Stripes: Bit-Serial Deep Neural Network Computing

Patrick Judd*, Jorge Albericio*, Tayler Hetherington†, Tor M. Aamodt†, Andreas Moshovos*

* Department of Electrical and Computer Engineering
University of Toronto
{juddpatr,jorge,moshovos}@ece.utoronto.ca

† Department of Electrical and Computer Engineering
University of British Columbia
{taylerh,aamodt}@ece.ubc.ca

***Abstract*—Motivated by the variance in the numerical precision requirements of Deep Neural Networks (DNNs) [1], [2], *Stripes* (*STR*), a hardware accelerator is presented whose execution time scales almost proportionally with the length of the numerical representation used. *STR* relies on bit-serial compute units and on the parallelism that is naturally present within DNNs to improve performance and energy with no accuracy loss. In addition, *STR* provides a new degree of adaptivity enabling on-the-fly trade-offs among accuracy, performance, and energy. Experimental measurements over a set of DNNs for image classification show that *STR* improves performance over a state-of-the-art accelerator [3] from 1.30x to 4.51x and by 1.92x on average with no accuracy loss. *STR* is 57% more energy efficient than the baseline at a cost of 32% additional area. Additionally, by enabling configurable, per-layer and per-bit precision control, *STR* allows the user to trade accuracy for further speedup and energy efficiency.**

I. Introduction

Deep neural networks (DNNs) are the state-of-the-art technique in many recognition tasks such as object [4] and speech recognition [5]. DNNs comprise a feed-forward arrangement of *layers*, each exhibiting high computational demands and parallelism which are commonly exploited with the use of Graphics Processing Units (GPUs). However, the high computation demands of DNNs and the need for higher energy efficiency motivated special purpose architectures such as the state-of-the-art DaDianNao (*DaDN*) which was reported to be up to 330x more energy efficient than a GPU [3]. As power tends to be the limiting factor in modern high-performance designs, it is essential to achieve better energy efficiency in order to improve performance further under the given power constraints [6].

This work presents *Stripes* (*STR*), an implementation of a DNN performance improvement technique that 1) is complementary to existing techniques which exploit parallelism across computations, while 2) improving energy efficiency, and 3) enabling accuracy vs. performance trade offs. *STR* goes beyond parallelism across computations and exploits the data value representation requirements of DNNs. *STR* is motivated by the observation that the precision required by DNNs varies significantly not only across networks but also across the layers of the same network [1], [2]. Most existing implementations rely on a *one-size-fits-all* approach, using the *worst-case* numerical precision for all values. For example most software implementations use 32-bit floating-point [7], [8], [9] while accelerators and some recent GPUs use 16-bit fixed-point [3], [10], [11].

Rather than using a fixed precision for all layers, *STR* instead allows per-layer selection of the precision used providing a new dimension upon which to improve performance. To do so, *STR*'s execution units are designed so that execution time scales linearly with the length, in *bits*, of the numerical precision needed by each layer. *STR* is presented as an extension to the state-of-the-art accelerator *DaDN*. Since *DaDN* uses a 16-bit fixed-point representation, *STR* would ideally improve performance at each layer by $16/p$ where p is the layer's required precision length in bits. This is done by using serial-parallel multiplication where the computational time is determined by the length of the serial input. To compensate for the longer compute latency, both the reduced circuit size and abundance of parallel computation in DNNs are exploited to increase compute throughput.

Beyond offering a new dimension upon which to improve DNN execution performance, using bit serial computations enables static and on-the-fly energy, performance, and accuracy trade offs. By reducing precision an application may choose to reduce accuracy in exchange for improved performance and energy efficiency. This capability would be useful, for example: 1) on a battery operated device where a user or the operating system may opt for sligthly reduced accuracy in exchange for longer up time, or 2) under strict time constraints where an application may opt for a less accurate but timely response.

Experiments demonstrate that over eight DNNs, *STR* improves performance by 1.92x, and energy efficiency by 57% on average without any loss in accuracy at an overall area overhead of 32%. When small accuracy losses are acceptable further improvements are possible and demonstrated. For example, when an *up to* 1% loss in relative accuracy can be tolerated, *STR* improves performance by 2.08x and energy efficiency by 68% on average. Finally, this work demonstrates that *STR* improves performance over a decomposable unit approach similar to that used in multimedia instruction set extensions, e.g., [12].

The rest of the paper is organized as follows: Section II corroborates the per-layer precision requirement variability of DNNs motivating *STR*, Section III explains the key idea behind *STR* using a simplified example, Section IV reviews

978-1-5090-3508-3/16/$31.00

Network	Relative Accuracy			
	100%		99%	
	Per Layer Neuron Precision in Bits	Ideal Speedup	Per Layer Neuron Precision in Bits	Ideal Speedup
LeNet	3-3	5.33	2-3	7.33
Convnet	4-8-8	2.89	4-5-7	3.53
AlexNet	9-8-5-5-7	2.38	9-7-4-5-7	2.58
NiN	8-8-8-9-7-8-8-9-9-8-8-8	1.91	8-8-7-9-7-8-8-9-9-8-7-8	1.93
GoogLeNet	10-8-10-9-8-10-9-8-9-10-7	1.76	10-8-9-8-8-9-10-8-9-10-8	1.80
VGG_M	7-7-7-8-7	2.23	6-8-7-7-7	2.34
VGG_S	7-8-9-7-9	2.04	7-8-9-7-9	2.04
VGG_19	12-12-12-11-12-10-11-11-13-12-13-13-13-13-13-13	1.35	9-9-9-8-12-10-10-12-13-11-12-13-13-13-13-13	1.57

TABLE I: Per Convolutional layer neuron precision profiles needed to maintain the same accuracy as in the baseline (100%) and to reduce it within 1% of the baseline (99%). *Ideal*: Potential speedup with *Stripes* over a 16-bit baseline.

the operation of convolutional layers and the *DaDN* design, Section V presents *Stripes* in detail, Section VI demonstrates *STR*'s benefits experimentally, Section VII reviews related work, and Section VIII concludes.

II. Motivation

To motivate *STR* this section estimates the performance improvements that may be possible if execution time scaled proportionally with the length of the numerical representation. In more detail, this section shows: 1) most of the execution time in DNNs is taken by convolutional layers which are the layers that *STR* targets, 2) the numerical precision needed varies across layers and networks, 3) significant performance potential exists if execution time could scale with precision, and 4) having execution time depend on precision enables further performance improvements provided that a loss in accuracy is acceptable. Section VI describes the experimental methodology.

1) Convolutional Layers Take Most of the Execution Time: *STR* targets the convolutional layers of DNNs since, for the selected networks, 85% of the *overall* time during classification is taken by these layers on *DaDN*.

2) Numerical Precision Requirements Vary Across and Within Networks: Table I shows the required per-layer neuron precisions (precision profiles) for a set of DNNs while maintaining 100% relative classification accuracy with respect to a full precision implementation. The methodology for finding these precision profiles is described in Section VI-A.

These precision profiles highlights a key property: Numerical precision requirements vary not only across networks but also across the layers of the same DNN [1], [2]. This property allows the representation length to be tuned per layer without a loss in accuracy and has been used to simplify hardwired DNN implementations [2] and to reduce memory footprint [13]. *STR* exploits this property to improve compute performance and energy efficiency by using bit-serial compute units whose execution time scales proportionally with the numerical precision.

3) Precision Variability Could Improve Performance: To motivate *STR*, the "Ideal Speedup" column in Table I reports the performance improvement that would be possible if execution time scaled linearly with the neuron precision length. Specifically, given that the baseline design uses a 16-bit fixed-point representation, the reported speedups are calculated assuming that execution time is $p/16$ for convolutional layers when using a neuron representation of p bits. The potential speedup varies from 1.35x (VGG19) to 5.33x (LeNet) and is 2.29x on average.

