IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 5, 2022, accepted January 21, 2022, date of publication January 28, 2022, date of current version February 22, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3147674

Deep Reinforcement Learning Acceleration for Real-Time Edge Computing Mixed Integer Programming Problems

GERASIMOS GEROGIANNIS[1,2], MICHAEL BIRBAS[1], AIMILIOS LEFTHERIOTIS[1], ELEFTHERIOS MYLONAS[1], NIKOLAOS TZANIS[1], AND ALEXIOS BIRBAS[1]

[1]Department of Electrical and Computer Engineering, University of Patras, 26500 Patras, Greece
[2]Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, Urbana, IL 61801, USA

Corresponding author: Michael Birbas (mbirbas@ece.upatras.gr)

The work of Michael Birbas, Eleftherios Mylonas, Nikolaos Tzanis, and Alexios Birbas was supported in part by the H2020 Project "ENERgy-efficient manufacturing system MANagement (ENERMAN)" under Grant 958478.

ABSTRACT In this work, we present the design and implementation of an ultra-low latency Deep Reinforcement Learning (DRL) FPGA based accelerator for addressing hard real-time Mixed Integer Programming problems. The accelerator exhibits ultra-low latency performance for both training and inference operations, enabled by training-inference parallelism, pipelined training, on-chip weights and replay memory, multi-level replication-based parallelism and DRL algorithmic modifications such as distribution of training over time. The design principles can be extended to support hardware acceleration for other relevant DRL algorithms (embedding the experience replay technique) with hard real time constraints. We evaluate the accuracy of the accelerator in a task offloading and resource allocation problem stemming from a Mobile Edge Computing (MEC/5G) scenario. The design has been implemented on a Xilinx Zynq Ultrascale+ MPSoC ZCU104 evaluation kit using High Level Synthesis. The accelerator achieves near optimal performance and exhibits a 10-fold decrease in training-inference execution latency when compared to a high-end CPU-based implementation.

INDEX TERMS Accelerator, deep reinforcement learning, edge computing, FPGA, high level synthesis, mixed integer programming, 5G.

I. INTRODUCTION

Reinforcement Learning (RL) has been adopted as a solution mechanism for various problems in edge computing, including Mobile Edge Computing (MEC) scenarios (supported by the expansion of 5G networking technologies) ranging from task offloading and resource allocation to routing, caching placement and energy harvesting. The main use cases for RL in edge computing can be divided in two large categories, namely NP-hard problems (e.g. Mixed Integer Programming-MIP) and problems with inherent Information Uncertainty/Asymmetry about the underlying network parameters and computational resources. For the former case, Deep RL (DRL) algorithms have shown superior performance in comparison with state-of-the-art conventional techniques, such as Linear Relaxation [1], resulting in both faster convergence and solutions of higher quality. In the latter case, RL algorithms such as Bandit Learning have been successfully employed [2], [3] to solve the exploration-exploitation dilemma, resulting in action selection policies with near-optimal performance. As demanding -Industrial mostly- IoT prompts towards the next generation of edge computing, hard real time constraints should be met by the algorithms orchestrating the computational resource allocation; a latency bottleneck in the control operations might result in malfunction of the overall computational network. Therefore, the need for hardware acceleration of DRL algorithms arises, whenever real-time is a necessity.

In contrast to conventional Neural Network (NN) accelerators which are inference oriented (the training procedure of most Deep Learning algorithms is implemented offline), DRL accelerators should be training oriented in online mode (offline training is infeasible, since the training samples are generated online during the algorithm execution).

The associate editor coordinating the review of this manuscript and approving it for publication was Vicente Alarcon-Aquino.

For example, where the Experience Replay Technique [4] is used, samples of past actions and rewards are stored in the Replay Memory. At fixed time intervals, a batch of samples is randomly selected from the Replay Memory and is used for NN training. This results in a significant bottleneck; the training should be completed and the NN weights should be updated before the next inference step starts. Given the higher complexity imposed on training (both back propagation (BP) and forward propagation (FP) are employed) compared to the inference (which employs only FP) and in conjunction with the fact that each training step uses a batch of samples, it is obvious that training accounts for most of the execution time. Additionally, the increased training latency is a significant challenge for applications that impose high inference throughput constraints; the input samples which arrive right before training starts must wait for its completion before the corresponding actions are selected via the Inference procedure (Fig. 1).

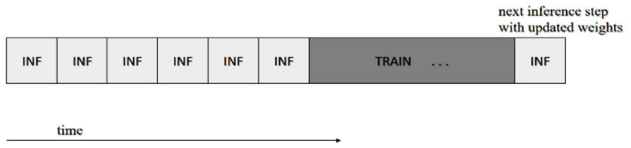

FIGURE 1. Latency bottleneck induced by DRL training (INF stands for inference).

The deployment of fast hardware and software simulators and cyber-twins of the environment in which the DRL agents act and train on, strengthens the need for real time DRL acceleration. The environment simulators allow for the utilization of generated inputs and provide action feedback at a much higher rate than the physical instance of the environment. This leads to the availability of a large volume of training samples which require acceleration of both training and inference in order to be successfully utilized by the DRL algorithm. Simulators not only increase the availability of training samples and therefore the quality of the final action selection policy, but also reduce the risk of exposing the physical environment to inference actions produced at the early stages of training. Those actions obviously tend to be more sub-optimal in comparison with the ones produced at later stages.

Fig. 2 illustrates the concept of using generated samples for training the NN, where the inference and training procedures using generated inputs and feedback correspond to the cyber training loop. Assuming that during the algorithm's execution the environment configuration changes, the past training is invalidated. Moreover, the environment provides the accelerator with real inputs at a rate which is much smaller than the generated input creation rate. With the usage of a simulator-evaluator configuration the NN is re-trained through evaluating the selected actions on the simulated environment. When the next real input occurs, the NN has already undergone some training and therefore provides the environment with actions that are less sub-optimal in comparison to a

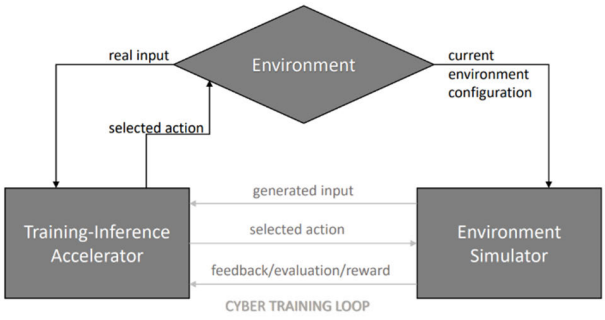

FIGURE 2. Cyber training loop and interaction with the physical environment.

scenario without a cyber training loop. In order to meet the requirements of the cyber-training loop, both the environment simulator-evaluator and the DRL accelerator should operate with the minimum latency possible.

For solving NP-hard control problems, the literature suggests a small number of layers (typically 1 or 2 hidden layers) which are usually fully connected [5], [6]. Although convolutional layers are occasionally suggested, in this work we focus on fully-connected layers since our typical scenario is that the input features do not exhibit space correlations. Additionally, typical batch sizes are 64-128 samples and there are frequent training steps between the inference ones. The aforementioned configurations are unsuitable for acceleration onto GPUs for three main reasons [7]:

- Batch sizes of that kind lead to underutilization of the GPU resources.
- Limited off-chip data bandwidth.
- For frequent weight updates the kernel launch overhead associated with typical GPU programming models accounts for a high percentage of the overall execution time, which significantly increases for smaller NNs.

For those reasons, FPGA based implementations are gradually gaining popularity in DRL applications [7]–[9].

Within this context, in this paper we present the design of an FPGA based Accelerator for DRL with on-chip weight and replay memory and application specific resource utilization.

For the Accelerator design we employ Xilinx's Vivado High Level Synthesis (Vivado HLS) framework. HLS was selected instead of VHDL coding for two main reasons:

1) VHDL coding for architectures of such complexity is a difficult and lengthy task which lacks flexibility.
2) The HLS directives abstraction favors the reusability of the code for the design of other architectures which accelerate similar DRL algorithms with minimal effort.

The main contribution of this paper is:

- The design and implementation of an ultra-low latency DRL accelerator that can be employed for the solution of MIP problems with hard real time constraints. It is the first time, to our knowledge, that a DRL accelerator targeting real-time online training and inference is presented in the literature. The key points of the

design include a distribution of training over time and a pipelined Training Module. The evaluation of our design (using the Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit) shows that for a generic use-case a decrease in the training-inference execution latency by a factor of 10-200 is achieved when compared to CPU implementations with state-of-the-art embedded processors.

Additional contributions of this paper include:

- The adaptation of the algorithm under acceleration and its combination with the Lagrange Multipliers optimization method for the solution of a real-time Task Offloading and Resource Allocation edge computing problem.
- A methodology for extending the proposed architecture to support other on-line DRL algorithms which use the Experience Replay Technique, under the assumption of joint NN and training configurations as described earlier in this section. The scalability of the design allows for the acceleration of DRL algorithms using larger NNs, in terms of number of fully connected layers, number of neurons per layer and batch size-training interval ratio.
- The provision of practical HLS guidelines for deployment in other similar design scenarios.

II. EXISTING WORK IN DRL ALGORITHMS FOR EDGE COMPUTING AND HARDWARE ACCELERATION

A wide range of DRL algorithms have been employed for solving NP-hard problems in edge computing and task offloading orchestration. The authors of [5] propose the DROO algorithm which is the main inspiration for the developed and accelerated algorithm in this paper. The DROO algorithm near-optimally adapts task offloading decisions and wireless resource allocations to the time-varying wireless channel conditions in a wireless MEC scenario, while decreasing the computation time by more than an order of magnitude in comparison with existing optimization methods. Deep Q-learning (DQL) [10] with Experience Replay has also been adopted as a solution mechanism for the problems of interest. In [11] the authors are employing a DQL based method to jointly make computation offloading and bandwidth allocation decisions in a MEC scenario in order to minimize the overall offloading cost in terms of energy, computation, and delay. Although the problem formulation results in a mixed integer nonlinear programming optimization problem, the DQL based method achieves near-optimal performance. DQL has been also employed by the authors of [6] and [12] in MEC scenarios, while in [13] is used for Vehicle Edge Computing and Networks. Policy Gradient Methods [14] have been also employed for the problems of interest. As an example, consider [15] where the authors tackle the issue of Quality of Experience (QoE) in edge-enabled Internet of Things (IoT). Recently, Meta-DRL methods are starting to gain popularity due to the adaptive nature of these algorithms. For application of Meta-DRL schemes in edge computing and task offloading the reader can refer to [16], [17].

