A Neural Network Engine for Resource Constrained Embedded Systems

Jelcicova, Zuzana; Mardari, Adrian; Andersson, Oskar; Kasapaki, Evangelia; Sparsø, Jens

Published in:
Proceedings of 54th Asilomar Conference on Signals, Systems, and Computers

Link to article, DOI:
10.1109/IEEECONF51394.2020.9443426

Publication date:
2020

Document Version
Peer reviewed version

A Neural Network Engine for Resource Constrained Embedded Systems

Zuzana Jelčicová*†, Adrian Mardari*, Oskar Andersson*, Evangelia Kasapaki*, and Jens Sparsø†
*Demant A/S, Smørum, DK; †Technical University of Denmark, Kgs. Lyngby, DK
Email: zuje@demant.com, admd@demant.com, oand@demant.com, evka@demant.com, jspa@dtu.dk

***Abstract*—This paper introduces a *dedicated neural network engine* developed for resource constrained embedded devices such as hearing aids. It implements a novel *dynamic two-step scaling* technique for quantizing the activations in order to minimize word size and thereby memory traffic. This technique requires neither computing a scaling factor during training nor expensive hardware for on-the-fly quantization. Memory traffic is further reduced by using a 12-element vectorized multiply-accumulate datapath that supports data-reuse. Using a keyword spotting neural network as benchmark, performance of the neural network engine is compared with an implementation on a typical audio digital signal processor used by Demant in some of its hearing instruments. In general, the neural network engine offers small area as well as low power. It outperforms the digital signal processor and results in significant reduction of, among others, power (5×), memory accesses (5.5×), and memory requirements (3×). Furthermore, the *two-step scaling* ensures that the engine always executes in a deterministic number of clock cycles for a given neural network.**

I. Introduction

Deep neural networks (DNNs) are ubiquitous, finding their application in various areas such as image and video processing [1], robotics [2], medicine [3], games [4] as well as audiology [5]. They are typically executed in the cloud due to their computational complexity and size. The results are then deployed wirelessly to power-constrained edge devices such as hearing instruments. However, sharing data with the cloud is not desirable due to issues such as security, privacy, latency, and connectivity [6]. On the other hand, embedding NNs directly in always-on devices that are extremely limited in area, memory, power budget, and throughput, is a challenging task.

To minimize power consumption, hearing instruments are typically implemented using heterogeneous platforms that include specialized accelerators and DSPs. These DSPs often contain vector datapaths that support 16 or 24-bit fixed-point vector elements matching the accuracy of the audio samples. [7]–[9].

Vector operations are also attractive for NNs, especially for performing multiply-accumulate (MAC) operations. Many works have demonstrated that a wordlength of 8 bits is sufficient for inference, having no or insignificant impact on accuracy [10]–[12]. The benefits are again reduced computational complexity, reduced memory requirements, and – if a vectorized datapath is used – processing of more vector elements per instruction. On the other hand, it increases the risk of overflows when the final MAC product in a wide accumulator needs to be stored back to memory in a reduced format.

A common approach to handle this issue is *quantization*. One of the most widely used quantization techniques is a *static-precision quantization*, where the scale factor is determined for the entire NN [13]–[15]. Opposed to it, a *dynamic-precision quantization* firstly proposed by [16] enables varying multi-precision fixed-point for every layer.

Although DSPs support MAC operations, they are generally not able to exploit input sharing, a fundamental *data reuse optimization* for NNs. This can be solved by moving away from DSPs and developing customized hardware accelerators specifically for NNs. Such accelerators exploit characteristic dataflows found in NNs to enhance parallelism and reduce data movement [6].

This paper targets the issues discussed above and introduces the following contributions:

1) *A dedicated NN engine (NNE)* used as a co-processor to Demant's DSP (*xDSP*). The NNE is optimized by applying a set of mutually dependent techniques with a novel *dynamic two-step scaling*.
2) *A dynamic two-step scaling* mechanism capable of fitting MAC products into the required wordlength at runtime without analysing the data ranges during training and adding expensive hardware for quantization on-the-fly. This method ensures that the NNE always executes in a deterministic number of clock cycles for any arbitrary NN.

