PeakEngine: A Deterministic On-the-Fly Pruning Neural Network Accelerator for Hearing Instruments

Jelcicova, Zuzana; Kasapaki, Evangelia; Andersson, Oskar; Sparso, Jens

Published in:
IEEE Transactions on Very Large Scale Integration (VLSI) Systems

Link to article, DOI:
10.1109/TVLSI.2023.3300910

Publication date:
2023

Document Version
Peer reviewed version

PeakEngine: A Deterministic On-the-Fly Pruning Neural Network Accelerator for Hearing Instruments

Zuzana Jelčicová, Evangelia Kasapaki, Oskar Andersson, *Member, IEEE*, and Jens Sparsø, *Member, IEEE*

***Abstract*— Recurrent neural networks (RNNs) are well-suited for sequential tasks such as speech enhancement (SE). However, their performance comes with high-computational complexity and latency. This impedes their deployment to battery-powered and resource-constrained hearing instruments (HIs) that need to operate for 16–18 h daily at only a few milliwatts (mW). In this article, we introduce *PeakEngine*, a configurable ASIC accelerator that decreases the amount of computation and memory accesses, and thus latency, in a gated recurrent unit (GRU) by means of adaptive inference. The reduction is achieved by on-the-fly pruning that selects the top K elements based on magnitudes of delta changes across timesteps from both input and hidden state sequences. Since K is constant, it results in a deterministic execution time. *PeakEngine* is synthesized in a 22-nm CMOS process, and the simulations show that it dissipates 11.83 μJ per inference for the baseline (unpruned) network and only 4.14–5.04 μJ for the pruned networks, with maximum acceptable degradation to no degradation in the improvement in audio quality and intelligibility. Moreover, the inference is on average sped up 2.2–2.97×, hence meeting the real-time requirements imposed by a HI application. To the best of our knowledge, *PeakEngine* is the first ASIC accelerator for deterministic and dynamic pruning in RNNs targeting HIs and SE.**

***Index Terms*— Deterministic execution time, dynamic pruning, hardware accelerator, hearing instruments (HIs), min-heap, recurrent neural networks (RNNs), speech enhancement (SE), top K.**

I. Introduction

UNDERSTANDING speech-in-noise is critically important yet one of the biggest problems for hearing instrument (HI) users [1], [2], [3]. *Speech enhancement* (SE) addresses this issue by attenuating the background noise, thus improving speech quality and/or intelligibility. Traditional methods for SE are very advanced, but most of them either assume that speech and noise are uncorrelated or heavily rely on *domain knowledge* [4] that is unknown in advance and hence must be estimated, such as the a priori signal-to-noise ratio (SNR). This is not the case for deep neural networks (DNNs) that learn directly from the data and have already demonstrated their superior performance for SE over the traditional methods [5], [6], [7], [8].

Recurrent neural networks (RNNs) are an attractive solution for SE due to their powerful capabilities of processing sequential data. This is possible thanks to their feedback connection that is shared between timesteps. The feedback connection enables RNNs to retain information, making them superior for applications that use data with temporal structures, such as video processing [9], natural language processing [10], translations [11], and speech recognition [12]. The two most typical variants of RNNs are a long short-term memory (LSTM) [13] and a gated recurrent unit (GRU) [14].

At the same time, RNNs suffer from high-computational complexity and latency. Since the sequences are dependent on each other they cannot be processed simultaneously, which prevents exploiting parallelism. Moreover, the dominant operation in RNNs is a matrix–vector multiplication that grows quadratically with the number of hidden units, which consequently increases the amount of memory accesses, latency, and power consumption. As shown in [15], memory accesses dominate over arithmetic operations in terms of energy dissipation, and thus become the biggest bottleneck in RNNs. All these issues impede the deployment and execution of RNNs in low-power and resource-constrained HIs that operate with a few milliwatts (mW) and require audio latency below 30 ms [16], [17] to ensure optimal sound quality and comfort. Moreover, they need to last around 16–18 h every day [18] on a miniature-sized battery. Therefore, to enable RNN inference in HIs, it is crucial to reduce multiply-accumulates (MACs) and corresponding memory accesses that are the main sources of power consumption and latency.

A typical approach to decrease the number of memory accesses (and consequently MACs) is *static pruning*. Static pruning compresses the size of the original model by permanently removing weights that contribute very little to the final outcome. A myriad of pruning methods have been proposed throughout the years, as surveyed in [19] and [20]. Some of the RNN approaches include magnitude-based and load-balance-aware pruning of weights using an empirical threshold [21],

Manuscript received 22 January 2023; revised 30 June 2023; accepted 24 July 2023. This work was supported in part by the Innovation Fund Denmark under Grant 9065-00139B. *(Corresponding author: Zuzana Jelčicová.)*

Zuzana Jelčicová is with Demant A/S, 2765 Smørum, Denmark, and also with the Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kongens Lyngby, Denmark (e-mail: zuje@demant.com).

Evangelia Kasapaki and Oskar Andersson are with Demant A/S, 2765 Smørum, Denmark (e-mail: evka@demant.com; oand@demant.com).

Jens Sparsø is with the Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kongens Lyngby, Denmark (e-mail: jspa@dtu.dk).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2023.3300910.

Digital Object Identifier 10.1109/TVLSI.2023.3300910

structured pruning with a learnable threshold [22], and iterative compression of randomly selected weight blocks [23]. Static pruning of RNNs is generally challenging as a recurrent unit is shared across all the timesteps. Compressing the unit thus impacts all the steps in the sequence. Permanently discarding weights destroys the original network structure which may lead to decreased capability, representation power, and efficiency of a model [20], [24]. It has also been shown that small SE networks have difficulties learning the necessary relationship between the noisy features and the target SNRs [25].

Dynamic pruning, on the other hand, is a data-driven approach, where computations are conditioned on the input at runtime. Dynamic pruning on its own does not reduce the model size. However, it counterbalances this issue with several other advantages [24] that are missing in static models such as the following:

1) *Efficiency:* Allocating computations on demand at runtime.
2) *Representation power:* Identifying "easy" and "hard" samples and hence computational redundancy.
3) *Adaptiveness:* Achieving a desired trade-off between accuracy and efficiency at runtime.
4) *Generality:* Seamlessly adapting to a wide range of applications.

An example of dynamic pruning includes an accelerator in [26] that prunes attention layers in transformer neural networks using the top cumulative importance scores. Furthermore, *temporal sparsity* based on a threshold is exploited in several works. These include [27] that skips state updates using also a skip-criterion, [28] that prunes hidden state vectors in LSTMs, and [29] implemented as GRU [30] and LSTM [31] accelerators, where dense state vectors are substituted with their sparse delta versions obtained as a temporal difference across two adjacent timesteps. However, the actual computation time in all of these state-of-the-art threshold-based dynamic pruning approaches is *unpredictable*, which is an issue for real-time systems like HIs.

To the best of our knowledge, none of the existing accelerators supports dynamic pruning of RNNs that results in deterministic inference, targeting HIs and relevant use cases such as SE. Our accelerator fills this gap and offers adaptive yet computationally predictable inference that is absent in the state-of-the-art RNN accelerators. The main contributions of this work are as follows.

1) *Min-heap engine*, a low-area and energy hardware unit, that selects the top K elements in a stream of N elements, where $N > K$. The engine is used to support our deterministic PeakRNN [32] pruning algorithm. Moreover, its application is versatile and not limited to neural networks only.
2) *PeakEngine*, the first ASIC accelerator for HIs that enables deterministic on-the-fly pruning of RNNs. *PeakEngine* is *configurable* and *portable* thanks to its standard interfaces, and it encompasses the *Min-heap engine*.
3) A thorough investigation of *PeakEngine* for different K values with regard to saved energy and reduced latency.
4) A bit-accurate *software framework* for parameter space exploration of different Q formats, wordlengths, and K values for GRU-based and fully connected (FC) neural networks.

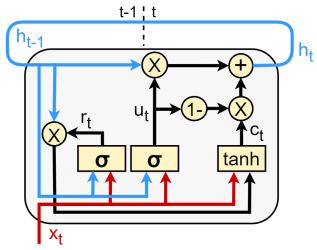

Fig. 1. Illustration of a GRU. The multiplications with weight matrices and summations of the partial dot products are excluded for clarity.

The rest of the article is structured as follows. Section II provides the necessary background for the PeakRNN algorithm, while Section III describes the algorithm itself. Section IV introduces an emulated HI setup used for experiments with SE. Information about the DNN architecture, datasets, and training is stated in Section V. Section VI details the *PeakEngine* design, and Section VII talks about the software framework. The experimental setup is described in Section VIII. Section IX presents and discusses the results as well as comparisons to state-of-the-art works. Finally, Section X concludes the article.

II. Background

This section presents the original GRU [14] and DeltaGRU [29] algorithms, on top of which our modified version called PeakGRU [32] is built.

A. GRU Algorithm

A GRU (see Fig. 1) has a feedback connection called a *hidden state* $h(t)$ that maintains both short- and long-term dependencies. It is controlled by two internal *gate* mechanisms, *reset* $r(t)$ and *update* $u(t)$ gate, that regulate the flow of information in the unit using sigmoid (σ) activation

$$r(t) = \sigma(W_{xr}x(t) + W_{hr}h(t-1) + b_r) \quad (1)$$

$$u(t) = \sigma(W_{xu}x(t) + W_{hu}h(t-1) + b_u). \quad (2)$$

A GRU processes two types of inputs: 1) an input sequence $x(t)$ for the current timestep and 2) a sequence of previous hidden states $h(t-1)$. The sequences are multiplied with their respective *weight matrices* W and summed with *bias values* b. The reset gate then determines how much of the past information should be forgotten. The relevant past information, on the other hand, is stored in a *candidate state* $c(t)$ that applies hyperbolic tangent (tanh) activation function

$$c(t) = \tanh(W_{xc}x(t) + r(t) \odot (W_{hc}h(t-1)) + b_c). \quad (3)$$

The update gate decides on the importance of both the past information $h(t-1)$ and the new information $c(t)$

$$h(t) = u(t) \odot h(t-1) + (1 - u(t)) \odot c(t). \quad (4)$$

Finally, a new hidden state $h(t)$ that also serves as an output for the current timestep is generated. Based on (1)–(3), the

number of matrix–vector multiplications (and corresponding memory fetches) in a GRU can be expressed as

$$3 \times (XH + H^2) \tag{5}$$

where X and H correspond to the dimensions of input sequences $x(t)$ and $h(t-1)$, respectively. The computational complexity grows quadratically with respect to the $h(t-1)$ that consequently increases the number of memory accesses and hence power.

