2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing

Jorge Albericio[1] Patrick Judd[1] Tayler Hetherington[2]
Tor Aamodt[2] Natalie Enright Jerger[1] Andreas Moshovos[1]

[1] University of Toronto [2] University of British Columbia

***Abstract*—This work observes that a large fraction of the computations performed by Deep Neural Networks (DNNs) are intrinsically ineffectual as they involve a multiplication where one of the inputs is zero. This observation motivates *Cnvlutin* (*CNV*), a value-based approach to hardware acceleration that eliminates most of these ineffectual operations, improving performance and energy over a state-of-the-art accelerator with no accuracy loss. *CNV* uses hierarchical data-parallel units, allowing groups of lanes to proceed mostly independently enabling them to skip over the ineffectual computations. A co-designed data storage format encodes the computation elimination decisions taking them off the critical path while avoiding control divergence in the data parallel units. Combined, the units and the data storage format result in a data-parallel architecture that maintains wide, aligned accesses to its memory hierarchy and that keeps its data lanes busy. By loosening the ineffectual computation identification criterion, *CNV* enables further performance and energy efficiency improvements, and more so if a loss in accuracy is acceptable. Experimental measurements over a set of state-of-the-art DNNs for image classification show that *CNV* improves performance over a state-of-the-art accelerator from 1.24× to 1.55× and by 1.37× on average without any loss in accuracy by removing zero-valued operand multiplications alone. While *CNV* incurs an area overhead of 4.49%, it improves overall *EDP* (Energy Delay Product) and ED^2P (Energy Delay Squared Product) on average by 1.47× and 2.01×, respectively. The average performance improvements increase to 1.52× without any loss in accuracy with a broader ineffectual identification policy. Further improvements are demonstrated with a loss in accuracy.**

I. Introduction

Deep Neural Networks (DNNs) are becoming ubiquitous thanks to their exceptional capacity to extract meaningful features from complex pieces of information such as text, images, or voice. For example, DNNs and in particular, Convolutional Neural Networks (CNNs) currently offer the best recognition quality versus alternative object recognition algorithms, or *image classification*. DNN are not new [1], but are currently enjoying a renaissance [2] in part due to the increase in computing capabilities available in commodity computing platforms such as general purpose graphics processors [3].

While current DNNs enjoy several practical applications, it is likely that future DNNs will be larger, deeper, process larger inputs, and used to perform more intricate classification tasks at faster speeds, if not in real-time. Accordingly, there is a need to boost hardware compute capability while reducing energy per operation [4] and to possibly do so for smaller form factor devices.

Given the importance of DNNs, recent work such as the DianNao accelerator family [5], [6] targets hardware acceleration of DNNs. The approach taken by these accelerators exploits the computation *structure* of DNNs. Our work is motivated by the observation that further opportunities for acceleration exist by also taking into account the *content* being operated upon. Specifically, Section II shows that on average 44% of the operations performed by the dominant computations in DNNs are products that are undoubtedly *ineffectual*; one of the operands is a zero and the result is reduced along with others using addition. The fraction of these operations does not vary significantly across different inputs suggesting that ineffectual products may be the result of intrinsic properties of DNNs. The corresponding operations occupy compute resources wasting time and energy across inputs. This result along with the voluminous body of work on valued-based optimizations in software (e.g., constant propagation) and hardware (e.g., cache deduplication [7]) for general purpose processing, motivates *value-based DNN acceleration* approaches in software and hardware.

We present *Cnvlutin*[1] (*CNV*), a DNN accelerator that follows a value-based approach to dynamically eliminate most ineffectual multiplications. The presented *CNV* design improves performance and energy over the recently proposed DaDianNao accelerator [6]. *CNV* targets the convolutional layers of DNNs which dominate execution time (Sections II and V-B).

DaDianNao takes advantage of the regular access pattern and computation structure of DNNs. It uses wide SIMD (single-instruction multiple-data) units that operate in tandem in groups of hundreds of multiplication lanes. Unfortunately, this organization does not allow the lanes to move independently and thus prevents them from "skipping over" zero-valued inputs. *CNV* units decouple these lanes into finer-grain groups. A newly proposed data structure format for storing the inputs and outputs of the relevant layers is generated on-the-fly and enables the seamless elimination of most zero-operand multiplications. The storage format enables *CNV* to move the decisions on which computations to eliminate off the critical path allowing the seamless elimination of work without experiencing control divergence in the SIMD units. The assignment of work to units is modified enabling units to

[1]The name is derived from removing the o's from convolution: Cønvølutiøn.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.11

be kept busy most of the time independently of the distribution of zeroes in the input. A simple work dispatch unit maintains wide memory accesses over the on-chip eDRAM buffers.

Once the capability to skip zero-operand multiplications is in place, more relaxed ineffectual operation identification criteria can be used enabling further improvements with no accuracy loss and to dynamically trade off accuracy for even further performance and energy efficiency improvements.

CNV's approach bears similarity to *density-time* vector execution [8] and related graphics processor proposals [9], [10], [11], [12], [13], [14] for improving efficiency of control-flow intensive computation. *CNV* directly examines data values rather than skipping computation based upon predicate masks. Owing to the application-specific nature of *CNV*, its proposed implementation is also simpler. As Section VI explains, *CNV* also bears similarity to several sparse matrix representations sharing the goal of encoding only the non-zero elements, but sacrificing any memory footprint savings to maintain the ability to perform wide accesses to memory and to assign work at the granularity needed by the SIMD units.

Experimental measurements over a set of state-of-the-art DNNs for image classification show that *CNV* improves performance over a state-of-the-art accelerator from 24% to 55% and by 37% on average by targeting zero-valued operands alone. While *CNV* incurs an area overhead of 4.49%, it improves overall Energy Delay Squared (ED^2) and Energy Delay (ED) by 2.01× and 1.47× on average respectively. By loosening the ineffectual operand identification criterion, additional performance and energy improvements are demonstrated, more so if a loss in accuracy is acceptable. Specifically, on average performance improvements increase to 1.52× with no loss of accuracy by dynamically eliminating operands below a per-layer prespecified threshold. Raising these thresholds further allows for larger performance gains by trading-off accuracy.

The rest of this manuscript is organized as follows: Section II motivates *CNV*'s value-based approach to acceleration for DNNs by reporting the fraction of multiplications where a runtime calculated operand is zero. Section III presents the key design choice for *CNV*, that of decoupling the multiplication lanes in smaller groups by means of an example. Section IV details the *CNV* architecture. Section V reports the experimental results. Section VI comments on related work and Section VII concludes.

II. Motivation: Presence of Zeroes in Inter-Layer Data

CNV targets the convolutional layers of DNNs. In DNNs, as Section V-B corroborates, convolutional layers dominate execution time as they perform the bulk of the computations [15]. For the time being it suffices to know that a convolutional layer applies several three-dimensional filters over a three dimensional input. This is an inner product calculation, that is, it entails pairwise multiplications among the input elements, or *neurons* and the filter weights, or *synapses*. These products are then reduced into a single output neuron using addition.

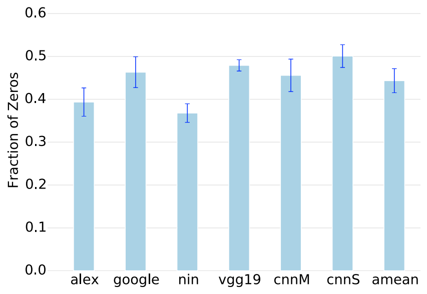


Fig. 1: Average fraction of convolutional layer multiplication input neuron values that are zero.

The key motivating observation for this work is that in practice, many of the neuron values turn out to be zero, thus the corresponding multiplications and additions do not contribute to the final result and could be avoided. Accordingly, this section characterizes the fraction of input neurons that are equal to zero in the convolutional layers of popular DNNs that are publicly available in Modelzoo [16]. For these measurements the DNNs were used to classify one thousand of images from the Imagenet dataset [17]. Section V-A details the networks and methodology followed.

Figure 1 reports the average total fraction of multiplication operands that are neuron inputs with a value of zero across all convolutional layers and across all inputs. This fraction varies from 37% for *nin*, to up to 50% for *cnnS* and the average across all networks is 44%. The error bars show little variation across input images, and given that the sample set of 1,000 images is sizeable, the relatively large fraction of zero neurons are due to the operation of the networks and not a property of the input.