The NNE and two-step scaling technique further incorporate the following methods: i) *wordlength reduction* from 24 to 8 bits for all NN parameters ii) *12 optimized MACs in parallel*, and iii) *input* and *output stationary* techniques [17] to significantly reduce memory accesses. A keyword spotting (KWS) NN [18] with a Google Speech Command Dataset (GSCD) [19] is used as benchmark and implemented on both the NNE and the xDSP. The NNE outperforms the xDSP in all aspects (significantly smaller power consumption, memory requirements, and number of memory accesses) thanks to the set of the proposed techniques.

The rest of the paper is structured as follows: Section II presents related work. Section III provides background on the xDSP and the KWS NN. Section IV describes the proposed optimization techniques along with the NNE design. Section V discusses the results, and finally Section VI concludes the paper.

II. Related work

Lower precision introduces higher risk of overflows when the activations need to be stored back to memory in a reduced wordlength. This issue is usually solved by dynamically quantizing activations during inference, or defining a quantization factor during training that is fixed once the model is deployed. Since weights and biases are fixed once training is completed, quantizing them can be done statically.

Authors in [20] propose an overflow management scheme which accumulates partial INT32 results (INT16 inputs) into FP32, along with trading off input precision with length of accumulate chain to gain performance.

In [12], MAC products in INT32 accumulator are firstly downscaled using a multiplier, then cast down to uint8 (with saturation), and finally run through an activation function to produce the final 8-bit output.

The authors of [15] convert the final accumulated value by either clipping it to the predefined limits set by integer and fractional length (wordlength) or rounding to fractional length bits using a specific rounding mode. The wordlength for the fixed-point representation is set to 16 bits.

Using the static approaches in the works above might result in encountering values outside the observed ranges at runtime that must be clipped, likely causing additional loss of accuracy.

A dynamic, multi-precision per-layer data quantization flow is introduced in [16] and adopted in [18]. The weight quantization phase analyses the dynamic ranges of weights in each layer to find the optimal fractional length per layer. Fractional length is then initialized to avoid data overflow. The intermediate data of the fixed-point CNN model and the floating-point CNN model are compared layer by layer using a greedy algorithm to reduce the accuracy loss.

All the above techniques require very deep and detailed analysis of the used NN during training or additional computation resources that are usually not an option for resource-limited devices such as hearing instruments.

Our *dynamic two-step scaling* method that handles overflows on-the-fly does not require to compute a scaling-factor in advance, and it does not introduce any instruction overhead. Furthermore, it works on a per-layer basis, the HW implementation is cheap (implemented as an arithmetic shift operation), and it makes the NNE execute in a deterministic number of cycles.

III. Background

A. Keyword Spotting (KWS) Neural Network

KWS systems in edge devices have limited power budget since they must be always-on and operate real-time, while still delivering high accuracy [18]. These requirements are even more strict for extremely resource-constrained devices like hearing instruments. The pretrained KWS model [18] used as a benchmark in this paper is a fully-connected feed-forward NN with a 250x144x144x144x12 configuration trained for 32-bit floating-point (FP32). The input to the network is a flattened feature matrix where a one-second audio recording is divided into $40ms$ frames with a stride of $40ms$, producing 25 frames analysed in 10 frequency bins (250 inputs). Each hidden layer consists of 144 neurons. The output layer has 12 neurons, each representing one category. The first two neurons correspond to *"silence"* (no speech present in the recording) and *"unknown"* (NN is unable to classify the word). The remaining ten neurons represent the following keywords: *"yes"*, *"no"*,*"up"*, *"down"*, *"left"*, *"right"*, *"on"*, *"off"*, *"stop"*, *"go"*. The dataset [19] has more than 65,000 one-second recordings of 30 short words. The final test set consists of 4,890 audio files.

Both inputs and all parameters (weights and biases) were statically quantized to an 8-bit integer representation without any re-training (post-training quantization) before running the inference. Two quantization modes were tested, namely *asymmetric* and *symmetric*, where the latter one proved to work better for our use case. This technique resulted in a small accuracy loss when going from FP32 to 8-bits. Further fine-tuning could improve the accuracy even more. The activations are quantized dynamically to an 8-bit fixed-point using a *dynamic two-step scaling* technique.

