performance and energy efficiency by controlling neuron values judiciously in order to reduce their essential bit content while maintaining accuracy. This work explores such an alternative, where the software explicitly communicates how many prefix and suffix bits to discard after each layer.

Experimental measurements with state-of-the-art DNNs demonstrate that most straightforward *PRA* variant, boosts average performance for the convolutional layers to 2.59x over the state-of-the-art *DaDN* accelerator compared to the 1.85x performance improvement of *STR* alone. *Pragmatic*'s average energy efficiency is 1.48x over *DaDN* and its area overhead is 1.35x. Another variant further boosts performance to 3.1x over *DaDN* at the expense of an additional 0.7% area. Software guidance accounts for 19% of these performance benefits.

II. Motivation

Let us assume a p-bit bit-parallel multiplier using a straightforward implementation of the "Shift and Add" algorithm where $n \times s$ is calculated as $\sum_{i=0}^{p} n_i \cdot (s \ll i)$, where n_i the i-*th* bit of n. The multiplier computes p *terms*, each a product of s and of a bit of n, and adds them to produce the final result. The terms and their sum can be calculated concurrently to reduce latency [1].

With such a hardware arrangement there are two sources of ineffectual computations that result from: 1) an *Excess of Precision* (EoP), and 2) *Lack of Explicitness* (LoE). Figure 1 shows an example illustrating these sources with a bit-parallel multiplier using an 8-bit unsigned fixed-point number with 4 fractional and 4 integer bits. While $10.101_{(2)}$ requires just five bits, our 8-bit bit-parallel multiplier will zero-extend it with two prefix and one suffix bits. This is an example of EoP and is due to the fixed-precision hardware. Two additional ineffectual bits appear at positions 1 and -2 as a result of LoE in the positional number representation. In total, five ineffectual bits will be processed generating five ineffectual terms.

Our number could be represented with an explicit list of its three constituent powers of 2: (1,-1,-3). While such a representation may require more bits and thus be undesirable for storage, coupled with the abundant parallelism that is present in DNNs layers, it provides an opportunity to revisit hardware design improving performance and energy efficiency.

The rest of this section motivates *Pragmatic* by: 1) measuring the fraction of non-zero bits in the neuron stream of state-of-the-art DNNs for three commonly used representations, and 2) estimating the performance improvement which may be possible by processing only the non-zero neuron bits.

A. Essential Neuron Bit Content

Table V reports the *essential bit content* of the neuron stream of state-of-the-art DNNs for two commonly used fixed length representations: 1) 16-bit fixed-point of DaDianNao [3], 2) 8-bit quantized of Tensorflow [5]. The essential bit content is the average number of non-zero bits that are 1. Two measurements are presented per representation: over all neuron values ("All"), and over the non-zero neurons ("NZ") as accelerators that can skip zero neurons for fixed-point representations have been recently proposed [10], [11].

When considering all neurons, the essential bit-content is at most 12.7% and 38.4% for the fixed-point and the quantized representations respectively. The measurements are consistent with the neuron values following a normal distribution centered at 0, and then being filtered by a rectifier linear unit (ReLU) function [12]. Even when considering the non-zero neurons the essential bit content remains well below 50% and as the next section will show, there are many non-zero valued neurons suggesting that the potential exists to improve performance and energy efficiency over approaches that target zero valued neurons.

These results suggest that a significant number of ineffectual terms are processed with conventional fixed-length hardware. *Stripes* [4], tackles the excess of precision, exploiting the variability in numerical precision DNNs requirements to increase performance by processing the neurons bit-serially. *Pragmatic*'s goal is to also exploit the lack of explicitness. As the next section will show, *Pragmatic* has the potential to greatly improve performance even when compared to *Stripes*.

B. Pragmatic's Potential

To estimate *PRA*'s potential, this section compares the number of terms that would be processed by various computing engines for the convolutional layers of state-of-the-art DNNs (see Section VI-A) for the two aforementioned baseline neuron representations.

16-bit Fixed-Point Representation: The following computing engines are considered: 1) baseline representative of *DaDN* using 16-bit fixed-point bit-parallel units [3], 2) a *hypothetical* enhanced baseline ZN, that can skip *all* zero valued neurons,

	Alexnet	NiN	Google	VGGM	VGGS	VGG19
16-bit Fixed-Point						
All	7.8%	10.4%	6.4%	5.1%	5.7%	12.7%
NZ	18.1%	22.1%	19.0%	16.5%	16.7%	24.2%
8-bit Quantized						
All	31.4%	27.1%	26.8%	38.4%	34.3%	16.5%
NZ	44.3%	37.4%	42.6%	47.4%	46.0%	29.1%

TABLE I
AVERAGE FRACTION OF NON-ZERO BITS PER NEURON FOR TWO FIXED-LENGTH REPRESENTATIONS: 16-BIT FIXED-POINT, AND 8-BIT QUANTIZED. **ALL:** OVER ALL NEURONS. **NZ:** OVER NON-ZERO NEURONS ONLY.

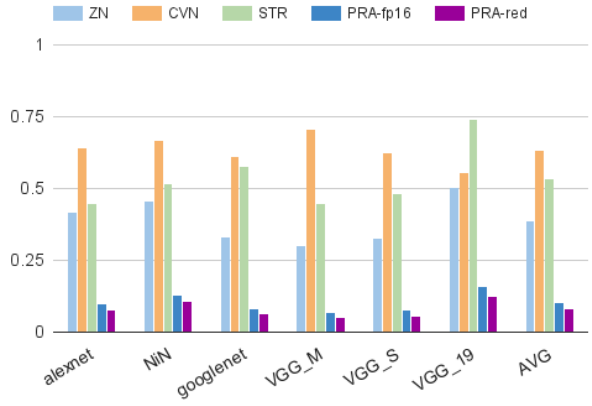


Fig. 2. Convolutional layer computational demands with a 16-bit fixed-point baseline representation. Lower is better.

3) Cnvlutin (CVN) a practical design that can skip zero value neurons for all but the first layer [11], 4) *STR* that avoids EoP (see Table II, Section VI-A) [4], 5) an ideal, software-transparent *PRA*, *PRA*-fp16 that processes only the essential neuron bits, and 6) an ideal *PRA*, *PRA*-red, where software communicates in advance how many prefix and suffix bits can be zeroed out after each layer (see Section V-F).

Figure 2 reports the number of terms normalized over *DaDN* where each multiplication is accounted for using an equivalent number of terms or equivalently additions: 16 for *DaDN*, ZN, and CVN, p for a layer using a precision of p bits for *STR*, and the number of essential neuron bits for *PRA*-fp16, and for *PRA*-red. For example, for $n = 10.001_{(2)}$, the number of additions counted would be 16 for *DaDN* and CVN+, 5 for *STR* as it could use a 5-bit fixed-point representation, and 2 for *PRA*-fp16 and *PRA*-red.

On average, *STR* reduces the number of terms to 53% compared to *DaDN* while skipping just the zero valued neurons could reduce them to 39% if ZN was practical and to 63% in practice with CVN. *PRA*-fp16 can ideally reduce the number of additions to just 10% on average, while with software provided precisions per layer, *PRA*-red reduces the number of additions further to 8% on average. The potential savings are robust across all DNNs remaining above 87% for all DNNs with *PRA*-red.

