Laconic Deep Learning Inference Acceleration

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

Alberto Delmas Lascorz
a.delmaslascorz@mail.utoronto.ca
University of Toronto

Mostafa Mahmoud
mostafa.mahmoud@mail.utoronto.ca
University of Toronto

Milos Nikolic
milos.nikolic@mail.utoronto.ca
University of Toronto

Kevin Siu
kcm.siu@mail.utoronto.ca
University of Toronto

Dylan Malone Stuart
dylan.stuart@mail.utoronto.ca
University of Toronto

Zissis Poulos
zpoulos@ece.utoronto.ca
University of Toronto

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

ABSTRACT

We present a method for transparently identifying ineffectual computations during inference with Deep Learning models. Specifically, by decomposing multiplications down to the bit level, the amount of work needed by multiplications during inference can be potentially reduced by at least 40× across a wide selection of neural networks (8b and 16b). This method produces numerically identical results and does not affect overall accuracy. We present *Laconic*, a hardware accelerator that implements this approach to boost energy efficiency for inference with Deep Learning Networks. *Laconic* judiciously gives up some of the work reduction potential to yield a low-cost, simple, and energy efficient design that outperforms other state-of-the-art accelerators: an optimized DaDianNao-like design [13], Eyeriss [15], SCNN [71], Pragmatic [3], and BitFusion [83]. We study 16b, 8b, and 1b/2b fixed-point quantized models.

ACM Reference Format:
Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. 2019. Laconic Deep Learning Inference Acceleration. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322255

1 INTRODUCTION

Hardware performance improvements and software innovations have been fueling one another for decades with semiconductor technology scaling facilitating both. As hardware has become energy-constrained, improving performance necessitates increasing energy efficiency [27]. Accordingly, hardware acceleration is flourishing. This work targets hardware acceleration of Deep Learning inference, where the need for hardware acceleration, especially at the "edge", has been well documented.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322255

The Landscape of Prior Related Work: The bulk of the energy and work in neural networks during inference is due to the data transfers and the computation needed to perform multiply-accumulate operations (MACs) of weight (W) and activation (A) values into partial sums (psum). One approach to reduce energy would be to re-architect the neural model from scratch. An example is MobileNet V2 [81] which was designed from the ground up so that it can execute on mobile devices, albeit at some loss in accuracy compared to models designed for more capable hardware. Related are Network Architecture Search methods that automate the process of developing neural network architectures given a set of constraints e.g., [10, 107]. Once a model architecture has been decided, there are techniques in software and hardware that further improve its energy efficiency. These broadly fall into at least four categories depending on the property they target: 1) data reuse, 2) data type and width, 3) zero values, 4) tolerance to approximate computation.

The first class includes dataflow scheduling and data blocking which exploit data reuse to reduce the energy needed by data movement [14, 30, 52, 55, 65, 92, 95, 96, 100]. It also includes techniques that fuse computation across multiple levels [5]. Overall, these techniques maximize on-chip storage use, drastically reducing the disproportionately more expensive off-chip transfers [42].

The second class includes per layer precision selection [46, 47, 61, 76, 84], per group precision adaptation [22, 23], and quantization [11, 17, 34, 43, 49, 68, 69, 72, 74, 90, 104] all of which "shrink" the data type used, benefiting on- and off-chip storage, communication and computation. As traditional hardware supports only a few fixed-size data types, much attention has been given to quantization, broadening its applicability. Specifically, improvements made quantization possible for a wider set of models and with shorter data widths. For example, whereas initially 16b fixed-point was required by state-of-the-art image classification networks [12, 13], today many of these models can be quantized to 8b fixed-point with little or no loss in accuracy. Furthermore, ternary [60, 69, 106], logarithmic [102] or even binary quantization [18, 25, 53, 78] have been demonstrated for some networks. However, it is not currently possible to apply arbitrary forms of quantization to any network. Moreover, some forms of quantization require higher precision or even floating-point scaling or biasing at the output of certain layers [68]. Centroid-based and in general dictionary-based quantization coupled with encoding [38]

and follow-up methods [1, 16, 19, 35, 41, 45, 64, 88] also drastically reduce off-chip traffic and storage.

The third class includes techniques that target *sparsity*, that is, zero values. Zero values, be it weights or activations, result in ineffectual MACs which can be safely eliminated, saving energy and/or storage and communication [4, 15, 24, 36, 37, 54, 71, 101, 105]. Pruning further increases the presence of zeros by eliminating weights [26, 36, 39, 56, 62, 93, 98].

The fourth class of solutions use approximate computing principles. These include approximate MAC units, e.g., [63] and techniques that selectively mask computation such as conditional models [7, 44] and runtime prediction-based methods [2, 57]. Some of the aforementioned techniques such as quantization could be thought of as utilizing approximate computing principles as well.

Our Focus: This work targets *bit sparsity*, that is zero bits, as doing so exposes more ineffectual work than past approaches, and another dimension of parallelism. Specifically, not processing zero bits in a multiply-accumulate operation does not affect the outcome. The results are numerically identical. This becomes apparent if we decompose the multiplication into a series of simpler ones: for example, an $8b \times 8b$ product into 64 $1b \times 1b$ products. However, skipping zero bits may seem counter productive since highly optimized multipliers rely on bit-level parallelism processing all bits regardless of value. However, we find that neural nets exhibit behavior that favors targeting bit sparsity: First, bit sparsity is exceptionally high for both weights and activations — more than 90% for 8b and 16b models after Booth encoding (see Section 2). Second, parallelism and energy efficiency can be recuperated by exploiting inter-value and other forms of parallelism which are abundant. Exploiting bit sparsity is particularly appealing for neural nets since it: a) is complementary to data reuse, b) is amplified by advances in pruning, and c) as we show, remains high under quantization.

Contribution #1: Section 2 motivates this work by demonstrating that neural networks exhibit high bit sparsity in their activations and weights. Our findings corroborate that more than 90% of the activation bits in 16b fixed-point image classification models are zero [3]. Further, they demonstrate that high bit sparsity extends to the weights and persists: 1) in 8b or even 1b/2b quantized models, 2) in pruned models, and 3) across models targeting various applications beyond image classification.

Contribution #2: We propose *Laconic*, a hardware accelerator design that exploits bit sparsity in activations and weights. To do so *Laconic* converts (using for example Booth-Encoding) each input into a list of signed powers of two, or *terms* and then multiplies and accumulates those terms serially. While each $A \times W$ multiplication requires multiple cycles, overall throughput is higher; *Laconic* exploits the inherent parallelism of neural networks optimizing for throughput and not for individual MAC latency. As a result, overall inference latency is greatly reduced with *Laconic*.

The key architectural innovation in *Laconic* is a data-parallel histogram-based *Laconic* processing element (LPE) that is much smaller (5× in 8b) and energy efficient than a bit-parallel PE. As Section 3.2 explains, a small and energy efficient LPE is essential for *Laconic* to outperform other designs. Unfortunately, a straightforward modification of prior PE designs to exploit bit sparsity including that of Pragmatic [3] — the closest prior related work that exploits bit sparsity but for activations only — proved considerably larger than needed; it was 1.5× larger than the bit-parallel PE.

Table 1: Laconic and Other Accelerators: GOPS/W.

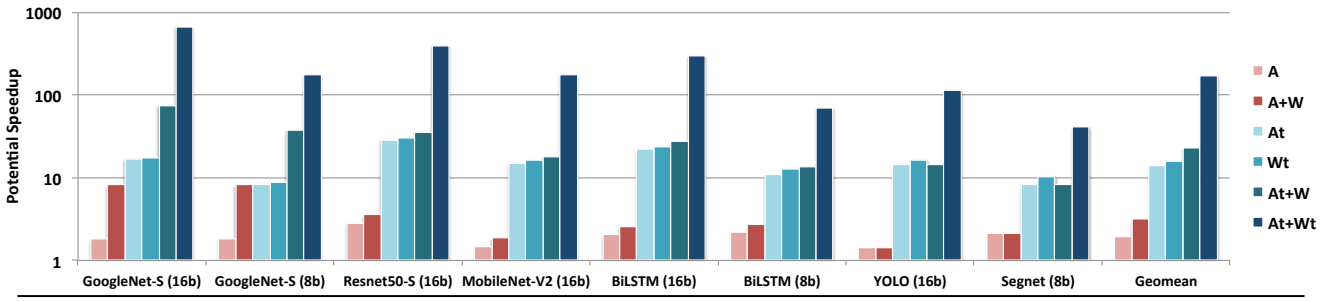
Effective GOPS/W: Iso-Area, 65nm			
8-bit Accelerators		**16-bit Accelerators**	
DaDianNao++	703	**Eyeriss**	294
BitFusion (45nm)	Section 4.5	**Pragmatic**	304
SCNN-8b	787	**SCNN-16b**	182
Laconic-8b	**805**	**Laconic-16b**	**441**
Effective GOPS/W, 15nm			
Laconic-8b	**1,997**	**Laconic-16b**	**1,092**

LPE forgoes multipliers and shifters, components that prove expensive, and instead utilizes narrow adders to perform the multiplication of terms. It decomposes the accumulation of multiple term products into three steps which collectively require much less area and energy to operate. In the first step, a unique histogram-like unit performs partial accumulation of terms according to their magnitude. In the second, an ensemble of simple concatenation units effectively reduce the histogram's multiple outputs into few partial sums. Finally, in the third step, an adder tree reduces these terms into the final sum. All three steps combined are implemented using just combinatorial logic resulting in a novel functional unit which can be pipelined as needed. The result is an accelerator that delivers best of class performance and energy efficiency compared to other state-of-the-art inference accelerators.

We further extend the *Laconic* PE to allow computation with larger data types, e.g., compute with 16b inputs while using units that are designed to process 8b. The resulting designs support spatial or temporal composition. Bit-Fusion also supports spatial and temporal composition [83] but does not exploit bit sparsity. Our spatial approach is not specific to *Laconic*. Finally, *Laconic* exposes *value-aware intra-value* parallelism, a novel form which has not be exploited before. This is bit-level parallelism across only those bits of a value that are effectual. It is different than the value-agnostic bit-level parallelism typically exploited by the functional units of conventional bit-parallel hardware.

