2021 31st International Conference on Field-Programmable Logic and Applications (FPL) | 978-1-6654-3759-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/FPL53798.2021.00059

Accelerating Continual Learning on Edge FPGA

Duvindu Piyasena*, Siew-Kei Lam† and Meiqing Wu‡
Nanyang Technological University, Singapore
{*gpiyasena, †assklam,‡meiqingwu}@ntu.edu.sg

***Abstract*—Real-time edge AI systems operating in dynamic environments must learn quickly from streaming input samples without needing to undergo offline model training. We propose an FPGA accelerator for continual learning based on streaming linear discriminant analysis (SLDA), which is capable of class-incremental object classification. The proposed SLDA accelerator employs application-specific parallelism, efficient data reuse, resource sharing, and approximate computing to achieve high performance and power efficiency. Additionally, we introduce a new variant of SLDA and discuss the accuracy-efficiency trade-offs. The proposed SLDA accelerator is combined with a Convolutional Neural Network (CNN). which is implemented on Xilinx DPU to achieve full continual learning capability at nearly the same latency as inference. Experiments based on popular datasets for continual learning, CoRE50 and CUB200, demonstrate that the proposed SLDA accelerator outperforms the embedded CPU and GPU counterparts, in terms of speed and energy efficiency.**

I. Introduction

FPGAs have demonstrated significant success in accelerating Convolutional Neural Networks (CNNs) at the edge, due to their ability to meet real-time performance requirements at extremely high energy efficiency [1]–[3]. However, the majority of edge CNN accelerators focus only on inference task and hence, they lack the ability to learn and adapt to dynamic environments. This capability is essential in autonomous robots, drones and self-driving cars, where the deep learning models are likely to encounter new scenarios which were not present in the training dataset.

The conventional CNN training based on backpropagation has high computation and memory requirements in comparison to inference. Due to the tight resource constraints on edge accelerators, CNNs are usually trained first on a server, typically on a GPU or a specialized accelerator such as Google TPU [4], and deployed on the edge accelerator for inference. Hence, to continuously accumulate knowledge, the edge accelerator would require constant data transfer and model retrieval from a remote server. This mode of training is ill-suited for applications that require quick adaptation to new knowledge, due to the round-trip communication overhead. Also, the edge device may need to operate offline if constant network connectivity cannot be guaranteed. Additionally, some applications have strict data privacy constraints, where sharing data with a remote server is not possible.

Furthermore, it is extremely challenging for edge devices to learn continuously, as conventional Deep Neural Networks (DNNs) suffer from *catastrophic forgetting*, a phenomenon where learning new knowledge leads to loss of previous knowledge [5]. As such, they need to be retrained with the entire dataset to add new knowledge, which is slow and incur large amount of storage on the edge device.

Continual learning[1] is an actively researched area to allow deep learning models to acquire new knowledge continuously while preventing catastrophic forgetting [6]–[9]. In this paper, we propose an edge accelerator for continual learning on FPGA, based on a recently proposed model, *Deep SLDA*, where a SLDA classifier performs incremental training at the last layer of a CNN [10]. In particular, the contributions of this paper are as follows:

1) We present an on-chip continual learning system that combines a novel edge accelerator for SLDA with Xilinx DPU [11]. To the best of our knowledge, this is the first full continual learning system that has been demonstrated on FPGA.
2) We introduce a novel variant of SLDA that has extremely low compute and memory requirements, which leads to good accuracy trade-off.
3) We present custom optimization strategies for the SLDA architecture, which include approximate computing, memory organization, on-chip data reuse, and resource sharing to reduce latency, resources and power. Experimental results show that the proposed architecture undergoes learning at nearly the same latency as inference, and significantly outperforms embedded CPU and GPU.

II. Related Work

A. Continual Learning

The main challenge for continual learning in DNNs is catastrophic forgetting of existing knowledge [5]. To mitigate this, existing works use various techniques [6]–[9] which can be broadly classified as regularization of model parameters [12]–[14], dynamic architectures with parameter isolation [15]–[19], and replay of past samples [18], [20]–[23]. These works consider continual learning as either incremental batch learning or online/streaming learning.

In class-incremental continual learning, an incremental batch learning model learns a new class using a batch of data [12]–[14], [20]. In contrast, a streaming learning model has to learn from data streams as they arrive and may use a sample only once [10], [21]–[23]. In comparison to incremental batch learning, streaming learning enables faster learning, and has low computing and memory overhead. Hence, models capable of streaming learning is ideal for on-device learning in resource constrained edge AI systems. The edge accelerator

[1]The terms *lifelong/incremental learning* are also used in literature.

978-1-6654-3759-2/21/$31.00 ©2021 IEEE
DOI 10.1109/FPL53798.2021.00059

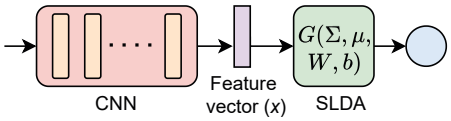


Fig. 1: Deep SLDA model [10]

in our work is based on deep SLDA, a recently proposed streaming learning method for continual learning [10].

