2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)

MASR: A Modular Accelerator for Sparse RNNs

Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe,
Alexander M. Rush, Gu-Yeon Wei, David Brooks
ugupta@g.harvard.edu

***Abstract*—Recurrent neural networks (RNNs) are becoming the *de facto* solution for speech recognition. RNNs exploit long-term temporal relationships in data by applying repeated, learned transformations. Unlike fully-connected (FC) layers with single vector matrix operations, RNN layers consist of *hundreds* of such operations chained over time. This poses challenges unique to RNNs that are not found in convolutional neural networks (CNNs) or FC models, namely large dynamic activation. In this paper we present MASR, a principled and modular architecture that accelerates bidirectional RNNs for on-chip ASR. MASR is designed to exploit sparsity in both dynamic activations and static weights. The architecture is enhanced by a series of dynamic activation optimizations that enable compact storage, ensure no energy is wasted computing null operations, and maintain high *MAC* utilization for highly parallel accelerator designs. In comparison to current state-of-the-art sparse neural network accelerators (e.g., EIE), MASR provides 2× area 3× energy, and 1.6× performance benefits. The modular nature of MASR enables designs that efficiently scale from resource-constrained low-power IoT applications to large-scale, highly parallel datacenter deployments.**

I. Introduction

Automatic speech recognition (ASR) is at the foundation of many popular services, streamlining the human-machine interface [1], [2]. Recent advances in ASR have come from replacing traditional methods based on Gaussian Mixture Models and Hidden Markov Models with deep learning, namely recurrent neural networks (RNNs). RNNs learn relationships in time series data by establishing a temporal context called the *hidden state*—partial predictions between time-adjacent neurons that improve the interpretation of sequential data (e.g., spoken utterances). Today, RNNs are the state-of-the-art solution for highly-accurate ASR [3], [4], [5], [6].

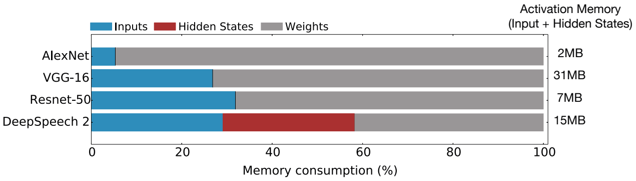


Fig. 1: The memory footprint of activations is higher in Deep Speech 2 (DS2), an ASR RNN, than in standard CNNs. Thus, to reduce storage costs of ASR RNNs, memory system optimizations are needed for *both* activations and weights.

The hidden state of RNNs introduces a unique memory consumption problem that is addressed in this paper. Figure 1 compares the fraction of memory used by activations and weights across four deep learning models. Well-known, CNN-based image classification models devote most of their memory resources to storing weights. In contrast, nearly 60% of the memory needed for Deep Speech 2 (DS2) —a state-of-the-art, RNN-based ASR model—is for activations (both inputs and hidden states), which consumes significant on-chip storage. This does not preclude the issue of weights also consuming significant memory (14MB for Deep Speech 2). These memory requirements are a result of ASR RNNs often using bidirectional layers — inputs to each layer are processed twice (once forwards in time and once backwards) and work over hundreds to thousands of time steps (i.e., 1 to 30 seconds) [7]. The hidden state size scales with the number of time steps, and separate weights are maintained for forward and backward passes.

Aggressive optimizations are needed to reduce the memory costs of storing *both* activations and weights, as well as the heavy processing load. One promising solution is leveraging sparsity for storage and computational efficiency. However, while many techniques for weight pruning and compression have been proposed, relatively little has been done to compress activations. To improve computational efficiency, inferences can be computed directly on the sparse encoding. Sparse processing allows the hardware to elide all null operations at the expense of introducing irregularity. Irregularity leads to hardware inefficiency from low utilization, and the optimal sparse encoding is application dependent. In addition to not considering activation sparsity, existing, CNN-centric solutions [8], [9], [10], [11] are either not applicable to RNNs or perform poorly. Enabling ubiquitous ASR requires accelerating RNNs with algorithm-architecture co-design for sparse storage and efficient execution.

This paper presents MASR: A modular accelerator to efficiently process sparse RNNs. Through algorithm-architecture co-design, MASR achieves high hardware utilization while never wasting area nor energy on superfluous computation. To demonstrate the efficacy of the proposed technique, we start by aggressively optimizing our baseline RNN with knowledge distillation, language modeling, weight pruning, and quantization. The key research contributions of MASR fall into three categories: a hardware accelerator that exploits sparsity in both weights and activations to skip null values in execution and storage, a co-designed sparse encoding technique for both activations and weights that enables highly parallel architectures, and a mechanism for dynamic load balancing to maximize hardware utilization.

Sparse ASR RNN accelerator *Algorithmic optimizations (i.e., knowledge distillation) and MASR's co-designed micro-*

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00009

TABLE I: Comparing MASR to related work in terms of support for sparse execution and storage running RNNs.

	Sparse Weight		Sparse Act.		Dyn. Load
	Exec.	Storage	Exec.	Storage	Balancing
E-PUR [13]					
Minerva [14]			x		
SparseNN [14]			x		
Camb-X [10]	x	x			
ESE/EIE [8]	x	x	x		
MASR	x	x	x	x	x

architecture exploit sparsity in weights and activations leading to improved performance, area, and energy by 14×, 2×, and 15× compared to a dense ASR RNN baseline.

In order to reduce storage and computational burdens of ASR RNNs, activations must be sparse. However, unlike CNNs, RNN activations (inputs and hidden states) are not typically sparse. To achieve hidden state sparsity we use knowledge distillation [12] to train RNNs with ReLU, at no loss in accuracy compared to GRU baselines with tanh. Furthermore, we maintain input sparsity across layers by refactoring the batch normalization operation. These modifications expose sufficient sparsity in RNNs to co-design our sparse activation-weight encoding.

Sparse encoding *MASR's low-cost and scalable sparse encoding technique, provides a 2× area, 3× energy, and 1.6× performance benefit relative to a start-of-the-art sparse DNN accelerator [8].*

MASR's sparse encoding format is co-designed with the underlying architecture to address both compute and storage bottlenecks. Existing sparse encodings, in addition to not compressing activations, exhibit high meta-data costs stemming from encoding overheads. MASR proposes a binary-mask sparse encoding scheme for both weights and activations. In storing bits rather than pointers, MASR replaces expensive memory addressing with cheap bit-wise operations.

Dynamic load balancing *MASR dynamically balances load from the irregular distribution of non-zero activations to improve performance by up to 30% and achieve high MAC-utilization across a wide range of parallel design points.*

Irregularity introduced by sparsity can lead to poor hardware utilization [10], [15], [16]. MASR is designed to maximize utilization by, (1) considering both intra- and inter-neuron parallelism, and (2) employing a decoupled pipeline to separate the irregularity from sparsity from the computation of partials. Once work is issued to the backend, the pipeline does not stall, regardless of the sparsity pattern. The remaining source of low utilization arises from load imbalance — pipelines with more sparsity complete before others. To improve hardware utilization we propose a dynamic load balancing technique to re-distribute activations at run-time with negligible area and energy overheads.

II. Related Work

Accelerating ASR RNNs Deep neural networks entail a general class of machine learning models that have been deployed across a wide set of applications and platforms [15], [17], [18], [19]. Given their ability to achieve state-of-the-art accuracy in a broad range of applications, DNNs have gained a lot of attention from the architecture community. However, much of the effort has been devoted to optimizing DNNs with only FC and CNN layers [8], [9], [10], [11], [13], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34]. RNNs, used widely in ASR and natural language processing, pose unique challenges. For instance, activations (inputs and hidden-states) generated at run-time comprise a higher fraction of the memory consumption in RNNs than in FC/CNNs (Figure 1).

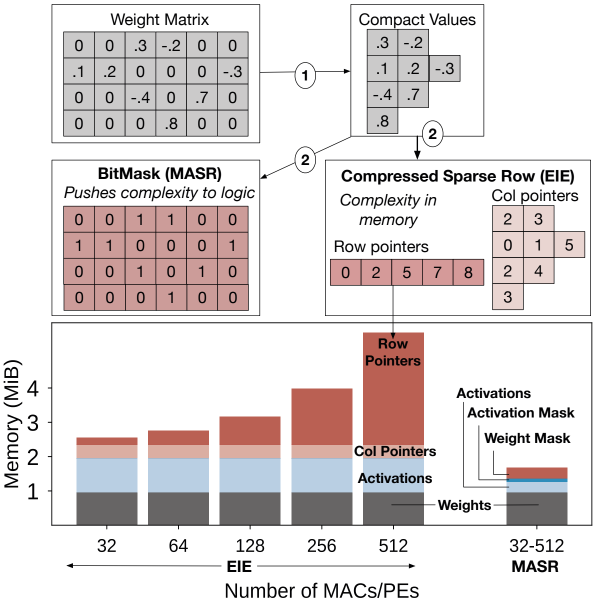


Fig. 2: Compared to compressed sparse row encoding (e.g., EIE [8]), MASR's bit-mask encoding pushes the complexity in sparse encoding away from storing pointers to logic. While storage for costly row pointers in EIE scales with the number of PEs, MASR's sparse encoding storage overhead is constant — providing scalability.

