CSCMAC - Cyclic Sparsely Connected Neural Network Manycore Accelerator

Hirenkumar Paneliya, Morteza Hosseini, Avesta Sasan, Houman Homayoun and Tinoosh Mohsenin

Abstract— This paper presents an energy-efficient, domain-specific manycore accelerator also referred to as the "**CSCMAC**" - Cyclic Sparsely Connected Neural Network Manycore Accelerator, which effectively maps and executes deep neural networks (DNNs) compressed with cyclic sparsely connected (CSC) architectures. CSC layers are architectures that structurally compress and sparsify DNNs, which can reduce the memory footprint of fully connected (FC) layers from $O(N^2)$ to $O(N \log N)$ with respect to layers nodes, and is shown to be hardware implementable-friendly. We implement CSC layers for inference on a manycore unit, take advantage of their cyclic architecture, and show that their implementation in software even for a parallel-computing processor is affable. To further take advantage of their implementation simplicity, we propose customized instructions for the manycore that fuse frequently used sequences of machine codes and evaluate the optimization gained by the customization. Our experimental results using a LeNet300100 on MNIST and a Multi-Layer Perceptron (MLP) on Physical Activity Monitoring indicate that by replacing FC layers with CSC layers, we can achieve 46× and 6× compression respectively within a margin of 2% accuracy loss. A 64-cluster architecture of the CSCMAC is fully placed and routed using 65nm, TSMC CMOS technology. The layout of each cluster occupies an area of 0.73 mm^2 and consumes 230.2 mW power at 980 MHz clock frequency. Our proposed CSCMAC achieves 1.48× higher throughput and 1.49× lower energy compared to its equivalent predecessor manycore (PENC). Also, the CSCMAC achieves 85× higher throughput and consumes 66.4× lower energy compared to CPU implementation of the NVIDIA Jetson TX2 platform.

Keywords—Programmable Manycore accelerator, Model Compression, Complexity Reduction, Cyclic Sparsely Connected layers,

I. INTRODUCTION

Deep neural networks (DNNs) have become very popular in different areas of research such as image style transfer [1] and image reconstruction [2], as well as physiological data processing including physical activity monitoring [3–7] and EEG [8, 9]. With the advancements made in technology and fabrication of embedded devices, processing DNNs is being pushed toward the edge where resource-bound devices such as wearable gadgets can implement DNNs that are trained to process specific data such as activity features sensed from a human's body. However, there are two main issues while implementing DNNs on small-size and battery-limited devices: 1) the DNN model size and 2) the computational complexity. In order to compress DNNs and facilitate their hardware implementation, several methods have been proposed including quantization [10], binarization [11,12] and pruning [13]. Typically, in pruning methods, a DNN is trained such that much of the weights are pushed toward zero, and then weak weights in the model are identified and removed by defining a threshold value (e.g. a quality value times the standard deviation of the distributed weights [14]), and the strong weights are fine-tuned by retraining the thresholded model such that the accuracy drop caused by the removal of the weak weight is compensated. This method is also referred to as fine-grain pruning, the main drawback of which is the irregular patterns of non-zero weights in the DNN model that necessitate having an extra memory space to locate non-zero entries. We refer to fine-grain pruning methods as *random pruning* in this work.

In contrast, there are works that target coarse-grain pruning or structurally sparsifying DNNs where the indexing issue can be minimized in the former or eliminated in the latter. In coarse-grain pruning methods [15], a DNN model can be pruned similarly to fine-grained pruning methods but on a filter-wise, kernel-wise, or layer-wise basis, and the groups of non-zero entries can together, rather individually, be located with single indicators, thus minimizing the indexing issue. In structurally sparsifying DNNs [16, 17], the DNN model is trained on a predesignated sparse architecture, and as a result of the fixed and known location of the non-zero entries, the indexing issue is eliminated.

Despite the existence of a great amount of work and research on implementing random pruned models efficiently on CPU and GPU (e.g. cuSparse libraries from NVIDIA), to the best of our knowledge, there exists no work of implementing structurally compressed models on multi-core programmable devices, and almost all of the proposed compression methods have designated application-specific integrated chips (ASIC) or specific FPGA implementations to adopt the model [16–19].