Hardware Accelerators for DRL have been considerably less addressed in the literature compared to inference-focused ones. In [18], the authors propose a many-core accelerator design with a customized Network-on-Chip which achieves a significant energy-efficiency boost in comparison with GPU-based implementations. In [19] and [20] ASIC designs based on Transposable PE Arrays and Experience Compression are presented. The target is again energy-efficiency, in contrast to the target of this paper, which is latency minimization. Accelerators based on FPGAs are presented in [7], [8] and [9]. In [8], the authors present a CPU-FPGA heterogeneous architecture, which however only targets the acceleration of the Inference operation. In [7], an accelerator for the Asynchronous Advantage Actor-Critic DRL algorithm which achieves 27.9% better performance than that of a state-of the-art GPU-based implementation and 62% better energy efficiency in 6 games of the Atari 2600 benchmark collection is presented. In [9], the authors propose an Accelerator for the Actor-Critic algorithm based on Quantization-Aware Training and Adaptive Parallelism. Although [7] and [9] focus on both training and inference, the complex structure of the NN they are targeting does not allow real time operation since the computational complexity is not consistent with the strict latency constraints of the problems of interest.

III. BACKGROUND

A. THE ALGORITHM UNDER ACCELERATION

The designed Accelerator is based on the idea of the DROO algorithm, presented by Huang *et al.* in [5]. The DROO algorithm was selected due to its relatively lower complexity when compared to approaches such as DQL. We will first give a brief overview of the algorithm and how it can be employed for solving MIP problems. Although the DROO algorithm is adjusted to the problem formulation of [5] and includes a one-dimensional bi-section search for the solution of a convex sub-problem, we only employ the part of the algorithm which refers to the solution of MIP problems, since this is the target of our work.

We assume a problem formulation according to which at each timestep a vector $\boldsymbol{r}$ is given as input to the algorithm. We furthermore assume that the algorithm should choose an action represented by a vector $\boldsymbol{x}$ of binary values in order to minimize a cost function Q. The optimal selection of the vector's elements $\boldsymbol{x}^*$ (equivalently the optimal policy if the vector's values represent actions selection, which is the common case in the DRL framework) is given by:

$$\boldsymbol{x}^*(r) = argmin_{\{\boldsymbol{x}\}} Q(\boldsymbol{r}, \boldsymbol{x}) \tag{1}$$

If we denote the dimension of vector $\boldsymbol{x}$ as N, then there are 2^N possible values for the vector (equivalently possible actions) and therefore a brute force method for solving (1) is impractical even for small to medium values of N.

In the paper describing the DROO algorithm [5], the authors propose a DRL mechanism according to which the NN learns from the past action selections and limits the number of vector values to be tested to N + 1 at each timestep,

which assuming that the computation cost function Q is not computationally heavy, can be implemented with very low latency. This is not the case in [5], since the computation of the function Q accounts for most of the execution time (this is due to the complexity of the solution of the second sub-problem that Huang *et al.* include in their problem formulation). Our proposed accelerator instead, targets applications where the main computational bottleneck is not stemming from the Q function but rather from the NN-related procedures (inference and training). In Fig. 3 we illustrate the main idea of the DRL mechanism.

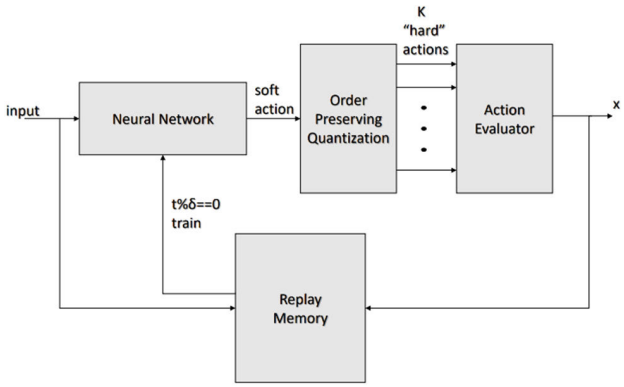

FIGURE 3. The idea of the DRL mechanism.

The input $\boldsymbol{r}$ is fed into the NN, which through the inference process provides an output vector $\widehat{\boldsymbol{x}}$, containing real values in the interval [0,1], which represents the ''soft'' or ''relaxed'' action. The ''relaxed'' action is then quantized using an Order Preserving method [5] and K binary actions are obtained. Although according to [5] K can be set from 1 to $N+1$ and can also be adjusted dynamically during the algorithm's execution, in applications where the main computational burden is the NN and not the action evaluation (the computation of the value of the cost function for each one of the K binary actions) we can set K to be constantly equal to N + 1.

Next, the K binary actions are evaluated and the action with the minimum or maximum value of function Q is selected depending on whether the MIP problem is a minimization or a maximization problem respectively. Both the input $\boldsymbol{r}$ and the selected action $\boldsymbol{x}$ are stored in the Replay Memory to be used in the future for training. When the Replay Memory has been filled with training samples, the newer samples replace the older ones. The presence of the evaluator acts as a simulation of the physical environment, which ranks the quality of the actions proposed by the NN, providing feedback about the most rewarding option. Every δ timesteps ($t\%\delta = 0$) a batch of training samples ($\boldsymbol{r}, \boldsymbol{x}$) is randomly selected from the replay memory and feeds the training procedure. The training is conducted with $\boldsymbol{x}$ being used as the ground truth output labels which correspond to input $\boldsymbol{r}$. As the algorithm progresses and the NN is trained, it learns to propose constantly better binary actions, based on the input experience from the past and the feedback from the evaluator. The generalization offered by the NN, results in near-optimal performance regardless of whether an input has been observed in the past or not. Since the environment configuration is dynamic and may change during the algorithm's execution (something which is reflected as a change in the cost function Q) the algorithm must be able to quickly adapt to potential changes through retraining with a new cost function. Therefore, both training and inference are bound to be implemented online with the minimum latency possible.

B. NEURAL NETWORK COMPUTATIONS

The core part of the algorithm under acceleration are the NN computations, which are divided into 3 categories, namely FP, BP and Network's Gradients Computation (GC). FP is performed both in inference and training, while BP and GC are performed only in training.

During the FP process, the outputs of the neurons of the previous layer are used, in combination with the network's weights, to calculate the output of the neurons of the next layer. We denote the vector containing the output of the neurons of layer i as $\boldsymbol{A_i}$ and the matrix containing the network's weights used for the calculation of the outputs of layer i as W_i. If the vector $\boldsymbol{Z_i}$ is defined by the following equation:

$$\boldsymbol{Z_i} = W_i \cdot \boldsymbol{A_{i-1}} + b_i \tag{2}$$

where b_i denotes the bias term of layer i, then we have:

$$\boldsymbol{A_i} = g\,(\boldsymbol{Z_i}) \tag{3}$$

where g is the layer's activation function, which is applied elementwise on vector $\boldsymbol{Z_i}$.

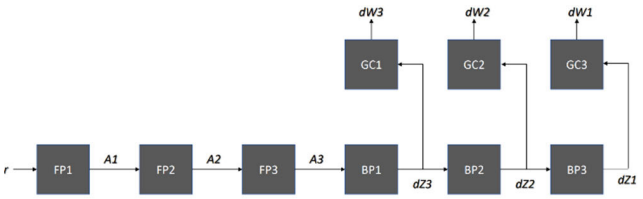

FIGURE 4. The training computations flow.

During BP, the gradients of the network's selected loss function with respect to the vectors $\boldsymbol{Z_i}$ are calculated and are stored in the vectors denoted as $\boldsymbol{dZ_i}$. The calculations start from the last layer and continue towards the first (in order to compute $\boldsymbol{dZ_i}$, the value of $\boldsymbol{dZ_{i+1}}$ is required). Finally, the values of the vectors $\boldsymbol{dZ_i}$ are used for the calculation of the matrices dW_i (GC computations), which are then used by the selected learning algorithm to update the current values of W_i. Fig. 4 illustrates the training computations flow.

IV. EFFICIENCY DRIVEN ALGORITHMIC MODIFICATIONS

In order to design an efficient hardware Accelerator, we apply modifications to the DRL training process, the effect of which in the algorithm's accuracy is generally application dependent. In section VI. we present this effect through our benchmark problem set-up. All the modifications are tested

directly in the C/C++ code used by the HLS tool and the algorithm's performance is compared against the optimal one, which is calculated using the brute force method for all the possible values of the action vector $\boldsymbol{x}$. In the remaining of this section, we will describe the applied modifications, as well as justify the reasons for which they lead to a more efficient hardware implementation.

A. LEARNING ALGORITHM AND DISTRIBUTION OF TRAINING OVER TIME

The designed Accelerator is suitable for algorithms employing Batch Gradient Descent (BGD) [21] as the learning algorithm, according to which the gradient values for a batch of training samples are accumulated. The learning update rule is given by:

$$\theta \leftarrow \theta - a \cdot \frac{d\theta_{batch}}{batch_{size}} \tag{4}$$

where $d\theta_{batch}$ denotes the sum of the gradient values for all the samples in the batch, θ is the parameter to be updated (the network's weights) and a is the learning rate. Using a learning algorithm such as Stochastic Gradient Descent [21], with parameter updating after each individual sample's gradient evaluation is not a suitable option for hardware acceleration due to the large amount of Memory reads and writes. Other batch-oriented algorithms such as batch ADAM [21], [22] or batch RMSPROP [21] could also be used, with a minor increase in hardware complexity, but since in our benchmark problem BGD showed very good performance we decided to use it as the learning algorithm in our accelerator.