B. Digital Signal Processor (xDSP)

The xDSP is a processor optimized for DSP applications with multiple datapaths, register files, and custom functional units. It has a 96-bit vector datapath that supports $4\times$ 24-bit fixed-point elements in a Q5.19 format. This format is used for all NN parameters (weights, biases) and activations. The MAC unit can multiply four elements at a time and store the intermediate results in a single accumulator. This design has limitations since neurons can only be calculated sequentially.

Weights and inputs/activations are fetched from memory as vectors while biases as 24-bit scalars since only one neuron is processed at a time. The 24-bit datapath is also important for overflow management. Scaling out-of-range values involves a global decision across a layer of neurons to keep the same ratio among the outputs. Scaling and storing outputs of a layer individually requires fetching the previously computed outputs whenever a new largest overflow occurs. If, e.g. the first result in a layer needs one shift, but the second one requires two shifts to fit into 8 bits, the first neuron has to be re-fetched from memory and shifted by the missing number of positions.

IV. The NNE accelerator

Significant improvements can be gained by using the dedicated NNE instead of DSPs, to improve resource utilization during inference by exploiting a set of the proposed optimization techniques.

A. Reduced wordlength

The wordlength and representation of the NN model parameters has a big impact on the inference performance. Reduced parameter wordlength enables reading more elements through the same memory interface in a single cycle, and lower precision multipliers require less silicon area and power.
The original accuracy using FP32 on the KWS task is 81.77%. Using 24, 12, 8, and 6-bit representation for all NN parameters

results in 81.24%, 80.36%, 80.28%, and 76.13% accuracy, respectively. The FP32 and 24-bit xDSP implementations achieve comparable accuracy, and a negligible drop is also observed using 12 and 8 bits. Further reduction to 6 bits results in a significant drop, and 8 bits are therefore selected as a convenient trade-off between wordlength and accuracy, decreasing memory requirements up to 3×.

B. Parallel MACs

The MAC unit in the xDSP can only process one neuron at a time. Considering a 96-bit memory interface and an 8-bit wordlength, the MAC is designed such that it can compute 12 intermediate results in parallel (see Figure 1). The intermediate values are accumulated in accumulators with larger precision to avoid overflows and loss of precision. Additionally, a feature for preloading the biases in the accumulators is included. This step saves one addition for each neuron, and serves as reset for the accumulators. The theoretical minimum number of MAC operations required for the inference of an arbitrary network using our NNE is given by:

$$MAC_{op} = \left(\sum_{i=1}^{N} A_i \times O_i \right) / V, \tag{1}$$

where MAC_{op} is the number of vectorized MAC operations, A is the number of activations/inputs to a given layer, O is the number of outputs in the layer, N is the number of layers (excluding input layer), and V is the number of elements in a vector. The minimum number of MAC operations required for the KWS application is therefore 6600 per inference in the NNE.

C. Data reuse techniques

Input stationary and *output stationary* dataflows [17] are used in our design to minimize the data movement between the NNE and memory. An *Input stationary* dataflow reduces the power by multiplying the input with weights of 12 different neurons in a layer, and an *output stationary* dataflow does so by accumulating 12 intermediate results. The final results are transferred to memory as a 12× 8-bit vector. By combining all the optimization techniques introduced previously, memory accesses are reduced by at least 5.5× as shown in Table I.

In the xDSP, only four elements are fetched in a vector. We thus need 63 vectors to represent 250 inputs, and 36 vectors to represent 144 inputs. All these vectors are loaded for each neuron in a corresponding layer since parallel processing is not possible in the xDSP.

In contrast, the NNE efficiently handles 12 elements at a time (both inputs and outputs), resulting in 21 vectors for the layer with 250 neurons, and 12 input vectors for the layers with 144 neurons. These are then multiplied with vectors of 12 neurons instead of a single one. This gives further reduction from 144 to 12 vectors, and 12 to one vector. The number of memory reads (inputs and weights) is, thus, considerably reduced from 39,744 to only 7,176. The number of loads for biases is negligible, and is therefore omitted from the calculations. Moreover, the total number of memory accesses might grow even more for the xDSP, depending on how often the overflows occur.