We highlight the following key results:

- Bit sparsity is exceptionally high for 16b and 8b models and the potential for work reduction exceeds 40× for all models and is much higher than approaches that target whole-value sparsity.
- Table 1 summarizes the relative energy efficiency of *Laconic* and various state-of-the-art accelerators [3, 13, 15, 71, 83] under iso-area constraints in 65nm. The table also reports energy efficiency for an 8b *Laconic* configuration at 15nm.
- Compared to a bit-parallel design, the area and energy efficiency of *Laconic* with 8b processing elements improves, albeit slightly, with a 15nm process technology.
- As the data width shrinks so does the area advantage of *Laconic*'s processing elements compared to bit-parallel ones. In 65nm, the 16b *Laconic* processing element is more than 7.3× smaller. The 4b *Laconic* is instead 3.5× smaller than a 4b bit-parallel one.



Policy	Representative Accelerators	Property Exploited	Ineffectual Multiplications Removed
A	Cnvlutin [4], ZeNa [51], EIE [37]	Activation Sparsity	Zero Activation
A+W	SCNN [71]	Activation + Weight Sparsity	Zero Activation or Weight
At	Pragmatic [3]	Activation Bit Sparsity	Zero Activation Terms (Booth Encoded)
Wt	Pragmatic* [3]	Weight Bit Sparsity	Zero Weight Terms (Booth Encoded)
At+W	Tactical [24]	Weight Sparsity + Activation Bit Sparsity	Zero Weight or Zero Activation Terms (Booth Encoded)
At+Wt	Laconic (this work)	Activation + Weight Bit Sparsity	Zero Activation or Weight Terms (Booth Encoded)

Figure 1: Performance improvement potential (logarithmic scale) with various ineffectual multiplication removal policies.

Table 2: Neural Networks Studied. (†) Pruned.

Model	Dataset	Domain
	8b Quantized	
Alexnet-S† [97]	ImageNet12 [80]	Classification
BiLSTM [89]	Flickr8k [77]	Captioning
GoogleNet-S† [97]	ImageNet12 [80]	Classification
Inception-v3 [87]	ImageNet12 [80]	Classification
MobileNet-v2 [81]	ImageNet12 [80]	Classification
Segnet [6]	CamVid [8]	Segmentation
	16b Quantized	
JointNet [32]	CBSD68 [66]	Demosaicking/Denoising
ResNet50-S† [75]	ImageNet12 [80]	Classification
VDSR [59]	CBSD68 [66]	Super-Resolution
YOLO-v2 [79]	Pascal VOC [28]	Object Recognition
	16b A/4b Log W Quantized	
ResNet18-INQ [103]	ImageNet12 [80]	Classification
	2b A/1b W Quantized	
ResNet18 [40]	ImageNet12 [80]	Classification

2 BIT SPARSITY IS ABUNDANT

This section shows the work reduction potential when targeting whole-value or bit-level sparsity. Table 2 details the studied networks. They include quantized, dense, and pruned models covering a variety of applications. The bulk of the work performed in neural networks is due to convolutional and fully-connected layers. For completeness, we review their operation.

Convolutional Layers: A convolutional layer takes an input feature map *imap*, which is a $C \times H \times W$ (channels, height, and width) 3D array of activations, applies K $C \times H_F \times W_F$ 3D filter maps, or *fmaps*, repeatedly using a stride S and produces an output map, or *omap*, which is a $K \times H_O \times W_O$ 3D array of activations. Each output activation is the inner product of a filter and a *window*, a sub-array of the imap of the same size as the filter. If a, o and w^n are respectively the imap, the omap and the n-*th* filter, the output activation $o(n, y, x)$ is computed as the following inner product $\langle w^n, \cdot \rangle$:

$$o(n, y, x) = \sum_{k=0}^{C-1} \sum_{j=0}^{H_F-1} \sum_{i=0}^{W_F-1} w^n(k, j, i) \times a(k,\ j + y \times S,\ i + x \times S) \quad (1)$$

Input windows form a grid with a stride S and the omap dimensions are respectively K, $H_O = (H - H_F)/S + 1$, $W_O = (W - W_F)/S + 1$. In the discussion that follows, we assume without loss of generality that $S = 1$. However, the concepts apply regardless.

Fully-connected layers can be thought of as a special case of convolutional layers where there is a single window. The proportion of time spent in convolutional vs. fully-connected layers varies per network. Typical image classification models use a few, relatively small fully-connected layers at the end of the network. Presently, most natural language processing models use predominantly fully-connected layers [20], and others use a mix of both fully-connected and convolutional layers [21, 29]. Section 3.7 further discusses how *Laconic* interacts with non-convolutional layers.

Figure 1 reports the ideal reduction in multiplication work expressed as potential speedup. A speedup of 10× means that 90% of the work performed is ineffectual. Policies "A" and "A+W" target whole-value sparsity. "A" avoids multiplications where the activation is zero. This is representative of the first generation of value-based accelerators that were motivated by the relatively large fraction of zero activations that occur in convolutional neural networks, e.g., [4, 37, 51], The "A+W" policy skips those multiplications where either the activation or the weight are zero and is representative of accelerators that target sparse models where a significant fraction of synaptic connections have been pruned [50, 71]. "At" and "Wt" exploit bit sparsity in the activations or weights respectively [3]. Pragmatic implemented "At", however, its tiles are symmetric and could implement "Wt". The measurements corroborate past work on accelerator designs that exploited the respective properties, and affirm that the underlying properties persist across all models, albeit to varying degrees.

"At+Wt" shows greater work reduction potential than all other policies as it decomposes multiplications at the bit level, e.g., for 8b:

$$A \times W = \sum_{i=0}^{7} \sum_{j=0}^{7} A_i \text{ AND } W_j \quad (2)$$

where A_i and W_j are bits of A and W respectively. When decomposed down to 64 $1b \times 1b$ operations, it is only those multiplications where both A_i and W_j are non-zero that are *effectual*. Even more ineffectual work can be exposed by Booth-encoding A and W, thus representing them as a series of signed powers of two, or *terms*:

$$A \times W = \sum_{i=0}^{A_{terms}} \sum_{j=0}^{W_{terms}} At_i \times Wt_j \quad (3)$$

where At_i and Wt_i are of the form $\pm 2^x$. With this representation it is only those products where both At_i and Wt_i are non-zero that are effectual. The potential speedup is nearly two orders of magnitude higher than the whole value sparsity "A" and "A+W" and often one order of magnitude higher than exploiting bit sparsity only for activations ("At"). Tactical implements a "W+At" policy which avoids ineffectual work due to whole-value sparsity for weights and bit sparsity for activations [24]. It uses a datatype agnostic front-end for skipping zero weights with Pragmatic's bit-skipping ability. The potential with "At+W" is significantly lower than with "At+Wt" and typically better than "At".

3 LACONIC

We illustrate the key concepts behind *Laconic* via the example of Figure 2 where we are to process two input feature map (imap) windows over two filters (fmaps). For each window we are to multiply and accumulate two activation and weight pairs:

Window 0 / Filter 0	$psum_0$ += $(A_0 \times W_0 + A_1 \times W_1)$
Window 0 / Filter 1	$psum_1$ += $(A_0 \times W_0' + A_1 \times W_1')$
Window 1 / Filter 0	$psum_3$ += $(A_0' \times W_0 + A_1' \times W_1)$
Window 1 / Filter 1	$psum_4$ += $(A_0' \times W_0' + A_1' \times W_1')$

The example assumes 8b weights and activations.

Bit-Parallel Processing: Figure 2a shows a bit-parallel processing element (PE) which performs two $8b \times 8b$ Multiply-Accumulate (MAC) operations per cycle. It reads the activations and weights from scratchpads (spads) and accumulates the partial sums (psum) into another spad. Such adder-tree PEs amortize the cost of reading-modifying-writing the psum spad over multiple products. For this reason they are more energy efficient than single MAC PEs which read-modify-write the psum spad once per product. Multiple PEs and their spads can be organized in numerous ways to best exploit data reuse temporally and spatially. This PE needs four cycles to perform our example calculations.

Term-Serial Processing/Laconic: Figure 2b shows how a *Laconic* PE (LPE) processes window 0 and filter 0. *Laconic* reads A_0, A_1, W_0 and W_1 from their scratchpads (not shown for clarity) and then Booth-encodes them into lists of signed powers of two or *terms*. For example, it encodes A_1 = (0011 1100) as $(+2^6, -2^2)$. The *Laconic* PE needs to "multiply" each activation term with every corresponding weight term. Each cycle the *Laconic* PE can multiply and accumulate two pairs of terms. It takes two cycles to process the example values and thus the LPE is 2× slower than the bit-parallel PE. However, as we will show, the LPE is considerably smaller than the bit-parallel PE. Accordingly, for the same compute area *Laconic* can use more LPEs to boost performance.

Figure 2c shows how a 2×2 grid of Laconic PEs can process all eight activation and weight pairs in 2 cycles, a speedup of 2× over the bit-parallel PE. Naturally, the grid requires twice the input and output bandwidth compared to the bit-parallel PE. However, since each activation term needs to be "multiplied" with every weight term, on average activations and weights will not need to be read every cycle. Further, since the weights and the activations are reused spatially over two LPEs each, the cost of each Booth encoder is also amortized over two LPEs. Accordingly, the larger the grid the lower the relative cost of the Booth encoders.

Laconic will be faster than an equivalent bit-parallel PE based design only if it can afford more LPEs. It is thus essential for the LPE to be smaller and more energy efficient than the bit-parallel PE. In the worst case, an 8b value translates into 5 terms when Booth encoded. This means that *Laconic* would need 25 LPEs per bit-parallel PE to be always at least as fast for *all* possible input value combinations. However, such a design is grossly over-provisioned to handle the case where *all* values in *all* layers need 5 terms. In practice *Laconic* will be faster overall as long as it faster for enough of the values. As Section 2 suggests and as Section 4 will show, the expected number of terms per input value is much lower and thus *Laconic* needs only a few LPEs per PE.