8-bit Quantized Representation: Figure 3 shows the relative

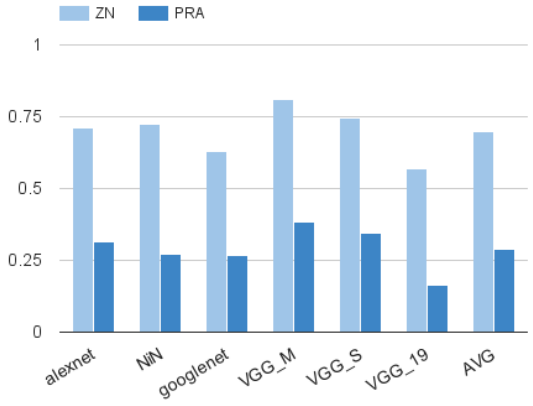


Fig. 3. Convolution layer computational demands with an 8-bit quantized baseline representation. Lower is better.

number of terms processed for: 1) a bit-parallel baseline, 2) an ideal, yet impractical bit-parallel engine that skips all zero neurons, and 3) *PRA*. In the interest of space and since *PRA* subsumes *STR* and CVN they are not considered. *Pragmatic*'s potential benefits are significant even with an 8-bit quantized representation. On average, skipping all the zero valued neurons would eliminate only 30% of the terms whereas *Pragmatic* would remove up to 71% of the terms.

In summary, this section corroborated past observations that: a) many neuron values are zero [10], [11], [13], [14], and b) only close to a half of the computations performed traditionally is needed if numerical precision is properly adjusted [4]. It further showed that far less computations are really needed, 10% and 29% on average for the 16-bit fixed-point and 8-bit quantized representations respectively, if only the essential neuron bits were processed. Finally, software can boost the opportunities for savings by communicating per layer precisions.

III. *Pragmatic*: A SIMPLIFIED EXAMPLE

This section illustrates the idea behind *Pragmatic* via a simplified example. For the purposes of this discussion suffices to know that in a convolutional layer there are typically hundreds to thousands of neurons, each multiplied with a corresponding synapse, and that the synapses are reused several times. Section IV-A describes the relevant computations in more detail.

The bit-parallel unit of Figure 4a multiplies two neurons with their respective synapses and via an adder reduces the two products. The unit reads *all* neuron and synapse bits, $(n_0 = 001_{(2)}, n_1 = 010_{(2)})$ and $(s_0 = 001_{(2)}, s_1 = 111_{(2)})$ respectively in a single cycle. As a result, the two sources of inefficiency EoP and LoE manifest here: n_0 and n_1 are represented using 3 bits instead of 2 respectively due to EoP. Even in 2 bits, they each contain a zero bit due to LoE. As a result, four ineffectual terms are processed when using standard multipliers such as those derived from the Shift and

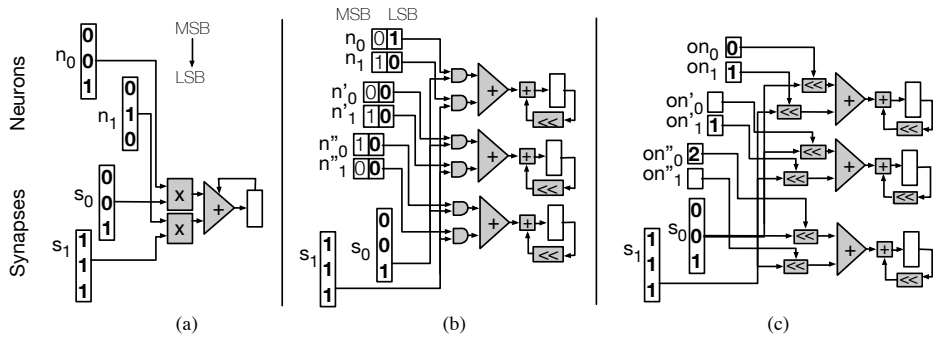


Fig. 4. a) Bit-parallel unit. b) Bit-serial unit with equivalent throughput (*Stripes*[4]). c) *Pragmatic* unit with equivalent throughput where only essential information is processed.

Add algorithm. In general, given N neuron and synapse pairs, this unit will take $\lceil N/2 \rceil$ cycles to process them regardless of their precision and the essential bit content of the neurons.

The hybrid, bit-serial-neuron/bit-parallel-synapse unit in Figure 4b is representative of *STR* which tackles EoP. Each cycle, the unit processes one bit from each neuron and hence it takes three cycles to compute the convolution when the neurons are represented using 3 bits each, a slowdown of 3x over the bit-parallel engine. To match the throughput of the bit-parallel engine of Figure 4a, *STR* takes advantage of synapse reuse and processes multiple neurons groups in parallel. In this example, six neurons ($n_0 = 001_{(2)}, n_1 = 010_{(2)}, n'_0 = 000_{(2)}, n'_1 = 010_{(2)}, n''_0 = 010_{(2)}, n''_1 = 000_{(2)}$) are combined with the two synapses as shown. Starting from the least significant position, each cycle one bit per neuron is *ANDed* with the corresponding synapse. The six AND results are added via the reduction tree and the result is accumulated after being shifted by one bit. Since the specific neuron values could be represented all using 2 bits, *STR* would need 2 cycles to process all six products compared to the 3 cycles needed by the bit-parallel system, a $3/2\times$ speedup. However, *Stripes* still processes some ineffectual terms. For example, in the first cycle, 4 of the 6 terms are zero yet they are added via the adder tree, wasting computing resources and energy.

Figure 4c shows a simplified *PRA* engine. In this example, neurons are no longer represented as vectors of bits but as vectors of offsets of the essential bits. For example, neuron $n_0 = 001_{(2)}$ is represented as $on_0 = (0)$, and a neuron value of $111_{(2)}$ would be represented as $(2, 1, 0)$. An out-of-band bit (wire) not shown indicates the neuron's end. A shifter per neuron uses the offsets to effectively multiply the corresponding synapse with the respective power of 2 before passing it to the adder tree. As a result, *PRA* processes only the non-zero terms avoiding all ineffectual computations that were due to EoP or LoE. For this example, *PRA* would process six neuron and synapse pairs in a single cycle, a speedup of $3\times$ over the bit-parallel engine.

IV. Background

This work presents *Pragmatic* as a modification of the state-of-the-art *DaDianNao* accelerator. Accordingly, this section provides the necessary background information: Section IV-A reviews the operation of convolutional layers, and Section IV-B overviews *DaDN* and how it processes convolutional layers.

