Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks

Alberto Delmas Lascorz
delmasl1@ece.utoronto.ca

Patrick Judd*
pjudd@nvidia.com

Dylan Malone Stuart
malones2@ece.utoronto.ca

Zissis Poulos
zpoulos@ece.utoronto.ca

Mostafa Mahmoud
mostafa.mahmoud@mail.utoronto.ca

Sayeh Sharify
sayeh@ece.utoronto.ca

Milos Nikolic
milos.nikolic@mail.utoronto.ca

Kevin Siu
kcm.siu@mail.utoronto.ca

Andreas Moshovos
moshovos@ece.utoronto.ca

University of Toronto

Abstract

Weight and activation sparsity can be leveraged in hardware to boost the performance and energy efficiency of Deep Neural Networks during inference. Fully capitalizing on sparsity requires re-scheduling and mapping the execution stream to deliver non-zero weight/activation pairs to multiplier units for maximal utilization and reuse. However, permitting arbitrary value re-scheduling in memory space and in time places a considerable burden on hardware to perform dynamic at-runtime routing and matching of values, and incurs significant energy inefficiencies. Bit-Tactical (*TCL*) is a neural network accelerator where the responsibility for exploiting weight sparsity is shared between a novel static scheduling middleware, and a co-designed hardware front-end with a lightweight sparse shuffling network comprising two (2- to 8-input) multiplexers per activation input. We empirically motivate two back-end designs chosen to target bit-sparsity in activations, rather than value-sparsity, with two benefits: a) we avoid handling the dynamically sparse whole-value activation stream, and b) we uncover more ineffectual work. *TCL* outperforms other state-of-the-art accelerators that target sparsity for weights and activations, the dynamic precision requirements of activations, or their bit-level sparsity for a variety of neural networks.

CCS Concepts • **Computer systems organization → Neural networks**; **Special purpose systems**.

*Also with NVIDIA. Work completed while at the University of Toronto.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304041

Keywords Sparsity, Deep Learning Acceleration

ACM Reference Format:
Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. 2019. Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304041

1 Introduction

Deep Neural Networks (DNNs) have become a prevalent tool in a wide range of applications, from image processing, where Convolutional Neural Networks (CNNs) offer state-of-the-art performance [5], to neural machine translation (*e.g.,* text translation, speech recognition/synthesis) [8, 16, 25], where both CNNs and Recurrent Neural Networks (RNNs) exhibit best-to-date results. Despite their efficacy in these domains, the computation- and memory-intensive nature of such models limits their energy efficiency and performance on commodity hardware. As a result, hardware accelerators for DNNs have enjoyed viral attention [2, 4–7, 14, 31, 34].

Meanwhile, it is well established that these networks are often heavily over-parameterized [15]. This observation can be leveraged to reduce the footprint of DNNs by *pruning* (setting to zero) nonessential network weights, without significantly degrading accuracy. Pruning results in a *sparse* network, in which a large proportion (more than 90% [13, 15] on earlier models, and closer to 50% for more recent models [32]) of the weights are zero. Sparsity presents a great opportunity for inference acceleration.

However, when processed on value-agnostic hardware, a sparse network will not see any performance or energy benefits compared to its dense (unpruned) counterpart. Instead, the network sparsity will result in a large amount of *ineffectual* computations, which occur any time a pruned weight is multiplied by an activation, producing a value of zero, thus having no effect on the output neuron. Analogously,

ineffectual work also results when a zero-valued activation is multiplied by a non-pruned weight.

Fully benefiting from sparsity requires constructing a compute *schedule* containing only the non-zero weights and activations and then efficiently *mapping* that schedule onto the available execution resources, in order to maximize utilization and minimize memory accesses. *Constructing* such a schedule is difficult for two reasons. First, which activations will be non-zero is input-dependent and thus known with certainty only at runtime. Second, while non-zero weights are known statically, at runtime in convolutional layers each weight is used multiple times, once per window, each time pairing with a different activation. Similarly, in fully-connected layers, each activation is multiplied with different weights. *Mapping* the schedule onto hardware is equally challenging since it requires that: 1) each non-zero weight (activation) to find its corresponding activation (weight), 2) as long as both values in the pair are non-zero, having them meet at a multiplier, and 3) having their product appear at the accumulator assigned to the corresponding output neuron.

In hardware this proved expensive as it required dynamic routing and pairing of weights and activations [39], or partial sums [31], ideally without constraint. Indicatively, SCNN, the state-of-the-art sparse CNN accelerator, eliminates all ineffectual products [31] by taking advantage of weight and activation reuse in convolutional layers. Eliminating *all* ineffectual products, however, introduces inefficiencies eroding overall benefits. Specifically, rearranging the input values in memory renders it impossible to statically route every product to a known accumulator. Instead SCNN dynamically routes products through an over-provisioned crossbar to a set of relatively large accumulator banks. This type of routing accounts for over 21% of each processing element's area and, as a result of work imbalance due to activation tiling, suffers from a 2.05× performance loss compared to upper-bound opportunity on performance [31]. Similarly, in Cambricon-X, which exclusively targets pruned weights, the Indexing Module accounts for over 31% of total die area and for over 34% of on-chip power [39].

In summary, fully exploiting activation and weight sparsity translates into motion of values in the *memory space*, in *time* (execution schedule), and *spatially* over the available multipliers and accumulators. Existing sparse network accelerators allow nearly unrestricted motion of values in memory space and in time, placing the burden on hardware to take advantage of sparsity at runtime. There is untapped potential to better balance the responsibility of exploiting sparsity between hardware and software. We demonstrate that by introducing a programmer transparent pre-processing software layer, and by co-designing it with the underlying hardware, it is possible to not only exploit sparsity but to also substantially reduce hardware overhead and inefficiencies.

We propose Bit-Tactical (*TCL*), which targets sparse DNNs, but which does not sacrifice energy efficiency and performance for dense DNNs. *TCL* software and hardware are co-designed from the ground up so that all weight movements in time and space are orchestrated statically via a novel scheduling middleware. *TCL* supports a limited, yet structured set of weight movements within the schedule. This enables *TCL*'s hardware to use a lightweight *sparse* interconnect and to perform all corresponding activation movements at runtime while incurring a very low hardware overhead.

To fully exploit this limited hardware functionality we design a novel software scheduling algorithm that approximates an optimal weight schedule within the constraints imposed by this restricted hardware connectivity. The scheduler performs a window-based search to identify spatio-temporal weight movements that, during runtime, will effectively keep multiplier units as busy as possible. Weights are only permitted to (a) temporally advance by a limited number of time steps, and (b) spatially reposition to occupy a predetermined set of multipliers. As a result, *TCL* achieves significant zero-weight skipping, while maintaining a lean hardware implementation. The design uses direct 1-to-1 scratchpad-to-multiplier connections, and a *sparse* interconnect with two multiplexers (one for movement in time, one for movement in space) to route activations to the appropriate multiplier. We show that 2- to 8- input multiplexers are sufficient.

TCL processes *all* activations, avoiding the challenges of eliminating zero-valued activations at runtime. Despite this, *TCL* manages to remove more ineffectual work than accelerators that target zero activations and weights. Specifically, two versions of the design, *TCLp* and *TCLe*, target the ineffectual *bit* content of activations, which exceed 90% and 75% respectively for recent 16b and 8b fixed-point CNN models [1], a phenomenon that persists in a variety of other models [29]. *TCLp* is a variant that employs a bit-serial synchronized back-end, which exploits the variable, fine-grain *dynamic* precision requirements of activations [11]. It entirely skips the zero-bit content at both the prefix and suffix of the activation value, ideally reducing execution time proportionally to the precision needs vs. an accelerator that uses fixed precisions even if these precisions were adjusted per layer. *TCLe* is a more aggressive back-end which processes *only* the effectual activation terms (non-zero powers of two of the Booth-encoded values) serially [1].

This work departs from the dynamic routing paradigms present in state-of-the-art accelerators [31, 39], and instead trades hardware overhead for software complexity in order to achieve the twin goals of a competitive speedup and high area- and energy-efficiency.

Our key contributions are:

1. We motivate a novel approach to accelerate sparse neural networks: Rather than skipping zero weights and activations we propose to skip zero weights and the ineffectual

bits of the activations (bit-level sparsity). This approach 1) eliminates the need to handle the dynamically sparse activation stream, and 2) exposes more ineffectual work.
2. We present a novel class of front-end hardware designs that allow zero weight skipping where all weight promotions are performed by a software scheduler and all corresponding activation promotions are pre-planned by the scheduler and performed at runtime via a lightweight, sparse interconnect.
3. We formulate a novel weight scheduling algorithm co-designed with the aforementioned hardware connectivity that extracts most of the performance possible from a sparse neural network given the hardware constraints.
4. By separating the front-end weight scheduling from the back-end multiply-accumulate (MAC) operations, we explore the energy/area/performance trade-off for two designs (*TCLe* and *TCLp*) that offer different performance, area, and energy trade-offs. We show that the benefits of the front- and back-end are nearly multiplicative.