To face the issue of the latency bottleneck introduced by training as described in section I. and displayed in Fig. 1, instead of performing one training step with a batch size of B after δ inference steps (where δ denotes the training interval) we continuously perform a combined inference and training step, using a batch size of B/δ and accumulate the gradient values (Fig. 5).

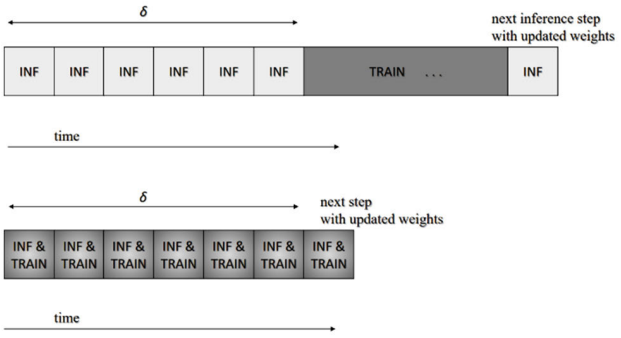

FIGURE 5. Distribution of training over time.

Every δ timesteps we perform a weight update using the accumulated values of the gradients and reset the accumulators. This way the workload is split between each timestep and an architecture with constant inference throughput can be designed. As described in the architecture description section (sub-section V.), the training and inference are performed in parallel. This does not result in an increase in the inference latency since the FPGA resources are allocated in a way that the latencies of inference and training computations per timestep are balanced.

B. HARDWARE DRIVEN MODIFICATIONS

In order to ensure that our hardware design is efficient we compress the Neural Network using fixed point arithmetic for all the inference and training computations. To decide on the fixed-point representation of each variable, we select the representation with the minimum number of bits that does not lead to significant accuracy degradation. This choice depends on the application and therefore an accuracy evaluation should be conducted. When it comes to the activation function, although the hidden layers of the accelerated NN configuration employ the Rectified Linear Unit (ReLU), the hardware implementation of which is trivial, the Sigmoid function has been selected for the output layer. This is due to the fact that the outputs of the NN, which represent relaxed actions are bound to belong in the interval [0,1]. A direct hardware implementation of the Sigmoid function would not be efficient due to the exponential term. Instead, we are using LUTs. Additionally, the Experience Replay Technique suggests that a randomly selected batch of training samples should be chosen at each training step. To implement the random choice of Replay Memory indices we are using 16-bit Fibonacci Linear Feedback Shift Registers (LFSR) [23]. Simulation shows that using the Fibonacci LFSRs as the pseudo-random number generators does not lead in any performance degradation in comparison with conventional software generators such as the C/C++ rand() function.

V. ACCELERATOR ARCHITECTURE

Before proceeding with presenting the architecture in detail we will first summarize the design principles of our Accelerator. Those principles together with the software/hardware modifications described in section IV. result in the significant performance increase both in terms of latency as well as energy efficiency.

- **Task Level Parallelism of Inference and Training**
  The inference and training processes are parallelized and their latencies are balanced. Thus, ultra-low overall latency can be achieved and hard real time constraints can be met.
- **On chip-memory for the storage of Weights and Replay Memory**
  One of the main burdens in CPU and GPU based implementations is the off-chip data traffic bottleneck. By utilizing properly the FPGA memory resources (BRAMS and distributed LUT-RAMS) to accommodate all the memory requirements this bottleneck is removed.
- **Input-Stage Level Pipeline in Training**
  Although the inference process solely involves FP computations, the Training process involves FP, BP and GC ones. Consequently, the relative complexity of training

is higher than inference and thus if training and inference are to be conducted in parallel, we must significantly reduce the training latency. To deal with this issue, we propose an input-stage level pipeline scheme by splitting the FP, BP and GC computations into separate pipeline stages. By employing this pipeline scheme there is no need for balancing the total inference and training latencies. Instead, we should balance the total inference latency with the training's Initiation Interval (the latency of one Training Stage). Since training can be considered as throughput-dependent this pipeline scheme does not affect the algorithm's accuracy (for the simulation results please refer to section VI.).

- **Multiple Levels of Replication based parallelism**
  -Sample Level: Since at every timestep a batch of size B/δ is used for training, to speed up the process the accelerator is performing the computations for all these samples in parallel. This further decreases the training Initiation Interval and allows for balancing the training-inference latency (since the inference computations are always performed on a single input sample). High B/δ ratios can be supported but with an increase in the Inference latency due to the Training overhead introduced when the FPGA resources are not sufficient for the parallel computation of all the samples in the batch.
  -Neuron Output Calculation Level: There are no dependencies between the outputs' calculations for Neurons belonging in the same layer. Thus, we can split the Neurons of a given layer in groups and assign them to different processing elements (PEs), operating in parallel. This type of parallel processing can be employed in all the types of computations (FP, BP and GC).
  -Tree adders and Parallel Multipliers in Inner Product Calculations: All the computation blocks involve inner products in their calculations. To speed up the inner product calculations we are employing tree adders and parallel multipliers and we are partially unrolling the corresponding loops.
- **Loop Level Pipeline**
  All the inner loops, both computational and memory copy loops, are pipelined, leading to faster execution.

A. HIGH-LEVEL OVERVIEW, INTERACTION WITH CPU AND SCHEDULING

Our Accelerator is responsible of all the NN related computations. Thus, the current input, which is denoted as $\boldsymbol{r}(t)$ is passed to the accelerator and the inference process is conducted so that the relaxed action vector $\widehat{\boldsymbol{x}}(t)$ is calculated.

The quantization and action evaluation-selection process are performed in the CPU and the action $\boldsymbol{x}(t)$ is selected. This does not introduce any significant latency bottleneck since we are targeting applications where the relative complexity of the NN related computations is higher than the action evaluation. The CPU-Accelerator interaction is displayed in Fig. 6.

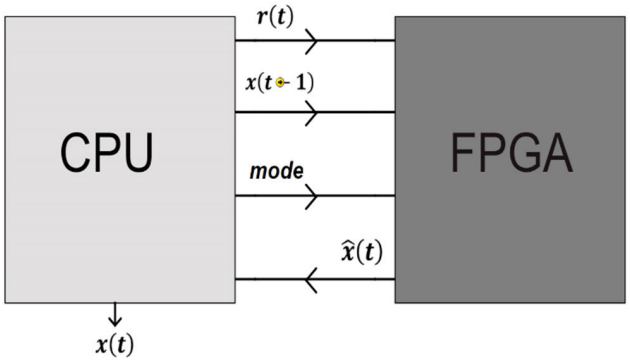

FIGURE 6. CPU-accelerator interaction (Mode 1).

Apart from the current input $\boldsymbol{r}(t)$, the selected action in the previous timestep $\boldsymbol{x}(t-1)$ is also passed to the Accelerator so it can be stored in the on-chip Replay Memory along with the previous input $\boldsymbol{r}(t-1)$ and be used later for training. The training process is performed every timestep as described in section IV. A. and is transparent to the CPU. All the weights are stored on-chip and therefore there are no off-chip data traffic related bottlenecks. This of course comes under the assumption that the NN configuration complexity allows for the on-chip storage of the weights. As mentioned in section I., this assumption is valid for most of the NN used for DRL based solution mechanisms for NP-hard problems in edge computing.

There are two modes that determine the Accelerator's functionality. Mode 0 refers to the required initialization after reset. The Weights' initial values and the contents of the Sigmoid LUTs are passed and stored in the Accelerator. No effort was devoted for accelerating Mode 0, since it is only executed once after reset. Mode 1 refers to the nominal functionality as described in the previous paragraphs of this section.

In cases where the input generation rate from the environment is lower than the inference throughput that can be provided by the system, the CPU can act as an environment simulator and action evaluator, providing the FPGA Accelerator with both generated inputs $\boldsymbol{r}$ and feedback regarding the best action choices (via the vector $\boldsymbol{x}$). This way the algorithm can benefit from the Cyber Training Loop, as described in section I. In our use case the CPU did not need to employ any sophisticated algorithm for the input generation. Instead, providing the Accelerator with random inputs, which resulted in the availability of a larger volume of training samples for the Cyber Training Loop, was enough for the quick adaptation of the algorithm to the changing environment configurations. Since the action evaluation is conducted in the CPU, there is no need for reconfiguring the Accelerator every time the environment changes. The impact of the change is reflected on the different way that actions are evaluated (different cost function Q) and consequently in the selected actions $\boldsymbol{x}$, which are used to train the NN. After a change in the environment

the newer training samples in the Replay Memory contain the required information for the Accelerator to re-adapt to the new configuration.

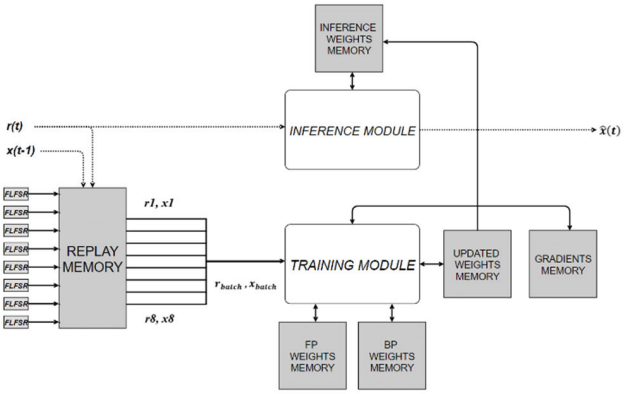

FIGURE 7. Architecture overview.

In Fig. 7 the core blocks of the architecture are presented. There are two core modules, namely the Inference and Training Modules which operate in parallel. Since both operations are using the same weights, to avoid memory conflicts there are two weight memories one for inference and one for training. Additionally, since the training module is split into pipeline stages which operate simultaneously and both the FP and BP training computations access the same weights, the training weights memory consists of two replicas. There are two additional memories, the first of which accumulates the gradients dW_i as described in sub-section IV.A. The role of the Updated Weights Memory (which also contains two replicas) will become clearer in sub-section B. All memory blocks, including the Replay Memory are implemented using on-chip Dual Port Block RAMs (BRAMs). Although the total memory requirements are increased by a factor of 3 in comparison with a serial CPU based implementation (serial execution of inference and training with no pipeline stages in training) which would use a single weight memory and a memory for accumulating the gradient values, the number of weights in the typical NN configurations for the applications of interest does not impose high memory requirements. Thus, all the weight memory blocks in our designed architecture can be effectively implemented and partitioned without utilizing a large percentage of the FPGA memory resources.