B. DeltaGRU Algorithm

DeltaGRU, also called *Delta Networks* [29], dynamically reduces the number of MAC operations and memory fetches by transforming a dense matrix–vector multiplication into a highly sparse matrix–vector multiplication in every timestep

$$\hat{x}(t) = \begin{cases} x(t), & \text{if } |x(t) - \hat{x}(t-1)| > \Theta \\ \hat{x}(t-1), & \text{otherwise} \end{cases} \tag{6}$$

$$\hat{h}(t-1) = \begin{cases} h(t-1), & \text{if } |h(t-1) - \hat{h}(t-2)| > \Theta \\ \hat{h}(t-2), & \text{otherwise} \end{cases} \tag{7}$$

$$\Delta x(t) = \begin{cases} x(t) - \hat{x}(t-1), & \text{if } |x(t) - \hat{x}(t-1)| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

$$\Delta h(t-1) = \begin{cases} h(t-1) - \hat{h}(t-2), & \text{if } |h(t-1) - \hat{h}(t-2)| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

$$M_r(t) = W_{xr}\Delta x(t) + W_{hr}\Delta h(t-1) + M_r(t-1) \tag{10}$$

$$M_u(t) = W_{xu}\Delta x(t) + W_{hu}\Delta h(t-1) + M_u(t-1) \tag{11}$$

$$M_{xc}(t) = W_{xc}\Delta x(t) + M_{xc}(t-1) \tag{12}$$

$$M_{hc}(t) = W_{hc}\Delta h(t-1) + M_{hc}(t-1) \tag{13}$$

$$r(t) = \sigma[M_r(t)] \tag{14}$$

$$u(t) = \sigma[M_u(t)] \tag{15}$$

$$c(t) = \tanh[M_{xc}(t) + r(t) \odot M_{hc}(t)] \tag{16}$$

$$h(t) = u(t) \odot h(t-1) + (1 - u(t)) \odot c(t). \tag{17}$$

The dense-to-sparse transformation is achieved by applying a threshold Θ on the magnitude of input change across adjacent timesteps; see (6)–(9). The magnitude of change, i.e., the absolute value of the *delta change*, is calculated by subtracting previously cached inputs, $\hat{x}(t-1)$ and $\hat{h}(t-2)$, from the current inputs, $x(t)$ and $h(t-1)$. The initial value of *hat states* $\hat{x}(t-1)$ and $\hat{h}(t-2)$ is 0. The magnitudes above Θ are then selected, and their actual subtraction results are used in MACs as $\Delta x(t)$ and $\Delta h(t-1)$, as shown in (10)–(13). The hat state updates are subsequently only performed for the magnitudes above Θ. *DeltaGRU* needs additional *delta memory states* M to track the delta changes across timesteps. Finally, a new hidden state $h(t)$ [see (17)] for the current timestep is generated the same way as shown in (4) in Section II-A.

The hat and memory states inflict memory and computational overhead. However, this is compensated with substantial reduction of MACs and memory fetches as demonstrated with *PeakGRU* in Section IX.

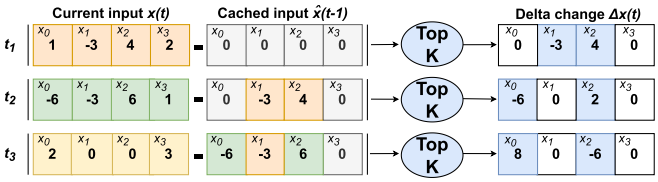

Fig. 2. Example of calculating a *sparse delta vector* along with the hat states across three timesteps (t_1–t_3) for $x(t)$ using PeakGRU. $X = 4$, $K_x = 2$, and the black vertical lines around the subtraction represent absolute value.

III. PeakGRU Algorithm

Our PeakGRU algorithm builds on the top of DeltaGRU. This section describes the differences between the two approaches, and explains how PeakGRU is efficiently realized in hardware.

A. Top-K Magnitudes

PeakGRU and DeltaGRU share the underlying computations in (6)–(17). The difference between the two algorithms is in the method of how the Δ elements are obtained in (8)–(9). While DeltaGRU uses a threshold-based approach, PeakGRU selects the top K_x and the top K_h subsets of elements from $x(t)$ and $h(t-1)$ sequences, respectively

$$\hat{x}(t) = \begin{cases} x(t), & \text{if } |x(t) - \hat{x}(t-1)| \text{ among } K_x \\ \hat{x}(t-1), & \text{otherwise} \end{cases} \tag{18}$$

$$\hat{h}(t-1) = \begin{cases} h(t-1), & \text{if } |h(t-1) - \hat{h}(t-2)| \text{ among } K_h \\ \hat{h}(t-2), & \text{otherwise} \end{cases} \tag{19}$$

$$\Delta x(t) = \begin{cases} x(t) - \hat{x}(t-1), & \text{if } |x(t) - \hat{x}(t-1)| \text{ among } K_x \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

$$\Delta h(t-1) = \begin{cases} h(t-1) - \hat{h}(t-2), & \text{if } |h(t-1) - \hat{h}(t-2)| \\ & \text{among } K_h \\ 0, & \text{otherwise.} \end{cases} \tag{21}$$

This modification is illustrated in Fig. 2, where $\Delta x(t)$ is calculated across three timesteps, along with the propagation of $\hat{x}(t-1)$. The matching colors in $x(t)$ and $\hat{x}(t)$ vectors denote the updates of $\hat{x}(t-1)$ with $x(t)$. The same visualization applies to $\Delta h(t-1)$ and $\hat{h}(t-2)$.

Since the number of elements to process is known in advance, the actual computation time in PeakGRU is *deterministic*. This is an important characteristic for real-time and resource-constrained embedded devices, where the worst case execution guarantees are imperative. Moreover, since the algorithm selects elements based on the range of inputs and not a threshold, it is robust to variations in data compared to threshold-based approaches such as DeltaGRU [29]. Therefore, PeakGRU prevents significant fluctuations in the number of nonzero values from one timestep to another that are inherent in threshold-based methods.

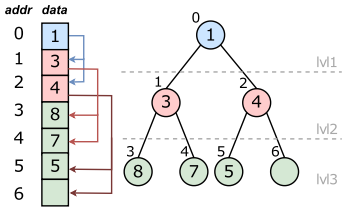

Fig. 3. Example of a *min-heap* with $K = 7$ and its memory implementation as a simple array, where the last level is not fully filled (index 6).

B. Top-K Selection

Processing only the top K elements offers numerous advantages but it also imposes a challenge of how the elements should be selected. For instance, sorting the N elements first and selecting a top K subset afterwards would cause unnecessary computational and memory overhead since the order of elements is irrelevant. Instead, a *binary heap* [33] can be used for efficient selection, imposing: 1) minimal storage requirements $O(K)$ when implemented as a simple array and 2) low worst-time computational complexity $O(N \log K)$.

A binary heap is a complete binary tree, where all the levels ($\lceil \log_2(K+1) \rceil$) are fully filled, except possibly the deepest one, as shown in Fig. 3. The elements are inserted into the heap from left to right and level by level, with the worst-time complexity of $O(\log K)$ per element, which corresponds to a swap across all the heap levels. The nodes in the deepest level (leaves) start at index $\lfloor K/2 \rfloor$. To support PeakGRU, we employ a *min-heap* binary tree, where all the nodes within a level are numerically greater than or equal to their parent nodes in the level above. The parent, left, and right child nodes in an array implementation have indices $\lfloor (i-1)/2 \rfloor$, $(2\times i)+1$, and $(2\times i)+2$, respectively, where i is the index of the current node.

In our design, the very first data are directly inserted into the min-heap. Subsequent data become a leaf and is compared against its parent node. It is traversed up the tree (swapped) until numerically smaller than the parent, or the *root*, the smallest element in the min-heap, is reached. Once the min-heap is full, the data are inserted from the top. The first comparison is hence always done against the *root*. If the new data are smaller than/equal to the *root*, data are immediately skipped, and no more comparisons are needed. Otherwise, data replace the *root* and are swapped with one of their child nodes until they are greater or become a leaf node.

All these operations are executed by the proposed *Min-heap engine* that is used to support the PeakGRU algorithm. However, the engine design is not tied to the algorithm or neural networks, and it could be applied in many other contexts such as priority queues [34] and data compression [35].

C. Top-K Storage

An on-chip implementation of the min-heap memories can be realized with either: 1) a standard-cell based memory (SCM), i.e., an addressable array of flip-flops or latches or 2) an SRAM macro. Since the memory requirements for our min-heaps are small (3.25 kb each) and below the area break-even point with SRAMs [36], [37], SCMs (with latches) are selected for storing the top K_x and K_h values. Furthermore, we use three-port (2R1W) SCMs to parallelize min-heap computations (fetching of the child elements). Our SCM design is based on [38].

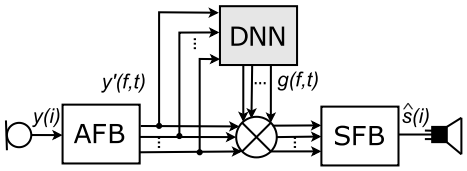

Fig. 4. Overview of the gain-based SE system in a HI used for the experiments, where the DNN generates postfilter gains $g(f,t)$.

IV. HI Application

Deterministic on-the-fly pruning supported in *PeakEngine* is demonstrated in a SE task. The sections below present details about the emulated HI setup and objective metrics used for evaluating the performance of the system. The setup is based on our previous work on the PeakGRU [32] algorithm.