We highlight the following findings:

- The most aggressive design studied, *TCLe*, provides an average speedup over a dense baseline accelerator and SCNN of 11.3× and 5.14× respectively, whilst being 2.13× and 2.23× more energy efficient. *TCLe* also outperforms both Pragmatic [1] and Dynamic Stripes [11, 17].
- *TCLe* improves performance by 7.2× even with 8b range-oblivious quantization.
- Against the same comparison points, a more conservative and lower cost design, *TCLp*, provides speedups of 6.7× and 3.05× respectively, while incurring just a 10% area overhead and being slighty more energy efficienct.
- *TCLe* and *TCLp* achieve an average effective 17.6 and 10.4 TOPS at 1.04 and 1.11 TOPS/W, respectively estimated post layout with a 65nm technology node.
- Targeting whole value sparsity for weights and bit-level sparsity for activations is in principle compatible with SCNN's approach. However, for the relatively low resolution inputs of IMAGENET [12, 35], doing so exacerbates load imbalance. Considering higher resolution inputs is left for future work.

2 Revisiting Sparsity

Hardware accelerators for neural networks are typically structured in a way that guarantees, for a given dataflow, that values can be statically arranged in memory such that weight and activation pairs meet at a desired multiplier (space) and cycle (time). When leveraging weight and activation sparsity in order to achieve a speedup, this is no longer the case — dynamic datapath routing must allow movement and filtering of operands or products in order to allow non-zero input values to be processed ahead of their dense schedule.

One approach to exploiting sparsity would be to match and route weights or activations from anywhere within their respective scratchpad memories to appear simultaneously at the desired multiplier. This requires an interconnect *before* the multipliers that allows arbitrary motion of weights and/or activations along with some logic for dynamically identifying and matching those pairs where both operands are non-zero. The Cambricon-X accelerator implements a restricted form of this approach as it eliminates only zero weights [39]. We corroborate the findings of Parashar *et al.* that not eliminating zero activations too leaves a lot of potential untapped [31].

Alternatively, we can take advantage of the observation that in convolutional layers with a unit stride, the product of any weight and any activation (ignoring the few values near the edges) within the same channel contributes to some output activation. This is the approach followed by SCNN, which allows arbitrary motion of weight-activation *products* using a crossbar *after* the multipliers to route them to the accumulator assigned to their corresponding output activation [31]. In either approach, the dynamic routing of values to or from the compute units using a crossbar or equivalent dense interconnect is an energy-intensive option.

As our first contribution we show that, combined with targeting zero weights, more ineffectual work can be found when targeting zero activation *bits* instead of zero activations. This is of particular value for sparse model acceleration, as it eliminates the need to handle the dynamically sparse whole-value activation stream. This will enable *TCL* to push most of the functionality to software, and to utilize only a sparse interconnect for the activations.

Table 1 shows the fraction of total work that can ideally be removed from a set of sparse networks (16b fixed-point, see Section 6.5 for 8b fixed-point models), expressed as a relative

Policy	Representative Accelerators
A	Cnvlutin [9], ZeNa [24], EIE [14]
W	Cambricon-X [39]
W+A	SCNN [14]
Ap	Dynamic Stripes [9, 23]
Ae	Pragmatic [1]
W+Ap	*Bit-Tactical* (this work, *TCLp*)
W+Ae	*Bit-Tactical* (this work, *TCLe*)

Model	A	W	W+A	Ap	Ae	W+Ap	W+Ae
Alexnet-SS [32]	1.6x	6.7x	11.6x	3.6x	8.0x	26.2x	58.4x
Alexnet-ES [38]	1.5x	4.3x	5.1x	3.2x	7.1x	10.6x	23.7x
Googlenet-SS [32]	1.8x	4.4x	7.8x	3.9x	8.8x	16.6x	37.2x
Googlenet-ES [38]	1.9x	2.5x	4.6x	3.4x	8.2x	8.5x	20.7x
Resnet50-SS [32]	2.5x	1.7x	3.9x	9.0x	17.9x	14.0x	28.0x
Mobilenet [41]	1.8x	2.2x	3.3x	2.7x	6.5x	5.0x	12.0x
Bi-LSTM [28]	1.6x	3.7x	5.6x	2.5x	6.1x	9.2x	22.5x
Geomean	**1.8x**	**3.3x**	**5.5x**	**3.7x**	**8.4x**	**11.4x**	**26.0x**

Table 1. Performance improvement potential.

speedup potential over performing all computations, for various sources of ineffectual computations: **A:** zero activations, **W:** zero weights, **Ap:** the dynamic precision requirements of activations, **Ae:** the bit-level sparsity of activations, and various combinations of the above.

Whereas skipping activations or weights alone (A and W) could improve performance on average by 1.8× or 3.3× respectively, skipping both (W+A) boosts the potential to 5.5×. Compared to W+A, targeting W+Ap or W+Ae increases the average potential benefits by more than 2.1× and 4.7×, respectively, to 11.4× and 26.0×. The potential is higher for the more recent models, GoogleNet and Resnet50, where even Ae exhibits higher potential than W+A.

3 A Zero Weight Skipping Front-End

We split the functionality of *Bit-Tactical* into a zero weight skipping and activation pairing *front-end*, and an activation zero-bit skipping *back-end*. This section discusses the front-end, the most innovative part of *TCL*, where the functionality needed is judiciously shared between software and hardware.

We observe that to remove the vast majority of zero weights, it is not necessary to allow unrestricted weight movement. Indeed, there is a spectrum of interconnect densities that allow *restricted* movement of values, trading off performance potential for energy and area savings. In *TCL*, weight movement is performed by a software scheduler that arranges the weights in the memory space, according to the desired dataflow. At runtime, a lightweight interconnect implements the corresponding movement of activations so that each weight/activation pair appears at a known multiplier.

In order to explore this design space, we define two basic *weight movement* primitives: *lookahead* and *lookaside*. Lookahead *promotes* weights ahead in *time*, so that they are processed sooner than what their original dense schedule dictates. With lookahead, a weight appears at the same multiplier (lane) as it would in the dense schedule, however it does so earlier in time. Lookaside reduces serialization of multiple non-zero weights over the same multiplier by also allowing movement in *space*; a weight that cannot be promoted in time because its dense-schedule multiplier is currently occupied is allowed to move to another multiplier which is still assigned to the same output activation. By reducing workload imbalance across multiplier lanes, lookaside results in a more compact schedule.

Weight Lookahead: Figure 1 shows an example of lookahead 1 (denoted $h = 1$) for a sparse filter. The dense schedule is shown for reference, and parts (a) through (c) illustrate how lookahead reduces execution time to 3 cycles. We use a basic computing element (CE) containing several multipliers (four in the figure) feeding an adder tree. This CE is more energy efficient for inner-products compared to single multiply-accumulate units as it amortizes the cost of reading and writing the partial sum over multiple products. This CE can be used as the building block for many accelerator organizations, a topic Section 5.3 discusses further. Regardless, lookahead and lookaside are not specific to this CE.

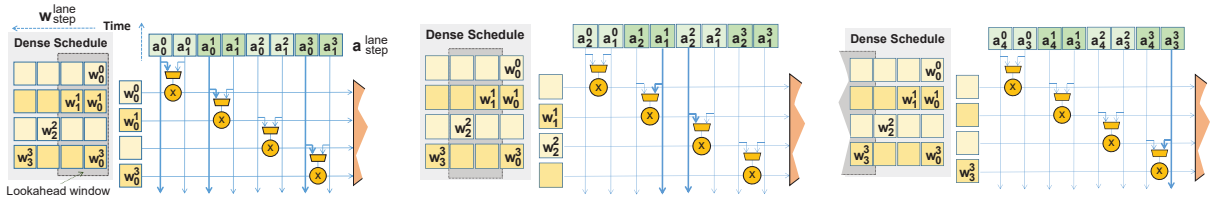
Conceptually, lookahead amounts to establishing a sliding window of $h + 1$ within which weights can be promoted over earlier ineffectual weights that would have appeared in the same lane. *TCL* must pair each weight with the corresponding activation at runtime. To achieve this pairing, *TCL* requires that all activations for the full lookahead window be available. If $h = 1$, for each weight lane there are now 2 activation lanes corresponding to time steps t and $t + 1$. *TCL* selects the appropriate activation via a per weight lane *2-to-1* multiplexer. The control signal for the multiplexer is determined statically when the schedule is created, and is stored along with the weight. In general, for a lookahead h, *TCL* maintains a pool of $h+1$ activations per weight lane and an *(h+1)-to-1* multiplexer to select the appropriate activation. In practice, we show that a lookahead of 1 or 2 is sufficient. Lookahead should be increased with care as it determines the size of the search window of activations per weight, and the number of activation lanes physically present.

Weight Lookaside: With lookahead, the weight lane with the most effectual weights is the bottleneck, leading to imbalance. Lookaside introduces further scheduling flexibility wherein a lane can "steal" work from another lane contributing to the same output activation. For our CE, this amounts to moving a weight to a different multiplier within the CE. Figure 2 shows that with lookaside of 1 (denoted $d = 1$), *TCL* processes our example using the minimum possible 2 cycles.

Lookaside requires no additional activation lanes. It only requires an activation multiplexer with more inputs, and thus is less costly than lookahead. In general, our font-end needs the equivalent of an *(h+d+1)-to-1* multiplexer per multiplier (4- to 8- input prove sufficient) for lookaside h and lookahead d (denoted <h,d>). As Section 5.1 explains, the data input connections for these multiplexers are statically determined and regular.