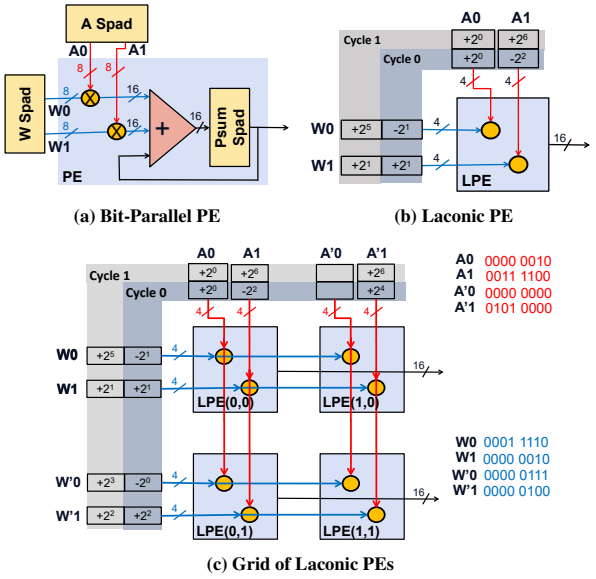
The rest of this section presents the *Laconic* design emphasizing how it utilizes area and energy more efficiently compared to conventional bit-parallel designs. Section 3.1 discusses how values are stored, communicated so that on-chip and off-chip traffic is unaffected. Section 3.2 describes the *Laconic* PE, the most innovative component of *Laconic*. Subsequent sections discuss how to organize multiple PEs into tiles, how to exploit data reuse and the memory system used. Finally, we present two other innovative aspects of *Laconic*. Section 3.5 explains how *Laconic* can support temporal and/or spatial composition of larger data widths, while Section 3.6 explains an additional dimension upon which the bit-level parallelism that *Laconic* exposes can be exploited.

3.1 Activation and Weight Representation

In general, *Laconic*'s goal is to process only the "essential" activation and weight bits. Encoding and storing these values as lists of terms however is neither space nor energy efficient. Accordingly, *Laconic* stores values using the conventional positional encoding in memory. *Laconic* converts these values into lists of terms *after* reading them from the on-chip memories and just before passing them to the processing elements. Each term is represented as a $(sign, magnitude)$ pair with an extra wire signaling a zero value.

3.2 Laconic Processing Element

Without loss of generality we describe a *Laconic* PE (LPE) that multiplies 16 weights, W_0,...,W_{15}, by 16 input activations, A_0,...,A_{15} per cycle all originally using 8b. We have found that processing 16 weight and activation pairs balances well input spad reuse and psum spad traffic reduction.



(a) Bit-Parallel PE (b) Laconic PE

(c) Grid of Laconic PEs

Figure 2: Laconic processes weights and activations as a series of signed powers of two.

Formally, *Laconic* calculates the product of a weight $W = (W_{terms})$ and an input activation $A = (A_{terms})$ where each term is a $(sign, magnitude) = (s_i, t_i)$ as follows:

$$
\begin{aligned}
W \times A = & \sum_{\forall (s,t) \in W terms} (-1)^s 2^t \times \sum_{\forall (s',t') \in A terms} (-1)^{s'} 2^{t'} \\
= & ((-1)^{s_0} 2^{t_0} + (-1)^{s_1} 2^{t_1} + \cdots + (-1)^{s_n} 2^{t_n}) \times \\
& ((-1)^{s'_0} 2^{t'_0} + (-1)^{s'_1} 2^{t'_1} + \cdots + (-1)^{s'_m} 2^{t'_m}) \\
= & ((-1)^{s_0}(-1)^{s'_0}(2^{t_0} \times 2^{t'_0}) + \cdots + ((-1)^{s_0}(-1)^{s'_m} 2^{t_0} \times 2^{t'_m})) + \\
& \cdots + ((-1)^{s_n}(-1)^{s'_0}(2^{t_n} \times 2^{t'_0}) + \cdots + ((-1)^{s_n}(-1)^{s'_m} 2^{t_n} \times 2^{t'_m})) \\
= & ((-1)^{(s_0+s'_0)} 2^{(t_0+t'_0)} + \cdots + (-1)^{(s_0+s'_m)} 2^{(t_0+t'_m)}) + \\
& \cdots + ((-1)^{(s_n+s'_0)} 2^{(t_n+t'_0)} + \cdots + (-1)^{(s_n+s'_m)} 2^{(t_n+t'_m)})
\end{aligned}
\tag{4}
$$

That is, instead of processing the full $A \times W$ product in a single cycle, *Laconic* processes each product of a single t' term of the input activation A and of a single t term of the weight W individually. Since these terms are powers of two so will be their product. Accordingly, *Laconic* can first add the corresponding exponents $t'+t$. If a single product is processed per cycle, the $2^{t'+t}$ final value can be calculated via a decoder. The same approach can be used when processing 16 weight and activation pairs in parallel. However, the resulting design proved prohibitively expensive. Accordingly, the key innovation in *Laconic* is the design of an area and energy efficient PE which is based on a histogram front-end and an optimized adder tree back-end. We first emphasize the histogram-based front-end and defer the description of the efficient adder tree until Section 3.2.2.

3.2.1 A Histogram-Based PE. Figure 3 illustrates a non-optimized LPE design which calculates the 16 products in 6 steps (the figure omits the handling of the "value is zero" signals):

In **Step 1**, the LPE accepts 16 3-bit weight terms, $t_0,\ldots,t_{15}$ and their 16 corresponding sign bits $s_0,\ldots,s_{15}$, along with 16 3-bit activation terms, $t'_0,\ldots,t'_{15}$ and their signs $s'_0,\ldots,s'_{15}$, and calculates 16 term pair products. Since all terms are powers of two, their products will also be powers of two. Accordingly, to multiply 16 activations by their corresponding weights *Lac* adds their terms to generate the 4-bit exponents $(t_0+t'_0),\ldots,(t_{15}+t'_{15})$ and uses 16 XOR gates to determine the signs of the products.

In **Step 2**, for the i^{th} activation and weight pair, where $i \in \{0,...,15\}$, the LPE calculates $2^{(t_i+t'_i)}$ via a 4b-to-16b decoder which converts the 4-bit exponent result $(t_i+t'_i)$ into its corresponding one-hot format, i.e., a 16-bit number with one "1" bit and 15 "0" bits. The single "1" bit in the j^{th} position of a decoder output corresponds to a value of either $+2^j$ or -2^j depending on the sign of the corresponding product ($E_i.sign$ on the figure).

Step 3: The LPE generates the equivalent of a *histogram* of the decoder output values. Specifically, the PE accumulates the 16 16-bit numbers from Step 2 into 16 buckets, $N^0,\ldots,N^{15}$ corresponding to the values of $2^0, 2^1, ..., 2^{15}$ as there are 16 powers of two. The signs of these numbers $E_i.sign$ from Step 1 are also taken into account. At the end of this step, each "bucket" contains the count of the number of inputs that had the corresponding value. Since each bucket has 16 signed inputs the resulting count would be in a value in [-16, ..., 16] and thus is represented by 6 bits in 2's complement.

Step 4: Naïvely reducing the 16 6-bit counts into the final output would require first "shifting" the counts according to their weight converting all to 15+6 = 21b and then using a 16-input adder tree as shown in Figure 3a(4)-(5). Instead *Laconic* reduces costs and energy by exploiting the relative weighting of each count by grouping and concatenating them in this stage as shown in Figure 3b(4′.1). For example, rather than adding N^0 and N^6 we can simply concatenate them as they are guaranteed to have no overlapping bits that are "1". This is explained in more detail in Section 3.2.2.

Step 5: As Section 3.2.2 details, the concatenated values from Step 4′.1 are added via a 6-input adder tree as shown in Figure 3b(5′) producing a 22b partial sum.

Step 6: The partial sum from the previous step is accumulated into the psum accumulator. This way, the complete $A \times W$ product can be calculated over multiple cycles.

The aforementioned steps are not pipeline stages. They can be merged or split as desired. In fact, the implementation logic up to the psum accumulator is fully *combinatorial*.

3.2.2 Shrinking the Adder Tree. Step 5 of Figure 3a has to add 16 6b counts each corresponding to a different power of 2 between 0 and 15. A more area and energy efficient "adder tree" replaces steps 4 and 5. It takes advantage of the fact that the outputs of Step 3 contain groups of numbers that have no overlapping bits that are "1". For example, in relation to the naïve adder tree of Figure 3a(5) consider adding the 6^{th} 6-bit input ($N^{\mathbf{6}}$=$n_5^{\mathbf{6}}n_4^{\mathbf{6}}n_3^{\mathbf{6}}n_2^{\mathbf{6}}n_1^{\mathbf{6}}n_0^{\mathbf{6}}$) with the 0^{th} 6-bit input ($N^{\mathbf{0}}$=$n_5^{\mathbf{0}}n_4^{\mathbf{0}}n_3^{\mathbf{0}}n_2^{\mathbf{0}}n_1^{\mathbf{0}}n_0^{\mathbf{0}}$). We have to first shift N^6 by 6 bits which amounts to adding 6 zeros as the 6 least significant bits of the result. In this case, there will be no bit position in which both N^6« 6 and N^0 will have a bit that is 1. Accordingly, adding (N^6« 6) and N^0 is equivalent to concatenating either N^6 and N^0 or (N^6-1) and N^0 based on the sign bit of N^0 (Figure 4a):

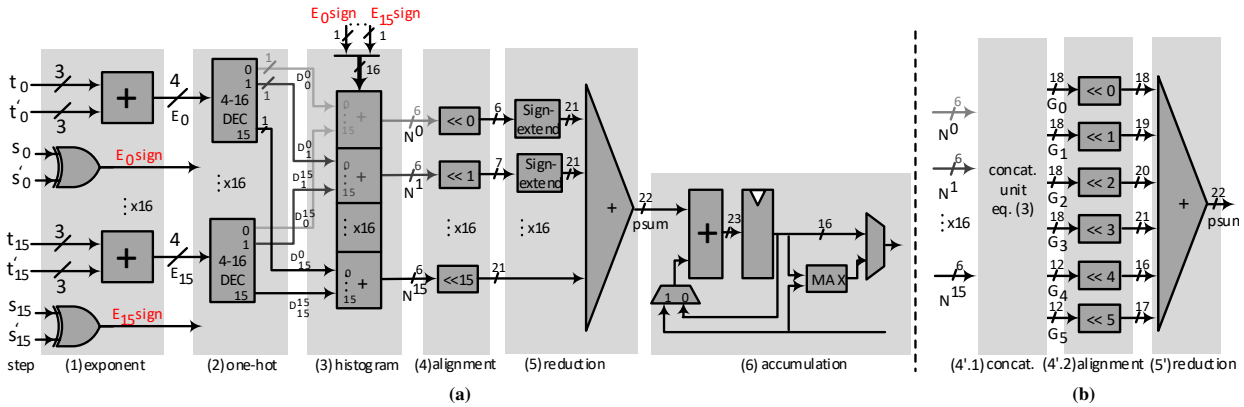


Figure 3: *Laconic* processing element: a) 1) "Multiplying" the terms producing exponents, 2) one-hot encoding of the exponents, 3) Tallying how many times each power of two appears, 4) Shifting the counting results according to the bucket weight, 5) Sign-extending the shifted values and reducing the results into a single 42-bit partial output, 6) Accumulating the partial outputs over multiple cycles. b) Alternate implementation of steps (4) and (5).