A. Convolutional Layer Computation

A convolutional layer processes and produces neuron arrays, that is 3D arrays of real numbers. The layer applies N 3D filters in a sliding window fashion using a constant stride S to produce an output 3D array. The input array contains $N_x \times N_y \times I$ *neurons*. Each of the N filters, contains $F_x \times F_y \times I$ *synapses* which are also real numbers. The output neuron array dimensions are $O_x \times O_y \times N$, that is its depth equals the filter count. Each filter corresponds to a desired *feature* and the goal of the layer is to determine where in the input neuron array these features appear. Accordingly, each constituent 2D array along the i dimension of the output neuron array corresponds to a *feature*. To calculate an output neuron, the layer applies one filter over a *window*, a filter-sized, or $F_x \times F_y \times I$ sub-array of the input neuron array. If $n(x, y, i)$ and $o(x, y, i)$ are respectively input and output neurons, and $s^f(x, y, i)$ are the synapses of filter f. The output neuron at position $(\mathbb{k}, \mathbb{l}, \mathbb{f})$ is given by:

$$\underbrace{o(\mathbb{k},\mathbb{l},\mathbb{f})}_{\substack{output\\neuron}} = \underbrace{\sum_{y=0}^{F_y-1}\sum_{x=0}^{F_x-1}\sum_{i=0}^{I-1}\underbrace{s^{\mathbb{f}}(y,x,i)}_{synapse}\times\underbrace{n(y+\mathbb{l}\times S, x+\mathbb{k}\times S, i)}_{input\ neuron}}_{window}$$

The layer applies filters repeatedly over different windows positioned along the X and Y dimensions using a constant stride S, and there is one output neuron per window and filter. Accordingly, the output neuron array dimensions are $O_x = (I_x - F_x)/S + 1$, $O_y = (I_y - F_y)/S + 1$, and $O_i = N$. .

1) Terminology – Bricks and Pallets:: For clarity, in what follows the term *brick* refers to a set of 16 elements of a 3D neuron or synapse array which are contiguous along the

i dimension, e.g., $n(x, y, i)...n(x, y, i + 15)$. Bricks will be denoted by their origin element with a B subscript, e.g., $n_B(x, y, i)$. The term *pallet* refers to a set of 16 bricks corresponding to adjacent, using a stride S, windows along the x or y dimensions, e.g., $n_B(x, y, i)...n_B(x, y + 15 \times S, i)$ and will be denoted as $n_P(x, y, i)$. The number of neurons per brick, and bricks per pallet are design parameters.

B. Baseline System: DaDianNao

Pragmatic is demonstrated as a modification of the *DaDianNao* accelerator (*DaDN*) proposed by Chen *et al.* [3]. Figure 5a shows a *DaDN* tile which processes 16 filters concurrently calculating 16 neuron and synapse products per filter for a total of 256 products per cycle. To do, each cycle the tile accepts 16 synapses per filter for total of 256 synapses, and 16 input neurons. The tile multiplies each synapse with only one neuron whereas each neuron is multiplied with 16 synapses, one per filter. The tile reduces the 16 products into a single partial output neuron per filter, for a total of 16 partial output neurons for the tile. Each *DaDN* chip comprises 16 such tiles, each processing a different set of 16 filters per cycle. Accordingly, each cycle, the whole chip processes 16 neurons and $256 \times 16 = 4K$ synapses producing $16 \times 16 = 256$ partial output neurons.

Internally, each tile has: 1) a synapse buffer (SB) that provides 256 synapses per cycle one per synapse lane, 2) an input neuron buffer (NBin) which provides 16 neurons per cycle through 16 neuron lanes, and 3) a neuron output buffer (NBout) which accepts 16 partial output neurons per cycle. In the tile's datapath, or the *Neural Functional Unit* (NFU) each neuron lane is paired with 16 synapse lanes one from each filter. Each synapse and neuron lane pair feed a multiplier and an adder tree per filter lane reduces the 16 per filter products into a partial sum. In all, the filter lanes produce each a partial sum per cycle, for a total of 16 partial output neurons per NFU. Once a full window is processed, the 16 resulting sums, are fed through a non-linear activation function, f, to produce the 16 final output neurons. The multiplications and reductions needed per cycle are implemented via 256 multipliers one per synapse lane and sixteen 17-input (16 products plus the partial sum from NBout) adder trees one per filter lane.

DaDN's main goal was minimizing off-chip bandwidth while maximizing on-chip compute utilization. To avoid fetching synapses from off-chip, *DaDN* uses a 2MB eDRAM SB per tile for a total of 32MB eDRAM. All inter-layer neuron outputs except for the initial input and the final output are stored in a 4MB shared central eDRAM *Neuron Memory* (NM) which is connected via a broadcast interconnect to the 16 NBin buffers. Off-chip accesses are needed only for reading the input image, the synapses once per layer, and for writing the final output.

Processing starts by reading from external memory the first layer's filter synapses, and the input image. The synapses are distributed over the SBs and the input is stored into NM. Each cycle an input neuron brick is broadcast to all units. Each units reads 16 synapse bricks from its SB and produces a partial output neuron brick which it stores in its NBout. Once computed, the output neurons are stored through NBout to NM and then fed back through the NBins when processing the next layer. Loading the next set of synapses from external memory can be overlapped with the processing of the current layer as necessary.

V. Pragmatic

This section presents the *Pragmatic* architecture. Section V-A describes *PRA*'s processing approach while Section V-B describes its organization. Sections V-D and V-E present two optimizations that respectively improve area and performance. For simplicity, the description assumes specific values for various design parameters so that *PRA* performance matches that of the *DaDN* configuration of Section IV-B in the worst case.

A. Approach

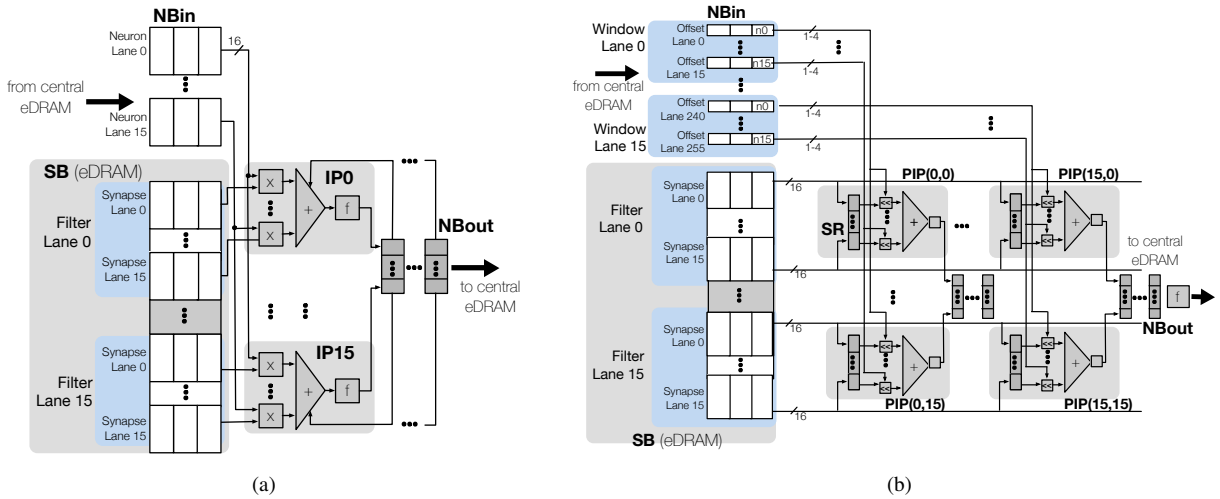
PRA's goal is to process only the essential bits of the input neurons. To do so *PRA* a) converts, on-the-fly, the input neuron representation into one that contains only the essential bits, and b) processes one essential bit per neuron and a full 16-bit synapse per cycle. Since *PRA* processes neuron bits serially, it may take up to 16 cycles to produce a product of a neuron and a synapse. To always match or exceed the performance of the bit-parallel units of *DaDN*, *PRA* processes more neurons concurrently exploiting the abundant parallelism of the convolutional layers. The remaining of this section describes in turn: 1) an appropriate neuron representation, 2) the way *PRA* calculates terms, 3) how multiple terms are processed concurrently to maintain performance on par with *DaDN* in the worst case, and 4) how *PRA*'s units are supplied with the necessary neurons from NM.