But why would a network produce so many zero neurons? We hypothesize that the answer lies in the nature and structure of DNNs. At a high level, DNNs are designed so that each DNN layer attempts to determine whether and where the input contains certain learned "features" such as lines, curves or more elaborate constructs. The presence of a feature is encoded as a positive valued neuron output and the absence as a zero-valued neuron. It stands to reason that when features exist, most likely they will not appear all over the input, moreover, not all features will exist. DNNs detect the presence of features using the convolutional layers to produce an output encoding the likelihood that a feature exists at a particular position with a number. Negative values suggest that a feature is not present. Convolutional layers are immediately followed by a Rectifier, or ReLU layer which lets positive values pass through, but converts any negative input to zero.

While there are many zero-valued neurons, their position depends on the input data values, and hence it will be challenging for a static approach to eliminate the corresponding computations. In particular, there were *no* neurons that were always zero across all inputs. Even, if it was possible to

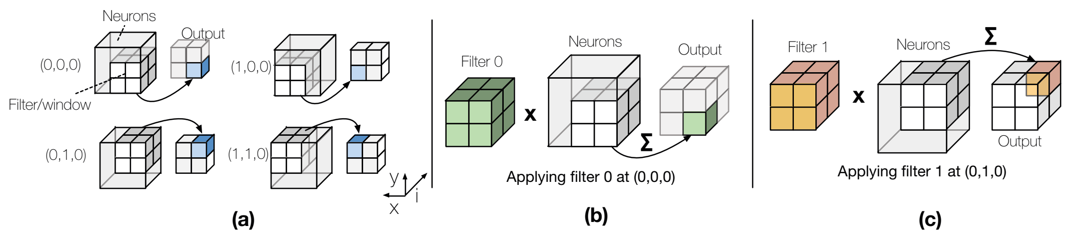


Fig. 2: The output neuron at position $(0,0,0)$ or $o(0,0,0)$ is produced by applying the filter on a $2\times2\times2$ *window* of the input with origin $n(0,0,0)$. Each synapse $s(x,y,z)$ is multiplied by the corresponding input neuron $n(x,y,z)$, e.g., $n(0,0,0)\times s(0,0,0)$, and $n(0,1,0)\times s(0,1,0)$, for a total of $2\times2\times2$ or eight products. The eight products are reduced into a single output neuron using addition. Then the window is slide over by S first along the X dimension to produce $o(1,0,0)$ using the neuron input window at origin $n(1,0,0)$. For example, now $s(0,0,0)$ is multiplied with $n(1,0,0)$ and $s(1,1,0)$ with $n(2,1,0)$. Once the first dimension is exhausted, then the window slides by S along the Y dimension and starts scanning along the X dimension again, and so on as the figure shows. In total, the result is a $2\times2\times1$ output neuron. The depth is one since there is only one filter. Parts (b) and (c) show a convolutional layer with two $2\times2\times2$ filters. The output now is a $2\times2\times2$ array, with each filter producing one of the two planes or layers of the output. As part (b) shows, the first filter produces output elements $o(x,y,0)$. Part (c) shows that the second filter produces output neurons $o(x,y,1)$.

eliminate neurons that were zero with high probability, there would not be many. For example, only 0.6% of neurons are zero with 99% probability. Accordingly, this work proposes an architecture that detects and eliminates such computations at runtime.

Since the time needed to compute a convolutional layer increases mostly linearly with the number of elements processed and since convolutional layers dominate execution time, these measurements serve as an upper bound on the potential performance improvement for an architecture that manages to skip the computations corresponding to zero-valued neurons.

III. Enabling Zero Skipping: A Simplified Example

Having shown that many of the neurons are zero, this section explains the two key ideas behind *CNV* that enable it to skip over the corresponding computations: 1) lane decoupling, and 2) storing the input on-the-fly in an appropriate format that facilitates the elimination of zero valued inputs. Section III-A first describes the computations that take place in a convolution layer identifying those that could be avoided when the input neurons are zero. Since our goal is to improve upon the state-of-the-art, Section III-B describes a state-of-the-art accelerator architecture for DNNs whose processing units couple several groups of processing lanes together into wide SIMD units. Finally, Section III-C describes a basic *CNV* architecture which decouples the processing lane groups enabling them to proceed independently from one another and thus to skip zero-valued neurons. A number of additional challenges arise once the lane groups start operating independently. These challenges along with the solutions that result in a practical, simple *CNV* design are described in Section IV-B.

A. Computation of Convolutional Layers

The operations involved in computing a CNN are of the same nature as in a DNN. The main difference is that in the former, weights are repeated so as to look for a feature at different points in an input (i.e. an image). The input to a convolutional layer is a 3D array of real numbers of dimensions $I_x \times I_y \times i$. These numbers are the input data in the first layer and the outputs of the neurons of the previous layer for subsequent layers. In the remainder of this work, we will call them *input neurons*. Each layer applies N filters at multiple positions along x and y dimensions of the layer input. Each filter is a 3D array of dimensions $F_x \times F_y \times i$ containing *synapses*. All filters are of equal dimensions and their depth is the same as the input neuron array's. The layer produces a 3D output neuron array of dimensions $O_x \times O_y \times N$. The output's depth is the same as the number of the filters.

To calculate an *output neuron*, one filter is applied over a *window*, or a subarray of the input neuron array that has the same dimensions as the filters $F_x \times F_y \times i$. Let $n(x,y,z)$ and $o(x,y,z)$ be respectively input and output neurons, and $s^f(x,y,z)$ be synapses of filter f. The output neuron at position $(\mathbb{k},\mathbb{l},\mathbb{f})$, before the activation function, is calculated as follows:

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

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 $O_x = (I_x - F_x)/S + 1$, and $O_y = (I_y - F_y)/S + 1$. Figure 2 shows a example with a $3\times3\times2$ input neuron array, a single $2\times2\times2$ filter and unit stride producing an output neuron array of $2\times2\times1$.

When an input neuron is zero the corresponding multiplication and addition can be eliminated to save time and energy without altering the output value.

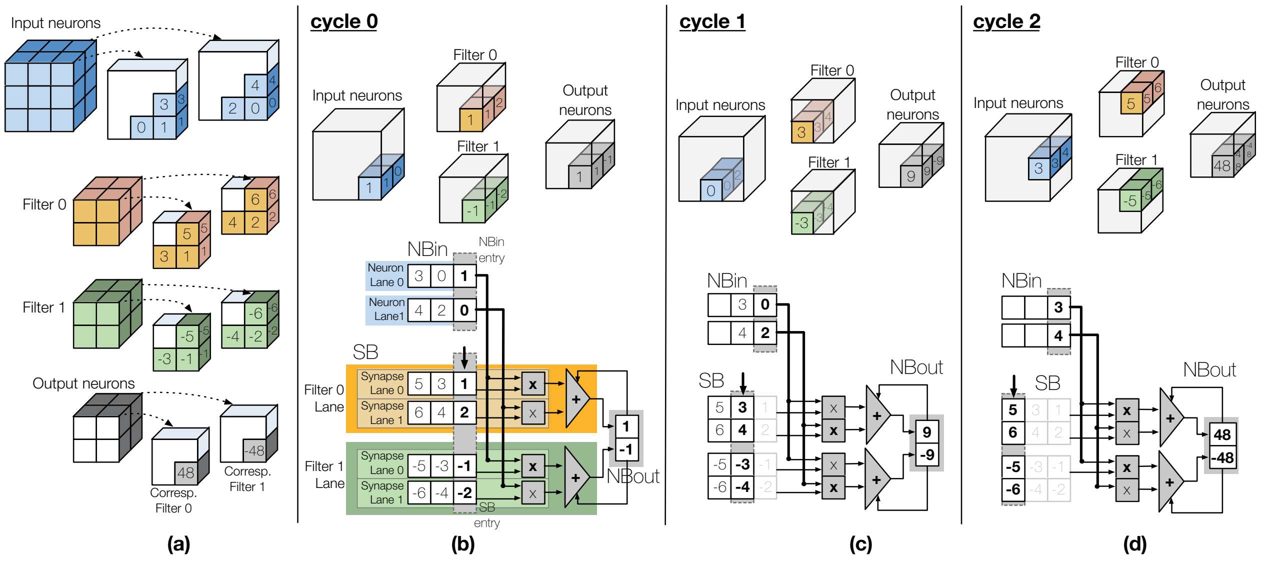


Fig. 3: Simplified state-of-the-art DNN unit: Example. The calculation of the complete filter would take one additional cycle, only the first three cycles are shown here. The elements of both filters have the same values with opposite signs only for the sake of clarity. **a)** A partial set of input neurons and synapses. **b)-d)** Three cycles of processing. Top part: which neurons and synapses are being processed. Bottom part: unit processing. **b)** Cycle 0: the first two neurons from NBin (1 and 0), are multiplied with the respective synapses of the two filters, ((1,2) and (-1,-2)), each product pair per filter is reduced through the adder and stored in NBout (1 and -1). The SB pointer advances by one and the neuron is discarded from NBin. **c)-d)** Cycles 1 and 2: The same sequence of actions for the next input neuron and filter synapse pairs. The NBout partial sums are read and used as extra inputs to the adder tree making progress toward calculating the final output neurons.