B. Offline On-Chip Learning

As mentioned earlier, conventional CNN training based on backpropagation is highly computationally and memory intensive in comparison to inference, and is typically performed using GPUs or ASICs. A popular ASIC for CNN training is the Google's Tensor Processing Unit (TPU) device, which a multi-chip platform [4]. Existing works based on FPGAs rely on multi-chip platforms to handle larger batch sizes and state-of-the-art deep models [24], [25]. However, such multi-chip platforms are more suited to data-center applications rather than for edge computing.

C. On-Chip Continual Learning

Typically in edge AI deployments, the CNN models are initially trained on a large representative dataset. This knowledge could be utilized to train only the last layer(s) of the CNN model, when learning new classes [10], [17], [21], [26]. In comparison to training the entire model, this strategy is more amenable for edge implementation, and is adopted by our work. Similar to our work, [26]–[28] presented FPGA-based hardware accelerators for class-incremental continual learning classifiers that leverage transfer learning using a pre-trained CNN. The works in [27], [28] propose incrementally learning classifiers based on product quantization [29]. However, these works only demonstrated the results for small datasets (CIFAR-10) and the scalability to larger datasets is not discussed. The work in [26] propose an accelerator for a self-growing neural network which grows dynamically. However, the dynamic growth likely makes the model less scalable to a larger number of classes. Additionally, the above works do not implement a full continual learning system in hardware, which will necessitate the integration with a CNN feature extractor.

III. Continual learning model

The continual learning model used in our work is based on the *Deep SLDA* model [10]. The model achieves high accuracy, with high efficiency and scalability, which makes it amenable for resource constrained edge devices [30]. As shown in Fig. 1, the Deep SLDA model combines a CNN as a feature extractor and the SLDA as a classifier, with SLDA replacing the last Fully-Connected (FC) layer. As the model learns new classes, SLDA incrementally adapts its parameters to add new knowledge, while the convolutional (CONV) layers remain frozen. SLDA relies on the CONV layers of a CNN trained on a large dataset (Imagenet [31]) to act as a generic feature extractor [32]. We have used the Resnet18 CNN model [33] pre-trained on Imagenet dataset, where features extracted from *Avg Pool* (dimension (D) 512), is fed into the SLDA.

A. SLDA algorithm

The SLDA learns the distribution of embedding feature space, by updating the mean feature of each class (μ) and a shared covariance matrix (Σ) during training. During inference, features are classified by assigning them to the nearest Gaussian, which takes the form of a linear classifier (Eq. 7).

The computations of SLDA obtained from [10] are as follows. During training, as a new input feature belonging to class k ($x_t \in \mathrm{R}^{D\times 1}$) arrives, μ ($\in \mathrm{R}^{D\times N_c}$) and Σ ($\in \mathrm{R}^{D\times D}$) are updated as follows,

$$z_t = x_t - \mu_k \tag{1}$$

$$\Sigma_{t+1} = (t/t+1)\cdot\left[\Sigma_t + (1/t+1)\cdot z_t.z_t^T\right] \tag{2}$$

$$\mu_{k,t+1} = \left[c_k\cdot\mu_{k,t} + x_t\right]/(c_k+1) \tag{3}$$

where N_c is the current number of classes, t is the total number of encountered training samples, while c_k is the number of encountered samples from class k. To derive the linear classifier for inference, μ and Σ are converted into weights ($W \in \mathrm{R}^{D\times N_c}$) and bias ($b \in \mathrm{R}^{1\times N_c}$) as follows.

$$\Lambda = \left[(1-\epsilon)\cdot\Sigma + \epsilon\cdot I\right]^{-1} \tag{4}$$

$$W = \Lambda\mu \tag{5}$$

$$b = -0.5\cdot(\mu\cdot W) \tag{6}$$

where ϵ is shrinkage parameter, and $I \in \mathrm{R}^{D\times D}$ is the identity matrix. During inference, the classification is done as follows,

$$scores_t = W^T x_t + b^T \tag{7}$$

$$\widehat{y_t} = argmax(scores_t) \tag{8}$$

The work in [10] proposes two variants of SLDA, a) *SLDAPlasticCov*, where the covariance gets updated for each training sample, and b) *SLDAStaticCov*, where the covariance is initialized during a base initialization step and remains static throughout. The latter has lesser accuracy, but has extremely lower computational and memory requirements.

B. Proposed SLDA variant

We propose a new variant of SLDA, *SLDADiagCov*, which updates only the diagonal of the Σ in Eq. 2. In other words, *SLDADiagCov*, learns the variance in each dimension, and ignores the covariance among dimensions of the embedding feature space. This simple approximation leads to a best-case memory savings of $D\cdot(D-1)/2$ parameters, and savings of $3\cdot D\cdot(D-1)/2$ multiplications and $D\cdot(D-1)/2$ additions (Eq. 2), compared to *SLDAPlasticCov*. The experimental results show that the *SLDADiagCov* provides good accuracy-efficiency trade-offs (see Section V, Table I and III).