1) Input Neuron Representation: PRA starts with an input neuron representation where it is straightforward to identify the next essential bit each cycle. One such representation is an explicit list of *oneffsets*, that is of the constituent powers of two. For example, a neuron $n = 5.5_{(10)} = 0101.1_{(2)}$ would be represented as $n = (2, 0, -1)$. In the implementation described herein, neurons are stored in 16-bit fixed-point in NM, and converted on-the-fly in the *PRA* representation as they are broadcast to the tiles. A single oneffset is processed per neuron per cycle. Each oneffset is represented as (pow, eon) where pow is a 4-bit value and eon a single bit which if set indicates the end of a neuron. For example, $n = 101_{(2)}$ is represented as $n^{PRA} = ((0010, 0)(0000, 1))$. In the worst case, all bits of an input neuron would be 1 and hence its *PRA* representation would contain 16 oneffsets.

2) Calculating a Term: PRA calculates the product of synapse s and neuron n as:

$$s \times n = \sum_{\forall f \in n^{PRA}} s \times 2^f = \sum_{\forall f \in n^{PRA}} (n \ll f)$$

That is, each cycle, the synapse s multiplied by f, the next constituent power two of n, and the result is accumulated. This multiplication can be implemented as a shift and an AND.



(a) (b)

Fig. 5. a) DaDianNao Tile. b) Pragmatic Tile.

3) Boosting Compute Bandwidth over DaDN: To match *DaDN*'s performance *PRA* needs to process the same number of effectual terms per cycle. Each *DaDN* tile calculates 256 neuron and synapse products per cycle, or $256 \times 16 = 4K$ terms. While most of these terms will be in practice ineffectual, to guarantee that *PRA* always performs as well as *DaDN* it should process $4K$ terms per cycle. For the time being let us assume that all neurons contain the same number of essential bits, so that when processing multiple neurons in parallel, all units complete at the same time and thus can proceed with the next set of neurons in sync. The next section will relax this constraint.

Since *PRA* processes neurons bits serially, it produces one term per neuron bit and synapse pair and thus needs to process $4K$ such pairs concurrently. The choice of which $4K$ neuron bit and synapse pairs to process concurrently can adversely affect complexity and performance. For example, it could force an increase in SB capacity and width, or an increase in NM width, or be ineffective due to unit underutilization given the commonly used layer sizes.

Fortunately, it is possible to avoid increasing the capacity and the width of the SB and the NM while keeping the units utilized as in *DaDN*. Specifically, a *PRA* tile can read 16 synapse bricks and the equivalent of 256 neuron bits as *DaDN*'s tiles do (*DaDN* processes 16 16-bit neurons or 256 neuron bits per cycle). Specifically, as in *DaDN*, each *PRA* tile processes 16 synapse bricks concurrently, one per filter. However, differently than *DaDN* where the 16 synapse bricks are combined with just one neuron brick which is processed bit-parallel, *PRA* combines each synapse brick with 16 neuron bricks, one from each of 16 windows, which are processed bit-serially. The same 16 neuron bricks are combined with all synapse bricks. These neuron bricks form a *pallet* enabling the same synapse brick to be combined with all. For example, in a single cycle a *PRA* title processing filters 0 through 15 could combine combine $s_B^0(x, y, 0), ..., s_B^1 5(x, y, 0)$ with $n_B^{PRA}(x, y, 0), n_B^{PRA}(x + 2, y, 0), ...n_B^{PRA}(x + 31, y, 0)$ assuming a layer with a stride of 2. In this case, $s^4(x, y, 2)$ would be paired with $n^{PRA}(x, y, 2)$, $n^{PRA}(x + 2, y, 2)$, ..., $n^{PRA}(x + 31, y, 2)$ to produce the output neurons $on(x, y, 4)$ through $on(x + 15, y, 4)$.

As the example illustrates, this approach allows each synapse to be combined with one neuron per window whereas in *DaDN* each synapse is combined with one neuron only. In total, 256 essential neuron bits are processed per cycle and given that there are 256 synapses and 16 windows, *PRA* processes $256 \times 16 = 4K$ neuron bit and synapse pairs, or terms per cycle producing 256 partial output neurons, 16 per filter, or 16 partial output neuron bricks per cycle.

4) Supplying the Input Neuron and Synapse Bricks: Thus far it was assumed that all input neurons have the same number of essential bits. Under this assumption, all neuron lanes complete processing their terms at the same time, allowing *PRA* to move on to the next neuron pallet and the next set of synapse bricks in one step. This allows *PRA* to reuse *STR*'s approach for fetching the next pallet from the single-ported NM [4]. Briefly, with unit stride the 256 neurons would be typically all stored in the same NM row or at most over two adjacent NM rows and thus can be fetched in at most two cycles. When the stride is more than one, the neurons will be spread over multiple rows and thus multiple cycles will be needed to fetch them all. Fortunately, fetching the next pallet can be overlapped with processing the current one. Accordingly, if it takes NM_C to access the next pallet from NM, while the current pallet requires P_C cycles to process, the next pallet will begin processing after $max(NM_C, P_C)$ cycles. When $NM_C > P_C$ performance is lost waiting for NM.

In practice it highly unlikely that all neurons will have the same number of essential bits. In general, each neuron lane if left unrestricted will advance at a different rate. In the worst case, each neuron lane may end up needing neurons from a

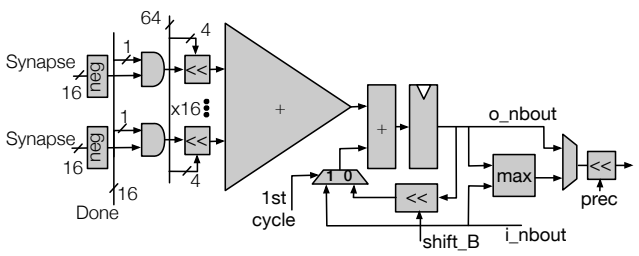


Fig. 6. Pragmatic Inner Product Unit.

different neuron brick, thus breaking *PRA*'s ability to reuse the same synapse brick. This is undesirable if not impractical as it would require partitioning and replicating the SB so that 4K unrelated synapses could be read per cycle, and it would also increase NM complexity and bandwidth.