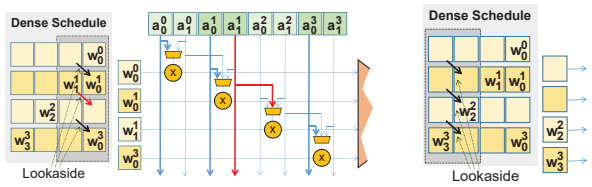
3.1 Hardware Connectivity and Software Implications

Using combinations of the lookahead and lookaside weight movement primitives, we can implement arbitrary intra-filter interconnects, up to and including a crossbar. We are interested, however, in much less costly, judiciously designed interconnect patterns. The simplest of these interconnect patterns that we study is a contiguous pattern consisting of a lookahead of h and a lookaside of d, resulting in an 'L'-shaped search window, as shown in Figure 3a. It is preferable to design non-contiguous, that is *sparse*, connectivity patterns such as the trident-like T<2,5> pattern of Figure 3b. These benefit from a reduction in overlapping connections between neighboring lanes, which empirically results in more promotion opportunities and reduced contention between nearby weights.



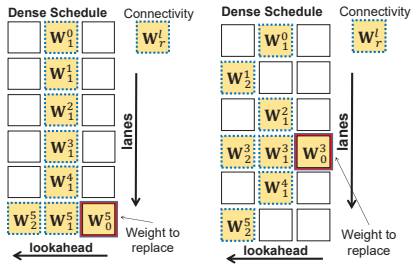
(a) Cycle 0 **(b)** Cycle 1 **(c)** Cycle 2

Figure 1. *TCL* Accelerator with Lookahead of 1 processes the sparse NN of part (a) in 3 cycles. **(a) Cycle 0**: lookahead fails to utilize weight lane 2 since weight w_2^2 is at lookahead distance 2. **(b) Cycle 1**: lookahead promotes w_2^2 to replace w_1^2. However, w_3^3 is out of reach as lane 1 is now processing w_1^1 limiting lookahead to weights that appear up to step 2 in the dense schedule. As there are no weights left to process in step 2, the lookahead window now progresses two steps. **(c) Cycle 2:** w_3^3 is processed.



(a) Cycle 0 **(b)** Cycle 1

Figure 2. *TCL* with Lookahead of 1 and Lookaside of 1 processes the sparse NN of Fig. 2a in 2 cycles, the minimum possible. **(a) Cycle 0:** lane 2 "steals" w_1^1 from lane 1 and avoids staying idle while also allowing the lookahead window to progress by two steps. **(b) Cycle 1:** through lookahead, lane 3 can process w_3^3 at the same time as lane 2 is processing w_2^2.



(a) L<2,5> **(b)** T<2,5>

Figure 3. Two potential interconnect patterns: (a) a contiguous pattern , and (b) a sparse interconnect in a trident shape. Both only require an 8 input mux at the activation input of each multiplier.

As soon as connectivity is reduced below fully-associative (arbitrary weight movement), it may not be possible to remove *all* zero weights from the schedule. Any reduction in connectivity requires decisions to be made, in any given cycle, as to *which* weights should be promoted, and to *where*. Therefore, our reduced-connectivity interconnect removes hardware cost/complexity by shifting this responsibility to a *software scheduler* that determines, statically, which weight

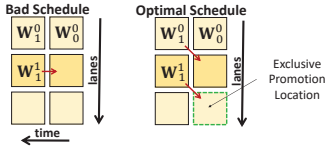


Figure 4. A toy example with 3 multiplier lanes and 3 weights, assuming a lookahead of one and a lookaside of one. An optimal schedule can process this filter in a single cycle, whereas a suboptimal schedule takes two.

movements result in the best overall speedup. Fortunately, our software scheduler achieves nearly optimal performance.

4 The Software Scheduler

Determining a compact schedule that maximizes weight-skipping and maintains good workload balance across lanes is a complex task. This optimization relates to the problem of *Minimum Makespan Scheduling* [3], variants of which are generally addressed by employing a class of greedy algorithms [33]. Imposing the schedule movement constraints implied by the pre-specified sparse interconnect adds an extra layer of complexity, since promotion decisions can be interdependent: each effectual weight may have many possible ineffectual weights it could replace in the schedule, and vice versa. Further, weight promotions at cycle t may have second order effects on promotion opportunities at cycle $t+1$, and so on.

To handle these inter-dependencies, we developed the heuristic greedy Algorithm 1, which iteratively performs promotions to *exclusive* ineffectual positions first; that is, ineffectual positions for which there is only a single promotion candidate weight. By doing so, we reduce the amount of potential promotions that are blocked due to other nearby sub-optimal promotions. Figure 4 shows how sub-optimal promotions can cause reduced performance.

For brevity, we only show the scheduling procedure for a single filter F and the dense schedule time T of a single window in F. The skipping connectivity induces a function $S(u, v)$, which returns a set of $(time, lane)$ positions from which a weight can be promoted to replace an ineffectual

weight w_{uv} appearing at time u and lane v in the dense schedule. In lines 6-12, for each ineffectual weight in time t, we maintain and update a count (*Candidates*) of all effectual weights in the lookahead window that can be promoted to replace it (*i.e., Candidates*$[l]$ maintains this tally for a weight in time t and lane l). Then, in lines 13-24, we identify the ineffectual weights with the smallest such count (*i.e.,* least flexibility for replacement) and replace these with higher priority. In the common case, the smallest count, denoted $Overlap_{min}$, will equal 1, and so line 18 will only perform promotions to *exclusive* positions, where only a single promotion is possible.

Algorithm 1 Scheduling Algorithm

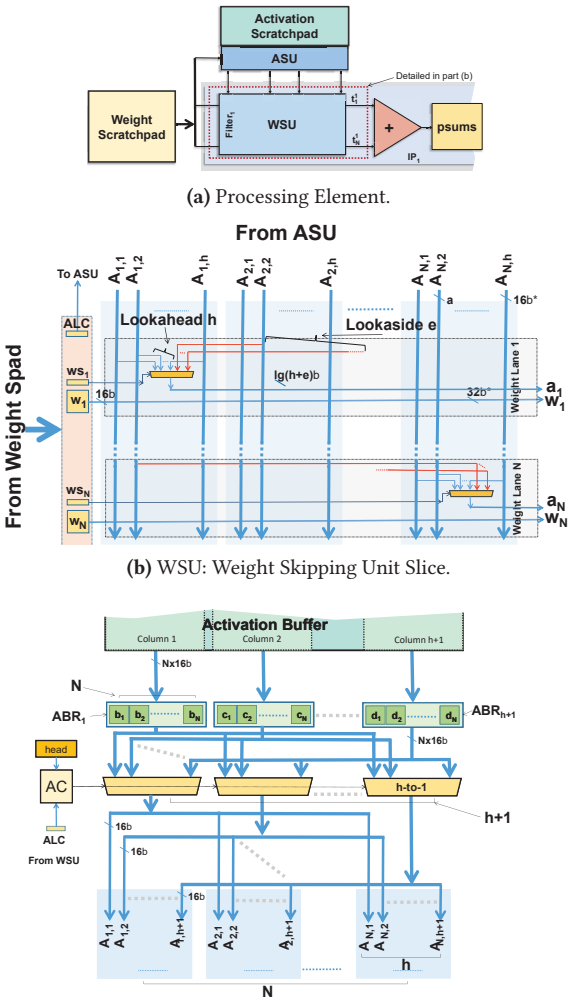
```
1:  procedure SCHEDULE(F, T)
2:    Promotions ← ∅
3:    for t = 0 : T − 1 do
4:      Candidates[0, ..., L − 1] ← ∞
5:      while max(Candidates) > 0 do
6:        Candidates[0, ..., L − 1] ← 0
7:        S ← ⋃_{u=t, 0≤v≤L−1} S(u, v)
8:        Effectuals ← {(i, j) : (i, j) ∈ S, w_ij ≠ 0}
9:        for (i, j) ∈ Effectuals do
10:         Ineffectuals ← {(i′, j′) : (i, j) ∈ S(i′, j′), i′ = t, w_i′j′ = 0}
11:         for (i′, j′) ∈ Ineffectuals do
12:           Candidates[j′] + +
13:       Overlap_min ← min{Candidates[j′] : (i′, j′) ∈ Ineffectuals}
14:       for (i, j) ∈ Effectuals do
15:         Ineffectuals ← {(i′, j′) : (i, j) ∈ S(i′, j′), i′ = t, w_i′j′ = 0}
16:         for (i′, j′) ∈ Ineffectuals do
17:           if Candidates[j′] == Overlap_min then
18:             Promotions ← Promotions ∪ {((i, j), (i′, j′))}
19:             Ineffectuals ← Ineffectuals − {(i′, j′)}
20:             Effectuals ← Effectuals − {(i, j)}
21:             Candidates[j′] ← 0
22:             for (i′, k) ∈ Ineffectuals do
23:               Candidates[k] − −
24:             Overlap_min ← min{Candidates[k] : (i′, k) ∈ Ineffectuals}
25:             Break
26:   return Promotions
```

5 TCL Architecture

5.1 Weight-Skipping Front-End

Here we describe just the ineffectual weight skipping *front-end* of *TCL*. Section 5.2 completes the design with the back-end. For clarity our discussion assumes 16b fixed-point activation and weights. The designs can be straightforwardly adapted for other data widths. Recall that all weight motion is preplanned by the software scheduler and implemented by storing weights in the appropriate order in memory. The *TCL* front-end implements the corresponding activation motions. For clarity we describe the implementation for a specific front-end configuration where: a) Lookaside can promote by only one step in time from any of the following d neighboring lanes, that is w^{lane}_{step} can "steal" any $w^{(lane+d) \bmod (N-1)}_{step-1}$. b) Activations and weights use 16b and thus there are 16 input/communication wires per activation and weight (the designs of Section 5.2 use either 1 or 4 wires per activation, a much reduced cost).