B. *The Simplified Baseline Architecture*

The baseline architecture is based on the DaDianNao state-of-the-art accelerator proposed by Chen *et al.* [6]. This section explains via an example how a simplified unit of this architecture calculates a convolutional layer and why, as it stands, it cannot skip over zero valued input neurons.

In Figure 3(a) a $3 \times 3 \times 2$ neuron array is convolved with unit stride by two $2 \times 2 \times 2$ filters producing a $2 \times 2 \times 2$ output neuron array. In Figure 3(b) the example unit comprises: 1) two *neuron* lanes, and 2) two *filter* lanes each containing two synapse *sublanes*. Each neuron lane and synapse sublane is fed respectively with a single element from an Input Neuron Buffer (NBin) lane and a Synapse Buffer (SB) lane. Every cycle, each neuron lane broadcasts its neuron to the two corresponding synapse sublanes resulting into four pairs of neurons and synapses, one per synapse sublane. A multiplier per synapse sublane multiplies the neuron and synapse inputs. An adder tree per filter lane reduces two products into a partial sum that accumulates into an Output Neuron Buffer (NBout) lane per filter.

Taking advantage of the structure of the layer computations, the unit couples all neuron and filter lanes so that they proceed in lock-step. This is an excellent decision if one considers only the structure of the computation assuming that most if not all computations ought to be performed. However, as is, this unit cannot skip over zero neurons. In our example, the zeros in both neuron lanes are unfortunately coupled with non-zero neurons. There are four multiplications that could be safely avoided potentially improving performance and energy.

C. *The Simplified Cnvlutin Architecture*

To exploit the significant fraction of zeroes in the neuron stream, we rethink the decision to couple all neuron lanes together. *CNV* decouples the neuron lanes allowing them to proceed independently. Figure 4 shows the equivalent simplified *CNV* design and how it proceeds over two cycles. The DaDianNao units are now split into 1) the back-end containing the adder trees and NBout, and 2) the front-end containing the neuron lanes, synapse sublanes, and multipliers. While the back-end remains unchanged, the front-end is now split into two *subunits* one per neuron lane. Each subunit contains one neuron lane and a synapse sublane from each of the two filters. Each cycle each subunit generates two products, one per filter. The products are fed into the two adder trees as before producing the partial output neuron sums. With this organization the neuron lanes are now capable of proceeding independently from one another and thus have the potential to skip over zeroes.

Instead of having the neuron lanes actively skip over zero neurons as they appear in the input, *CNV* opts for a dynamic hardware approach where the zero neurons are eliminated at the output of the preceding layer. As a result, only the non-zero

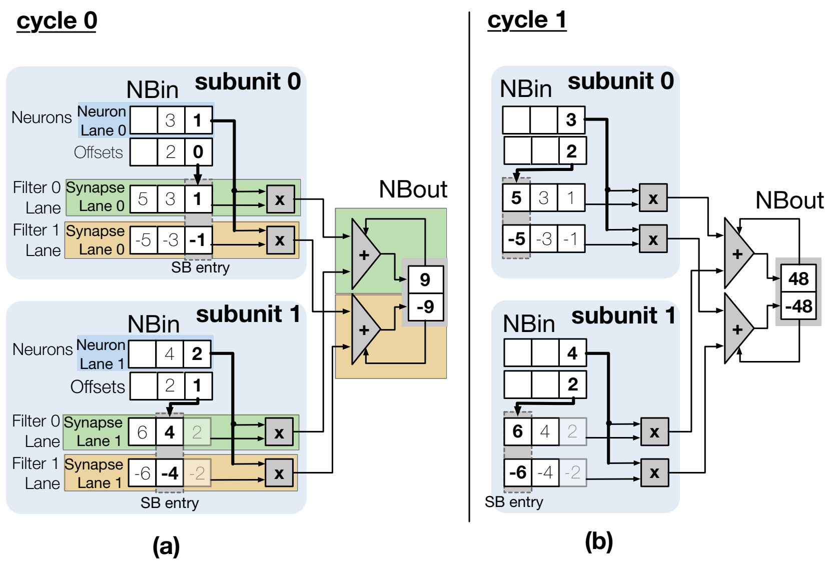


Fig. 4: The simplified *CNV* unit produces the same output as Figure 3 in just two cycles. The elements of both filters have the same values with opposite signs only for the sake of clarity. **a)** Organization details and Cycle 0: Subunit 0 reads the next NB neuron value 1 and its offset 0. Using the offset it indexes the appropriate SB synapses 1 and -1 corresponding to filter 0 and 1. The resulting products 1 and -1 are added to output neurons for the corresponding filters using the dedicated adder trees. Similarly subunit 1 will fetch neuron 2 with offset 1 and multiply with synapses 4 and -4 feeding the corresponding adder trees for the filters. **b)** Cycle 1: The operation repeats as before with subunit 0 fetching neuron 3 at offset 2 and subunit 1 fetching neuron 4 at offset 2. The same result as in the baseline (48, -48) is calculated in only two cycles.

neurons appear in the NBin. For this purpose, the input neuron array is stored in the Zero-Free Neuron Array format (ZFNAf) which Section IV-B1 describes in detail. Here we present a simplified version of this format explaining how it allows individual neuron lanes to see only the non-zero neurons proceeding independently from the other neuron lanes. Once the zero neurons are eliminated, the key challenge is matching each non-zero neuron with the appropriate SB entry. ZFNAf augments each non-zero input neuron with an offset for this purpose. For example, if the original stream of neurons would have been $(1,0,0,3)$ they will be encoded as $((1,0),(3,3))$. The offsets can adjust the SB sublane's index so that it can access the appropriate synapse column. *CNV* generates the ZFNAf on-the-fly as described in Section IV-B4.

Decoupling the neuron lanes and encoding the input neuron array are not the only techniques that are necessary to be able to improve performance over the baseline accelerator. Understanding these challenges and the design choices is easier if presented as modifications over the baseline architecture. Accordingly, the next section reviews the baseline architecture and operation, and then proceeds to present the *CNV* architecture.

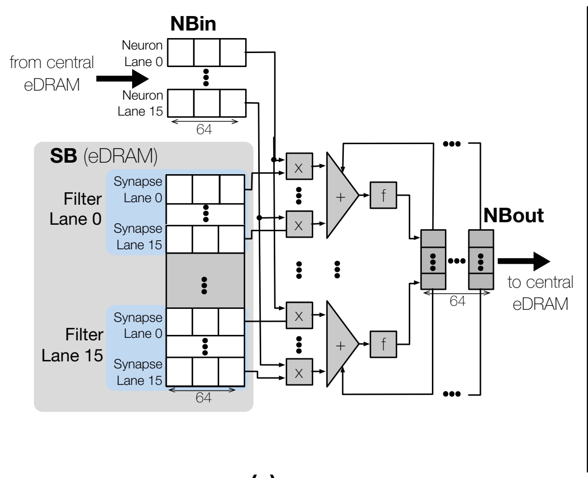
IV. The Cnvlutin Accelerator

Section IV-A completes the description of the baseline state-of-the-art accelerator *CNV* builds upon so that Section IV-B can present the complete *CNV* design and justify the relevant design choices.