IV. Proposed Hardware Design

A. Overview

An overview of the proposed hardware architecture is shown in Fig. 2, which consists of 3 main units *a) Compute*, *b) Controller* and *c) Scratchpad Memory*. The compute unit contains a *General Matrix Multiplication (GEMM) core* and a *Vector processing unit (VPU)*, to support matrix and vector computations of all SLDA variants. The scratchpad memory contains on-chip memory organized into two levels (*L1* and *L2*), for DRAM access buffering and on-chip caching.

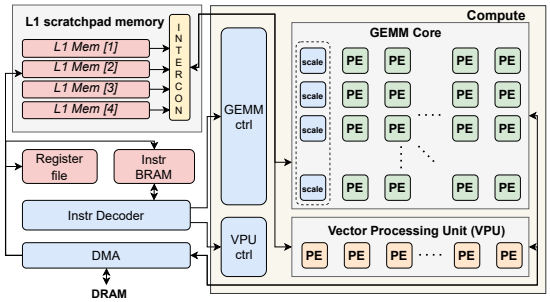
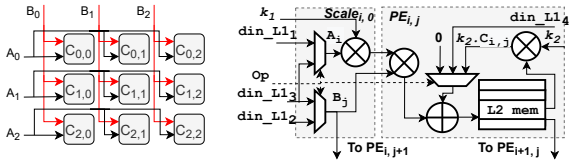


Fig. 2: Hardware architecture



(a) Dataflow (b) Processing element

Fig. 3: GEMM core

B. Compute units

1) GEMM Core: The GEMM core supports general matrix multiply and accumulation as expressed in the form,

$$C = k_1 \cdot A * B + k_2 \cdot C \tag{9}$$

where k_1, k_2 are scaling constants, and A, B, C are matrices.

The GEMM core implements the SLDA operations, by performing covariance update during training (Eq. 2), and labeling score calculation during inference (Eq. 7). These two operations require support for vector outer product and accumulation, and matrix-vector multiplication respectively.

As shown in Fig. 3b, the GEMM core consists of a $T \times T$ array of processing elements (PE). This allows the core to operate on a $T \times T$ matrix tile at a time. For matrices whose dimensions are larger than T, the hardware supports tiled operations, wherein the output tiles are computed sequentially as elaborated in Listing 1. The outer loop traverses over output tiles, while the inner loop describes the computations within a single tile. The matrix dimensions and shapes are run-time configurable via instructions, which provide the flexibility to *a)* support different types of matrix operations required for SLDA operations (Eq. 2 and 7), and *b)* support embedding features from different CNN backbone networks.

```
//Tile outer loop (<D, D> : T_m = T_p = D/T)
for(int m=0; m<T_m; m++){
  int p_start = (is_triangular)? m : 0;
  int p_end   = (is_diagonal) ? m+1 : T_p;
  for(int p=p_start; p<p_end; p++){
    for(int n=0; n<n1; n++){
//Tile inner loop
      for(int i=0; i<n2; i++){
        for(int j=0; j<T; j++){
          #ifdef SLDA_PLASTIC_COV
            for(int k=0; k<T; k++){
          #elif
            int k=j;
          #endif
          //[Processing Element logic]
```

Listing 1: GEMM core loop

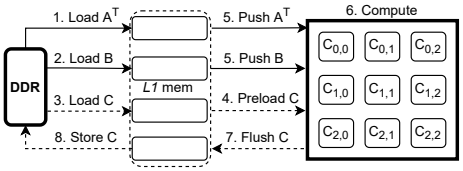


Fig. 4: GEMM core tile operation

The GEMM core follows Output Stationary (OS) dataflow [34], where each PE accumulates a single element of the output tile, while the inputs *A, B* are broadcast into each PE as shown in Fig. 3a. The inputs are routed from *L1* scratchpad memory, and can be switched between multiple banks via instruction control for flexibility. Each PE contains two multipliers and an adder (see Fig. 3b). In relation to Eq. 9, two multipliers perform input multiplication ($k_1.A_i * B_j, i \leq T, j \leq T$) and accumulation scaling ($k_2 * C_{i,j}$), while the adder performs the product accumulation. The *L2* scratchpad memory inside the PE facilitates multiple partial accumulations for future reuse, to minimize off-chip DRAM access.

The operation of a single output matrix tile in GEMM core in the general case is shown in Fig. 4. The DRAM accesses, pre-loading/flushing accumulations, shown in dotted lines are configurable during run-time and could be skipped to give better performance. Details on how SLDA application-specific optimizations leverage this configurability of GEMM core will be described in subsection IV-D.