4) Enabling Accuracy vs. Performance Trade-offs: Further reductions in precision are possible if a loss in accuracy is acceptable. As an example, Table I shows how the precision requirements can be further relaxed when an *up to* 1% drop in relative accuracy is acceptable. Performance improvements increase and vary from 1.57x (VGG19) to 7.33x (LeNet), and are 2.54x on average. Further trade offs between accuracy and numerical precision, and thus performance improvements, are possible as Section VI-E demonstrates.

Section VI will demonstrate that *STR* achieves within 2% of the ideal performance reported in this section.

III. Stripes' Approach: A Simplified Example

This section presents a simplified example illustrating the concept behind *STR*'s design. The computations of a DNN can mostly be broken down into inner products. As such, DNN accelerators, like *DaDN* are specialized for inner product computation.

Figure 1a shows a simplified inner product compute unit (IP) processing vectors $A = (1, 0)$ (neurons) and $B = (1, 3)$ (synapses). The example assumes a fixed numerical representation length of two bits for all values. A conventional compute unit would use bit-parallel multipliers and would in a single cycle compute the inner product of A and B, $(1 \times 1, 0 \times 3)$, or $(1, 0)$. The two products are then added through a bit-parallel adder to calculate the final inner product $A \cdot B = (1)$. In total, the input bandwidth of this unit is 8 bits per cycle (2 neurons of 2 bits per cycle and 2 synapses of 2 bits per cycle), and its output bandwidth is 2 bits per cycle.

Figure 1b shows a Serial Inner Product unit (SIP) where A's values have been transposed and are now processed bit-serially over two cycles (this example assumes unsigned numbers for simplicity and Section V-C discusses support for signed numbers). During the first cycle, the most significant bit from each A value is *ANDed* with the corresponding B value producing two 2-bit results $(0, 0)$. These two results

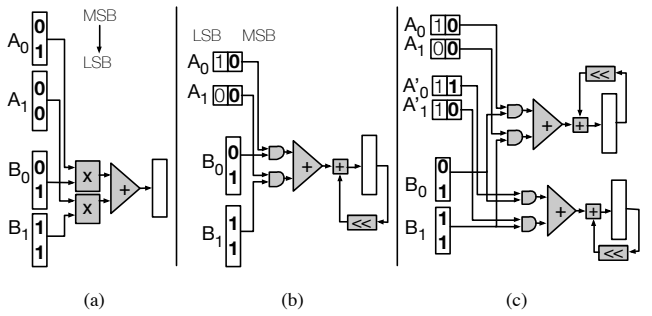


Fig. 1: Implementing an inner product in hardware. (a) Bit-Parallel Computation: Fixed numerical representation length for both A and B and execution time is constant independent of A's representation. (b) Bit-serial computation for A: Execution time depends on A's length. (c) Bit-serial computation for A and A' matching (a)'s throughput.

are added together producing a partial output (0). During the second cycle, the next set of bits from A are ANDed with the corresponding B values, and the two 2-bit results $(3, 0)$ are added to the partial output after shifting the latter by a single bit. Since A's elements are processed bit-serially, it now takes two cycles to compute the inner product. In general, execution time is now proportional to the bit-width of A's values and as long as there is enough precision in the accumulator and the adders, a range of A precisions can be naturally supported. This is a crucial step towards designing a convolutional layer execution unit whose execution time scales linearly with the input precision as per the discussion of Section II.

However, overall output bandwidth has now been reduced to half. Whereas the baseline unit would compute the inner-product in one cycle, the bit-serial design takes two cycles when A needs the full two-bit precision. Fortunately, there is abundant parallelism in the convolutional layers of DNNs which *STR* exploits to offer at least the same computational bandwidth as the conventional bit-parallel design.

In our example, the unit could process another pair of neurons A' in parallel as long as the inner product $A' \cdot B$ is also needed. Figure 1c shows the unit reading one bit from each of the elements of the A and A' vectors for a total of 4 neuron bits per cycle, the same input neuron bandwidth as the baseline design. Since B is used for both inner-products, the input synapse bandwidth remains at 4 bits per cycle. This approach can be successful only as long as there is sufficient reuse of one of the synapse inputs (B in our example). Fortunately, as the next section will explain in detail, convolutional layers do exhibit such reuse.

IV. Background

This work presents *Stripes* as a modification of the state-of-the-art DaDianNao (*DaDN*) accelerator [3]. Accordingly, this section provides the necessary background information: Section IV-A reviews the operation of convolutional layers, and Section IV-B reviews the baseline accelerator design and how it implements convolutional layers.

A. Convolutional Layer Computation

The input to a convolutional layer is a 3D array. 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 $N_x \times N_y \times I$ real numbers, or *neurons*. The layer applies N filters, each containing $F_x \times F_y \times I$ real numbers, or *synapses*. The layer outputs a $O_x \times O_y \times N$ neuron array (its depth is the filter count). The neuron arrays can be thought of as comprising several *features*, that is, 2D arrays stacked along the i dimension, each corresponding to an output feature. The output has N features, each produced by a different filter. Applying the filter identifies where in the input each feature appears. To calculate an output neuron, one filter is applied over a *window*, a sub-array of the input neuron array that has the same dimensions as the filters $F_x \times F_y \times I$. Let $n(x, y, i)$ and $o(x, y, i)$ be respectively input and output neurons, and $s^f(x, y, i)$ be synapses of filter f. The output neuron at position $(\mathbb{k}, \mathbb{l}, \mathbb{f})$ is calculated as:

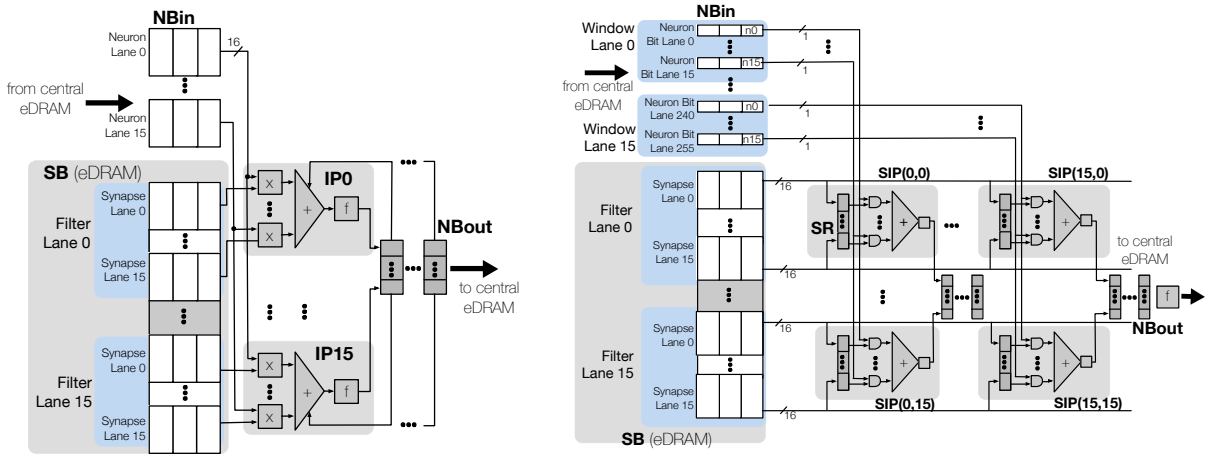
$$\underbrace{o(\mathbb{k},\mathbb{l},\mathbb{f})}_{\substack{output\\neuron}} = \underbrace{\sum_{y=0}^{F_y-1}\sum_{x=0}^{F_x-1}\sum_{i=0}^{I-1}\underbrace{s^{\mathbb{f}}(y,x,i)}_{synapse} \times \underbrace{n(y+\mathbb{l}\times S, x+\mathbb{k}\times S, i)}_{input\ neuron}}_{window} \quad (1)$$

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

In the rest of this paper, the term *brick* is used to refer to a set of elements continuous along the i dimension of a 3D neuron or synapse array, 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., $n_B(x, y, i)$.