A. Baseline Architecture

Section III-B described a simplified version of the baseline unit which was able to process two input neurons and two synapses of two filters at a time. *CNV* as presented in this work, builds upon *DaDianNao*, the state-of-the-art accelerator proposed by Chen *et al.* [6]. Each DaDianNao chip, or node, contains 16 *Neural Functional Units (NFUs)*, or simply *units*. Figure 5(a) shows one such unit. Each cycle the unit processes 16 input neurons, 256 synapses from 16 filters, and produces 16 partial output neurons. In detail, the unit has 16 neuron lanes, 16 filter lanes each with 16 synapse lanes (256 in total), and produces 16 partial sums for 16 output neurons. The unit's SB has 256 lanes (16×16) feeding the 256 synapse lanes, NBin has 16 lanes feeding the 16 neuron lanes, and NBout has 16 lanes. Each neuron lane is connected to 16 synapse lanes, one from each of the 16 filter lanes. The unit has 256 multipliers and 16 17-input adder trees (16 products plus the partial sum from NBout). The number of neuron lanes and filters per unit are design time parameters that could be changed. All lanes operate in lock-step.

DaDianNao is designed with the intention to minimize off-chip bandwidth and to maximize on-chip compute utilization. The total per cycle synapse bandwidth required by all 16 units of a node is 4K synapses per cycle, or 8TB/sec assuming a



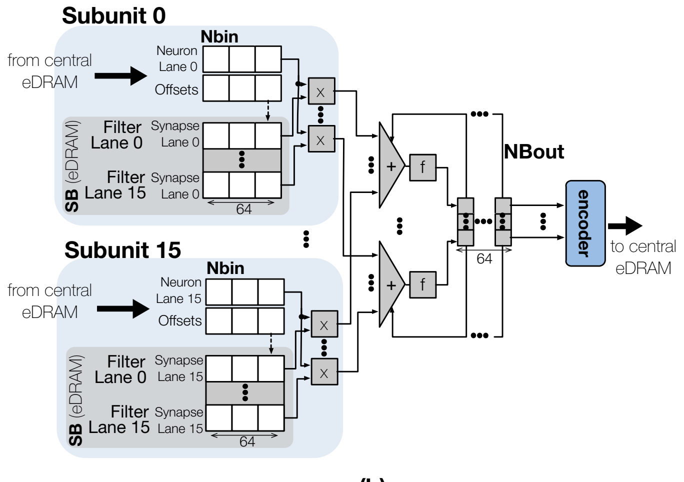


Fig. 5: Compute Units. **a)** DaDianNao NFU. **b)** *CNV* unit.

1GHz clock and 16-bit synapses. The total SB capacity is designed to be sufficient to store all synapses for the layer being processed (32MB or 2MB per unit) thus avoiding fetching synapses from off-chip. Up to 256 filters can be processed in parallel, 16 per unit. All inter-layer neuron outputs except for the initial input and final output are also stored in an appropriately sized central eDRAM, or *Neuron Memory* (NM). NM is shared among all 16 units and is 4MB for the original design. The only traffic seen externally is for the initial input, for loading the synapses once per layer, and for writing the final output.

Processing starts by reading from external memory: 1) the filter synapses, and 2) the initial input. The filter synapses are distributed accordingly to the SBs whereas the neuron input is fed to the NBins. The layer outputs are stored through NBout to NM and then fed to the NBins for 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. Multiple nodes can be used to process larger DNNs that do not fit in the NM and SBs available in a single node. NM and the SBs are implemented using eDRAM as the higher the capacity the larger the neurons and filters that can be processed by a single chip without forcing external memory spilling and excessive off-chip accesses.

1) Processing Order in DaDianNao: Figure 6(a) shows how the DaDianNao architecture processes an input neuron array applying 256 filters simultaneously. Each unit processes 16 filters, with unit 0 processing filters 0 through 15 and unit 15 processing filters 240 through 255. For simplicity, the figure only shows the position of the elements on the i dimension (for example, the position $(0,0,15)$ of filter 7 would be shown as s^7_{15}). Every cycle, a *fetch block* of 16 input neurons (each 16-bits long)f to all 16 units. The fetch block contains one neuron per synapse lane for each of the 16 filter lanes per unit. For example, in cycle 0, the fetch block will contain neurons $n(0,0,0)$ through $n(0,0,15)$. Neuron $n(0,0,0)$ will be multiplied in unit 0 with synapses $s^0(0,0,0)$ through $s^{15}(0,0,0)$, and with synapses $s^{240}(0,0,0)$ though $s^{255}(0,0,0)$ in unit 15. Neuron $n(0,0,1)$ is multiplied with synapses $s^0(0,0,1)$ though $s^{15}(0,0,1)$ in unit 0, and so on. The synapses are stored in the SBs in the order shown in the figure, so that the units can fetch the appropriate synapses in parallel. For example, the first entry (column) of SB in Unit 0 contains the following 256 synapses: $s^0(0,0,0)-s^0(0,0,15)$, ..., $s^{15}(0,0,0)-s^{15}(0,0,15)$.

Once the current window has been processed, the next window can be easily initiated since the location where the corresponding neurons start can be directly calculated given their coordinates. Since the window has to be processed by all filters, other work assignments are possible. The assignment chosen interleaves the input across lanes at a neuron level as it can also be seen on the figure. Since no attempt is made to skip over zeroes, a single 16-neuron wide fetch from NM can provide work for all lanes across all units achieving 100% utilization.

B. Cnvlutin Architecture

Figure 5(b) shows a *CNV* unit that offers the same computation bandwidth as a DaDianNao unit. As per the description of Section III-C, the front-end comprising the neuron lanes and the corresponding synapse lanes is partitioned into 16 independently operating subunits, each containing a single neuron lane and 16 synapse lanes. Each synapse lane processes a different filter for a total of 16. Every cycle, each subunit fetches a single $(neuron, offset)$ pair from NBin, uses the offset to index the corresponding entry from its SBin to fetch 16 synapses and produces 16 products, one per filter. The backend is unchanged. It accepts the 16×16 products from 16 subunits which are reduced using 16 adder trees. The adder trees produce 16 partial output neurons which the unit accumulates

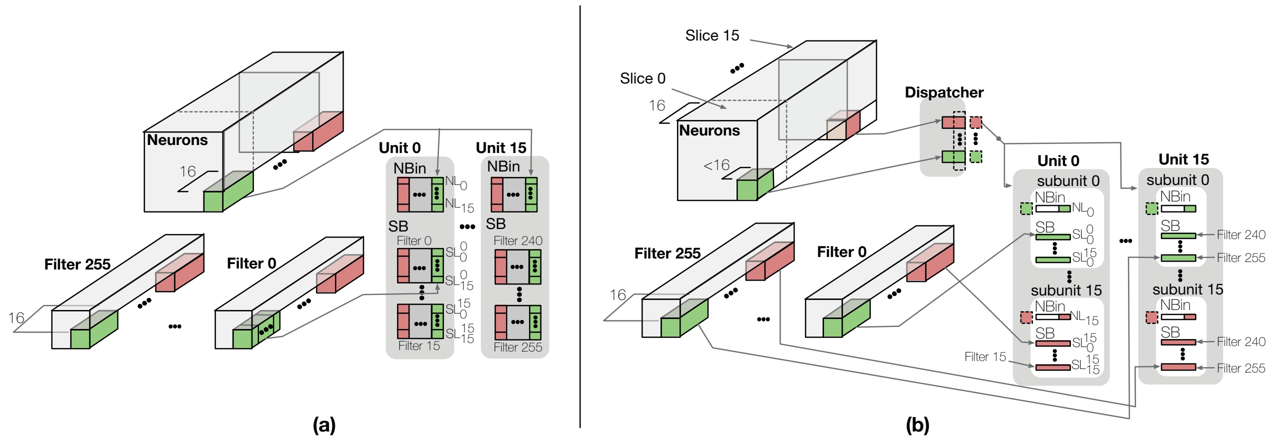


Fig. 6: Processing order and work assignment in **a)** baseline accelerator and **b)** *CNV*.

using 64 NBout entries. The subunit NBin is 64 entries deep with each entry containing a 16-bit fixed-point value plus an offset field. The total SB capacity remains at 2MB per unit as per the original DaDianNao design, with each subunit having an SB of 128KB. Each subunit SB entry contains 16×16 bits corresponding to 16 synapses. In summary, each subunit corresponds to a single neuron lane and processes 16 synapses, one per filter. Collectively, all subunits have 16 neuron lanes, 256 synapse lanes and produce 16 partial output neurons each from a different filter.