In this paper, we propose "CSCMAC" - Cyclic Sparsely Connected Neural Network Manycore Accelerator, a novel energy-efficient programmable cluster-based manycore, implemented on 65 nm technology, that is designed specifically for DNN trained with CSC layers. The proposed architecture has novel instruction referred to as **CSC** which replaces frequently used functions that would have taken 11 clock cycles with 1 clock cycles. To evaluate the performance of the CSCMAC, we train two DNN models with no compression method. Then, we choose CSC architecture [17] as a structurally compressing method and, based on which, train compact representations of

the two DNN models. The original models, as well as their compressed version is then implemented on NVIDIA Jetson TX2 CPU as well as on a manycore unit with no customization. Finally, we propose customized logic and optimized instructions to accelerate the CSC model for better implementation efficiency.

This paper makes the following contributions:

- Proposed a specific instruction for the CSC layer that can generate index addresses and perform zero skippings.
- Designed and implemented optimized hardware for CSC specialized instruction without slowing the operating frequency.
- Fully synthesized, placed and routed ASIC layout of Cyclic Sparsely Connected Manycore Accelerator (CSCMAC) using 65nm TSMC CMOS technology.
- Evaluated and compared the performance analysis of the proposed CSCMAC with an embedded NVIDIA Jetson TX2 CPU platform.

II. COMPRESSED DNNs USING SPECIALIZED CSC INSTRUCTION AND CASE STUDIES

Authors in [17] introduced structurally compressing CSC layers that replace fully-connected layers in a DNN. A CSC architecture is characterized by a handful of hyper-parameters including the size of its layers, denoted by $\boldsymbol{N}$, number of layers, $\boldsymbol{L}$, fan-out of every node in every layer excluding the output layer as well as fan-in of every layer excluding the input layer, denoted by $\boldsymbol{F}$, and $\boldsymbol{C}$ that represents connectivity in layer. Input and output size are denoted as $\boldsymbol{N_I}$ and $\boldsymbol{N_O}$ respectively. As per suggested in [17], the number of parameters in a CSC architecture is calculated as:

$$\boldsymbol{E_{param} = NF(L-2) + N_I F + N_O F,} \qquad (1)$$

and these hyper-parameters satisfy the following equation:

$$\boldsymbol{F^L = NC} \qquad (2)$$

Figure 1 shows the case study networks, LeNet300100 and 3-layer MLP with original FC layers and replacing CSC layers. For both case studies, $\boldsymbol{C = 1}$ and the other hyper-parameters are calculated using equation 1. Table I shows the detailed summary of two case studies and characteristics of the network configurations.

A. CASE STUDY 1: MNIST

A LeNet300100 for MNIST image classification dataset is implemented to evaluate the CSCMAC after replacing FC layers with CSC layers. The MNIST dataset contains 60K training images and 10K testing images. After replacing FC with CSC, the size of the LeNet300100 network is reduced from 267K to 5860 parameters which is 46× reduction at the expense of 1.2% accuracy loss. Figure 1 (A) and (B) show the network architecture of LeNet300100 with FC and CSC layers respectively.

