Bit-Pragmatic Deep Neural Network Computing

Jorge Albericio[*]
NVIDIA
jalbericiola@nvidia.com

Alberto Delmás
University of Toronto
delmasl1@ece.utoronto.ca

Patrick Judd
University of Toronto
juddpatr@ece.utoronto.ca

Sayeh Sharify
University of Toronto
sayeh@ece.utoronto.ca

Gerard O'Leary
University of Toronto
gerard.oleary@eecg.utoronto.ca

Roman Genov
University of Toronto
roman@eecg.utoronto.ca

Andreas Moshovos
University of Toronto
moshovos@ece.utoronto.ca

ABSTRACT

Deep Neural Networks expose a high degree of parallelism, making them amenable to highly data parallel architectures. However, data-parallel architectures often accept inefficiency in individual computations for the sake of overall efficiency. We show that on average, activation values of convolutional layers during inference in modern Deep Convolutional Neural Networks (CNNs) contain 92% zero bits. Processing these zero bits entails ineffectual computations that could be skipped. We propose *Pragmatic* (*PRA*), a massively data-parallel architecture that eliminates most of the ineffectual computations on-the-fly, improving performance and energy efficiency compared to state-of-the-art high-performance accelerators [5]. The idea behind *PRA* is deceptively simple: use serial-parallel shift-and-add multiplication while skipping the zero bits of the serial input. However, a straightforward implementation based on shift-and-add multiplication yields unacceptable area, power and memory access overheads compared to a conventional bit-parallel design. *PRA* incorporates a set of design decisions to yield a practical, area and energy efficient design.

Measurements demonstrate that for convolutional layers, *PRA* is 4.31× faster than *DaDianNao* [5] (*DaDN*) using a 16-bit fixed-point representation. While *PRA* requires 1.68× more area than *DaDN*, the performance gains yield a 1.70× increase in energy efficiency in a 65nm technology. With 8-bit quantized activations, *PRA* is 2.25× faster and 1.31× more energy efficient than an 8-bit version of *DaDN*.

[*]This work was developed while Jorge Albericio was a postdoctoral fellow at the Univ. of Toronto.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-50, October 14–18, 2017, Cambridge, MA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123982

CCS CONCEPTS

• **Computing methodologies → Machine learning**; **Neural networks**; • **Computer systems organization → Single instruction, multiple data**; • **Hardware → Arithmetic and datapath circuits**;

KEYWORDS

Hardware Accelerators, Machine Learning, Neural Networks

ACM Reference format:
Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-Pragmatic Deep Neural Network Computing. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123982

1 INTRODUCTION

Deep neural networks (DNNs) have become the state-of-the-art technique in many classification tasks such as object [9] and speech recognition [14]. Given their breadth of application and high computational demand, DNNs are an attractive target for fixed-function accelerators [5, 6, 12]. With power consumption limiting modern high-performance designs, achieving better energy efficiency is essential [8].

DNNs comprise a pipeline of *layers* that primarily compute inner products of *activation* and *weight* vectors. A typical layer performs hundreds of inner products, each accepting thousands of activation and weight pairs. DNN hardware typically uses either 16-bit fixed-point [5] or quantized 8-bit numbers [29] and bit-parallel compute units.

As Section 2 shows, in Deep Convolutional Neural Networks (CNNs) for image classification, no more than 13% of all activation bits are non-zero when represented using a 16-bit fixed-point format. Traditional bit-parallel compute units process all input bits regardless of their value. Recent work on CNN accelerators has proposed how to leverage sparsity for performance. Recent accelerators exploit this characteristic by skipping zero, or near zero valued activations or ineffectual weights [2, 12, 24]. Stripes exploits the non-essential bits of an activation by using reduced precision and processing activations bit-serially [18]. However, the reduced precision format is determined *statically* and shared across all concurrent activations, and thus does not exploit the additional zero bits that appear dynamically.

This work presents *Pragmatic* (*PRA*), a CNN accelerator whose goal is to exploit all zero bits of the input activation and to process only the *essential* (non-zero) bits. The idea behind *Pragmatic* is to process the activations bit-serially, while compensating for the loss in computation bandwidth by exploiting the abundant parallelism of convolutional layers, which represent 92% of the processing time of modern CNNs [5]. However, as Section 3 explains, a straightforward implementation of a zero-bit-skipping processing engine proves impractical as it suffers from unacceptable energy, area, and data access overheads. Specifically, Section 6.2 shows such an implementation requires 1.94× the area and 3.37× the power of the baseline chip, while only increasing performance by 2.59×.

To improve performance and energy efficiency over a state-of-the-art bit-parallel accelerator without a disproportionate increase in area and data access demands, *PRA* combines the following techniques: 1) on-the-fly conversion of activations from a storage representation into an explicit representation of the essential bits only, 2) bit-serial activation/bit-parallel weight processing, 3) judicious SIMD (single instruction multiple data) lane synchronization that i) maintains wide memory accesses, ii) avoids fragmenting and enlarging the on-chip weight and activation memories, and iii) avoids a disproportionate area and complexity overhead, 4) computation re-arrangement to reduce datapath area, and 5) optimized encoding to further reduce the number of 1 bits and to increase performance. Pragmatic combines these techniques and balances their trade-offs to produce a design with high performance and energy efficiency.

We simulate *Pragmatic* with modern CNNs for image classification to measure performance and produce a circuit layout with a 65nm technology to estimate power, performance and area. We present multiple design points with different synchronization constraints, flexibility, and data encoding to trade performance for lower area and power. The most power efficient design with optimized encoding yields 4.31× speedup on average over the *DaDN* accelerator. *Pragmatic*'s average energy efficiency is 1.70× over *DaDN* and its area overhead is 1.68× .

The rest of the paper is organized as follows: Section 2 motivates *Pragmatic* in the context of prior work. Section 3 describes *Pragmatic*'s approach with a simplified example. Section 4 provides the necessary background. Section 5 describes *Pragmatic* in detail. Section 6 presents our methodology and results. Section 7 describes related work and Section 8 concludes.

2 MOTIVATION

Binary multiplication can be broken down into a summation of single bit multiplications (ANDs). For example, $a \times w$ can be calculated as $\sum_{i=0}^{p} a_i \cdot (w \ll i)$, where a_i is the i-*th* bit of a. The multiplier computes p *terms*, each a product of the shifted operand w, and a bit of a, and adds them to produce the final result. The terms and their sum can be calculated concurrently to reduce latency [28], or over multiple cycles to reuse hardware and reduce area [30]. In either case, zero bits in a result in ineffectual computation.

We will categorize these zero bits as either statically or dynamically ineffectual. *Statically* ineffectual bits are those that can be determined to be ineffectual *a priori*. They result from using a data format with more precision than is necessary. In this case, 1's may also be statically ineffectual. Hardware typically uses fixed bit widths for generality resulting in such ineffectual bits. In contrast, *dynamically* ineffectual bits are those that cannot be known in advance.

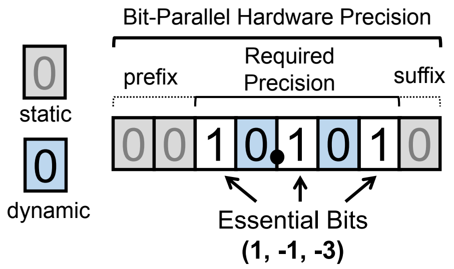

Figure 1: Sources of ineffectual computation with conventional positional representation and fixed-length hardware precision.