$$
\begin{aligned}
N^6 \times 2^6 + N^0 \quad &= \\
\text{1) if } n_5^0 = 0: \quad &= n_5^6 n_4^6 n_3^6 n_2^6 n_1^6 n_0^6 000000 + 000000 n_5^0 n_4^0 n_3^0 n_2^0 n_1^0 n_0^0 \\
&= n_5^6 n_4^6 n_3^6 n_2^6 n_1^6 n_0^6 n_5^0 n_4^0 n_3^0 n_2^0 n_1^0 n_0^0 = \{\mathbf{N^6}, \mathbf{N^0}\} \\
\text{2) if } n_5^0 = 1: \quad &= n_5^6 n_4^6 n_3^6 n_2^6 n_1^6 n_0^6 000000 + 111111 n_5^0 n_4^0 n_3^0 n_2^0 n_1^0 n_0^0 \\
&= (n_5^6+1)(n_4^6+1)(n_3^6+1)(n_2^6+1)\cdots \\
&\cdots (n_1^6+1)(n_0^6+1) n_5^0 n_4^0 n_3^0 n_2^0 n_1^0 n_0^0 = \{(\mathbf{N^6}\text{-}1), \mathbf{N^0}\}
\end{aligned}
\tag{5}
$$

This process can be applied recursively by grouping those N^i where (i MOD 6) is equal. Figure 4b shows an example for those N^i inputs where (i MOD 6) = 0. While the figure shows the concatenation done in a stack other arrangements are possible and the process can be pipelined. For the 16 product unit there are six groups:

$$
\begin{aligned}
&G_0 = \{N^{12}, N^6, N^0\},\ G_1 = \{N^{13}, N^7, N^1\},\ G_2 = \{N^{14}, N^8, N^2\}, \\
&G_3 = \{N^{15}, N^9, N^3\},\ G_4 = \{N^{10}, N^4\},\ G_5 = \{N^{11}, N^5\}
\end{aligned}
\tag{6}
$$

and the partial sum becomes: $psum = \sum_{i=0}^{5} (G_i \ll i)$

Since the shift amounts are constant there is no need for shifters in the implementation.

3.2.3 A Lower Area Histogram. The histogram buckets in Figure 3 accept 16 2b inputs each encoding $(+1, -1, 0)$. Every bucket connects to all decoders and at its corresponding power of 2 output. We can reduce area by using buckets with only 8 such inputs, To do so, we first "merge" the decoder outputs in pairs. Since the decoder outputs are in one-hot format, there will have be at most a single overlapping "1" bit. When there is no such overlap, we can pass the signs of the corresponding decoders. When there is overlap, the corresponding *Esign* bits of the decoders dictate the action necessary. If the sign bits are opposite the "1" bits simply cancel each other, otherwise the next bit should be set to "1".

3.3 Tile Organization

Thus far we have described a *Laconic* PE that can process 16 (A, W) pairs per cycle, all contributing to the same output. How many (A, W) pairs the LPE processes is a design time choice. Several LPEs can be grouped together in multiple ways to form a tile, and so on. Here we present one such *Laconic* tile organization. Figure 4b shows an example configuration where 32 LPEs are organized as an 8×4 grid. Two input scratchpads, *ASpad* and *WSpad* respectively provide the activation and weight inputs (or vice versa depending on the layer), and a third scratchpad *SPad* is used to store partial or complete output neurons. A bus per row of LPEs connects them to the PSpad. Psums are read out of the LPE grid and accumulated or written to the PSpad one column at a time. There is enough time to drain the LPE array via the common bus since: a) processing of even a single group of (A, W) inputs takes multiple cycles, and b) we can usually process multiple (A, W) groups prior having to read out the psums. A set of *encoders* between the input spads and the LPE grid converts values into a series of terms. An optional *composer* column, detailed in Section 3.5, provides support for the spatial composition of 16b arithmetic while maintaining 8b LPEs. This design enables reuse of activations and weights in *space and time* and minimizes the number of connections/wires needed to supply it with activations and weights from the rest of the memory hierarchy.

In more detail, processing using the tile of Figure 4b can proceed with 4 windows of activations and $K = 8$ filters per cycle. In this case the WSpad provides 16 weights per filter and the ASpad provides the 16 corresponding activations per window. LPEs along the same column share the same input activations and LPEs along the same row receive the same weights. The encoders convert these values into terms at the maximum rate of one term per cycle.

Every cycle LPE(i,j) receives the next term from each input activation from the j^{th} window and multiplies it by a term of the corresponding weight from the i^{th} filter. Once all activation terms have been multiplied with the current weight term, the next weight

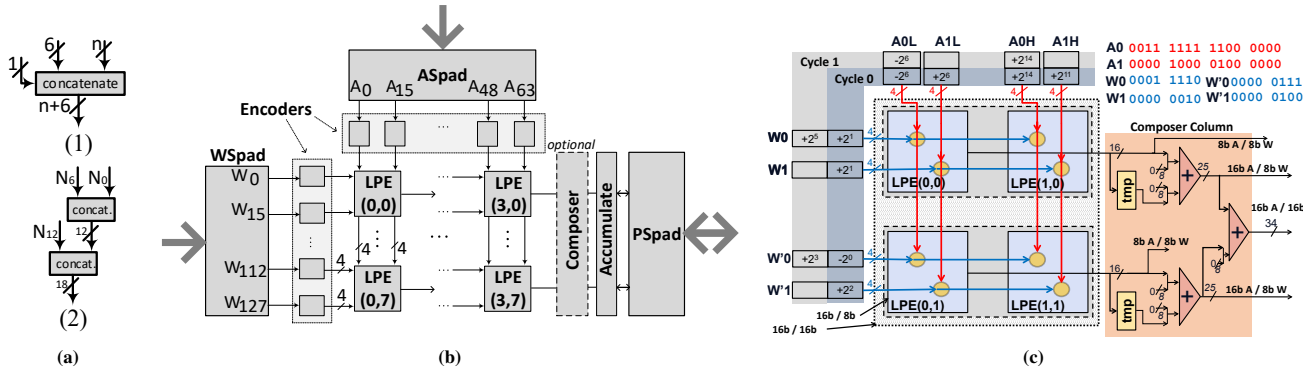


Figure 4: (a) One of *Laconic*'s concatenation units. (b) A *Laconic* tile with 8×4 LPEs. (c) Optional spatial composition: supporting 16b activations and/or weights with 8b LPEs.

term is produced. The unit cycles through all the corresponding activation terms again to multiply them with the new weight term. The product is complete once all weight and activations terms have been processed. For example, if there are 3 activation terms and 4 weight terms, at least 12 cycles will be needed. In total the tile processes 4 windows, 16 activations/window, and 8 filters, i.e., $4 \times 16 \times 8$ activation and weight pairs concurrently.

3.4 Tile Synchronization

In general the number of terms will vary across weight and activation values and as a result some LPEs will need more cycles than others to process their product. In the simplest organization, the tile implicitly treats all concurrently processed (A, W) pairs as a group and synchronizes processing across different groups. That is, the tile starts processing the next group when all the LPEs are finished with processing all the terms of the current group. A disadvantage of this approach is that it gives up some of the speedup potential. We will refer to this synchronization scheme as *tile* synchronization.

A better performing approach allows computation to proceed in 16 independent groups. For our example tile, the first synchronization group contains $A_0, A_{16}, A_{24}, ..., A_{48}$ and weights $W_0, W_{16}, ..., W_{112}$. The second group contains $A_1, A_{17}, A_{25}, ..., A_{49}$ and weights $W_1, W_{17}, ..., W_{113}$, and so on for the remaining 14 groups. We will refer to this synchronization scheme as *comb* synchronization since the groups physically form a comb-like pattern over the grid. A set of buffers at the inputs of the Booth-Encoders can be used to allow the groups to slide ahead of one another.

3.5 Data Type Composability

Some networks, e.g., YOLO, require 16b precisions only for some layers. Others do so only for the activations [83], whereas only few values typically require more than 8b [22, 23, 73, 74]. Thus far, our *Laconic* designs use LPEs that natively support the worst case data width required across *all* layers and *all* values. This section describes *Laconic* designs that support data type composition in space and/or time. Without loss of generality we present designs that enable 16b calculations over 8b LPEs for activations and optionally weights.

Spatial composition uses multiple yet unmodified 8b LPEs. Temporal composition alters the LPE to support both 8b and 16b operation albeit at a lower area cost than a native 16b LPE. Temporal composition requires extra cycles whenever it has to process 16b values. Spatial composition requires extra LPEs whenever it processes a layer that has 16b values.

Both forms of composition are useful for networks that require more than 8b for some of their layers only. Since temporal composition does not reserve resources for whole layers and given that values that require more than 8b will be relatively few [23, 73], it will achieve higher throughput per LPE than spatial composition. However, temporal composition requires larger LPEs and more sophisticated control at the Booth encoders. In the interest of space, we focus on the spatial design and report only the area overhead of the temporal: an 8b LPE capable of temporal extension to 16b is 22% smaller compared to a native 16b LPE (see Section 4 for the methodology). A designer can choose which approach is best for their target use and even combine the two (for example supporting spatial composition for weights and temporal for activations).