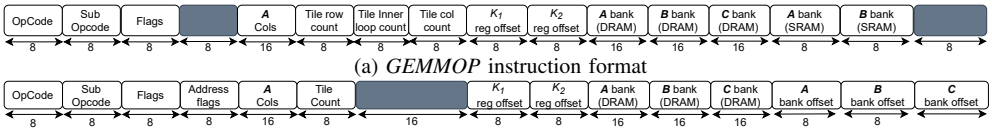
2) Vector Processing unit: The VPU performs element-wise vector addition, subtraction and scaling to support operations in Eq. 1 and 3. VPU contains T processing elements, and utilizes tiled operations to support generic vector sizes.

C. Instruction Set Architecture (ISA)

The *GEMM* and the *VPU* cores are controlled by an instruction set comprising of two high-level instructions *GEMMOP* and *VECOP* for the two cores respectively, whose formats are shown in Fig. 5. The instructions specify a two level opcode, address modes, memory offsets (*Reg, Scratchpad and DRAM*) and control flags to enable/disable input/accum and load/store operations. The instruction control provides the flexibility to *a)* reuse same resources for different operations (inference/-training) by controlling the loop bounds to handle various matrix dimensions and shapes, *b)* access different locations in DRAM, c) switch between local scratchpad memory banks for data reuse, and *d)* enable/disable DRAM access for optimal performance. The instructions are stored in a separate on-chip memory and during runtime, the instruction decoder executes instructions in order, where instructions are converted into control signals and address information.

D. Hardware Execution of SLDA

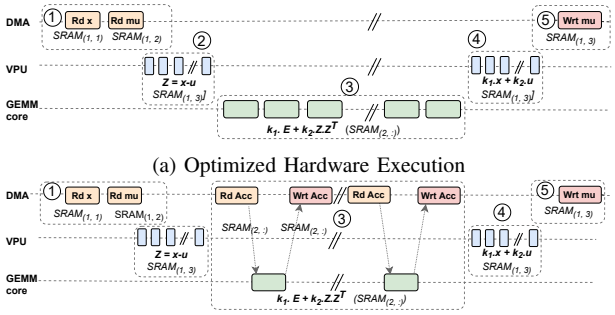
As discussed in Section III-A, SLDA model has 3 phases of computation: *a)* Updating mean (μ) and covariance ((Σ)) per each train sample (Eq. 1, 2 and 3), *b)* Deriving inference weights (W) and bias (b) (Eq. 4, 5 and 6), and *c)* Running inference for each test sample (Eq. 7 and 8). While steps *a)* and *c)* are accelerated on the proposed hardware architecture,



(a) *GEMMOP* instruction format

(b) *VECOP* instruction format

Fig. 5: Instruction set formats



(a) Optimized Hardware Execution

(b) Hardware execution with memory read/write

* In $SRAM_{[x,y]}$, x = scratchpad memory level, y = bank respectively

Fig. 6: Hardware Execution of SLDA training steps per sample

step *b)* is performed on the host (ARM *PS* on Zynq MPSoC). For experimental evaluations, steps *a)* and *b)* are undertaken as distinct phases sequentially one after another.

The execution steps of mean (μ) and covariance (Σ) updates (Eq. 1, 2 and 3) in the proposed hardware is shown in Fig. 6a. The mean of the training sample class (μ_k) and input feature vector (x) is read from DRAM into *L1* memory and the feature vector is centered using the VPU (Eq. 1), whose output is used by the GEMM core to perform the Σ update (Eq. 2). Finally, μ_k is updated and written to DRAM via *L1* memory (Eq. 3). For *SLDAStaticCov*, only the μ_k update is performed.

For the first sample of a training phase, covariance has to be loaded from DRAM to *L2* memory, per each tile of computation. For the last sample of a training phase, it has to be flushed back into DRAM (Fig. 6b). This loading and flushing accumulations is controlled by instruction flags. At the beginning of an inference phase, the host computes the Weights (W) and Bias (b) (Eq. 4, 5 and 6), and the bias is transferred from DRAM into *L2* memory of GEMM core. For each testing sample, input (x) and Weights (W) are loaded into the *L1* memory from DRAM and the GEMM core computes matrix vector product (Eq. 7) in a tiled approach. The final scores are transferred to host for label prediction (Eq. 8).

E. Hardware Optimizations

1) Tile skipping: The SLDA covariance (Σ) is a symmetric matrix. For *SLDAPlasticCov*, we exploit this to compute only the upper triangular portion of Σ in order to achieve compute and memory savings. The skipping is done at tile granularity. Thus, given Σ is of dimension $D \times D$, this leads to a saving of on-chip storage and latency of operation in Eq. 2 by a factor of $T_D.(T_D - 1)/2$, where $T_D = D/T$ (T is Tile size).

2) Fixed-Point Quantization: To achieve higher efficiency in the hardware design, we adopt a fixed-point data representation scheme in place of floating-point for lower latency arithmetic, reduced resource, and power consumption [35].