A. SE System

A DNN architecture described in Section V-A is used in a *gain-based single-microphone SE* system that simulates the inner parts of a HI. The system is illustrated in Fig. 4, where a DNN substitutes a traditional signal processing-based module for generating postfilter gain values $g(f,t)$, with f and t corresponding to a frequency bin and time-frame, respectively. The input for a DNN is generated by preprocessing samples (i) of a noisy 20-kHz single-microphone signal $y(i)$ in an analysis filter bank (AFB), where the noisy signal is clean speech $s(i)$ corrupted with noise $n(i)$. The AFB applies a 1024-point fast Fourier transform (FFT) and a square-root Hanning window, downsampling the time-domain microphone signal to 40 Hz and producing a time-frequency representation $y'(f,t)$. In our specific scenario, the result is a new frame of 512 frequency bin values every 25 ms without overlapping. The $y'(f,t)$ values are passed to a DNN that generates 512 postfilter gain values $g(f,t)$. The gain values are applied on the noisy signal $y'(f,t)$ to obtain an estimate of the clean speech magnitude spectrum. Finally, the synthesis filter bank (SFB) reconstructs the time-domain signal $\hat{s}(i)$ and forwards the result to the speaker.

B. Objective Measures

To evaluate the performance of a DNN under a varying number of the top K elements and full- and reduced-precision, the HI system in Fig. 4 is extended with *postprocessing*. The postprocessing phase saves the enhanced signals from the SFB along with their corresponding clean speech and noise that are used for calculating the following three objective metrics.

1) *Perceptual Evaluation of Speech Quality (PESQ)*: Evaluates the *quality* of noisy speech by estimating *mean opinion score* (MOS). MOS is judged using a discrete scale of 1 (bad) to 5 (excellent). The result is an average of these ratings. *Mean opinion score-listening quality objective* (MOS-LQO) [39] is used as a unit.

2) *Short-Time Objective Intelligibility (STOI)*: Evaluates the *intelligibility* of noisy speech (ability to recognize word, syllables, etc.), and it thus produces a scalar value in a range of 0 to 1, where 1 corresponds to fully intelligible speech.
3) *Signal-to-Noise Ratio (SNR)*: Compares the level of a desired signal to the level of background noise, expressed in dBs.

PESQ and STOI approximate human ranking and thus replace time-consuming and expensive listening tests [40].

V. DNN for SE

After introducing the HI and SE setup, we can now focus on the DNN itself, the selected architecture, datasets, as well as training procedure. The architecture and datasets were also used in our original algorithmic study [32].

A. DNN Architecture

The neural network used in the experiments consists of three layers. The first and the last layer are FC, and the hidden layer is a GRU. Each of the layers has 512 output neurons. A nonlinearity is introduced after the first FC layer with the ReLU activation function. The GRU layer is swapped with a PeakGRU layer to evaluate and compare the performance and computational savings of the two methods against each other. A GRU accounts for a significant part (75%) of the vectorized operations, i.e., MACs and memory accesses, while the remaining 25% accounts for both of the two FC layers.

B. Dataset

A noisy DNN input, y, is created by combining 30-s segments of clean speech, s, with noise, n, i.e., $y = s + n$. The resulting noisy speech has up to three speakers with a silence gap of approximately 300 ms and up to 30% overlap to mimic a regular conversation. The speech dataset is composed of VCTK Corpus [41] and Akustiske Database for Dansk [42]. Thirteen noise environments were selected to represent a wide variety of the typical daily situations. These were obtained from EigenScape [43] (Beach, Busy Street, Park, Pedestrian Zone, Quiet Street, Shopping Centre, Train Station, Woodland), and Demant's database (Bar, Cafe, Canteen, Car, Office). Additionally, two stationary types of noise, pink and white, were simulated. The 25-h noisy speech consisting of both left and right channel data is divided into training (19.5 h), validation (2.7 h), and test (2.7 h) subsets.

C. Training Target and Hardware-Aware Training

The DNN learns to match its output against a linear *Ideal Ratio Mask* (IRM) target. IRM represents an ideal scenario, i.e., when speech and noise are perfectly separated. The IRM outputs a continuous gain value between $[0, 1]$ that is computed as a ratio between the magnitude of a clean speech signal and a sum of the magnitudes of the clean and noise signal

$$\text{IRM} = \left(\frac{|s(f,t)|}{|s(f,t)| + |n(f,t)|} \right) \tag{22}$$

where $s(f,t)$ and $n(f,t)$ are the time-frequency representations of the clean speech and noise, respectively. The error between the IRM and the postfilter gain values estimated by the DNN is obtained with the mean-squared error loss function. The baseline DNN with a GRU layer, i.e., when all computations are performed, was trained in TensorFlow using 32-bit floating point with a batch size of 128, where each input sequence was 2.5 s long (100 samples). The trained parameters were then transferred to the PeakGRU network for inference to: 1) replace computationally demanding and tedious training from scratch; 2) enable a fair comparison of both methods; and last but not least 3) simulate a real-world scenario, where new weights would not be transferred to a HI whenever a different number of K values should be processed. Instead, a single, robust DNN model capable of executing both a full and a pruned model should be used while still delivering sufficient performance in terms of objective measures. Naturally, retraining the model for a specific number of K values further improves its performance, as also demonstrated in our previous work [32].

To prepare a DNN for inference in a hardware environment with limited precision and computational resources, several hardware-aware constraints were incorporated into the training. First, the weights are limited to the $[-1, 1]$ range. Second, a challenge is represented by the activation functions. For instance, the output of ReLU can theoretically be within a range of $[0, +\infty]$. While a 32-bit floating point is sufficient to handle large numbers, a fixed-point representation needs too many bits on the integer part. Therefore, we use Capped ReLU that applies a maximum upper bound (in our case 6). Furthermore, computing sigmoid (σ) and hyperbolic tangent (tanh) activation functions would be expensive due to their exponential terms

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{23}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{24}$$

Typical methods for computing these functions are based on piecewise linear/nonlinear approximations [44], lookup tables (LUTs) [45], and hybrid techniques [46]. Although LUTs outperform the other methods in terms of speed as they need the fewest computations, they require additional area, which, depending on the necessary precision, might grow into a significant overhead. Therefore, we train the DNN and run inference with approximated versions called *hard sigmoid* and *hard tanh* that are expressed as

$$\sigma(x) = \begin{cases} 0, & \text{if } x <= -2.5 \\ 1, & \text{if } x >= 2.5 \\ 0.2 \times x + 0.5, & \text{otherwise} \end{cases} \tag{25}$$

$$\tanh(x) = \begin{cases} -1, & \text{if } x <= -1.25 \\ 1, & \text{if } x >= 1.25 \\ 0.75 \times x, & \text{otherwise.} \end{cases} \tag{26}$$

In terms of hardware, each of these approximations requires only one multiplier and two comparators [and an adder for $\sigma(x)$]. Moreover, whenever the input value exceeds the upper

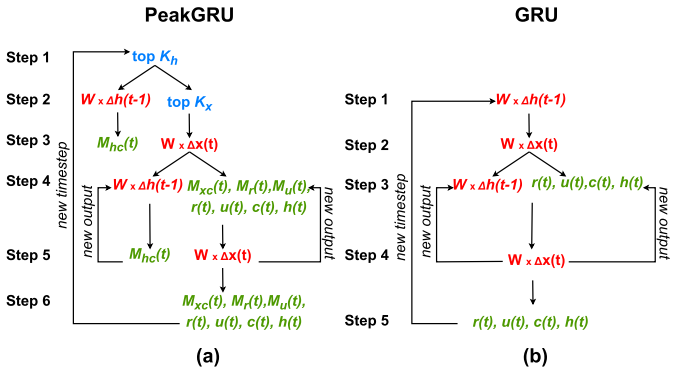

Fig. 5. Potential concurrence that can be exploited in (a) PeakGRU and (b) GRU, where blue, red, and green correspond to operations in the *Peak Unit*, the *Mac Unit*, and the *Activation Unit*, respectively.

and lower bounds, the output values are immediately saturated without performing any further computations.

VI. *PeakEngine* Design

This section presents architectural design choices (Section VI-A) and describes the design of *PeakEngine* (Section VI-B) along with implementation details (Section VI-C).

A. Architectural Design Choices

PeakEngine is designed to support both dense and dynamically pruned GRU layers with a focus on low energy, resource sharing, and configurability to make DNN inference viable in HIs. Each of these points is further described below.

1) Opportunities for Parallelism: The architecture of *PeakEngine* is tailored for the needs of GRU and PeakGRU. It is derived from (10)–(21) that can be grouped based on functionality, suggesting three core units in the system: 1) *Peak Unit* - selecting the top K elements (18)–(21); 2) *Mac Unit* - computing dot products (10)–(13); and 3) *Activation Unit* - producing M states and output activations (14)–(17). Further analysis of the equations shows intrinsic parallelism and dependencies between these units, also illustrated in Fig. 5(a).

PeakGRU computation begins with finding the K_h subset (Step 1). Thereafter, a dot product with $\Delta h(t-1)$ (and weights) is obtained, while the selection of the top K_x is executed in parallel (Step 2). Once dot products with all $\Delta h(t-1)$ for the first neuron are produced, the operations for an output activation start [loading previous memory states $M_{hc}(t-1)$]. Computations of dot products with $\Delta x(t)$ commence as well, assuming that the K_x subset is available (Step 3). Completing a dot product with all $\Delta x(t)$ enables computation of $M_r(t)$, $M_u(t)$, and $M_{xc}(t)$ memory states, and consequently $r(t)$, $u(t)$, and $c(t)$ terms, along with the first output activation $h(t)$. Concurrently, $\Delta h(t-1)$ dot product computations for the next neuron begin (Step 4). Steps 4–5 are repeated until all output activations have been computed (Step 6). This means that the current timestep for PeakGRU is completed, and the system will start again from Step 1 in the next timestep. The steps are almost identical for the baseline GRU in Fig. 5(b). The main difference is the absence of the K subsets, which reduces the flow by one step. Also, no M states are produced.