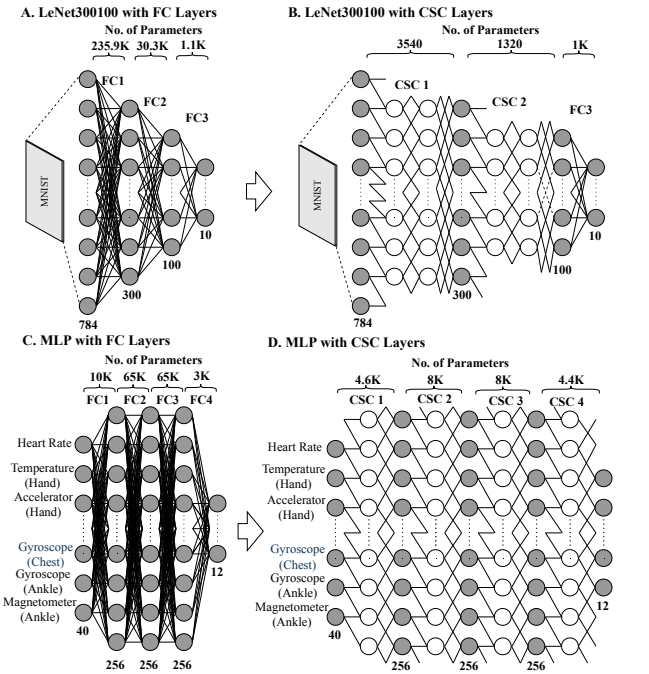

Fig. 1: Configuration of LeNet300100 and MLP with FC and CSC layers

TABLE I: Detailed Summary of the case studies and characteristics of the network configurations used in this paper

Dataset	Input Shape	# of Labels	# of Params	Model Size	Total Ops	Avg Accuracy
MNIST	1×28×28	10	5860	11.6KB	11.6K	97.2%
PAMAP2	40×1×1	12	25K	50KB	50K	97%

B. CASE STUDY 2: PHYSICAL ACTIVITY MONITORING

Physical Activity Monitoring dataset (PAMAP2) [20] is used to evaluate MLP using the TensorFlow framework. PAMAP2 records 12 physical activities performed by 9 subjects. The data are recorded from 12 different physical activities such as standing, walking, lying and sitting and using sensors such as inertial measurement units (IMU) and heart rate monitor. PAMAP2 includes 40 labeled valid channels. We use the 3-layer MLP network as proposed in [12]. Each layer in the network has a size of 256 nodes while input nodes and output nodes are 40 and 12 respectively. Figure 1 (C) and (D) show the network architecture of MLP with FC and CSC layer respectively. In total, the MLP with CSC requires 25K parameters which results in 6× compression. The average classification accuracy is 97%.

III. DOMAIN-SPECIFIC MANYCORE

The proposed CSCMAC architecture is composed of 64 processing clusters (192 cores) connected through routers in a three-level Globally Asynchronous Locally Synchronous (GALS) hierarchical tree. The design is Multiple Instruction Multiple Data (MIMD). Figure 2(A) shows the CSCMAC architecture that includes 64 clusters interconnected using routers. Figure 2(B) shows the

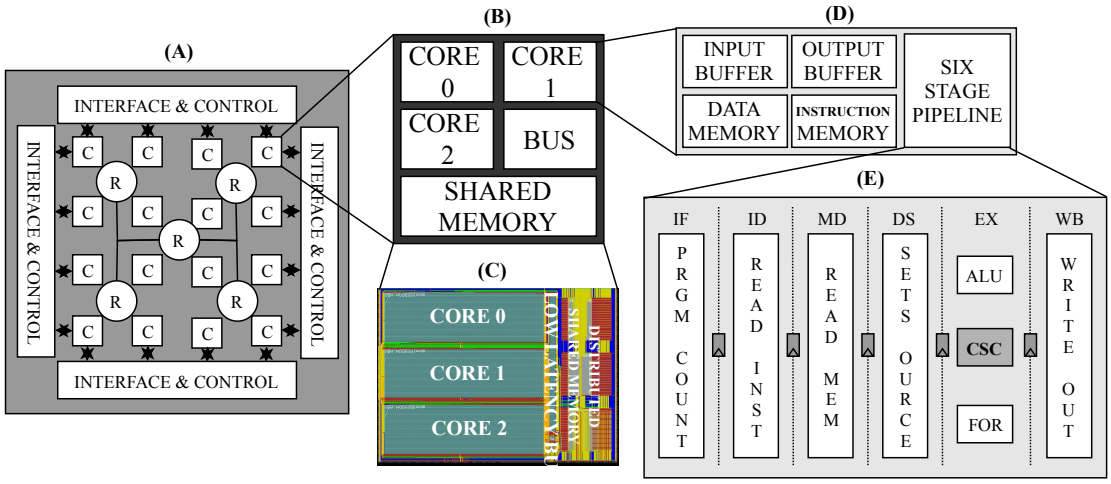

Fig. 2: (A) Cyclic Sparsely Connected Neural Network Manycore Accelerator (CSCMAC) Architecture. All clusters are interconnected with routers. (B) Bus-based Cluster Architecture. It consists of three processing cores, shared memory and a low latency bus. (C) Post-layout view of bus-based cluster architecture implemented in 65nm and 1V TSMC CMOS Technology. (D) Block diagram the of core architecture. It consists of the input buffer, an output buffer, data memory, instruction memory and six-stage pipeline. (E) Block Diagram of the six-stage pipeline. CSC instruction is added to the execution stage