Beyond accelerators for DNNs and CNNs, other work has investigated RNNs, and search algorithms for ASR and machine translation [31], [35], [36], [37], [38], [39], [40]. Shown in Table I, E-PUR [13] provides a hardware accelerator that maximizes weight locality in dense RNNs. Similarly, the authors in [41], [42], [43] leverage the temporal locality of dense RNNs to accelerate them on FPGAs. In contrast, MASR exploits sparsity in both weights and activations to further improve performance, area, and energy efficiency.

To accelerate ASR, the authors of [44] design a memory-efficient Viterbi search accelerator. This targets language models that are run after processing all timesteps and layers in the RNN. However, with even large language models, state-of-the-art ASR models [3], [45] spend over 90% of their execution time on the RNNs (Section III-C), making RNNs the performance bottleneck and the focus of this paper.

Exploiting sparsity for hardware efficiency Table I compares MASR to previous hardware accelerators based on their

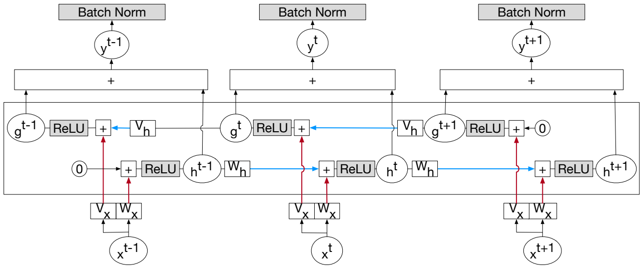


Fig. 3: Bidirectional RNN layer. x^t, h^t, and g^t are the input and hidden states at time step t. W_x, W_h, V_x, and V_h are the forward and backward weights.

support for sparsity in weights and activations, and dynamic load balancing. Typically, previous work either exploits sparsity in weights or activations, but not both [10], [11], [13], [14], [22], [46], [47] leaving key performance, area, and energy savings on the table. DNN accelerators that do exploit sparsity in both weights and activations use dataflows and sparse encodings specific to CNNs [9], [23]. Thus, to highlight the key contributions made in this paper, we provide in depth comparisons to EIE [8].

While EIE [8] exploits sparsity in both weights and activations, it does not store activations in a compressed format. Furthermore, EIE uses compressed-sparse row (CSR) encoding. As shown in Figure 2, CSR maintains separate row and column pointers to track non-zero weights. Row-pointer storage scales with the number of hardware PEs, levying high memory costs in more parallel architectures. In contrast, MASR uses a simpler sparse encoding that pushes the complexity of computing addresses for sparse parameters away from memory and into low-cost logic. This facilitates scaling the architecture to highly parallel designs (see Section IV-B for details).

Load balancing for sparse neural networks Exploiting sparsity in weights and activations comes at the expense of introducing irregularity into an otherwise regular workload. Irregularity leads to low hardware utilization from load imbalance. Prior work considers pruning to statically balance weight sparsity [47]. However, we find that the main source of imbalance in RNNs is the distribution of non-zero activations. Thus, MASR exploits a novel dynamic load balancing technique that balances non-zero activations at run-time (Section VIII).

III. Automatic Speech Recognition

Automatic speech recognition (ASR) transcribes an input x into text. The input speech is represented as a discrete time series of continuous feature vectors $x^1, \ldots, x^T$ derived from a spectrogram of power normalized audio clips. Current state-of-the-art models for ASR rely heavily on deep learning for acoustic modeling [48], [49]. Recently, RNNs have become the standard end-to-end deep learning approach for ASR [3], [4]. This section first provides an overview of RNNs and how they are used in ASR. We then simplify the neural networks to establish an efficient baseline RNN for ASR.

TABLE II: DS2 [52] model before optimizations (21.9 WER)

	Convolution	Bidirectional GRU	Fully-connected
Layers	2	5	1
Parameters	250K	38M	20K

A. Recurrent Neural Networks

Recurrent layers build context in time series data using hidden states to learn feature patterns in long-term data. There are three popular recurrent layers: vanilla RNN (hereafter referred to as RNN), GRU [50], and LSTM [51]. They differ in how new inputs are used to update hidden state. RNNs use two sets of weights: one for inputs and one for hidden states. GRUs and LSTMs expand upon RNNs with additional gated skip connections. These can improve accuracy by increasing expressiveness; however, the additional connections increase model size, where GRUs and LSTMs have 3× and 4× more weights than RNNs, respectively.

Recurrent layers can either be unidirectional or bidirectional. Unidirectional layers update hidden states based entirely on information from previous time steps. Bidirectional layers maintain two separate hidden states, one based on inputs from the past and one based on inputs from the future. While bidirectional layers can achieve higher accuracy, they require twice the number of parameters and operations.

Figure 3 illustrates a bidirectional RNN layer with ReLU activation and batch normalization. From the bottom, first a time-series input x^t is transformed by matrices W_x and V_x to produce *input intermediates* (red). Hidden states h^{t-1} and g^{t+1} are transformed by matrices W_h and V_h, respectively, to produce forward and backward *hidden intermediates* (blue). New hidden states h^t and g^t are then computed by passing the sum of the input and hidden intermediates through ReLU. The sum of these hidden states is output as y^t.

B. Target Model: Deep Speech 2

Deep Speech 2 (DS2) [3] is an industry and academic standard speech-to-text benchmark [52]. It directly maps input speech spectrograms to characters. Table II describes the architecture using an implementation based on GRUs. First, a pair of CNN layers extract relevant features from the input spectrogram and reduce the length. Next, bidirectional recurrent layers, which can either be GRU or RNN layers [3], learn time-series context. Finally, a FC layer makes output predictions, a probability distribution over characters at each time step. This distribution can be combined with a language model to produce better transcribed text.

Our models for ASR are trained using the open-source DS2 implementation in PyTorch [53], [54], [55] on the open-source LibriSpeech corpus [7]. The GRU network (described in Table II) has a word-error rate (WER) of 21.9, comparable to DS2 networks with a greedy decoder [3] and the target for standardized speech-to-text benchmarks [52]. The GRU layers make up over 99% of the model's parameters. Thus, this paper focuses on optimizing the recurrent layers of DS2.

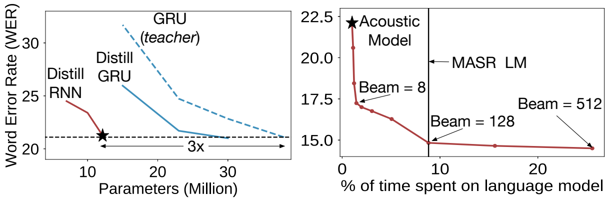


Fig. 4: **Left**: After distillation, a 5-layer bidirectional RNN reaches the accuracy of a 5-layer bidirectional GRU reducing the number of parameters by 3×. **Right**: Language modeling reduces the WER from 22 down to 14.5 with a beam-width of 128. Language modeling accounts for only 10% of CPU time.

C. An Efficient RNN Baseline

MASR builds on a very efficient baseline design that includes several previously-proposed optimization techniques to improve performance, on-chip area, and energy costs of DNNs. These techniques—knowledge distillation, language modeling, weight pruning, and quantization—were adapted for ASR RNNs.

Knowledge Distillation is a technique used to train a smaller, less complex *student* network to mimic the predictions of a large, pre-trained *teacher* network by penalizing it for diverging from the teacher's scores [12]. The *teacher* network is the 5-layer bidirectional GRU, shown in Table II, while the student models are a 4-layer GRU and a 5-layer RNN. Using distillation alone is insufficient to recover the baseline accuracy of the teacher network. Instead we start with distillation and then fine-tune for ASR with CTC [55] for 5 epochs. This combination yields *student* networks with the same accuracy as the teacher (Figure 4). Compared to the 5-layer *teacher* GRU and a 7-layer RNN (iso-accuracy using traditional supervised learning), the distilled 5-layer RNN has 3× and 1.4× fewer parameters, respectively. (All recurrent layers have 800 hidden units, wherein weight matrices are 800x800.) Knowledge distillation—reducing a dense DNNs to smaller ones —improves performance and energy significantly and provides immediate benefits on CPUs, GPUs, and specialized hardware.

Language Modeling is a post-processing step that reduces the WER by modifying the output of the RNNs (after all layers and timesteps) based on language semantics and structure [55]. This can be done greedily or using beam search, which maintains many likely speech-to-text transcriptions (determined by the beam-width). Figure 4(right) shows the decrease in WER as we increase the beam width from 1 (greedy) to 512 in increments of power of two. While the execution time for previous generations of ASR models has been dominated by the language model [44], this is not the case for newer ones like DS2. With a beam width of 128, only 10% of the CPU time is spent on performing beam search; the rest is spent running the RNN. Furthermore, previous work has proposed specialize hardware to accelerate beam search by at least 5× [56]. Thus, for the purposes of this study, we focus on accelerating the core RNN layers, the main performance bottleneck.