The computations in the *Peak Unit* and the *Mac Unit* can be further refined. While the insertion into the min-heap is carried out, delta changes for the subsequent neurons are calculated. Such optimization minimizes idle time and maximizes hardware utilization. Similarly, when the K subset stored in the min-heap is used in the *Mac Unit*, the fetched data (specifically an index of each K, i.e., a neuron index) are simultaneously used to update $\hat{x}(t)$ and $\hat{h}(t-1)$ in (18)–(19). This avoids expensive re-fetching of the same data. The hat update operations can thus be coupled with the *Mac Unit*, producing an *UpdateMac Unit* in *PeakEngine*.

This analysis leads to a design of five hierarchically interacting units (FSMDs). Such a co-operation results in optimized execution of GRU-based layers and reduced processing time at negligible hardware costs. The FSMDs are orchestrated by the Main FSM that handles a coarse-grain top control, as shown in Fig. 6(a). The final *PeakEngine* architecture can be seen in Fig. 6(b).

2) Minimizing Energy: High-speed processing is not a driving factor in low-power devices such as HIs. Instead, some HIs may operate at lower clock frequencies and need to finish computations just *"in time."* When the units complete the execution before the timestep is over, they idle and still consume a certain amount of dynamic and static (leakage) power. Moreover, memories that DNNs heavily rely on are often the major source of leakage. Such factors have a significant impact on battery-powered wearables like HIs. Therefore, we employ *clock-gating* and *memory retention* techniques to prevent additional leakage and energy dissipation when *PeakEngine* idles.

3) Resource-Sharing and Reuse: To accomplish the objective of minimizing hardware resources yet completing the execution of DNNs in time, it is necessary to perform additional optimizations on multiple levels. First, we only use single-port SRAMs to minimize area. Second, we employ multiple smaller memories instead of a single shared one, which enables the proposed units to operate concurrently. Most of the memories are shared among multiple units. The scheduling of memory accesses in the five cooperating FSMDs minimizes memory access conflicts and stalling. The memory accesses are time-multiplexed via memory management units (MMUs) to guarantee that only one unit accesses any memory at any given point in time. Finally, our small *Mac* unit represents a tradeoff between area and computations with a focus on minimum hardware resources, while still delivering parallelized dot product computations.

4) Configurability: It is essential to enable the execution of different DNN configurations (type of layers/activation functions, number of neurons, etc.) to provide the necessary flexibility. Therefore, *PeakEngine* is *configurable* (see Section VI-C1) and supports, among others, three layer types (FC, GRU, PeakGRU) and four activation functions (ReLU, Capped ReLU, hard sigmoid, and hard tanh).

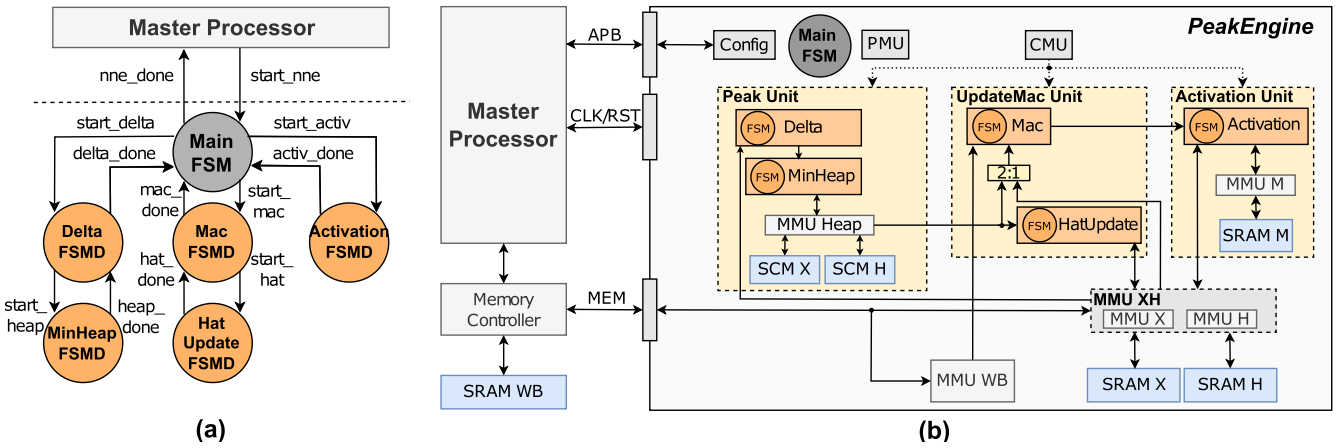

Fig. 6. (a) Hierarchical relationship among the Main FSM and the FSMDs in the system. (b) Top-level overview of *PeakEngine* with a simplified dataflow indicated by the arrows.

B. Top-Level Architecture

Fig. 6(b) shows a high-level architecture of *PeakEngine* with the arrows representing a simplified dataflow. Besides the previously mentioned MMUs, SRAMs, and the three main units (*Peak*, *UpdateMac*, *Activation*), the top level consists of a configuration module (*Config*), a clock management unit (*CMU*) for coarse-level clock-gating, and a power management unit (*PMU*) for the retention of SRAMs. Model parameters are stored in a big *SRAM WB* outside *PeakEngine*. All these modules are further described in Section VI-C.

The accelerator communicates with the *Master Processor* via a 24-bit *advanced peripheral bus* (APB) interface that is used for writing and reading configuration registers in the *Config* module. This includes writing the *start_nne* register to trigger the execution of *PeakEngine* and reading the *nne_done* register to check whether the accelerator has completed inference for the current timestep. *PeakEngine* has two other interfaces: 1) a 32-bit *memory interface* (MEM) for writing inputs and reading the final results for the current timestep stored in *SRAM X* and *SRAM H*, and reading weights and biases from *SRAM WB* and 2) a clock and reset interface (CLK/RST). Thanks to these three generic interfaces, *PeakEngine* is easily configurable and can be ported to any system that supports such communication.

C. Implementation

This section describes details about the main modules and submodules in the system.

1) Config: *PeakEngine* is *configurable*, i.e., all network parameters can be specified at runtime by writing configuration registers via the APB interface. These parameters are: the number of inputs, layers, and neurons per each layer, layer and activation type, starting weight address for each layer, K values to process in a PeakGRU layer, input and result locations (*SRAM X*, *SRAM H*), and a fixed-point format per layer, along with the previously mentioned *start_nne* and *nne_done* registers. In the future, the K values could also be specified as a percentage out of N to generalize the configuration across architectures, which can be easily supported by *PeakEngine*.

2) SRAMs and MMUs: *SRAM X* and *SRAM H* are 1536-word memories that store 16-bit inputs, outputs, and hat states in separate memory blocks. Corresponding *MMU X* and *MMU H* track which of the blocks should be used for reading and writing in different layers. If neural network inference is not needed, the memories can be reused to store other data for a HI.

SRAM WB stores up to ∼2.07 MB of 96-bit vectors composed of 12×8-bit model parameters (see Table I). When smaller networks are executed, the unused part of the memory can be utilized for another purpose. *SRAM WB* is implemented as eleven memory banks of 16 384 words each. Unused memory banks are put into a retention mode during inference. For GRU and PeakGRU, each bias and weight vector is a concatenation of four r, u, and c terms, which enables four new $h(t)$ states to be calculated simultaneously.

SRAMs are supplied by the foundry and operated on two supply voltages: 0.6 V (logic) and 0.8 V (bit cells). All memories in the system (including SCMs) have a delay of one clock cycle for both read and write.

3) Peak Unit: The *Peak Unit* consists of two main submodules, *Delta* and *MinHeap* (see Fig. 7), *MMU Heap*, and two SCMs for storing the top K_x and K_h elements. If a PeakGRU layer is not used in a neural network, the entire module is clock-gated by *CMU*.

PeakEngine executes inference on a per-layer basis and performs the top K selection in parallel with computations in other layers. For instance, while the first FC layer in the demonstrated DNN performs MACs, the *Peak Unit* meanwhile selects the top K_h values for $\Delta h(t-1)$. Similarly, when MAC operations with $\Delta h(t-1)$ values are performed in the PeakGRU layer, the selection of K_x for $\Delta x(t)$ begins. Hence, the latency of selecting the top K elements in both cases is hidden. We also optimize the algorithm by skipping the selection of K_h in the first timestep since both the hidden and hat states are 0.

a) Delta: This unit performs the subtraction, absolute value, and comparison operations in (18)–(19). It reads $x(t)$, $\hat{x}(t-1)$, $h(t-1)$ and $\hat{h}(t-2)$ from *SRAM X* and *SRAM H*. The unit has registers for storing a tuple composed of

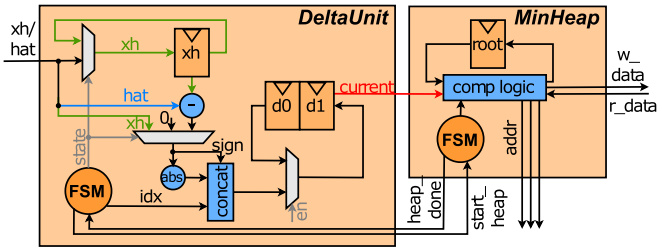

Fig. 7. Simplified overview of the *Delta* and *MinHeap* submodules.