To support 16b/8b computation the activation terms are split into those corresponding to their lower and upper bytes which are processed respectively by two adjacent LPE columns. In the example, LPE(0,0) and LPE(0,1) process the lower bytes of A_0 and A_1 and LPE(1,0) and LPE(1,1) process the upper bytes. Rows 0 and 1 respectively process filters 0 and 1 as before. Processing proceeds until the psum registers have accumulated the sums of the upper and lower bytes for the data block. At the end, the psums are drained one column at time as done with the *Laconic* designs thus far. A "Composer" column is added at the grid's output. When the psums of column 0 are read out, they are captured in the respective temporary registers (tmp). Next cycle, the psums of Column 1 appear on the output bus. The per row adders of the Composer add the two halves and produce the final psum. While the example shows a 2×2 grid, the concept applies without modification to larger LPE grids. A single "Composer" column is sufficient as the grid uses a common bus per row to output the psums one column at a time. While not shown, the "Composer" can reuse the "Accumulate" column adders instead of introducing a new set. In general supporting 16b activations would require two adjacent per row LPEs.

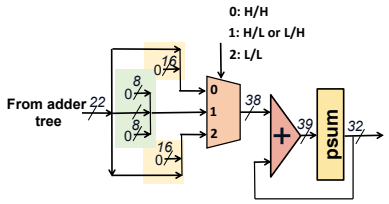


Figure 5: Temporal Composition: Supporting 8b and 16b activations and weights within the *Laconic* PE.

The design also supports 16b weights by splitting them along two rows. This requires four LPEs each assigned to one of the four combinations of lower and upper bytes. In the figure, LPE(0,0), LPE(1,0), LPE(0,1) and LPE(1,1) will respectively calculate (A_L, W_L), (A_H, W_L),(A_L, W_H), and (A_H, W_H). The second level adder in the "composer" column takes care of combining the results from the rows by zero-padding row 1 appropriately. Section 4.6 evaluates this design.

This approach to type composition is not specific to *Laconic*. It works with grids of other processing elements, such as for example those used in Stripes [23, 48] or even the bit-parallel PE of Figure 2a.

Temporal Composition: Similarly, we can split the terms of activations and weights into those belonging to the upper and the lower bytes and process them separately in time. Figure 5 shows the modified *Laconic* PE. The output from the front-stage adder is appropriately padded with zeros and then added to the extended precision psum. There are three cases based on the source of the activation and weight terms being processed: both belong to lower bytes (L/L), both belong to upper bytes (H/H), or one belongs to an upper byte and the other to a lower one (H/L or L/H). The multiplexer has to select the appropriately padded value. The multiplexer's select signal can be shared among all LPEs in a tile. Processing 8b values incurs no overhead. Processing 16b activations and 8b weights (or vice versa) requires an extra cycle, whereas processing 16b weights and activations requires 3 extra cycles. However, this time overhead has to be paid only when there is a value that really needs 16b.

Temporal Data Type Extension: In some networks and for some layers (especially the first layer), the data type needed as determined by profiling sometimes exceeds 8b albeit only slightly, e.g., 9b or 10b are needed. Executing these layers is possible with the unmodified 8b LPE and with a minor modification to the Booth-Encoder. For example, for a value whose 9th bit is 1 the Booth-Encoder can effectively synthesize $+2^8$ by sending $+2^7$ twice. As an added benefit, this flexibility makes quantization easier.

3.6 Spatiotemporal Term Processing

Laconic thus far exploits *inter-value* bit-level parallelism, however, it can be extended to also exploit *intra-value* bit-level parallelism and to do so differently than bit-parallel hardware. This is possible, since the LPE produces the correct result even if we process the terms of a value *spatially* instead of *temporally*. For example, we can choose to assign two input lanes per weight and modify the Booth-encoder so that it outputs up to two terms per cycle. This also enables *Laconic* to exploit bit-level parallelism within values, a capability that can be useful in the following ways: 1) to reduce synchronization overheads, 2) to improve utilization for layers where there isn't enough reuse of weights and/or activations to fill in all columns and/or rows respectively. This is the case for example for fully connected layers where there is no reuse of weights. It is also useful for depth-separable convolutional layers. 3) When there are not enough filters to fill in all rows. This is different than the intra-value bit-level parallelism exploited by conventional bit-parallel units: they process all bits regardless of value whereas *Laconic* would process only the effectual ones. We leave the investigation of this optimization for future work.

3.7 Layer Support

Since fully-connected layers do not have weight reuse and their filters are by comparison large they can utilize only one LPE column per tile. Fully-connected layers are typically off-chip memory bound. Thus reducing off-chip traffic is paramount for these layers. As Section 3.8 explains, we use zero compression and per group precision to reduce off-chip bandwidth for all layers [22, 23]. Other compression schemes may be more effective for these layers [38]. In the image classification networks, fully-connected layers account for less than 10% of the overall execution time. Other networks such as VDSR have no fully-connected layers. In Multilayer Perceptrons, and many recursive neural networks, fully-connected layers account for a significant portion of execution time. For some of those models, softmax operations are also taxing.

Long-Short Term Memory models have element-wise multiplications which utilize 1/16 of the PE. Moreover, some recent models use layers with less or no intra-filter parallelism such as depth-separable convolutions. However, generally these layers account for a small fraction of overall execution time. As with fully-connected layers, exploiting intra-value bit-level parallelism may increase utilization. Using a separate vector unit may be more appropriate.

3.8 Memory System

For all designs, we introduce separate second level memories for the activations and the weights which were sized as follows: for most of the models, weight storage and traffic dominate that of activations. Thus, we opt for an on-chip memory system that has: 1) an Activation Memory (AM) that can hold the input and output Activations of any *one* layer at a time, and 2) a Weight Memory (WM) that fits a set of filters that need to be processed concurrently for all but the fully-connected layers. The latter tend to be off-chip memory bound and increasing the WM would not help.

However, the computational imaging models are fully convolutional and there the activations dominate. For those models it is best to use: 1) an AM that can fit a full row of input and output windows of any *one* layer at a time, and 2) a WM that can fit all the filters for the one layer [85]. 2MB of AM and 512KB of WM were sufficient. Here we use AM (WM) exclusively for activations (weights). However, depending on the layer dimensions further improvement in energy efficiency and performance may be possible by switching feeding activations into WM and vice versa.

To reduce off-chip traffic and footprint we use *DPRed* lossless compression [22, 23]. DPRed relies on the expected distribution of values in neural networks: most values tend to be small and few values tend to be of high magnitude. Thus, encoding values using a model-wide or even a layer-wide data width over-provisions for

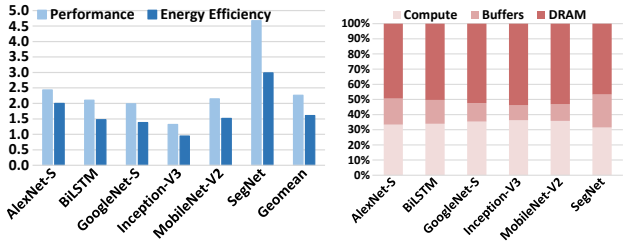
the worst case. DPRed encodes values in groups using only as many bits as necessary to represent the highest-magnitude value within the group. A header identifies the datawidth used for the group. As a result, most groups end up using a lot fewer bits than what would be needed if all values were stored with the maximum precision needed. This results in *fractional* effective precisions per layer since precision varies per group. General purpose applications with arbitrary off-chip access streams such a variable-length encoding scheme would present a challenge. However, access in neural network workloads are typically blocked and thus off-chip accesses tend to be sequential and for large, continuous chunks of memory. The off-chip access unit only needs to know where these chunks are located. In our case, since we access activations and weights only once per layer, only the beginning and size of these arrays are needed.

4 EVALUATION

This section evaluates *Laconic*'s performance, energy and area comparing with several state-of-the-art accelerators: *DaDianNao++* an optimized bit-parallel design using DaDianNao-like PEs [13], *SCNN* [71], *Eyeriss* [15], *Pragmatic* [3], and *BitFusion* [83]. Except for the comparison with SCNN where we study the convolutional layers only, we run all layers. A custom cycle-accurate simulator models execution time for standalone accelerator subsystems (incorporating our models to simulators for full systems is left for future work [58]). Energy and area results are reported based on post layout simulations of the designs. Synopsys Design Compiler [86] was used to synthesize the designs with a TSMC 65nm library. Layouts were produced with Cadence Innovus [9] using synthesis results. Intel PSG ModelSim is used to generate data-driven activity factors and to estimate power. The clock frequency of all designs is set to 1GHz. The scratchpad SRAM buffers and the activation and weight memories were modeled with CACTI [70].

4.1 Comparison with DaDianNao++

We model *DaDianNao++* a bit-parallel accelerator that uses DaDianNao-like PEs [13]. *DaDianNao++* uses spads to exploit reuse in time in addition to exploiting reuse spatially and the same AM and WM as *Laconic*. We configure *Laconic* and *DaDianNao++* with 8b PEs and for the same tile area. A *DaDianNao++* PE multiplies 16 8b activation/weight pairs per PE. Each tile has ten PEs. Based on the post layout results the tile area for *DaDianNao++* is $0.73mm^2$. We configure *Laconic* to use a grid of 16×9 LPEs in the same tile area as follows: The original DaDianNao used input and output activation buffers entries, which afforded it some flexibility in dataflow choices. One such choice is using the output buffer to cycle over different output activations while keeping the weight values constant. As highlighted in Stripes [48], such a choice of dataflow (referred to as "WT - window tiling" - which we use here to allow *DaDianNao++* to cycle through 32 values) can improve energy efficiency by 2.5×. Accordingly, activation and output scratchpads are configured with 64 entries each of 16 and 10 values respectively. *Laconic*'s layout has inherent weight reuse due to its 2D layout, where multiple columns share weights (on top of multiple rows sharing activations, as in DaDianNao). This reduces the pressure on internal buffers. Accordingly, the dataflow we employed for *Laconic* does not use window tiling and this enables *Laconic* to use



(a) *DaDianNao++*/8b Models **(b) *Laconic* Energy Breakdown /8b Models**

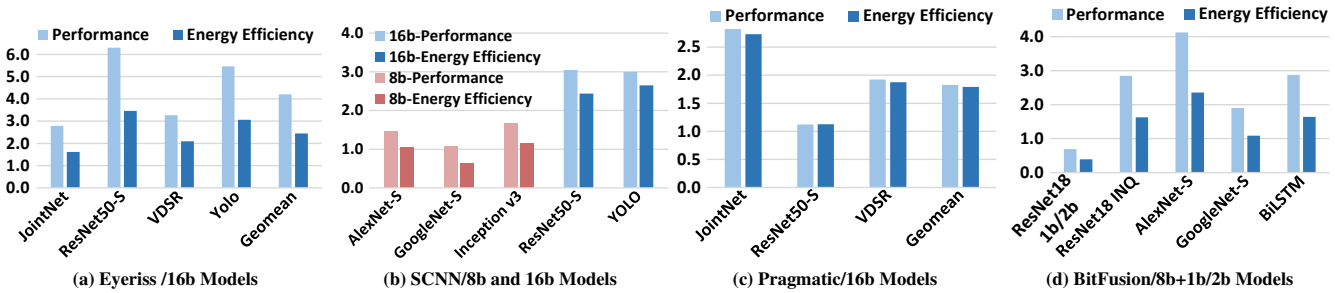
Figure 6: Comparing 8b *Laconic* and 8b *DaDianNao++* for 8b quantized networks.

less on-tile buffering. Since *Laconic*'s LPEs are much smaller than DaDianNao's PEs, reducing the amount of SRAM buffers leads to a correspondingly larger increase in their number, whereas the same trade-off is not as beneficial for DaDianNao. Table 5 reports the area breakdown for the tiles. Recall each *Laconic* LPE contains two 16b output registers. Regardless, while here we limit attention to these two configurations, exploring different dataflows with their buffering requirements and their impact in efficiency and performance is essential future work [95].