	Alex	NiN	Goog	vggM	vggS	vgg19	Geom
16-bit Fixed-Point							
All	7.8%	10%	6.4%	5.1%	5.7%	13%	7.6%
NZ	18%	22%	19%	17%	17%	24%	19%
8-bit Quantized							
All	31%	27%	27%	38%	34%	17%	28%
NZ	44%	37%	43%	47%	46%	29%	41%

Table 1: Average fraction of non-zero bits per activation for two fixed-length representations: 16-bit fixed-point, and 8-bit quantized. All: over all activations. NZ: over non-zero activations only.

Figure 1 shows an example illustrating these categories of ineffectual bits using an 8-bit unsigned fixed-point number with 4 fractional and 4 integer bits. Assume that is known ahead of time that the data only needs 5 bits, then there are 3 statically ineffectual bits as a prefix and suffix to the required precision. While $10.101_{(2)}$ requires just five bits, two dynamically generated zero bits appear at positions 0 and -2. In total, five ineffectual bits will be processed generating five ineffectual terms.

The non-zero bits can instead be encoded with their corresponding exponents (1,-1,-3). While such a representation may require more bits and thus be undesirable for storage, dynamically generating them and only computing the non-zero terms may benefit performance and energy efficiency. This work shows that with the multitude of zero bits and data reuse in CNNs, there is significant potential to improve performance while reusing hardware to save area.

The rest of this section motivates *Pragmatic* by: 1) measuring the fraction of non-zero bits in the activation stream of modern CNNs, and 2) estimating the performance improvement which may be gained by processing only the non-zero activation bits.

2.1 Essential Activation Bit Content

16-bit fixed-point is commonly used for DNN hardware implementations [5, 6]. Recent work has shown that fewer bits are necessary for

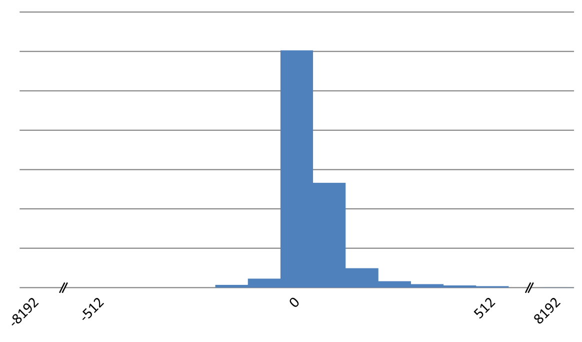

Figure 2: Average distribution of activations for the networks studied in 16 bit fixed-point (14 integer and 2 fractional bits). Center bin is just zero valued activations.

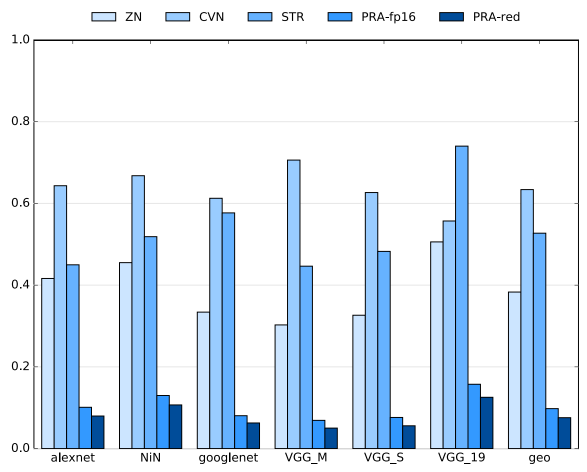

Figure 3: Convolutional layer computational demands with a 16-bit fixed-point baseline representation. Lower is better.

CNNs [17] and how to exploit this reduced precision requirements to save bandwidth [16] and improve performance [18]. Recently, it has also been shown that 8 bits of precision are sufficient when linear quantization is used for many CNNs [1, 29]. These techniques exploit statically ineffectual bits.

Table 1 reports the *non-zero bit content* of **all** activations of modern CNNs for two commonly used representations: 16-bit fixed-point and 8-bit quantized activations [29]. Figure 2 shows the distribution of activation values. From these measurements we can infer that the large percentage of zero bits are due to two factors: 1) Activations have a normal distribution of positive values. Most activations are close to zero, relative to the range of the representation, thus their most significant bits are zero. 2) The rectified linear (ReLU) activation function which is used in most modern CNNs converts negative activations to 0, resulting in many zero activations and no negative activations, except in the inputs to the first layer. Weights exhibit the first property, but not the second, and exploiting their bit content is left for future work.

Skipping the computation of zero valued activations is an optimization employed in recently proposed DNN accelerators, both to save power [6, 26] and processing time [2, 12, 24]. These works exploit the presence of dynamic zero bits only for zero valued or near-zero valued activations or weights [24]. Table 1 also shows the percentage of non-zero bits in the non-zero activations only (**NZ**). Zero bits still make up over 50% of the non-zero activation values in all networks. Therefore, there is still a significant opportunity to exploit the zero bit content of non-zero values.

In contrast to the techniques discussed, this work targets both statically and dynamically ineffectual bits in the activations. When considering all activations, the non-zero bit content is *at most* 13% and 38% for the fixed-point and 8-bit quantized representations respectively.

These results suggest that a significant number of ineffectual terms are processed with conventional fixed-length hardware. *Stripes* [18], tackles the statically ineffectual bits by computing arbitrary length fixed-point activations serially for improved performance. *Pragmatic*'s goal is to exploit both static and dynamic zero bits. As the next section will show, *Pragmatic* has the potential to greatly improve performance even when compared to *Stripes*.

2.2 Pragmatic's Potential

To estimate *PRA*'s potential, this section compares the number of terms that would be processed by various compute engines for the convolutional layers of our target CNNs (see Section 6.1) for the two aforementioned baseline two's complement representations.

The following compute engines are considered: 1) the baseline *DaDN* with its 16-bit fixed-point bit-parallel units [5], 2) **ZN**, an *ideal* engine that can skip *all* zero valued activations, 3) Cnvlutin (**CVN**) a practical design that can skip most zero value activations [2], 4) *Stripes* (**STR**) a practical design that uses reduced precisions (see Table 2) [19], 5) ***PRA*-fp16** an *ideal* engine that processes only the essential activation bits in the native 16-bit fixed-point representation, and 6) ***PRA*-red**, an *ideal* engine that processes only the essential activation bits of the *reduced precision* formats used in *STR*.

Figure 3 reports the number of terms processed, normalized over *DaDN*, where each multiplication is accounted for using an equivalent number of terms: 16 for *DaDN*, ZN, and CVN, p for a layer using a precision of p bits for *STR*, and the number of essential activation 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 activation 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 CNNs remaining above 87% with *PRA*-red.

In summary, this section showed that only 8% of the computation is strictly necessary for the target networks due to the presence of zero bits in the activations. However, even with reduced precision and dynamic zero value skipping many ineffectual computations remain. Therefore, a more aggressive approach is needed.

3 PRAGMATIC: A SIMPLIFIED EXAMPLE

This section illustrates the idea behind *Pragmatic* via a simplified example. For the purposes of this discussion, it suffices to know that a convolutional layer performs inner products where both weights and activations are reused. Section 4.1 describes the relevant computations in more detail.