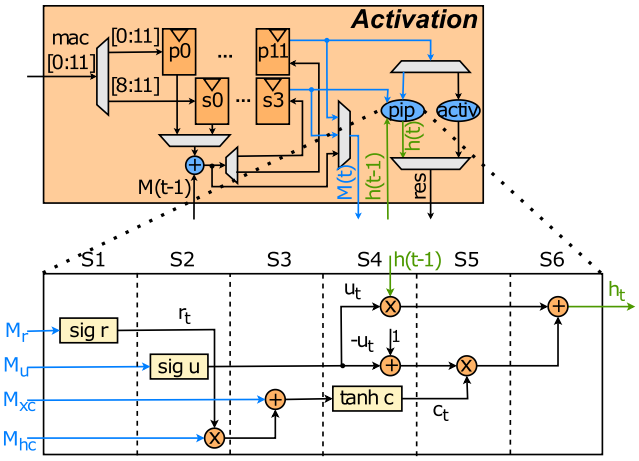

Fig. 8. Simplified overview of *Activation* along with its six-stage pipeline for the GRU and PeakGRU layers.

sign-magnitude-index data, where the sign of the subtraction is necessary for decoding the magnitudes later during the MAC operations. Furthermore, only *nonzero* data are passed to *MinHeap*. We optimize the delta algorithm by skipping subtractions in the first timesteps when $\hat{x}(t-1)$ and $\hat{h}(t-2)$ are still 0.

b) MinHeap: *MinHeap* selects the top K elements by utilizing a *min-heap* data structure (see Section III-B). The K elements are stored as 26-bit tuples (sign-magnitude-index) in the three-port latch-based *SCM X* and *SCM H* that consist of 128 words each. Since the *root* is the most accessed element in the min-heap, we store it locally to avoid unnecessary memory fetches. To further reduce memory accesses, we write a new element to a memory only when its correct position in the heap is found.

Mac and *MinHeap* alternate their access between the two SCMs. While the *Mac* module reads data from *SCM X*, *MinHeap* uses *SCM H* for the top K_h selection, and vice versa. Hence, both SCMs are fully utilized. To optimize subsequent multiplications, the amount of inserted nonzero K_x and K_h elements is stored locally as they can be fewer than K specified in the *Config*.

4) UpdateMac Unit: It receives input data from either: 1) *SRAM X* or *SRAM H* for GRU and FC layers or 2) *SCM X* or *SCM H* for a PeakGRU layer. In the latter case, it checks the sign of the sign-magnitude-index input tuple to extract the original value stored as a magnitude. This value is needed for the multiplications in the *Mac* unit.

a) Mac: The *Mac* submodule loads biases and performs vectorized multiplications between inputs and weights. It utilizes: 1) *output stationary technique*, where the twelve intermediate dot products are kept in local registers until the final result has been computed, which is then passed to the *Activation Unit*; and 2) *input parallelism*, where the input is used for twelve multiplications at a time.

For GRU and PeakGRU layers, the accumulators store four r, u, and c dot products. While the accumulators for r and u hold the final weighted sum, the accumulators for c always keep the results of multiplications with either the input or hidden state vector at a time since these are not directly summed together, as shown in (3) and (16). The four partial c weighted sums are passed to the *Activation Unit*. The weights for PeakGRU are fetched based on the extracted tuple index.

b) HatUpdate: This small submodule updates $\hat{x}(t)$ and $\hat{h}(t-1)$; see (18)–(19). It is triggered only when a new K input, for which a hat update has not been performed yet, has been fetched. If a PeakGRU layer is not used, *HatUpdate* is clock-gated.

When a new element is fetched from one of the SCMs for a MAC operation, its index is stored in a small local buffer. The hat update process requires two cycles to update a single hat value. First, it fetches $x(t)$ or $h(t-1)$ based on the stored index. Then a write is initiated in the next clock cycle to update $\hat{x}(t)$ or $\hat{h}(t-1)$ with the previously fetched value, where the address is generated using the same index.

5) Activation Unit: It consists of *Activation* (see Fig. 8), *MMU M*, and *SRAM M*. When a DNN does not contain a PeakGRU layer, *SRAM M* is powered down.

a) Activation: This unit uses 16 registers for storing the MAC results; twelve primary for all the layers, and four secondary specifically for the GRU and PeakGRU layers to temporarily store the c dot products as described in the *Mac* submodule.

When a FC layer is processed, the MAC results are extracted and stored in the primary registers. The twelve output activations are calculated and written one by one to a memory (*SRAM X* or *SRAM H*) defined in the *Config* module. If no activation function is specified, the final result will be extracted directly from the weighted sum.

For GRU and PeakGRU, output activations are computed in a six-stage *pipeline* illustrated in Fig. 8. As shown in (13) and in Fig. 5(a) for PeakGRU, previous 24-bit M_{hc} delta memory values are fetched from 2048-word *SRAM M* and added to the dot product with $\Delta h(t-1)$. Once the multiplications with $\Delta x(t)$ are completed, the results are written to the primary registers. Thereafter, previous M_r, M_u, and M_{xc} states are fetched from *SRAM M*, updated, and stored back while the pipeline runs. We optimize the memory accesses and computations by skipping zero M states in the first timesteps.

VII. Parameter Space Exploration Framework

A *bit-accurate* in-house software framework supporting FC, GRU, and PeakGRU layers was developed to find the most optimal wordlengths, Q formats, and K values for DNNs

TABLE I
FINAL WORDLENGTH AND Q FORMAT VALUES USED IN THE PRESENTED EXPERIMENTS

	Wordlength	Q format
DNN inputs	16	Q1.15
PeakGRU magnitudes	16	Q3.13
Hat states	16	Q2.14
Input/Output Activations	16	Q2.14
Weights/biases	8	Q2.6
Accumulators	26	flexible
M states	24	Q8.16

executed on *PeakEngine*. It was also used to verify the accelerator outputs. The framework contains identical modules as *PeakEngine* to mimic it bit-accurately during inference. It supports both floating-point and custom fixed-point formats, where the latter is based on the library in [47]. The framework contains a configuration file where wordlengths and Q formats can be defined independently for the network inputs, weights (and biases), outputs, accumulators, and M states. The formats of the remaining intermediate terms are derived automatically based on the other wordlengths specified. The framework supports the creation of a neural network with any number of layers, where each layer is defined as: layer type, number of outputs, activation function, and K value for a PeakGRU layer. The framework performs quantization of network inputs and model parameters, and stores the DNN outputs for the subsequent postprocessing phase. During inference, it exploits multiprocessing to run several noisy speech recordings concurrently and hence speeds up the fixed-point execution.

Table I shows the selected configuration where almost no drop in the objective measures (SNR, PESQ, and STOI) compared to its hardware-aware model (see Section V-C) was observed. The framework was validated against TensorFlow with a maximum error of 10^{-6} (floating point).

VIII. EXPERIMENTAL SETUP

The *PeakEngine* accelerator is evaluated by executing two main DNN architectures.

1) The baseline model with a GRU layer that performs all computations every timestep.
2) The PeakGRU-based model that performs a subset of computations every timestep based on the top K values. Various K values were tested.

The objective of the PeakGRU is, besides reducing computations and power consumption, to decrease latency and hence make real-time inference of big DNNs possible. At the same time, it is important to ensure that the impact of the reduced computations on the objective measures is within acceptable boundaries. Therefore, the selection of the top K values was guided by both: 1) the need to fit within a specified time window and 2) the amount of degradation in the improvement of the objective measures. The following four top K configurations were selected for the final hardware tests: $\{128, 96, 64, 48\}$. $K = 128$ represents a setup with no performance degradation compared to the baseline GRU, while $K = 48$ corresponds to improvements in SNR and PESQ, but no improvement in STOI compared to the unprocessed speech. Lower top K values than 48 result in worse STOI than the unprocessed speech. The same number of K values is used for both input and hidden state sequences, i.e., $K_x = K_h$.

When *PeakEngine* completes inference, it idles until the 25 ms have elapsed and the *Master Processor* writes new data to *SRAM X* or *SRAM H*. Our objective is low-power inference and completing computations *"in time"* instead of achieving extremely low latency and idling for the remainder of the timestep. Therefore, the design operates at a low clock frequency to utilize the majority of the 25 ms window. Several timesteps of a 30-s noisy speech with *Busy Street* background noise are used for the demonstration of *PeakEngine* for all the setups. The results are presented in Section IX.

IX. RESULTS AND DISCUSSION

The *PeakEngine* design (including *SRAM WB*) was synthesized in a 22-nm ultralow leakage CMOS process for a 4-MHz clock frequency. The entire system uses ultrahigh density and ultralow leakage SRAMs that are supplied by the foundry and operated on two supply voltages: 0.6 V (logic) and 0.8 V (bit cells).

This section presents results in terms of area, energy, latency, memory requirements, and objective measures for the tested configurations. They are divided into three main sections: A. comparison between the algorithmic study [32] and the hardware implementation of GRU and PeakGRU, B. comparison when *PeakEngine* executes GRU- and PeakGRU-based DNNs, and C. Comparison of *PeakEngine* against previous works.

A. Algorithmic Versus Hardware Implementation

Table II compares 32-bit floating-point DNN models (FP) from the algorithmic study [32] and our fixed-point DNN models (FX) executed by *PeakEngine* in terms of memory requirements and relative improvements in the objective measures to the unprocessed speech. The FX models use word lengths from Table I. The objective measures for most FP models are derived from the plots in [32] (marked with *), since those experiments were executed for a different number of K than our final experiments. The unprocessed speech has starting values of 4.39 dB (SNR), 1.85 MOS-LQO (PESQ), and 0.83 (STOI). The results are averaged across all the noisy speech in the test subset. The performance of the network under a specific K value is influenced by the type of noise. The impact of using the same K value on all the tested noisy speech is illustrated in our algorithmic study [32].