B. Baseline System

We demonstrate *STR* as an extension over the *DaDianNao* (*DaDN*) accelerator [3]. Figure 2a shows a *DaDN* tile. Each *DaDN* chip comprises 16 tiles. Each tile has a synapse buffer (SB) which provides 256 synapses per cycle, one per synapse lane. The tile also has an input neuron buffer (NBin) which provides 16 neurons per cycle, one per neuron lane, and an output neuron buffer (NBout) which can accept 16 output neurons per cycle. The computational logic is called the *Neural Functional Unit (NFU)*, or *unit*. Every cycle, each NFU produces a brick $o_B(q, w, f)$ of partial output neurons. The NFU does so by processing one input neuron brick $n_B(x, y, i)$ and 16 synapse bricks, one from each of 16 filters: $s_B^f(k, l, i)$ through $s_B^{f+15}(k, l, i)$. For this purpose, the NFU has 16 neuron lanes and 16 filter lanes each with 16 synapse lanes for a total of 256 synapse lanes. Each neuron lane is connected



(a) (b)

Fig. 2: a) DaDianNao Tile. b) Stripes Tile.

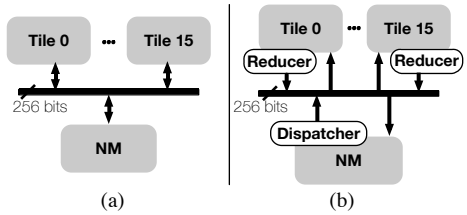
to 16 synapse lanes, one from each of the 16 filter lanes. A synapse lane multiplies its synapse with an input neuron and the 16 products from all synapse lanes of a filter are reduced into a partial sum. In all, the filter lanes each produce a partial sum per cycle, for a total 16 output neurons per unit. We refer to the logic associated to the production of one output neuron as an *inner product unit (IP)*. Once a full window is processed the 16 resulting sums are fed though a non-linear activation function, f, to produce 16 output neurons. Each IP contains sixteen 16-bit multipliers, a 16 input adder tree and an adder to add the tree out put to a previously computed partial sum from NBout.

DaDN's main goal was minimizing off-chip bandwidth while maximizing on-chip compute utilization. It does this with 32MB of on chip eDRAM distributed as 2MB chuncks (SB), one per NFU. Overall, a *DaDN* node can process up to 256 filters in parallel, 16 per unit. All inter-layer neuron outputs except for the initial input and final output are stored in shared, 4MB central eDRAM, or *Neuron Memory* (NM). 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 the external memory the first layer's synapses, and the input image. The synapses are distributed over SBs and the input is stored into NM. Each cycle an input neuron brick, $n_B(x, y, i)$, is broadcast to all units. The layer's 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. Stripes

This section describes the *Stripes* design. Figure 3 shows an overview of the complete *DaDN* and *STR* systems. Both systems use a fat tree interconnect that connects 256 bits to each tile, either by broadcasting 256 bits or distributing 4096



(a) (b)

Fig. 3: Overview of the system components and their communication. a) *DaDN*. b) *Stripes*.

bits across the 16 tiles. Nm stores neurons in 16-bit words regardless of the precision used. *STR* introduces a *Dispatcher* unit to read neurons from NM using the existing bit-parallel interface while broadcasting them to the tiles bit-serially. Each *STR* tile uses a *Reducer* unit before writing the output neurons to NM.

The rest of this section is organized as follows: Section V-A discusses the approach *STR* uses to enable bit-serial processing of neurons while matching or exceeding *DaDN*'s performance. Section V-B describes the organization and operation of *STR*'s tiles. Section V-C describes the core bit-serial inner-product units used in the tiles. Section V-D describes the Dispatcher, and Section V-E describes the Reducers. Section V-F describes how *STR* implements the non-convolutional layers, and Section V-G concludes the description of *STR* by explaining how the per layer precision information is communicated. Finally, Sections V-H and Section V-I compare *STR*'s approach with increasing the number of tiles and with using decomposable units respectively.

A. Processing Approach

Since *STR* uses bit-serial computation for neurons, it needs to process more neurons in parallel than *DaDN* to maintain performance when the baseline precision is used. Specifically, in the worst case, *STR* needs 16 cycles to calculate a product

involving a 16-bit neuron. Given that a *DaDN* tile processes a 16 neuron brick in parallel, *STR* needs to process 16 bricks, or 256 neurons in parallel. The parallelism of convolutional layers offers a multitude of options for processing neurons in parallel. *STR* opts to process 16 windows in parallel using a neuron brick from each window so that the same 16 synapses from each of the 16 filters can be used to calculate 16×16 output neurons in parallel. For example, for a layer with a stride of 2 an *STR* tile will processes 16 neuron bricks $n_B(x, y, i)$, $n_B(x + 2, y, i)$ through $n_B(x + 31, y, i)$ in parallel, a single bit per neuron per cycle. In this case, assuming that the tile processes filters f_i though f_{i+15}, after p cycles it would produce the following *partial* output neurons: $o_B(x/2, y/2, f_i)$, through $o_B(x/2 + 15, y, f_i)$, or a *pallet* of 16 output neuron bricks that are contiguous in the scan order of the (x, y) plane. Whereas *DaDN* would process 16 neuron bricks over 16 cycles, *STR* processes them concurrently but bit-serially over p cycles. If p is less than 16, *STR* will outperform *DaDN* by $16/p$, and when p is 16, *STR* will match *DaDN*'s performance.

B. Tile Organization and Operation

As Figure 2b shows, each *STR* tile is organized as follows: the tile's NBin is logically organized in 16 *window lanes*, each a group of 16 bit-serial neuron lanes for a total of 256 neuron lanes. Each window lane processes one of the 16 input neuron array windows. The SB is identical to *DaDN* and is logically organized in 16 filter lanes, each containing 16 synapse lanes. The SB and NBin connect to an array of 16×16 *Serial Inner Product* (SIP) units, where each SIP is responsible for one output neuron. The SIP(f,w) at row f and column w processes filter lane f and neuron window w. The SB filter lane f connects via a bus to all SIPs along row f, whereas the NBin window lane w connects via a bus to all SIPs along column w. Each SIP accepts as input 16 neuron bits and a synapse brick which is latched onto a *synapse register* (SR). The SR is needed to support fully connected layers as Section V-F explains. The figure shows simplified SIPs and a more complete description is given in Section V-C. Each SIP contains an adder tree for a total of 256 adder trees whereas *DaDN* requires only 16, one per IP. It may seem that would increase area considerably for *STR*, however, each *DaDN* IP requires 256 multipliers, whereas *STR* requires none. Section VI will show that the *STR*'s inner product compute area overhead is 107% compared to *DaDN*.

Processing in a tile proceeds in phases of p cycles cycles each, where p is the precision of neurons in bits. At the first cycle of a phase, SB provides 16 bricks of 16-bit synapses, one brick per filter. Each SIP latches its corresponding synapse brick in its SR. Every cycle, NBin provides 256 neuron bits and each neuron bit is bit-wise *ANDed* with 16 synapses, one per SIP along the same column. Each AND operation produces a 16-bit *term*. Thus, each SIP calculates 16 terms corresponding to one filter and one window. The SIP sums its 16 terms into a partial output neuron using a dedicated 16-input adder tree. For the remaining $p - 1$ cycles of a phase, each SIP shifts its partial output neurons by one bit, while accumulating another 16 terms implementing bit-serial multiplication. After p cycles, an output neuron pallet, that is 256 16-bit partial output neurons, is produced in full.