When designing using High Level Synthesis, the tool will not treat functions that access the same memory locations as concurrent and therefore will not synthesize hardware modules operating in parallel. Thus, the usage of distinct memories which are appropriately partitioned is obligatory for the synthesis of parallel logic from HLS.

In Fig. 8, the time scheduling of the basic operations that define an operational cycle of our Accelerator is presented. Please note that under the term operational cycle we refer to all the operations of the Accelerator from the moment the CPU provides a new input, until all the inference, training and Replay Memory related procedures for this timestep are completed. Operations that are vertically stacked are executed in parallel. The operations that appear in light color are performed in the same way in every operational cycle. The functionality of the operations appearing in dark color is determined by a Finite State Machine (FSM), namely the Bubble Shift Register (BSR), which is presented in subsection B. and controls the training pipeline and the weight updates. As an example, consider the UPD_W_INF operation. Since weights updates are performed once every δ timesteps (and consequently once every δ operational cycles), this operation is either performed or omitted. On the other hand, although the training operation is performed in every timestep, the functionality of the different stages is determined by the BSR.

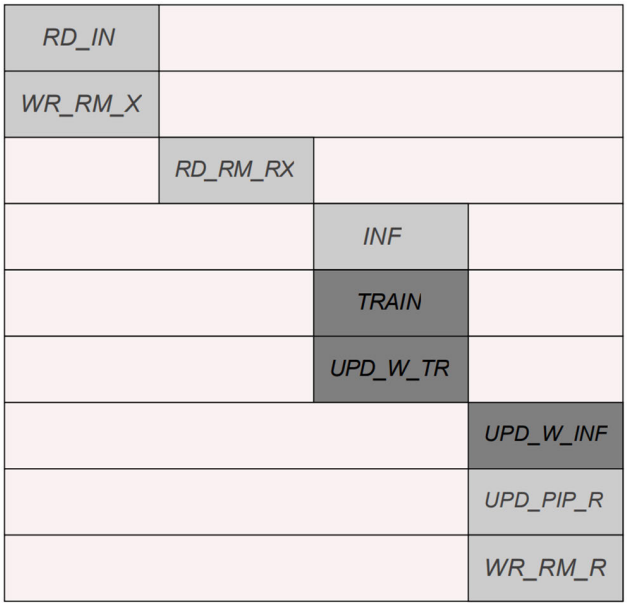

FIGURE 8. Time scheduling of operations.

B. INPUT-STAGE LEVEL PIPELINE IN TRAINING

Before proceeding with the presentation of the architecture of the training module, we will describe the NN configuration used for our use case. The NN consists of one input layer, two hidden layers and one output layer, all of which are fully connected and consist of 20,80,64 and 20 neurons respectively. Thus, there are 3 FP, 3 BP and 3 GC computation tasks. Batch size B is set to 64 and training interval δ is set to 9. Those values were selected after proper experimentation due to the

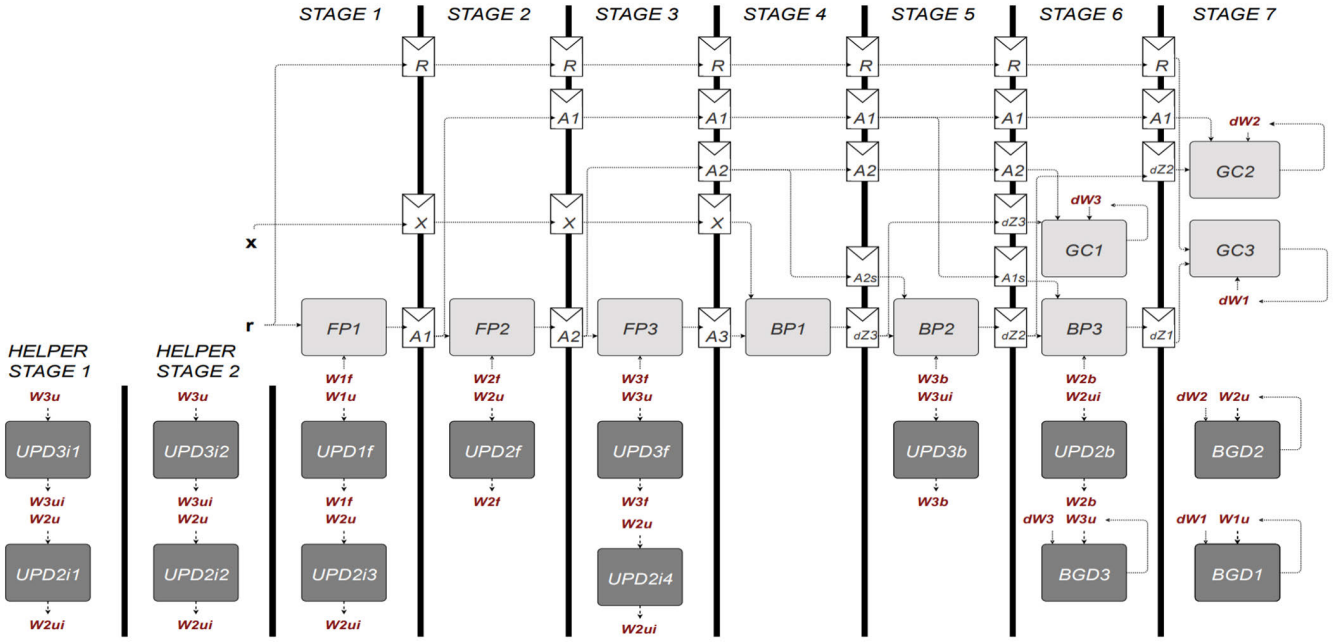

FIGURE 9. The training module.

fact that they combined low computational complexity without accuracy degradation in our use-case. The accelerator is designed to use 8 samples for training per timestep. Training with larger B/δ ratios can also be achieved by using larger FPGAs, so that the available resources allow for training with more samples per timestep or by inserting pure training operational cycles between the mixed inference-training ones, resulting in a decrease in the inference throughput. Training sample gradients are accumulated during the first 8 timesteps and at the final timestep of the training cycle, the training module performs a weight update and resets the Gradient Accumulators.

The architecture of the training module is illustrated in Fig. 9. There are 7 stages that involve both NN computations and weight update-copy modules and 2 helper stages that only involve weight update-copy modules. All the stages are operating in parallel. Each stage consists of two modes of operation, namely the normal operation which is associated with NN computations and is represented by the light-coloured blocks and the ''bubble'' operation which is associated with weight updates and copies and is represented by the dark-coloured blocks.

Helper Stages 1 and 2 do not include light-coloured modules and therefore they only operate in bubble mode (in normal mode they do not perform any operation), while Stage 4 does not include dark-coloured modules and therefore only operates in normal mode.

Only one stage can operate in bubble mode during a given operational cycle of the Accelerator, determined by the BSR, which is a 9-bit register containing a single 1 in the position which corresponds to the stage currently in bubble mode. When the current operational cycle is over, a right circular shift by one position is performed on the BSR and thus in the next operational cycle the next stage will operate in bubble mode. This is illustrated in Fig. 10. If the chosen training interval δ is higher than 9, then the BSR would have δ-bits and the 9 LSBs would determine the modes of the Stages (if all of the 9 LSBs are 0 then all the stages of the stage-pipeline will operate in normal mode). Even though the designed architecture allows for δ values larger than 9, it does not support δ values smaller than 9 (not common in the DRL algorithms employing the Experience Replay Technique).

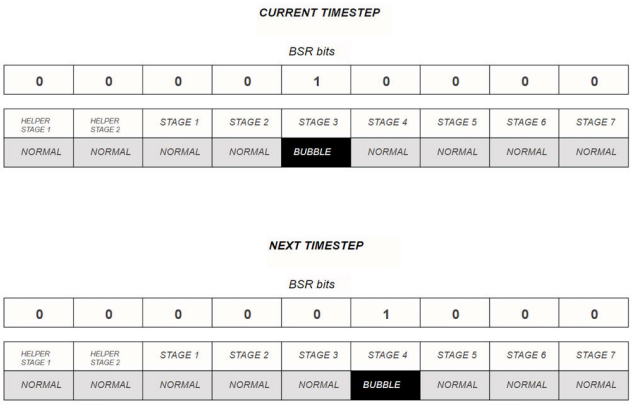

FIGURE 10. BSR operation.

The fact that only one stage can operate in bubble mode at a given timestep removes memory conflicts between the stages since the same memory is never accessed by more than one

module simultaneously and thus the synthesis tool is able to design hardware operating in parallel for all the stages. The memory accesses are illustrated as arrows depicted in red to and from the BRAMs. Please note that each memory block of Fig. 7 consists of different distinct BRAMS (e.g. the distinct BRAMS $W1f$, $W2f$, $W3f$ all belong to the FP Weights Memory block).

The normal mode of operation is responsible for the calculation of the gradients dW_i that correspond to the input batch. It consists of all the FP, BP and GC computations and the required pipeline registers. The role of the pipeline registers is justified by the fact that computations in later stages need as inputs vectors calculated in earlier stages. Please note that although the memory elements responsible for the propagation of the outputs of earlier stages which are needed as inputs in later stages are depicted and referred to as ''pipeline registers'', in fact are not implemented using registers. Instead, we are using distributed LUT RAMS, an option which is more efficient since it significantly reduces the number of multiplexers (MUX) that the synthesis tool generates.