The bit-parallel unit of Figure 4a multiplies two activations with their respective weights and via an adder reduces the two products. The unit reads *all* activation and weight bits, $(a_0 = 001_{(2)}, a_1 = 010_{(2)})$ and $(w_0 = 001_{(2)}, w_1 = 111_{(2)})$ respectively in a single cycle. As a result, the two sources of inefficiency manifest here: Bit 2 of a_0 and a_1 is statically zero, meaning only 2 bits are required. Even in 2 bits, each activation contains a zero bit that is dynamically generated. As a result, four ineffectual terms are processed when using standard multipliers.

The hybrid, bit-serial-activation/bit-parallel-weight unit in Figure 4b is representative of *STR* which tackles statically ineffectual bits. *STR* performs serial-parallel shift-and-add multiplication, with the shifters merged and placed after the adder tree since all serial inputs are synchronized. Each cycle, the unit processes one bit from each activation and hence it takes three cycles to compute the convolution when the activations 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 weight reuse and processes multiple activations groups in parallel. In this example, six activations $(a_0 = 001_{(2)}, a_1 = 010_{(2)}, a'_0 = 000_{(2)}, a'_1 = 010_{(2)}, a''_0 = 010_{(2)}, a''_1 = 000_{(2)})$ are combined with the two weights as shown. Starting from the least significant position, each cycle one bit per activation is *ANDed* with the corresponding weight. The six AND results feed into an adder tree and the result is accumulated after being shifted right by one bit. Since the specific activation values could be represented all using 2 bits, *STR* would need 2 cycles to process all six products, 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. *Pragmatic* uses shift and add multiplication, where one input is the number of bits to shift. Since the shift offset can be different for each multiplier, the shifter can not be shared as it was in *STR*. In this example, activations are no longer represented as vectors of bits but as vectors of offsets of the essential bits. For example, activation $a_0 = 001_{(2)}$ is represented as $on_0 = (0)$, and an activation value of $111_{(2)}$ would be represented as $(2, 1, 0)$. There is also a valid signal for each offset (not shown) to force the input to the tree to zero, since a shifter per activation uses the offsets to effectively multiply the corresponding weight 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. For this example, *PRA* would process six activation and weight pairs in a single cycle, a speedup of $3\times$ over the bit-parallel unit.

3.1 Key Challenges

Unfortunately, implementing *Pragmatic* as described in this section results in unacceptable overheads. Specifically: a) As Section 5 will explain, to guarantee that *Pragmatic* always matches and if possible exceeds the performance of *DaDN* it needs to process 16 times as many activations bit-serially. As Section 6.2 shows this straightforward implementation requires 1.94x the area and 3.15x the power of the baseline chip, while only increasing performance by 2.59x. This is due to: i) the weight shifters need to accommodate the worst case scenario where one of the activation powers is 0 and another is 15, and ii) consequently, the adder trees need to support 32 bit inputs instead of just 16 bit. b) As activations will have a different number of essential bits, each activation will advance at a different pace. In the worst, but most likely typical case, each activation will be out-of-sync. This will require 256 concurrent and independent narrow accesses for activations and/or $256\times$ the activation memory bandwidth. Similarly, there will be 256 concurrent and independent accesses for the corresponding weights and a $256\times$ increase in weight memory accesses. Finally, generating these memory references and tracking the progress of each lane will increase overall complexity.

To avoid these overheads *Pragmatic* incorporates several design decisions that result in a practical design that captures most of the potential for performance improvement while judiciously sacrificing some of that potential to maintain area, energy, and memory access overheads favourable.

4 BACKGROUND

This work presents *Pragmatic* as a modification of the *DaDianNao* accelerator. Accordingly, this section provides the necessary background information: Section 4.1 reviews the operation of convolutional layers, and Section 4.2 overviews *DaDN* and how it processes convolutional layers.

4.1 Convolutional Layer Computation

A convolutional layer processes and produces activation arrays, that is 3D arrays of real numbers. The layer applies N_n 3D filters in a sliding window fashion using a constant stride S to produce an output 3D array. The input array contains $N_x \times N_y \times N_i$ *activations*. Each of the N_n filters, contains $K_x \times K_y \times N_i$ *weights* which are also real numbers. The output activation array dimensions are $O_x \times O_y \times N_n$, that is its depth equals the filter count. The layer computes the inner product of a filter and a *window*, a filter-sized, or $K_x \times K_y \times N_i$ sub-array of the input activation array. The inner product is then passed through an *activation function*, such as ReLU, to produce an output activation. If $a(y, x, i)$ and $o(y, x, i)$ are respectively input and output activations, $w^n(x, y, i)$ are the weights of filter n and f is the activation function. The output activation at position (x', y', n) is given by:

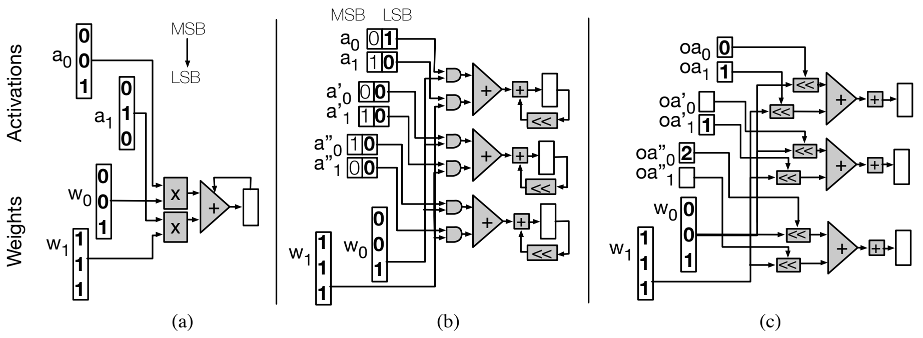

Figure 4: a) Bit-parallel unit. b) Bit-serial unit with equivalent throughput (*Stripes*[19]). c) *Pragmatic* unit with equivalent throughput where only essential information is processed.

$$\underbrace{o(y', x', n)}_{output\ activation} = f(\underbrace{\sum_{x=0}^{K_x-1}\sum_{y=0}^{K_y-1}\sum_{i=0}^{N_i-1} \underbrace{w^n(x, y, i)}_{weight} \times \underbrace{a(x + x' \times S, y + y' \times S, i)}_{input\ activation}}_{window})$$

The layer applies filters repeatedly over different windows positioned along the X and Y dimensions with stride S, and there is one output activation per window and filter. Accordingly, the output activation array dimensions are $O_x = (N_x - K_x)/S + 1$, $O_y = (N_y - K_y)/S + 1$, and $O_i = N_n$. Convolutions exhibit data reuse in three dimensions: 1) activations are reused for each filter, 2) weights are reused for each window, and 3) activations are reused due to overlap in the sliding window.

4.1.1 Terminology – Bricks and Pallets: For clarity, in what follows the term **brick** refers to a set of 16 elements of a 3D activation or weight 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 consecutive windows along the x or y dimensions with a stride of S, 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 activations per brick, and bricks per pallet are design parameters.