Weight Pruning eliminates less important weights and transforms dense matrix-vector multiplications to sparse ones. Pruning is performed by iteratively zeroing and masking out parameters, based on absolute value, and then retraining the network [57], [58], [59]. The number of non-zero parameters can be reduced to 33% in the already distilled RNNs (iso-accuracy).

Sparse Linear Quantization reduces the storage overheads of parameters by transforming them from 32-bit floating point type to reduced precision. By applying simple linear quantization [8], [9], [22], [47], the pruned and distilled network can be represented in fixed-point format with 14 bits. However, after pruning, the remaining non-zero parameters follow a skewed distribution with either high-negative or high-positive magnitude. Thus, before applying quantization, we separately scale the magnitude of the positive and negative weights to fit within the range $[0, 1]$. This enables further reducing the precision down to 10 bits without sacrificing accuracy.

TABLE III: **Efficient RNN baseline** after model optimizations.

Layer Type	Activation	Layers	Params	Bitwidth	NZ %
Bi-dir RNN	ReLU	5	13M	10	33

Together, the above-mentioned optimizations improve performance, area, and energy by 4.2×, 3×, and 8×, respectively. The parameters for the efficient baseline are shown in Table III.

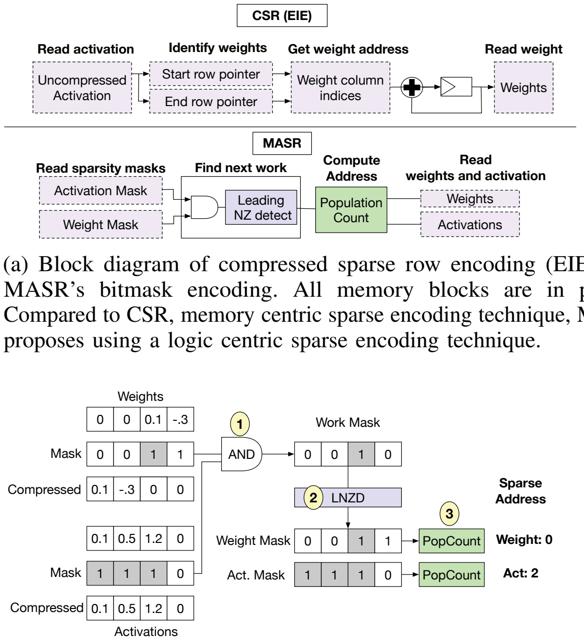
D. Supporting recurrent networks more generally

While the remainder of this paper focuses on accelerating the ASR RNN baseline, the key contributions of MASR apply to recurrent neural networks with weight and activation sparsity more generally. First, RNNs have crafted various speech recognition networks, notably transducer (e.g., DS2), seq2seq, and attention based architectures. Previous work has trained these architectures with ReLU activated RNNs, enabling the activation sparsity that MASR exploits [60]. Next, the DS2-style RNN studied in this paper forms the encoder in multi-stage ASR networks [60], [61]. Finally, the core micro-architectural contributions also apply to GRU-based networks with pruned weights and ReLU non-linearity for sparse activations. Such ReLU activated GRUs have also been used in transducer, seq2seq, and attention based speech recognition networks [3], [60], [62].

In order to optimize this vast design space of recurrent neural networks, MASR can be configured with a combination of dynamic and design-time parameters. Dynamic, run-time parameters include number of hidden-units, number of timesteps, and whether the RNN is uni-directional or bi-directional (see Section V for details). Design-time parameters include whether the network is an RNN or GRU, and the maximum recurrent network size supported.

IV. Optimizing Dynamic Activations

As shown in Figure 1, activations are the primary memory bottleneck of RNNs. This section presents the methods used to enforce sparsity in activations and the proposed sparse encoding algorithm.



(a) Block diagram of compressed sparse row encoding (EIE) and MASR's bitmask encoding. All memory blocks are in purple. Compared to CSR, memory centric sparse encoding technique, MASR proposes using a logic centric sparse encoding technique.

(b) Concrete example of MASR's sparse encoding. Step 1 determines the pairs of non-zero weights and activations, to produce the work mask, using a logical AND. Step 2 computes the next non-zero weight and activation to fetch from using with a leading non-zero detect. Finally, step 3 evaluates the address of sparse weights and activations stored compactly in memory.

Fig. 5: MASR's sparse encoding compute sparse addresses for weights and activations in logic using low-cost hardware.

A. Activation sparsification

Sequential processing: The core computation kernels of RNNs are the matrix-vector multiplications for the input and hidden states, $h^t = ReLU(W_x x^t + W_h h^{t-1})$. These kernels can be computed either in parallel or sequentially in time. E-PUR [13] proposes maximizing weight locality by first computing the input connections, $W_x x^t$, for all time steps in parallel, followed by the recurrent connections. Even with aggressive 10-bit quantization, this approach requires significant on-chip storage (1.25MB), outweighing the benefits of reducing on-chip storage through weight reuse.

MASR computes each time step sequentially. Sequential processing halves the amount of intermediate values to store. More importantly, as we use a ReLU activation function, by sequentially processing each time step intermediates are sparse and amenable to compression.

Hidden state sparsity To further reduce on-chip storage requirements for activations, MASR makes use of the sparsity in inputs and hidden states. Recall that our efficient baseline model is a 5-layer RNN with ReLU, i.e., $\max\{0,x\}$. Training with ReLU causes 80% of the hidden state values to be zero.

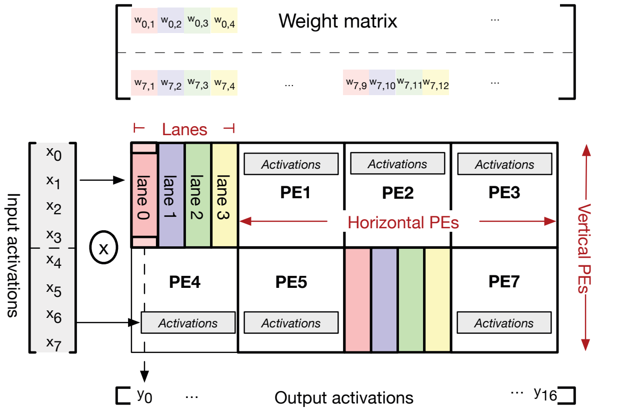


Fig. 6: Overall topology of how weight matrices, input activations, and output activations are split across the MASR architecture. MASR is organized as a 2D-array of horizontal (output neuron dimension) and vertical (input neuron dimension) PEs/lanes.

Input sparsity Input sparsity is lost due to batch normalization, a regularization technique that makes training larger models easier, between layers. The operation adjusts and scales activations to have a zero mean and unit variance: $x = \frac{x_{sp}-\mu}{\sqrt{\sigma^2+\varepsilon}} \times \gamma + \beta$. where x_{sp} represents the sparse inputs and μ, σ,ε, γ, and β represent learned parameters. The linear transform employs non-zero shifts (i.e., μ, β) that map sparse inputs to dense ones. However, during inference, the linear transformation can be statically refactored into the next layer's weights at zero cost:

$$x = K_0 x_{sp} + K_1 \quad K_0 = \frac{\gamma}{\sqrt{\sigma^2+\varepsilon}} \quad K_1 \quad = \beta - \frac{\gamma\mu}{\sqrt{\sigma^2+\varepsilon}}$$

To refactor this computation, we multiply the next layer's weights and biases by the K_0 and K_1 constants, respectively. Note that this refactoring is applicable to a broader set of neural networks that use batch normalization through depth [60], [62]. After refactoring, inputs are on average 60% zeros.

B. Compact activation storage

Operating over compressed weights and activations introduces two challenges: (1) aligning pairs of non-zero weights and activations, and (2) generating addresses for weights and activations stored compactly in memory. MASR addresses these challenges by co-designing a sparse encoding technique for both activations and weights. As shown in Figure 5a, the sparse encoding technique uses a combination of bitmasks, a leading non-zero detects (LNZD), and population counts. We start by reading the weight and activation bitmasks. The bitmasks track the sparsity pattern as bit vectors, where non-zero entries are represented as ones. Next, a bitwise AND between the weight and activation masks, determines pairs of non-zero weights and activations and produces the work mask. The ones in the work mask denote the absolute minimum work to compute. A LNZD over the work mask determines the index of the next non-zero weight and activation to fetch from memory. Finally, population counts of the weight and activation masks, up to the index specified by the LNZD, evaluates addresses of sparse weight and activations stored compactly in memory.