Fortunately, these complexities can be avoided with *pallet-level neuron lane synchronization* where all neuron lanes "wait" (a neuron lane that has detected the end of its neuron forces zero terms while waiting) for the one with the most essential bits to finish before proceeding with the next pallet. Under this approach, it does not matter which bits are essential per neuron, only how many exist. Since, it is unlikely that most pallets will contain a neuron with 16 essential terms, *PRA* will improve performance over *DaDN*. Section VI-B will show that in practice, this approach improves performance over *DaDN* and *STR*. Section V-E will discuss finer-grain synchronization schemes that lead to even better performance. Before doing so, however, the intervening sections detail *PRA*'s design.

B. Tile Organization and Operation

Figure 5b shows the *Pragmatic* tile architecture which comprises an array of $16 \times 16 = 256$ *pragmatic inner product units (PIPs)*. PIP(i,j) processes a neuron oneffset from the i-th window and its corresponding synapse from the j-th filter. Specifically, all the PIPs along the i-th row receive the same synapse brick belonging to the i-th filter and all PIPs along the j-th column receive an oneffset from each neuron from one neuron brick belonging to the j-th window.

The necessary neuron oneffsets are read from NBin where they have been placed by the Dispatcher and the Oneffset generators units as Section V-C explains. Every cycle NBin sends 256 oneffsets 16 per window lane. All the PIPs in a column receive the same 16 oneffsets, corresponding to the neurons of a single window. When the tile starts to process a new neuron pallet, 256 synapses are read from SB through its 256 synapse lanes as in *DaDN* and are stored in the synapse registers (SR) of each PIP. The synapse and oneffsets are then processed by the PIPs as the next section describes.

1) Pragmatic Inner-Product Unit: Figure 6 shows the PIP internals. Every cycle, 16 synapses are combined with their corresponding oneffsets. Each oneffsets controls a shifter effectively multiplying the synapse with a power of two. The shifted synapses are reduced via the adder tree. An AND gate per synapse supports the injection of a null terms when necessary. In the most straightforward design, the oneffsets use 4-bits, each shifter accepts a 16-bit synapse and can shift it by up to 15 bit positions producing a 31-bit output. Finally, the adder tree accepts 31-bit inputs. Section V-D presents an enhanced design that requires narrower components improving area and energy.

C. Dispatcher and Oneffset Generators

The *Dispatcher* reads 16 neuron bricks from NM, as expected by the *PRA* tiles. The *oneffset generator* converts their neurons on-the-fly to the oneffset representation, and broadcasts one oneffset per neuron per cycle for a total of 256 oneffsets to all titles. Fetching and assembling the 16 neuron bricks from NM is akin to fetching words with a stride of S from a cache structure. As Section V-A4 discussed this can take multiple cycles depending on the stride and alignment of the initial neuron brick. *PRA* uses the same dispatcher design as *STR* [4].

Once the 16 neuron bricks have been collected, 256 oneffset generators operate in parallel to locate and communicate the next oneffset per neuron. A straightforward 16-bit leading one detector is sufficient. The latency of the oneffset generators and the dispatcher can be readily hidden as they can be pipelined as desired overlapping them with processing in the *PRA* tiles.

D. 2-Stage Shifting

Any shift can be performed in two stages as two smaller shifts: $a \ll K = a \ll (K' + C) = ((a \ll K') \ll C)$. Thus, to shift and add T synapses by different offsets $K_0, ..., K_T$, we can decompose the offsets into sums with a common term C, e.g., $K_i = K'_i + C$. Accordingly, PIP processing can be rearranged using a two stage processing where the first stage uses the synapse specific offsets K'_i, and the second stage, the common across all synapses offset C:

$$\underbrace{\sum_{i}^{T}(S_i \ll K_i)}_{1-stage\ shifting} = \sum_{i}^{T}(S_i \ll (K'_i + C)) =$$

$$\underbrace{(\sum_{i}^{T}(S_i \ll K'_i))}_{1st\ stage} \underbrace{\ll C}_{2nd\ stage}$$

This arrangement can be used to reduce the width of the synapse shifters and of the adder tree by sharing one common shifter after the adder tree as Figure 7a shows. A design parameter, L, defines the number of bits controlling the synapse shifters. Meaning the design can process oneffsets which differ by less than 2^L in a single cycle. This reduces the size of the synapse shifters and reduces the size of the adder tree to support terms of $16+2^L-1$ bits only. As Section VI-B shows, this design reduces the area of the shifters and the adder trees which are the largest components of the PIP. Figure 7b shows an example illustrating how this PIP can handle any combination of oneffsets. Section VI-B studies the impact of L on cost and performance.

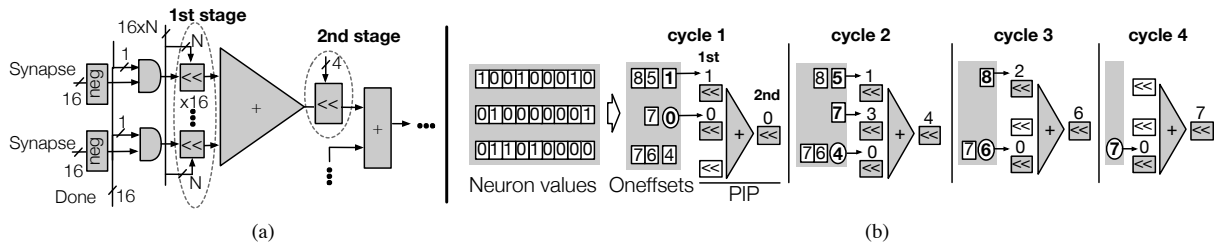


Fig. 7. 2-stage shifting. a) Modified PIP. b) Example: Processing three 9-bit synapse and neuron pairs with $L = 2$. The oneffset generator reads the neuron values, and produces a set of three oneffests per cycle. Each cycle, the control logic, which is shared and amortized across the entire column of PIPs, compares the oneffsets being processed, (1, 0, 4) in the first cycle of our example and picks the lowest, 0, indicated by a circle. This minimum oneffset controls the second stage shifter. The control subtracts this offset from all three oneffsets. The difference per oneffset, as long as it is less than 2^L controls the corresponding first level shifter. In the first cycle, the two shifters at the top are fed with values $1 - 0 = 1$ and $0 - 0 = 0$, while the shifter at the bottom is stalled given that it is not able to handle a shift by $4 - 0 = 4$. On cycle 2, the oneffsets are (6,7,4) and 4 is now the minimum, which controls the 2nd stage shifter, while (1, 3, 0) control the first-level shifters. On cycle 3, only the first and the third neurons still have oneffsets to process. The computation finishes in cycle 4 when the last oneffset of the third neuron controls the shifters.