4.2 Baseline System: DaDianNao

Figure 5a shows a *DaDN* tile. Each *DaDN* chip comprises 16 tiles and a central 4MB eDRAM *Neuron Memory* (NM). Internally, each tile has: 1) a weight buffer (SB) that provides 256 weights per cycle one per weight lane, [1] 2) an input activation buffer (NBin) which provides 16 activations per cycle through 16 activation lanes, and 3) an output activation buffer (NBout) which accepts 16 partial output activations per cycle. The compute pipeline consists of 16 Inner Product units (**IP**), where each IP computes an inner product with 16 parallel multipliers and an adder tree to computes the inner product of a brick of weights and a brick of activation each cycle. A brick of input activations is broadcast from NBin to each IP, while each IP processes a different brick of weights from different filter lanes of SB. In total 256 weights, and 16 activations are consumed each cycle. Each IP accumulates partial sums of brick inner products to compute large inner products over multiple cycles. Once the large inner product is computed, the are passed through an activation function, f, to produce an output activation. The 16 IPs together produce a brick of output activations. Accordingly, each cycle, the whole chip processes 16 activations and $256 \times 16 = 4K$ weights producing $16 \times 16 = 256$ partial sums.

DaDN's main design goal is minimizing off-chip bandwidth while maximizing on-chip compute utilization. To avoid fetching weights from off-chip, *DaDN* uses a 2MB eDRAM SB per tile for a total of 32MB eDRAM. All inter-layer activations are stored in 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 weights once per layer, and for writing the final output.

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

5 PRAGMATIC

This section presents the *Pragmatic* architecture. Section 5.1 describes *PRA*'s processing approach while Section 5.2 describes its organization. Section 5.3 describes the additional circuitry needed. Sections 5.4 and 5.5 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 4.2 in the worst case.

5.1 Approach

PRA's goal is to process only the essential bits of the input activations. To do so *PRA* a) converts, on-the-fly, the input activation representation into one that contains only the essential bits, and b) processes one essential bit per activation and a full 16-bit weight per cycle. Since *PRA* processes activation bits serially, it may take up to 16 cycles to produce a product of an activation and a weight. To always match or exceed the performance of the bit-parallel units

[1]Chen *et al.*, refer to activations as neurons and weights as synapses [5]. We maintain the original acronyms for the various storage structures but use the more commonly used activation and weight terms in our discussion.

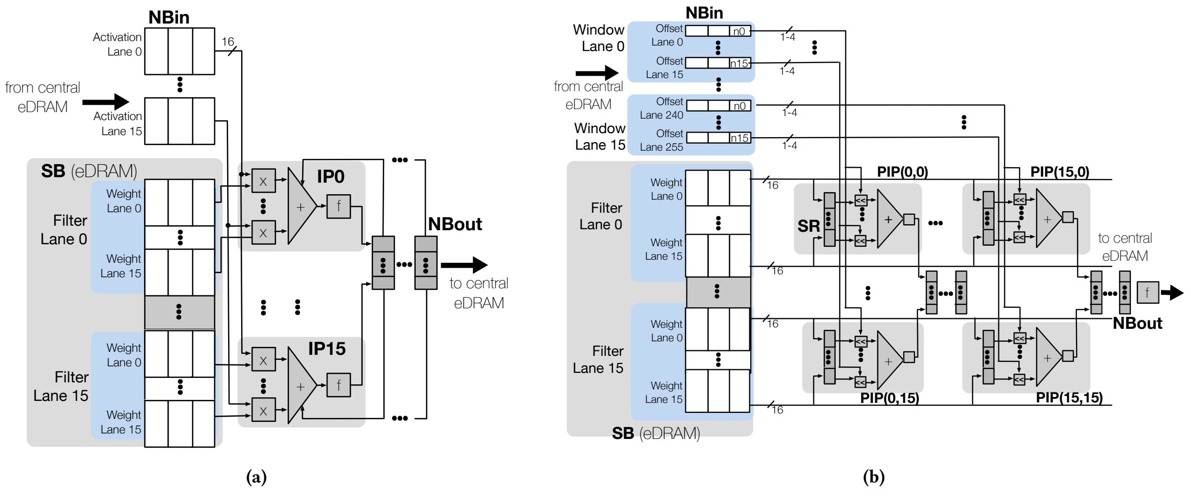

Figure 5: a) DaDianNao Tile. b) Pragmatic Tile.

of *DaDN*, *PRA* processes more activations concurrently exploiting the abundant parallelism of the convolutional layers. The remaining of this section describes in turn: 1) an appropriate activation 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 activations from NM.

5.1.1 Input activation Representation. *PRA* starts with an input activation 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, an activation $a = 5.5_{(10)} = 0101.1_{(2)}$ would be represented as $a = (2, 0, -1)$. In the implementation described herein, activations 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 activation 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 an activation. For example, $a = 101_{(2)}$ is represented as $a^{PRA} = ((0010, 0)(0000, 1))$. In the worst case, all bits of an input activation would be 1 and hence its *PRA* representation would contain 16 oneffsets.

5.1.2 Calculating a Term. *PRA* calculates the product of weight w and activation a as:

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

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

5.1.3 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 activation and weight 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 activations contain the same number of essential bits, so that when processing multiple activations in parallel, all units complete at the same time and thus can proceed with the next set of activations in sync. The next section will relax this constraint.

Since *PRA* processes activations bits serially, it produces one term per activation bit and weight pair and thus needs to process $4K$ such pairs concurrently. The choice of which $4K$ activation bits and weight 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 of the NM while keeping the units utilized as in *DaDN*. Specifically, a *PRA* tile can read 16 weight bricks and the equivalent of 256 activation bits as *DaDN*'s tiles do (*DaDN* processes 16 16-bit activations or 256 activation bits per cycle): As in *DaDN*, each *PRA* tile processes 16 weight bricks concurrently, one per filter. However, differently than *DaDN* where the 16 weight bricks are combined with just one activation brick which is processed bit-parallel, *PRA* combines each weight brick with 16 activation bricks, one from each of 16 windows, which are processed bit-serially. The same 16 activation bricks are combined with all weight bricks. These activation bricks form a *pallet* enabling the same weight brick to be combined with all. For example, in a single cycle a *PRA* tile processing filters 0 through 15 could combine $w^0_B(x, y, 0), ..., w^{15}_B(x, y, 0)$ with $a^{PRA}_B(x, y, 0), a^{PRA}_B(x + 2, y, 0), ...a^{PRA}_B(x + 31, y, 0)$ assuming a layer with a stride of 2. In this case, $w^4(x, y, 2)$ would be paired with $a^{PRA}(x, y, 2), a^{PRA}(x + 2, y, 2), ..., a^{PRA}(x + 31, y, 2)$ to produce the output activations $on(x, y, 4)$ through $on(x + 15, y, 4)$.

As the example illustrates, this approach allows each weight to be combined with one activation per window whereas in *DaDN* each weight is combined with one activation only. In total, 256 essential activation bits are processed per cycle and given that there are 256

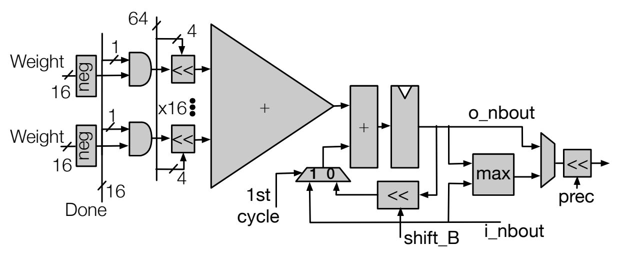

Figure 6: Pragmatic Inner Product Unit.

weights and 16 windows, *PRA* processes $256 \times 16 = 4K$ activation bit and weight pairs, or terms per cycle producing 256 partial output activations, 16 per filter, or 16 partial output activation bricks per cycle.