Figure 6a shows the performance of *Laconic* relative to *DaDianNao++*. On average for the studied networks *Laconic* outperforms *DaDianNao++* by 2.3× while being 1.6× more energy efficient. SegNet benefits the most from *Laconic*. As a result of the quantization method for this network, the value range for both weights and activations in most layers fits in 5b or less. This translates into high bit sparsity which *Laconic* exploits. Inception-V3 benefits the least from *Laconic* performance improves by 1.3× but energy efficiency is 5% less for *Laconic*. SegNet benefits the most despite being the model with the less potential. The variance of term sparsity across values in SegNet is lower by comparison. As a result, there is potential lost due to "outliers". Figure 6b shows a breakdown of where energy is expended in *Laconic*. As expected most of the energy is due to external memory accesses. Using off-chip memory compression and enough on-chip buffering to avoid having to access each value more than once in most layers, effectively reduces the burden of off-chip access balancing on-chip and off-chip energy. Energy for on-chip buffer is relatively higher for the sparse models and SegNet. In general the higher the bit-sparsity the less computation *Laconic* has to perform per value read.

4.2 Comparison with Eyeriss

This section compares *Laconic* with Eyeriss [14, 15] and for 16b models. We use 16b models since for VDSR, ResNet50-S [75], and JointNet there were no 8b quantized versions that are available (ResNet50-S is pruned). Moreover, VDSR and JointNet are image tranformation models and naturally operate on raw imaging sensor pixel values which are typically 10b or longer. Eyeriss [15] as originally proposed is a 16b accelerator that is optimized for data reuse and that avoids computations with zero values. To compare with Eyeriss we modified *Laconic* to support 16b MACs and scaled it so that it uses the same area as the original Eyeriss design. The core area of Eyeriss in 65nm is $12.9mm^2$, from which $4.1mm^2$ is



(a) Eyeriss /16b Models (b) SCNN/8b and 16b Models (c) Pragmatic/16b Models (d) BitFusion/8b+1b/2b Models

Figure 7: *Laconic* performance and energy efficiency relative to Eyeriss, *SCNN*, Pragmatic, and BitFusion

dedicated to the PE array, excluding the spads. The configuration of 16b *Laconic* which uses slightly less compute area than Eyeriss, has a grid of 32×16 LPEs processing 32 filters and 16 activation windows concurrently. In addition, we configure Eyeriss and *Laconic* to use the same total on-chip memory. We measure the energy consumption and performance of Eyeriss using the simulator of Gao et al. [31]. We have assumed that Eyeriss can operate at the same clock frequency of 1GHz as *Laconic*. This is optimistic for Eyeriss as the pipelining that will be needed in practice will incur area overheads.

Figure 7a reports the measured performance and energy efficiency for *Laconic* relative to Eyeriss. On average *Laconic* is 4.2× faster and 2.4× more energy efficient that Eyeriss. ResNet50-S benefits the most from *Laconic*. It is a pruned network and *Laconic* takes advantage of its elevated value sparsity. Despite not using ReLU and thus not clipping negative activations to zero, YOLO also benefits considerably from *Laconic*. JointNet benefits the least but still considerably at nearly 2.8×. JointNet similar to the other computational imaging models performs per-pixel prediction. For this reason, activation values resemble the input pixel-values throughout the layers. This is unlike the image classification models where activation values in deeper layers have little correlation with input image pixels. Despite exhibiting activation values which closely resemble the input pixel values, there is significant bit sparsity, more so given the use of Booth encoding. While here we compared to the original 16b Eyeriss design, Eyeriss can be configured to use 8b units or even *Laconic* processing elements. In the latter case, it would be interesting build upon the scheduling methodology of Eyeriss for such PEs by constraining values in groups of 16 (for the specific LPE configuration) to co-locate. Moreover, adding a data type composer row or column to Eyeriss should allow it to benefit from *Laconic*'s spatial approach to type composition.

4.3 Comparison with SCNN

This section compares *Laconic* with *SCNN*, a state-of-the-art sparse accelerator. Since SCNN's performance excels on sparse models, we include three pruned networks. Alexnet-S and Googlenet-S are 8b quantized networks and Resnet50-S is a 16b network. We do include Inception-V3 and YOLO which are not pruned to illustrate that *Laconic* can deliver benefits even if the network is not pruned. YOLO in particular does not use ReLU and thus may exhibit less activation sparsity that other models. For the 8b networks we modified *SCNN* to use 8b MACs and memories whereas for the 16b model we use a 16b SCNN as originally proposed. In both cases, we scale SCNN to the same post-layout area as the corresponding *Laconic* design. The 16b and 8b *Laconic* configurations use respectively a grid of 32×16 and 32×6 LPEs. The 8b SCNN was configured as a grid of 5×6 PEs, each multiplying four 8b activations by two 8b weights, processing a total of $5 \times 6 \times 4 \times 2$ MACs per cycle. For the 16b networks we have configured 16b *SCNN* to have 72 PEs, each multiplying two 16b weights by four 16b activations matching the same compute area as in 16b *Laconic*. While the original SCNN used tiles having 4×4 multipliers, we found that using the slightly different per tile configurations described above performed equally well.

Figure 7b reports *Laconic*'s performance relative to the corresponding *SCNN* configuration. We limit attention to the convolutional layers only since SCNN's performance suffers considerably for the fully-connected layers. We first discuss the three pruned models. The two accelerators perform similarly well for Googlenet-S. Googlenet-S is pruned with an energy-aware approach and the distribution of zero weights matches well the hardware. This method favors the pruning of nearby synapses. *Laconic* performs better for Alexnet-S which was also pruned with the same energy-aware method. Finally, *Laconic* excels at Resnet50-S which is pruned with a different method than the other models. The results suggest that SCNN is more sensitive to the pruning method than *Laconic*. Moreover, *Laconic* can effectively exploit whole-value sparsity and on top of that, delivering additional benefits by also exploiting bit-sparsity. It does so without explicitly targeting pruned models. As expected *Laconic* ourperforms SCNN for the dense models. While not explicitly pruned they do exhibit sparsity in the activations and weights which SCNN exploits. However, *Laconic* also exploits bit sparsity for the non-zero values.

4.4 Comparison with Pragmatic

We compare with a *Pragmatic* configuration with the same compute area as 16b *Laconic*. This configuration has 10×16 Serial Inner Product units. In the interest of space we restrict attention to a subset of the 16b models. Figure 7c reports the relative speedup and energy-efficiency of *Laconic* over Pragmatic showing that *Laconic* outperforms Pragmatic for all networks studied. The benefits with *Laconic* for Resnet50-S are modest. Since this is a pruned model there is considerably more bit sparsity in the activations (due to the zero values) compared to other models. Pragmatic exploits activation bit sparsity well too. *Laconic*'s advantage is that it can also exploit

bit sparsity in the weights. However, it is the weight with the most terms within each synchronization group that dictates progress. The distribution of weight values is such that sychronization stalls are more pronounced for this model.

4.5 Comparison with BitFusion

BitFusion exploits per layer data precisions. It supports spatial composition of 2b, 4b, and 8b operations, and temporal composition for 16b operations. To compare *Laconic* to BitFusion [83] we have scaled the area and energy consumption of *Laconic* to 45nm using the methodology of Weste et al. [94]. The compute area for *Laconic*, excluding the activation buffers, is $0.46mm^2$ after scaling down to 45nm. The equivalent configuration of BitFusion which uses the same compute area has 214 Fusion Units, each with 16 BitBricks. Using the constant voltage scaling method of Weste et al. [94] *Laconic* consumes 1.68W in 45nm. To model its execution time, we assumed the best case for BitFusion where all Fusion units always operate at their peak computational bandwidth given the precision of each layer and without any synchronization overhead. Figure 7d reports *Laconic* performance and energy efficiency relative to BitFusion for 8b models, the ResNet18-INQ model, and the one 1b/2b ReseNet18 model. The incremental network quantization method (INQ) constrains weights to be either powers of two or zero [103]. The ResNet18-INQ network uses 8b weights and 16b activations while maintaining accuracy. To process the 16b activations *Laconic* uses spatial composition whereas BitFusion temporal composition. For ResNet18-INQ *Laconic* outperforms BitFusion by 2.8× as it exploits the term-sparsity of the logarithmic weights which have at most one term. For the 8b models *Laconic* outperforms BitFusion as it exploits sparsity. For the 1b/2b model as expected BitFusion is about 2× faster and more energy efficient illustrating that if extreme quantization becomes possible, other designs such as BitFusion will be preferable. However, the relative TOP-1 accuracy loss with this model is more than 20% compared to the original ResNet18.

4.6 Composable 8-bit Laconic

We compare the performance of using spatially composable 8b LPEs (Section 3.5) vs. native 16b LPEs. The composable LPEs natively support 8b and/or 16b weights and activations via spatial composition. We have scaled up the 8b composable *Laconic* to use the same area as the 16b *Laconic*, and report performance relative to the native 16b *Laconic* for GoogleNet-S and Resnet50-S, two models for which we profiled to determine per layer precisions; the composable *Laconic* uses multiple LPEs as needed only for those layer that require more than 8b. As expected compared to 16b *Laconic*, 8b composable *Laconic* is 1.4× and 1.2× faster for GoogleNet-S and Resnet50-S respectively.

