Proceedings of the International Conference on Artificial Intelligence and Smart Systems (ICAIS-2021)
IEEE Xplore Part Number: CFP21OAB-ART; ISBN: 978-1-7281-9537-7

A Deep Neural Network based efficient Software–Hardware Co-design Architecture for Sparsity Exploration

[1]S. John Pimo
Dept of CSE,
St. Xavier's Catholic College of Engineering,
Chunkankadai, Nagercoil, India
Mail id: johnpimo@sxcce.edu.in

[2]Priyadarsini.K
Dept of CSE
Vels Institute of Science Technology and Advanced Studies,
Chennai, Tamilnadu, India
Mail id: priyadarsini.se@velsuniv.ac.in

[3]Damaraju Sri Sai Satyanarayana
Dept of ECE,
Sreyas Institute of Engineering and Technology,
Hyderabad, India
Mail id: errorbots01@gmail.com

[4]Dr. K.G.S. Venkatesan
Dept. of CSE,
MEGHA Institute of Engg & Tech for Women,
Edulabad, Hyderabad, Telangana, India
Mail id: venkatesh.kgs@gmail.com

2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS) | 978-1-7281-9537-7/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICAIS50930.2021.9395805

***Abstract*— The presence of many zero values – is a pervasive property of modern deep neural networks, as it is inherently induced by state-of-the-art algorithmic optimizations. Recent efforts in hardware design for acceleration of neural networks have targeted the structure of computation of these workloads. However, when run on these value-agnostic accelerators, value sparsity is not exploited to provide performance or efficiency benefits, and instead results in wasted computation. This paper presents the architectural optimizations that efficiently leverage value sparsity in network weights in order to achieve significant performance benefits, with minimal hardware overhead. The culmination of this work is a hardware front-end (data fetching and staging unit) which, when paired with our novel, co-designed software scheduling algorithm, achieves more than a 2× speedup on average for the networks studied, with just an 8.2% overhead in compute area.**

Keywords— Software hardware codesign, Deep neural network, CNN, Sparsity.

I. INTRODUCTION

Deep learning is a machine learning (ML) technique that has enjoyed widespread attention from industry and academia in recent years. Deep neural network (DNN) models have emerged as powerful tools in a wide range of fields in which traditional algorithms have struggled to achieve satisfactory proficiency. Perhaps the most widely deployed type of DNN model is the convolutional neural network (CNN), which is dominant in computer vision tasks [1, 2, 3, 4], but has also seen success in fields as varied as speech recognition [5], reinforcement learning [6], and text translation [7].

From a hardware perspective, the deep neural network models that are used to implement deep learning represent a compelling workload for acceleration due to their widespread deployment in consumer and commercial settings, along with their unique computational structure and dataflow. The vast amount of computation required to run modern DNNs during inference (often on the order of tens of giga-operations (GOPs) [8]), along with their large memory footprint (commonly hundreds of MBs [8]) also make them a prime target for custom hardware. Indeed, many recent works have tackled the design and evaluation of hardware architectures for the acceleration of CNN inference processing. Some seminal works have investigated architectural techniques for efficiently exploiting the structure and forms of parallelism present in CNNs, with many highly influential application-specific integrated circuit (ASIC) architectures as well as field-programmable gate array (FPGA) implementations targeting CNNs, multi-layer perceptrons (MLPs), recurrent neural networks (RNNs), and other DNN types.

Alongside their basic computational structure, neural networks exhibit unique and interesting value properties – the unique distribution of values that appear during runtime for these workloads. Certain value properties can offer further opportunities for optimizing the hardware architectures designed to run these networks. Reminiscent of classical architectural approaches for exploiting workload characteristics such as cache hierarchies, which leverage the spatio-temporal locality of memory accesses, much investigation has been put into leveraging the implicit value properties of neural networks in hardware. This is an attractive prospect for several reasons, not least of which is that the massive computational complexity of neural network inference poses problems in terms of latency, energy, and power constraints, meaning techniques that can reduce this computational complexity are highly valuable. Additionally, the excessive memory footprint of modern CNNs is problematic for a multitude of reasons, including high memory transfer latency and energy, and large on-chip memory requirements. This makes model compression methods commonplace, many of which introduce even more opportunities for value-aware computation engines.

Authorized licensed use limited to: Cornell University Library. Downloaded on June 19,2021 at 01:02:23 UTC from IEEE Xplore. Restrictions apply.

II. BACKGROUND

A. Neural Networks

Machine learning (ML) is a field of artificial intelligence (AI) research that utilizes data-driven algorithms which iteratively improve their performance on some task, without explicit programming on how to complete that task. A popular machine learning technique that claims state-of-the-art results across a wide range of application areas is the neural network.