TABLE I
NUMBER OF INPUT AND WEIGHT VECTORS TO LOAD IN THE XDSP VS NNE.

		xDSP	NNE
Layer 1 (144)	**Inputs**	63 x 144	21 x 12
	Weights	63 x 144	21 x 144
Layer 2 (144)	**Inputs**	36 x 144	12 x 12
	Weights	36 x 144	12 x 144
Layer 3 (144)	**Inputs**	36 x 144	12 x 12
	Weights	36 x 144	12 x 144
Layer 4 (12)	**Inputs**	36 x 12	12 x 1
	Weights	36 x 12	12 x 12
Total		**39,744**	**7,176**

D. Two-step scaling

As mentioned in section III-B and IV-C, additional memory accesses are necessary in the xDSP to reload and scale already computed outputs in a layer if an overflow occurs. This increases the power consumption, and makes the cycle count non-deterministic which may compromise real-time processing. These issues are handled in the NNE by introducing a *dynamic two-step scaling* method. It is divided in the following parts:

1) **Within a vector (when writing results to the memory)** - this step handles overflows when writing accumulator values back to memory. The MAC products that are stored in accumulator registers are scaled down. This is done in vectors of 12 neurons (see Figure 1) that are computed simultaneously.
   Once the computations per vector are finished, the biggest positive number among the 12 results is found. Negative numbers are excluded from the calculations since the ReLU activation function zeros them out afterwards. The biggest positive value determines the scale factor per vector, i.e. the number of shifts necessary to fit the result into 8 bits. This is shown in Table II. The scaled vector elements are stored in memory and the corresponding per-vector scaling factors (green 2, blue 1, orange 3) in a special register in the register file.
   When all the vectors of neurons have been calculated (the layer has been completed), the biggest number of shifts among all the vectors is determined (3 in this example).

TABLE II
TWO-STEP SCALING, PART 1 - FINDING THE BIGGEST SCALE FACTOR PER VECTOR OF 12 NEURONS AS WELL AS PER LAYER (RED NUMBER).

Group	Number of shifts per vector	Additional shifts
Green	2	?*
Blue	1	?*
Orange	**3**	?*

* Additional shifts are not known at this point

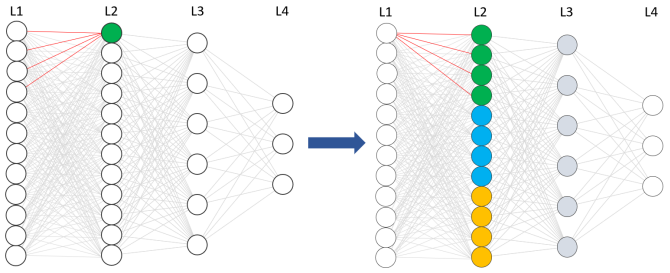

Fig. 1. xDSP (left) vs NNE (right) MAC unit. The xDSP is able to process one neuron at a time using four 24-bit inputs. The NNE processes vectors of twelve 8-bit neurons in parallel (vectors of four are shown for simplicity). The two-step scaling firstly identifies scale factors for vectors in L2, and then performs additional shifts when these vectors are loaded as inputs for L3.

2) **Across layer (when reading results from the memory in a subsequent layer)** - this step maintains the ratio across the entire layer. In the previous step, a scaling factor for each vector of 12 neurons and the biggest scaling factor across a layer were found. When computing a new layer L3 as shown in Figure 1, the previously scaled outputs are retrieved as inputs.
The biggest number of shifts determined in the previous step is used to specify missing shifts for each vector in order to scale the inputs correctly. Therefore, the biggest number of shifts per vector needs to be subtracted from the biggest number of shifts in the previous layer (see Table III). The additional scaling is performed when the vector is read from memory in the next layer.

TABLE III
TWO-STEP SCALING, PART 2 - CALCULATING THE MISSING NUMBER OF SHIFTS FOR EACH INPUT VECTOR.