4.7 Sensitivity Studies

Tile LPE Configuration: We study how performance scales as the number of LPE rows and columns varies. Figure 8 reports speedup with various scaled-up *Laconic* configurations relative the *Laconic* configuration with 32 rows (R) and 6 columns (C) of 16b LPEs that we use for the comparison with SCNN. We limit attention to GoogleNet and ResNet50-S. Similarly to other accelerators performance improves well yet sub-linearly. As the number of LPEs increases so does the synchronization group size. As a result, value pairs with larger term counts impact more groups. Breaking the tile into multiple smaller ones would reduce this effect.

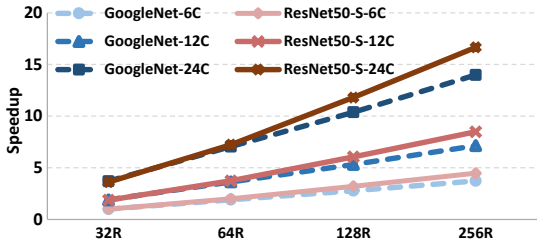


Figure 8: *Laconic*: Performance scaling as the number of LPE columns and rows grows.

Table 3: PE Area (μm^2) in 65nm: Scaling with Data Type.

	16b	8b	4b
Bit-Parallel	60,319	17,450	5,443
Laconic	8,298	3,456	1,565
Ratio PE/LPE	7.3×	5.0×	3.5×

Table 4: Area μm^2 with 8b PEs: Tech node scaling to 15nm.

	Tile		PE	
	65nm	**15nm**	**65nm**	**15nm**
Bit-Parallel	721,348	107,025	17,450	2,589
Laconic	749,339	109,874	3,456	507
Ratio	0.96×	0.97×	5.00×	5.10×

Data Type: This section studies how PE area scales for *Laconic* and *DaDianNao++* as the data width shrinks. While 16b quantization was state-of-the-art earlier on, today 8b quantization is possible for many image-based deep learning applications, and quantization to 4b is actively being pursued. e.g., [11]. We use *DaDianNao++* as representative of designs with bit-parallel PEs.

There are two major blocks in the PE: the multipliers and the adder-tree plus accumulator. In addition, there is area for the wiring of activations which are shared among all adder trees. In the 16b *DaDianNao++* these respectively occupy 80% and 12% of the PE area. The *Laconic* PE replaces the multipliers with small exponent adders. Moreover, rather than broadcasting 16b activations, the PE accepts 4b+sign exponents. *Laconic*'s adder tree and histogram block is slightly larger than the *DaDianNao++* adder tree. Hence, most of the area savings in *Laconic* PEs come from replacing the 16b parallel multipliers by exponent adders which also reduces wiring. As the data width shrinks, the multiplier area decreases quadratically while the area reduction for the exponent adders is linear. Accordingly, for the smaller data types, the ratio of *DaDianNao++* PE area over *Laconic* PE should decrease. Table 3 confirms this expectation. For 16b and 8b, the bit-parallel PE is respectively 7.3× and 5.0× larger than the *Laconic* PE. For 4b this ratio is 3.5×.

15nm Technology Node: This section studies how *Laconic* scales to a newer technology node: the 15nm FreePDK NanGate which uses multi-gate FinFET transistors [67]. Table 4 reports post-layout area and energy efficiency measurements for the 8b *Laconic* and the *DaDianNao++* PEs. *Laconic*'s PE scales well to 15nm. In fact, in

Table 5: 8b Tile Area Breakdown μm^2.

	Bit-Parallel	*Laconic*
PE Array	174,500	†497,553
SRAM	457,400	96,483
Activation Unit	89,448	143,116
Offset Gen.	N/A	12,187
Total	721,348	749,339

†*Includes 2 accumulators per LPE.*

Table 6: GOPS/W per Tile, 8b PEs: Tech node scaling.

	65nm	**15nm**
Bit-Parallel	703	1,694
Laconic	805	1,997
Ratio	0.87×	0.84×

15nm it is slightly smaller than the bit-parallel PE vs. 65m tech. node (6.82× vs. 6.74×). However, such a small difference is insufficient to draw any further conclusions.

Table 6 reports how energy per operation scales to 15nm. *Laconic*'s dynamic power consumption reduces by 2.48×. There two major reasons why: The supply voltage reduction from $1V$ in 65nm to $0.8V$ yields a 1.56× improvement. Capacitance reduction yields another 1.58× improvement. Leakage power reduces only by 1.05× and remains only 1% of the total power consumption.

5 RELATED WORK

Like Pragmatic [3], *Laconic* processes activations term-serially, however, *Laconic* also processes the weights term-serially exposing more ineffectual work. Stripes [48], exploits per-layer precisions for activations whereas BitFusion [83], and Loom [82] do so also for the weights and avoid ineffectual work due to a prefix or suffix of ineffectual bits. Dynamic Stripes can adjust the precision on-the-fly exposing more zero bits [22, 23]. *Laconic* exposes more ineffectual work as it exploits ineffectual bits that do not have to be part of a continuous group of prefix or suffix ineffectual bits.

Quantization reduces the data width of activations and weights, e.g., [90, 91]. Park *et al.,* propose an architecture where two different data types are used per layer and where the few values that require more bits are processed in wider and thus more energy hungry units. For the 8b MobileNet V2 maintaining TOP-1 accuracy required using 5b for the short data type [73]. For those networks where this quantization is possible there will be dynamically ineffectual bits which *Laconic* can exploit. Other work reports that with 8b quantization it was not possible to maintain accuracy for some recent image classification models [45]. Chen et al., report being able to maintain accuracy for ResNet models with 4b quantization for all convolutional layers but the first and last [11] by adding a trainable clipping parameter into the ReLU activation function. *Laconic* will benefit from such quantization methods without requiring that all models be amenable to them. If all models of interest can be quantized to an extremely low data width (e.g., 1b to 3b) using conventional MAC units will be more area efficient.

Pruning can eliminate synaptic connections by converting the corresponding weights to zero [33, 38]. The Cambricon-X accelerator exploits weight sparsity [101]. Furthermore, many activations tend to be zero primarily as a result of the ReLU activation function [4, 37, 99]. SCNN exploits zero weights and activations [71]. *Laconic* rewards sparsity in both the activations and the weights. Instead of relying on whole value sparsity, *Laconic* exploits on the more bit-level sparsity which is naturally present in either unpruned (dense) and pruned (sparse) networks. Ideally, in the very unlikely scenario where all values that are concurrently processed are zero, *Laconic* will take a single cycle to process them. That is, in this case, *Laconic* incurs an opportunity loss of 1/64 in 8b vs. an accelerator that completely eliminates zero values. In practice, it will be the pair with the highest number of terms that will determine how many cycles will be required to process the group. This represents a speedup opportunity loss for *Laconic* compared to accelerators that target zero values. However, *Laconic* speedups up computation for all values zero or not. In practice, the gain from accelerating the processing of non-zero values far outweighs the opportunity loss from not completely skipping zero values. Furthermore, the processing approach of *Laconic* is in principle compatible with SCNN as its units can be replaced with term-serial ones. However, such an SCNN variant will require more units and thus will suffer more from inter- and intra-tile work imbalance for the models studied here. However, when processing higher resolution images, this imbalance may not be significant. Investigating this option is left for future work.

Advances at the algorithmic level would impact *Laconic* as well or may even render it obsolete. For example, work on using binary weights [18] would obviate the need for an accelerator whose performance scales with weight precision. However, it is not presently possible to drastically reduce precision for every network. Moreover, *Laconic* rewards any advances in reducing precision, pruning, and quantization providing a gradual path towards innovation.

6 CONCLUSION

Laconic exploits an additional dimension of parallelism and sparsity, that of the bit-level and does so in a value-aware manner. As techniques that reduce the impact of off-chip access are perfected, *Laconic*'s approach can be used to improve on-chip energy efficiency. It benefits from advances in pruning and persists under quantization to 8b while not requiring either. Its PE design is unlike other multiply-accumulate units and its approach sufficiently different than others that can influence further work in the area. In particular, it will benefit and encourage quantization approaches that better balance bit-sparsity across weights and activations and enable experimentation with approximation approaches that reduce the number of terms per value or exploit intra-value effectual term-parallelism.
Acknowledgements: This work was supported by the NSERC COHESA Research Network, an NSERC Discovery Grant, and a Department of National Defense Discovery Grant Supplement.

REFERENCES

[1] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc J. Van Gool. 2017. Soft-to-Hard Vector Quantization for End-to-End Learning Compressible Representations. In *Advances in Neural Information Processing Systems 30: Annual Conf. on Neural Information Processing Systems 2017, 4-9 Dec. 2017, Long Beach, CA, USA*. 1141–1151.

[2] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. 2018. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *Intl' Symp. on Computer Architecture*.

[3] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic Deep Neural Network Computing. In *Intl' Symp. on Microarchitecture*.

[4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. CNVLUTIN: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Intl' Symp. on Computer Architecture*.

[5] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Intl' Symp. on Microarchitecture*.

[6] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. 2017. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence* (2017).

[7] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. 2015. Conditional Computation in Neural Networks for faster models. *CoRR* abs/1511.06297.

[8] Gabriel J. Brostow, Julien Fauqueur, and Roberto Cipolla. 2008. Semantic Object Classes in Video: A High-Definition Ground Truth Database. *Pattern Recognition Letters* (2008).

[9] Cadence. 2019. Encounter RTL Compiler. (2019). https://www.cadence.com

[10] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *Intl' Conf. on Learning Representations*. https://arxiv.org/pdf/1812.00332.pdf

[11] Chia-Yu Chen, Jungwook Choi, Kailash Gopalakrishnan, Viji Srinivasan, and Swagath Venkataramani. 2018. Exploiting approximate computing for deep learning acceleration. In *Design, Automation & Test in Europe Conf.*

[12] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Conf. on Architectural support for programming languages and operating systems*.

[13] 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 *Intl' Symp. on Microarchitecture*.

[14] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Intl' Symp. on Computer Architecture*.

[15] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan 2017).