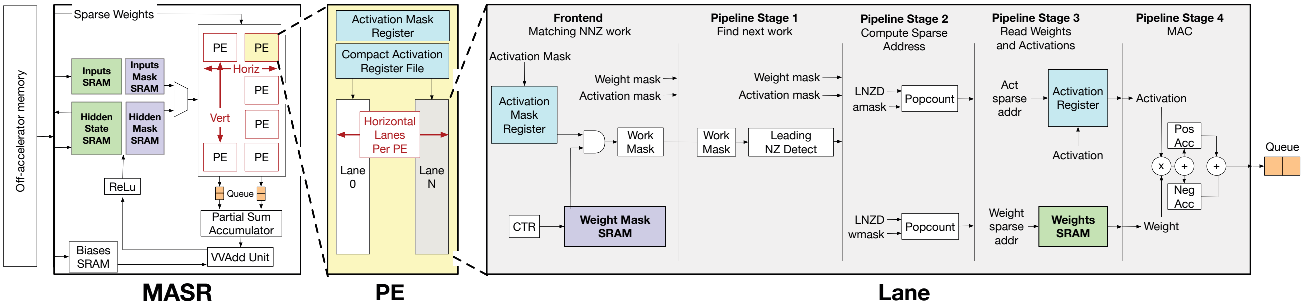


Fig. 7: MASR accelerator design highlighting the overall system architecture (left), a PE (center), and a lane (right). Blocks outlined in red represent tunable micro-architectural parameters swept in the design space exploration. Also in color: activation registers (blue), and SRAMs for binary masks (purple) and for compressed sparse activations and weights (green).

Example MASR encoding Figure 5b provides a concrete example of MASR's sparse encoding. The logical AND between the weight (0011) and activation (1110) masks produces the work mask (0010). The LNZD over the work mask points to index 2. Population counts up to index 2 for the weight (**00**11) and activation (**11**10) masks, compute the weight (0) and activation (2) addresses, respectively.

Comparing MASR to run-length and CSR The optimal encoding is application specific and depends on sparsity and matrix size. Previous sparse DNN accelerators typically use run-length encoding or CSR [8], [9], [10], [47].

Run-length encoding maintains a step index that stores the distance between non-zero weights [10]. However, it does not design for sparsity in activations, leaving key storage, performance, and energy savings on the table.

CSR considers sparsity in both weights and activations. As shown in Figure 5a, CSR first reads the non-zero activation address, encoded using a run-length style step index. The non-zero activation address then indexes separate row pointer memories to identify the first and last non-zero weights corresponding to the given input activation. Finally, the row pointers are used to read column indices, also encoded using a run-length style step index, which generate the address of weights stored compactly. While this approach works well for model with high sparsity, it suffers from two main drawbacks. First, null activations are skipped in execution not storage. Second, while column pointers scale with the number of non-zero weights, each *MAC*/PE maintains its own set of row pointers in CSR. As a result, row pointer memory scales with the number parallel *MAC*s/PEs. Figure 2 shows that as the hardware scales from 32 to 512 parallel *MAC*s/PEs, row pointers dominate the memory footprint.

Low overhead and scalable sparse encoding In contrast, the memory footprint for MASR's sparse encoding technique does not scale with the number of parallel *MAC*s/PEs. This is a result of eliminating the row pointers and identifying the necessary sparse weights and activations by computing the alignment in logic. For instance, MASR computes the address of non-zero weight and activation pairs in logic, as shown in Figure 5a. The memory overheads for encoding sparsity in MASR are limited to binary masks, which are determined by the size neural network model and *not* the number of *MAC*s/PEs. Thus, MASR has a significantly lower memory footprint compared to previous sparse neural network accelerators (i.e., EIE, ESE [8], [47]); see Section IX for a detailed quantitative comparison.

V. The MASR Architecture

As shown in Figure 6, MASR is composed of a 2D-array of processing elements (PEs)/lanes that evenly split each weight matrix in the horizontal (i.e., output neurons) and vertical (i.e., input neurons) dimensions. Each PE is a collection of lanes that share a local activation register file. Each lane has its own local weight and weight mask SRAMs that store an equal portion of the matrix. Compact weight matrices (only non-zero elements) are loaded from off-accelerator memories directly into local SRAMs. Output neurons are computed by accumulating the partial products across lanes in vertical PEs (i.e., lane 0 in PE0 and P4 determine y_0 in Figure 6). Decoupling the execution across the 32 to 1024 lanes is crucial for extracting parallelism of the irregular sparse computations at scale. Figure 7 shows the detailed architecture for MASR, focusing on RNN computations outlined in Figure 3. The modular design is centered around the 2D array of PEs and decoupled, pipelined lanes. In this section, we first explain how bidirectional RNN computations are mapped to MASR. Then we show how the underlying lane micro-architecture handles sparsity in weights and activations.

A. Mapping RNN computations to MASR

To process speech samples, the accelerator runs each layer of the bidirectional RNN in order. Within each layer, the accelerator first executes all time steps in the forward direction and then in the backward direction. Recall that each time step of the RNN comprises two matrix-vector multiplications, a vector-vector addition and ReLU: $h^t = \mathrm{ReLU}(W_x x^t + W_h h^{t-1})$. To begin processing a layer, all weights for the forward pass (W_x and W_h) are loaded from off-accelerator memory into the compact weight SRAMs within the lanes. Weight SRAMs have a word width of 10 bits (1 weight each). Likewise, all inputs (x^t) are loaded into compact activation SRAMs, which have a word width of 60 bits (six activations each).

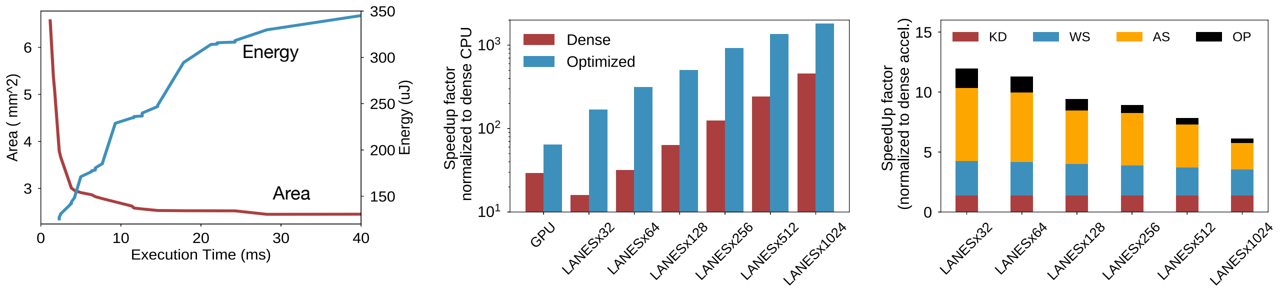


Fig. 8: **Left**: Energy-performance and area-performance Pareto frontiers of accelerator designs, sweeping microarchitectural parameters shown in Figure 7. **Center**: Speedup of varying MASR designs normalized to a CPU running a dense baseline RNN. **Right**: Breakdown of performance benefits for each proposed optimization on MASR designs.

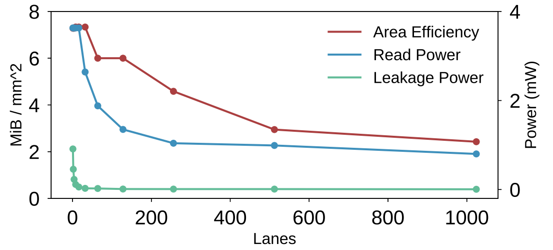


Fig. 9: Impact of increasing parallelism (lanes) on SRAM area efficiency, dynamic read power, and leakage power.

TABLE IV: MASR design parameters

Lanes	32	256	1024
Weights per lane (KB)	32	4	1
Weight masks per lane (KB)	10	1.25	0.3
Total weight (KB)	1280		
Total activations (KB)	450		
Weight width (bits)	10		
Activation width (bits)	10		
Technology node	16nm		
Frequency (MHz)	1000		

While MASR processes all time steps in the forward pass, weights for the backward pass are concurrently loaded into separate SRAMs. This double buffering of the forward and backward weights reduces performance penalties from not having the entire layer's weights stored locally on-chip. Similarly, activations beyond 333 timesteps (the average length of speech samples in Librispeech) are also double buffered. Section VII-C discusses the design decisions of double buffering.

Hidden state computation: For each time step t, the accelerator first processes the matrix-vector multiplication for hidden state, $W_h h^{t-1}$. This computation is initiated by loading the previous time step's hidden states from the compact activation SRAM to compact activation register files within each PE. The entirety of the matrix-vector multiplication is parallelized across the 2D array of PEs with multiple decoupled lanes. Horizontal lanes evenly split columns (output dimension) of the matrix, while vertical lanes evenly split rows (input dimension). This enables balancing parallelism across the input and output dimensions. Each lane is responsible for computing partial products for a subset of rows and columns in the matrix-vector product. As lanes finish processing each column, the partial sum accumulator sums the partial products for each output. An 800 element register file stores the outputs.