Group	Number of shifts per vector	Additional shifts
Green	2	3 - 2 = 1
Blue	1	3 - 1 = 2
Orange	3	3 - 3 = 0

Using this *dynamic two-step scaling* never requires to retrieve and process already computed results to perform additional scaling as is the case when using the xDSP. Moreover, the number of memory accesses to execute is always calculable using the *dynamic two-step scaling*. It also makes the NNE always execute in a deterministic number of cycles without adding any additional overhead cycles related to activation scaling. Furthermore, quantizing the fixed-point by powers of two is considerably cheaper than other presented approaches, since it only requires arithmetic shift operations. Finally, the following equation yields the total number of cycles needed for inference of an arbitrary network:

$$2N + \sum_{i=1}^{N} \left\lceil \frac{O_i}{12} \right\rceil \times \left(3 + 13 \left\lceil \frac{A_i}{12} \right\rceil \right) + \left\lceil \frac{O_i}{12} \right\rceil, \qquad (2)$$

where N is the number of layers excluding the input layer; O is the number of output neurons in the current layer; and A is the number of inputs/activations to the layer.

E. The NNE Design

Figure 2 illustrates the NNE datapath which implements the optimizations explained above. The control path that consists of address generation modules (reading/writing data from/to memory), a configuration module (registers storing parameters such as the number of layers, starting read/write address etc.), and a finite state machine (handling the entire NNE flow) is not shown for clarity.

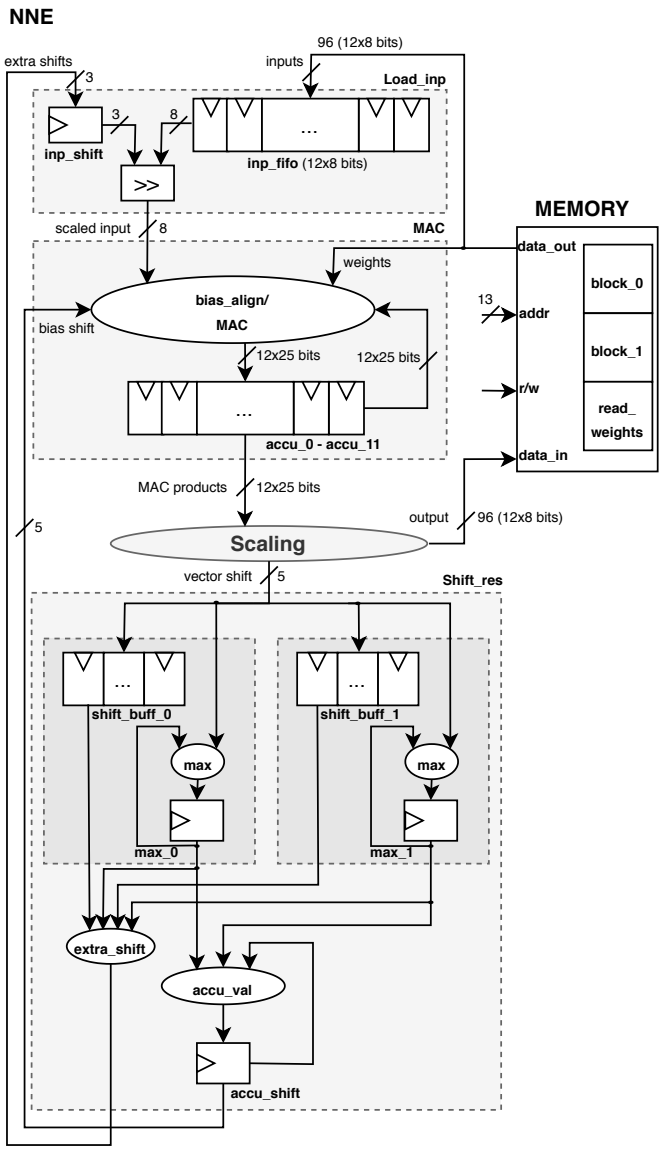

Fig. 2. A high-level overview of the NNE datapath that implements the four optimization techniques.

The NNE MEMORY consists of seven memory instances. Smaller memories are preferred over a single, big memory to decrease dynamic power for a read operation that dominates the inference. The increased leakage is solved by switching

on/off the individual memory blocks when necessary. The memory instances (see Figure 2) are split into three address blocks: block_0, block_1, and read_weights, where the first two blocks switch the *read inputs/write results* role after every layer. The memory block used for reading the inputs in one layer will become a block for storing the results in the next layer and vice versa. The weight memory block (read_weights) does not change, and the weight address is incremented after each retrieved weight vector. Biases are stored in the same memory block. The first address in the read_weights block contains biases for the vector of neurons to be processed. The following addresses represent their weights. Summing all the required vectors together for the KWS task results in 6,694 96-bit vectors (∼78.45kB) that can be represented with a 13-bit address range (addr).