5.1.4 Supplying the Input Activation and Weight Bricks. Thus far it was assumed that all input activations have the same number of essential bits. Under this assumption, all activation lanes complete processing their terms at the same time, allowing *PRA* to move on to the next activation pallet and the next set of weight bricks in one step. This allows *PRA* to reuse *STR*'s approach for fetching the next pallet from the single-ported NM [19]. Briefly, with unit stride the 256 activations 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 activations 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 is improbable that all activations will have the same number of essential bits. In general, each activation lane if left unrestricted will advance at a different rate. In the worst case, each activation lane may end up needing activations from a different activation brick, thus breaking *PRA*'s ability to reuse the same weight brick. This is undesirable if not impractical as it would require partitioning and replicating the SB so that 4K unrelated weights could be read per cycle, and it would also increase NM complexity and bandwidth.

Fortunately, these complexities can be avoided with *pallet-level activation lane synchronization* where all activation lanes "wait" (an activation lane that has detected the end of its activation 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 activation, only how many exist. Since, it is unlikely that most pallets will contain an activation with 16 essential terms, *PRA* will improve performance over *DaDN*. Section 6.2 will show that in practice, this approach improves performance over *DaDN* and *STR*. Section 5.5 will discuss finer-grain synchronization schemes that lead to even better performance. Before doing so, however, the intervening sections detail *PRA*'s design.

5.2 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 an activation oneffset from the i-th window and its corresponding weight from the j-th filter. Specifically, all the PIPs along the i-th row receive the same weight brick belonging to the i-th filter and all PIPs along the j-th column receive an oneffset from each activation from one activation brick belonging to the j-th window.

The necessary activation oneffsets are read from NBin where they have been placed by the Dispatcher and the Oneffset generators units as Section 5.3 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 activations of a single window. When the tile starts to process a new activation pallet, 256 weights are read from SB through its 256 weight lanes as in *DaDN* and are stored in the weight registers (SR) of each PIP. The weight and oneffsets are then processed by the PIPs as the next section describes.

5.2.1 Pragmatic Inner-Product Unit. Figure 6 shows the PIP internals. Every cycle, 16 weights are combined with their corresponding oneffsets. Each oneffsets controls a shifter effectively multiplying the weight with a power of two. The shifted weights are reduced via the adder tree. An AND gate per weight supports the injection of null terms when necessary. In the most straightforward design, the oneffsets use 4-bits, each shifter accepts a 16-bit weight and can shift it by up to 15 bit positions producing a 31-bit output. Finally, the adder tree accepts 31-bit inputs. Section 5.4 presents an enhanced design that requires narrower components improving area and energy.

5.3 Dispatcher and Oneffset Generators

The *Dispatcher* reads 16 activation bricks from NM, as expected by the *PRA* tiles. The *oneffset generator* converts their activations on-the-fly to the oneffset representation, and broadcasts one oneffset per activation per cycle for a total of 256 oneffsets to all tiles. Fetching and assembling the 16 activation bricks from NM is akin to fetching words with a stride of S from a cache structure. As Section 5.1.4 discussed this can take multiple cycles depending on the stride and alignment of the initial activation brick. *PRA* uses the same dispatcher design as *STR* [19].

Once the 16 activation bricks have been collected, 256 oneffset generators operate in parallel to locate and communicate the next oneffset per activation. 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.

5.3.1 Local Oneffset Generation. Oneffset generation converts a single bit to 5 bits: a 4 bit offset and a bit to indicate the last offset. Doing this at the dispatcher requires a 5× increase in broadcast bandwidth and NBin capacity to match the worst case activation capacity. Instead, this work opts for a configuration which generates oneffsets at each tile, between NBin and the PIP array, to maintain the baseline NBin capacity. Broadcast bandwidth is still

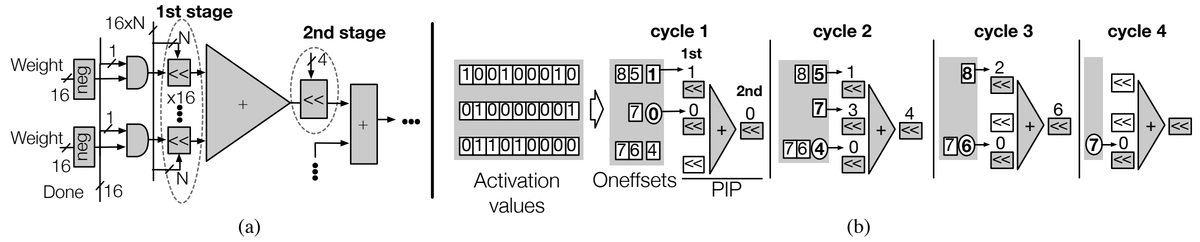

Figure 7: 2-stage shifting. a) Modified PIP. **b)** Example: Processing three 9-bit weight and activation pairs with $L = 2$. The oneffset generator reads the activation values, and produces a set of three oneffests per cycle. Each cycle, the control logic, which is shared and amortized across the entire 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 activations still have oneffsets to process. The computation finishes in cycle 4 when the last oneffset of the third activation controls the shifters.

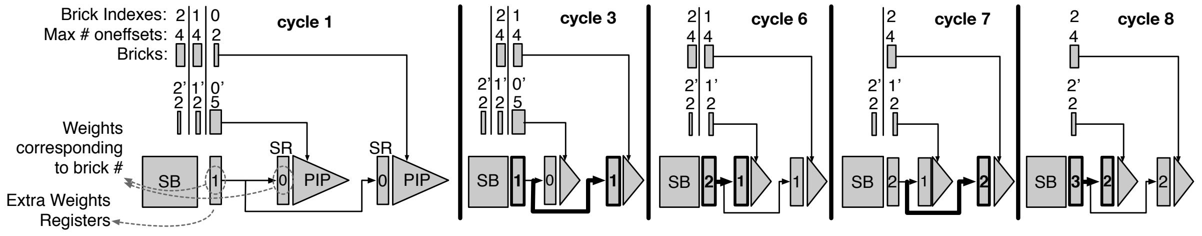

Figure 8: Per-column synchronization example: one extra weight register and 1x2 PIP array capable of processing two windows in parallel. The two numbers per brick show: the first from the top is the brick's index, (0, 1, 2) and $(0', 1', 2')$ for the bricks of the first and second window. The second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

increased to support the higher compute throughput. This is mitigated by transmitting activations serially in reduced precision, as in *Stripes* [18].

5.4 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 weights 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 weight specific offsets K_i', and the second stage, the common across all weights offset C:

$$\underbrace{\sum_i^T (S_i \ll K_i)}_{1-stage\ shifting} = \sum_i^T (S_i \ll (K_i' + C)) = (\underbrace{\sum_i^T (S_i \ll K_i'))}_{1^{st}\ stage} \underbrace{\ll C}_{2^{nd}\ stage}$$

This arrangement can be used to reduce the width of the weight 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 weight shifters. Meaning the design can process oneffsets which differ by less than 2^L in a single cycle. This reduces the size of the weight shifters and reduces the size of the adder tree to support terms of $16 + 2^L - 1$ bits only. As Section 6.2 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 6.2 studies the impact of L on cost and performance.