Neural networks are loosely inspired by the operation of biological neurons in the brain, in that the artificial neurons in a neural network produce a positive output only if the sum of its weighted inputs exceed some threshold, as modelled by the McCulloch-Pitts neuron in Figure 1 [19]. Neural networks are built up of layers of artificial neurons, with each layer containing many neurons. Neurons in adjacent layers are connected to one another by synaptic weights, with the output of one layer becoming the input of the next layer, and so on. In a fully connected (FC) layer, all output activations are connected by synapses to every neuron of the proceeding layer. Other forms of inter-layer connectivity are possible. By building up many layers, a deep neural network is created. Neural network computation has two phases: training and inference. During training, the synaptic weights are 'learned' by a training algorithm which attempts to maximize the accuracy of the network in completing some task.

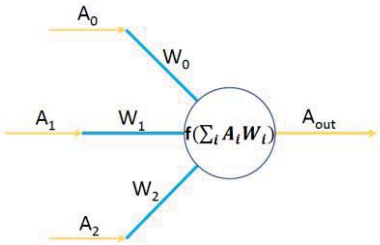

Figure 1: The McCulloch-Pitts model of a neuron

B. Value Sparsity in Neural Networks

Value sparsity is a specific value property in which a significant proportion of values in a given tensor are equal to zero. Pruning is a common optimization step during neural network training during which a significant fraction of the network weights are set to zero, as in Figure 2. This results in a sparse neural network, as the weight tensors are sparse objects containing many zero-valued elements. Weight pruning was originally introduced by LeCun et al. in their Optimal Brain Damage algorithm [14] as a way of reducing overfitting/increasing generalization, but it has seen increased favour in the ML community for its secondary benefit of acting as a network compression scheme. Pruning has been studied thoroughly due to its surprising efficacy, and numerous hypotheses have attempted to explain why pruning doesn't seem to affect model accuracy much until very high sparsity levels are reached [28, 29]. Other works have explored the trade-off between pruning, network size, and relative accuracy. Zhu & Gupta [10] argue that it is better in terms of memory footprint to train a large neural network and prune it, than it is to train a smaller neural network with a similar number of final non-zero parameters. The intuition behind the findings of Frankle & Carbin [28] is that there is exists a small 'subnetwork' within any large DNN that has been initialized in such a way that it is amenable to training to convergence successfully – called a 'winning ticket' subnetwork – and that is responsible for most of the accuracy of the network. Pruning reveals these subnetworks without affecting accuracy by removing redundant weights. Training a larger dense network increases the likelihood of there being a winning ticket subnetwork, thus it will always be easier to train a large network and prune away dense connections than it is to train a compact, dense network from scratch. This corroborates the findings of Zhu & Gupta [10], and suggests that weight sparsity is a value property that will likely continue to pervade neural networks in the future.

Modern pruning algorithms are implemented either as a post-processing step after the unpruned network has converged (with retraining to recover any lost accuracy), or as a part of the training process [10, 9]. A number of heuristics can be used to decide which weights to eliminate, the most common being the magnitude of the weight's value [10]. Others remove weights to which the output has the least sensitivity first [14, 15], however computing complex metrics like this is too costly for modern DNNs. In part because weights are randomly initialized at the start of training, the heuristics used in pruning result in sparsity that is relatively uniformly randomly distributed throughout the weight tensors [10]. This leads to irregularities in the network computation, which some works have tried to address by imposing constraints on how weights can be pruned, forcing them to be eliminated in groups [30, 31, 32]. These algorithms lead to structured sparsity, where either an entire contiguous group of weights (e.g., an entire filter channel) are zero, or none of them are. However, though designed to be more hardware-friendly, in practice these structured pruning techniques are rarely used as they make training a network to convergence without accuracy loss much more difficult. Mao et al. find that unstructured sparsity can reach much higher sparsity levels whilst maintaining model accuracy when compared to pruning at the granularity of filter-rows, filter-channels, or entire filters [33, 34].

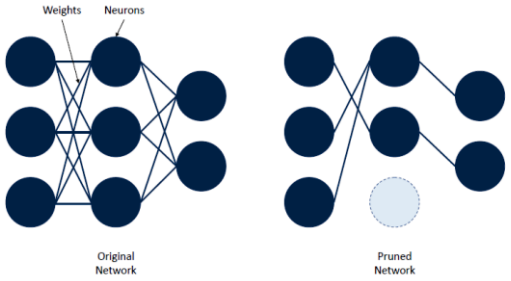

Figure 2: Example of weight pruning, which deletes weights using an heuristic algorithm

Algorithm1: Purne

1: Train and Mask ones W.shape, Prune (W, X, Y, LR, Epochs, S)
2: ← ( )
3: **for** $t = 0 : Epochs - 1$ **do**
W $W \odot Mask$

Authorized licensed use limited to: Cornell University Library. Downloaded on June 19,2021 at 01:02:23 UTC from IEEE Xplore. Restrictions apply.