The bubble mode of operation is responsible for all the weight updates (UPD modules) and the Batch Gradient Descent updates (BGD modules). Since no more than 1 stage can operate in bubble state at a given timestep, the weight updates are not performed simultaneously. A design according to which all the weight updates were performed simultaneously would introduce two main problems:

- The worst-case stage latency would increase significantly. On one hand, the updates of the FP and BP Weight Memories corresponding to the same layer would require simultaneous accesses in the same Gradients Memory (e.g. the updates of $W2f$ and $W2b$ would both require access of $dW2$) and thus could not be parallelized. Most importantly, the different partition types used in the FP, BP and Gradients Memories impose limitations on the amount of parallel memory reads/writes that can be supported per clock cycle. To avoid overloading a stage with the latency introduced by accessing memory A to update memory B (e.g. updating $W3b$ by using $dW3$) when the partition types of the memories A and B differ, we split the update workload between different stages, taking advantage of both normal and helper stages. If all weight updates were to be performed at the same time, the idea of splitting this latency burden between different stages would be unapplicable.
- We assume that at an arbitrary timestep t all the weights are simultaneously updated. Since the pipeline is constantly fed with training samples and calculates the corresponding NN's gradient values, the calculations of the samples that were in the first three pipeline stages (FP stages) prior to the weights update that are now stored in the pipeline registers were conducted using the old weight values. However, in the following timesteps when those calculations propagate to the BP and GC stages the corresponding calculations will be conducted using the updated weight values. Therefore, the training procedure would violate the Backpropagation Algorithm semantics. In order to resolve this inconsistency a full pipeline flush would be required after every update and thus the Accelerator's performance would be significantly degraded.

The implemented progressive weight update method using the bubble mode of operation and the BSR tackles both these problems, with the cost of utilizing more memory resources (the Updated Weights Memory 1 and 2 blocks). The main idea of the update mechanism is the usage of the Updated Weights Memories as temporary storage for the updated weight values. The BGD modules perform the batch gradient descent update and store the updated weight values in the Updated Weights Memory 1 BRAMs ($W1u$, $W2u$, $W3u$) when the Stages 6 and 7 operate in bubble mode. When the bubble ''leaves'' Stages 6 or 7 the Gradient Accumulators $dW1, dW2, dW3$ reset and begin to accumulate the gradient values of the next batch of samples. The updated weight values will pass to the FP Weights Memory Block and BP Weights Memory Block when the bubble reaches the corresponding stages which include the UPDf and UPDb modules. This introduces a delay between the calculation of the updated values and the update of the weights which we will refer to as the Training Lag. The Training Lag can be understood as follows:

The Weights which are used for the calculation of the gradients of batch i have been updated with the gradients of all previous batches except for batch $i-1$.

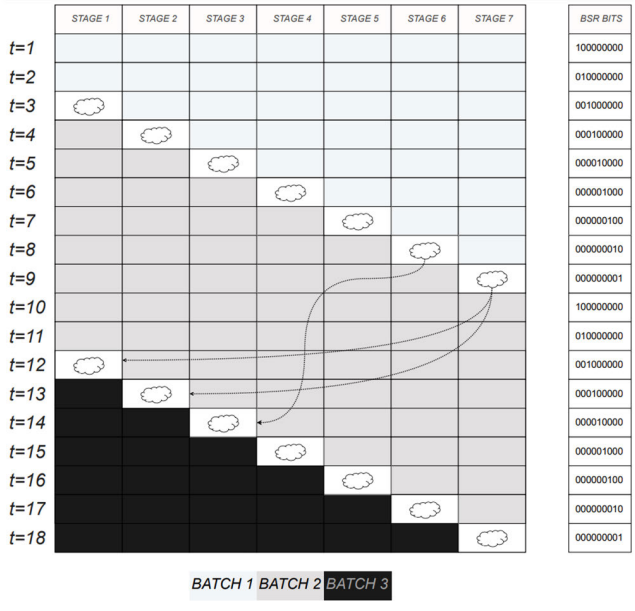

FIGURE 11. The training pipeline.

Fig. 11 illustrates the Training Pipeline and the Training Lag. A bubble-like image indicates that the stage operates in bubble mode. The input-stage level pipeline in the training module is a key-element of the designed architecture since it decreases the Accelerator's latency by approximately a

factor of 7 (under the assumption that the stage latencies are balanced), without introducing any weight-update related bottlenecks and therefore enabling a constant high inference throughput. Although this comes under the cost of the Training Lag, simulation shows that the Training Lag does not degrade the algorithm's accuracy (Section VI.).

At this point we will present some HLS guidelines in order to achieve the required functionality of the input-stage level pipeline (all the stages to be operating in parallel). As mentioned earlier in this section, the synthesis tool will be able to synthesize parallel operating hardware provided that there are no accesses to the same memory element by more than one stage during a given operational cycle. With the term ''memory element'' we are not only referring to the weight and gradient BRAMs but also to the pipeline registers. As Fig. 9 illustrates, the FP1 module in Stage 1 writes register $A1$, while the FP2 module in Stage 2 reads register $A1$. Therefore, the synthesis tool will detect dependencies between the 2 stages and will synthesize serial operating hardware. To tackle this issue the pipeline registers are implemented as pairs, as displayed in Fig. 12. Please note that since the UPD_PIP_R operation is done in parallel with the essential UPD_W_INF and WR_RM_R operations (Fig. 8) after the TRAIN-UPD_W_TRAIN operations complete, it does not introduce any extra latency.

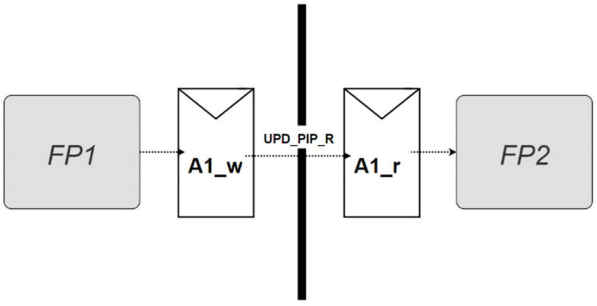

FIGURE 12. Pipeline register pair.

C. REPLICATION BASED PARALLELISM

The delay of the training module at a given operational cycle is determined by the Stage with the highest latency. Since not all Stage computations have the same complexity (e.g. the FP2 module calculates the outputs of the 64 neurons of the second hidden layer using the outputs of the 80 neurons of the first hidden layer, while the FP3 module calculates the outputs of the 20 neurons of the output layer using the values of the 64 neurons of the second hidden layer), stages with higher complexity require more parallelism in order for their latencies to be balanced with those of stages with lower complexity. Therefore, the FPGA resources should be carefully allocated between the different stages in order to achieve:

- **Balanced Stage Latencies**: The acceleration of stages with low latencies does not yield any gain, since the delay is determined by the maximum Stage Latency, but rather only increases the FPGA resource utilization.
- **Low Maximum Stage Latency**

When it comes to the Inference Module which only consists of FP sub-modules, we should note that it does not comprise the Input-Stage Level Pipeline that is included in the Training Module. This is a rather rational decision since the Accelerator is expected to be able to provide the output $\widehat{\boldsymbol{x}}(t)$, corresponding to the input $\boldsymbol{r}(t)$ in a single timestep-operational cycle, in order to be consistent with the ultra-low latency constraints of the applications of interest. Thus, there is no requirement of balanced delay between the 3 FP sub-modules in the Inference Module since the Inference delay is determined by the sum of the 3 FP delays. However, since the Training and Inference Modules are operating in parallel, the inference delay should not be higher than the Stage Latency of the Training Module, otherwise that would lead to a design bottleneck. Thus, since 3 sub-modules contribute to the Inference delay, while only a single sub-module contributes to the Training Stage Latency, the FP computation blocks in the Inference Module should have increased levels of parallelism when compared to the Training computation blocks. Please note that to achieve increased parallelism, increased partitioning of the memory blocks associated with the corresponding sub-modules is needed. Thus, the delay of the UPD_W_INF operation is also decreased, since the increased memory partition allows for more parallel memory reads and writes. Although the typical delay of the UPD submodules in the Training Modules, which are responsible for the memory copies, is comparable to that of a computational sub-module (FP, BP, GC), the increased partition of the Inference Weights Memory Block leads to much faster execution. In fact, the Inference weight update is accelerated to such an extent that it can be scheduled after the completion of the parallel operation of the Training and Inference modules, without introducing significant overhead to the Accelerator's operational cycle.

In order to decrease, balance and adjust the latencies of the sub-modules we employ two methods, namely Replication in the Neuron Output Calculation Level and employment of Tree Adders and Parallel Multipliers in Inner Loop Calculations.

The notion of Replication in the Neuron Output Calculation Level is presented in Fig. 13. We split the Neurons of a given layer in groups and assign the groups to different PEs, operating in parallel. Although this notion is more straight-forward for the FP blocks, which indeed calculate the outputs of Neurons, the outer loops of the other kinds of submodules can also be split and assigned to identical parallel operating PEs.

The way that the Replication in the Neuron Output Calculation Level is employed in all three types of sub-modules is described in detail in Fig. 14. Please note that for simplicity we have only illustrated the split of the computations in two PEs, while the number of replicated PEs differs for each sub-module of the Training and Inference modules

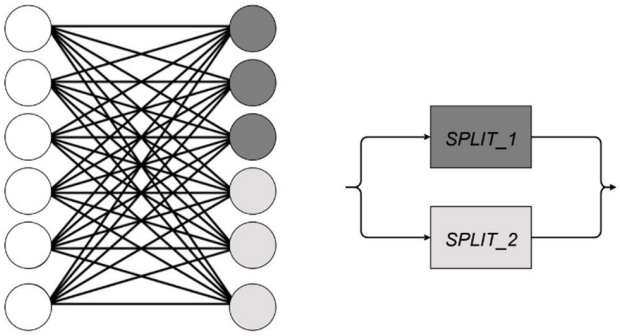

FIGURE 13. The notion of Replication in the Neuron Output Calculation Level.

(e.g. sub-modules in the Training Module with larger computational complexity require more parallelism in order for the Stage Latencies to be balanced and therefore require more replicated PEs).

Additionally, the same figure illustrates the required steps for the parallel operation of the PEs, namely:

- Partitioning of the Memory Elements
- Replication of the inputs that are fully accessed by all the PEs.