To identify the resiliency of SLDA to quantization error in fixed-point mapping of model parameters/inputs, a sensitivity analysis was performed. The parameters/inputs were put into groups sharing the same arithmetic resources in hardware, and the precision requirement of each group was evaluated by measuring accuracy at different levels of precision. The most optimal precision of each group without loss of accuracy was selected for hardware design. Here, the integer bit allocation of each group is predetermined to satisfy the range requirement, and only the fractional bits are varied. The results of this analysis are discussed in the next section.

3) Accuracy vs. Efficiency trade off: For *SLDADiagCov* and *SLDAStaticCov*, the PE array in GEMM core is flattened to a single column (listing 1:line 13), since only the diagonal of PE array is utilized during *SLDADiagCov* training, while during inference only a single row is utilized across all variants. This pre-synthesis optimization results in removal of $T \cdot (T-1)$ GEMM core PEs, leading to resource and power savings.

4) Scratchpad memory design: The scratchpad memory based on on-chip memory is used for buffering, and caching data and intermediate results to minimize DRAM access. This scratchpad memory is split into two levels (*L1* and *L2*), and organized to maximize memory bandwidth to compute units.

- *L1 scratchpad memory*
  The *L1* memory acts as a buffer between DRAM and compute units and also for caching intermediate results. As shown in Fig. 2, the L1 memory is split into multiple banks with the flexibility to configure the input and output banks at instruction level. This allows sufficient on-chip caching opportunities. The banks are split into distinct BRAM units to reduce multiplexing. Fig. 6 shows the bank selection during each step of training. The x and u loaded to bank 1 and 2 during Eq. 1 is reused for Eq. 3, while VPU output from step Eq. 1 stored in bank 3 is reused by GEMM core in Eq. 2. During inference, GEMM core uses bank 1 and 2 for w and x, while output is written to DRAM via bank 4.
- *L2 scratchpad memory*
  L2 memory is distributed across PEs in the GEMM core and is used for caching accumulations for reuse. During training, memory unit of each PE stores overlapping elements of covariance tiles. Similarly during inference, overlapping bias elements are stored in each PE. This eliminates the need for accumulator flushing and reading to and from the DRAM between two output tiles as shown in Fig. 6b), which leads to significant latency savings.

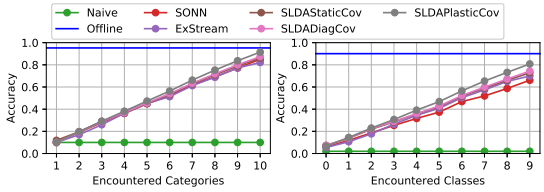


Fig. 7: Top-1 accuracy vs encountered classes (*CoRE50*)

TABLE I: Model comparison

	Top-1 Accuracy		Peak Memory consumption (MB)			Train time (s)
	Object	Category	Params	Data	Param + Data	
			CoRE50			
Naive[a]	2	10.09	-	-	-	-
Offline[b]	90.08	95.24	-	-	-	-
ExStream	71.49	83.76	2.98	1.56	4.54	524.24
SONN	66.12	85.01	1.60	0.00	1.60	391.78
SLDAStaticCov	72.78	86.12	0.20	0.00	0.20	224.28
SLDADiagCov	74.34	87.41	0.20	0.00	0.20	265.06
SLDAPlasticCov	80.79	91.46	1.20	0.00	1.20	285.15
			CUB200			
Naive[a]		0.67	-	-	-	-
Offline[b]		60.54	-	-	-	-
ExStream		52.46	2.55	6.25	8.80	526.31
SONN		44.85	1.87	0.00	1.87	89.10
SLDAStaticCov		58.26	0.78	0.00	0.78	75.52
SLDADiagCov		61.11	0.79	0.00	0.79	78.21
SLDAPlasticCov		62.98	1.78	0.00	1.78	88.72

[a] *epochs*=5, [b] *epochs:CoRE50*=20, *CUB-200*=250, [a],[b] *lr*=0.001

V. Experimental Results

A. Datasets

For the experiments we use two different datasets:

1) CoRE50 [36]: A popular benchmark dataset for continuous object recognition based on video sequences recorded at 20 fps. The dataset consists of 50 object classes belonging to 10 categories. We used the train/test split suggested in [36]. The images were sampled at 4 fps for training and 1 fps for testing as the consecutive images are similar, resulting in 480 and 50 images per class for training and testing respectively.

2) CUB200-2011 [37]: Caltech-UCSD Birds 200 is a dataset used for fine-grained classification evaluations consisting of 200 bird species with 44-60 images per class. We used 30 images per class for training and the rest for testing.

B. Baseline Models

The following baselines were used to benchmark SLDA performance. As a naive approach to class-incremental learning, a Resnet18 CNN was trained sequentially on batches each containing a unique class, using mini-batch gradient descent (*Naive*). To estimate the upper bound accuracy, a Resnet18 CNN was trained offline with the entire dataset without a class specific split (*Offline*). Further, the model is evaluated with two continual learning methods capable of streaming learning: a) *Exstream* [21], which only trains the last FC layer and prevents catastrophic forgetting of past classes via sample replay, and b) *SONN* [26], which tackles forgetting by employing a self-organizing neural network in place of the FC layer.