4: ←
5: $Y' \leftarrow ForwardPass(W, X)$
6: $W_{Grad} \leftarrow BackwardPass(W, X, Y, Y')$
7: $W \leftarrow W - W_{Grad} \odot LR$
8: $Mask \leftarrow GenerateMask(W, S, t)$

Algorithm 1 gives a high-level, generic description of (one class of) pruning algorithms which prune iteratively during training. The procedure will operate on the network as defined by its weight tensors, W, the training data inputs X and labels Y , the learning rate LR, the number of training epochs, and the target sparsity, S. Before each forward pass, a binary mask is applied to the weights to zero-out pruned weights. The weights are updated using whatever learning algorithm is desired (vanilla gradient descent is shown in the algorithm). Finally, a new mask is generated (i.e., more weights are pruned) as a function of the current epoch and the final target sparsity, as the sparsity level is gradually increased during training. The GenerateMask() function is one of the key defining factors of a pruning algorithm, and its operation is what differentiates different pruning approaches. A simple function might simply sort all elements of W by magnitude, and prune the smallest) weights, meaning sparsity scales linearly as training progresses through epochs.

Another form of value sparsity present in modern neural networks is activation sparsity, where a significant fraction of activation values are equal to zero. Activation sparsity primarily occurs due to the rectified linear unit (ReLU) activation function, which is applied element-wise to the output of a convolutional or fully connected layer of a CNN, and clamps all negative values to zero, whilst letting positive values pass through unaffected, as shown in Figure 3. Activation sparsity is typically between 40% − 50% per-layer in a modern CNN [35].

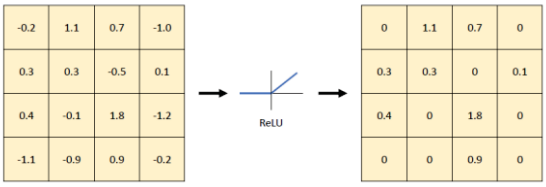

Figure 3: Activation sparsity of a 2D activation plane due to the ReLU element-wise non-linearity.

C. *Hardware Acceleration for Convolutional Neural Networks*

Convolutional neural networks have seen adoption and deployment across a range of industries and consumer applications. They are an attractive candidate workload for hardware acceleration as they are often employed in applications that where they have the following characteristics:

- Widespread deployment
- Computationally intensive
- Latency constrained
- Energy and/or power constrained

As such, a large amount of research and development activity has taken place in the CNN accelerator space in the last few years.

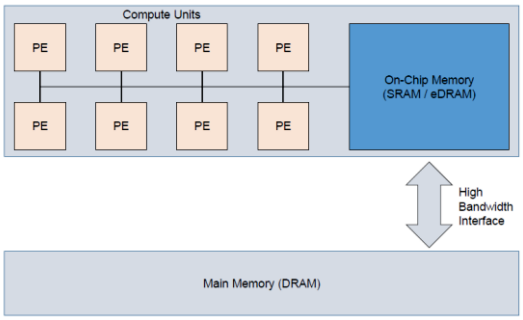

Figure 4: An abstract view of hardware architectures for CNN acceleration.

The majority of these designs are inference accelerators, though some more recent works also target the training phase of CNNs. Seminal works in this space explored how best to exploit the structure of CNN computation and leverage the large amount of parallelism in the workloads efficiently [42, 43, 44], however many more recent designs target the unique value properties of CNNs in order to increase the performance potential and efficiency of the hardware [17, 18, 35, 45, 46, 47, 48]. Thus, most hardware in the accelerator space can be classified as either value-aware or value-agnostic. Accelerators of both types share many basic traits. The primary operation in a CNN is the MAC, of which there are potentially billions per inference – see Table 1. All CNN accelerators will therefore support a large amount of parallel multiply and accumulate throughput in hardware. Figure 4 shows the basic structure of most modern CNN accelerators, the organization of which contains an array of processing elements to exploit the abundant data parallelism, on-chip buffers and scratchpads to exploit data reuse, and a high-bandwidth interface to main memory to load inputs and weights without causing a bottleneck.