Figure 5a shows a *TCL* processing element that processes $N(= 16)$ products in parallel. Each cycle, a weight scratchpad



(a) Processing Element.

(b) WSU: Weight Skipping Unit Slice.

(c) ASU: Activation Selection Unit.

Figure 5. *TCL* front-end Architecture. Section 5.2 reduces the number of wires needed per activation to 1 or 4.

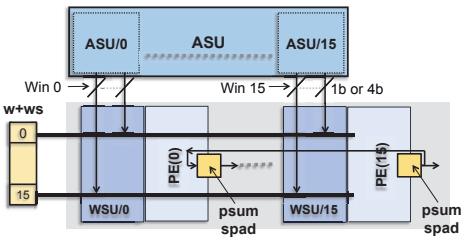


Figure 6. Adding the capability to skip zero activation bits.

(WS) reads out, via a single port, a column of N (w_i, ws_i) or *(weight, mux control signal)* pairs, plus an activation lane control (ALC) field (see below). The *Activation Select Unit* (ASU) buffers activations as they are provided by the activation

scratchpad (AS) and "rearranges" them into the lookahead window that the *Weight Skipping Unit* (WSU) needs.

Weight Skipping Unit: Figure 5b shows a WSU slice. Each cycle the WSU selects N weight and activation pairs as inputs to the N back-end multipliers. The weight inputs come directly from the WS and no further movement of weights is performed by the hardware. An *(h+d+1)-to-1* multiplexer matches each w_i weight with the appropriate activation a_i as directed by the corresponding ws_i signal. The first multiplexer input implements the case where a weight stayed at its original dense schedule position, another h inputs implement lookahead and the final d inputs implement lookaside.

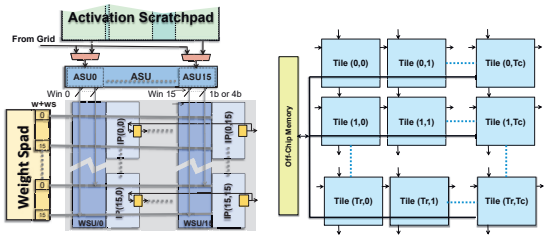
Activation Select Unit: For each weight w_i there are $h + 1$ activations, $A_{i,0}$ through $A_{i,h}$, that implement the lookahead window. The ASU in Figure 5c ensures that the physical and the logical lookahead order of the activations coincide. This allows WSU to implement lookahead and lookaside by statically assigning $A_{lane,lookahead}$ signals to multiplexer inputs. For example, the lookaside 1 connection for w_2 is to $A_{3,1}$ and its lookahead 2 connection is to $A_{2,2}$.

The ASU contains $h + 1$ *Activation Block Registers* (ABRs) each holding N input activations. Each ABR contains the N activations needed by all weight lanes at some specific lookahead distance $l = 0$ to h. The ABRs operate *logically* as a circular queue with the *head* register pointing to the ABR holding the activations at $lookahead = 0$. This implements the sliding lookahead window. An array of $h + 1$ *(h+1)-to-1* multiplexers shuffle the ABR outputs on the $A_{lane,lookahead}$ signals maintaining the logical order WSU expects. This way no data moves between the ABRs avoiding the energy that data copying would require. The ALC metadata from WM is used to advance the *head* register and implements the sliding lookahead window and also allows skipping entirely over schedule columns where all weights happen to be ineffectual.

An *Activation Buffer* (AB) buffers activations as they are read from AS. The AB has $h + 1$ banks, each connected to one ABR via a dedicated read port. This way, any number of ABRs can be updated per cycle concurrently effectively advancing the lookahead window as instructed by the ALC.

5.2 TCLe and TCLp: Zero-Bit Skipping Back-Ends

TCLe introduces a back-end which aims to process *only* the non-zero activation bits bit-serially so that total execution time scales proportionally. For example, ideally, *TCLe* will process the activation value {0000 0000 1000 1111b} over 3 cycles respectively multiplying the corresponding weight by the following powers of two: $\{+2^7, +2^4, -2^0\}$ (Booth-encoding). *TCLe* modifies the *Pragmatic* accelerator (*PRA*) design for its back-end [1]. Like *PRA*, *TCLe* processes activations bit-serially one power of two at a time. A per ABR unit converts the activations into a stream of effectual powers of two, or *oneffsets* after applying a modified Booth encoding. *TCLe* uses shifters to multiply weights with oneffsets and the result is added or subtracted via the adder tree according to



(a) TCLe tile **(b)** Overall Architecture

Figure 7. Evaluated TCL Tile and Overall Chip Architecture

the oneffset sign. Since each PE is slower than a bit-parallel PE *TCLe* needs more PEs to exceeds the throughput of an equivalent bit-parallel design. Accordingly, the configuration presented processes 16 activation windows concurrently reusing each weight spatially across 16 PEs.

Figure 6 shows a *TCLe* PE row where the single bit-parallel PE unit of Figure 5 has been replaced by a row of 16 simpler bit-serial PEs. The key modifications over *Pragmatic* are the inclusion of the WSU and ASU slices and the ability to move partial sums by one column using a per row ring to support additional dataflows and to avoid having to broadcast any activation to any PE column. Specifically, the original WSU is sliced in 16 columns, WSU/0 through WSU/15. Each PE has a 16-input adder tree and instead of 16 multipliers it has 16 shifters. Each of these shift the 16b weight input as directed by the activation oneffset input. All PEs share the same w and ws signals and perform exactly the same lookahead and lookaside activation selections. Unlike Figure 5 the multiplexers here select 4b activation oneffsets greatly reducing area. These oneffsets encode a shift by up to 3 positions plus a sign and an enable. For each column, a corresponding ASU slice provides as before data for 16 activation groups, one per weight lane, each containing data for h activations to support lookahead. Unlike Figure 5 the ASU provides 4b oneffsets. Since all WSU columns execute the same weight schedule, all 16 ASU slices access the activation buffer in tandem and share the same activation selection logic and signals.

TCLp is a lower cost design that exploits the variable, dynamic and per group precision requirements of the activations [9] to skip only some of the zero bits (prefix and suffix) of the activations. For example, ideally, *TCLp* will process the activation value {0000 0000 1000 1110b} in 7 cycles skipping the 8 prefix zero bits and the 1 trailing zero bit. The implementation of *TCLp* is virtually identical at the block level to that of *TCLe*. The primary difference is that the ASU sends activations a single bit at a time and the PEs processes them bit-serially similar to *Dynamic Stripes* [9, 11]. The overall cost is lower: one wire is needed per activation, there are no shifters, and the input width of the adder tree is 16b.

5.3 Overall Architecture

Tiles: Our processing elements can be organized in numerous ways to construct an accelerator. Here we use a hierarchical organization comprising several tiles connected in a grid. As Figure 7a shows, each tile comprises a 16×16 PE grid. The PEs along the same column share the same AS and ASU slice. The PEs along the same row share the same WS. Each PE has several output psum registers. **Tile Grid:** Each tile delivers the equivalent of 256 $16b \times 16b$ multiplications or more depending on activation bit and weight sparsity. Several tiles are connected together into a grid where each tile can broadcast a set of activations to all other tiles as shown in Figure 7b. **Memory System:** The per tile AS and WS are banked to sustain the bandwidth needed by the PEs. Data is loaded from an off-chip memory and is copied to individual AS or WS tiles or multicast to multiple ones. *TCL* uses dynamic per group precision adaptation to reduce off-chip traffic [11]. The total on-chip WS and AS capacity is selected so that for most layers each input activation and weight needs to be read at most once as per the approach of Siu *et al.* [36]. **Data Reuse:** Our hierarchical PE and tile organization exploits data reuse across several dimensions in space and time and avoids moving weights and activations multiple times. Within each tile each activation is shared *spatially* along each of the 16 PEs per column, whereas each weight is shared spatially along the 16 PEs per row. By exploiting the output partial sum registers (four in the configurations studied), activations and weights can be reused another 4× also in time. Since each non-zero activation takes multiple cycles to process new weights are not read from WS every cycle and thus weights are further reused in time. **Dataflow:** The resulting organization can support several dataflows. For the purposes of this work we assign each filter to a specific tile and PE row. Psums move within each PE row in a round-robin fashion to avoid having to move activations across columns. For this purpose activations are distributed across tiles and are statically partitioned along the x dimension along PE columns. For a typical conv layer, the dataflow can be described as follows: A set of filters are loaded into WS, one filter assigned to each PE row. Each cycle a PE processes 16 weight and activation pairs, each from a different input channel. The filters in WS are computed with all activations present in AS before a new set of filters are loaded. By design each value is reused spatially along PE columns (activations) and rows (weights). Further temporal reuse may be possible by exploiting the four psum registers per PE. **Other Layers:** *TCL* matches the performance of an equivalent bit-parallel accelerator for pooling layers whereas for fully-connected layers *TCL* speedup will come only from removing zero weights. The adder-tree based CEs will underutilized for some layers, such as the depthwise part of depthwise-separable convolutional layers, which don't reuse activations over multiple filters. The Tartan extension to Stripes [10] combined with per group precision adaptation [11] could improve performance for fully-connected layers at additional cost, an option we do not evaluate. Long-Short Term Memory layers which perform element wise multiplications. However, generally these layers account for a small fraction of overall execution time. Alternatively, we can introduce a small vector unit similar to the NVDLA [30] or the TPU [22].