C. Model Comparison

To evaluate the feasibility of SLDA for continual learning on the edge, we evaluated its accuracy, memory consumption, training time, and scalability based on models running on PyTorch [38]. The memory requirement was calculated from the parameters that undergo continuous training and the data storage explicitly required for continual learning (e.g., replay samples), while the total training time was measured on Nvidia GTX1080 GPU. The results of this evaluation along with comparison with above baselines is summarised in Table I. Fig. 7 shows the variations in top-1 accuracy (on entire validation set) as classes are learnt sequentially for the CoRE50 dataset.

The accuracy results show that continual learning models prevent forgetting in various degrees, in comparison to the *Naive* approach. All SLDA variants outperform baseline continual learning models, with *SLDAPlasticCov* displaying the highest accuracy, while the proposed *SLDADiagCov* and *SLDAStaticCov* exhibit the highest efficiency. Additionally, all SLDA variants are extremely scalable, as learning a new class only increases memory consumption by 4KB, and has no impact on training latency. The high efficiency and scalability makes SLDA more amenable for edge implementation. While *SLDAPlasticCov* is the best choice for applications with high accuracy requirements, proposed *SLDADiagCov* can be used as an alternative in resource-constrained devices deployed in applications where accuracy is not a critical requirement.

D. Fixed-Point Quantization

To identify the optimal fixed-point representation of inputs/-parameters, a sensitivity analysis was performed as described in Section IV-E2, where inputs/parameters were grouped in a hardware-aware manner, and precision of each group was tested. While one group was being tested, the other was kept at 32 bits. The results of this analysis is listed in Table III, along with the groupings. For simplicity in hardware design, we only explored schemes where the total bit widths are multiples of 8. It can be observed that the two groups exhibit varying precision requirements. On both datasets, *Group1* requires at least 16 bits while *Group2* requires 32 bits. This observation implies that a mixed-precision fixed-point design is necessary to achieve the most efficient and accurate hardware design.

E. Hardware Evaluation

The proposed hardware was implemented with Xilinx Vitis HLS C++ on Vivado 2020.2 targeting Xilinx ZCU102 kit (ZCU9EG) at 200Mhz. A Linux device driver was developed for memory mapping the I/O and a DMA memory region (for storing μ, Σ, W, b and transferring x), while a host application controls hardware execution and performs Eq. 5 and 6.

Table III shows the resource, latency and power results of the implemented hardware with comparison of fixed-point and floating-point designs for all SLDA variants. The tile size (T) is set to 16, such that the *GEMM* core contains a 16*16 PE grid, while the *VPU* contains 16 PEs. The power is measured from on-board power rails using ZCU102 Board interface tool [39], while latency is measured from the host application.

The results indicate that fixed-point design leads to at least 40% of DSP and FF and 30% LUT savings across all variants. Additionally, for *SLDAPlasticCov*, the latency and power reduce approximately by 60% and 20% respectively. However, significant power savings cannot be observed for

TABLE II: Fixed-point sensitivity evaluation

		SLDAPlasticCov			*SLDAStaticCov*			*SLDADiagCov*		
		CoRE50 (obj)	CoRE50 (cat)	CUB200	CoRE50 (obj)	CoRE50 (cat)	CUB200	CoRE50 (obj)	CoRE50 (cat)	CUB200
Baseline	FP32	80.79	91.46	61.95	72.88	86.02	58.72	74.34	87.14	60.73
	FxP<32, 4, 28>	81.01	91.55	61.82	72.83	86.08	58.78	74.38	87.32	60.72
Group1	FxP<24, 4, 20>	81.01	91.55	61.82	72.83	86.08	58.78	74.38	87.32	60.72
(Activation/	FxP<16, 4, 12>	81.05	91.64	61.69	72.87	86.08	58.85	74.38	87.32	60.91
Mean/ Weights)	FxP<8, 4, 4>	34.77	49.36	14.60	39.48	70.34	35.79	27.97	54.33	17.38
Group2	FxP<24, 4, 20>	3.82	12.74	61.43	72.83	86.08	58.79	74.39	87.33	60.66
(Covariance/	FxP<16, 4, 12>	0.84	6.76	2.90	72.79	86.08	58.85	74.57	87.28	60.40
Bias)	FxP<8, 4, 4>	2.05	12.14	2.19	72.43	86.31	59.37	2.00	10.00	0.52