Table 1: Image Classification CNNs

Network	Number of Layers		MAC Operations (mils.)	
	CONV	FC	CONV	FC
AlexNet [36]	5	3	665.8	58.6
GoogleNet [37]	57	1	1233.0	1.0
VGG-M [38]	5	3	1141.8	85.9
VGG-S [38]	5	3	1901.5	96.4
VGG-19 [38]	16	3	14999.8	123.6
MobileNet [13]	27	1	567.7	1.0
DenseNet-121 [39]	120	1	3062.0	1.0
DPNet-92 [40]	95	1	7384.5	2.7
ResNet-50 [41]	53	1	3855.9	2.0

Exact specification of the PEs and their specific architecture and organization can vary widely, and is at least partially a function of the desired dataflow – of which there are many possible. As in Figure 5, PEs may be simple MAC units, or

Authorized licensed use limited to: Cornell University Library. Downloaded on June 19,2021 at 01:02:23 UTC from IEEE Xplore. Restrictions apply.

even just multipliers as is the case in some systolic array-based designs [20], or may contain internal buffers or scratchpads with broadcast/multicast connectivity from memory [43], or multiple multipliers with accumulation capability, as in coarse-grained reconfigurable array (CGRA) style architectures [49]. PEs may also contain additional functional units to apply activation functions or pooling operations.

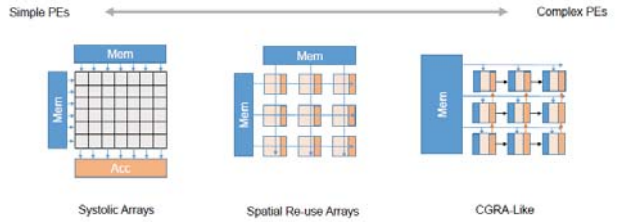

Figure 5: Hardware architectures for accelerating CNNs vary in their organization and in the complexity of their processing elements.

Dataflow and memory hierarchy are important aspects of the accelerator design, as together they define the memory access energy associated with inference, which can be a major contributor to energy efficiency. Table 2 shows the relative energy costs of arithmetic and memory access operations. Nearly all recent works in this space target 16-bit or 8-bit fixed point arithmetic as a baseline, due to the fact that modern neural networks suffer little-to-no accuracy loss at this data width compared to their native 32-bit floating point format [47], whilst integer multipliers and adders are many times more area efficient Table 2: Relative energy cost of various operations in 45nm 0.9V CMOS technology [50].

Table 2 shows the relative energy costs of arithmetic and memory access operations.

Bit width	Energy (pJ)	Relative Cost
32-bit int ADD	0.1	1
32-bit float ADD	0.9	9
32-bit Reg File access	1	10
32-bit int MULT	3.1	31
32-bit float MULT	3.7	37
32-bit 32KB SRAM access	5	50
32-bit DRAM access	640	6400

III. BASELINE ARCHITECTURE

Due to its popularity, scalability, and the efficiency of its adder-tree based design, a DaDianNao like design is used as the baseline architecture on top of which significant modifications are made to accommodate the weight sparsity.

DaDianNao (DaDN) is a wide vector-like machine, with separate, moderately sized on-chip buffers for weights and activations. DaDianNao is an extensible tile-based architecture, in which multiple tiles can be chained together to scale up the design. Figure 6 shows an overview of a DaDianNao tile. Each tile has a heavily banked weight memory (WM) which feeds k PEs – also called filter lanes, as each PE is assigned an entire filter at a time. Every tile also contains a slice of activation memory (AM), from which activations are broadcast to every PE within a tile. Each PE contains N multipliers, meaning it processes the inner product of N weights and N activations per cycle. The multipliers feed an N-input adder tree, which amortizes the cost of accumulation of an output activation.4.

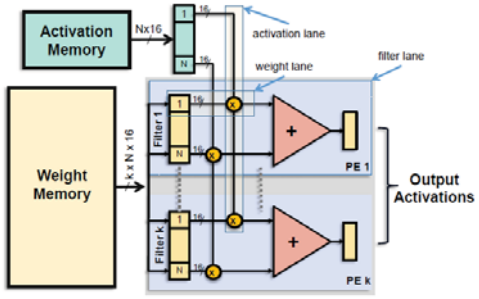

Figure 6: Baseline Architecture based on DaDianNao.

DaDianNao uses a channel-first dataflow, in which activations and weights are processed in contiguous chunks of N values from the same X–Y and R–S dimensions, respectively. That is, a slice of 1×1×N weights and activations are processed in each PE each cycle, with the N values being contiguous in the channel dimension, as shown in Figure 7. If the number of channels in a filter is not a multiple of N, zeros will be inserted in the weight tensor as 'padding' to ensure values are aligned correctly, as dictated by the dataflow.