Finishing one time step: The above sequence repeats to process the matrix-vector multiplication for inputs, $W_x x^t$. Once both matrix-vector multiplications have been processed, the vector-vector add unit accumulates the biases, input intermediates, and hidden intermediates. The resulting output values are thresholded with ReLU and compactly written to the hidden-state SRAM for the subsequent time step. This completes one time step of the RNN layer.

Hardware implications of parallelism: The number of parallel lanes determines the degree of parallelism and how weights are partitioned across SRAMs. With 1024 lanes, each lane's weight and weight mask SRAM stores $\frac{1}{1024}$ of the parameters. Similarly, number of vertical lanes determines the size of the compact activation register files. For example, with 32 vertical lanes, each register file only tracks $\frac{1}{32}$ of the values. Horizontal lanes in a row process the same portion of the activation vectors. To reduce the cost of duplicated activations, horizontal lanes within a PE share a physical activation register file. Given that lanes are decoupled, increasing the number of lanes per PE requires additional ports to the physical register file, which increases the register file's size and cost per access. Note that MASR's decoupled PE/lane architecture does not depend on complex crossbar architectures that can limit efficiency of highly parallel sparse DNN accelerators. Section VII explores the design space encompassed by these parameters. Table IV illustrates the parameters for LANESx32, LANESx256, and LANESx1024.

Outside of the PEs, the MASR architecture has two additional parameters: depth of the back end queues and number of banks for activation SRAMs. The partial sum accumulators accumulate the output of each column once all lanes in the given vertical slice finish generating their partial product. Lanes that finish early are stalled, reducing the performance of the overall design. Increasing the depth of the back end queues reduces this back pressure. However, this comes at an area and energy cost, given each lane pushes partial products to a separate back end queue. In addition, the vector-vector add unit can be parallelized. This involves not only duplicating the number of adders but also partitioning the activation SRAMs into multiple banks (see Section VIII for details).

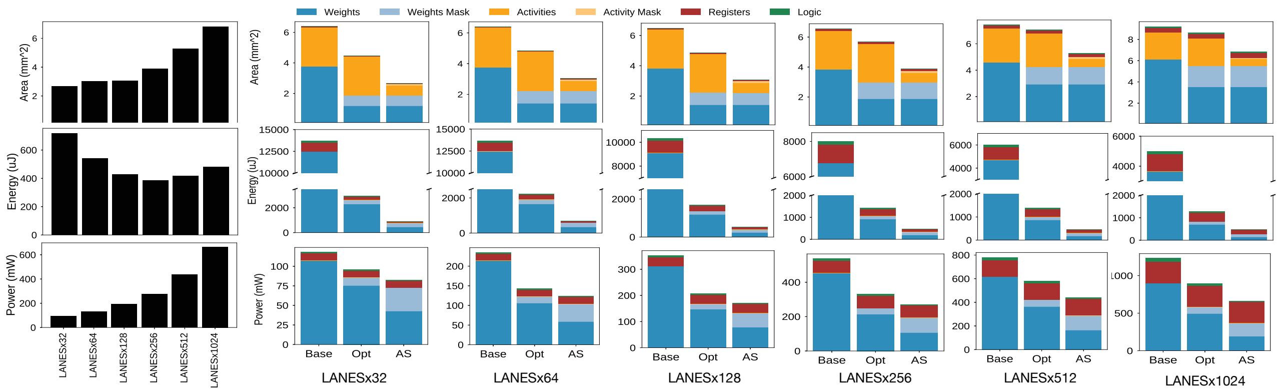


Fig. 10: The plots on the left summarizes area (top row), energy (middle row), and power (bottom row) tradeoffs for the fully optimized designs for various MASR design points. On the right we breakdown each optimization and each resource (weights, activations, sparse encoding masks, registers, and logic). The breakdowns are for the optimized baseline (distilled, Section III-C), optimized (weight pruning), and Act (sparse activations, fully optimized).

B. The MASR Lane

The MASR lane is the main computational workhorse for sparse vector-matrix multiplication. Each pipelined lane is organized in two phases, front end and back end. Intuitively, the front end decodes work from weight and activation masks, whereas the back end performs *MAC*s after accessing compact weights and activations. This eliminates wasted work in the back end, regardless of the distribution of sparse weights, activations, and outputs.

The front end first reads the binary masks, for both weights and activations. The binary masks are then `ANDED` together and the resulting *work* mask represents the absolute minimum non-zero weights and activations to accumulate.

The back end has four pipeline stages. Stage 1 receives the work mask and uses a single-cycle LNZD to find the next pair of non-zero weights and activations. Stage 2 computes the relative addresses using the LNZD output and population counts of the weight and activation masks. Stage 3 reads the weights SRAM and activation register file. Stage 4 evaluates the *MAC*. Separate accumulators are maintained for the positive and negative weights as they were quantized separately (see in Section 2). When the computation for the output neuron finishes, the partial sum is pushed onto the queue.

VI. Evaluation Methodology

The accelerator design space we explore is vast and each point is evaluated running the entire forward and backward passes of the bidirectional RNN. We validate a custom cycle-level C++ simulator of the accelerator with a synthesized RTL implementation. We annotate the simulator based on PPA characterizations from synthesized RTL using a commercial 16nm FinFET standard cell library at 1GHz. To model the SRAM area, energy, and power consumption, we use a commercial memory compiler in the same process.

We also evaluate the benefits of sparsity on CPUs and GPUs by profiling GEMM and SPMV kernels on real machines. For CPU baselines we run the Eigen library on a desktop Intel Core i7-6700K with SIMD support, using the `-O3` and `-ffast-math` compiler flags. The GPU baselines run GEMM and SPMV kernels provided by Deep Bench [63], using cuBLAS/cuSparse libraries on a NVIDIA GTX 1080 GPU.

TABLE V: Topology of MASR Pareto front points.

Accel	Horiz Lanes	Vert Lanes	Horiz PEs
LANESx32	16	2	2
LANESx64	32	2	2
LANESx128	32	4	2
LANESx256	32	8	2
LANESx512	32	16	1
LANESx1024	32	32	1

VII. Perf., Area, Energy, and Power Benefits

Optimal configuration of MASR's modular architecture depends on the intended use case. This section presents results of an extensive design space exploration of MASR's free parameters that exposes energy-performance tradeoffs. We then analyze the performance, area, and energy/power breakdowns for points along the Pareto frontier of the design space in order to quantify the benefits of each optimization and identify where resources are being consumed. For all experiments in this section, we fix the depth of the back end accumulator queue to a single element and assume one-bank activation SRAMs. We report the performance, area, and energy consumed for an accelerator provisioned to run a full seven seconds of speech, the average sample length in the Librispeech corpus, across multi-layer bidirectional RNNs. Finally, we discuss how the design scales when running shorter and longer speech samples.

A. Design Space Exploration

MASR's modularity enables both high performance and low power solutions. The tunable microarchitectural parameters considered in the design space, outlined in Figure 7, include the number of horizontal lanes, number of vertical lanes, and number of horizontal PEs. All possible configurations are swept so that the total number of lanes ranges from 1 to 1024 at powers of 2, with a maximum of 32 lanes in either dimension.

Sweeping the total number of lanes produces the energy-area-performance Pareto frontiers illustrated in Figure 8 (left). As parallelism increases, execution time and energy consumption decrease while area increases. This is a result of partitioning SRAMs into smaller arrays in order to support the bandwidth needed for more parallel datapaths. Figure 9 shows that partitioning SRAMs decreases the power per read and per-bit area efficiency. Even in highly parallel architectures such as LANESx1024, SRAM leakage is a small fraction of the overall energy due to the highly optimized 16nm FinFET libraries.

In addition to lane count the organization of lanes/PEs has an impact on accelerator performance. Table V shows Pareto optimal designs tend to have more horizontal lanes than vertical ones. Increasing the number of vertical lanes reduces the number of rows each lane processes, and thus also reduces the activation register file size. For example, with 8 vertical lanes, the activation register files contain 64 words; with 32 vertical lanes, they contain 16. Processing a small number of activations leads to load imbalance across lanes, degrading performance. A solution to this problem is discussed in Section VIII. The following sections detail the performance, area, energy, and power characteristics of the optimal designs.

B. *Performance*

Figure 8 (center) shows the speedup by running a dense 7-layer bidirectional RNN and the efficient baseline on MASR and a GPU, normalized to running the dense network on a CPU. Performance is measured as the execution time to process the full 7-seconds of speech. While the more programmable GPU benefits from knowledge distillation (1.4×) and weight pruning (3×), it is unable to exploit activation sparsity. The MASR designs benefit from knowledge distillation, weight pruning, and sparse activation execution, improving the performance of the accelerator beyond that of the more programmable systems.