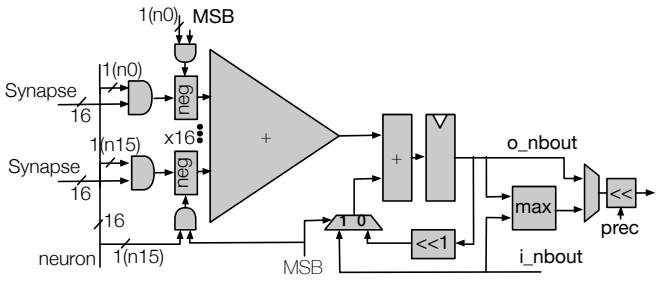


Fig. 4: SIP components.

In *STR*, The 64 entry NBout is distributed across SIP columns, with 4 entries per column. The activation function unit is moved to the output of NBout, since the activation function is only applied to the full sum before it is written back to NM.

C. SIP: Bit-Serial Inner-Product Units

In the described implementation, *STR* tiles produce 256 output neurons concurrently over multiple cycles. Each output neuron computation is an inner product. Whereas the baseline design calculates this inner product 16 input neurons and 16 synapses at a time, *STR* does so differently. For each output neuron and at each cycle, 1 bit from each of 16 input neurons along with 16 synapses are combined.

Multiplying neurons bit-serially is straightforward where the neuron is fed serially and the synapse is fed in-parallel. Specifically, given an input neuron n of length p bits, n's binary representation is $\sum_{b=0}^{p} n_b \times 2^b$, where n_b is n's bth bit. Given a synapse s, the multiplication $s \times n$ can be rewritten as $\sum_{b=0}^{p} 2^b \times n_b \times s$. This leads to a circuit implementation where $n_b \times s$ is an AND, multiplication with 2^b is a shift and the summation is performed with an accumulator over p cycles.

To apply this naively to *DaDN*, we could simply convert each of the parallel multipliers to serial ones. However, we can simplify this design using the commutative property of addition as described by White [14]. Formally, the terms of the inner product of Equation (1) can be reorganized as follows where n^b the b*th* bit of n and $Ni = 16$ is the size of the vectors.

$$\sum_{i=0}^{N_i-1} s_i \times n_i = \sum_{i=0}^{N_i-1} s_i \times \sum_{b=0}^{p-1} n_i^b \times 2^b = \sum_{b=0}^{p-1} 2^b \times \sum_{i=0}^{N_i-1} n_i^b \times s_i \tag{2}$$

In terms of logic, this shows that we can first perform the reduction on the products $n_i^b \times s_i$ with an adder tree, and then perform the shift and accumulate on the resulting sum. This simplifies the serial inner product unit by moving the

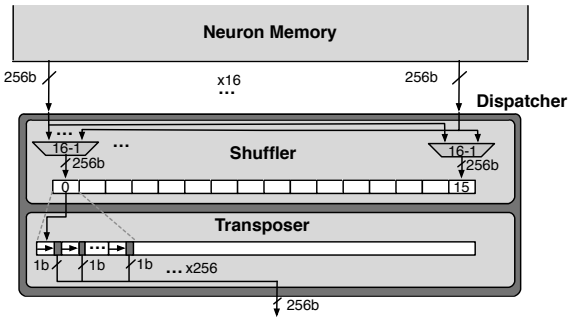


Fig. 5: Dispatcher.

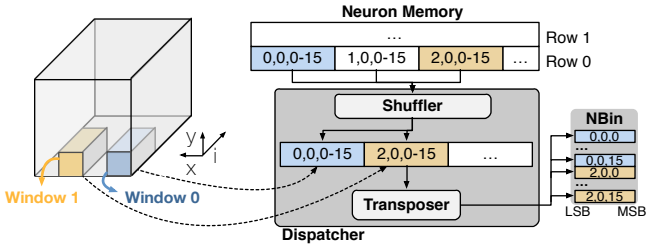


Fig. 6: Memory mapping of elements from different windows.

shifted accumulator from each of the 16 inputs in the adder tree to just one on the output. Figure 4 shows the resulting serial inner product circuit. To support signed 2's complement neurons, the SIP must subtract the synapse corresponding to the MSB from the partial sum when MSB is 1. This is done with negation blocks for each synapse before the adder tree. To support inner products larger than Ni data is read back from NBout and used to initialize the accumulator. Each SIP also includes a comparator (max) to support max pooling layers. At the output of the SIP a shifter is used to align the partial sum with the output of the adder tree on the first cycle of the next serial inner product computation. The amount of shifting depends on the precision of the input.

D. Dispatcher: Supplying Input Neurons

DaDN's neuron memory broadcasts a brick, that is 16 16-bit neurons, or 256 bits per cycle to all tiles and each tile processes the same brick over different filters. *STR* needs to also broadcast 256 bits per cycle to all tiles, but where each bit corresponds to a different neuron. *STR* currently opts to maintain the same neuron storage container format in central neuron memory (NM) as in *DaDN* aligning each neuron at a 16-bit granularity. A *Dispatcher* unit is tasked with reading neurons from NM and feeding them to the *STR* tiles bit-serially. Section V-D1 describes how neurons are read from the NM, and Section V-D2 describes how the Dispatcher combines the raw neuron data from the NM and feeds them serially to the tiles.

1) Reading the Neurons from NM: Reading the necessary neurons from NM is best understood by first considering a layer using a unit stride. In this case, at each cycle, the *STR* units need to be fed with bits from 16 bricks, contiguous along the x dimension: $n_B(x, y, i))$, $n_B(x+1, y, i)$ through $n_B(x+15, y, i)$. Provided that these 16 neuron slices can be read in parallel, all the dispatcher has to do, is feed them bit serially over the next p cycles. To enable reading the 16 bricks in parallel *STR* maps them on consecutive locations in NM. In many cases, this will result in the 16 bricks being stored onto the same NM row. In this case, the dispatcher can read them all in a single cycle (given that the 2MB NM comprises several subarrays, reading 256 neurons in parallel is feasible).

Depending on the input neuron array y dimension, the 16 bricks may spread over two NM rows (this is similar to reading a misaligned block from a data cache). In this case, the dispatcher will have to read and combine the appropriate bricks from up to two rows over two cycles before it can feed the *STR* tiles. As long as p is at least 2, the next set of 16 bricks will not be needed until p cycles have elapsed. Accordingly, the whole process can be pipelined and thus the *STR* units can be kept busy all the time.

Figure 6 shows an example with two windows and stride of 2. When the stride S is more than one, the 16 bricks could be spread over $R = min[S + 1, 16]$ rows. Accordingly, the dispatcher will have to read R rows over R cycles before feeding the next set of 256 neurons to the tiles. As long as R is less than p there will be enough time to keep the units busy all the time. Only when R is more than p the units will have to stall for $R-p$ cycles. It may be possible to reduce the number of rows that the dispatcher needs to read by mapping bricks to NM taking into account the stride. However, as Section VI will show, such stalls were observed only for one layer of LeNet and only for 2% of the time. Accordingly, this is left for future work.

2) Dispatcher Design: As the previous section described, given a layer stride S, the dispatcher needs to read up to $max[S+1, 16]$ rows, one per cycle to collect 16 bricks. Given these 16 bricks, the dispatcher then needs to send one bit from each of the 256 neurons they contain, for a total 256 bits per cycle, over p cycles to the *STR* tiles.

As Figure 5 shows, the dispatcher is composed of two parts: a *Shuffler* and a *Transposer*. The Shuffler reads 16 bricks from NM and the Transposer communicates them bit-serially to the tiles. The dispatcher needs to communicate a new set of 16 bricks every p cycles.