FxP<N, I, F>, N= Bit-width, I=No. of integer bits, F=No. of fractional bits,

TABLE III: Hardware performance impact due to quantization

	LUT	FF	DSP	BRAM	Latency (*us*)	Power (W)
SONN [26]	115,656	104,680	1,184	415.50	4.35	-
SLDAPlasticCov						
FP32	133,794	226,972	2,273	295	141.14	4.77/5.04
FxP<16, 32>	73,435	83,485	914	317.50	47.85	3.61/3.97
SLDADiagCov						
FP32	24,856	43,355	353.00	55.00	45.95	3.26/3.34
FxP<16, 32>	18,068	25,283	194	69.5	8.3	3.16/3.34
SLDAStaticCov						
FP32	24,856	43,355	353	55.00	38.15	3.26/3.37
FxP<16, 32>	18,068	25,283	194	69.5	4.76	3.16/3.34

* *FxP*<N_1, N_2>, N_1=bit-width of Group1, N_2=bit-width of Group2

TABLE IV: Hardware execution comparison

	CPU	GPU*	FPGA
SLDAPlasticCov			
Av. latency (us)	1,159.21	182.01	47.85
Latency factor	**24.23**	**3.80**	**1.00**
Power (static/total) (W)	3.6 / 4.04	3.23 / 6.47	3.61 / 3.97
Energy/ train sample (J)	4.68E-03	1.18E-03	1.90E-04
Energy factor	**24.62**	**6.20**	**1.00**
SLDADiagCov			
Av. latency (us)	25.69	98.25	8.30
Latency factor	**3.09**	**11.84**	**1.00**
Power (static/total) (W)	3.16 / 3.51	3.23 / 4.34	3.16 / 3.34
Energy/ train sample (J)	9.02E-05	4.26E-04	2.77E-05
Energy factor	**3.25**	**15.38**	**1.00**
SLDAStaticCov			
Av. latency (us)	2.80	76.07	4.76
Latency factor	**0.59**	**15.98**	**1.00**
Power (static/total) (W)	3.16 / 3.29	3.23 / 3.49	3.16 / 3.34
Energy/ train sample (J)	9.21E-06	2.65E-04	1.59E-05
Energy factor	**0.58**	**16.69**	**1.00**

* 15W, 6 core mode

SLDADiagCov and *SLDAStaticCov*, due to their lower resource utilization and compute complexity. The results show that the proposed *SLDADiagCov* is equivalent to *SLDAStaticCov* in terms of hardware resources and power while being more accurate. Additionally, fixed-point designs of all SLDA variants are significantly more resource efficient than SONN, and achieves higher accuracy across both datasets as shown in Table I.

The SLDA variants were also implemented on embedded CPU (ARM A53 on ZCU102) and GPU (Nvidia Jetson Xavier NX). The CPU implementations were developed using C++, based on the Eigen linear algebra library [40], while a combination of cuBLAS library [41] and custom CUDA kernels were used for the GPU. Specifically, for the upper triangular covariance update (Eq. 2), BLAS routine symmetric rank1 operation (*SSYR*) was used. The O2 optimization level was used for compilation. Table IV shows the average latency and energy comparison among the devices, observed from 24000 training samples from CoRE50 dataset. FPGA significantly outperforms GPU across all SLDA variants, and

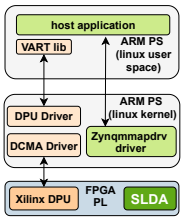


Fig. 8: Platform

	*DPU**	*DPU** + *SLDAPlasticCov*	*DPU** + *SLDADiagCov*
LUT	58,703	142,215	90,059
FF	106,024	210,541	138,076
DSP	704	1616	896
BRAM	261	553.5	313.5
Power (W)	4.42 /	5.29 /	4.81/
(Static/Total)	8.16	9.00	8.69
Latency(us)	5,875.44	5,929.00	5,883.70
FPS	170.20	168.67	169.96
Energy/Image (J)	0.048	0.056	0.054
Accuracy (Obj/Cat)	NA	78.79 / 89.46	71.81 / 85.37

* B4096, no. of cores= 1

TABLE V: HW results

Fig. 9: Xilinx DPU + SLDA results

ARM CPU for *SLDAPlasticCov* and *SLDADiagCov*, in terms of both latency and energy. This is a result of the application-specific parallelism and pipelining in the PEs, custom memory organization for optimal data access and the mixed-precision arithmetic units. This demonstrates the proposed FPGA accelerator is a better choice for continual learning at the edge compared to the CPU and GPU counterparts.

To demonstrate the full on-chip continual learning system, the proposed accelerator was combined with Xilinx DPU [11] that implements the CNN, as shown in the Fig. 8. Vitis AI v1.3 was used for quantizing/compiling a pre-trained Resnet18 CNN on DPU. At runtime, DPU extracts features from each image, which gets transferred to the SLDA accelerator via ARM host with an approximate delay of 600*ns*. Table V compares the hardware resource usage and runtime results of the continual learning systems (*DPU* + *SLDAPlasticCov*, *DPU* + *SLDADiagCov*) with an inference only design (*DPU*). The results demonstrate that learning can be achieved at almost similar latency as inference, with high energy efficiency.