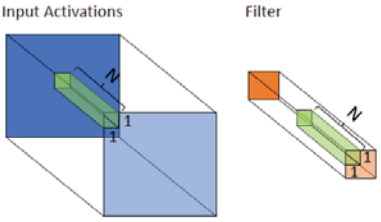

Figure 7: Data flow used by DaDianNao

Though DaDianNao was originally conceived as a multi-node accelerator with enough on-chip eDRAM to store all weights and activations per-layer on-chip during inference, this design choice is inefficient and over-provisioned. Instead, we size the activation memory to be large enough to keep input and output activations on-chip at all times using double buffering, but size our weight memory to store only one working set of filters at a time, and hide off-chip latency using double buffering, using our previously proposed heuristics [55]. We refer to this modified baseline design as DaDianNao++.

heuristic weight scheduling algorithm is described in Algorithm 2. The algorithm is shown for a single 'warp' of filters. A warp is defined as a set of K filters assigned to a K PEs simultaneously, before the next K filters are processed, i.e., PEs are synchronized on a warp boundary. Inputs to the algorithm include N, which is the number of multipliers per PE, the matrix of weights W, which is assumed to already be in

Authorized licensed use limited to: Cornell University Library. Downloaded on June 19,2021 at 01:02:23 UTC from IEEE Xplore. Restrictions apply.

in-memory layout and has dimensionality R × L, where R is the number of 'rows' of weights are to be processed, and is equivalent to the number of cycles the filter would take to process on DaDianNao++, and where L is the total number of multiplier lanes available in the accelerator, which for a single tile is equal to k × N. The interconnect is described by a list, I, of (lookahead, lookaside)

Algorithm 2 Scheduling Algorithm

Input
  $W \in ZR \times L$ weight matrix in memory layout
  I list of interconnect connection coordinates,
  N number of multiplier lanes per PE
Output
  $W \in ZR \times L$ modified weight matrix
  $W_S \in ZR \times L$ weight select signal matrix
procedure schedule (W, I, N)
  $W_S \leftarrow 0$
  for $r = 0$: $R - 1$ **do**
    if containsNonZero ($W[r,0: L - 1]$) **then**
      $numCandidates \leftarrow$ countCandidates(W, I, N, r)
      while containsNonZero($numCandidates$) **do**
      $W, W_S \leftarrow$ promote (W, W_S, I, N, r, $numCandidates$)
        $numCandidates \leftarrow$ countCandidates(W, I, N, r)
    else
      deleteRow(W,r)
      deleteRow(W_S,r)
  return W, W_S

procedure countCandidates(W, I, N, r)
  $numCandidates[0,...,L - 1] \leftarrow 0$
  for $l = 0 : L - 1$ **do**
    if W$[r,l] == 0$ **then**
      for $(ahead,aside) \in I$ **do**
        if W$[r + ahead,(l + aside)\%N)] != 0$ **then**
          $Candidates[l]++$
  return $numCandidates$

procedure promote (W, W_S, I, N, r, $numCandidates$)
  $Candidates_{min} \leftarrow min(nonZeros(numCandidates))$
  for $l = 0$: $L - 1$ **do**
    if W$[r,c] == 0$ && $Candidates[c] == Candidates_{min}$ **then**
      for $(ahead,aside) \in I$ **do**
        if W$[r + ahead, (l + aside)\%N] != 0$ **then**
          W$[r, l] \leftarrow$ W$[r + ahead,(l + aside)\%N]$
          W$[r + ahead,(l + aside)\%N] \leftarrow 0$
          $W_S[l] \leftarrow$ getIndexOf($(ahead, aside),I$)
          Break
  return W, W_S

IV. METHODOLOGY AND RESULTS

The performance of Bit-Tactical is evaluated using a custom cycle-accurate simulator which models the performance of the front-end and allows the exploration of various interconnect designs and scheduling algorithms. The simulator also provides detailed performance counters that allow analysis of the results.

Table 3: Baseline DaDianNao++ and TCT configurations.