As shown in Table II, the FX models have almost the same performance as the FP DNNs, while being better suited for hardware inference due to their fixed-point nature. The differences between the two corresponding models cannot be perceived in audio recordings. The knee point in [32], i.e., when the PeakGRU configurations show the first decrease in performance, is around $K = 111$ for SNR. Until then the measures are unchanged. Considering the reported PESQ, the actual knee point is already around $K = 128$, matching the FX model. The FP and FX results are therefore comparable, where

TABLE II

REQUIRED MEMORY AND OBTAINED IMPROVEMENTS (Δ) IN OBJECTIVE MEASURES COMPARED TO THE UNPROCESSED SPEECH FOR: 1) 32-BIT FLOATING-POINT MODEL *(FP)* [32] AND 2) FIXED-POINT MODEL *(FX)* RUNNING ON *PeakEngine*. THE (*) DENOTES APPROXIMATED VALUES BASED ON [32]

	GRU 512		Peak K=128		Peak K=96		Peak K=64		Peak K=48	
Representation	**FP**	**FX**	**FP***	**FX**	**FP***	**FX**	**FP***	**FX**	**FP***	**FX**
#Params	2 099 712									
Memory (MB)	8.01	2	8.01	2	8.01	2	8.01	2	8.01	2
ΔSNR (dB)	8.11	7.81	8.14	7.89	8.07	7.78	7.8	7.46	7.48	7.09
ΔPESQ (MOS-LQO)	0.43	0.42	0.41	0.41	0.39	0.39	0.36	0.36	0.32	0.32
ΔSTOI	0.039	0.037	0.032	0.031	0.026	0.024	0.013	0.007	0.003	0.000

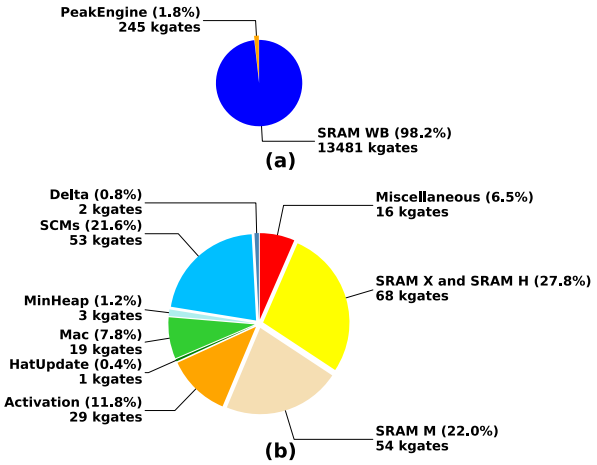

Fig. 9. Area breakdown of (a) entire system and (b) *PeakEngine* only.

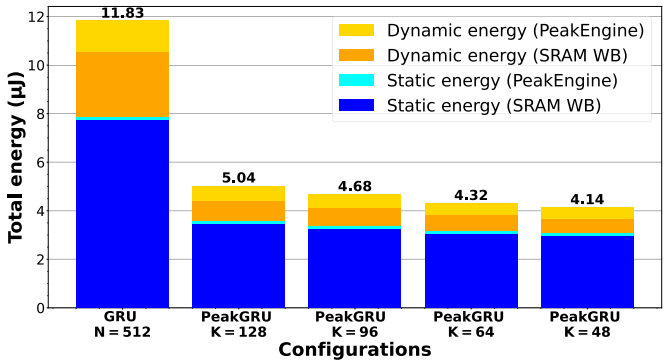

Fig. 10. Comparison of total energy for GRU and four PeakGRU configurations per inference, further divided into static and dynamic.

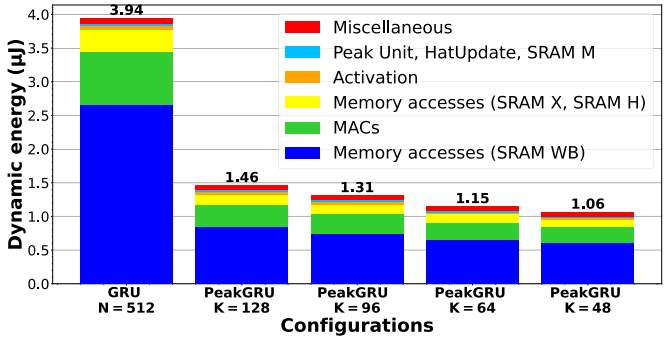

Fig. 11. Comparison of dynamic energy for GRU and four PeakGRU configurations per inference.

4× less memory is needed for 8-bit FX models, corresponding to 6 MB saved.

B. PeakGRU Versus GRU

1) Area: Fig. 9(a) shows the area breakdown of *PeakEngine* with *SRAM WB* and Fig. 9(b) without *SRAM WB* in kgates. As expected, the majority, i.e., 98.22% of the whole area (13 726 kgates, 2.95 mm^2) is dominated by the large *SRAM WB* (13 481 kgates, 2.9 mm^2), while the accelerator itself represents only the remaining 1.78% (245 kgates, 0.053 mm^2). Almost 30% of the *PeakEngine* area [Fig. 9(b)] is used by *SRAM X* and *SRAM H* for storing inputs, results, and hat states. The *Activation* and *Mac* modules used for all layer types occupy ∼20%. The PeakGRU-related modules, i.e., the *Peak Unit* (*Delta*, SCMs, *MinHeap*), *HatUpdate*, and *SRAM M*, comprise 46.1% of the area, where the SCMs and SRAMs clearly dominate. The total PeakGRU overhead area is in general insignificant (113 kgates), especially when compared to the obtained energy savings described next. Small modules such as MMUs, *Config*, and *CMU* along with the logic in the top level are summed together and represented as *Miscellaneous*. They account for only 6.5% of the area.

2) Energy: Our energy evaluations are divided into two parts: 1) comparing the total savings in terms of static and dynamic energy to see the overall effect of pruning (Fig. 10) and 2) comparing the savings of dynamic energy for all the configurations (Fig. 11).

Fig. 10 shows the total energy dissipation (y-axis) of *PeakEngine* together with *SRAM WB* for different configurations (x-axis) as stacked bars. In total, the baseline GRU DNN dissipates 11.83 μJ per inference compared to only 5.04–4.14 μJ for the PeakGRU configurations. The energy is further divided into *static* and *dynamic*, where the static energy clearly dominates for all five configurations (66.7%, 71.1%, 72.1%, 73.5%, and 74.4%, left to right). Almost all the static energy comes from *SRAM WB*. Although the PeakGRU algorithm targets dynamic pruning, it also assists in decreasing the total leakage by 54.6%–61%, i.e., from 7.89 μJ (GRU) to only 3.58–3.08 μJ per inference. These savings are a result of completing the inference faster and consequently putting *SRAM WB* into retention. Without the leakage of *SRAM WB*,

PeakEngine in total dissipates 4.08 μJ for GRU and only 1.56–1.16 μJ for the PeakGRU configurations per inference.

Fig. 11 shows the total dissipated dynamic energy per inference, where our objective was minimizing energy spent on weight memory accesses and MACs that dominate the computations. GRU dissipates 3.94 μJ per inference. MACs account for 0.78 μJ and weight memory accesses for 2.66 μJ, which corresponds to 19.8% and 67.5% of the total dynamic energy. All the PeakGRU-related modules are clock-gated in the GRU setup, namely the *Peak Unit* (*Delta*, SCMs, *MinHeap*) and *HatUpdate*, while the unused *SRAM M* is powered down. A noticeable portion of dynamic energy (0.34 μJ, 8.5%) is spent on *SRAM X* and *SRAM H* memory accesses. The *Activation* and the rest of the system (*Miscellaneous*) constitute only 1.5% (0.057 μJ) and 2.2% (0.09 μJ) of the total, respectively.

PeakGRU DNNs dissipate only 1.46–1.06 μJ per inference. Although the PeakGRU-related modules (light blue bar) are clock-gated in the GRU DNN, at the same time, they dissipate less than 0.028 μJ per timestep for each of the four K configurations, while saving 2.26–2.6 μJ of the original dynamic energy spent on MACs and memory fetches. Moreover, as it can be noticed in Fig. 11, the PeakGRU configurations also decrease ∼55.2–67.1% of dynamic energy spent on *SRAM X* and *SRAM H* (yellow bar). This reduction corresponds to fewer $x(t)$ and $h(t-1)$ reads since their delta versions for the MAC operations are fetched from smaller and cheaper SCMs instead. *Activation* and *Miscellaneous* dissipate only ∼0.03 and ∼0.07 μJ, respectively.

3) Latency: The baseline GRU model performs 2 097 152 MACs and 175 104 96-bit vector memory fetches of weights to compute 512 gain values every timestep. Approximately 75% of both MACs (1 572 864) and vector memory fetches (131 072) are needed for the GRU layer itself. Executing this amount of computations and memory fetches is not only energy-intensive, but it also exceeds the real-time budget of 30 ms when running at a low clock frequency necessary for HIs. The baseline GRU DNN executes inference in 44.04 ms, where the GRU layer needs ∼33 ms and each FC layer ∼5.52 ms. Pruning considerably reduces the total inference latency down to ∼20–14.83 ms (2.2–2.97×), i.e., below both the limit and the 25 ms input data rate defined in Section IV-A. PeakGRU layers require only ∼8.8–3.7 ms, which is comparable with or even below the latency of a FC layer. The latency of the first FC layer in the PeakGRU DNN slightly increases to ∼5.68 ms. This is a result of configuring the inputs and outputs of the first FC layer to be stored in the same memory (*SRAM X*). Hence the *Mac* unit is stalled for few cycles every time *Activation* writes the final outputs. This setup enables a fast selection of the top K_h elements since the other memory (*SRAM H*) is only accessed by the *Peak Unit*.

Overall, the achieved reductions enable DNN inference to be completed within the real-time and energy budget of a HI.

C. PeakEngine Versus State-of-the-Art

Table III shows a comparison of *PeakEngine* to the state-of-the-art accelerators. These span a wide variety of pruning (model compression, skipping updates/computations) and non-pruning techniques, target different platforms (FPGAs, microcontrollers, ASICs), network architectures, use cases, and requirements in terms of latency, power consumption, and model size. All the stated ASIC accelerators report *synthesis* results.

The *EdgeDRNN* [30] accelerator is most similar to our work in terms of pruning approach, network type, and the number of MAC operations. It utilizes threshold-based *Delta Networks* [29], a GRU layer, and performs eight multiplications per cycle. The weights are stored in an off-chip memory. However, the accelerator is developed for FPGA platforms, consumes 2.3 W, and targets extremely low latency in the range of μs. Additionally, the computation time is *unbounded* due to using a threshold-based pruning.