The NNE performs the computations on a per-layer basis. When the NN inference begins, the bias values are preloaded in the accumulator registers (bias_align/MAC and accu_0-accu_11 in the MAC unit). This step saves one addition and also resets the accumulators.

The first 96-bit vector of 12 inputs is then loaded into the inp_fifo in the Load_inp unit, along with a 3-bit value (inp_shift) for additional realignment (two-step scaling, 2nd part).

The next step are 12 parallel MAC operations (bias_align/MAC in the MAC unit). A 96-bit input vector consisting of 12 weights for 12 different output neurons is loaded. Intermediate results are stored in twelve 25-bit accumulator registers to avoid overflows and any loss of precision.

When all input vectors have been loaded, and all MACs have been performed for the current vector of 12 neurons, a local scaling factor is found (two-step scaling, 1st part). The Scaling logic applies ReLU on 12 accumulated values, and outputs both twelve 8-bit results as a 96-bit vector, and a 5-bit output representing a scaling factor per vector.

The scaling factor is found among positive inputs by counting leading zeros. Once it is determined, the number of shifts for each input is obtained by subtracting the computed minimum leading zero count and final result word length (8 bits) from the accumulator word length (25 bits). If the incoming 25-bit input is negative, the 8-bit result is directly set to zero (ReLU). Else the input is shifted by the calculated value. The 96-bit result will be stored in the memory in the next clock cycle. The 5-bit shift value is mapped as an input for the shift_buffs in the Shift_res unit.

The shift_buff keeps track of the arithmetic shift count performed for each vector of activations, such that each vector is appropriately aligned when loaded into the inp_fifo. Two shift_buffs are necessary (shift_buff_0 and shift_buff_1). One shift_buff is used for storing 5-bit unsigned shift values for the vectors in the layer currently being computed. The second shift_buff contains 5-bit unsigned shift values from the previous layer that will be used to calculate the scaling factors for the current inputs (two-step scaling, 2nd part, extra_shift). The Shift_res unit decides which shift_buff is for reading and writing the shift values. The roles of the shift_buffs are swapped at the end of every layer. The accu_val logic populates the accu_shift register with the accumulated number of shifts from previous layers in order to correctly align biases in the accumulator.

V. Results and Discussion

The performance of the NNE and the xDSP is compared in terms of memory traffic, memory capacity, power, area, and accuracy. Similar comparisons with other works are provided as well.

Both the xDSP and the NNE design were synthesized with 28nm CMOS technology. Power simulations for the KWS application were run at 0.7V and 2MHz. The results per inference, i.e. processing of a one-second audio file, are shown in Table IV. Accuracy is given as an average over 4,890 audio recordings.

TABLE IV
24-bit DSP vs 8-bit NNE. Both designs were synthesized with 28nm CMOS technology at 0.7V and 2MHz.

	xDSP (24-bit)	NNE (8-bit)
Memory accesses (SIMD4, 96 bits)	39,744	7,250
Memory accesses (scalar, 24 bits)	888*	-
Bits transferred (kB)	468.35	84.97
Memory occupied (kB)	235.37	78.45
Power (μW)	43.3	9
Clock cycles	∼45,000	7,332
Area (mm^2)	0.71	0.19
Accuracy (%)	81.24	80.28

* This number can grow depending on how often overflows occur

The total memory capacity required for storing all parameters along with results and inputs is 235.37kB and 78.45kB for the xDSP and NNE, respectively. The decrease in memory capacity requirements of more than 3× for the NNE is mostly thanks to the reduced wordlength from 24 to 8-bit for all parameters. The second contributing factor is an optimized approach of reusing the memory space as described in Section IV-E. The xDSP memory has limited capacity. Therefore, a KWS NN with 4× fewer neurons in each layer (63x36x36x36x3) was executed instead, and the memory leakage was scaled as well as the execution time such that it corresponds to the processing time of one frame in order to match the NNE.