5.5 Per-Column Activation Lane Synchronization

The pallet activation lane synchronization scheme of Section 5.1.4 is one of many possible synchronization schemes. Finer-grain activation lane synchronization schemes are possible leading to higher performance albeit at a cost. This section presents *per column* activation lane synchronization, an appealing scheme that, as Section 6.3 shows, enhances performance at little additional cost.

Here each PIP column operates independently but all the PIPs along the same column wait for the activation with the most essential bits before proceeding to the next activation brick. Since the PIPs along the same column operate in sync, they all process one set of 16 weight bricks which can be read using the existing SB interface. However, given that different PIP columns operate now out-of-sync, the same weight bricks would need to be read multiple times, leading to the following challenges: 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 contributor of energy consumption.

Pragmatic address these challenges as follows: 1) only one SB access can proceed per cycle thus a PIP column may need to wait when collisions occur. This way, *PRA* does 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 *weight set registers* (WSRs) are introduced in front of the SB each holding a recently read set of 16 weight bricks. Since all PIP columns will eventually need the same set of weight bricks, temporarily buffering them avoids fetching them repeatedly from the SB reducing energy costs. Once a weight set has been read into an WSR, 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 weight set). This policy guarantees that the SB is accessed the same number of times as in *DaDN*. However, stalls may happen as a PIP column has to be able to store a new set of weights into an WSR when it reads it from the SB. Figure 8 shows an example. Section 6.3 evaluates this design.

Since each activation lane advances independently, in the worst case, the dispatcher may need to fetch 16 independent activation bricks each from a different pallet. The Dispatcher can buffer those pallets to avoid repeated NM accesses, which would, at worst, require a 256 pallet buffer. However, given that the number WSRs restricts how far apart the PIP columns can be, and since Section 6.3 shows that only one WSR is sufficient, a two pallet buffer in the dispatcher is all that is needed.

5.6 Improved Oneffset Encoding

Since PIPs in *Pragmatic* can negate any input term, it is possible to enhance the oneffset generator to generate fewer oneffsets for activation values containing runs of ones by allowing signed oneffsets [4].

This improved generator reduces runs of adjacent oneffsets $a...b$ into pairs of the form $a+1, -b$. Single oneffsets or gaps inside runs are represented by a positive or negative oneffset, respectively. For example an activation value of 11011 that would normally be encoded with oneffsets $(4, 3, 1, 0)$ can instead be represented with $(5, -3, +2, -0)$ or even more economically with $(5, -2, -0)$. This is equivalent to a Radix-4 Booth encoding and will never emit more than $\lfloor \frac{x}{2} + 1 \rfloor$ oneffsets, where x is the activation precision.

This encoding will never produce more oneffsets compared to the baseline encoding. However, this encoding may increase the number of cycles needed when the oneffset distribution among the bit groups being processed together due to the 2-stage shifting technique.

5.7 Reduced Precision Profiling

PRA enables an additional dimension upon which hardware and software can attempt to further boost performance and energy efficiency, that of controlling the essential activation value content. This work investigates a software guided approach where the

Network	Per Layer activation 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 2: Per layer activation precision profiles.

precision requirements of each layer are used to zero out the statically ineffectual bits at the output of each layer. Using the profiling method of Judd *et al.,* [17], software communicates the precisions needed by each layer as meta-data. The hardware trims the output activations before writing them to NM using AND gates and precision derived bit masks.

6 EVALUATION

The performance, area, and energy efficiency of *Pragmatic* is compared against *DaDN* [5] and *Stripes* [19]. *DaDN* is a bit-parallel accelerator that processes all activation regardless of their values and the *de facto* standard for reporting the relative performance of DNN accelerators. *STR* improves upon *DaDN* by exploiting the per layer precision requirements of DNNs.

The rest of this section is organized as follows: Section 6.1 presents the the experimental methodology. Sections 6.2 and 6.3 explore the *PRA* design space considering respectively single- and 2-stage shifting configurations, and column synchronization. Section 6.4 evaluates the benefit of improved offset encoding. Section 6.5 reports energy efficiency for the best configuration. Finally, Section 6.7 reports performance for designs using an 8-bit quantized representation.

6.1 Methodology

All systems were modelled using the same methodology for consistency. A custom cycle-level simulator models execution time. Computation was scheduled such that all designs see the same reuse of weights and thus the same SB read energy. To estimate power and area, all tile pipeline designs were synthesized with the Synopsys Design Compiler [27] for a TSMC 65nm library and laid out with Cadence Encounter. Circuit activity was captured with Mentor Graphics ModelSim and fed into Encounter for power estimation. The NBin and NBout SRAM buffers were modelled using CACTI [22]. The eDRAM area and energy were modelled with *Destiny* [25]. To compare against *STR*, the per layer numerical representation requirements reported in Table 2 were found using the methodology of Judd et al. [19]. All performance measurements are for the convolutional layers only which account for more than 92% of the computation in the networks we consider.

6.2 Single- and 2-Stage Shifting

This section evaluates the single-stage shifting *PRA* configuration of Sections 5.1 and 5.2 , and the 2-stage shifting variants of Section 5.4. Section 6.2.1 reports performance while Section 6.2.2 reports area

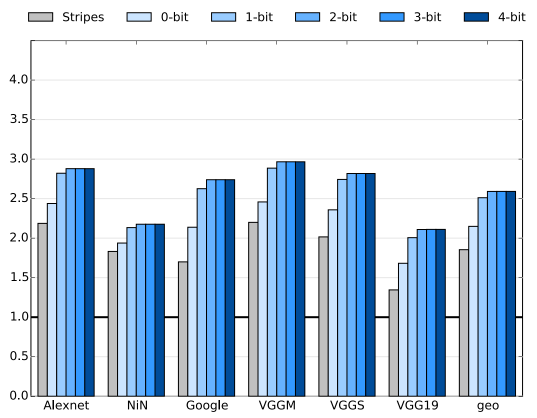

Figure 9: 2-stage shifting and per-pallet synchronization: *Pragmatic* performance relative to *DaDianNao*.

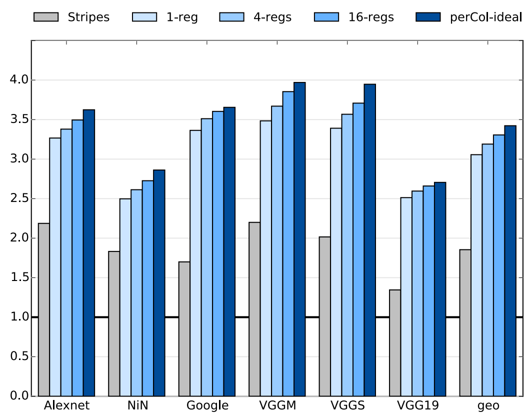

Figure 10: Column Synchronization: Relative performance of PRA_{2b} as a function of the SB registers used.