The *CNV* units can be used to process both encoded and conventional neuron arrays. A single configuration flag set by software for each layer controls whether the unit will use the neuron offset fields.

There are several challenges that have to be addressed for the *CNV* design to improve performance over the baseline: 1) Generating the encoded neuron arrays on-the-fly; 2) Keeping the units and all lanes busy; 3) Maintaining orderly, wide accesses to the central eDRAM. The key to overcoming these challenges lies in the format used to encode the input neuron arrays and in the way the work is divided across units. The rest of this section presents and justifies these choices. Section IV-B1 presents the format used for the input neuron array. Section IV-B2 explains how work is assigned across the various units and subunits. Section IV-B3 explains how *CNV* maintains wide NM accesses. Section IV-B4 presents how *CNV* generates the neuron array format used by the units on-the-fly. Finally, Section IV-B5 explains how the units process different windows.

1) The Zero-Free Neuron Array Format: Figure 7 shows the Zero-Free Neuron Array format (ZFNAf) that enables *CNV* to avoid computations with zero-valued neurons. As Section III-C explained, only the non-zero neurons are stored, each along with an offset indicating its original position. The ZFNAf allows *CNV* to move the decisions of which neurons to process off the critical path and to place them at the end of the preceding layer. Accordingly, the ZFNAf effectively implements what would have otherwise been control flow decisions.

The ZFNAf encoding bares some similarity to the Compressed sparse row (CSR) format [18]. However, CSR, like most sparse matrix formats that target matrices with extreme levels of sparsity have two goals: store only the non-zero elements and reduce memory footprint, ZFNAf only shares the first. In CSR, it is easy to locate where each row starts; however, to keep units busy, *CNV* allows direct indexing at a finer granularity sacrificing any memory footprint savings. Specifically, ZFNAf encodes neurons as $(value, offset)$ pairs in groups called *bricks*. Each brick corresponds to a fetch block of the DaDianNao design, that is an aligned, continuous along the input features dimension i group of 16 neurons, i.e., they all have the same x and y coordinates. Bricks are stored starting at the position their first neuron would have been stored in the conventional 3D array format adjusted to account for the offset fields and are zero padded.

This grouping has two desirable properties: 1) It maintains the ability to index into the neuron array at a brick granularity using just the coordinates of the first neuron of the brick. 2) It keeps the size of the offset field short and thus reduces the overhead for storing the offsets. The first property allows work to be assigned to subunits independently and also allows *CNV* to easily locate where windows start. As Section IV-B2 will explain, bricks enable *CNV* to keep all subunits busy and to proceed independently of one another and thus skip over zeroes or start processing a new window as needed. Figure 7 shows an example of the ZFNAf. Since *CNV* uses bricks of 16 neurons, the offset fields need to be 4-bit wide, a 25% capacity overhead for NM or 1MB for the studied configuration. Given that the bulk of the area is taken up by the SBs (32MB), overall the resulting area overhead proves small at 4.49% (Section V-C).

Section IV-B4 explains how the neuron arrays are encoded in ZFNAf on-the-fly. Before doing so, however, it is necessary to explain how *CNV* processes a neuron array encoded in this

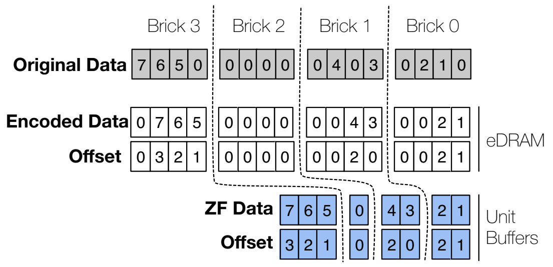


Fig. 7: Top: ZFNAf for 4-element bricks. *CNV* uses 16-element bricks. Bottom: NBin store format.

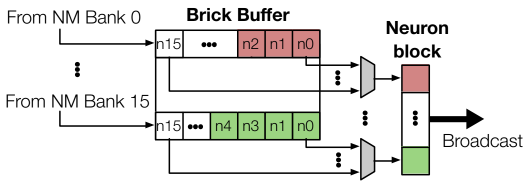


Fig. 8: Dispatcher

format.

2) Processing Order in CNV: As Section IV-A1 explained, DaDianNao fetches a single fetch block of 16 neurons per cycle which it broadcasts to all 16 units. This blocks contains work for all synapse lanes across 256 filters. Unfortunately, the same distribution of work across neuron lanes is not sufficient to keep all units busy for *CNV*. As Figure 6(b) shows, a fetch block in ZFNAf contains a single brick which with the baseline work assignment would contain work for all neuron lanes only if the corresponding original neuron array group contained no zero neurons.

In order to keep the neuron lanes busy as much as possible, *CNV* assigns work differently to the various neuron lanes. Specifically, while DaDianNao, as originally described, used a *neuron interleaved* assignment of input neurons to neuron lanes, *CNV* uses a *brick interleaved* assignment (which is compatible with DaDianNao as well). As Figure 6(b) shows, *CNV* divides the window evenly into 16 *slices*, one per neuron lane. Each slice corresponds to a complete vertical chunk of the window (all bricks having the same starting z coordinate). Each cycle, one neuron per slice is fetched resulting into a group of 16 neurons one per lane thus keeping all lanes busy. For example, let $e(x,y,z)$ be the $(neuron, offset)$ pair stored at location (x,y,z) of an input array in ZFNAf. In cycle 0, the encoded neurons at position $e(0,0,0)$, $e(0,0,16)$, ..., $e(0,0,240)$ will be fetched and broadcast to all units and processed by neuron lanes 0 through 15, respectively. As long as all 16 bricks have a second non-zero neuron, in cycle 1, $e(0,0,1)$, $e(0,0,17)$, ..., $e(0,0,241)$ will be processed. If, for example, brick 0 had only one non-zero neuron, in the next cycle the first neuron that will be fetched will be $e(1,0,0)$ assuming an input neuron depth i of 256.

Since each neuron lane proceeds independently based on how many non-zero elements each brick contains, it is necessary to have a different fetch pointer per neuron lane. A naïve implementation would perform 16 single neuron accesses per cycle, unduly burdening the NM. Section IV-B3 presents a simple extension that requires the same number of 16-neuron-wide and aligned NM accesses as DaDianNao.

Since the order in which the input neurons are assigned to neuron lanes in the units has changed, it is also necessary to change the order in which the synapses are stored in the SBs as the figure shows. For example, in cycle 0, if j is the offset of $e(0,0,0)$, Subunit 0 of Unit 0 will need $s^0(0,0,j)$ through $s^{15}(0,0,j)$, Subunit 15 of Unit 0, will need $s^{15}(0,0,240+j)$ through $s^{15}(0,0,240+j)$, and Subunit 0 of Unit 15 will need to $s^{240}(0,0,j)$ through $s^{255}(0,0,j)$. This proves to be equivalent to transposing the SB store order per subunit. Since the synapses are known in advance this rearrangement can be done statically in software. Thus, accessing the appropriate synapses in parallel per subunit is straightforward.

This work assignment does not change the output neuron values that each unit generates which remain identical to DaDianNao. The assignment only changes the order in which the input neurons are processed to produce an output neuron.

3) The Dispatcher: To avoid performing 16 independent, single-neuron-wide NM accesses per cycle, *CNV* uses a *dispatcher* unit that makes 16-neuron wide accesses to NM while keeping all neuron lanes busy. For this purpose, the subarrays the NM is naturally composed of are grouped into 16 independent banks and the input neuron slices are statically distributed one per bank. While the dispatcher is physically distributed across the NM banks, explaining its operation is easier if it is thought of as a centralized unit.

Figure 8 shows that the dispatcher has a 16-entry Brick Buffer (*BB*) where each entry can hold a single brick. Each BB entry is connected to one NM bank via a 16-neuron-wide bus and feeds one of the neuron lanes across all units via a single-neuron-wide connection. For example, *BB*[0] accepts neuron bricks from NM bank 0 and can broadcast any of its neurons to neuron lane 0 in all units. Initially, the dispatcher reads in parallel one brick from each bank for a total of 16 neuron bricks. In subsequent cycles, the dispatcher broadcasts the non-zero neurons, a single neuron from each BB entry at a time, for a total of 16 neurons, one per BB entry and thus per neuron lane each cycle. Before all the non-zero neurons of a brick have been sent to the units, the dispatcher fetches the next brick from the corresponding NM bank. To avoid stalling for NM's response, the fetching of the next in processing order brick per bank can be initiated as early as desired since the starting address of each brick and the processing order are known in advance. Since the rate at which each BB will drain will vary depending on the number of non-zero neurons encountered per brick, the dispatcher maintains a per NM bank fetch pointer.