Figure 8 (right) breaks down the performance benefit that each optimization offers. We find that the overall benefits of sparse optimizations diminish as parallelism increases. While each design observes speedup from knowledge distillation and sparse weight execution, speedup from sparse activation execution does not scale as gracefully. For example, we find sparse activations provide up to 3.75× speedup on the LANESx32 design, while their benefit on LANESx1024 is reduced to 2.2× due to higher load imbalance in more parallel accelerator topologies. Section VIII proposes low-cost solutions to balance dynamic activations at run-time — improving speedup from sparse execution by 1.7× and allowing even the most parallel designs to achieve near-linear speedup.

Output sparsity In addition to exploiting input neuron sparsity, x^t and h^t, prior work exploits sparsity in output neurons [14], [64]. This requires predicting output neurons that will be masked by ReLU. While the focus of paper is on exploiting weight and activation sparsity, MASR can also support output sparsity. In particular, input intermediates, $W_x x^t$, and hidden-state intermediates, $W_h h^t$, follow distinct distributions. Batch-normalizaton only operates on inputs, x^t (zero mean) causing hidden-state intermediates to be more negative than input intermediates. Highly negative hidden intermediates, computed first, will likely be zeroed out by the ReLU function even after accumulating with input intermediates. Thus, input intermediate calculations are skipped if the corresponding hidden intermediate is sufficiently negative, akin to the output sparsity predictor in [14]. Figure 8(right) shows that output predication (*OP*) improves performance by up to 15%.

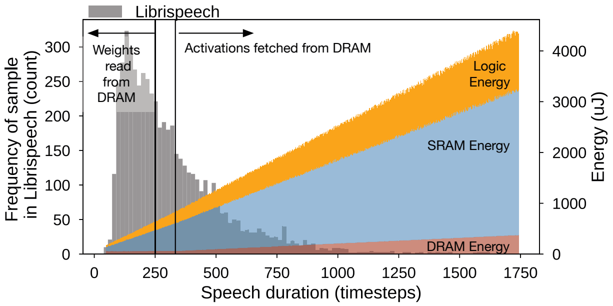


Fig. 11: Energy breakdown of LANESx256 running speech of varying length overlaid with the distribution of samples found in Librispeech. DRAM energy cost, for double buffering weights and activations, is small.

C. *Area, Energy, and Power*

Figure 10 shows the area, energy, and power breakdowns for each design point along the energy-performance Pareto frontier. The left column illustrates the overall trends as the accelerator scales to more parallel design points. The remaining columns on the right side provide detailed resource breakdowns. To understand the benefits of each optimization in accelerators with varying degrees of parallelism, three sets of resource breakdown bars correspond to design variants provisioned to run a dense 7-layer bidirectional RNN (*Base*), the efficient baseline that applies knowledge distillation and weight pruning (*Opt*), and with sparse activations (*AS*).

Area: Figure 10 (top row, right) shows the area footprints of each accelerator design. Partitioning weight and weight mask SRAMs diminishes the benefits of compactly storing weights in more parallel designs. For instance, for the LANESx64 architecture, starting from (*base*), weight sparsity (*opt*) reduces area from 5.5mm^2 down to 4.0mm^2 (1.5mm^2 benefit). On the other hand, weight sparsity reduces LANESx1024 area from 8.3mm^2 to 7.8mm^2 (0.5mm^2 benefit).

After compressing the weights, quantized activations consume up to 50% of the accelerator area, especially in the smaller LANESx32-256 designs. Compact activation storage reduces the memory area consumed by activations by 3×. This corresponds to reducing the area devoted to activations alone from 1.7mm^2 to 0.6mm^2. Given MASR's modular design, compact activation storage provides the same area benefit to all design variants; input and hidden-state memories are maintained outside of the PEs/lanes. This modularity also facilitates scaling the architecture to domains that may require processing much larger speech samples [2]. For instance, provisioning MASR to

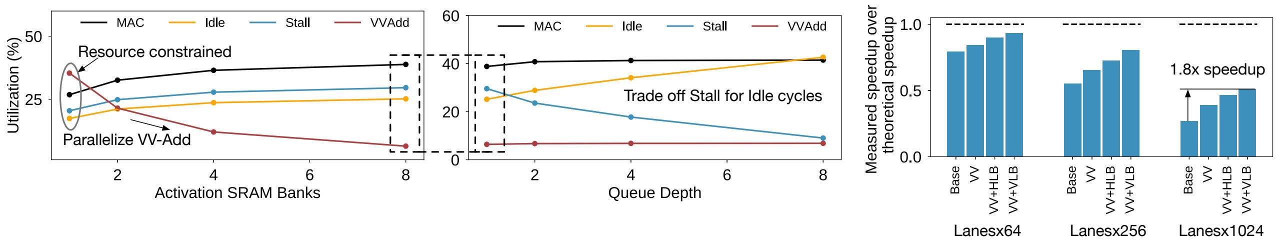


Fig. 12: **Left**: As we scale the number of activation SRAM banks in the LANESx1024 architecture, the cycles spent on VVAdd decreases. **Center**: Scaling the depth for back-end accumulator queue trades off stalls for idle cycles. **Right**: The impact of increasing the number of activation SRAM banks (VV) combined with either horizontal load balancing (VV+HLB) or vertical load balancing (VV+VLB) on the performances of LANESx64, LANESx256, and LANESx1024..

process up to 15 seconds of speech, compact activation storage would save $2.2mm^2$, reducing overall area by 1.8×.

After weights and activations, the remaining area is consumed by registers and logic. The increase in register area across more parallel designs is dominated tracking more weights and activation bitmasks per lane. These bitmasks account for over 90% of the register area. The secondary consumers (8%) of the register area are the backend queues.

Energy and Power: The energy breakdown across each accelerator design is shown Figure 10(middle row). The left column shows that LANESx256 is the energy-optimal design point even though LANESx1024 uses smaller SRAMs that dissipate less read power. The reason is two-fold: per-read energy cost plateaus in the most parallel accelerator designs, and the proportion of power consumed by registers increases for larger accelerators. As previously discussed, the first effect occurs because the smaller memories used in the more parallel designs (LANESx512 and LANESx1024) do not reduce dynamic read power proportionally to capacity reduction. The second effect comes from more parallel designs requiring the maintenance of more active states.

Generally, energy savings come from doing less work and making fewer SRAM accesses. For example, Figure 10 (middle) shows that compared to the dense RNN (*base*), the efficient baseline reduces the energy consumed by 4.2× (i.e., 1.4× from knowledge distillation, 3× from weight pruning) across all accelerator designs. Similarly, sparse activation execution further reduces energy by around 2.5×.

Because of MASR's sparse encoding mechanism, sparsity optimizations impact energy more than power. As long as work remains, a *MAC* is issued to the lane on every cycle, keeping power relatively constant. The 1.4× power reduction between dense RNN (*base*) and optimized baseline (*opt*) comes from decreasing the size of weight SRAMs by storing fewer non-zero parameters (Figure 10, middle and bottom rows).

D. Supporting Speech of Arbitrary Length

ASR models comprise millions of parameters which cannot be realistically stored on-chip. The storage requirements are further exacerbated when considering activation memory for longer speech samples. To minimize on-chip SRAM, MASR double buffers both weights and activations.

Performance and area Using LPDDR4 as off-accelerator memory, the performance and energy penalty of double buffering is *relatively* low. MASR double buffers forwards and backwards weights in separate SRAMs. LPDDR4 supports a bandwidth of 25.6GB/s while dissipating $200mW$ of power [65]. At this rate it takes 0.019ms to read in a layer's weights. This corresponds, roughly, to the time it takes process 250 timesteps of speech. Thus, to avoid memory contention, MASR stores activations for the first 333 timesteps (the average length in Librispeech) on-chip. Activations for later timesteps are double buffered within an 800-element register, which incurs negligible area overheads. Similarly, there is no performance penalty since the time to read activations from LPDDR4 is strictly less than the time to process a single timestep.

Energy Figure 11 illustrates the DRAM energy cost relative to on-chip SRAM and logic for samples from 50 timesteps to 1600 timesteps. For samples less than 333 timesteps, the DRAM energy consists solely of reading weights. This energy overhead is amortized with longer speech samples. For instance, at 333 timesteps, DRAM consumes about 10% of the energy. Speech samples longer than 333 timesteps, incur an additional energy penalty for reading activations from DRAM; however, DRAM energy remains a small fraction compared to SRAM and logic. Thus, MASR supports arbitrary length speech samples at negligible performance, area, and energy cost.

VIII. Scalability for End-to-End RNN

As the number of parallel lanes increases, two main performance bottlenecks emerge: vector-vector add (VVAdd) operations and load imbalance. We address these bottlenecks by: (1) parallelizing the VVAdd operations with multi-banked activation SRAMs; and (2) dynamic load balancing for sparse activations. These optimizations improve performance by up to 1.8×, allowing highly parallel designs to achieve high *MAC* utilization while executing with sparse weights and activations.