Shuffler: The Suffler's goal is to collect the 16 bricks that are needed to keep the *STR* units busy over p cycles. The collection of the next group of 16 bricks, can be overlapped with the processing of the current group, hence the Shuffler needs to collect a 16 brick group every p cycles.

Each cycle, the Shuffler reads a row of 256 neurons from NM. Given that bricks are stored contiguously in NM every row contains 16 bricks, however, these many not all be the bricks currently needed. A 16-to-1 multiplexer per output brick is sufficient to select the appropriate brick when that appears on the input row. Accordingly, the Shuffler comprises

16 16-to-1 256-bit (one brick of 16 neuron 16-bit containers) multiplexers. The input neurons are collected on 256 16-bit registers organized in groups of 16, one per input brick. Once the Shuffler has collected all 16 bricks, it transfers them to the input registers of the Transposer.

Transposer: The Transposer converts the neurons read from memory by the Shuffler to serial bit streams. Once the shuffler has collected all 16 bricks, it writes them bit-parallel into 256 16-bit registers. Each register provides a 16-bit write port, and a single-bit read port. Over the next p cycles, the Transposer outputs one bit per neuron for a total of 256 bits per cycle. These are broadcast to all NFUs using the same interconnect as in *DaDN*.

E. Reducers: Writing the Output Neurons

STR's NFUs produce output neurons in *DaDN*'s 16-bit fixed-point format. The *Reducer* units serve a dual purpose: 1) they convert to the precision needed by the output layer, and 2) they write the output neuron bricks to NM.

Writing the output neurons back to NM uses the same interconnect as in *DaDN*. The only difference is that since *STR* outperforms *DaDN*, it exhibits higher output neuron bandwidth demand. Fortunately, since calculating an output neuron requires processing a full input neuron window, there is enough time to meet this demand with the existing interconnect. Specifically, while *DaDN* produces a single output neuron brick, or 16 output neurons concurrently (e.g., $o_B(x, y, f_i)$), *STR* produces a pallet, or 16 bricks, (e.g., $o_B(x, y, f_i)$) through $o_B(x+15, y, f_i)$). This pallet needs to be stored contiguously in the NM address space as expected by the Dispatcher when processing the next layer. *STR*'s tiles send a single brick at a time as in the baseline and take multiple cycles to write all 16. Since the tiles write a single brick per cycle, and since bricks never span an NM row, there is no need for supporting misaligned brick writes.

STR's computational throughput is increased by roughly $16/p$ over *DaDN*. If a layer is relatively small, it is in principle possible to need extra cycles to drain all output neurons. However, even in the baseline output neurons typically take hundreds of cycles to be computed as producing an output neuron requires processing a full input neuron window. Accordingly, there is sufficient time to write all output bricks.

F. Executing Other Layers

Fully Connected: *DaDN* computes fully connected (FC) layers as a convolution layer where the filter dimensions match that of the input neuron array. In this case there is only one window necessitating a different data access and execution schedule to match *DaDN*'s performance. When processing a convolutional layer, the synapses are read only once every p cycles. To process an FC layer, the unit can load synapses in a round-robin fashion, one SIP column per cycle via the single SB read port and bus, keeping all SIPs busy processing neurons. For example, with reference to Figure 2b, a unit can load 256 synapses to SIP(0,0)..SIP(0,15) in cycle 0, then load the next 256 synapses to SIP(1,0)..SIP(1,15) in cycle 1, etc.

Cycle 0:
SIP(0,0)...SIP(0,15): latch $s_B^0(0,0,0), ..., s_B^{15}(0,0,0)$
via window lane 0: receive bit 0 of $n_B(0,0,0)$
Cycle 1:
SIP(0,0)...SIP(0,15):
via window lane 0: receive bit 1 of $n_B(0,0,0)$
SIP(1,0)...SIP(1,15): latch $s_B^0(1,0,0), ..., s_B^{15}(1,0,0)$
via window lane 1: receive bit 0 of $n_B(1,0,0)$
...
Cycle 15: Fully Utilized
SIP(0,0)...SIP(0,15):
via window lane 0: receive bit 15 of $n_B(0,0,0)$
SIP(1,0)...SIP(1,15):
via window lane 1: receive bit 14 of $n_B(1,0,0)$
...
SIP(15,0)...SIP(15,15): latch $s_B^0(15,0,0), ..., s_B^{15}(15,0,0)$
via window lane 15: receive bit 0 of $n_B(15,0,0)$
Cycle 16: Fully Utilized
SIP(0,0)...SIP(0,15): latch $s_B^0(0,0,16), ..., s_B^{15}(0,0,16)$
via window lane 0: receive bit 0 of $n_B(0,0,16)$
...
SIP(15,0)...SIP(15,15):
via window lane 15: receive bit 1 of $n_B(15,0,0)$

TABLE II: Processing an FC layer: The first 17 cycles.

The loading of synapses can be overlapped with processing neurons by staggering the neuron stream to synchronize with the loading of synapses. This mode of operation (Round robin synapse loading and staggered neurons streams) only requires modification to the control. Table II shows an example, illustrating how synapse loading and computation is overlapped. This approach improves performance for FC layers when *batching* is used, a common strategy when synapse bandwidth becomes a bottleneck. Batching computes each layer over multiple images at a time, so that synapses can be reused for neurons from different images. In this case, each synapse loaded to an SIP would be used for $p \times b$ cycles, where b is the batch size. There is a small overhead from staggering the neuron streams, which is 0.16% of the layer runtime on average. The potential for further improving performance by accelerating FC layers is small since they account for a small fraction of the overall execution time. Furthermore, the current trend in DNNs is for reducing or eliminating the number of FC layers, e.g., [15]. While there is no performance improvement over *DaDN* when processing a single image, using a shorter precision in the FC layers can be used to reduce power as only $16/p$ SIPs are needed to keep up with the bandwidth of SB. This can being exploited by either power-gating or data-gating the remaining SIPs. We leave the evaluation of this as future work.

Other Layers: For pooling layers neurons are transmitted bit-parallel from NM and bypass the adder tree in the SIPs. The dispatcher is only designed to broadcast serial data at 256 bits/cycle whereas pooling layers read 4096 bits/cycle as neurons are distributed across tiles. Max pooling is supported with comparators in the SIPs. Average pooling is supported by accumulating neurons in the SIPs and using the activation unit to scale the result. Local response normalization layers

require the use of the inner product units but due to the limited bandwidth of the dispatcher, cannot be serialized while matching the baseline performance. We leave an efficient implementation of this layer type for future work.

G. Communicating the Per Layer Precisions

This work assumes that the per layer precisions are pre-calculated using a method such as that described in Section VI-A and provided along with the network's other metadata such as the dimensions, padding and stride of each layer. Several complete profiles of per layer precisions can be supplied by the DNN to enable accuracy vs. performance trade-offs at run-time. This metadata information is read by the *STR* controller and used to control the operation of the units, the Dispatcher and the Reducers.

H. STR vs. More Bit-Parallel Units

Since the convolutional layers are highly parallel, improving *DaDN*'s performance is possible by exploiting parallelism by adding more tiles. As Section VI shows, *STR* increases tile area by 35%. Assuming ideal performance scaling, we could instead use this extra area to introduce an additional 35% more bit-parallel compute bandwidth. In reality, ideal scaling will not be possible as it may not be possible to keep all neuron lanes busy. For example, a *DaDN* chip can be fully utilized only as long as there are multiple of 256 filters in a layer (16 filters in each of the 16 tiles). As a result, depending on the layer, there are cases where units are underutilized and having more units will make such cases more common.