DaDianNao++ or TCT			
Tiles	4	**Filters/Tile**	16
AS/Tile	$32KB \times 32$ Banks	**Weights/Filter (N)**	16
WS/Tile	$2KB \times 32$ Banks	**Precision**	16b
Act.Buffer/Tile	$1KB \times (h+1)$	**Frequency**	1GHz
Main Memory	8GB various tech nodes.	**Tech Node**	65nm
Lookahead	0-4	**Lookaside**	0-6
DaDianNao++			
Peak Compute BW	2 TOPS	**Area**	61.29 mm^2
Power	5.92 Watt		

The hardware configuration for the baseline design and TCT is outlined in Table 3. Area and energy measurements are performed post-layout using representative circuit activity. Layouts are generated for a TMSC 65nm technology using Cadence Innovus after synthesis using Synopsys Design Compiler. SRAMs are modeled via CACTI [58]. Off-chip memory energy consumption is modeled using Micron's DDR4 power calculator [59] along with access counts from the cycle-accurate simulations. All designs operate at 1GHz, with pipelining of the Datapath as needed to reach this target frequency. Both TCT and DaDianNao++ use k = 16 PEs per tile, with N = 16 multipliers per PE, all operating on 16-bit fixed point inputs. We initially show results assuming sufficient off-chip bandwidth so that no off-chip stalls occur, but later show the effect of various main memory technologies. We use run-length based zero compression as in [18] for weights, and fine-grain per group precision as in [60] for activations to reduce off-chip bandwidth for all layers.

A. Frontend Methodology

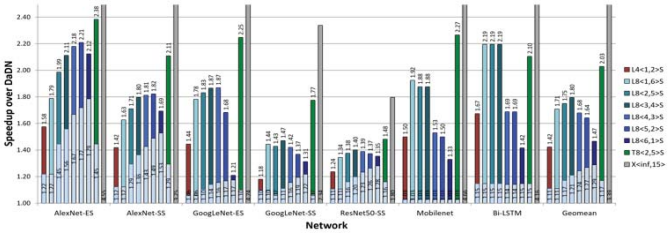

Figure 8: Speedup of bit Tactical Configurations over the baseline design.

We compare the performance of a variety of front-end designs against the baseline DaDianNao++ architecture across our benchmark suite in Figure 8. The lower portion of each stacked bar in the figure is the performance achieved by lookahead alone.

We also explore the effect of filter shuffling on front-end performance, and find that this optimization can result in up to a 18% performance increase (Bi-LSTM) on the studied networks, at no extra hardware cost. Figure 9 shows the performance results due to filter shuffling across networks.

Authorized licensed use limited to: Cornell University Library. Downloaded on June 19,2021 at 01:02:23 UTC from IEEE Xplore. Restrictions apply.

For the best performing network, AlexNet-ES, filter shuffling boosts performance by nearly 10%, increasing its speedup to 2.62×. This optimization always provides a performance increase, even if modest (2.2% for GoogleNet-ES). The speedups suggest that in most sparse network layers, sparsity is reasonably uniformly distributed, as there isn't a large disparity between the slowest and the fastest filters to process. Figure 10 explores the execution time breakdown for representative layers of each neural network, and for the total network. Multiplier cycles are normalized to dense execution time for each network and layer, and are categorized into four classes: ineffectual multiplier cycles due to processing ineffectual weights, effectual weights that were promoted using lookahead or lookaside, and effectual weights that remained unpromoted - i.e., weights that remain in their original position as in the dense schedule.

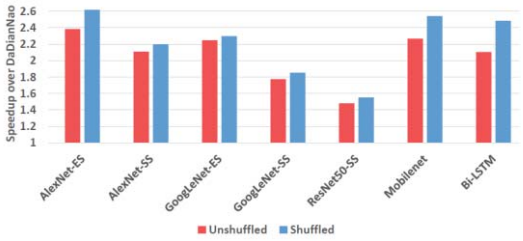

Figure 9: Speedup of the T8h2,5i configuration over the baseline, with and without filter shuffling.

Related to the execution time breakdown is the amount of sparsity that the TCT scheduler is able to effectively remove from execution. Though this information is derivable from Figure 10, it is explicitly listed per-network in Table 4. TCT manages to remove approximately of the ineffectual work due to zero weights and padding for all of our benchmarks, leaving at most 35.4% of ineffectual work on the table (GoogleNet-ES).

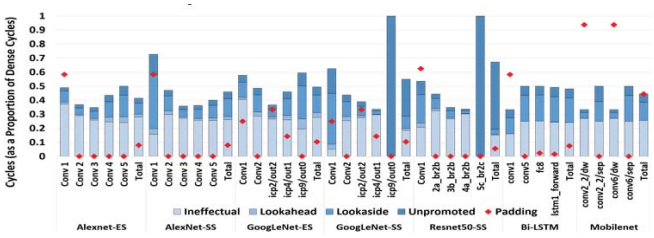

Figure 10: Breakdown of execution time normalized to dense time for representative layers of each network.