The dispatcher may issue up to 16 NM requests concurrently, one per slice/bank. In the worst case, when bricks happen to have only zero valued neurons, an NM bank will have to supply a new brick every cycle. This rarely happens in practice, and the NM banks are relatively large and are sub-banked to sustain this worst case bandwidth.

In DaDianNao, a single 16-neuron wide interconnect is used to broadcast the fetch block to all 16 units. The interconnect structure remains unchanged in *CNV* but the width increases to accommodate the neuron offsets.

4) Generating the ZFNAf: The initial input to the DNNs studied are images which are processed using a conventional 3D array format. The first layer treats them as a 3-feature deep neuron array with each color plane being a feature. All other convolutional layers use the ZFNAf which *CNV* generates on-the-fly at the output of the immediately preceding layer.

In *CNV* as in DaDianNao, output neurons are written to NM from NBout before they can be fed as input to another layer. Since the eDRAM NM favors wide accesses, these writes remain 16 neurons wide. However, before writing to the NM, each 16-neuron group is encoded into a brick in ZFNAf. This is done by the *Encoder* subunit. One encoder subunit exists per *CNV* unit.

While *CNV* processes the input neuron array in an order different than DaDianNao, *CNV*'s units still produce the same output neurons as DaDianNao. Recall, that each output neuron is produced by processing a whole window using one filter. The assignments of filters to units remain the same in *CNV*. Accordingly, the output neurons produced by a *CNV* unit correspond to a brick of the output neuron array. All the encoder unit has to do, is pack the non-zero neurons within the brick.

The *Encoder* uses a 16-neuron input buffer (IB), a 16-encoded-neuron output buffer (OB), and an offset counter. Conversion begins by reading a 16-neuron entry from NBout into IB while clearing all OB entries. Every cycle the encoder reads the next neuron from IB and increments its offset counter. The neuron is copied to the next OB position only if it is nonzero. The current value of the offset counter is also written completing the encoded neuron pair. Once all 16 IB neurons have been processed, the OB contains the brick in ZFNMf and can be sent to NM. The same interconnect as in DaDianNao is used widened to accommodate the offset fields. The encoder can afford to do the encoding serially since: 1) output neurons are produced at a much slower rate, and 2) the encoded brick is needed for the next layer.

5) Synchronization: In DaDianNao, all units process neurons from the same window and processing the next window proceeds only after the current window is processed. *CNV* follows this approach avoiding further modifications to the unit's back-end and control. As neuron lanes process their bricks independently, unless all slices have exactly the same number of non-zero neurons, some neuron lanes will finish processing their window slice earlier than others. These neuron lanes will remain idle until all other lanes complete their processing.

Network	**Conv. Layers**	**Source**
alex	5	Caffe: bvlc_reference_caffenet
google	59	Caffe: bvlc_googlenet
nin	12	Model Zoo: NIN-imagenet
vgg19	16	Model Zoo: VGG 19-layer
cnnM	5	Model Zoo: VGG_CNN_M_2048
cnnS	5	Model Zoo: VGG_CNN_S

TABLE I: Networks used

V. Evaluation

This section evaluates the performance, area and power of the *CNV* architecture demonstrating how it improves over the state-of-the-art DaDianNao accelerator [6]. Section V-A details the experimental methodology. Section V-B evaluates the performance of *CNV*. Sections V-C and V-D evaluate the area and power of *CNV*, and Section V-E considers the removal of non-zero neurons.

A. Methodology

The evaluation uses the set of popular [3], and state-of-the-art convolutional neural networks [19][16][20][21] shown in Table I. These networks perform image classification on the ILSVRC12 dataset [19], which contains 256×256 images across 1000 classes. The experiments use a randomly selected set of 1000 images, one from each class. The networks are available, pre-trained for Caffe, either as part of the distribution or at the Caffe Model Zoo [22].

We created a cycle accurate simulator of the baseline accelerator and *CNV*. The simulator integrates with the Caffe framework [23] to enable on-the-fly validation of the layer ouput neurons. The area and power characteristics of *CNV* and DaDianNao are measured with synthesized implementations. The two designs are implemented in Verilog and synthesized via the Synopsis Design Compiler [24] with the TSMC 65nm library. The NBin, NBout, and *CNV* offset SRAM buffers were modeled using the Artisan single-ported register file memory compiler [25] using double-pumping to allow a read and write per cycle. The eDRAM area and energy was modeled with *Destiny* [26].

B. Performance

Figure 9 shows the speedup of *CNV* over the baseline. The first bar (*CNV*) shows the speedup when only zero neurons are considered, while the second bar (*CNV* + *Pruning*) shows the speedup when additional neurons are also skipped without affecting the network overall accuracy as Section V-E describes. The rest of this section focuses on the first bar.

On average *CNV* improves performance by 37%, at most by 55% (cnnS) and at least by 24% (google). The performance improvements depend not only on the fraction of zero-valued neurons but also on the fraction of overall execution time taken by the corresponding layers (*CNV* does not accelerate the first layer) and on the potential lost when subunits idle waiting for the current window to be processed by all others. While google exhibits a higher than average fraction of zero neurons, its first

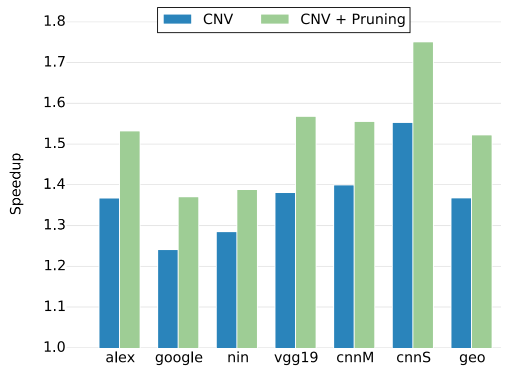


Fig. 9: Speedup of *CNV* over the baseline.

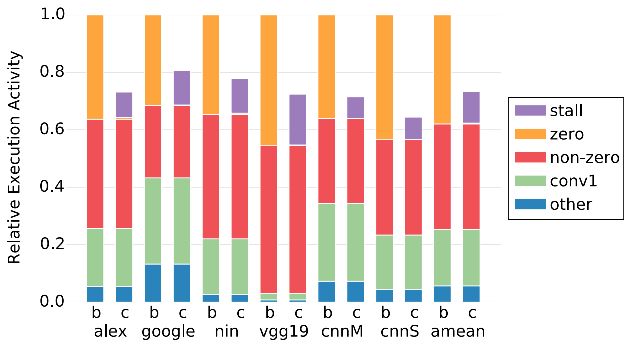


Fig. 10: Breakdown of Execution Activity (see main text for a definition). *CNV* (c) is normalized to the baseline (b).

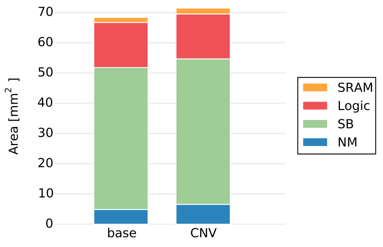


Fig. 11: Area Breakdown

layer has a relatively longer runtime than the other networks accounting for 35% of the total runtime vs. 21% on average as measured on the baseline. Google also spends a higher portion of its timing computing other layers.

The performance results for the networks can be better understood by looking at the breakdown of where time goes in the baseline (b) and *CNV* (c) per network as shown in Figure 10. Execution activity is divided into the following categories: 1) processing non-convolutional layers (other), 2) executing the first convolutional layer (conv1), 3) processing non-zero neurons (non-zero), 4) processing zero neurons (zero), and 5) idling (stall) as explained in Section IV-B5. It is not possible to assign time units, that is cycles, uniquely to each category; For example, during the same cycle in the baseline some neuron lanes may be processing zero neurons while others maybe processing non-zero ones. In addition, in *CNV* some neuron lanes may be idle waiting for all others to finish processing the current window. Accordingly, the figure reports a breakdown of *execution activity* which accounts for each neuron lane (equivalent to a subunit in *CNV*) separately per cycle. The total number of events accounted for is $units \times \frac{neuron\_lanes}{unit} \times cycles$, resulting in a metric that is directly proportional to execution time and that allows each event to be assigned to a single category.