Please note that we are using the term rows to refer to the first dimension and the term columns to refer to the second dimension of 2-dimensional memory elements.

As mentioned previously in this sub-section, the second method of achieving replication-based parallelism is to employ Tree Adders and Parallel Multipliers in Inner Loop Calculations. This, in combination with loop-level pipeline in the inner loops, increases significantly the performance of the designed Accelerator. Both those concepts are presented in Fig. 15. Please note that the C/C++ code in Fig. 14 displays calculation for a single sample, while the calculations in the Training Module are executed on a minibatch of 8 samples in parallel. The level of parallelism used in Fig. 15 is not indicative of the actual design, but is selected for simplicity reasons (in fact, in our design the FP2 sub-module employs more tree adders and parallel multipliers due to its high computational complexity when compared to other sub-modules).

Regarding the memory accesses, in order for the loop iterations to be executed in the minimum number of cycles, the module should be able to read the memory positions $W2[i][j]$, $W2[i][j+1]$, $W2[i][j+2]$ and $W2[i][j+3]$ in the same cycle. That is inapplicable if the weight memories are only partitioned in the first dimension. Thus, we further partition the weight memories in the second dimension using the cyclic partition option. Fig. 15 further explains the notion of the cyclic partition with the partitioning factor selected to be equal to 4.

Cyclic partitioning is obviously, also required in arrays $A1[0]\ldots A1[7]$, which store the outputs of the layer 1 Neurons for each one of the 8 samples. Regarding the BP

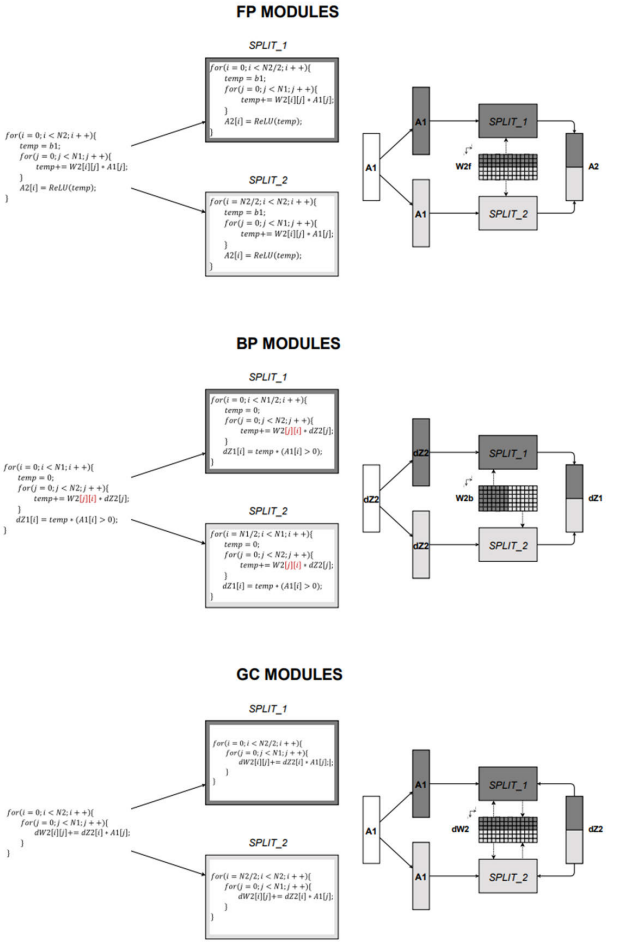

FIGURE 14. Replication in the Neuron Output Calculation Level in detail.

sub-modules, it is evident that the cyclic partition should be applied on the first dimension of the corresponding weight memories. The vertical columns in Fig. 15 determine the pipeline stages. In practical design the pipeline includes one extra stage related to the increment of the loop iteration index (since the FP2_inner_loop is not unrolled completely) and the calculation of the corresponding memory addresses, but it has been left out of the figure for simplicity. It is worth mentioning that the loop-level pipeline is not only employed in the computational inner loops but also to the memory copy loops.

D. EXTENSIBILITY TO OTHER DRL ALGORITHMS

The general principles of the designed architecture can be applied to a wide range of DRL algorithms using the Experience Replay Technique, assuming consistency with the typical NN configurations described in section I. The key differences are expected to be:

- **The way that the training samples are generated**, which in our design is not included in the Accelerator's responsibilities, since they are given as input to

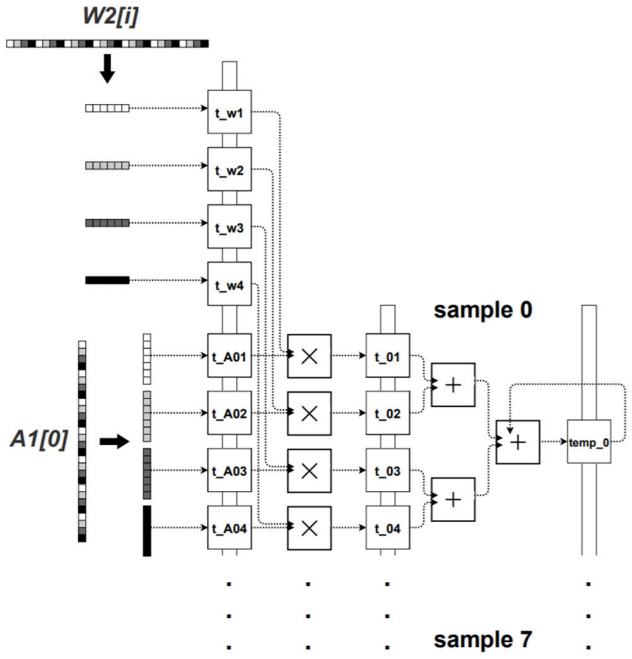

FIGURE 15. Tree adders, parallel multipliers and loop-level pipeline.

the Accelerator. If we consider Deep Q-Learning as an example the training samples consist of state-actions and q-values.

- **The NN loss function and the gradient calculation**. In scenarios with different loss functions/gradient calculations the BP and GC modules can be re-designed to fit to the problem of interest but the design principles of our Accelerator (e.g. Input-Stage Level Pipeline in Training, Replication for Parallelism etc.) are still applicable.

Modifications in the Accelerator's design may also be performed to be conformant with different DRL algorithms. As an example, the reader can consider the addition of an extra Inference Module, so as to achieve modelling of the Target NN in a Double Deep Q-Learning framework [24].

VI. USE CASE AND ACCURACY EVALUATION

In this section we present the problem formulation of our use-case and propose a solution based on the accelerated DRL algorithm. We compare the algorithm's performance against the optimal performance, achieved using the brute-force search and evaluate the effect of the algorithmic modifications of section IV. and the Training Lag described in section V. on the algorithm's performance.

A. PROBLEM FORMULATION AND SOLUTION MECHANISM

We assume a Mobile Edge Computing (MEC) scenario (Fig. 16) according to which a set of N Users can offload their computational tasks to an Edge Server (ES). At each timestep the Users may choose to undertake the computations of their tasks utilizing their own computational resources

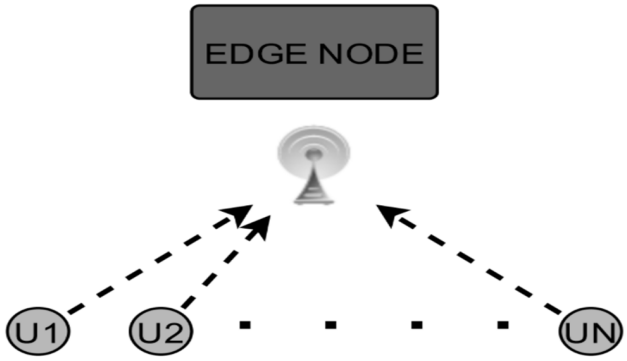

FIGURE 16. Mobile edge computing-task offloading scenario.

(local execution) or offload them to the ES. The users can transmit their tasks to the ES wirelessly, e.g. by utilizing 5G technology. For simplicity we assume that the transmission rate through the channel between User i and the Edge Node is described by a variable r_i and that it is independent of the other channels' states. In our problem we assume that r_i is a random variable, with changing values over time, in order to capture the dynamic nature of the wireless channel.

We assume that at each timestep all the Users generate a task characterized by the following parameters:

- s: input size of tasks which affects the transmission delay.
- c: required computational cycles for the execution of the tasks which affects the execution delay.
- q_i: priority weight of User's i tasks. The minimization of the delay of tasks with higher priority is of greater importance when compared to tasks with lower priority.

If a User chooses to compute their task locally at a given timestep t then the task delay is given by:

$$d_{L(i)}(t) = \frac{c}{f_i} \tag{5}$$

where f_i denotes the local computing frequency available to User i.

If a User chooses to offload their task to the ES at a given timestep t then the task delay is given by:

$$d_{ES(i)}(t) = \frac{s}{r_i(t)} + \frac{c}{k_i(t) \cdot f_s} \tag{6}$$

where f_s denotes the ES computing frequency, under which a task would be executed if all the ES resources were allocated to this task. After multiple Users have offloaded their tasks to the ES at the same timestep the ES resources shall be allocated between the Users and $k_i(t) \in [0, 1]$ denotes the fraction of the total ES resources that are allocated to User i at timestep t.

At every timestep an offloading decision $x_i(t)$ is made for every User according to (7).

$$x_i(t) = \begin{cases} 0 & \textit{if user i computes their task locally} \\ 1 & \textit{if user i offloads their task to the ES} \end{cases} \tag{7}$$

Thus, the vector $\boldsymbol{x}(t)$ contains the offloading actions at timestep t.

The goal in this problem formulation is the construction of a task offloading and resource allocation policy (equivalently the specification of the vectors $\boldsymbol{x}(t)$ and $\boldsymbol{k}(t)$) which minimizes the weighted sum of the experienced task delays. The equivalent minimization problem is presented in (8).

$$min_{\boldsymbol{x}(t),\boldsymbol{k}(t)} \left\{ \sum_{i=1}^{N} q_i \cdot \left[x_i(t) \cdot d_{ES(i)}(t) + (1 - x_i(t)) \cdot d_{L(i)}(t) \right] \right\}$$
$$subject\ to: \quad x_i(t) \in \{\{0\}, \{1\}\}, k_i(t) \geq 0, \quad \sum_{i=1}^{N} k_i(t) \leq 1 \tag{8}$$

It is easy to notice that the formulated minimization problem is NP-hard.