bus-based cluster architecture. It consists of three tiny processing cores, a low latency bus and a shared memory of 3072×16 size. Figure 2(C) shows the post-implementation view of the bus-based cluster implemented in 65nm, 1V TSMC CMOS technology using Cadence Encounter. Figure 2(D) shows the block diagram of the processing core architecture. Each processing core has an input buffer, an output buffer, a RISC-like instruction set architecture including **CSC** instruction, 16 quick-access registers, 128-bit instruction memory and 128-bit data memory. Data is 16-bit wide and instructions are 30-bit wide. Figure 2(E) shows the six-stage pipeline. In the block diagram, IF, ID, MD, DS, EX, and WB are instruction fetch, instruction decode, memory decode, data set, execution and write back pipeline stage respectively.

Two main instructions, IN and OUT are used to pass data between different cores. A low latency bus is used to read data from the output FIFO and to place data into the input FIFO of the destination cores. From the five-port bus, three ports link to routers, one links to the shared memory, and the other links to the external router for communication with outside core clusters. The cluster uses Load (LD) and Store (ST) instruction to access the data from shared memory.

LeNet300100 and MLP are taken as two case studies to evaluate the performance of the proposed CSCMAC. Pre-trained model parameters are stored in shared memory in all clusters which are shared among all active cores in the CSCMAC. The cores that are not used in the CSCMAC are shut down to reduce power consumption. Instruction level parallelism is achieved by dividing the weight matrix in all active cores and send specific rows to a specific core. Figure 3 shows the architecture and memory of the CSCMAC with MNIST configuration. As we discussed in section **2**, we require 5860 weight parameters to implement MNIST and

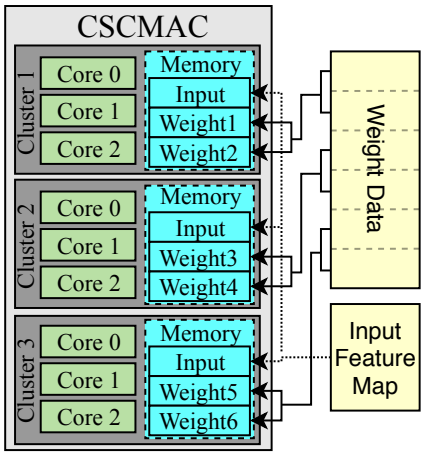

Fig. 3: Hardware architecture of the CSCMAC for three clusters when executing LeNet300100 with MNIST dataset. Memory stands for cluster memory of size 3072×16.

as per equation 3, three clusters with 9 cores are required to store weights data as well as input feature map data. Figure 3 shows the hardware architecture of the CSCMAC for MNIST.

A. CSCMAC WITHOUT SPECIALIZED ISA

Figure 4 shows the assembly code of full matrix multiplication with considering zeros (A), CSC layer without using specialized ISA (Skipping Zeros) (B) and random pruning (C). We didn't mention NOP operation in figure 4. In full matrix multiplication, while considering zeros, we consider every element in the weight matrix as important and do matrix multiplication. In the CSC layer without using specialized ISA, we consider the cyclic pattern of non-zero elements in the weight matrix such as the number of non-zero elements in one row (F), and the number of zero elements between two non-zero elements

in the same row (R). In random pruning, we stored the non-zero weight elements and locations of the non-zero element such as row and column.