Table 2. Baseline *DaDianNao++* and *TCL* configurations.

DaDianNao++* or *TCL			
Tiles	4	**Filters/Tile**	16
AS/Tile	$32KB \times 32$ Banks	**Weights/Filter**	16
WS/Tile	$2KB \times 32$ Banks	**Precision**	16b
PSum SPad/PE	128B *DaDianNao++* / 8B *TCL*		
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		

5.4 A Reduced Memory Overhead Front-End

As presented *TCL*'s front-end uses per weight multiplexer signals (WS – Figure 5c) which allow each weight lane to perform a weight promotion *independently* of the others. However, these signals represent a memory overhead. Reducing this overhead is preferable and more so the narrower the weight data width. To this end, we make the following observations: 1) Using per weight WS signals amounts to over-provisioning as, when considering all WS signals per PE, not all combinations are valid. 2) Eliminating even some of the valid combinations — e.g., never occurring or infrequent ones — may not adversely affect *TCL*'s ability to exploit *enough* of the sparsity. Accordingly, we can restrict the *combinations* of weight movements that the *TCL* front-end supports and thus reduce the number of bits needed to specify which schedule to use at every step. For example, we can store a *schedule select* field (SS) per group of weights. *TCL* can expand the SS into per weight WS signals in the tiles, a surgical modification to the design. For example, a 4-bit SS field per group of 16 weights can support $2^{SS} = 16$ different schedule patterns, each mapping to a $3b \times 16 = 48b$ vector comprising 16 WS signals. The mapping of SS signals to WS can be static or programmable. In the latter case it can be provided at an appropriate granularity such as per filter or per layer. For our example, a $16x48b$ table can map these SS signals to a set of 16 schedule steps per filter. Profiling shows that such an arrangement will not impact performance considerably for the networks studied (e.g., it covers 96% of all scheduling steps in GoogleNet-ES). Due to the limited space we do not evaluate this design further here.

6 Evaluation

We model execution time via a custom cycle accurate simulator. All area and energy measurements were performed over layout using circuit activity for representative data inputs. The layouts were generated for a TMSC 65nm technology using Cadence Innovus after synthesizing them with Synopsys Design Compiler. 65nm is the best technology available to us. We used the typical case design library as it yields more pessimistic results for our designs which scale better than bit-parallel designs for the worst design corner. SRAMs were modeled via CACTI [27]. We size the on-chip memories so that each weight and activation has to be read at most once per layer for *most* layers according to the method of Siu *et al.,* [36]. Off-chip memory energy consumption is modeled using Micron's DDR4 power calculator [26] along with access counts from the cycle-accurate simulations. All designs operate at 1GHz and the results are normalized against the *DaDianNao++* accelerator design detailed in Table 2. *DaDianNao++* uses DaDianNao-like tiles [1] with 16 multipliers per PE and 16 PEs per tile. Unlike DaDianNao, *DaDianNao++* is tiled as in Figure 7b and uses the same on-chip memory hierarchy as *TCL*. Normalizing over *DaDianNao++* enables comparisons with prior work. Since SCNN [31] was evaluated with 1K multipliers we configure *TCL* and *DaDianNao++* with 4 tiles. We first explore the designs assuming infinite off-chip bandwidth, and then consider several off-chip memory nodes. We use zero compression and fine-grain per group precision to reduce off-chip bandwidth for all layers [9, 11]. However, other compression schemes could be used [15]. We use unmodified pruned network models where available [32, 38]. We model pruning of MobileNet v1 and Bi-LSTM for 75% sparsity. To approximate the distribution of sparsity in the corresponding pruned and fine-tuned networks, we follow magnitude-based pruning rules on a per-layer basis, as proposed in [28, 41]. The dataflow is optimized to minimize energy for *DaDianNao++*.

6.1 Front-End Weight Skipping

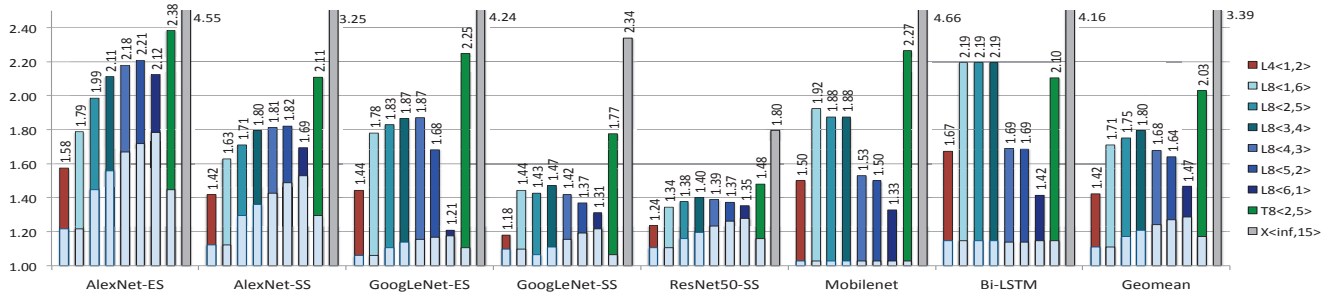
Figure 8a reports the relative speedup of just the front-end weight skipping of *TCL* vs. *DaDianNao++*. In this configuration, *TCL* exploits weight sparsity *only*. The bottom portion of each bar shows speedup when only lookahead is possible. The top portion demonstrates the additional speedup that is achieved by adding lookaside. Configurations are labelled with their lookahead distance, h, their lookaside distance, d, their connectivity shape, and their multiplexer size, n, as *Shape-n<h,d>*. We restrict our attention to designs with small input mutliplexers such that $h + d + 1 = n$, $n = \{4, 8\}$ in order to limit power and area overheads. The two shapes considered are those detailed in Section 3. All experiments make use of the scheduling algorithm described in Section 4. The plots also include the potential speedup with *X<inf,15>*, an *impractical* design which allows arbitrary weight promotion within each filter lane. This serves as an upper bound on the potential speedup from leveraging weight sparsity for *TCL*.

Our hardware/software weight skipping approach robustly improves performance across all networks with the benefits remaining high even for the more recent and tightly optimized MobileNet. In addition, *most* of the potential speedup is attained using very modest hardware. Indeed, the most performant configuration achieves, on average, 60% of the potential of *X<inf,15>* at a small fraction of the overhead. Additionally, the efficacy of lookaside and its ability to balance work over multiple multiplier lanes is clear, with most of the average speedup being due to the addition of lookaside functionality. Further, the superiority of hardware/software co-design is evident in the improvement that the Trident (T8<2,5>) shape - which was designed alongside the scheduling algorithm - achieves over the L8<2,5> configuration, offering an additional 16% improvement at little-to-no area overhead. What's more, the Trident interconnect is the best performing design across all networks apart from Bi-LSTM, in which the structured sparsity present in the weights (an artifact of the pruning method) marginally favors the 'L' shaped interconnect. The same structured sparsity cannot be adequately leveraged by the lookahead-only designs for Mobilenet and Bi-LSTM.

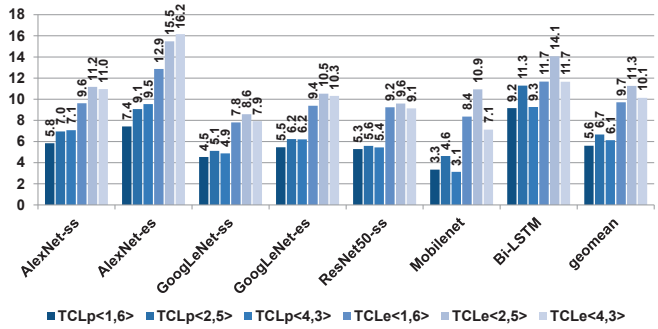
6.2 Front-End and Back-End

Performance: Figure 8b reports the performance of *TCLe* and *TCLp* configurations relative to *DaDianNao++* for *all* layers. Adding the capability to skip ineffectual bits improves performance robustly across all models. The benefits are much higher for the '-ES' variants than for '-SS', suggesting that optimizing for energy efficiency (Yeng *et al.,* prioritize pruning according to execution frequency [38]) also benefits *TCL*. As expected, all *TCLe* configurations outperform any *TCLp* configuration. The benefits are lower than the ideal as a result of: 1) cross activation lane synchronization, 2) zero weights that were not removed, 3) underutilization due to layer dimensions, and 4) off-chip stalls. The $\langle 2, 5\rangle$ designs use the Trident interconnect and for this reason sometimes outperform even the $\langle 3, 4\rangle$ ones which use the L interconnect.