Parallelizing VVAdd Each time step in RNNs includes two matrix-vector multiplications and a VVAdd (i.e., $W_x x^t + W_h h^{t-1}$). Although the matrix-vector multiplications are the core kernels, Figure 12 (left) shows that with a single activation SRAM bank, the LANESx1024 design's *MAC* utilization is only 27%, while the largest fraction of cycles (35%) devoted to VVAdd. This is due to bandwidth limitations of the activation

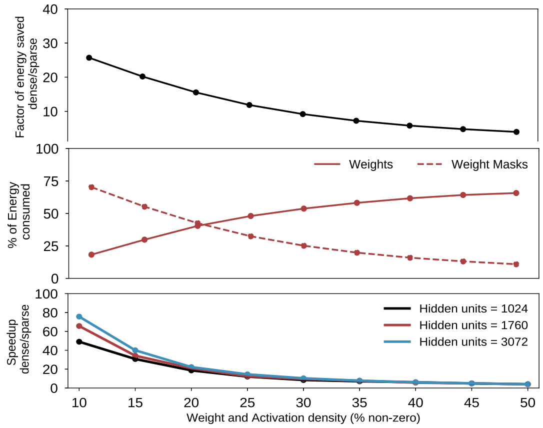


Fig. 13: Energy savings (**top**), energy consumption (**middle**), and performance impact of scaling to larger RNNs (**bottom**) as density in weights and activations varies on LANESx256.

SRAMs. Recall that the word width of the activation SRAM is 60 bits, limiting VVAdd operations to 6 per cycle. Partitioning the compact activation memory into smaller banks enables parallelizing the computation. Partitioning the memory into 8 banks decreases the fraction of cycles spent on VVAdd operations from 35% to 6%, and increases the *MAC* utilization from 27% to 39% at a negligible area penalty.

A. Dynamic Activation Load Balancing

After parallelizing the VVAdd operation, the next bottleneck to scaling performance is load imbalance, a result of irregular sparsity. Previous work uses load balance-aware pruning, where the network is pruned during training such that each *MAC* gets the same number of non-zero weights [47]. This static method does not work for dynamic activation sparsity. Moreover, the main source of load imbalance in RNNs is the uneven distribution of non-zero activations across PEs. To address this dynamic load imbalance, we first trade off stall cycles for idle cycles by increasing the depth of back end queues. We then propose a low-cost solution to dynamically redistribute work to idle lanes, which improves the *MAC* utilization and performance of the LANESx1024 architecture by 1.3×.

Figure 12 (center) illustrates the trade-off between stall and idle cycles as the back end accumulator queue depth increases. Stalls are caused by back pressure from the accumulator. Idle cycles are caused by some lanes computing their partial outputs faster than others. With a queue depth of one, lanes spend 28% of their time stalled, and 22% of their time in the idle state. By increasing the depth of the queues to 8, the fraction of stall cycles falls to 9% whereas the fraction of idle cycles climbs to 42%. $0.3mmm^2$. This comes at a negligible area penalty of 0.3mm^2 for the largest LANESx1024 design.

Balancing non-zero activations The high percentage of idle cycles suggests redistributing work to lanes that finish early, by balancing load across both horizontal lanes and vertical lanes. With horizontal load balancing, work is distributed to lanes within the same PE. Since all lanes within a PE process the same activations, horizontal load balancing requires storing additional copies of compact weights for neighboring lanes. In practice, we only duplicate about 10% of the weights. Vertical load balancing distributes work to lanes across vertical PEs. This involves duplicating not only weights but also activations. Given activations are stored in local registers, the cost of duplicating them is negligible. Duplicating weights and activations to enable load balancing also eliminates the need for complex crossbar inter-connects that often limit efficiency for sparse DNN accelerators at scale.

Hardware utilization Figure 12 (right) illustrates the impact of each optimization on the performance of LANESx64, LANESx256, and LANESx1024. To highlight how well each design variant parallelizes sparse RNNs, we normalize the performance of each to its theoretical speedup over serial execution. Vertical load balancing outperforms horizontal load balancing, because redistributing work across PEs balances the number of non-zero activations, the main source of load imbalance. Moreover, LANESx64, LANESx256, and LANESx1024 designs achieve 90%, 80%, and 50% utilization.

B. Scaling RNN Size and Sparsity

Recent advances in the machine learning community allow training sparser networks without sacrificing accuracy [57], [66], [67]. This suggests further performance and energy improvements may be possible with even higher sparsity. To study the robustness and scalability of MASR, we artificially scale weight and activation non-zero ratios, using synthetic RNN benchmarks, for the energy-optimal LANESx256 design.

Figure 13(top, middle) plots the energy savings and consumption as the non-zero ratio in weights and activations scales from 10% to 50%. As the energy saved from sparse execution depends on the non-zero ratio (not model size), we consider RNNs with 3072 hidden states. MASR's energy efficiency improves with greater sparsity, a result of fewer memory accesses. For example, the energy savings at non-zero ratios of 25% and 10% are 12× and 26× respectively. At lower non-zero ratios, sparse encoding overheads limit savings as weight masks dominate energy consumption.

Figure 13(bottom) plots the impact of sparsity on performance across a range of network sizes. As expected, speedup from sparse execution improves with greater sparsity. For RNNs with 3072 hidden units, sparse execution yields a 14.4× and 76× speedup with 25% and 10% non-zeros, respectively. Finally, we find that performance improvements of sparse execution scale better for larger models. For instance, with 10% non-zeros, the speedup is 49× for the RNNs with 1024 hidden unit. This due to better load balancing in larger models. Thus, we expect MASR's architecture to scale well with larger ASR RNNs and advanced pruning techniques are applied.

C. Hardware Implementation

Results shown thus far are based on cycle-level C++ simulations with power models derived from synthesized RTL. A PE of a LANESx32 RTL was placed-and-routed, as shown in Figure 14, using a commercial 16nm FinFET standard cell

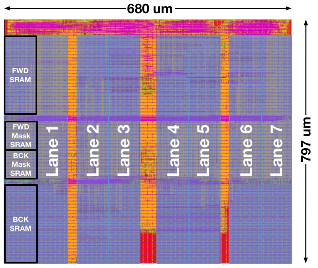


Fig. 14: MASR LANESx32 placed-and-routed layout

library and memory compiler. We validate our simulation results within 10% power and 12% area and find negligible difference in performance. A fabricated SoC based on LANESx32 design has been received from fabrication.

IX. Discussion

This section provides a quantitative comparison between the MASR accelerator and two other accelerators, shown in Table I, for sparse neural networks with different design objectives: ESE [47] (weight sparsity) and EIE [8] (both weight and activation sparsity). Based on each accelerator's memory access patterns, we accumulate the cost of the weights memory, activations memory, and sparse indexing. For fair comparison, each design is implemented as a specialized ASIC with the same 16nm FinFET process and the same optimized RNN, see Table III.

Performance Assuming the same weight sparsity, activation sparsity, and number of parallel *MACs/PEs*, hardware utilization determines performance differences between the accelerators. The LANESx256 design for MASR demonstrates an 80% utilization, compared to 50% in EIE [8]. This is a result of ensuring no wasted work with the binary mask sparse encoding and re-distributing sparse activations for dynamic load balancing in MASR. For instance, CSR adds superfluous non-zero values (up to 40% wasted work) and does not account for imbalance in non-zero activations. By exploiting activation sparsity, MASR has 3× higher performance than ESE.

Area Figure 15 (top) compares the area footprints of MASR, ESE, and EIE, all normalized to the area of the smallest MASR design (LANESx32), as the designs scale from 32 to 512 parallel *MAC*s/lanes. The area for ESE and EIE are equivalent, as both store activations densely and weights compactly, using CSR. For smaller architectures, such as those with 32 or 64 parallel *MAC*s/lanes, MASR's area savings come from compactly storing sparse activations.

Area savings are more pronounced as the accelerators scale to higher parallelism due to MASR's lower-overhead sparse encoding mechanism. Storage for row pointers, used in CSR in ESE and EIE, scales with the number of *MAC*s and dominates for accelerators with more than 128 parallel *MAC*s . Instead of explicitly storing the row pointers in memory, MASR computes the addresses for sparse weights and activation in logic (see Section IV-B for details). This consumes a fixed amount of memory regardless of the number of parallel lanes. As a result, MASR has at least 2× smaller on-chip area footprint as accelerators scale beyond 128 *MAC*s/lanes.