(A)

```
MOVI R14,  72
MOVI R13,  0
MOVI R12,  64
MOVI R4,   72
MOVI R5,   64
MAC  R15,  $R13, $R12
INC  R12,  R12
INC  R13,  R13
BNE  5,    R12,  R4
MOVI R12,  64
MOV  $R14, R15
INC  R14,  R14
MOVI R15,  0
BNE  5,    R13,  R5
END
```

(B)

```
MUL  R10,  R1,   R3
ADD  R11,  R10,  R0
INC  R1,   R1
ADD  R11,  R11,  R9
BL   22,   R11,  R8
SUB  R11,  R11,  R5
MAC  R13,  $R11, $R12
INC  R12,  R12
BNE  10,   R1,   R7
INC  R0,   R0
MOV  $R6,  R13
INC  R6,   R6
MOVI R13,  0
BNE  9,    R0,   R4
END
```

(C)

```
BNE  14,  R5,  $R3
MOV  R7,  $R4
INC  R3,  R3
INC  R4,  R4
MAC  R6,  $R2, $R7
INC  R2,  R2
JMP  7
MOV  R5,  $R3
MOV  $R0, R6
INC  R8,  R8
INC  R0,  R0
MOVI R6,  0
BNE  7,   R8,  R1
END
```

Fig. 4: Assembly codes for (A) full matrix multiplication with considering Zeros, (B) CSC layer without using specialized ISA, and (C) Random Pruning.

B. CSCMAC WITH SPECIALIZED CSC INSTRUCTION

A CSC instruction has been added to the execution stage of manycore architecture to reduce the cycle overhead. While activating CSC instruction, it takes the number of non-zero (F) values in one row, and the number of zeros between two non-zero values (R) as input and gives weight address and data address as output. It requires $N \times F$ iterations *for* loop to fetch the address. Figure 5 shows the hardware architecture of specialized CSC instruction for the CSCMAC. The recursive counter (j) counts recursively from 0 to *F-1* and when counter j reaches to *F-1*, address counter (i) will increment by one. Instead of modulo operator, we use a comparator, subtractor and mux circuit to maximize the operating frequency. Figure 6 shows

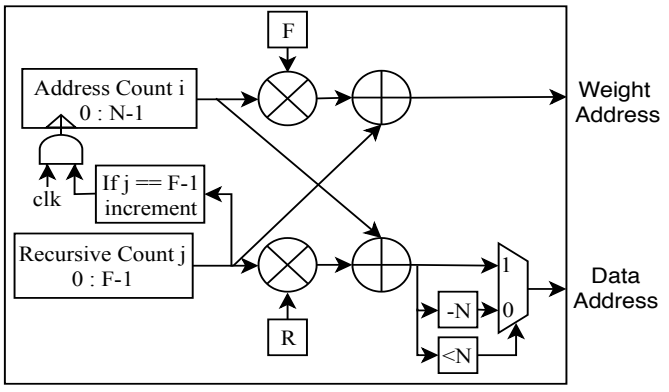

Fig. 5: Hardware architecture of CSC instruction. R, F and N are number of zeros between two non-zero values, number of zeros in one row and number of rows in weight matrix respectively.

the state machine of specialized CSC instruction for the CSCMAC. *S0*, *S1* and *S2* are initial state, column control state and row control state, respectively.

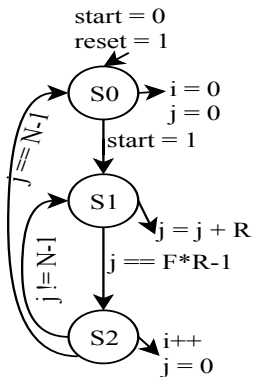

Fig. 6: State machine flow for the CSC instruction

C. COMPARISON BETWEEN DIFFERENT METHODS

For the comparison, we implement full matrix multiplication, CSC layer without using specialized instruction and random pruning method on manycore and compared it with CSC layer with specialized CSC instruction in terms of the number of cycles required for one MAC operation and memory requirements as per shown in figure 7. CSC layer without specialized instruction requires the maximum number of cycles for one MAC operation. But, it skips the zero weight data so, we do not need to perform all MAC operations. Whereas in CSC layer with specialized CSC instruction requires nearly the same number of cycle as considering zero, but it requires less memory to do computation. Random pruning technique takes more cycles and requires more memory because we need to store the locations of non-zero weight data.