Even under the unrealistic assumption that *DaDN*'s performance can be scaled by 35% with the same area overhead as *STR*, Section VI shows that *STR* outperforms this alternative and thus offers a better area vs. performance scaling. Furthermore, *STR*'s approach enables static or run-time performance vs. accuracy trade offs which are not possible with the baseline design.

I. STR vs. Decomposable Processing Units

A common approach to gaining performance from reduced precision is to use decomposable multipliers and adders [12] [16] [17]. For example, a 16-bit adder can easily be configured as two 8-bit adders with minimal overhead. This approach is commonly used in multimedia instruction set extensions of general purpose processors [12]. Since this does not increase latency of each operation it is simple to increase computational throughput.

When considering the characteristics of neural networks [1], decomposable units have three constraints that disadvantage them: 1) Decomposable units are typically constrained to power-of-2 precisions, meaning they cannot yield all of the potential benefit of variable reduced precision. For example, a layer requiring only 9 bits would still need to use 16. 2) Decomposable units require both inputs to be the same width. In the case of neural networks these are the weights and the neurons, weights typically require more than 8 bits [1] and as a result many layers will not see improvement. 3) Finally, if the baseline multipliers and adder trees were decomposable, the largest precision of the data and weights would have to be chosen for each layer. Section VI-F demonstrates that *STR* outperforms an variant of the *DaDN* design with ideal decomposable units.

VI. Evaluation

This section evaluates *STR*'s performance, energy and area and explores the trade-off between accuracy and performance. It also compares *STR* with a parallel, decomposable compute engine. This section mostly focuses on the execution of convolutional layers, and reports overall network performance at the end.

A. Methodology

Numerical Representation Requirements Analysis: The best per layer precision profiles are found via the methodology of Judd et al. [1] . Caffe [9] was used to measure how reducing the precision of each convolution layer affects the network's overall *top-1* prediction accuracy over 5000 images, that is, how often the network correctly classifies the input. The network definitions and pre-trained synaptic weights are taken from the Caffe Model Zoo [18].

While Judd et al., considered fixed point numbers as having $I \geq 0$ integer and $F \geq 0$ fractional bits, we explore dropping some of the less significant integer bits by parameterizing numbers as the MSB bit position, M, relative to the binary point, and the number of bits, N. This is an exponential search space problem with $M, N \in [0, 16]$ per layer and multiple layers. Our heuristic approach was: 1) Find the best per layer M profile using gradient descent, iteratively decreasing M by one bit, one layer at a time. 2) Given a fixed M-profile, explore the space of N-profiles, again using gradient descent. In both steps per layer analysis is used to determine a good starting point. Table I reports the corresponding results. Further exploration or a different search strategy may lead to better profiles.

Performance, Area and Energy: Both the *DaDN* and *STR* systems were modelled using the same methodology for consistency. A custom cycle-accurate simulator models execution time. Computation was scheduled as described by Chen et al. [19]. The logic components of the both systems were synthesized with the Synopsis Design Compiler [20] and laid out with Cadence Encounter with the TSMC 65nm library. The circuit is clocked at 980MHz. Post layout power and area measurements are reported. Power estimation was done using activity from gate level simulations with 100 sets of random neuron and synapse input values. The NBin and NBout SRAM buffers were modelled using CACTI [21]. The eDRAM area and energy were modelled with *Destiny* [22]. Both *Single* and *Batch* (each layer computed concurrently for multiple images) runs are performed. The batch size is chosen such that all images fit inside NM to avoid spilling neurons off-chip which would penalize performance and energy in both *STR* and *DaDN*.

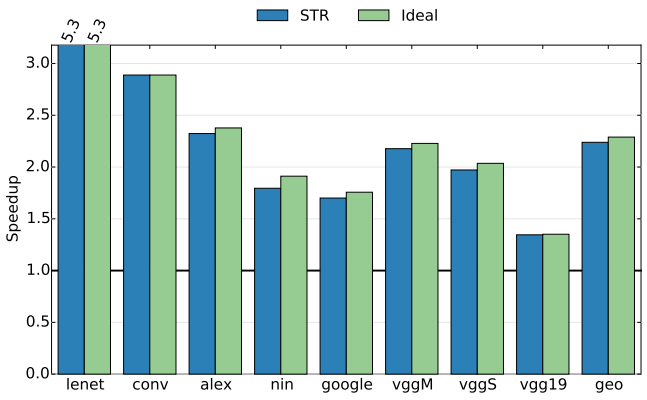


Fig. 7: Speedup for precision profiles with 100% relative accuracy over *DaDN* and the ideal speedup.

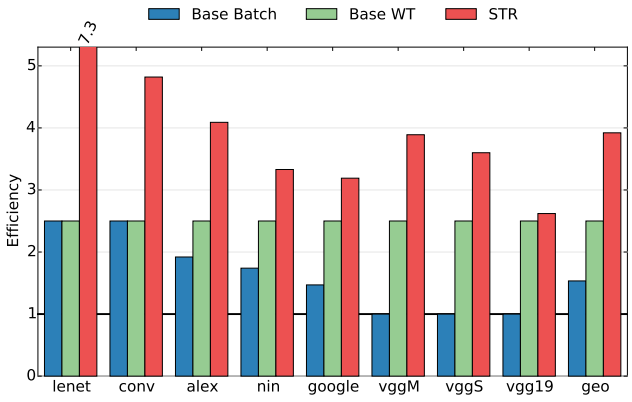


Fig. 8: Energy efficiency. Comparing *STR* with 100% profiles to *DaDN* batch mode and *DaDN* with window tiling normalized to *DaDN* in single image mode.

B. Performance Improvement

Figure 7 reports *STR*'s performance relative to *DaDN* for the precision profiles in Table I. Since batch processing does not impact the performance of convolutional layers in either *DaDN* or *STR*, the reported results are applicable to both. For the 100% profile, where no accuracy is lost, *STR* yields, on average, a speedup of 2.24x over *DaDN*. In the best case, LeNet which requires only 3 bits of precision in each layer the speedup is 5.33x, whereas VGG19 exhibit the least speedup, 1.35x, mostly due to its high precision requirements. In general, performance improvements follow the reduction in precision and are in line with the ideal speedup in Table I. The differences are due to the neuron lane under-utilization, which in the worst case is 7% (NiN). On average *STR* achieves a speedup that is within 2% of the ideal.

C. Area Overhead

Over the full chip, *STR* requires $122.1mm^2$, compared to $92.4mm^2$ in *DaDN*, an overhead of 32%. The Dispatcher accounts for 1% of the total area. Considering a single tile, *STR* increases area by 35%. While the 256 SIPs per tile in *STR* increase area by 107% compared to *DaDN*'s 16 IP units, the SB takes up a significant portion of the tile area resulting in the much lower per tile and overall chip area overhead.

D. Energy Efficiency Improvement

This section compares the energy efficiency of *STR* and *DaDN*. *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. To facilitate direct comparisons across all cases we use the energy of *DaDN* in single image mode as the numerator in all efficiency measurements reported in Figure 8.

Focusing on single image mode, the average efficiency improvement with *STR* across all networks for the 100% profiles is 3.92x, ranging from 7.27x in the best case (LeNet) to 2.62x in the worst case (VGG19). Ignoring secondary overheads, efficiency primarily depends on the reduction in precision length per layer since the energy savings are primarily derived from processing fewer neuron bits. Secondarily, the energy savings come from reducing the number of SB accesses which in *STR* occur only every p cycles.

Motivated by the reuse of synapses over multiple windows in *STR*, we evaluate an improved processing schedule for *DaDN* that interleaves the processing of multiple windows, via tiling in the scheduler, to match the synapse reuse seen in *STR*. The "BASE WT" bars report the efficiency of this *window tiling* approach which proves more efficient than the originally suggested schedule, but is less efficient than *STR*.