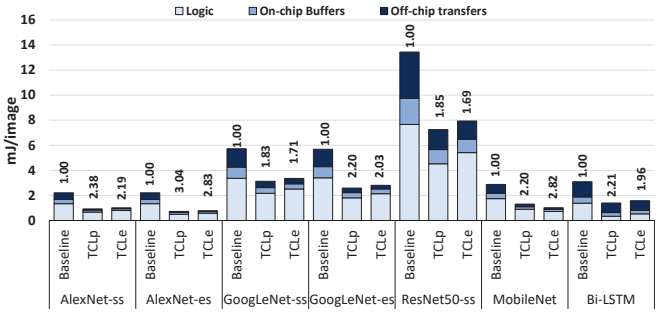
Figure 9 shows a breakdown of where time goes for the $T8\langle 2, 5\rangle$ configuration. Front-end time can be broadly categorized into effectual and ineffectual work. Processing of effectual weights is split into three categories: a) *lookahead promoted*, *lookaside promoted*, and c) *effectual unpromoted*. Ineffectual work emerges when processing zero weights that are either part of layer sparsity which the scheduler fails to fill-in, or that are induced by zero-padding. Zero-padding occurs when the number of filters in a layer is such that not all filter lanes can be utilized simultaneously, or when the channel depth is not a multiple of the filter lane width. Figures 9(a)-(g) report the above four categories per network (for some representative layers and over *all* layers). The diamond markers show the fraction of ineffectual work due to



(a) Speedup with Front-End Weight Skipping Only. Bottom: Lookahead only. Top: Lookahead and Lookaside.

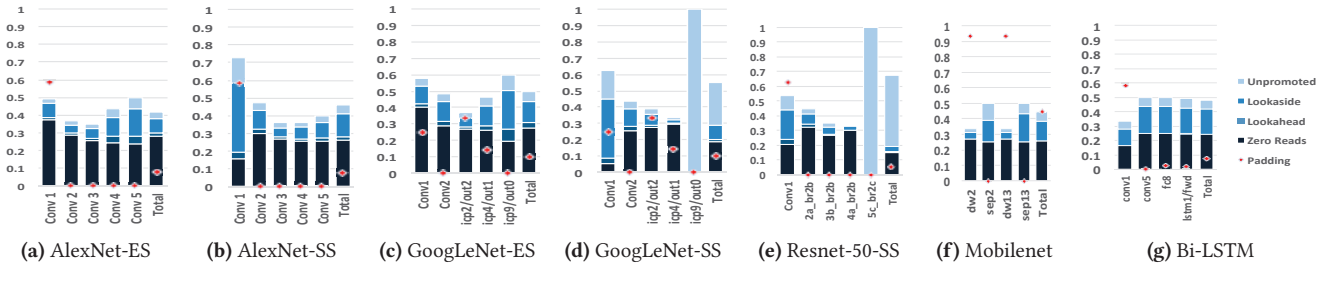


(b) Speedup with Activation Back-Ends

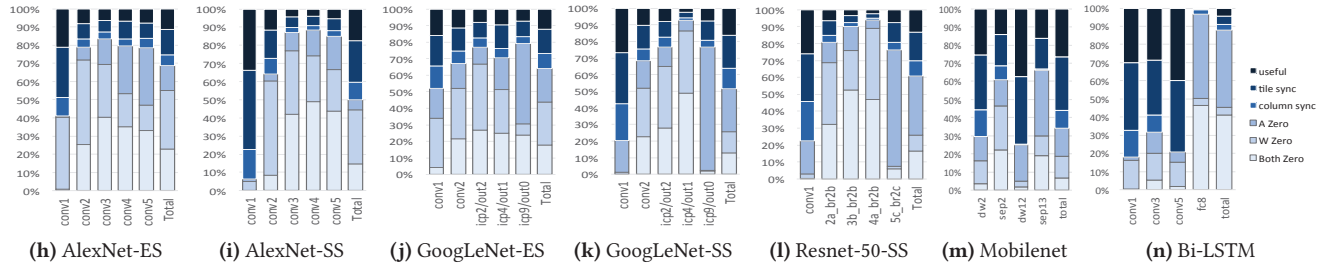


(c) Energy Breakdown and Relative Energy Efficiency

Figure 8. TCL front-end, TCLe and TCLp.



(a) AlexNet-ES **(b)** AlexNet-SS **(c)** GoogLeNet-ES **(d)** GoogLeNet-SS **(e)** Resnet-50-SS **(f)** Mobilenet **(g)** Bi-LSTM



(h) AlexNet-ES **(i)** AlexNet-SS **(j)** GoogLeNet-ES **(k)** GoogLeNet-SS **(l)** Resnet-50-SS **(m)** Mobilenet **(n)** Bi-LSTM

Figure 9. Execution Time Breakdown with TCLe $T8\langle 2,5\rangle$. (a)-(g) Front-End Only, (h)-(n) Front-End and Back-End.

padding in the *original* dense schedule. The scheduler can promote effectual weights into channel-induced padding and does so as the results show. The front-end captures most of the sparsity across all layers, achieving at least 2× speed-up for most. Lookaside promotions generally contribute the most in reducing ineffectual work. The breakdown for *TCLe* in Figures 9(h)-(n) shows that performance loss is partly due to processing the zero weights that the frontend fails to remove, along with zero padding ("W Zero" and "Both Zero"). This ineffectual work is amplified with the backend, as each ineffectual weight now spends more than 1 cycle in the multiplier, and hence accounts for a large amount

of the time breakdown. Cross-lane synchronization (along each CE column "Column Sync" and across CE columns "Tile Sync") also consumes multiplier cycles, as *TCLe* performs an implicit synchronization at the end of each group of concurrently processed activations. Multipliers will therefore be idle while they wait for the activation with largest effectual bit count per group. Increasing the lookahead window further exacerbates this phenomenon, as more activations are being processed in any given group. Lookaside has no further effect in this regard, as it requires no additional activations. Dense layers, such as `5c_br2c`, spend a proportionally larger amount of time processing zero activations ("A Zero"), and essentially operate as the *Pragmatic* accelerator. While *TCLe* outperforms *TCLp*, it is a more expensive design and application-specific trade offs should be taken into account.

Overall Energy Efficiency: Figure 8c reports a breakdown of the energy spent per frame for compute logic, on-chip and off-chip memory transfers. The energy efficiency relative to *DaDianNao++* is reported on top of each bar. Due to space limitations we limit attention to the T⟨2, 5⟩ configuration and consider the convolutional layers only in order to enable a comparison with SCNN in the next section. However, all *TCL* configurations remain more energy efficient even when considering all layers. Significant energy reductions come from the compute logic and off-chip transfers. Energy efficiency is influenced by 1) the level of sparsity in the weights, and 2) the level of bit sparsity in the activations. Since bit sparsity levels are high for all the models, variation in weight sparsity is the primary reason for the differences observed. It is for this reason that ResNet50-SS benefits the least. Generally, *TCLp* is more energy efficient than *TCLe* as it is a lower cost design. For *TCLp*, energy efficiency ranges from 1.83× for GoogLeNet-SS up to 3.04× for AlexNet-ES with the average being 2.22×. Meanwhile, the energy efficieny of *TCLe* is 1.69× for ResNet50-SS and up to 2.83× for AlexNet-ES with an average of 2.13×.

Area: Table 3 reports the area for various configurations. For clarity, we report detailed breakdowns only for *TCLe*⟨1, 6⟩, *TCLp*⟨1, 6⟩, and *DaDianNao++*. The area vs. performance trade off is sublinear which suggests that even if performance could scale linearly for *DaDianNao++* it would still trail in performance per area. In practice performance in *DaDianNao++* scales sublinearly with area as the typical hyper-parameter values (filter counts, and feature map and filter dimensions) result in higher underutilization for wider configurations. The area differences among the configurations are negligible since the sum of lookahead and lookaside is the same. Most of the area is taken by the on-chip memories, a more energy efficient and thus higher performing choice for energy-limited systems; doing so reduces off-chip accesses the energy of which dwarfs all other uses [37].

Off-Chip Memory: Figure 10 shows the speedups for *TCLp* and *TCLe* with the T⟨2, 5⟩ interconnect and for different off-chip memory configurations [18–21]. The rightmost point

Table 3. *TCLe* and *TCLp*: Area mm^2.