[16] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2016. Towards the Limit of Network Quantization. *CoRR* abs/1612.01543 (2016).

[17] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Low precision arithmetic for deep learning. *CoRR* abs/1412.7024 (2014).

[18] M. Courbariaux, Y. Bengio, and J.-P. David. 2015. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *CoPR* abs/1511.00363.

[19] Bin Dai, Chen Zhu, and David P. Wipf. 2018. Compressing Neural Networks using the Variational Information Bottleneck. *CoRR* abs/1802.10399 (2018).

[20] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. *CoRR* abs/1901.02860 (2019). http://arxiv.org/abs/1901.02860

[21] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. 2017. Language Modeling with Gated Convolutional Networks. In *Intl' Conf. on Machine Learning*.

[22] 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).

[23] Alberto Delmas, Sayeh Sharify, Patrick Judd, Milos Nikolic, and Andreas Moshovos. 2018. DPRed: Making Typical Activation Values Matter In Deep Learning Computing. *CoRR* abs/1804.06732 (2018).

[24] 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 *Intl' Conf. on Architectural Support for Programming Languages and Operating Systems*.

[25] Lei Deng, Peng Jiao, Jing Pei, Zhenzhi Wu, and Guoqi Li. 2018. GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework. *Neural Networks* 100 (2018), 49–58.

[26] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. 2017. CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Intl' Symp. on Microarchitecture*.

[27] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Intl' Symp. on Computer Architecture*.

[28] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2015. The Pascal Visual Object Classes Challenge: A Retrospective. *Intl' Journal of Computer Vision* 111, 1 (Jan. 2015), 98–136.

[29] Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical Neural Story Generation. *CoRR* abs/1805.04833 (2018).

[30] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Intl' Conf. on Architectural Support for Programming Languages and Operating Systems*.

[31] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 751–764.

[32] Michaël Gharbi, Gaurav Chaurasia, Sylvain Paris, and Frédo Durand. 2016. Deep Joint Demosaicking and Denoising. *ACM Trans. on Graphics* (2016).

[33] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic Network Surgery for Efficient DNNs. *CoRR* abs/1608.04493 (2016).

[34] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Intl' Conf. on Machine Learning*.

[35] Song Han and William J. Dally. 2018. Bandwidth-efficient Deep Learning. In *Design Automation Conf.*

[36] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Intl' Symp. on Field-Programmable Gate Arrays*.

[37] 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 *Intl' Symp. on Computer Architecture*.

[38] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *CoPR* abs/1510.00149 (2015).

[39] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Intl' Conf. on Neural Information Processing Systems*.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015).

[41] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *European Conf. on Computer Vision*.

[42] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). *IEEE Intl' Solid-State Circuits Conf.* 57 (02 2014), 10–14.

[43] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research* 18 (2017), 187:1–187:30.

[44] Yani Ioannou, Duncan P. Robertson, Darko Zikic, Peter Kontschieder, Jamie Shotton, Matthew Brown, and Antonio Criminisi. 2016. Decision Forests, Convolutional Networks and the Models in-Between. *CoRR* abs/1603.01250 (2016).

[45] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR* abs/1712.05877 (2017).

[46] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In *Workshop On Approximate Computing (WAPCO)*.

[47] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Raquel Urtasun, and Andreas Moshovos. 2015. Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets. *CoPR* abs/1511.05236v4 (2015).

[48] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial Deep Neural Network Computing . In *Intl' Symp. on Microarchitecture*.

[49] Supriya Kapur, Asit K. Mishra, and Debbie Marr. 2017. Low Precision RNNs: Quantizing RNNs Without Losing Accuracy. *CoRR* abs/1710.07706 (2017).

[50] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2017. A novel zero weight/activation-aware hardware architecture of convolutional neural network. In *Design Automation and Test Europe*.

[51] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2018. ZeNA: Zero-Aware Neural Network Accelerator. *IEEE Design Test* 35 (2018), 39–46.

[52] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-density 3D Memory. In *Intl' Symp. on Computer Architecture*.

[53] Minje Kim and Paris Smaragdis. 2016. Bitwise Neural Networks. *CoRR* abs/1601.06071 (2016).

[54] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. In *Intl' Conf. on Architectural Support for*

Programming Languages and Operating Systems.

[55] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Intl' Conf. on Architectural Support for Programming Languages and Operating Systems*.

[56] Vadim Lebedev and Victor S. Lempitsky. 2016. Fast ConvNets Using Group-Wise Brain Damage. In *Computer Vision and Pattern Recognition*.

[57] Dongwoo Lee, Sungbum Kang, and Kiyoung Choi. 2018. ComPEND: Computation Pruning through Early Negative Detection for ReLU in a deep neural network accelerator. In *Intl' Conf. on Supercomputing*.

[58] Jonathan Lew, Deval Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, and Tor M. Aamodt. 2018. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator. *CoRR* abs/1811.08933 (2018). arXiv:1811.08933 http://arxiv.org/abs/1811.08933

[59] Dingyi Li and Zengfu Wang. 2017. Video Superresolution via Motion Compensation and Deep Residual Learning. *IEEE Trans. on Computational Imaging*.

[60] Fengfu Li and Bin Liu. 2016. Ternary Weight Networks. *CoRR* abs/1605.04711 (2016).

[61] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Intl' Conf. on Machine Learning*.

[62] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime Neural Pruning. In *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc.

[63] Zhenhong Liu, Amir Yazdanbakhsh, Taejoon Park, Hadi Esmaeilzadeh, and Nam Kim. 2018. SiMul: An Algorithm-Driven Approximate Multiplier Design for Machine Learning. *IEEE Micro* 38 (2018), 50–59.

[64] Christos Louizos, Karen Ullrich, and Max Welling. 2017. Bayesian Compression for Deep Learning. In *Conf. on Neural Information Processing Systems*.

[65] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *Intl' Symp. on High Performance Computer Architecture*.

[66] David Martin, Charless Fowlkes, Doron Tal, and Jitendra Malik. 2001. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Intl' Conf. on Computer Vision*, Vol. 2. 416–423 vol.2.

[67] Mayler Martins, Jody Maick Matos, Renato P Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. Open cell library in 15nm FreePDK technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design*. ACM, 171–178.

[68] Szymon Migacz. 2017. 8-bit Inference with TensorRT. GPU Technology Conf.

[69] Asit K. Mishra, Eriko Nurvitadhi, Jeffrey J. Cook, and Debbie Marr. 2017. WRPN: Wide Reduced-Precision Networks. *CoRR* abs/1709.01134 (2017).

[70] Naveen Muralimanohar and Rajeev Balasubramonian. 2015. CACTI 6.0: A Tool to Understand Large Caches. (2015).

[71] 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 *Intl' Symp. on Computer Architecture (ISCA '17)*.

[72] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Weighted-Entropy-Based Quantization for Deep Neural Networks. In *Conf. on Computer Vision and Pattern Recognition*.

[73] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *Intl' Symp. on Computer Architecture*.

[74] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. 2018. Value-Aware Quantization for Training and Inference of Neural Networks. In *European Conf. on Computer Vision*.

[75] 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. In *Intl' Conf. on Learning Representations*.

[76] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Intl' Symp. on Field-Programmable Gate Arrays*.

[77] Cyrus Rashtchian, Peter Young, Micah Hodosh, and Julia Hockenmaier. 2010. Collecting Image Annotations Using Amazon's Mechanical Turk. In *NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*.

[78] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR* abs/1603.05279 (2016).

[79] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *CoRR* abs/1612.08242 (2016).

[80] 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. *CoRR* abs/1409.0575 (Sept. 2014).

[81] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation. *CoRR* abs/1801.04381.

[82] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. 2018. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *Proceedings of the 55th Annual Design Automation Conference*. ACM, 20.

[83] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *ISCA*. IEEE Computer Society, 764–775.

[84] Sungho Shin, Kyuyeon Hwang, and Wonyong Sung. 2015. Fixed Point Performance Analysis of Recurrent Neural Networks. *CoRR* abs/1512.01322 (2015).

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

[86] Synopsys. 2019. Design Compiler. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages. (2019).

[87] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *IEEE Conf. on computer vision and pattern recognition*.

[88] Karen Ullrich, Edward Meeds, and Max Welling. 2017. Soft Weight-Sharing for Neural Network Compression. *CoRR* abs/1702.04008 (2017).

[89] Cheng Wang, Haojin Yang, Christian Bartz, and Christoph Meinel. 2016. Image captioning with deep bidirectional LSTMs. In *ACM Multimedia Conf.*

[90] Peter Warden. 2016. Low-precision matrix multiplication. https://petewarden.com.

[91] Pete Warden. 2017. How to Quantize Neural Networks with TensorFlow. https://www.tensorflow.org/performance/quantization. (2017).

[92] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *Annual Design Automation Conf.*

[93] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Intl' Conf. on Neural Information Processing Systems*.

[94] Neil HE Weste, David Harris, and Ayan Banerjee. 2010. *CMOS VLSI design*. Pearson India.

[95] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Bell, Jeff Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz. 2018. DNN Dataflow Choice Is Overrated. *CoRR* abs/1809.04070 (2018).

[96] 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).

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

[98] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. *Intl' Symp. on Computer Architecture* (2017).

[99] Yu-Hsin Chen and Tushar Krishna and Joel Emer and Vivienne Sze. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE Intl' Solid-State Circuits Conf.* 262–263.

[100] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Intl' Symp. on Field-Programmable Gate Arrays*.

[101] 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 *Intl' Symp. on Microarchitecture*.

[102] Chuan Zhang Tang and Hon Keung Kwan. 1993. Multilayer Feedforward Neural Networks with Single Powers-of-Two Weights. *IEEE Trans. on Signal Processing* 41 (09 1993), 2724 – 2727.

[103] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. *CoRR* abs/1702.03044 (2017).

[104] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR* abs/1606.06160 (2016).

[105] Xuda Zhou, Zidong Du, Qi Guo, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through a Cooperative Software/Hardware Approach. In *Intl' Symp. on Microarchitecture*.

[106] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2016. Trained Ternary Quantization. *CoRR* abs/1612.01064 (2016).

[107] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2017. Learning Transferable Architectures for Scalable Image Recognition. *CoRR* abs/1707.07012 (2017). arXiv:1707.07012 http://arxiv.org/abs/1707.07012