In order to solve the NP-hard minimization problem we adopt a similar method as in [5], [25] and split the original problem in two sub-problems.

If the $x_i(t)$values were fixed, then the resulting optimization problem:

$$min_{\boldsymbol{k}(t)} \left\{ \sum_{i=1}^{N} q_i \cdot \left[x_i(t) \cdot d_{ES(i)}(t) + (1 - x_i(t)) \cdot d_{L(i)}(t) \right] \right\}$$
$$subject\ to: \quad k_i(t) \geq 0, \sum_{i=1}^{N} k_i(t) \leq 1 \tag{9}$$

could be reformulated as:

$$min_{\boldsymbol{k}(t)} \left\{ \sum_{i \epsilon O} q_i \cdot \left[\frac{s}{r_i(t)} + \frac{c}{k_i(t) \cdot f_s} \right] \right\}$$
$$subject\ to: \quad k_i(t) \geq 0, \sum_{i \epsilon O} k_i(t) \leq 1 \tag{10}$$

where O denotes the set of the offloading Users' indices. Obviously, the values of $k_i(t)$ for users that compute their tasks locally are set to 0.

Sub-problem (10) is a convex sub-problem and we can obtain a closed-form solution after employing the Lagrange Multipliers method as follows:

$$min_{\boldsymbol{k}(t),\lambda} \{D(\boldsymbol{k}(t), \lambda)\} \rightarrow min_{\boldsymbol{k}(t),\lambda} \left\{ \sum_{i \epsilon O} q_i \cdot \left[\frac{s}{r_i(t)} + \frac{c}{k_i(t) \cdot f_s} \right] - \lambda \left(\sum_{i \epsilon O} k_i(t) - 1 \right) \right\}$$

$$\frac{\partial(D)}{\partial \lambda} = 0 \rightarrow \sum_{i \epsilon O} k_i(t) - 1$$
$$= 0 \rightarrow \sum_{i \epsilon O} k_i(t) = 1 \tag{11}$$

$$\frac{\partial(D)}{\partial(k_j(t))} = 0 \rightarrow k_j(t) = \sqrt{\frac{c \cdot q_j}{f_s \cdot \lambda}} \tag{12}$$

$$\overset{(11)}{\rightarrow} \sum_{i \epsilon O} \sqrt{\frac{c \cdot q_i}{f_s \cdot \lambda}} = 1 \rightarrow \sqrt{\frac{f_s \cdot \lambda}{c}} = \sum_{i \epsilon O} \sqrt{q_i} \tag{13}$$

$$(12) \overset{(13)}{\rightarrow} k_j(t) = \frac{\sqrt{q_j}}{\sum_{i \epsilon O} \sqrt{q_i}} \tag{14}$$

Therefore, the sub-problem (10) has a closed-form solution. Please note that in the problem formulation of the paper describing the DROO algorithm, the convex sub-problem that the authors propose does not have a closed-form solution and is solved using an iterative method.

We have proposed an optimal way of finding the values of $\boldsymbol{k}(t)$, assuming the values of $\boldsymbol{x}(t)$ are fixed. In order to determine the values of $\boldsymbol{x}(t)$, we are employing the accelerated DRL method (the sub-problem of finding the optimal values for $\boldsymbol{x}(t)$is non-convex). Thus, the input $\boldsymbol{r}(t)$ is fed into the NN and through the Inference process, the "soft" offloading action $\widehat{\boldsymbol{x}}(t)$ is obtained. Next, $\widehat{\boldsymbol{x}}(t)$ is quantized using the Order Preserving Method and K candidate actions are obtained. The optimal values for $\boldsymbol{k}(t)$ corresponding to each candidate action are determined using (14). Finally, the candidate actions are evaluated (the weighted sum of the experienced task delays is calculated) and the best action from the candidates is selected. Both $\boldsymbol{r}(t)$ and $\boldsymbol{x}(t)$ are stored in the Replay Memory to be used for training. The described solution mechanism and the notion of the 2 sub-problems is illustrated in Fig. 17 (the convex sub-problem is depicted in light gray).

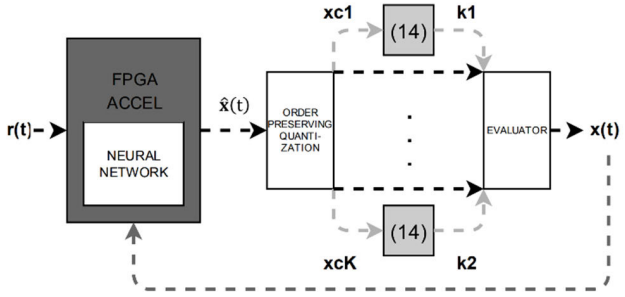

FIGURE 17. Solution mechanism based on the 2 sub-problems.

B. ACCURACY OF THE SOLUTION MECHANISM

Tables 1, 2, 3 define the values of the parameters of the problem and the algorithm that were used in the simulation. Please note that $\boldsymbol{r}(t)$ is firstly normalized before being given as input to the NN. Additionally, since the algorithm shall be able to adapt and retrain under changing environments, in contrast with common practice we do not initialize the weights to small values. Instead, we use initial values that belong in the whole dynamic range as defined in Table 3 (the weights initial values belong in the interval [−1,1]). This initialization models a scenario according to which the NN has adapted to a certain configuration before timestep 0 and a sudden change in the environment occurs. Simulation shows that the accelerated algorithm can efficiently adapt to this change. Please note that all the parameters of Table 2 are

normalized with respect to abstract base values to retain generality.

TABLE 1. Algorithm parameters.

N	20
K	21
NN configuration	20-80-64-20
Batch Size	64
δ	8 (9 effective)
Replay Memory Size	1024
Sigmoid LUT Size	128
BGD Learning Rate	0.1

TABLE 2. Problem parameters.

Parameter	Value
fs	4.0
$\boldsymbol{f}$	[0.56,0.26,0.52,0.53,0.61,0.13,0.22,0.51, 0.19,0.37,0.32,0.42,0.24,0.34,0.64,0.61, 0.42,0.48,0.10,0.40]
$\boldsymbol{q}$	[1.00,1.50,1.50,1.50,1.50,1.00,1.00,1.50, 1.00,1.50,1.50,1.00,1.50,1.00,1.00,1.00, 1.50,1.00,1.00,1.50]
$\boldsymbol{r}$	$r_i(t) \sim Uniform[0.0,2.0]$
c	1.0
s	1.0

TABLE 3. Fixed point representation of the NN.

Parameter	Fixed Point Representation (I,F)
W	(1,11)
Z	(4,8)
A	(4,8)
dZ	(3,10)
dW	(6,6)

We first compare the performance of the employed DRL-based algorithm against the following offloading schemes:

- **Optimal**: We compute the optimal $\boldsymbol{k}(t)$ vectors corresponding to each of the 2^{20} possible offloading actions using (14) and compute the resulting weighted sum of the experienced task delays. The $\boldsymbol{x}(t), \boldsymbol{k}(t)$ pair with the minimum delay is selected as the offloading and resource allocation action.
- **Random**: We select K = 21 random offloading action candidates $\boldsymbol{x}(t)$, compute the optimal $\boldsymbol{k}(t)$ vectors and compute the resulting weighted sum of the experienced task delays. The best $\boldsymbol{x}(t), \boldsymbol{k}(t)$ pair is selected as the offloading and resource allocation action. The reason we use K candidate actions is for fairness, since the accelerated algorithm also evaluates K candidate actions (which are obviously not generated randomly).
- **User-Based**: Each User decides on their offloading action after comparing the local execution delay of their task with the offloading one, without considering the other Users' decisions. However, in order to calculate the offloading execution delay of their task the Users should have a priori knowledge of the ES resources that will be allocated to them. A greedier scheme according to which each User assumes that all the ES resources will be devoted for the execution of their task (equivalently each User decides on their offloading action assuming no other User will offload their task to the ES) resulted in worse performance in comparison with the random scheme and thus is not presented. Instead, a more conservative approach (User-Based Scheme) according to which each User assumes that all the other Users will also offload to the ES and calculates the expected ES resources under this assumption resulted in better performance and is presented for comparison with the accelerated DRL-based algorithm.

The comparison of the average delay of the different offloading and resource allocation schemes is presented in Fig. 18. The accuracy of the proposed mechanism is directly correlated with the delay minimization since optimal offloading actions result in minimal delay. Please note that since the problem parameters of Table 2. are normalized with respect to abstract values, the resulting delay does not have units. As illustrated in the figure, the accelerated algorithm achieves near-optimal performance after 10000 timesteps. However, the performance is already satisfactory after 5000 timesteps. Please note that the y-axis corresponds to the moving average of the delay in the last 1000 timesteps. Fig. 19 displays the effect that the algorithmic modifications of section IV. and the Training Lag described in section V. have on the algorithm's performance.

According to Fig. 18 the training has been completed after timestep 10000 and no important performance increase is observed after that point. Thus, the y-axis of Fig. 19 (which is used as a training accuracy indicator) corresponds to the average delay from timestep 10000 until timestep 17500. The FLP bar depicts the accuracy of the floating-point version of the algorithm without any of the discussed modifications for efficient hardware implementation.

Simulation suggests that the distribution of training over time and the calculation of the Sigmoid function using LUTs do not affect the algorithm's accuracy. A slight accuracy decrease is observed due to the Training Lag, which is

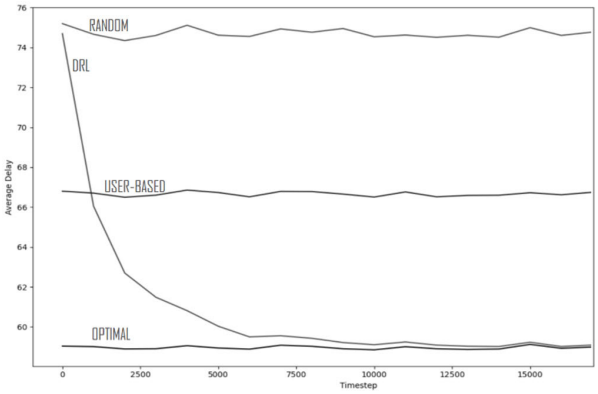

FIGURE 18. Comparison of offloading and resource allocation schemes.