The results corroborate that the convolutional layers which include the first layer, dominate execution activity across all networks on the baseline. The relatively small fraction of activity where *CNV* subunits are idle demonstrates that *CNV* manages to capture most of the potential that exists from eliminating zero-valued neurons.

C. Area

Figure 11 shows the area breakdown of the baseline and *CNV* architectures. Overall, *CNV* increases total area by only 4.49% over the baseline, a small overhead given the measured performance improvements. Area compares across the two architectures as follows: 1) The filter storage (SB) dominates total area for both architectures. While *CNV* partitions the SBin across subunits, the overhead for doing so is negligible as each chunk remains large (128KB per subunit). 2) *CNV* increases the neuron memory (NM) area by 34% since it a) requires 25% more storage storage for the offsets and b) uses 16 banks. 3) The additional cost of *CNV* in the unit logic is negligible. 4) *CNV* increases the SRAM area by 15.8%. This is due to the additional buffer space dedicated to the storage of the offsets.

D. Power

Figure 12 shows a breakdown of average power consumption in the baseline and *CNV*. Three bars are shown for each architecture corresponding to static, dynamic and overall power. Each category is further divided across the NM, SB, logic, and SRAM. The logic includes the datapath, control logic, the encoder and the memory dispatcher, while SRAM includes NBin and NBout. NM power is 53% higher in *CNV* than the baseline. This is expected, as NM is wider and banked. However, NM only accounts for 22% of the total chip power in the baseline so the overall power cost is small. The overhead of splitting the NBin and adding logic in the unit only increases its power by 2%. Reorganizing SB has little impact on its power cost and since synapses are not read when a subunit is stalled, the dynamic power of SB decreases by 18%. Overall, the 32MB of SB account for most of the total power consumption, and the savings in dynamic SB energy outweigh the overheads in NM, logic and SRAM. As a result, The power cost of *CNV* is 7% lower than the baseline on average.

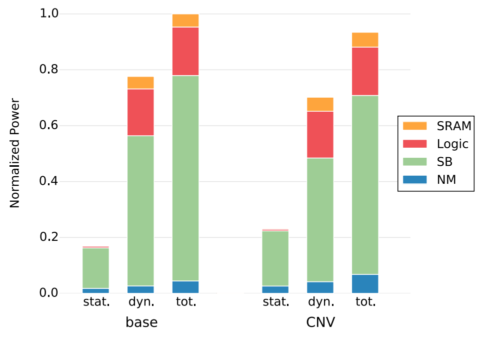


Fig. 12: Power Breakdown normalized to the baseline

Network	Thresholds per layer	Speedup
alex	8,4,8,16,8	1.53
nin	4,8,16,16,16,16,32,32,16,8,16,4	1.39
google	4,4,8,16,4,4,4,4,2,2,2	1.37
cnnM	8,2,4,4,2	1.56
cnnS	4,4,8,4,4	1.75
vgg19	8,4,16,64,64,64,64,128,256, 256,256,128,64,32,16,16	1.57

TABLE II: Lossless Ineffectual Neuron Thresholds

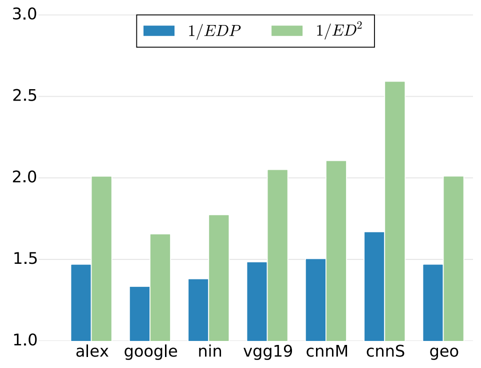


Fig. 13: Energy Delay Product and Energy Delay^2 Product Improvement of *CNV* over DaDianNao.

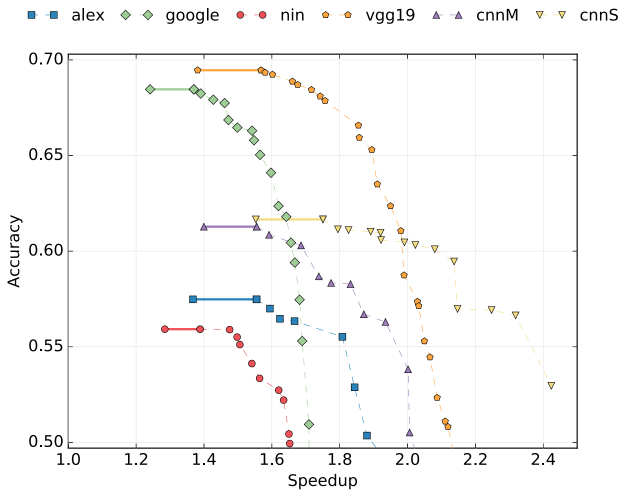


Fig. 14: Accuracy vs. Speedup trade-off from pruning neurons

1) EDP and ED^{2}P: This section reports the Energy-Delay Product (*EDP*) and Energy-Delay Squared Product (*ED^{2}P*) for the two architectures. While there is no unanimous consensus on how to properly compare two computing systems taking in consideration energy and performance, two commonly used metrics are the *EDP* [27] and *ED^{2}P* (*ET^2*) [28]. Figure 13 reports the *EDP* and *ED^{2}P* improvement of *CNV* over the baseline. On average *CNV*'s *EDP* improves by 1.47× and *ED^{2}P* by 2.01×.

E. Removing More Ineffectual Neurons

Pruning is a common computation reduction technique in neural networks that removes ineffectual synapses or neurons [29], [30]. *CNV*'s architecture allows for a form of *dynamic neuron pruning* by setting neuron values to zero so that their computation is skipped. To demonstrate this capability, this section considers a simple pruning approach where *near zero* neurons are set to zero when their magnitude is below a pre-specified, per-layer threshold. The baseline design incorporates comparators for max pooling which *CNV* reuses for threshold comparisons. The threshold value is determined in advance and is communicated with the layer meta-data, such as input dimensions, padding and stride.

To find a near optimal per-layer threshold configuration, exploration is done using gradient descent, similar to the approach used in previous work for finding per layer precision requirements [31]. For simplicity, power of two thresholds were explored, however, the hardware could support any fixed-point threshold. Network accuracy was measured across 5000 images from the ImageNet [17] validation set, sampled uniformly across each class.

Figure 14 shows the trade-off between accuracy (y-axis) and performance (x-axis) when neurons are dynamically pruned using per-layer thresholds. The pareto frontiers of the explored configurations for each network are shown. The leftmost point for each network corresponds to *CNV* in Figure 9 where only zero-valued neurons were removed. Generally, all networks exhibit an initial region where neurons can be pruned *without* affecting accuracy. This region is shown with a solid line in Figure 14. The maximum speedup without loss of accuracy is also reported as *(*CNV + Pruning) in Figure 9. Table II shows the the thresholds that yield the highest speedup without a loss in accuracy. On average, pruning increases the speedup to 1.52x, an increase of 11%. For google, thresholds are instead specified per 'inception module' [32].

For all networks, performance can improve further but at an accuracy *loss* with accuracy decaying exponentially with the performance improvement. For example, tolerating a drop in relative accuracy of up to 1% further increases the average performance improvement to 1.60× over the baseline, whereas allowing a drop in relative accuracy of up to 10% yields a 1.87× speedup over the baseline.

A limitation of the current study is that it does not prove that the specific ineffectual neuron identification thresholds generalize over other inputs. In particular, a concern is whether the neurons that are removed happen to not be excited for the given input set. This concern bears similarity to the general task that DNNs aim to tackle: classify previously unseen inputs using synapses learned over another set of images. To increase confidence in the conclusions drawn in this section, experiments were repeated with different input data sets and it was found that the specific accuracy vs. performance measurements do vary but not significantly.