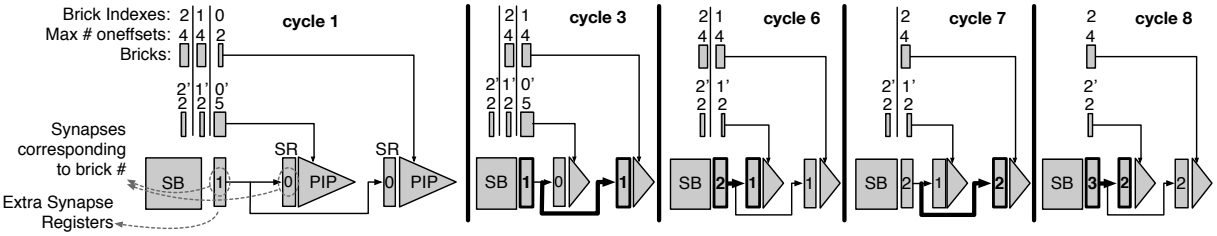


Fig. 8. Per-column synchronization example: one extra synapse register and 1x2 PIP array capable of processing two windows in parallel. The two numbers per brick show: the first from the top is the brick's index, $(0, 1, 2)$ and $(0', 1', 2')$ for the bricks of the first and second window. The second is the maximum count of oneffsets in its neurons, $(2, 4, 4)$ and $(5, 2, 2)$ respectively. The numbers in the registers indicate the index of the corresponding bricks, i.e., a synapse register containing a K stores the synapses corresponding to neurons bricks with indexes K and K'. In cycles 3 to 8, thicker lines indicate registers being loaded or wires being used.

E. Per-Column Neuron Lane Synchronization

The pallet neuron lane synchronization scheme of Section V-A4 is one of many possible synchronization schemes. Finer-grain neuron lane synchronization schemes are possible leading to higher performance albeit at a cost. This section presents *per column* neuron lane synchronization, an appealing scheme that, as Section VI-C shows, enhances performance at little additional cost.

Here each PIP column operates independently but all the PIPs along the same column wait for the neuron with the most essential bits before proceeding to the next neuron brick. Since the PIPs along the same column operate in sync, they all process one set of 16 synapse bricks which can be read using the existing SB interface. However, given that different PIP columns operate now out-of-sync, the SB would be read a higher number of times and become a bottleneck.

There are two concerns: 1) different PIP columns may need to perform two independent SB reads while there are only one SB port and one common bus connecting the PIP array to the SB, and 2) there will be repeat accesses to SB that will increase SB energy, while the SB is already a major contribution of energy consumption. These concerns are addressed as follows: 1) only one SB access can proceed per cycle thus a PIP column may need to wait when collisions occur. This way, we do not need an extra SB read port nor an extra set of 4K wires from the SB to the PIP array. 2) A set of SRAM registers, or *synapse set registers* (SSRs) are introduced in front of the SB each holding a recently read set of 16 synapse bricks. Since all PIP columns will eventually need the same set of synapse bricks, temporarily buffering them avoids fetching them repeatedly from the SB. Once a synapse set has been read into an SSR, it stays there until all PIP columns have copied it (a 4-bit down counter is sufficient for tracking how many PIP columns have yet to read the synapse set). This policy guarantees that the SB is accessed the same number of times as in *DaDN*. However, stalls may incur as a PIP column has to be able to store a new set of synapses into an SSR when it reads it from the SB. Figure 8 shows an example. Section VI-C evaluates this design.

Since each neuron lane advances independently, in the worst case, the dispatcher may need to fetch 16 independent neuron bricks each from a different pallet. The Dispatcher can buffer those pallets to avoid rereading NM, which would, at worst, require a 256 pallet buffer. However, given that the number SSRs restricts how far apart the PIP columns can be, and since Section VI-C shows that only one SSR is sufficient, a two pallet buffer in the dispatcher is all that is needed.

F. The Role of Software

PRA enables an additional dimension upon which hardware and software can attempt to further boost performance and energy efficiency, that of controlling the essential neuron value content. This work investigates a software guided approach where the precision requirements of each layer are used to zero out a number of prefix and suffix bits at the output of each layer. Using the profiling method of Judd *et al.,* [2], software communicates the precisions needed by each layer as meta-data. The hardware trims the output neurons before writing them to NM using AND gates and precision derived bit masks.

VI. Evaluation

The performance, area and energy efficiency of *Pragmatic* is compared against *DaDN* [3] and *Stripes* [4], two state-of-the-art DNN accelerators. *DaDN* is the fastest bit-parallel accelerator proposed to date that processes all neuron regardless of theirs values, and *STR* improves upon *DaDN* by exploiting the per layer precision requirements of DNNs. *Cnvlutin* improves upon *DaDN* by skipping most zero-valued neurons [11], however, *Stripes* has been shown to outperform it.

The rest of this section is organized as follows: Section VI-A presents the the experimental methodology. Sections VI-B and VI-C explore the *PRA* design space considering respectively single- and 2-stage shifting configurations, and column synchronization. Section VI-D reports energy efficiency for the best configuration. Section VI-E analyzes the contribution of the software provided precisions. Finally, Section VI-F reports performance for designs using an 8-bit quantized representation.

A. Methodology

All systems were modelled using the same methodology for consistency. A custom cycle-accurate simulator models execution time. Computation was scheduled such that all designs see the same reuse of synapses and thus the same SB read energy. To estimate power and area, all designs were synthesized with the Synopsis Design Compiler [15] for a TSMC 65nm library. The NBin and NBout SRAM buffers were modelled using CACTI [16]. The eDRAM area and energy were modelled with *Destiny* [17]. To compare against *STR*, the per layer numerical representation requirements reported in Table II were found using the methodology of Judd et al. [4]. All *PRA* configurations studied exploit software provided precisions as per Section V-F. Section VI-E analyzes the impact of this information on overall performance. All performance measurements are for the convolutional layers only which account for more than 92% of the overall execution time in *DaDN* [3]. *PRA* does not affect the execution time of the remaining layers.

B. Single- and 2-stage Shifting

This section evaluates the single-stage shifting *PRA* configuration of Sections V-A– V-B , and the 2-stage shifting variants of Section V-D. Section VI-B1 reports performance while Section VI-B2 reports area and power. In this section, All *PRA* systems use pallet synchronization.

Network	Per Layer Neuron Precision in Bits
AlexNet	9-8-5-5-7
NiN	8-8-8-9-7-8-8-9-9-8-8-8
GoogLeNet	10-8-10-9-8-10-9-8-9-10-7
VGG_M	7-7-7-8-7
VGG_S	7-8-9-7-9
VGG_19	12-12-12-11-12-10-11-11-13-12-13-13-13-13-13-13

TABLE II
Per layer neuron precision profiles.

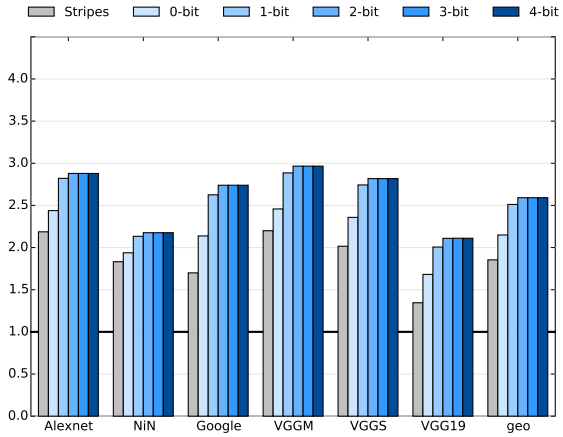


Fig. 9. *Pragmatic*'s performance relative to DaDianNao using 2-stage shifting and per-pallet synchronization.