SpAtten is [26] an ASIC accelerator that focuses on dynamic pruning in huge transformer neural networks [51] for natural language processing tasks. It also applies the top K selection, however by using a quick-select algorithm instead. Nonetheless, the average and worst case time complexity of quick-select is $O(N)$ and $O(N^2)$, compared to the min-heap's $O(1)$ and $O(N \log K)$, respectively. Moreover, quick-select requires significant memory space $O(N)$, while the min-heap only $O(K)$. Furthermore, *SpAtten* runs at 1–2 GHz and comprises of two parallel top-K engines that together perform 1024 MACs. This considerably higher parallelism results in area of 18.71 mm^2, excluding huge memories for storing 345 million weights, and it has a total power consumption of 8.3 W.

TinyLSTM [22] also focuses on supporting RNNs for SE in HIs. The authors reduce computations and memory accesses via static pruning and additionally introduce a scheme for skipping LSTM state updates, which can be seen as a form of dynamic temporal pruning. While the application and use case match perfectly with ours, this work does not propose an actual hardware accelerator. Instead, the STM32 microcontroller with ARM Cortex-M7 is used to run the inference, consuming 0.54 W.

The *E-PUR* ASIC accelerator [48] targets the execution of large LSTM networks (1–272 MB) in low-power mobile devices for various use cases such as video classification, speech recognition, and neural machine translation. Table III states the biggest supported model (272 MB) that is used for machine translation. The smallest model demonstrated for speech recognition (LibriSpeech dataset) uses 4× Bi-directional LSTMs and requires 42 MB of weight memory. The accelerator does not apply pruning. Instead, it exploits a novel technique, called maximizing weight locality (MWL), that improves the temporal locality of the synaptic weights. *E-PUR* provides 14 MB of on-chip memory: 8 MB for weights to store one layer at a time (MWL applied) and 6 MB as intermediate storage. The area and average power consumption of the accelerator itself, excluding the large off-chip memory, are 64.6 mm^2 and ∼1 W (averaged across applications), which is amenable for mobile devices, however, not for HIs.

Another ASIC accelerator called *SHARP* is presented in [49]. Like *E-PUR* [48], it is demonstrated on the same

TABLE III
COMPARISON OF *PeakEngine* WITH PRIOR STATE-OF-THE-ART WORKS. ALL THE STATED ASIC ACCELERATORS PRESENT *Synthesis* RESULTS

	EdgeDRNN [30]	**SpAtten [26]**	**TinyLSTM [22]**	**E-PUR [48]**	**SHARP [49]**	**Skip [28]**	**This Work (Peak128)**
Platform	FPGA XC7Z007S	ASIC 40 nm	STM32F746VE Arm Cortex-M7	ASIC 28 nm	ASIC 32 nm	ASIC 65 nm	**ASIC 22 nm**
NN Architecture	2×DeltaGRU (768)	GPT-2 Medium	2×LSTM(256) 2×FC(128)	17× LSTM(1 024)	17× LSTM(1 024)	Embedding(300) -LSTM(300)	**FC(512)-PeakGRU (128/512)-FC(512)**
Sparsity type	Temporal	Temporal	Weight/Temporal	None	None	Temporal	**Temporal**
Pruning approach	Threshold	Top K (quick-select)	Threshold	None	None	Threshold	**Top K (min-heap)**
Voltage (V)	N/A	N/A	N/A	0.78	0.85	N/A	**0.6 (logic), 0.8 (bit cells)**
Freq (MHz)	125	1 000-2 000	216	500	500	200	**4**
#Params (millions)	5.4	345	0.46	N/A	N/A	N/A	**2**
Activation/weight WL	FX16/FX8	FX12/FX8-FX12	FX8/FX8	FP16-FP32	FP16	FX8/FX8	**FX16/FX8**
Weight MEM (MB)	N/A (off-chip)	N/A (off-chip)	0.43 (flash memory)	272 (off-chip)	272 (off-chip)	N/A	**2**
Accelerator MEM (MB)	0.25	0.383	0.313	14	28.3	N/A	**0.013**
Area (mm^2) (synthesis)	-	18.71 (w/o param storage)	-	64.6 (storage for 1 layer)	101.1 (storage for 1 layer)	1.1 (w/o param storage)	**2.95**
Latency (ms)	0.536	27.88	2.39	N/A	N/A	N/A	**25**
Accelerator power (mW)	66	1 360	-	975 (averaged)	N/A	N/A	**0.029**
Total power (mW)	2 290	8 300	540	N/A	8 110 (averaged)	N/A	**0.202**
Total energy (mJ)	1.227	231.404	1.291	N/A	N/A	N/A	**0.005**
Application	Speech recognition	Text Generation	Speech enhancement	Neural Machine Translation	Neural Machine Translation	Language Modeling	**Speech enhancement**
Dataset	TIDIGIT	Wikitext, Penn Tree Bank, One-Billion Word	CHiME2	WMT	WMT	Penn Treebank	**VCTK [41], ADD [42], ES [43] Demant [50]**
#MACs	8	1 024	N/A	16	1 000	192	**12**

models (smallest 40 MB) and use cases, and does not apply any pruning method. The accelerator is benchmarked for different LSTM dimensions as well. *SHARP* also emphasizes the importance of adaptive computations, however, via a tiled-based dispatching mechanism to handle the data dependencies, and not via dynamic pruning. Table III again shows the biggest supported network. *SHARP* provides 26 MB of on-chip weight memory to store one LSTM layer at a time and 2.3 MB of buffers. The smallest configuration has 1 000 MACs, an area of 101.1 mm^2, and consumes 8.1 W (averaged across applications) - all infeasible for HIs.

Last but not least, the *Skip* [28] ASIC accelerator for edge devices performs dynamic pruning of hidden state vectors in an LSTM unit. The proposed method, however, requires training and cannot be applied directly during inference like PeakGRU. Additionally, it focuses on language modeling task, i.e., predicting a next word in the sequence. The authors do not report total power or energy, only the area of 1.1 mm^2, energy efficiency, and peak performance compared to previous works.

Our proposed *PeakEngine* serves as a first ASIC accelerator for dynamic and deterministic pruning of RNNs that targets HI applications and the SE use case. The example configuration with $K = 128$ (*Peak128*, Table III) consumes 29 μW without the big weight memory and 202 μW in total. *PeakEngine* has a small total area of 2.95 mm^2 while supporting big DNNs within the given time and energy constraints. The weight memory occupies 2.9 mm^2 and the accelerator itself only 0.053 mm^2. *PeakEngine* is configurable and easily portable, and it can be used as a coprocessor to a typical digital signal processor in HIs to take off the neural processing workload from the system. Overall, *PeakEngine* is suitable for low-power and resource-constrained embedded devices such as HIs.

X. CONCLUSION

This article presented *PeakEngine*, a configurable ASIC accelerator for low-power edge devices, such as HIs, that supports dynamic and deterministic pruning of input and hidden state sequences in a GRU layer. This is accomplished via the top K element selection by utilizing the small and efficient *Min-heap engine*. The algorithm-hardware codesign of the PeakGRU algorithm and *PeakEngine* significantly reduces total energy and latency up to 2.86× and 2.97×, respectively, making the energy-efficient and real-time execution of even bigger RNNs viable within the constraints imposed by HIs. The accelerator was demonstrated in the SE task, and it could be used as a coprocessor to a typical digital processor found in HIs. Additionally, we developed a framework for parameter space exploration to identify the most suitable data word length, Q formats, and K values for DNNs executed by *PeakEngine*. The accelerator is synthesized in a 22 nm CMOS technology and evaluated at a 4 MHz clock frequency while operating at two supply voltages: 0.6 V (logic) and 0.8 V (bit cells). It occupies 0.053 mm^2 without the weight memory and 2.95 mm^2 in total. To the best of our knowledge, *PeakEngine* is the first ASIC accelerator for low-power dynamic and deterministic pruning of RNNs that targets support of HI-relevant use cases such as SE.

REFERENCES

[1] D. Beck et al., "Audiologic considerations for people with normal hearing sensitivity yet hearing difficulty and/or speech-in-noise problems," *Hearing Rev.*, vol. 25, no. 10, pp. 28–38, 2018.

[2] S. Kochkin, "MarkeTrak V: 'Why my hearing aids are in the drawer' the consumers' perspective," *Hearing J.*, vol. 53, no. 2, pp. 34–36, 2000.

[3] H. B. Abrams and J. Kihm, "An introduction to MarkeTrak IX: A new baseline for the hearing aid market," *Hearing Rev.*, vol. 22, no. 6, p. 16, 2015.

[4] M. Kolbæk, "Single-microphone speech enhancement and separation using deep learning," 2018, *arXiv:1808.10620*.

[5] Y. Ephraim and D. Malah, "Speech enhancement using a minimum-mean square error short-time spectral amplitude estimator," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-32, no. 6, pp. 1109–1121, Dec. 1984.

[6] P. C. Loizou, *Speech Enhancement: Theory and Practice*, 2nd ed. Boca Raton, FL, USA: CRC Press, 2013.

[7] R. C. Hendriks, T. Gerkmann, and J. Jensen, "DFT-domain based single-microphone noise reduction for speech enhancement: A survey of the state of the art," *Synth. Lect. Speech Audio Process.*, vol. 9, no. 1, pp. 1–80, Jan. 2013.

[8] G. Kim, Y. Lu, Y. Hu, and P. C. Loizou, "An algorithm that improves speech intelligibility in noise for normal-hearing listeners," *J. Acoust. Soc. Amer.*, vol. 126, no. 3, pp. 1486–1494, Sep. 2009.

[9] Y. Fan, X. Lu, D. Li, and Y. Liu, "Video-based emotion recognition using CNN-RNN and C3D hybrid networks," in *Proc. 18th ACM Int. Conf. Multimodal Interact.*, Oct. 2016, pp. 445–450.

[10] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 27, 2014, pp. 3104–3112.

[11] Y. Cui, S. Wang, and J. Li, "LSTM neural reordering feature for statistical machine translation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol. (NAACL HLT)*, 2016, pp. 977–982.

[12] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 6645–6649.

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

[14] K. Cho et al., "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1724–1734.

[15] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, vol. 57, Feb. 2014, pp. 10–14.