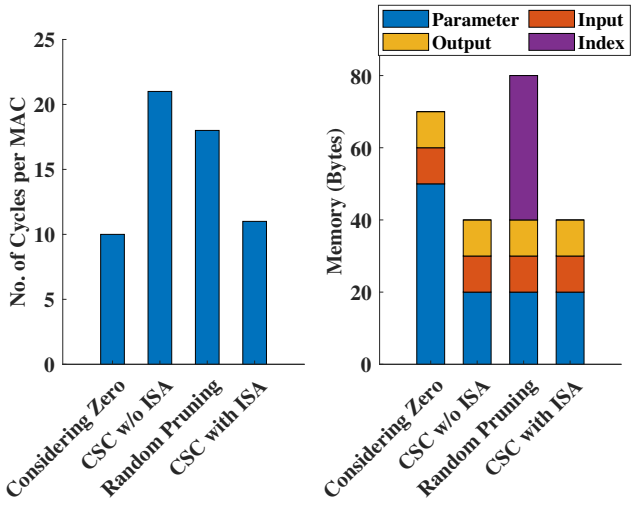

Fig. 7: Bar graphs on the left show the required no. of clock cycle for one MAC operation, and on the right showing the memory requirement for configuration of N = 5, M = 5, R = 1 and F = 2 with four different method full matrix, CSC layer without specialized CSC instruction, Random Pruning, CSC with specialized CSC instruction

D. PROGRAMMING MANYCORE AND SOFTWARE EVALUATION

To evaluate the CSCMAC, we developed a stand-alone hardware simulator and compiler in JAVA and python respectively. The manycore compiler converts the assembly code into binary files as well as converts the initial values of registers and data memories into binary. The compiler performs these operations on each core individually so that we can directly load the binary file on manycore. The manycore simulator is a software implementation of manycore hardware that is utilized to run, simulate, debug and test the assembly code. It reads the assembly code, initial registers and data memory, then simulates as per the assembly code and updates the final state of the registers and data memories. It also gives the summary of clock cycles to execute the assembly program as well as the status of the six-stage pipeline on each clock cycle.

IV. IMPLEMENTATION RESULTS AND COMPARISON

Both compressed LeNet300100 and 3-layer MLP are implemented on the CSCMAC and the results are evaluated in terms of throughput, execution time, energy and power analysis. For better comparison with the different manycore processors, we implemented a compressed LeNet300100 network on the NVIDIA Jetson TX2 SoC CPU platform. For different networks, different numbers of clusters are activated to store the network parameters and feature maps. The least number of required clusters, **A**, can be derived from equation 3 which guarantees to accommodate the processing data:

$$\boldsymbol{A} = \frac{\boldsymbol{Networkweights}}{\boldsymbol{ClusterMemorySize - FeatureMap}} \tag{3}$$

To enhance the performance of the CSCMAC, we need the maximum number of clusters and to achieve a low power platform, we need the least number of clusters. Table II shows the hardware implementation results of LeNet300100 implemented on the CSCMAC and comparison with PENC and NVIDIA Jetson TX2 CPU. For a fair comparison, TX2 CPU results are scaled to 65 nm technology.

A. COMPRESSED LENET300100 FOR MNIST

The implementation results for the LeNet300100 are given in table II. As per equation 3, at least 3 clusters are required for low power CSCMAC implementation. In which, 63.5% of their word memory is used to store the network weights and 25.5% accommodates the input feature map data. The implementation results of LeNet300100 are presented in the form of two different scenarios: one with using specialized CSC instruction (CSCMAC) and second without using specialized CSC instruction (PENC). The CSCMAC takes 5.4 μs to classify one single image which is 1.5× improvement over the PENC. The throughput and energy of the CSCMAC are

TABLE II: Hardware implementation results of LeNet300100 for the CSCMAC compared with PENC and NVIDIA Jetson TX2 CPU. For a fair comparison, TX2 CPU power and energy results are scaled to 65nm.