Batching improves energy efficiency in *DaDN* as synapses are reused over multiple images and thus SB reads are less frequent. However, the benefits from processing fewer neuron bits in *STR* far exceed those from synapse reuse. Window tiling in the *DaDN* improves efficiency by 2.50x. Since window tiling is not restricted by the size of NM, it allows for larger energy savings compared to batching in *DaDN*. When compared to the most efficient schedule in the baseline, *STR* is 57% more energy efficient.

E. Trading Accuracy for Performance

This section considers an approximate computing approach to improve performance by lowering precisions to the point where they start affecting overall network accuracy. By using serial computation and per layer neuron precisions *STR* enables the ability to fine-tune the accuracy and performance trade-off. Since performance does not depend on whether batching or single image mode is used, the results in the remaining sections are applicable to either processing mode.

Figure 9 shows the trade-off between network accuracy and speedup. The graph plots performance *relative* to the 100% profile performance of Figure 7 in order to show the performance vs. accuracy trade-off more clearly than normalizing over *DaDN*. Each point in the graph corresponds to a precision profile on the Pareto frontier of accuracy vs. performance. Attention is limited to profiles above 90% accuracy relative to the baseline, since accuracy drops off quickly below 90%

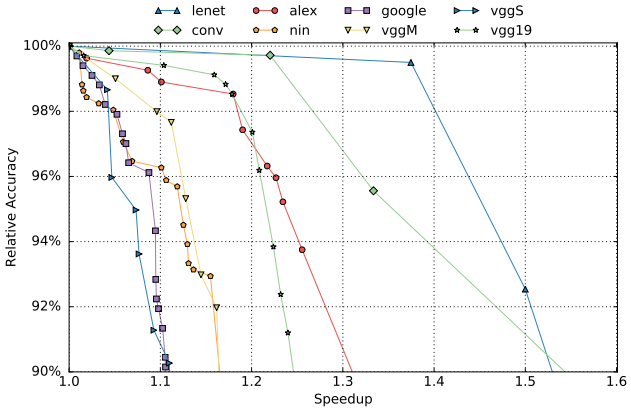


Fig. 9: *Additional* performance improvements possible when an accuracy loss is acceptable. Speedup is reported over the *STR* 100% relative accuracy profile.

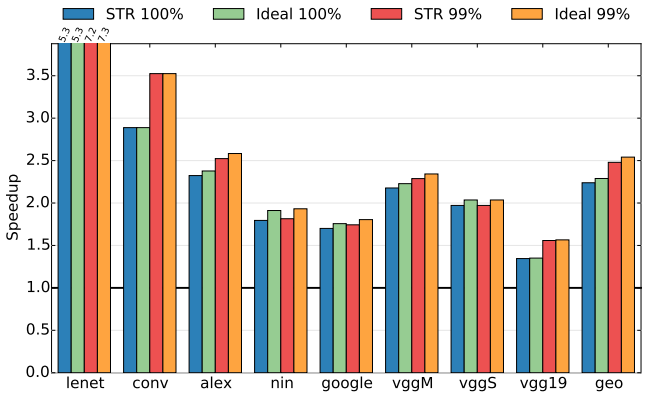


Fig. 10: Speedup for precision profiles with 100% and 99% relative accuracy over *DaDN* and the corresponding ideal speedups.

[1]. In all cases, additional performance improvements are possible with reduced accuracy, and the networks exhibit a knee in the accuracy vs. performance curve past which they incur rapid performance losses. The rest of this section focuses on the performance improvement possible when an up to 1% accuracy loss is acceptable.

Figure 10 compares the speedup of the 99% and 100% precision profiles from Table I relative to *DaDN*. By tolerating up to 1% *relative* prediction error, the average speedup increases to 2.48x, an incremental speedup of 11%. Speedups for the individual networks range from 1.56x for VGG19 to 7.23x for LeNet and generally follow the reduction in precision lengths. NiN benefits the most as it is able to use much smaller precisions in each layer when the accuracy constraint is loosened. On average, the 99% profiles increase efficiency to 68% over the most efficient baseline configuration.

With the 99% profile for LeNet, *STR* encounters the only instance of Dispatcher stalls. In this case, the precision of the first layer is 2 bits, thus the buffer is drained in 2 cycles. For some sets of window data in NM the Dispatcher needs more than 2 cycles to read the data, causing the dispatcher to stall. However, this situation is uncommon and only causes stalls for 2% of the total runtime. In all other cases *STR* is able to dispatch serial data continuously.

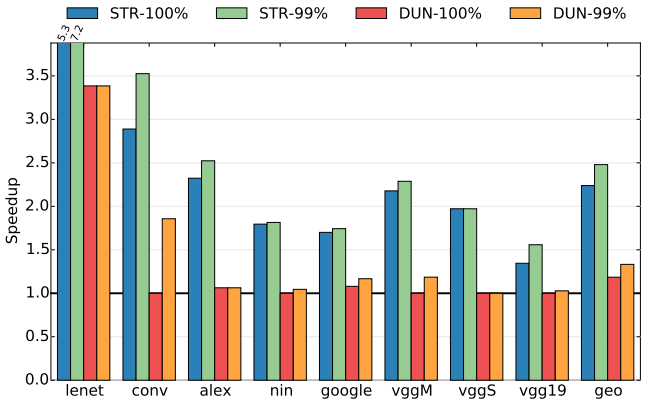


Fig. 11: Speedup for Serial vs. Decomposable compute units, for 100% and 99% precision profiles.

F. Decomposable Compute Units

This section compares *STR* to *DUN*, an ideal decomposable variance of *DaDN*, as described in Section V-I and which supports all power of 2 representation lengths up to 16. For this comparison it is assumed that the NFU utilization is the same for all precisions in DUN, e.g., a layer that performs 256 multiplications at 16 bits will perform 512 multiplications at 8 bits. In practice utilization will be worse for some layers due to the alignment constraints imposed by *DaDN*.

Figure 11 compares the speedup achieved by *STR* and the ideal DUN. With no accuracy loss DUN achieves 1.13x speedup vs. 2.29x for *STR* on average, while when an up to 1% accuracy loss is allowed DUN average speedup is 1.27x vs. 2.54x for *STR*. DUN is limited to profiles where the precision of each layers is a power of two and the same for both neurons and synapses. The size constraints of DUN severely limit its performance benefits even under ideal assumptions.

G. Overall Performance

Figure 12 shows the overall speedup of *STR* for each network when all convolutional, pooling and fully connected layers are processed. Recall that *DaDN* spends 85% of the time processing convolution layers, and for the other layers, *STR* sees no speedup. As a result, for the 100% profiles, *STR* improves overall network speed by 1.92x on average, from 1.3x for VGG_19 to 4.51x for LeNet. AlexNet, VGG_M and VGG_S do not see much of the speedup from the convolution layers since roughly a third of their time is spent loading synapses from off chip for their large fully connected layers. For the 99% profiles *STR* improves overall network speed by 2.08x on average, from 1.48x for VGG_19 to 5.73x for LeNet.

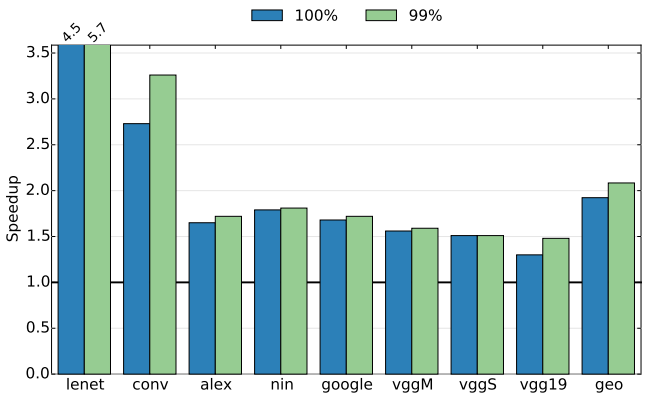


Fig. 12: Overall network speedup for 100% and 99% profiles.