[16] M. A. Stone and B. C. J. Moore, "Tolerable hearing aid delays. III. Effects on speech production and perception of across-frequency variation in delay," *Ear Hearing*, vol. 24, no. 2, pp. 175–183, Apr. 2003.

[17] *Acceptable Processing Delay in Digital Hearing Aids*. Accessed: Jan. 4, 2022. [Online]. Available: https://hearingreview.com/practice-building/practice-management/acceptable-processing-delay-in-digital-hearing-aids

[18] T. Litovitz, N. Whitaker, and L. Clark, "Preventing battery ingestions: An analysis of 8648 cases," *Pediatrics*, vol. 125, no. 6, pp. 1178–1183, Jun. 2010.

[19] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" in *Proc. Mach. Learn. Syst.*, vol. 2, 2020, pp. 129–146.

[20] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, Oct. 2021.

[21] S. Han et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 75–84.

[22] I. Fedorov et al., "TinyLSTMs: Efficient neural speech enhancement for hearing aids," in *Proc. Interspeech*, Oct. 2020, pp. 4054–4058.

[23] D. Kadetotad, S. Yin, V. Berisha, C. Chakrabarti, and J.-S. Seo, "An 8.93 TOPS/W LSTM recurrent neural network accelerator featuring hierarchical coarse-grain sparsity for on-device speech recognition," *IEEE J. Solid-State Circuits*, vol. 55, no. 7, pp. 1877–1887, Jul. 2020.

[24] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 11, pp. 7436–7456, Nov. 2022.

[25] J. Tchorz and B. Kollmeier, "SNR estimation based on amplitude modulation analysis with applications to noise suppression," *IEEE Trans. Speech Audio Process.*, vol. 11, no. 3, pp. 184–192, May 2003.

[26] H. Wang, Z. Zhang, and S. Han, "SpAtten: Efficient sparse attention architecture with cascade token and head pruning," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 97–110.

[27] J. Tao, U. Thakker, G. Dasika, and J. Beu, "Skipping RNN state updates without retraining the original model," in *Proc. 1st Workshop Mach. Learn. Edge Sensor Syst.*, Nov. 2019, pp. 31–36.

[28] A. Ardakani, Z. Ji, and W. J. Gross, "Learning to skip ineffectual recurrent computations in LSTMs," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1427–1432.

[29] D. Neil, J. H. Lee, T. Delbruck, and S.-C. Liu, "Delta networks for optimized recurrent network computation," in *Proc. Int. Conf. Mach. Learn. (ICML)*, vol. 70, 2017, pp. 2584–2593.

[30] C. Gao, A. Rios-Navarro, X. Chen, S.-C. Liu, and T. Delbruck, "EdgeDRNN: Recurrent neural network accelerator for edge inference," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 10, no. 4, pp. 419–432, Dec. 2020.

[31] C. Gao, T. Delbruck, and S.-C. Liu, "Spartus: A 9.4 TOp/s FPGA-based LSTM accelerator exploiting spatio-temporal sparsity," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Jun. 10, 2022, doi: 10.1109/TNNLS.2022.3180209.

[32] Z. Jelcicová, R. Jones, D. T. Blix, M. Verhelst, and J. Sparseø, "PeakRNN and StatsRNN: Dynamic pruning in recurrent neural networks," in *Proc. 29th Eur. Signal Process. Conf. (EUSIPCO)*, Aug. 2021, pp. 416–420.

[33] J. W. J. Williams, "Algorithm 232: Heapsort," *Commun. ACM*, vol. 7, no. 6, pp. 347–348, Jun. 1964.

[34] C. N. G. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones, "Hardware-software architecture for priority queue management in real-time and embedded systems," *Int. J. Embedded Syst.*, vol. 6, no. 4, pp. 319–334, Sep. 2014.

[35] S. Rigler, W. Bishop, and A. Kennings, "FPGA-based lossless data compression using Huffman and LZ77 algorithms," in *Proc. Can. Conf. Electr. Comput. Eng.*, 2007, pp. 1235–1238.

[36] X. Fan, J. Stuijt, R. Wang, B. Liu, and T. Gemmeke, "Re-addressing SRAM design and measurement for sub-threshold operation in view of classic 6T vs. standard cell based implementations," in *Proc. 18th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2017, pp. 65–70.

[37] P. Meinerzhagen, O. Andersson, B. Mohammadi, Y. Sherazi, A. Burg, and J. N. Rodrigues, "A 500 fW/bit 14 fJ/bit-access 4 kb standard-cell based sub-VT memory in 65 nm CMOS," in *Proc. Eur. Solid-State Circuit Conf. (ESSCIRC)*, Sep. 2012, pp. 321–324.

[38] O. Andersson, B. Mohammadi, P. Meinerzhagen, A. Burg, and J. N. Rodrigues, "Ultra low voltage synthesizable memories: A trade-off discussion in 65 nm CMOS," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 63, no. 6, pp. 806–817, Jun. 2016.

[39] *Perceptual Evaluation of Speech Quality (PESQ): An Objective Method for End-to-End Speech Quality Assessment of Narrow-Band Telephone Networks and Speech Codecs*, document ITU-T Recommendation P.862, 2001.

[40] P. C. Loizou, "Speech quality assessment," in *Multimedia Analysis, Processing and Communications*, vol. 346. Berlin, Germany: Springer, 2011, pp. 623–654.

[41] C. Veaux, J. Yamagishi, and K. Macdonald. (2019). *CSTR VCTK Corpus: English Multi-Speaker Corpus for CSTR Voice Cloning Toolkit*. [Online]. Available: https://datashare.ed.ac.uk/handle/10283/3443

[42] G. Andersen. (2011). *Akustiske Database for Dansk*. [Online]. Available: https://www.nb.no/sbfil/dok/nst_taledat_dk.pdf

[43] M. C. Green and D. Murphy, "EigenScape: A database of spatial acoustic scene recordings," *Appl. Sci.*, vol. 7, no. 11, p. 1204, 2017. [Online]. Available: https://www.mdpi.com/2076-3417/7/11/1204

[44] C.-W. Lin and J.-S. Wang, "A digital circuit design of hyperbolic tangent sigmoid function for neural networks," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2008, pp. 856–859.

[45] J. S. P. Giraldo and M. Verhelst, "Laika: A 5 uW programmable LSTM accelerator for always-on keyword spotting in 65 nm CMOS," in *Proc. IEEE 44th Eur. Solid State Circuits Conf. (ESSCIRC)*, Sep. 2018, pp. 166–169.

[46] P. Kumar Meher, "An optimized lookup-table for the evaluation of sigmoid function for artificial neural networks," in *Proc. 18th IEEE/IFIP Int. Conf. VLSI Syst.-on-Chip*, Sep. 2010, pp. 91–95.

[47] *Simple Python Fixed-Point Module (SPFPM)*. Accessed: Jan. 15, 2022. [Online]. Available: https://github.com/rwpenney/spfpm

[48] F. Silfa, G. Dot, J.-M. Arnau, and A. Gonzàlez, "E-PUR: An energy-efficient processing unit for recurrent neural networks," in *Proc. 27th Int. Conf. Parallel Architectures Compilation Techn.*, Nov. 2018, pp. 1–12.

[49] R. Y. Aminabadi, O. Ruwase, M. Zhang, Y. He, J.-M. Arnau, and A. González, "SHARP: An adaptable, energy-efficient accelerator for recurrent neural networks," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 2, pp. 1–23, Mar. 2023.

[50] *Demant A/S*. Accessed: Feb. 9, 2022. [Online]. Available: https://www.demant.com/

[51] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 5998–6008.

Zuzana Jelčicová received the M.Sc. and Ph.D. degrees in computer science and engineering from the Technical University of Denmark (DTU), Kongens Lyngby, Denmark, in 2019 and 2023, respectively.

Since then, she has been with Demant A/S, Smørum, Denmark, working as a Digital IC Engineer. Her research interests include low-power edge processing, neural networks, AI hardware accelerators, and algorithm-hardware codesign for resource-constrained embedded devices.

Dr. Jelčicová holds a patent for the algorithms proposed in her Ph.D. Her project was awarded as the best "3MT-Three Minutes Thesis" at EUSIPCO 2021.

Evangelia Kasapaki received the M.Sc. degree in computer architecture from the University of Crete, Rethymno, Greece, in 2008, and the Ph.D. degree in computer science and engineering from the Technical University of Denmark (DTU), Kongens Lyngby, Denmark, in 2015.

Since then, she has been with Demant A/S, Smørum, Denmark, currently working as a Senior Digital IC Engineer. Her interests include networks-ON-chip, multiprocessor system design, embedded and real-time systems, low-power design, and neural networks.

Oskar Andersson (Member, IEEE) received the M.Sc. degree in computer science and engineering and the Ph.D. degree in electrical engineering from Lund University, Lund, Sweden, in 2010 and 2016, respectively.

In 2015, he was an intern with Intel Laboratories, Intel Corporation, Hillsboro, OR, USA. Since 2016, he has been with Demant A/S, Smørum, Denmark, as a Senior Digital IC Engineer. His research interests include power optimization of ultralow voltage and near-threshold voltage circuits, energy-efficient circuits techniques, and biomedical circuits for implantable devices.

Jens Sparsø (Member, IEEE) is a Professor with the Technical University of Denmark (DTU), Kongens Lyngby, Denmark. He has published more than 100 refereed conference and journal papers and is coauthor of the book *Principles of Asynchronous Circuit Design—A Systems Perspective* (Kluwer, 2001), which has become the standard textbook on the topic. His research interests include: design of digital circuits and systems, design of asynchronous circuits, low-power design techniques, application-specific computing structures, multi-core processors, and networks-ON-chips; in short, hardware platforms for embedded and cyber-physical systems.

Mr. Sparsø received the Radio-Parts Award and the Reinholdt W. Jorck Award in 1992 and 2003, in recognition of his research on integrated circuits and systems. He received the Best Paper Award at ASYNC 2005, and one of his papers was selected as one of the 30 most influential papers of ten years of the DATE conference.