	PENC	CSCMAC	Improve. over PENC	TX2	Improve. over TX2
Technology (nm)	65	65	-	28	-
Cluster Area (mm^2)	0.73	0.73	-	-	-
Frequency (MHz)	1000	980	-	346	-
Latency (μs)	8	5.4	1.48×	460	85×
Throughput (K label/s)	124.9	183.1	1.48×	2.15	85×
Total Power (mW)	685.5	690.6	0.99×	533.9	0.77×
Energy (μJ)	5.5	3.7	1.49×	245.6	66.4×

183.1 Klabel/s and 3.7 μJ respectively which are 1.48× and 1.49× of improvement compared to PENC manycore.

B. IMPLEMENTATION RESULTS ON NVIDIA JETSON TX2 CPU SOC

The LeNet300100 is implemented on the NVIDIA Jetson TX2 CPU processors. One CPU core is turned on and the CPU clock frequency is 346 MHz. Table II summarizes the results of the LeNet300100 implementation. We also turned off all the other peripheral such as GPU, HDMI, etc. before taking the power and timing results. Based on the results, the CSCMAC achieves 85× higher throughput and 66.4× lower energy consumption compared to the TX2 CPU platform for the LeNet300100.

C. COMPRESSED 3-LAYER MLP FOR PHYSICAL ACTIVITY MONITORING

For the compressed MLP network, at least 9 clusters with 27 cores are required to be activated for power-efficient implementation. From a total of 27K word memory, 92.6% of memory is used for the network weights and the rest is used for the input feature map and intermediate data. The implementation results for the 3-layer MLP are presented in table III in terms of execution time, energy consumption and throughput. The results are presented in the form of two scenarios: with and without specialized CSC instruction. Without using specialized CSC instruction (PENC), it takes 21 μs to classify one single image, therefore classification throughput will be 47.7 Klabel/s, and a power of 2.05 W when running on 9 clusters with 27 cores at the clock frequency of 1 GHz. On the contrary, when using the specialized CSC instruction, the latency drops to 11.8 μs at the clock frequency of 980 MHz, thus throughput is improved by 1.8×. The latter implementation on the CSCMAC has an energy consumption improvement of 1.77× over an MLP implemented on the precedent version of the CSCMAC.

D. COMPARISON WITH EXISTING WORK

We compare our proposed hardware on the two case studies with two relevant works, [17] and [12]. Table IV

TABLE III: Hardware implementation results of the CSCMAC compared PENC for MLP

	PENC	**CSCMAC**	**Improvement over PENC**
Technology (nm)	65	65	-
# of Cluster (A)	27	27	-
Frequency (MHz)	1000	980	0.98×
Latency (μs)	21.0	11.8	1.8×
Throughput (Klabel/s)	47.7	84.8	1.8×
Total Power (W)	2.05	2.07	0.99×
Energy (μJ)	43.1	24.4	1.77×

summarizes the implementation results and comparisons with existing works. Compared to [17] and [12], CSCMAC achieves 9.2× and 6.6× higher throughput respectively.

TABLE IV: Comparison of the CSCMAC with Existing Work

Case Studies	**MNIST**		**Physical Activity**	
	[17]	**This Work**	**[12]**	**This Work**
Technology (nm)	28	65	65	65
Frequency (MHz)	150	980	1000	980
Latency (μs)	50.7	5.4	77	11.8
Throughput (K lable/s)	19.7	183.1	12.9	84.8

V. CONCLUSION

In this work, we proposed a high throughput and energy-efficient Cyclic Sparsely Connected Neural Network Manycore Accelerator (CSCMAC) with specialized CSC instruction. The proposed manycore is placed and routed using 65nm, 1V, TSMC CMOS technology. The post-layout implementation results show that the proposed CSCMAC with LeNet300100 configuration has a throughput of 183.1 Klabel/s which is 1.48× higher than PENC manycore and consumes 3.7 μJ energy which is 1.49× improvement over PENC. Similarly, CSCMAC with MLP configuration achieves 1.8× higher throughput and 1.77× lower energy consumption. We implemented LeNet300100 on NVIDIA Jetson TX2 CPU as well as CSCMAC which achieves 85× higher throughput and 66.4× lower energy consumption compared to the TX2 CPU platform.

References

[1] Gatys *et al.*, "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2414–2423.

[2] Cheng *et al.*, "Highly scalable image reconstruction using deep neural networks with bandpass filtering," *arXiv preprint arXiv:1805.03300*, 2018.

[3] M. P. Tarvainen *et al.*, "Kubios hrv–heart rate variability analysis software," *Computer methods and programs in biomedicine*, vol. 113, no. 1, pp. 210–220, 2014.