The full xDSP implementation of the KWS application requires ∼5.5x more vector memory fetches than the NNE. Moreover, it also needs a baseline of 888 scalar memory accesses to retrieve biases (444) and store the results (444), while the NNE only works with vector operations. The number of scalar memory fetches can grow depending on how often overflows occur. Whenever an overflow is encountered, previously computed results in the same layer must be reloaded from the memory. All these additional accesses significantly contribute to the total of ∼45k clock cycles for the xDSP, while the NNE reduces the clock cycles down to 7,332 (6×), which is very close to the theoretical MAC-based minimum.

Reducing wordlength and the number of memory operations had therefore a significant impact on memory traffic. Only ∼85kB are transferred in the NNE instead of original ∼469kB in the xDSP, resulting in reduction of 5.5×. This assumes

TABLE V
COMPARISON OF THE NNE WITH OTHER WORKS.

	ISSCC 2017 [21]	VLSI 2018 [22]	JSSC 2019 [23]	ESSCIRC 2018 [24]	ISSCC 2020 [25]	This work*
Tech (nm)	40	28	65	65	28	28
Algorithm	DNN	CNN	LSTM	LSTM	DSCNN	DNN
Voltage (V)	0.65	0.57	0.6	0.575	0.41	0.7
Memory (kB)	270	52	65	32	2	79
Area (mm^2)	7.1	1.29	2.56	1.04	0.23	0.19
Freq (MHz)	3.9	2.5	0.250	0.250	0.040	2.0
Latency (ms)	7	0.5-25	16	16	64	40
Keywords	10	11	10	4	1∼2	10
Power (μW)	288	141	10.6	5	0.51	9
Dataset	NA	TIDIGITS	GSCD	TIMIT	GSCD	GSCD
Accuracy (%)	NA	96.11	90.87	91.8	98@1 word 94.6@2 words	80.28

* Synthesis results.

the best-case scenario for the xDSP, i.e. excluding additional operations due to overflows.

Average power consumption during inference for the xDSP and the NNE is 43.3μW and 9μW, respectively, making the accelerator $\sim$5$\times$ more power efficient. As expected, most of the NNE power is used on the memory accesses that dominate the inference. The NN memory is accessed in 98.88% of the clock cycles with 7,213 vector load operations and 37 vector store operations. Memory operations consume almost 91% of the total power, while for the xDSP it is approximately 80%. Moreover, the NNE area (0.19mm^2) is 3.7$\times$ smaller than the xDSP area (0.71mm^2).

All the above improvements compensate for a negligible 1% loss of accuracy from 81.24% (xDSP) to 80.28% (NNE).

Table V is included for completeness and shows comparisons with prior works. As observed in the table, comparing all the works on equal terms is a difficult task since each varies from the rest in many different perspectives. The NNE accuracy is lowest from all the referred works. However, this is mainly due to the selected network topology, which has an accuracy of 81.77% in FP32 precision. The 8-bit implementation results in an almost negligible drop of 1.5% unit using post-training quantization. Retraining the network could bring the accuracy closer to the FP32 result. However, a different algorithm or topology would be required for a more significant improvement. Therefore, the main takeaway is that the NNE offers small area and low power consumption with decent accuracy for a given NN. It outperforms the xDSP significantly and can efficiently execute NN inference in low-power embedded devices such as hearing instruments.

VI. CONCLUSION

This paper presented a dedicated NNE for hearing instruments that implements a cheap, novel dynamic two-step scaling technique for fitting the extended MAC products back to memory in a reduced format. It is implemented as a shift operation that scales the fixed-point by powers of two within i) vectors of neurons, and ii) across all neurons in a layer. The two-step scaling makes the NNE always execute in a deterministic number of cycles. This number of cycles is close to the number of vectorized parameters that need to be retrieved from memory. The NNE also implements other complementary methods to further improve its performance. These are: reducing wordlength from 24 to 8 bits, executing 12 MAC units in parallel, and exploiting input and output stationary techniques. The combination of all of these approaches makes the NNE outperform a typical audio DSP in all aspects. The two implementations were tested using a benchmark KWS NN. In comparison to the xDSP, the NNE reduced power 5$\times$, memory accesses 5.5$\times$, clock cycles 6$\times$, memory requirements 3$\times$, and area 3.7$\times$, while the accuracy dropped only by less than 1%. In general, the NNE offers small area and low power, and it can be easily used in resource constrained embedded devices such as hearing instruments.