	TCLe	*TCLp*	
Config. ⟨h, d⟩	**L8⟨1, 6⟩**		***DaDN++***
Compute Core	19.28	9.22	3.27
Weight Memory	3.57	3.57	3.57
Activation Select Unit	0.15	0.03	-
Act. Input Buffer	0.17	0.17	0.09
Act. Output Buffer	0.11	0.11	0.11
Activation Memory	54.25	54.25	54.25
Dispatcher	0.37	0.39	-
Offset Generator	2.89	-	-
Total	80.8	67.75	61.29
Normalized Total	1.32×	1.11×	1.00×
Config. ⟨h, d⟩	**Normalized Total**		
L8⟨2, 5⟩	1.34×	1.10×	1.00×
L8⟨4, 3⟩	1.37×	1.11×	1.00×
T8⟨2, 5⟩	1.35×	1.10×	1.00×

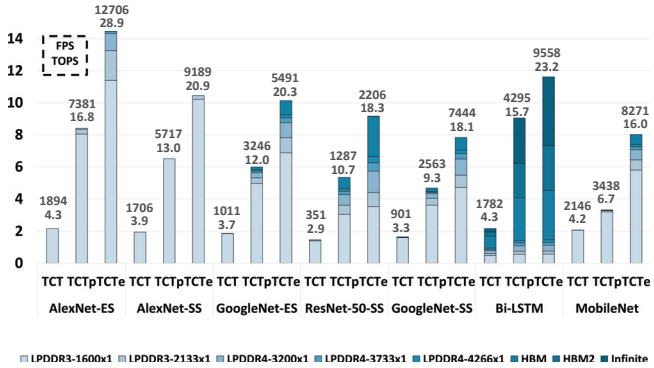


Figure 10. *TCL*: Speedup with off-chip memory technologies.

represents the infinite available off-chip bandwidth case used thus far. Even with the low-end LPDDR3-1600 our designs outperform the baseline. *TCLe* places more pressure than *TCLp* since it is faster. GoogLeNet and ResNet50 need an HBM memory to achieve peak performance yet even with a dual channel LPDDR4-3200 achieve performance within 16% and 5% of peak respectively. We also report the peak absolute performance per network as frames per second (fps) and TOPS by highlighting the corresponding configuration that requires the least capable off-chip memory system.

6.3 Sensitivity to Weight Sparsity

This section presents evidence that *TCL*'s front-end is resilient to changes in sparsity *patterns* and *levels* and thus that it should be broadly applicable. Additionally, these results further justify our choices of interconnect and scheduling algorithm. We report *TCL*'s performance as weight sparsity changes and for a variety of hypothetical models. To this end, we present performance results for various front-end designs operating on randomly sparsified 3×3 filters with 512 channels after scheduling. Each experiment is run with a single filter, and we generate 100 filters per target sparsity, in increments of 10% sparsity. Figure 11a demonstrates

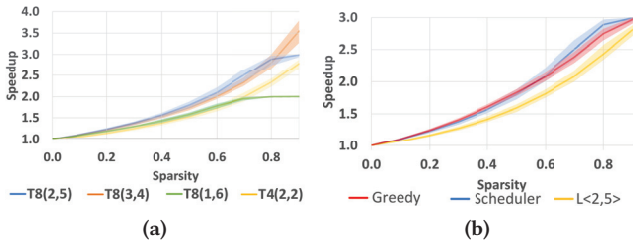


Figure 11. Speedup vs. weight sparsity. Bands show the range of results, and lines show geomean speedup. (a) The effect of Lookahead and Lookaside. (b) The effect of the scheduler and the interconnect.

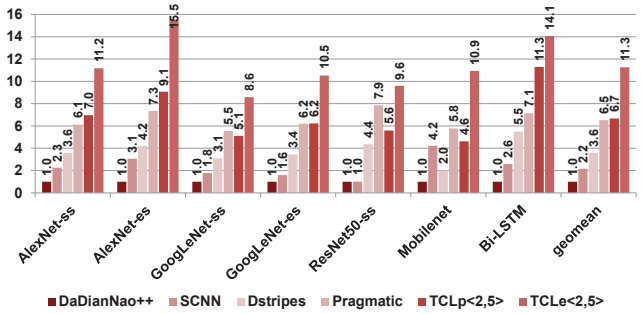


Figure 12. Performance vs. other accelerators

that *TCL* delivers most the benefits possible across the range of weight sparsity levels and for a variety of sparsity patterns. Furthermore, at all but the highest sparsity levels, the T8⟨2, 5⟩ design achieves superior results to a T8⟨3, 4⟩ design, whilst requiring the same input mux size and reduced lookahead. Given that the networks we study are between 45% and 87% sparse, the added hardware cost of increasing lookahead is not pragmatic.

We explore the other side of this trend by studying two designs with smaller overhead: one that requires reduced lookahead (T8⟨1, 6⟩), and one that requires smaller 4-input muxes (T4⟨2, 2⟩). The performance degradation of these configurations is notable at almost all sparsity levels. For example, at 70% weight sparsity, T8⟨2, 5⟩ outperforms T8⟨1, 6⟩ and T4⟨2, 2⟩ by 29% and 26%, respectively.

Figure 11b explores the effect of the scheduler at various sparsity levels and the effect of the hardware interconnect. On average, the optimized scheduler outperforms the greedy one as sparsity increases above 60%. However, even the simple greedy scheduler delivers adequate performance suggesting that a low runtime overhead scheduler implemented in software or hardware could be sufficient in certain applications. The figure shows a contrast in performance between the Trident- and L-shaped interconnects, emphasizing the benefits of HW/SW co-design.

6.4 Comparison With Other Accelerators

Figure 12 reports the relative performance of SCNN, Pragmatic, Dynamic Stripes, *TCLp* and *TCLe* for the T⟨2, 5⟩ configuration. As expected *TCLp* outperforms Dynamic Stripes and *TCLe* outperforms Pragmatic. In most cases *TCLp* outperforms or performs similar to Pragmatic as well. These results further support our observation that the front-end and the back-end benefits for *TCL* are nearly multiplicative.

SCNN is the state-of-the-art accelerator for sparse DNNs. We model the originally proposed 8×8 tile with $1K$ multipliers. Given that SCNN's peak compute bandwidth for fully-connected layers is only 1/4 of that for convolutional layers, we limit attention only to convolutional layers. Both *TCLp* and *TCLe* outperform SCNN. SCNN targets zero values only and thus cannot benefit from non-zero activations. Further, it incurs underutilization and inter-filter imbalance [31]. These are pronounced for ResNet50-SS which has many 7×7 feature maps which don't map well to the SCNN tiles, each requiring at least a 2×2 slice of the feature map for full utilization. We also experimented with SCNNp, a variant of SCNN where the multiplier arrays have been replaced with bit-serial MAC units to exploit dynamic precision variability as in *TCLp*. To compensate for the loss of peak compute bandwidth, SCNNp uses 16× more tiles (32×32), similar to *TCLp*. SCNNp illustrates that our choice of ineffectual computations to exploit is in principle compatible with SCNN's approach: SCNNp can outperform SCNN on the first layer of each network (by up to 3.4× for AlexNet-ES), where the feature map is largest in the x and y dimensions. However, SCNNp's performance degrades in deeper layers due to inter-tile imbalance; SCNNp cannot therefore sustain full utilization for feature maps smaller than 64×64, which represent most layers of modern networks designed for IMAGENET. Regardless, we believe that further exploration is warranted. Finally, we measured the relative off-chip traffic of *TCL* vs. SCNN assuming that SCNN has enough on-chip memory to read each filter only once per layer. On average *TCL* reads 2.4× less values from off-chip. At worst for GoogLeNet-SS *TCL* off-chip traffic is 2.1× less than SCNN's and at best it is 2.5× less for AlexNet-ES. This reduction is mostly due to the use of the per value group compression scheme.

Energy efficiency vs. SCNN : Compared to SCNN with its original weight buffer configuration and for the convolutional layers only, *TCLp* and *TCLe* with the T⟨2, 5⟩ configuration are on average 2.31× and 2.23× more energy efficient respectively. Most of the benefits come from the reduced off-chip data movement since SCNN needs to stream the weights multiple times. If we were to consider only the on chip energy, *TCLp* and *TCLe* will still be 1.74× and 1.59× more energy efficient than SCNN on average. The energy efficiency advantage of our designs would be higher if we were to consider all layers since SCNN is not designed to be efficient for layers other than the convolutional ones. We leave for future work an investigation of our proposed extensions to SCNN, SCNNp and SCNNe.

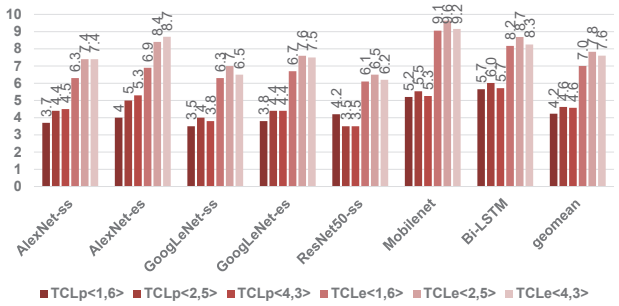


Figure 13. Speedup with 8b quantization

6.5 Quantization

Figure 13 reports speedups over *DaDianNao++* where *all* systems use 8-bit quantization. For these experiments we use linear quantization for all layers. The benefits remain considerable. While the activations now use a shorter datatype, there is still significant variability in their dynamic precision requirements and in their ineffectual bit content which are successfully exploited by *TCLp* and *TCLe*, respectively. The quantization method used is range-oblivious, that is while it will rightfully reduce the value range to fit within 8b for the layers where this is necessary, it will also *unnecessarily* expand the value range to 8b for layers that could have used a lower precision [11]. The benefits would have been higher with range-aware quantization (e.g., trimming precisions for GoogleNet-ES using profiling results in a model that uses at most 8b for all layers and less than that for most layers). The more quantization can reduce data widths the more preferable the modified design of Section 5.4 becomes.