[4] A. Jafari *et al.*, "Sensornet: A scalable and low-power deep convolutional neural network for multimodal data classification," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 1, pp. 274–287, Jan 2019.

[5] H. Blackburn *et al.*, "The electrocardiogram in population studies: a classification system," *Circulation*, vol. 21, no. 6, pp. 1160–1175, 1960.

[6] A. Page *et al.*, "Utilizing deep neural nets for an embedded ecg-based biometric authentication system," in *Biomedical Circuits and Systems (BioCAS) Conference*, Oct 2015, pp. 1–4.

[7] M. Nitzan *et al.*, "Low-frequency variability in the blood volume and in the blood volume pulse measured by photoplethysmography," *Journal of biomedical optics*, vol. 1, no. 2, pp. 223–230, 1996.

[8] M. Khatwani *et al.*, "A low complexity automated multi-channel eeg artifact detection using eegnet," in *2019 IEEE EMBS Conference on Neural Engineering*. IEEE, 2019.

[9] M. Khatwani, M. Hosseini *et al.*, "Energy efficient convolutional neural networks for eeg artifact detection," in *2018 IEEE Biomedical Circuits and Systems Conference*, pp. 1–4.

[10] Y. Gong *et al.*, "Compressing deep convolutional networks using vector quantization," *arXiv preprint :1412.6115*, 2014.

[11] M. Hosseini *et al.*, "Minimizing classification energy of binarized neural network inference for wearable devices," in *2019 20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2019.

[12] A. Jafari *et al.*, "Binmac: Binarized neural network manycore accelerato," in *ACM Proceedings of the 28th Edition of the Great Lakes Symposium on VLSI*. ACM, 2018.

[13] Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[14] S. Han and others., "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[15] T. Zhang *et al.*, "Adam-admm: A unified, systematic framework of structured weight pruning for dnns," *arXiv preprint arXiv:1807.11091*, 2018.

[16] C. Deng *et al.*, "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 189–202.

[17] M. Hosseini, M. Horton *et al.*, "On the complexity reduction of dense layers from $\boldsymbol{o(n^2)}$ to $\boldsymbol{o(nlogn)}$ with cyclic sparsely connected layers," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019.

[18] S. Han, Liu *et al.*, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[19] Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[20] A. Reiss *et al.*, "Introducing a new benchmarked dataset for activity monitoring," in *Wearable Computers (ISWC), 2012 16th International Symposium on*. IEEE, 2012.

[21] A. Mirzaeian *et al.*, "Nesta: Hamming weight compression-based neural proc. engine," *arXiv preprint arXiv:1910.00700*, 2019.

[22] C. Shea and T. Mohsenin, "Heterogeneous scheduling of deep neural networks for low-power real-time designs," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 4, pp. 1–31, 2019.

[23] F. Behnia, A. Mirzaeian *et al.*, "Code-bridged classifier (cbc): A low or negative overhead defense for making a cnn classifier robust against adversarial attacks," *arXiv preprint arXiv:2001.06099*, 2020.

[24] M. Malik *et al.*, "Ecost: Energy-efficient co-locating and self-tuning mapreduce applications," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[25] H. Sayadi *et al.*, "2smart: A two-stage machine learning-based approach for run-time specialized hardware-assisted malware detection," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 728–733.

[26] K. Neshatpour *et al.*, "Energy-efficient acceleration of mapreduce applications using fpgas," *Journal of Parallel and Distributed Computing*, vol. 119, pp. 1–17, 2018.

[27] B. Prakash *et al.*, "Guiding safe reinforcement learning policies using structured language constraints," in *SafeAI workshop Thirty-Fourth AAAI Conference on Artificial Intelligence*. AAAI, 2020.

[28] B. Prakash, M. Horton *et al.*, "On the use of deep autoencoders for efficient embedded reinforcement learning," in *ACM Proceedings of the 29th Edition of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2019.

[29] B. Prakash *et al.*, "Improving safety in reinforcement learning using model-based architectures and human intervention," in *The 32nd International Conference of the Florida Artificial Intelligence Society*. AAAI, 2019.