VII. Related Work

Serial computation, reduced precision computation, and neural network accelerator designs have been active areas of research. To the best of our knowledge *STR* is the first work to combine all three to improve performance on modern DNNs.

Serial Computation: Serial computation has been used extensively in research and practice. For example, Hillis' Connection Machine uses serial data paths, which reduces unit throughput but decreases area, increasing multi-unit parallelism [23]. Distributed Arithmetic (DA) [24], [25], [14] uses an energy efficient bit-serial implementation of simpler convolutions using small filters such as the 2D Discrete Cosine Transform with a single 4×4 filter. Due to the small filter size, DA precomputes all the possible filter weight combinations in a 2^{16} entry table. DA calculates an inner-product with 4×4 values by concatenating the same position bit from all input values, indexing the table and adding the table values over all bit positions over multiple cycles. DNN filters typically contain thousands of synapses making this method impractical.

Rigid custom logic neural networks implementations use bit-serial computation but they hardwire their layers and their synapses [26]. Salapura et al., use bit-serial computation for neural networks but use $n-1$ bits for a values up to n, as opposed to $log_2(n)$ bits, making it slower than 2's complement bit-serial multiplication [27].

Reduced Precision Computation: A common approach to exploiting reduced precision is to simply turn off upper bit paths improving energy [28], however this does not directly improve performance. Kim et al. [29] use reduced precision neurons (8 and 4 bits) and weights (2 bits) to reduce the circuit size and memory requirements of a fixed hardware realization of a neural network for phoneme recognition. Sinha et al., use reduced precision in the serial input of DA to improve performance in 2D image processing [24]. This precision is determined dynamically by detecting the magnitude of the serial input to compute only the useful bits. *STR* exploits DNN properties to determine the precision at a larger granularity (per-layer) and offline avoiding any runtime overhead. Furthermore, *STR* achieves smaller precisions since it removes from the MSB and the LSB, introducing error in the individual data values but still producing an accurate prediction.

DNN Accelerators: Additional DNN accelerator designs have been recently demonstrated. PuDianNao supports seven machine learning algorithms including DNNs [30], and ShiDianNao is a camera-integrated low power accelerator [31]. Eyeriss [32] is a low power, real-time DNN accelerator that exploits zero valued neurons by using run length coding for memory compression and data gating zero neuron computations to save power.

The Efficient Inference Engine (EIE) [33] exploits an efficient sparse matrix filter representation [34]. EIE targets fully connected layers which have a large number of synapses used each only once per image yielding 12x better efficiency on FC layers than *DaDN*, while being 2x worse on convolution layers. Origami [35] presents a small, energy efficient accelerator that yields 803 GOp/s/W.

While each of these accelerators offset similar or better efficiency compared to *DaDN*, their performance is much lower. It has not been demonstrated how these architectures would scale to match the performance of *DaDN*. As such we chose *DaDN* as our baseline for a high performance accelerator. However, incorporating *STR* concepts in other accelerators or GPUs is interesting future work which this study motivates.

Software/Hardware Mixed Precision Approaches: Other work on reduced precision neural networks uses hardware-software co-design to design networks with more convenient reduced precision data types such as 8- and 4-bit and even 1-bit [2], [36], [10], [29], [37], [38]. By modifying the network architecture and training the network with a target precision these approaches can reduce the precision further without sacrificing accuracy.

Further improvement may be possible by combining these approaches with *STR* to allow finer control of the precision profile. *STR* is a software agnostic approach that does not constrain the design or training of these networks and has the flexibility to adapt to off-the-shelf, pre-trained networks.

VIII. Conclusion

Stripes demonstrates how reduced, per layer precision can be used to improve performance and energy for DNNs. *Stripes*' approach enables support for per-layer, dynamically configurable, fine-grained numerical precision which translates directly to performance and energy benefits and enables dynamic trade offs between accuracy and performance/energy.

Demonstrating *Stripes* as a modification over a high-performance DNN accelerator illustrates the validity of the underlying approach: exploit precision variability via bit-serial inner-product units while taking advantage of parallelism to maintain compute bandwidth when the highest precision is required. This work serves as motivation for applying the same concepts over other accelerators or general purpose compute engines such as GPUs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and suggestions. We also thank the Toronto Computer Architecture group members for their feedback. This work was supported by an NSERC Discovery Grant, an NSERC Discovery Accelerator Supplement and an NSERC CGS-D3 Scholarship.

REFERENCES

[1] 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.

[2] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 2015, pp. 1131–1135.

[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*, Dec 2014, pp. 609–622.

[4] 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.

[5] 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.

[6] 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.

[7] AMD, "AMD GRAPHICS CORES NEXT (GCN). Whitepaper." "https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf", 2012.

[8] I. Buck, "NVIDIA's Next-Gen Pascal GPU Architecture to Provide 10X Speedup for Deep Learning Apps," "http://blogs.nvidia.com/blog/2015/03/17/pascal/", 2015.

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[10] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015.

[11] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014.

[12] A. Peleg and U. Weiser, "Mmx technology extension to the intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42–50, Aug 1996.

[13] 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.

[14] S. White, "Applications of distributed arithmetic to digital signal processing: a tutorial review," *IEEE ASSP Magazine*, vol. 6, no. 3, pp. 4–19, Jul. 1989.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385 [cs]*, Dec. 2015, arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385

[16] D. Tan, A. Danysh, and M. Liebelt, "Multiple-precision fixed-point vector multiply-accumulator using shared segmentation," in *16th IEEE Symposium on Computer Arithmetic, 2003. Proceedings*, Jun. 2003, pp. 12–19.

[17] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, "Low-Cost Binary128 Floating-Point FMA Unit Design with SIMD Support," *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 745–751, May 2012.

[18] Y. Jia, "Caffe model zoo," *https://github.com/BVLC/caffe/wiki/Model-Zoo*, 2015.

[19] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014.

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

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

[22] 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.

[23] W. D. Hillis, "The connection machine," Thesis, Massachusetts Institute of Technology, 1985, 00682. [Online]. Available: http://dspace.mit.edu/handle/1721.1/14719

[24] A. Sinha and A. Chandrakasan, "Energy efficient filtering using adaptive precision and variable voltage," in *ASIC/SOC Conference, 1999. Proceedings. Twelfth Annual IEEE International*, 1999, pp. 327–331.

[25] T. Xanthopoulos and A. Chandrakasan, "A low-power DCT core using adaptive bitwidth and arithmetic activity exploiting signal correlations and quantization," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 5, pp. 740–750, May 2000.

[26] T. Szabo, L. Antoni, G. Horvath, and B. Feher, "A full-parallel digital implementation for pre-trained NNs," in *IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, 2000*, vol. 2, 2000, pp. 49–54 vol.2.

[27] V. Salapura, "Neural networks using bit stream arithmetic: a space efficient implementation," in *, 1994 IEEE International Symposium on Circuits and Systems, 1994. ISCAS '94*, vol. 6, May 1994, pp. 475–478 vol.6.

[28] 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, no. 5, pp. 787–793, May 2010.

[29] 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.

[30] 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*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 369–381, puDianNao. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694358

[31] 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)*, Jun. 2015, pp. 92–104, shiDianNao.

[32] 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.

[33] 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

[34] 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

[35] L. Cavigelli and L. Benini, "A 803 gop/s/w convolutional network accelerator," 2016.

[36] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights #x002b;1, 0, and #x2212;1," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2014, pp. 1–6.

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

[38] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural Networks with Few Multiplications," *arXiv:1510.03009 [cs]*, Oct. 2015, arXiv: 1510.03009.