	DaDN	*STR*	0-bit	1-bit	2-bit	3-bit	4-bit
Unit Area	0.94	2.08	2.41	3.38	4.20	4.43	5.50
Total Area	78	96	36	52	130	134	151
Unit Pwr	424	762	887	1477	1782	2243	2736
Total Pwr	17.6	24.5	29.8	39.2	44.1	51.5	59.3

Table 3: Area $[mm^2]$ and power $[W]$ for the unit and the whole chip. Pallet synchronization.

	DaDN	*STR*	1-reg	4-reg	16-reg
Unit Area	0.94	2.08	4.24	4.39	4.99
Total Area	77.5	96.1	131	133	143
Unit Power	0.42	0.76	1.82	1.95	2.47
Total Power	17.6	24.5	44.7	46.7	55.0

Table 4: Area $[mm^2]$ and power $[W]$ for the unit and the whole chip for column synchronization and PRA_{2b}.

	Area	%	Power	%
NM	7.13	5.46%	1.61	3.61%
SB	48.11	36.83%	10.22	22.87%
SRAM	7.32	5.60%	0.79	1.78%
Interconnect	-	-	2.69	6.03%
PIP Array	62.65	47.96%	27.05	60.54%
Activation Unit	2.29	1.75%	1.49	3.33%
Offset Gen	2.91	2.23%	0.57	1.28%
Dispatcher	0.21	0.16%	0.25	0.56%
Total	**130.62**	**100.00%**	**44.67**	**100.00%**

Table 5: PRA^{1R}_{2b}: Area $[mm^2]$ and power $[W]$ breakdown.

and power. In this section, All *PRA* systems use pallet synchronization.

6.2.1 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, weight shifters, e.g., the weight 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 5.1 and 5.2 whose weight shifters can shift to 16 bit positions (0–15). It has no second stage shifter.

PRA_{single} improves performance by 2.59× on average over *DaDN* compared to the 1.85× average improvement with *STR*. Performance improvements over *DaDN* vary from 2.11× for VGG19 to 2.97× for VGGM. The 2-stage *PRA* variants offer slightly lower performance than PRA_{single}, however, performance with PRA_{2b} and PRA_{3b} is always within 0.01% of PRA_{single}. Even PRA_{0b} which does not include any weight shifters outperforms *STR* by 16% on average. Given a set of oneffsets, PRA_{0b} will accommodate the minimum non-zero oneffset per cycle via its second level shifter.

6.2.2 Area and Power: Table 3 shows area and power for *DaDN* and *Pragmatic*. Two 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, the compute area overheads are relatively small. PRA_{2b} is the most efficient configuration with an average speedup of 2.59× for an area and power cost of 1.68× and 2.50×, respectively, over *DaDN*. Accordingly, the rest of this evaluation restricts attention to this configuration.

The offset generators for one tile in this configuration requires 181,950 um^2, 35.8 mW and adds two stages to the compute pipeline. The average wire density across all metal layers is 11.1% for DaDN and 26.6% for *PRA*. Comparing post synthesis and post layout power reports, wiring increases power by 17.0% in DaDN and 22.5% in *PRA*.

6.3 Per-Column Synchronization

6.3.1 Performance: Figure 10 reports the relative to *DaDN* performance for PRA_{2b} with column synchronization and as a function of the number of WSRs as per Section 5.5. Configuration PRA^{xR}_{2b}

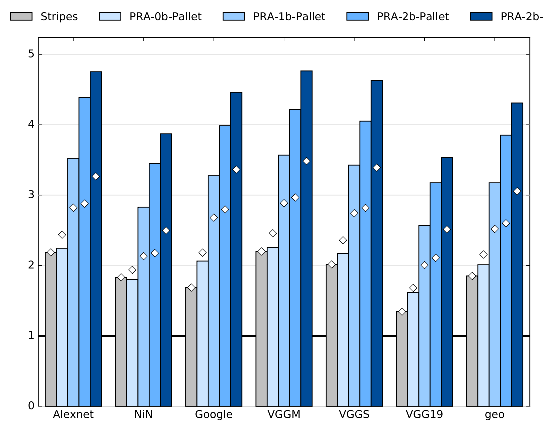

Figure 11: Relative performance of *Pragmatic* using Improved Oneffset Encoding for different configurations. Marked: performance not using IOE

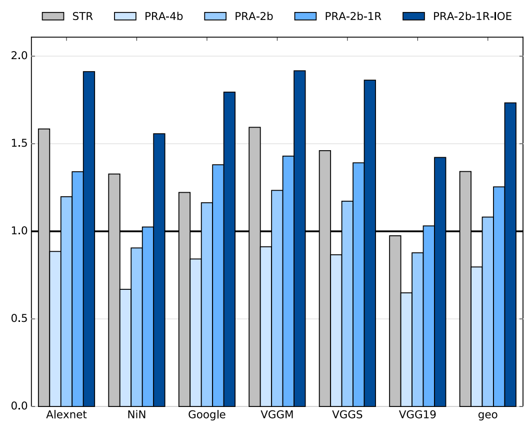

Figure 12: Relative energy efficiency

refers to a configuration using x WSRs. Even PRA_{2b}^{1R} boosts performance to 3.06× on average, close to the 3.45× that is ideally possible with $PRA_{2b}^{\infty R}$.

6.3.2 Area and Power: Table 4 reports the area per unit, and the area and power per chip. PRA_{2b}^{1R} offers most of the performance benefit with little additional hardware. This configuration increases chip area to only 1.68× and power to only 2.54× over *DaDN*.

Table 5 shows the area and power breakdown of this configuration. Since we do not layout a full chip design we estimated the interconnect cost separately. We assume the interconnect will be routed over the existing logic and does not increase chip area. An interconnect width of 4 bits per activation was chosen to balance performance and power. This configuration yields performance within 1.2% of the ideal (infinite bandwidth), while consuming 6% of the chip power.

6.4 Improved Oneffset Encoding

Figure 11 reports performance for *PRA* when using the improved offset encoding (IOE) described in Section 5.6. The considered configurations include PRA_{0b}, PRA_{1b} and PRA_{2b} (with pallet synchronization), and PRA_{2b}^{1R}. PRA_{0b} degrades performance by 7%, but the other configurations show improvements of 26%, 48%, and 41% respectively. A cause of degradation for PRA_{0b} is the increased spread of oneffset values (for example, the pair of activations 011101, 010101 takes 4 cycles with conventional encoding and 5 with enhanced encoding even though the total count of oneffsets is reduced from 7 to 6). On average and with the best configuration, this encoding improves speedup to 4.31x over *DaDN*.

6.5 Energy Efficiency

Figure 12 shows the energy efficiency of various *Pragmatic* configurations. *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. The power overhead of PRA_{single} (PRA_{4b}) is more than the speedup resulting in a circuit that is 23% less efficient than *DaDN*. PRA_{2b} reduces that power overhead while maintaining performance yielding an efficiency of 3%. PRA_{2b}^{1R} boosts performance with little hardware cost, increasing efficiency to 21%.

Finally, using IOE increases efficiency to 1.70× over *DaDN*, an improvement of 7.2x and 31x in *EDP* [10] and ED^2P [21] respectively.