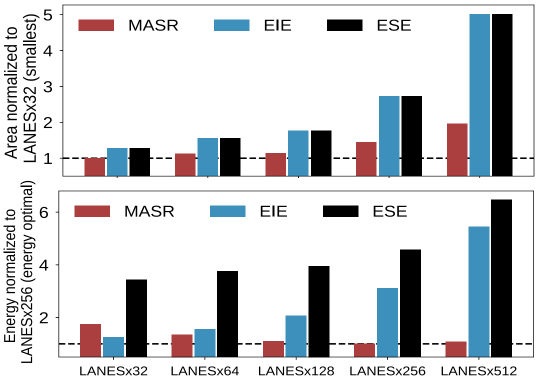


Fig. 15: Area normalized to LANESx32 (top) and Energy normalized to LANESx256 (bottom) consumed by MASR, EIE and ESE as parallel *MAC*s/lanes scale. All accelerators are implemented in same 16nm process technology.

Energy Figure 15 (bottom) compares the energy consumed by MASR, ESE, and EIE, all normalized to the energy of the energy-optimal design (LANESx256) as the designs scale from 32 to 512 parallel *MAC*s/lanes. In addition to the area benefits, MASR consumes 5× and 3× less energy than ESE and EIE, respectively, as it scales beyond 128 *MAC*s/lanes. These energy savings come from MASR's lower overhead sparse encoding mechanism. For each row in the matrix, a PE in ESE and EIE reads two row pointers to determine the first and last non-zero weights. Row pointer accesses scale with the number of parallel, and, like the area overheads, energy consumption is dominated by these row-pointers for more parallel designs. MASR eliminates the cost of reading row pointers by computing the sparse indexing in logic.

X. Conclusion

We present MASR, a novel bidirectional RNN accelerator for on-chip ASR that exploits sparsity in both dynamic activations and static weights, compacts storage of non-zero parameters, and wastes no energy at all on null computations. Compared to a state-of-the-art sparse DNN accelerator [8], MASR improves performance, area, and energy by 1.6×, 3×, and 2×, respectively. MASR's modular architecture provides scalable designs ranging from resource-constrained low-power IoT applications to highly parallel datacenter deployments.

Acknowledgements

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, the NSF under CCF-1704834, and Intel Corporation.

References

[1] Amazon, "What is automatic speech recognition (ASR)?." https://developer.amazon.com/alexa-skills-kit/asr, 2018.

[2] "Google Duplex: An AI system for accomplishing real-world tasks over the phone." https://ai.googleblog.com/2018/05/duplex-ai-system-for-natural-conversation.html, 2018.

[3] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *CoRR*, vol. abs/1512.02595, 2015.

[4] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Fifteenth annual conference of the international speech communication association*, 2014.

[5] H. Sak, A. W. Senior, K. Rao, and F. Beaufays, "Fast and accurate recurrent neural network acoustic models for speech recognition," in *INTERSPEECH*, 2015.

[6] "Google voice search: faster and more accurate." https://ai.googleblog.com/2015/09/google-voice-search-faster-and-more.html, 2015.

[7] D. P. Vassil Panayotov, Guoguo Chen and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," ICASSP'15, 2015.

[8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," *CoRR*, vol. abs/1602.01528, 2016.

[9] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. S. Emer, S. W. Keckler, and W. J. Dally, "SCNN: an accelerator for compressed-sparse convolutional neural networks," *CoRR*, vol. abs/1708.04485, 2017.

[10] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[11] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *ISSCC*, 2016.

[12] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *CoRR*, vol. abs/1503.02531, 2015.

[13] F. Silfa, G. Dot, J.-M. Arnau, and A. Gonzalez, "E-PUR: An energy-efficient processing unit for recurrent neural networks," 2017.

[14] J. Zhu, J. Jiang, X. Chen, and C.-Y. Tsui, "Sparsenn: An energy-efficient neural network accelerator exploiting input and output sparsity," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 241–244, IEEE, 2018.

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 1–12, ACM, 2017.

[16] NVIDIA, "Nvidia deep learning accelerator (NVDLA)," 2018.

[17] K. Hazzelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture*, HPCA '18, 2018.

[18] B. R. G.-Y. W. Robert Adolf, Saketh Rama and D. Brooks, "Fathom: Reference workloads for modern deep learning methods," IISWC'16, 2016.

[19] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, *et al.*, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *arXiv preprint arXiv:1811.09886*, 2018.

[20] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[21] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Teman, "Dadiannao: A machine-learning supercomputer," in *MICRO*, 2014.

[22] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *ISCA*, 2016.

[23] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *SIGARCH Comput. Archit. News*, vol. 44, pp. 1–13, June 2016.

[24] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "Circnn: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 395–408, ACM, 2017.

[25] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 14–26, IEEE Press, 2016.

[26] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," *CoRR*, vol. abs/1705.01626, 2017.

[27] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 13–26, 2017.

[28] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, "Prediction based execution on deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 752–763, IEEE, 2018.

[29] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 27–39, IEEE Press, 2016.

[30] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[31] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 764–775, IEEE Press, 2018.

[32] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 925–938, ACM, 2019.

[33] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 821–834, ACM, 2019.

[34] T. Jin and S. Hong, "Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 835–847, ACM, 2019.

[35] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, *et al.*, "E-rnn: Design optimization for efficient recurrent neural networks in fpgas," in *2019 IEEE International Symposium on*

High Performance Computer Architecture (HPCA), pp. 69–80, IEEE, 2019.

[36] J. F. K. O. M. Papamichael, T. M. M. Liu, D. L. S. A. M. Haselman, L. A. M. Ghandi, S. H. P. P. A. Sapek, and G. W. L. Woods, "A configurable cloud-scale dnn processor for real-time ai,"

[37] X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu, "Towards memory friendly long-short term memory networks (lstms) on mobile gpus," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 162–174, IEEE, 2018.

[38] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *ACM SIGPLAN Notices*, vol. 53, pp. 461–475, ACM, 2018.

[39] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou, "Astra: Exploiting predictability to optimize deep learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 909–923, ACM, 2019.

[40] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "Tangram: Optimized coarse-grained dataflow for scalable nn accelerators," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 807–820, ACM, 2019.

[41] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "Fpga-based accelerator for long short-term memory recurrent neural networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 629–634, Jan 2017.

[42] E. C. Andre Xian Ming Chang, Berin Martini, "Recurrent neural networks hardware implementation on fpga," 2016.

[43] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "Deltarnn: A power-efficient recurrent neural network accelerator," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, (New York, NY, USA), pp. 21–30, ACM, 2018.

[44] R. Yazdani, J.-M. Arnau, and A. González, "Unfold: A memory-efficient speech recognizer using on-the-fly wfst composition," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 69–81, ACM, 2017.

[45] A. Y. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep speech: Scaling up end-to-end speech recognition," *CoRR*, vol. abs/1412.5567, 2014.

[46] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[47] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "ESE: efficient speech recognition engine with compressed LSTM on FPGA," *CoRR*, vol. abs/1612.00694, 2016.

[48] A.-R. Mohamed, G. E. Dahl, and G. Hinton, "Acoustic modeling using deep belief networks," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 1, pp. 14–22, 2012.

[49] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[50] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[51] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[52] "A broad ml benchmark suite for measuring performance of ml software frameworks, ml hardware accelerators, and ml cloud platforms." https://mlperf.org/, 2018.

[53] "Pytorch." http://pytorch.org/, 2017.

[54] "deepspeech.pytorch." https://github.com/SeanNaren/deepspeech.pytorch, 2018.

[55] A. Graves, S. Fernandez, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks," *ICML'2006*, 2006.

[56] R. Yazdani, M. Riera, J.-M. Arnau, and A. Gonzalez, "The dark side of dnn pruning," in *ISCA*, 2018.

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

[58] B. Reagen, U. Gupta, R. Adolf, M. M. Mitzenmacher, A. M. Rush, G.-Y. Wei, and D. Brooks, "Weightless: Lossy weight encoding for deep neural network compression," *arXiv preprint arXiv:1711.04686*, 2017.

[59] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," *arXiv preprint arXiv:1704.05119*, 2017.

[60] E. Battenberg, J. Chen, R. Child, A. Coates, Y. G. Y. Li, H. Liu, S. Satheesh, A. Sriram, and Z. Zhu, "Exploring neural transducers for end-to-end speech recognition," in *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pp. 206–213, IEEE, 2017.

[61] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang, *et al.*, "Streaming end-to-end speech recognition for mobile devices," *arXiv preprint arXiv:1811.06621*, 2018.

[62] C.-C. Chiu and C. Raffel, "Monotonic chunkwise attention," *arXiv preprint arXiv:1712.05382*, 2017.

[63] "Deepbench." https://github.com/baidu-research/DeepBench, 2018.

[64] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "SnaPEA : Predictive early activation for reducing computation in deep convolutional neural networks," in *Proceedings of the 45th International Symposium on Computer Architecture*, 2018.

[65] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, , and K. Goossens, "Drampower: Open-source dram power and energy estimation tool." http://www.drampower.info.

[66] K. Ullrich, E. Meeds, and M. Welling, "Soft weight-sharing for neural network compression," *ICLR'2017*, vol. abs/1702.04008, 2017.

[67] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," *NIPS'2016*, vol. abs/1608.04493, 2016.