REFERENCES

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, Sep 2014.

[2] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search," *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 528–535, 2015.

[3] A. Esteva, B. Kuprel, R. Novoa, J. Ko, S. Swetter, H. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, p. 115–118, Jan 2017.

[4] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan 2016.

[5] L. Deng, J. Li, J. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, Y. Gong, and A. Acero, "Recent advances in deep learning for speech research at microsoft," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 8604–8608.

[6] B. Moons, D. Bankman, and M. Verhelst, *Embedded Deep Learning: Algorithms, Architectures and Circuits for Always-on Neural Network Processing*. Springer, Jan 2019.

[7] D. Harris and S. Harris, *Digital Design and Computer Architecture, Second Edition*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[8] M. Angoletta, "Digital signal processor fundamentals and system design," *CAS 2007 - CERN Accelerator School: Digital Signal Processing, Proceedings*, Jan 2008. [Online]. Available: https://cds.cern.ch/record/1100536

[9] *TMS320C6652 and TMS320C6654 Fixed and Floating-Point Digital Signal Processor*, Texas Instruments, Oct. 2019.

[10] V. V., A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[11] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 5151–5159.

[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2018, pp. 2704–2713.

[13] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2015, pp. 1131–1135.

[14] M. Courbariaux, Y. Bengio, and J.-P. David, "Low precision arithmetic for deep learning," *3rd International Conference on Learning Representations (ICLR2015)*, Dec 2014.

[15] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, p. 1737–1746.

[16] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, and et al., "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, Feb 2016, p. 26–35. [Online]. Available: https://doi.org/10.1145/2847263.2847265

[17] V. Sze, Y. Chen, T. 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, Dec 2017.

[18] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *CoRR*, vol. abs/1711.07128, 2017. [Online]. Available: http://arxiv.org/abs/1711.07128

[19] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *CoRR*, vol. abs/1804.03209, 2018. [Online]. Available: http://arxiv.org/abs/1804.03209

[20] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, A. Heinecke, P. Dubey, J. Corbal, N. Shustrov, R. Dubtsov, E. Fomenko, and V. Pirogov, "Mixed precision training of convolutional neural networks using integer operations," 2018, published as a conference paper at ICLR 2018. [Online]. Available: http://arxiv.org/abs/1802.00930

[21] S. Bang, J. Wang, Z. Li, C. Gao, Y. Kim, Q. Dong, Y. Chen, L. Fick, X. Sun, R. Dreslinski, T. Mudge, H. S. Kim, D. Blaauw, and D. Sylvester, "14.7 a 288μw programmable deep-learning processor with 270kb on-chip weight storage using non-uniform memory hierarchy for mobile intelligence," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 250–251.

[22] S. Yin, P. Ouyang, S. Zheng, D. Song, X. Li, L. Liu, and S. Wei, "A 141 uw, 2.46 pj/neuron binarized convolutional neural network based self-learning speech recognition processor in 28nm cmos," in *2018 IEEE Symposium on VLSI Circuits*, 2018, pp. 139–140.

[23] J. S. P. Giraldo, S. Lauwereins, K. Badami, and M. Verhelst, "Vocell: A 65-nm speech-triggered wake-up soc for 10- μ w keyword spotting and speaker verification," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 4, pp. 868–878, 2020.

[24] J. S. P. Giraldo and M. Verhelst, "Laika: A 5uw programmable lstm accelerator for always-on keyword spotting in 65nm cmos," in *ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC)*, 2018, pp. 166–169.

[25] W. Shan, M. Yang, J. Xu, Y. Lu, S. Zhang, T. Wang, J. Yang, L. Shi, and M. Seok, "14.1 a 510nw 0.41v low-memory low-computation keyword-spotting chip using serial fft-based mfcc and binarized depthwise separable convolutional neural network in 28nm cmos," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 230–232.