Tiles	Filters/tile	Terms/filter	Performance
4	16	16	4.27
8	16	16	4.29
16	16	16	4.31
16	16	8	5.10
16	16	4	5.94
16	16	2	6.67
16	16	1	7.79

Table 6: Tile Configuration: Performance relative to *DaDN*

6.6 Sensitivity to Tile Configuration

Table 6 reports performance of PRA_{2b}^{1R} over an equivalent *DaDN* for various *Tiles-Filters/Tile-Terms/Filter* configurations. The configuration studied thus far was *16-16-16*. Reducing the number of tiles does not change the relative performance improvement with *PRA*. Decreasing the number of terms per filter, however, greatly impacts relative performance. At the extreme, where only one term per filter is processed, *PRA* is almost 8× faster than *DaDN*. This result demonstrates that most of the performance potential loss is due to imbalance across activation lanes.

6.7 8-bit Quantization

Table 7 reports performance for *DaDN*, *STR*, and *PRA* configurations using the 8-bit quantized representation used in Tensorflow [11, 29]. This quantization uses 8 bits to specify arbitrary minimum and maximum limits per layer for the activations and the weights separately, and maps the 256 available 8-bit values linearly into the resulting interval.

	8-bit *DaDN*	8-bit *STR*	8-bit *PRA*
Unit Area	0.32	0.77	1.03
Total Area	36.4	43.7	47.8
Unit Power	0.189	0.294	0.456
Total Power	9.4	11.2	16.1
Speedup	1.00	1.00	2.25
Efficiency	1.00	0.84	1.31

Table 7: Area [mm^2], power [W], speedup and efficiency for 8-bit variants the three designs.

Table 7 reports energy efficiency relative to *DaDN* for 8-bit versions of *STR* and PRA^{1R}_{2b} with IOE. In these designs, both activations and weights are 8 bits. For *STR* and *PRA*, we reduce the number of SIP/PIP columns to 8 to match *DaDN*'s throughput in the worst case. *STR* yields no speedup since we did not profile reducing the precision on top of the 8-bit quantization. As a result it is less energy efficient than *DaDN*. *PRA*'s speedup is 2.25x with an area overhead of 1.31x and a power overhead of 1.71x, making it 1.31x more energy efficient on average.

7 RELATED WORK

The acceleration of Deep Learning is an active area of research and has yielded numerous proposals for hardware acceleration. Due to limited space this section comments only on works that exploit the various levels of *informational inefficiency* in DNNs.

At the value level it has been observed that many activations and weights are ineffectual. Han *et al.*,retrain fully-connected layers to eliminate ineffectual weights and eliminate computations with zero valued activations and ineffectual weights [12, 13]. Albericio *et al.*, exploit zero and near zero activations to improve performance for a *DaDN*-like accelerator [2]. SCNN eliminates both ineffectual weights and activations [24]. Ineffectual activations and weights have also been exploited to reduce power [6, 26] The occurrence of ineffectual activations appears to be an intrinsic property of CNNs as their neurons are designed to detect the presence of relevant features in their input [2]. This is amplified by the popularity of the ReLU activation function [23].

informational inefficiency manifests also in excess of precision. *Stripes* exploits the varying per layer precision requirements of CNNs [18] whereas Quantization uses a uniform reduced precision [29]. Profiling has been used to determine appropriate hardwired precisions [20]. At the extreme end of the spectrum are binarized [7], and ternary networks, e.g., [3]. By redesigning the network or by using binary or ternary weights they indirectly exploit weight precision requirements. Where such networks are possible, they are preferable due to their reduced area, power, and complexity. However, accuracy with these tends to suffer and they often require redesigning the network.

Other works show that it is possible to achieve the same accuracy for a given task with networks that are orders of magnitude smaller than the originally proposed for such task [15] suggesting that networks are often over-provisioned.

This work exposes informational inefficiency at the bit level, which subsumes both ineffectual activation values and excess of precision. Section 2 shows that significant potential exists for exploiting the informational ineffciency at the bit-level even if zero activations are eliminated or even with 8-bit quantization. Exploiting weight sparsity with *Pragmatic* would require decoupling the synapse lanes vertically. The synapse lanes across the same row can remain in sync as they all use the same weight. However, the complexity and overhead of such a design needs to be evaluated and is left for future work.

In all, the aforementioned body of work suggests that existing networks exhibit informational inefficiency at various levels and for various reasons. Whether these inefficiencies are best exploited statically, dynamically, or both is an open question. Furthermore, which forms of inefficiency will persist as networks evolve remains to be seen. Ideally, network designers would like to be able to identify which parts of a network perform which classification subtask. This would favour over-provisioning suggesting that informational inefficiency is desirable. The experience so far has been that designing networks is a difficult task and thus adding another level of complexity to exploit inefficiency by static means may be undesirable favouring dynamic solutions such as *Pragmatic* that work with out-of-the-box networks.

8 CONCLUSION

To the best of our knowledge *Pragmatic* is the first CNN accelerator that exploits not only the per layer precision requirements of CNNs but also the essential bit information content of the activation values. Future work may combine *Pragmatic* with sparse network accelerators, investigate alternate activation encoding that reduce bit density, or target improved synchronizations mechanisms.

ACKNOWLEDGEMENTS

This work was performed while Jorge Albericio was at the University of Toronto and was supported by an NSERC Discovery Grant and an NSERC Discovery Accelerator Supplement.

REFERENCES

[1] "How to Quantize Neural Networks with TensorFlow." [Online]. Available: https://www.tensorflow.org/performance/quantization

[2] 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.

[3] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient AI applications," *CoRR*, vol. abs/1609.00222, 2016. [Online]. Available: http://arxiv.org/abs/1609.00222

[4] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[5] 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*, Dec 2014, pp. 609–622.

[6] 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*, 2016, pp. 262–263.

[7] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *ArXiv e-prints*, Nov. 2015.

[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*, ser. ISCA '11. New York, NY,

USA: ACM, 2011, pp. 365–376.
[9] 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.
[10] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1277–1284, Sep 1996.
[11] Google, "Low-precision matrix multiplication," https://github.com/google/gemmlowp, 2016.
[12] 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. [Online]. Available: http://arxiv.org/abs/1602.01528
[13] 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. [Online]. Available: http://arxiv.org/abs/1510.00149
[14] 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.
[15] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360
[16] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. Enright Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Workshop On Approximate Computing (WAPCO)*, 2016.
[17] 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.
[18] 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*, ser. MICRO-49, 2016.
[19] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-serial Deep Neural Network Computing ," *Computer Architecture Letters*, 2016.
[20] 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)*, May 2014, pp. 7510–7514.
[21] A. J. Martin, M. Nyström, and P. I. Pénzes, "Et2: A metric for time and energy efficiency of computation," in *Power aware computing*. Springer, 2002, pp. 293–315.
[22] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."
[23] 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)*, 2010, pp. 807–814.
[24] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 27–40. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080254
[25] 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*, March 2015, pp. 1543–1546.
[26] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 267–278.
[27] Synopsys, "Design Compiler," http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages.
[28] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Computers*, vol. 13, no. 1, pp. 14–17, 1964. [Online]. Available: http://dx.doi.org/10.1109/PGEC.1964.263830
[29] P. Warden, "Low-precision matrix multiplication," https://petewarden.com, 2016.
[30] H. H. Yao and E. E. Swartzlander, "Serial-parallel multipliers," in *Proceedings of 27th Asilomar Conference on Signals, Systems and Computers*, Nov. 1993, pp. 359–363 vol.1.