Network	Alex Net-ES	Alex Net-SS	Google Net-ES	Google Net-SS	Resne t-50-SS	Bi-LST M	Mobile Net
Proportion of Zeros Removed	67.5%	67.4%	64.6%	70.8%	68.3%	68.3%	68.5%

Figure 11 shows the relative performance using the heuristic scheduler outlined and a greedy algorithm. The heuristic scheduler never performs worse than the greedy scheduler, but on the hard-pruned networks (MobileNet and Bi-LSTM), the structure in the induced sparsity means both algorithms produce equivalent schedules most of the time. The low level of sparsity in ResNet-50-SS also offers little opportunity for the heuristic scheduler to excel, as the greedy algorithm already extracts nearly all of the potential performance from the network. On the sparser networks, however, the heuristic scheduler offers significant improvements, outperforming the greedy approach by up to 28% on GoogleNet-SS. On average, the heuristic algorithm achieves a modest 8% performance improvement on the networks studied. This increases to 14% if we consider only the networks for which the choice of scheduler has any impact on performance.

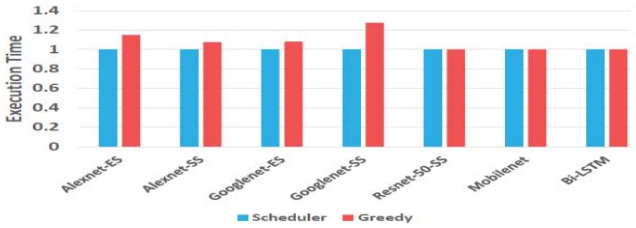

Figure 11: Effect of the scheduling algorithm on the networks studied.

Figure 12 shows the speedup of various TCT front-end designs, with varying lookahead distance and input multiplexer size, as sparsity is swept. All of the studied designs use a variant of the Trident connectivity shape. The T8h2,5i design is the best performing across all sparsity levels below 0.8, after which point the T8h3,4i design, with its increased lookahead distance, becomes dominant.

The additional lookahead does, however, make it a more expensive design. Given that all but one of the networks studied have a sparsity level less than 0.8, this extra hardware cost is not justified. On the other end of the design spectrum, the cheaper T8h1,6i design is substantially slower across most sparsity levels, with the T8h2,5i design being 1.45× faster at a sparsity level of 0.8.

Figure 13 illustrates the efficacy of the combination of the scheduling algorithm and the co-designed T8h2,5i interconnect. The greedy scheduler's performance will depend on the scan order, which in this experiment is set to target lookaside first. This explains why it performs slightly better at lower levels of sparsity than the heuristic scheduler, which is designed to make more globally-optimal scheduling decisions across the search window.

One key observation to make from these results is that the performance of the front-end is robust to changes in the sparsity distribution, with the range of speedups achieved by the T8h2,5i design never deviating from the average by more than 6%. Additionally, though initially an unremarkable observation, it is useful to note that Bit-Tactical never decreases performance below the baseline design – even at 0% sparsity, it achieves the same performance as DaDianNao++.

Authorized licensed use limited to: Cornell University Library. Downloaded on June 19,2021 at 01:02:23 UTC from IEEE Xplore. Restrictions apply.

This is not the case for SCNN, which suffers more than a 20% slowdown over their value-agnostic baseline design on dense networks, and only achieves speedups when both weight and activation sparsity surpass 15% each. This further validates the design principal of the Bit-Tactical front-end, which uses simple hardware and in doing so avoids potential performance overheads.

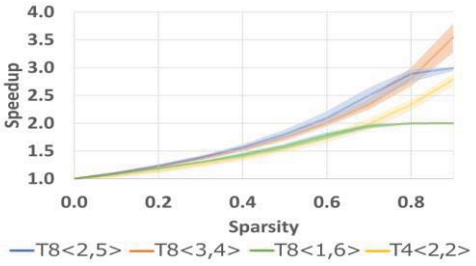

Figure 12: Performance of multiple interconnect patterns at varying sparsity levels.

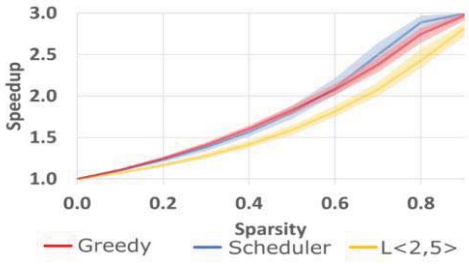

Figure 13: Performance variation of the Th2,5i design with different scheduling approaches as sparsity level changes, and compared against an L-shaped interconnect.