1) Performance:: Figure 9 shows the performance of *STR* (leftmost bars) and of *PRA* variants relative to *DaDN*. The *PRA* systems are labelled with the number of bits used to operate the first-stage, synapse shifters, e.g., the synapse shifters of *"2-bit"* , or PRA_{2b}, are able to shift to four bit positions (0–3). "4-bit" or PRA_{4b}, is the single-stage *Pragmatic*, or PRA_{single} of Sections V-A– V-B whose synapse shifters can shift to 16 bit positions (0–15). It has no second stage shifter.

PRA_{single} improves performance by $2.59\times$ on average over *DaDN* compared to the $1.85\times$ average improvement with *STR*. Performance improvements over *DaDN* vary from $2.11\times$ for VGG19 to $2.97\times$ for VGGM. As expected the 2-stage *PRA* variants offer slightly lower performance than PRA_{single}, however, performance with PRA_{2b} and PRA_{3b} is always within 0.2% of PRA_{single}. Even PRA_{0b} which does not include any synapse shifters outperforms *STR* by 20% on average. Given a set of oneffsets, PRA_{0b} will accommodate the minimum non-zero oneffset per cycle via its second level shifter.

2) Area and Power:: Table III shows the absolute and relative to *DaDN* area and power. Two area measurements are reported: 1) for the unit excluding the SB, NBin and NBout memory blocks, and 2) for the whole chip comprising 16 units and all memory blocks. Since SB and NM dominate chip area,

	DDN	STR	0-bit	1-bit	2-bit	3-bit	4-bit
Area U.	1.55	3.05	3.11	3.16	3.54	4.41	5.75
Δ Area U.	1.00	1.97	2.01	2.04	2.29	2.85	3.71
Area T.	90	114	115	116	122	136	157
Δ Area T.	1.00	1.27	1.28	1.29	1.35	1.51	1.75
Power T.	18.8	30.2	31.4	34.5	38.2	43.8	51.6
Δ Power T.	1.00	1.60	1.67	1.83	2.03	2.33	2.74

TABLE III
AREA $[mm^2]$ AND POWER $[W]$ FOR THE UNIT AND THE WHOLE CHIP. PALLET SYNCHRONIZATION.

	DDN	STR	1-reg	4-reg	16-reg
Area U.	1.55	3.05	3.58	3.73	4.33
Δ Area U.	1.00	1.97	2.31	2.41	2.79
Area T.	90	114	122	125	134
Δ Area T.	1.00	1.27	1.36	1.39	1.49
Power T.	18.8	30.2	38.8	40.8	49.1
Δ Power T.	1.00	1.60	2.06	2.17	2.61

TABLE IV
AREA $[mm^2]$ AND POWER $[W]$ FOR THE UNIT AND THE WHOLE CHIP FOR COLUMN SYNCHRONIZATION AND PRA_{2b}.

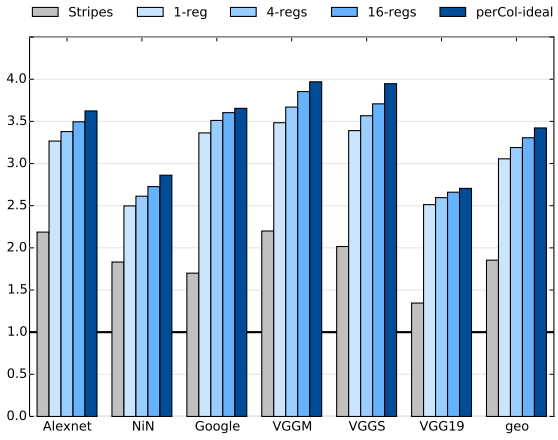


Fig. 10. Relative performance of PRA_{2b} with column synchronization and as a function of the SB registers used.

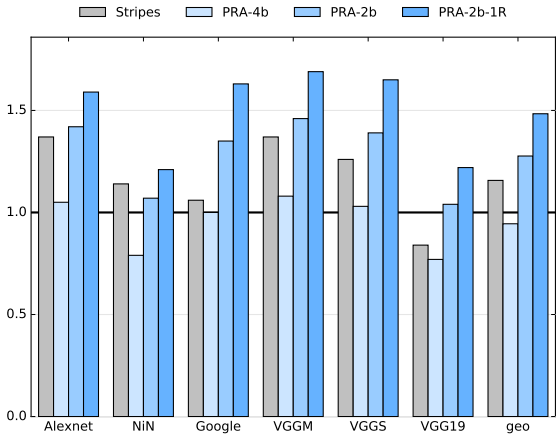


Fig. 11. Relative energy efficiency

the per area area overheads Given the performance advantage of *PRA*, the area and power overheads are justified. PRA_{2b} is particularly appealing as its overall area cost over $BASE$ is only $1.35\times$ and its power $2.03\times$ while its performance is $2.59\times$ on average. Accordingly, we restrict attention to this configuration in the rest of this evaluation.

C. Per-column synchronization

1) Performance:: Figure 10 reports the performance for PRA_{2b} with column synchronization and as a function of the number of SSRs as per Section V-E. of *Stripes* (first bar of each group) and *Pragmatic* (rest of the bars) relative to *DaDN*. Configuration PRA_{2b}^{xR} refers to a configuration using x SSRs. Even PRA_{2b}^{1R} boosts performance to $3.1\times$ on average close to the $3.45\times$ that is ideally possible with $PRA_{2b}^{\infty R}$.

2) Area and Power:: Table IV reports the area per unit, and the area and power per chip. PRA_{2b}^{1R} that offers most performance benefits increases chip area by only $1.35\times$ and power by only $2.19\times$ over *DaDN*.

D. Energy Efficiency

Figure 11 shows the energy efficiency of various configurations of *Pragmatic*. *Energy Efficiency*, or simply *efficiency* for a system NEW relative to BASE is defined as the ratio $E_{\text{BASE}}/E_{\text{NEW}}$ of the energy required by BASE to compute all of the convolution layers over that of NEW. For the selected networks, *STR* is 16% more efficient than *DaDN*. The power overhead of PRA_{single} (PRA_{4b}) is more than the speedup resulting in a circuit that is 5% less efficient than *DaDN*. PRA_{2b} reduces that power overhead while maintaining performance yielding an efficiency of 28%. PRA_{2b}^{1R} yields the best efficiency at 48% over *DaDN*.

E. The Impact of Software

All *PRA* configurations studied thus far, used software provided per layer precisions to reduce essential bit content. *PRA* does not require these precisions to operate. Table V shows what fraction of the performance benefits is due to the software guidance for PRA_{2b}^{1R}, the best configuration studied. The results demonstrate that: 1) *PRA* would outperform the other architectures even without software guidance, and 2) on average, software guidance improves performance by 19% which is on par with the estimate of Section II for ideal *PRA* (from 10% to 8%).