7 Related Work

We restrict attention to accelerators that exploit weight and activation sparsity. Table 4 highlights the most relevant characteristics of each design: (1) for which input data a) it skips the multiply-accumulate computation, b) it avoids a memory reference, c) it performs a reduced cost multiply-accumulate, or d) it performs a reduced cost memory access, (2) how is the input data routed to the appropriate compute unit or storage unit, and 3) the ordering used to compute inner-products.

Cnvlutin skips both the computation and the memory access for ineffectual activations (IA). It requires an independent read port per group of weights that pair up with each activation [2]. ZeNA also skips zero activations [24]. Cambricon-X exploits ineffectual weights (IW) in an inner product based accelerator [39]. It compacts non-zero weights in memory and tags them with deltas (distance between weights). Each cycle each PE fetches 16 weights and selects the corresponding 16 activations from a vector of 256. It uses a 256-wide *input activation crossbar* to pair up activations with the corresponding weights. This approach is similar to *TCL* with a very large 16x16 lookahead window and encoded mux selects. It requires a memory interface for 256 activations, 16 times that of DianNao [4]. The authors discuss that this activation bandwidth makes their approach impractical for scalable accelerators like DaDianNao [5]. Cambricon-S [40] leverages a co-designed pruning algorithm that results in structural sparsity, leading to a reduced complexity, yet dense, indexing and routing module. *TCL* fully supports this form of structural sparsity without requiring it. Cambricon-S skips all ineffectual activations. As demonstrated in Section 2, the potential workload reduction of this is smaller compared to skipping ineffectual activation terms. By design, SCNN's performance suffers for fully-connected layers and dense networks, both of which *TCL* handles well. Accordingly, a designer must take into account the relative pros and cons of each approach to decide which design fits the specific application best. *TCL* skips computations and memory accesses for ineffectual weights, albeit to a different degree than SCNN and Cambricon-X/-S. *TCL* reduces the bandwidth and energy cost of the memory accesses for both ineffectual and effectual activations (EA). It matches activations and weights using a hybrid *input weight-static/activation-dynamic* approach since it utilizes a *sparse shuffling network* for the input activations, and restricted static scheduling for the weights. To capture sparsity, SCNN, Cambricon-S(-X) use *dense* hardware interconnects instead.

Table 4. Qualitative Comparison of Accelerators.

	Skip MACC	**Skip Memory Access**	**Reduced MACC**	**Reduced Memory Access**
Cnvlutin, ZeNA	IA	IA	-	-
Cambricon-X	IW	IW	-	-
Cambricon-S	IA+IW	IA+IW	-	-
SCNN	IA+IW	IA+IW	-	-
Dynamic Stripes	-	-	IA+EA	IA+EA
Pragmatic	-	-	IA+EA	IA+EA
TCLe/TCLp	IW	IW	IA+EA	IW+EW+IA+EA
	Data Routing Type & Mechanism			**Inner Spatial Dataflow**
Cnvlutin	Weight-Dynamic/Activation-Static Sparse@Input: Independent Weight Ports			Dot Product Reduction
Cambricon-X/-S	Weight-Static/Activation-Dynamic Dense@Input: Activation Crossbar			Dot Product Reduction
SCNN	Weight-Dynamic/Activation-Dynamic Dense@Output: Product Crossbar			Cartesian Product
TCLe/TCLp	Weight-Static/Activation-Dynamic Sparse@Input: Sparse Shuffling Network for Activations			Dot Product Reduction

8 Conclusion

We believe that *TCL*'s approach to exploiting sparsity can be adapted to additional applications during inference and training to further facilitate optimizations that manifest as weight or activation value- or bit-sparsity. *TCL*'s lightweight approach with a scheduler implemented in software or in hardware presents an interesting framework for exploring such applications.

Acknolwedgements: This work was supported in part by an NSERC Discovery Grant, an NSERC DND Discovery Supplement, and the NSERC COHESA Research Network.

References

[1] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic Deep Neural Network Computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. 382–394.

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *2016 IEEE/ACM International Conference on Computer Architecture (ISCA)*.

[3] Peter Brucker. 2001. *Scheduling Algorithms* (3rd ed.). Springer-Verlag, Berlin, Heidelberg.

[4] T Chen, Z Du, N Sun, J Wang, C Wu, Y Chen, and O Temam. 2014. 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*.

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and O. Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. 609–622. https://doi.org/10.1109/MICRO.2014.58

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. 367–379.

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

[8] Ronan Collobert and Jason Weston. 2008. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*. ACM, New York, NY, USA, 160–167. https://doi.org/10.1145/1390156.1390177

[9] Alberto Delmas, Patrick Judd, Sayeh Sharify, and Andreas Moshovos. 2017. Dynamic Stripes: Exploiting the Dynamic Precision Requirements of Activation Values in Neural Networks. *CoRR* abs/1706.00504 (2017). arXiv:1706.00504 http://arxiv.org/abs/1706.00504

[10] Alberto Delmas, Sayeh Sharify, Patrick Judd, and Andreas Moshovos. 2017. Tartan: Accelerating Fully-Connected and Convolutional Layers in Deep Learning Networks by Exploiting Numerical Precision Variability. *CoRR* abs/1707.09068 (2017). arXiv:1707.09068 http://arxiv.org/abs/1707.09068

[11] A. Delmas, S. Sharify, P. Judd, K. Siu, M. Nikolic, and A. Moshovos. 2018. DPRed: Making Typical Activation Values Matter In Deep Learning Computing. *ArXiv e-prints* (Dec. 2018). arXiv:1804.06732

[12] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[13] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. 2016. ESE: Efficient Speech Recognition Engine with Compressed LSTM on FPGA. *CoRR* abs/1612.00694 (2016). arXiv:1612.00694 http://arxiv.org/abs/1612.00694

[14] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 243–254. https://doi.org/10.1109/ISCA.2016.30

[15] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015). arXiv:1510.00149 http://arxiv.org/abs/1510.00149

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780.

[17] J.L. Holt and T.E. Baker. 1991. Back propagation simulations using limited precision calculations. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, Vol. ii. 121–126 vol.2. https://doi.org/10.1109/IJCNN.1991.155324

[18] JESD209-3C 2015. *Low Power Double Data Rate 3 SDRAM (LPDDR3)*. Standard. JEDEC.

[19] JESD209-4-1 2017. *Addendum No. 1 to JESD209-4, Low Power Double Data Rate 4X (LPDDR4X)*. Standard. JEDEC.

[20] JESD209-4B 2017. *Low Power Double Data Rate 4 (LPDDR4)*. Standard. JEDEC.

[21] JESD235A 2015. *High Bandwidth Memory (HBM) DRAM*. Standard. JEDEC.

[22] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. 1–12.

[23] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial Deep Neural Network Computing . In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*.

[24] D. Kim, J. Ahn, and S. Yoo. 2018. ZeNA: Zero-Aware Neural Network Accelerator. *IEEE Design Test* 35, 1 (Feb 2018), 39–46. https://doi.org/10.1109/MDAT.2017.2741463

[25] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.* 19, 2 (June 1993), 313–330.

[26] Micron. 2017. Calculating Memory Power for DDR4 SDRAM. Technical Note TN-40-07. https://www.micron.com/resource-details/868646c5-7ee2-4f6c-aaf4-7599bd5952df.

[27] Naveen Muralimanohar and Rajeev Balasubramonian. [n. d.]. CACTI 6.0: A Tool to Understand Large Caches.

[28] Sharan Narang, Gregory F. Diamos, Shubho Sengupta, and Erich Elsen. 2017. Exploring Sparsity in Recurrent Neural Networks. *CoRR* abs/1704.05119 (2017). arXiv:1704.05119 http://arxiv.org/abs/1704.05119

[29] Milos Nikolic, Mostafa Mahmoud, Yiren Zhao, Robert Mullins, and Andreas Moshovos. 2019. Characterizing Sources of Ineffectual Computations in Deep Learning Networks. In *International Symposium on Performance Analysis of Systems and Software*.

[30] NVIDIA. [n. d.]. NVIDIA Deep Learning Accelerator. ([n. d.]). nvdla.org

[31] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer,

Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/3079856.3080254

[32] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2017. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. https://github.com/IntelLabs/SkimCaffe. In *5th International Conference on Learning Representations (ICLR)*.

[33] Michael L. Pinedo. 2008. *Scheduling: Theory, Algorithms, and Systems* (3rd ed.). Springer Publishing Company, Incorporated.

[34] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 267–278.

[35] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2014. ImageNet Large Scale Visual Recognition Challenge. *arXiv:1409.0575 [cs]* (Sept. 2014). arXiv: 1409.0575.

[36] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, and Andreas Moshovos. 2018. Memory Requirements for Convolutional Neural Network Hardware Accelerators. In *IEEE International Symposium on Workload Characterization*.

[37] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. 2016. A Systematic Approach to Blocking Convolutional Neural Networks. *CoRR* abs/1606.04209 (2016).

[38] Yang, Tien-Ju and Chen, Yu-Hsin and Sze, Vivienne. 2017. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[39] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

[40] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28. https://doi.org/10.1109/MICRO.2018.00011

[41] M. Zhu and S. Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. *ArXiv e-prints* (Oct. 2017). arXiv:stat.ML/1710.01878