V. Conclusion

The large computational complexity and memory requirements of modern deep neural networks motivates the need for algorithmic techniques to reduce both of these metrics. One prevalent technique for doing so is weight pruning, which sets a large fraction of network weights to zero, with the goal of reducing memory footprint and traffic of DNN models, whilst giving a large amount of potential for speedup during inference. This work has motivated the need for more efficient approaches to leveraging weight sparsity in DNN inference. The Bit-Tactical front-end architecture is presented, and a thorough design space exploration and optimizations have been detailed. By designing and optimizing a lightweight front-end interconnect, we show how to judiciously leverage weight sparsity in hardware. Novel algorithmic optimizations which can improve the performance and reduce synchronization overheads are shown, including a scheduling algorithm which increases the performance of the front-end design by up to 28%. Additionally, we present further scheduling and hardware improvements which increase performance and decrease memory overheads by up to an additional 18%, and by up to 82%, respectively. Combining both the front-end hardware with the scheduling algorithm and optimizations results in a front-end accelerator design which can achieve up to a 2.62 times speedup over a similarly provisioned value-agnostic baseline design, with just an 8.2% logic area overhead. In addition, despite targeting sparse neural networks, the design presented suffers no performance degradation for dense networks (unlike other sparse accelerators, which suffer from decreased performance on dense networks), and may even offer slight performance improvements due to zero-values from weight padding. Equivalently, Bit-Tactical's performance is robust across all sparsity levels, and so encourages weight pruning wherever possible, even if very high sparsity levels are not attainable. In summary, Bit-Tactical's novel, pragmatic approach to exploiting weight sparsity offers a compelling trade-off between hardware complexity and attainable performance that we hope motivates similar future efforts in value-aware acceleration for ML and other domains.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep cnns. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, NIPS 25, pages 1097–1105. Curran Associates, Inc., 2012.

[2] V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 39(12):2481–2495, Dec 2017.

[3] K. Zhang, W. Zuo, S. Gu, and L. Zhang. Learning deep cnn denoiser prior for image restoration. In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 2808–2817, July 2017.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. Commun. ACM, 60(6):84–90, May 2017.

[5] O. Abdel-Hamid, A. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu. Convolutional neural networks for speech recognition. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 22(10):1533–1545, Oct 2014.

[6] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, Proceedings of The 33rd International Conference on Machine Learning, volume 48 of Proceedings of Machine Learning Research, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[7] Jonas Gehring, Michael Auli, David Grangier, and Yann N. Dauphin. A convolutional encoder model for neural machine translation. CoRR, abs/1611.02344, 2016.

[8] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. CoRR, abs/1605.07678, 2017.

[9] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016.

[10] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv e-prints, page arXiv:1710.01878, Oct 2017.

[11] Tulasi Radhika Patnala, Jayanthi D, Shylu D.S, Kavitha K, Prathyusha Chowdary, "Maximal length test pattern generation for the cryptography

Authorized licensed use limited to: Cornell University Library. Downloaded on June 19,2021 at 01:02:23 UTC from IEEE Xplore. Restrictions apply.

applications" https://www.sciencedirect.com/science/article/pii/S2214785320305368, meterialstoday proceedings, In press, available online from 20.02.2020

[12] Tulasi Radhika Patnala, Jayanthi D, Sankararao Majji, Manohar Valleti, Srilekha Kothapalli, Santhosh Chandra Rao Karanam, "Modernistic way for KEY Generation for Highly Secure Data Transfer in ASIC Design Flow" https://ieeexplore.ieee.org/document/9074200, Published in IEEE digital Xplore, Electronic ISSN: 2575-7288, available from 23.04.2020

[13] Tulasi Radhika Patnala, Sankarrao Majji, Gopala Krishna Pasumarthi, "Optimization of CSA for Low Power and High Speed using MTCMOS and GDI Techniques", https://www.ijeat.org/wp-content/uploads/papers/v8i5S3/E10620785S319.pdf , International Journal of Engineering and Advanced Technology (IJEAT) ISSN: 2249 – 8958, Volume-8 Issue-5S3, July, 2019

[14] Sankararao Majji, Tulasi Radhika Patnala, Manohar Valleti, Srilekha Kothapalli, Santhosh Chandra Rao Karanam, "A Study on the Comprehensive Analysis of Electro Migration for the Nano technology trends", Published in IEEE digital Xplore, Electronic ISSN: 2575-7288, available from 23.04.2020

[15] Sajja Krishna Kishore;Tulasi Radhika Patnala;Arun S Tigadi;Aatif Jamshed "An On-chip Analysis of the VLSI designs under Process Variations" published in IEEE digital Xplore, 2020 International Conference on Smart Electronics and Communication (ICOSEC).

Authorized licensed use limited to: Cornell University Library. Downloaded on June 19,2021 at 01:02:23 UTC from IEEE Xplore. Restrictions apply.