F. Quantization

Figure 12 reports performance for *DaDN* and *PRA* configurations using the 8-bit quantized representation used in Tensorflow [5], [18]. This quantization uses 8 bits to specify arbitrary minimum and maximum limits per layer for the neurons and the synapses separately, and maps the 256 available 8-bit

Alexnet	**NiN**	**Google**	**VGGM**	**VGGS**	**VGG19**	**AVG**
23%	10%	18%	22%	21%	19%	19%

TABLE V
PERFORMANCE BENEFIT DUE TO SOFTWARE GUIDANCE

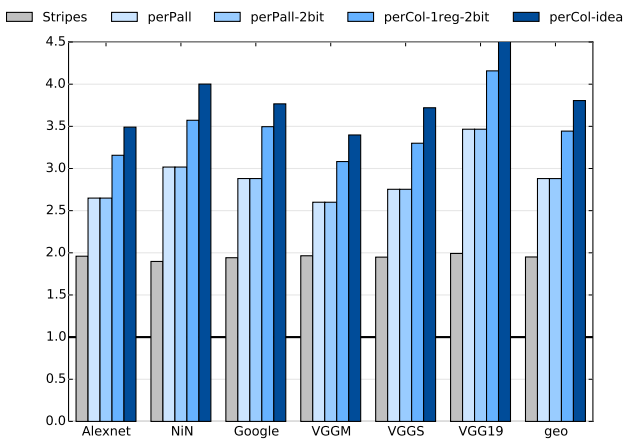


Fig. 12. Performance: 8-bit quantized representation.

values linearly into the resulting interval. This representation has higher flexibility and better utilization than the reduced precision approach of *Stripes* since the range doesnt have to be symmetrical and the limits dont have to be powers of two, while still allowing straightforward multiplication of the values. The limit values are set to the maximum and the minimum neuron values for each layer and the quantization uses the recommended rounding mode.

Figure 12 reports performance relative to *DaDN* for PRA_{single}, PRA_{2b}, PRA_{2b}^{1R}, and $PRA_{2b}^{\infty R}$. *PRA* performance benefits persist and are nearly $3.5\times$ for PRA_{2b}^{1R}. Measuring the area and energy of these designs is left for future work, however, the absolute area and energy needed by all will be lower due to the narrower representation. Moreover, given that the tile logic will occupy relatively less area for the whole chip and given that the SB and NM account for significant area and energy, the overall overheads of the *PRA* designs over *DaDN* will be lower than that measured for the 16-bit fixed-point configurations.

VII. Related Work

The acceleration of Deep Learning is an active area of research and has yielded numerous proposals for hardware acceleration. *DaDianNao* (*DaDN*) is the de facto standard for high-performance DNN acceleration [3]. In the interest of space, this section restricts attention to methods that are either directly related to *DaDN*, or that follow a value-based approach to DNN acceleration, as *Pragmatic* falls under this category of accelerators. Value-based accelerators exploit the properties of the values being processed to further improve performance or energy beyond what is possible by exploiting computation structure alone. Cnvlutin [11] and *Stripes* [4][19] are such accelerators and they have been already discussed and compared against in this work.

PuDianNao is a hardware accelerator that supports seven machine learning algorithms including DNNs [20]. *ShiDianNao* is a camera-integrated low power accelerator that exploits integration to reduce communication overheads and to further improve energy efficiency [21]. Cambricon is the first instruction set architecture for Deep Learning [22]. Minerva is a highly automated software and hardware co-design approach targeting ultra low-voltage, highly-efficient DNN accelerators [14]. Eyeriss is a low power, real-time DNN accelerator that exploits zero valued neurons for memory compression and energy reduction [13]. The Efficient Inference Engine (EIE) exploits efficient neuron and synapse representations and pruning to greatly reduce communication costs, to improve energy efficiency and to boost performance by avoiding certain ineffectual computations [10][23]. EIE targets fully-connected (FC) layers and was shown to be $12\times$ more efficient than *DaDN* on FC layers, and $2\times$ less efficient for convolutional layers. All aforementioned accelerators use bit-parallel units. While this work has demonstrated *Pragmatic* as a modification of *DaDN*, its computation units and potentially, its general approach could be compatible with all aforementioned accelerator designs. This investigation is interesting future work. As newer network architectures like GoogLeNet, NiN and VGG19 rely less on fully connected layers, this work used *DaDN* as an energy efficient and high performance baseline.

Profiling has been used to determine the precision requirements of a neural network for a hardwired implementation [24]. EoP has been exploited in general purpose hardware and other application domains. For example, Brooks *et al.* [25] exploit the prefix bits due to EoP to turn off parts of the datapath improving energy. Park *et al.* [26], use a similar approach to trade off image quality for improved energy efficiency. Neither approach directly improves performance.

VIII. Conclusion

To the best of our knowledge *Pragmatic* is the first DNN accelerator that exploits not only the per layer precision requirements of DNNs but also the essential bit information content of the neuron values. While this work targeted high-performance implementations, *Pragmatic*'s core approach should be applicable to other hardware accelerators.

References

[1] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Computers*, vol. 13, no. 1, pp. 14–17, 1964.

[2] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets, arXiv:1511.05236v4 [cs.LG] ," *arXiv.org*, 2015.

[3] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 609–622, Dec 2014.

[4] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-serial Deep Neural Network Computing ," *Computer Architecture Letters*, 2016.

[5] P. Warden, "Low-precision matrix multiplication."

[6] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[7] A. Y. Hannun, C. Case, J. Casper, B. C. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep speech: Scaling up end-to-end speech recognition," *CoRR*, vol. abs/1412.5567, 2014.

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[9] P. Ienne and M. A. Viredaz, "Bit-serial multipliers and squarers," *IEEE Transactions on Computers*, vol. 43, no. 12, pp. 1445–1450, 1994.

[10] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *arXiv:1602.01528 [cs]*, Feb. 2016. arXiv: 1602.01528.

[11] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 IEEE/ACM International Conference on Computer Architecture (ISCA)*, 2016.

[12] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814, 2010.

[13] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pp. 262–263, 2016.

[14] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernndez-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *International Symposium on Computer Architecture*, 2016.

[15] Synopsys, "Design Compiler." http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages.

[16] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."

[17] M. Poremba, S. Mittal, D. Li, J. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, pp. 1543–1546, March 2015.

[18] Google, "Low-precision matrix multiplication." https://github.com/google/gemmlowp, 2016.

[19] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, and A. Moshovos, "Stripes: Bit-serial Deep Neural Network Computing ," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, 2016.

[20] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "PuDianNao: A Polyvalent Machine Learning Accelerator," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 369–381, ACM, 2015. PuDianNao.

[21] 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 *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 92–104, June 2015. ShiDianNao.

[22] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *2016 IEEE/ACM International Conference on Computer Architecture (ISCA)*, 2016.

[23] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149 [cs]*, Oct. 2015. arXiv: 1510.00149.

[24] J. Kim, K. Hwang, and W. Sung, "X1000 real-time phoneme recognition VLSI using feed-forward deep neural networks," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7510–7514, May 2014.

[25] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, (Washington, DC, USA), pp. 13–, IEEE Computer Society, 1999.

[26] J. Park, J. H. Choi, and K. Roy, "Dynamic Bit-Width Adaptation in DCT: An Approach to Trade Off Image Quality and Computation Energy," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, pp. 787–793, May 2010.