VI. Related work

CNV bears similarities to related graphics processor proposals for improving efficiency of control-flow intensive computation [9], [10], [11], [12], [13], [14]. These works improve efficiency by filling idle SIMT/SIMD (single instrution multiple threads/single instruction multiple data) lanes caused by control-flow divergence with useful computation from other threads, whereas *CNV* replaces idle lanes known to produce zeroes with useful computation from later neurons. *Temporal SIMT* [12], [13], [14], which remaps spatially parallel SIMT execution groups temporally over a single SIMT lane, bears the most similarity to *CNV*. Temporal SIMT flips the warps on their side such that the threads from a single warp are executed on a single execution unit one after another. With this arrangement each execution unit operates on a separate warp, instead of all threads in a single warp. If there is no branch divergence, the whole warp (e.g., 32 threads) will execute this same instruction in the single execution unit. If there is branch divergence, then fewer threads will execute this instruction. In this way, branch divergence does not lead to idle SIMD lanes. Similarly, *CNV* removes zero, or idle, computations by rotating filter computations mapped spatially across multiple lanes temporally on a single lane. Computations that would have produced zeroes instead reduce the number of computations per window.

Qadeer *et al.* proposed the *Convolution Engine (CE)* [33], which in contrast to *CNV*, trades performance for a high degree of flexibility. CE is a system which can be programmed to target the convolution-like data-flow algorithms present in a wide range of applications.

Many previous works have looked at accelerating sparse vector matrix operations using FPGAs, GPUs, or other many-core architectures [34], [35], [36], [37], [38], [39], [40], [41]. Traditionally sparse matrices naturally appear in many engineering applications. There are several differences with the problem studied here and the approach followed. 1) The majority of these implementations operate on or modify one of the many different sparse matrix formats [42] which often incur a high per element overhead that is acceptable only for highly sparse matrices. Additionally, the matrices considered in these works exhibit very high sparsity, typically around 99%. While past work has evaluated alternative sparse storage formats, it still targets high sparsity matrices [36], [37]. Moreover, some of these representations exploit other matrix properties such as most of the values being on the diagonal, or being clustered, or parts of the matrix being symmetric. This work considered much lower sparsity between 40-50% zeroes (Figure 1) favoring a different representation and approach. 2) *CNV* is designed to operate on both encoded and conventional 3D arrays. 3) *CNV* is designed for the specific access and computation structure of convolutional layers of DNNs which differs from that of traditional engineering applications. Specifically, there is a difference in the number and size of the arrays being manipulated, in the sparsity and general matrix structure, and in where computations need to start at.

The Efficient Inference Engine (EIE) [43] performs inference using a recently proposed compressed network model [44] and accelerates the inherent modified sparse matrix-vector multiplication. Eyeriss [45] is a low power, real-time DNN accelerator that exploits zero valued neurons by using run length coding for memory compression. Eyeriss gates zero neuron computations to save power but it does not skip them as *CNV* does.

VII. Conclusion

Motivated by the observation that on average 44% of the run-time calculated neurons in modern DNNs are zero, this work advocates a value-based approach to accelerating DNNs in hardware and presents the *CNV* DNN accelerator architecture. While *CNV* is demonstrated as a modification over the state-of-the-art DNN accelerator DaDianNao, the key ideas that guided the *CNV* design can have broader applicability.

The *CNV* design serves as motivation for additional exploration such as combining *CNV* with approaches that exploit other value properties of DNNs. such as the variable precision requirements of DNNs [46]. Furthermore, *CNV*'s design principles and a valued-based approach can be applied in network training, on other hardware and software network implementations, or on other tasks such as Natural Language Processing.

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 PGS-D Scholarship.

References

[1] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 05 2015.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[4] 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.

[5] 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.

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

[7] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, “Last-level cache deduplication,” in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS ’14, (New York, NY, USA), pp. 53–62, ACM, 2014.

[8] J. E. Smith, G. Faanes, and R. Sugumar, “Vector instruction set support for conditional operations,” in *ISCA*, 2000.

[9] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, “Dynamic warp formation and scheduling for efficient GPU control flow,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 407–420, 2007.

[10] W. W. Fung and T. M. Aamodt, “Thread block compaction for efficient SIMT control flow,” in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 25–36, 2011.

[11] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, “Improving GPU performance via large warps and two-level warp scheduling,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 308–317, 2011.

[12] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovic, “Convergence and scalarization for data-parallel architectures,” in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pp. 1–11, 2013.

[13] R. Krashinsky, “Temporal simt execution optimization,” Feb. 14 2013. US Patent App. 13/209,189.

[14] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, “GPUs and the Future of Parallel Computing,” *Micro, IEEE*, vol. 31, no. 5, pp. 7–17, 2011.

[15] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, “An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices,” in *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, IoT-App ’15, (New York, NY, USA), pp. 7–12, ACM, 2015.

[16] M. Lin, Q. Chen, and S. Yan, “Network in network,” *CoRR*, vol. abs/1312.4400, 2013.

[17] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *arXiv:1409.0575 [cs]*, Sept. 2014. arXiv: 1409.0575.

[18] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2003.

[19] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, 2015.

[20] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, “Return of the devil in the details: Delving deep into convolutional nets,” *CoRR*, vol. abs/1405.3531, 2014.

[21] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014.

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

[23] 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.

[24] Synopsys, “Design Compiler.” http://www.synopsys.com/Tools/ Implementation/RTLSynthesis/ DesignCompiler/Pages/default.aspx.

[25] ARM, “Artisan Memory Compiler.” http://www.arm.com/products/physical-ip/embedded-memory-ip/index.php.

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

[27] R. Gonzalez and M. Horowitz, “Energy dissipation in general purpose microprocessors,” *Solid-State Circuits, IEEE Journal of*, vol. 31, pp. 1277–1284, Sep 1996.

[28] A. Martin, M. Nystrm, and P. Pnzes, “Et2: A metric for time and energy efficiency of computation,” in *Power Aware Computing* (R. Graybill and R. Melhem, eds.), Series in Computer Science, pp. 293–315, Springer US, 2002.

[29] Y. L. Cun, J. S. Denker, and S. A. Solla, “Optimal brain damage,” in *Advances in Neural Information Processing Systems*, pp. 598–605, Morgan Kaufmann, 1990.

[30] B. Hassibi, D. G. Stork, and G. J. Wolff, “Optimal Brain Surgeon and general network pruning,” in *, IEEE International Conference on Neural Networks, 1993*, pp. 293–299 vol.1, 1993.

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

[32] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” *CoRR*, vol. abs/1409.4842, 2014.

[33] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, “Convolution engine: Balancing efficiency and flexibility in specialized computing,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA ’13, (New York, NY, USA), pp. 24–35, ACM, 2013.

[34] L. Zhuo and V. K. Prasanna, “Sparse matrix-vector multiplication on FPGAs,” in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA ’05, (New York, NY, USA), pp. 63–74, ACM, 2005.

[35] M. deLorimier and A. DeHon, “Floating-point sparse matrix-vector multiply for FPGAs,” in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA ’05, (New York, NY, USA), pp. 75–85, ACM, 2005.

[36] S. Jain-Mendon and R. Sass, “A hardware–software co-design approach for implementing sparse matrix vector multiplication on FPGAs,” *Microprocessors and Microsystems*, vol. 38, no. 8, pp. 873–888, 2014.

[37] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, “FPGA based sparse matrix vector multiplication using commodity dram memory,” in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 786–791, IEEE, 2007.

[38] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos, “FPGA vs. GPU for sparse matrix vector multiply,” in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 255–262, IEEE, 2009.

[39] N. Bell and M. Garland, “Efficient sparse matrix-vector multiplication on CUDA,” tech. rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[40] F. Vazquez, G. Ortega, J.-J. Fernández, and E. M. Garzón, “Improving the performance of the sparse matrix vector product with GPUs,” in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 1146–1151, IEEE, 2010.

[41] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, “Efficient sparse matrix-vector multiplication on x86-based many-core processors,” in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 273–282, ACM, 2013.

[42] Y. Saad, “SPARSKIT: A basic tool kit for sparse matrix computation,” Tech. Rep. CSRD TR 1029, University of Illinois, 1990.

[43] 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,” *CoRR*, vol. abs/1602.01528, 2016.

[44] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding,” *CoRR*, vol. abs/1510.00149, 2015.

[45] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pp. 262–263, 2016.

[46] P. Judd, J. Albericio, and A. Moshovos, “Stripes: Bit-serial deep neural network computing,” *Computer Architecture Letters*, 2016.