VI. Conclusion

We proposed an FPGA edge accelerator for continual learning based on SLDA. Custom optimization strategies were introduced that led to significant savings in latency, resource, and power consumption. A novel SLDA variant was proposed to achieve good hardware efficiency with accuracy trade-offs. Additionally, we demonstrated the proposed accelerator can be combined with a CNN accelerator for on-chip full continual learning with high computational and energy efficiency.

VII. Acknowledgement

This work was supported by NTUitive Gap Fund (NGF-2020-09-028); National Research Foundation Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) Programme with the Technical University of Munich at TUMCREATE.

REFERENCES

[1] Y. LeCun, "1.1 deep learning hardware: Past, present, and future," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 12–19.

[2] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[3] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 5–14.

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and et al., "In-datacenter performance analysis of a tensor processing unit," vol. 45, no. 2, 2017.

[5] "Catastrophic interference in connectionist networks: The sequential learning problem," ser. Psychology of Learning and Motivation, G. H. Bower, Ed. Academic Press, 1989, vol. 24, pp. 109 – 165.

[6] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54–71, 2019.

[7] "Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges," *Information Fusion*, vol. 58, pp. 52–68, 2020.

[8] M. Delange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, "A continual learning survey: Defying forgetting in classification tasks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[9] R. Hadsell, D. Rao, A. A. Rusu, and R. Pascanu, "Embracing change: Continual learning in deep neural networks," *Trends in Cognitive Sciences*, 2020.

[10] T. L. Hayes and C. Kanan, "Lifelong machine learning with deep streaming linear discriminant analysis," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, Conference Proceedings, pp. 220–221.

[11] Xilinx. (2021, February) Zynq dpu v3.3. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_3/pg338dpu.pdf

[12] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," vol. 114, no. 13, pp. 3521–3526, 2017.

[13] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *ICML*, 2017.

[14] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 12, pp. 2935–2947, Dec 2018.

[15] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," 2016.

[16] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," *arXiv preprint arXiv:1708.01547*, 2017.

[17] G. I. Parisi, J. Tani, C. Weber, and S. Wermter, "Lifelong learning of spatiotemporal representations with dual-memory recurrent self-organization," *Frontiers in neurorobotics*, vol. 12, p. 78, 2018.

[18] R. Kemker and C. Kanan, "Fearnet: Brain-inspired model for incremental learning," *arXiv preprint arXiv:1711.10563*, 2017.

[19] A. Gepperth and C. Karaoguz, "A bio-inspired incremental learning architecture for applied perceptual problems," *Cognitive Computation*, vol. 8, no. 5, pp. 924–934, 2016.

[20] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, "icarl: Incremental classifier and representation learning," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2017, pp. 2001–2010.

[21] T. L. Hayes, N. D. Cahill, and C. Kanan, "Memory efficient experience replay for streaming learning," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, Conference Proceedings, pp. 9769–9776.

[22] D. Lopez-Paz and M. Ranzato, "Gradient episodic memory for continual learning," in *Advances in neural information processing systems*, Conference Proceedings, pp. 6467–6476.

[23] A. Chaudhry, M. Ranzato, M. Rohrbach, and M. Elhoseiny, "Efficient lifelong learning with a-gem," *arXiv preprint arXiv:1812.00420*, 2018.

[24] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt, "Fpdeep: Acceleration and load balancing of cnn training on fpga clusters," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 81–84.

[25] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2016, pp. 107–114.

[26] D. Piyasena, M. Thathsara, S. Kanagarajah, S. K. Lam, and M. Wu, "Dynamically growing neural network architecture for lifelong deep learning on the edge," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 262–268.

[27] G. B. Hacene, V. Gripon, N. Farrugia, M. Arzel, and M. Jezequel, "Budget restricted incremental learning with pre-trained convolutional neural networks and binary associative memories," *Journal of Signal Processing Systems*, vol. 91, no. 9, pp. 1063–1073, 2019.

[28] ——, "Efficient hardware implementation of incremental learning and inference on chip," 2019.

[29] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[30] E. Belouadah, A. Popescu, and I. Kanellos, "A comprehensive study of class incremental learning algorithms for visual tasks," *Neural Networks*, vol. 135, pp. 38–54, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608020304202

[31] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[32] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: An astounding baseline for recognition," in *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, June 2014, pp. 512–519.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[34] 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, 2017.

[35] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.

[36] V. Lomonaco and D. Maltoni, "Core50: a new dataset and benchmark for continuous object recognition," *arXiv preprint arXiv:1705.03550*, 2017.

[37] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona, "Caltech-UCSD Birds 200," California Institute of Technology, Tech. Rep. CNS-TR-2010-001, 2010.

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[39] Xilinx. (2019, July) Zcu102 board interface test. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/2019_1/xtp428-zcu102-bit-c-2019-1.pdf

[40] (2020, Dec) Eigen c++ template library for linear algebra. [Online]. Available: https://eigen.tuxfamily.org/index.php?title=Main_PageCredits

[41] Nvidia. (2020, Dec) cublas. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html