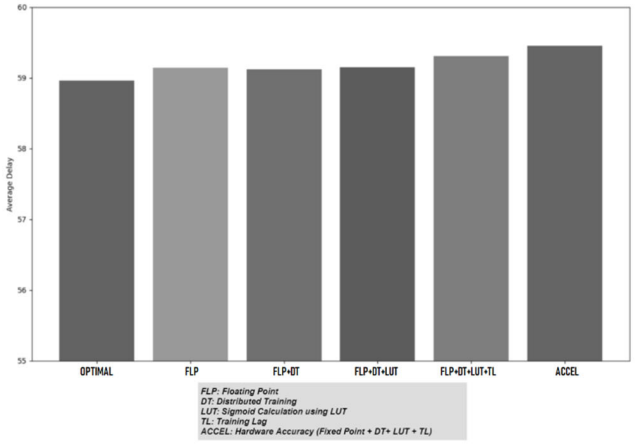

FIGURE 19. Effect of algorithmic modifications to the accuracy as illustrated by the resulting delay.

however insignificant for our use-case. When using our Accelerator for other problem formulations, consideration should be given to the accuracy-acceleration tradeoff due to the Pipelined Training Module. If a potential significant accuracy drop due to the Training Lag is observed, a pipeline flush between subsequent batches should be considered. Finally, an expected additional slight decrease in the accuracy is observed due to the usage of fixed-point arithmetic in the final hardware implementation. Overall, the algorithmic modifications have little effect to the accuracy and the Accelerator's performance remains near-optimal.

VII. IMPLEMENTATION AND LATENCY-ENERGY EVALUATION

The hardware implementation and performance evaluation was conducted using the Zynq Ultrascale+ MPSoC ZCU104 Evaluation Kit, which features the XCZU7EV-2FFVC1156 MPSoC, a quad-core ARM Cortex-A53 processor and a dual-core Cortex-R5 processor. The designed Accelerator (on the available MPSoC FPGA) was used as a coprocessing element for the Cortex-A53 CPU. The performance of the designed Accelerator is compared against implementations based on a high-end Cortex-A53 ARM core and an AMD Ryzen 5 4600H processor. In the CPU-based implementations, the CPU is responsible for both the NN related computations and the quantization and action evaluation-selection processes. In the Accelerator based implementation, the NN related computations are conducted on the FPGA while the quantization and action evaluation-selection processes are performed on the CPU (Cortex-A53).

The AXI4-Stream protocol is employed for the data transfers in mode 1 between the CPU and the Accelerator, which are controlled by an AXI-Direct Memory Access (AXI-DMA). Since we did not impose any latency constraints in the initialization mode (mode 0) during which the Weight's initial values and the contents of the Sigmoid LUTs are passed and stored in the Accelerator it was not necessary to employ the AXI4-Stream protocol for mode 0.

Table 4 displays the utilization of the FPGA's resources. Please note that the relatively high utilization of the BRAMs, in comparison with the other resources is due to the partitioning of the Weights' Memories. In scenarios with a larger number of weights (larger NN), but with the same partitioning factors the utilization is not expected to change (although for the parallelization of the memory accesses we are using distinct BRAMs, most of the space in each individual BRAM is not utilized). Therefore, our design can be also extended to support larger NNs. However, to increase the level of parallelism in larger NNs both the BRAM and the DSP utilization are expected to increase. Thus, the main factor determining scalability is not the size of the NN (assuming it is compatible with the typical configurations in the applications of interest), but rather the level of replication-based parallelism. Please note that the parallel computation of the gradients of more samples at every timestep is also possible without increasing the BRAM utilization (no further partitioning of the weight memories is needed to achieve sample-level parallelism).

It is worth noting that although the delay estimates of the HLS tool were accurate, some resource utilization estimates were much higher than the real utilization figures, as provided by the Vivado tool after the place and route process. Specifically, the LUT and FF estimates exceeded the practical utilization by approximately a factor of 3. The DSP utilization estimate, on the other hand, was accurate.

TABLE 4. Part resource utilization.

Resource	Utilization	Available	Utilization%
LUT	62722	230400	27.22
LUTRAM	14133	101760	13.89
FF	29131	460800	6.32
BRAM_18K	229	312	73.40
DSP	816	1728	47.22

Regarding the CPU based implementations, the high-performance C/C++ code was compiled using the GCC

compiler with the -Ofast optimization option enabled, targeting the architecture of the employed CPU. We present the results of the single-threaded execution using floating point arithmetic. The main reason for choosing the single-threaded execution lies in the fact that the thread creation and synchronization overhead introduced by a multi-threaded execution resulted in a larger latency due to the small size of the NN.

Table 5 compares the latency per timestep of the 3 implementations. Each timestep includes the NN related computations and the quantization and action evaluation-selection processes. The NN computations consist of an inference procedure (FP) with one sample and a training procedure (FP, BP and GC) with 8 samples per timestep (since the size of the batch sampled at each timestep is set to be 8).

TABLE 5. Latency per timestep.

Implementation	NN computations (μs)	AXI4-Stream data transfer (μs)	Quantization Evaluation Selection (μs)	Total (μs)
ARM Cortex-A53	830	NA	6	836
AMD Ryzen 5 4600H	44.60	NA	0.85	45.45
Cortex-A53 and Accelerator	4.30	2.06	6	12.36

As Table 5 illustrates, the designed Accelerator decreases the NN computations latency by a factor of 200 when compared to an ARM Cortex-A53 implementation and by a factor of 10 when compared to an AMD Ryzen 5 4600H implementation. Although the quantization and action evaluation-selection processes account for a tiny fraction of the total execution time in the CPU-based implementations, the acceleration achieved by our design decreases the NN computations latency to such a degree that the non-accelerated quantization and action evaluation-selection processes (which are executed on the Cortex-A53) account for ∼50% of the total execution time. Therefore, the total latency per timestep is decreased by a factor of 67 when compared to the ARM Cortex-A53 CPU-based implementation and by a factor of ∼4 when compared to the AMD Ryzen 5 4600H CPU-based implementation. In order to decrease the overhead introduced by the quantization and action evaluation-selection processes practical implementations could employ a CPU with higher floating point math capabilities in comparison to the ARM Cortex-A53 with which the ZU7EV device is equipped in the Zynq Ultrascale+ MPSoC ZCU104 Evaluation Kit. Another option is the design of an application specific IP for the hardware acceleration of the aforementioned processes. As mentioned in section I., this option can also be employed in cases where the environment simulator-evaluator has higher computational requirements, but this falls beyond the scope of this paper.

Table 6 compares the energy efficiency of the Cortex-A53 CPU-based implementation against the Cortex-A53 with the Accelerator as a co-processing element implementation. As it is shown, the energy dissipation in our design is decreased by a factor of 11.

TABLE 6. Energy dissipation per timestep.

Implementation	Energy Dissipation per timestep (μJ)
ARM Cortex-A53	83.64
Cortex-A53 and Accelerator	7.68

It should be mentioned here that no comparison with FPGA implementation was conducted since as analyzed in Section II. (Existing Work) very few such (FPGA-based) accelerators exist in the literature and the ones combining both inference and training [7], [9] exhibit a very complex NN structure, the computational complexity of which does not allow for real-time operation that our accelerator readily achieves.

VIII. CONCLUSION

We have shown that it is feasible to develop a hardware DRL Accelerator to address MIP problems at the Edge in real-time, targeting concurrently both the inference and training operations. The features of the Accelerator include training-inference parallelism, pipelined Training, on-chip Weights and Replay Memory, multi-level replication-based parallelism and DRL algorithmic modifications such as distribution of training over time. The performance of the Accelerator was demonstrated upon a real-time Edge Computing use-case (task offloading and resource allocation) achieving near-optimal accuracy and ultra-low latency. Comparisons against implementations based on a high-end ARM core (Cortex-A53) and a high-end desktop (AMD Ryzen 5 4600H) indicate a decrease in the training-inference execution latency by a factor of 200 and 10 respectively. The proposed accelerator design methodology can be extended to address more DRL algorithms (employing the Experience Replay Technique) with hard real-time constraints. Towards this purpose we additionally provide HLS guidelines for the implementation of the specified functionality.

REFERENCES

[1] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, vol. 1, no. 2. Belmont, MA, USA: Athena Scientific, 1995.

[2] Z. Zhou, H. Liao, X. Zhao, B. Ai, and M. Guizani, "Reliable task offloading for vehicular fog computing under information asymmetry and information uncertainty," *IEEE Trans. Veh. Technol.*, vol. 68, no. 9, pp. 8322–8335, Sep. 2019.
[3] H. Liao, Z. Zhou, S. Mumtaz, and J. Rodriguez, "Robust task offloading for IoT fog computing under information asymmetry and information uncertainty," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2019.
[4] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," in *Machine Learning*. Dordrecht, The Netherlands: Kluwer, 1992, pp. 69–97.
[5] L. Huang, S. Bi, and Y.-J.-A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 11, pp. 2581–2593, Nov. 2020.
[6] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4005–4018, Jun. 2019.
[7] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "FA3C: FPGA-accelerated deep reinforcement learning," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 499–513.
[8] M. J. Li, A. H. Li, Y. J. Huang, and S. I. Chu, "Implementation of deep reinforcement learning," in *Proc. 2nd Int. Conf. Inf. Sci. Syst.*, 2019, pp. 232–236.
[9] J. Yang, S. Hong, and J.-Y. Kim, "FIXAR: A fixed-point deep reinforcement learning platform with quantization-aware training and adaptive parallelism," 2021, *arXiv:2102.12103*.
[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.
[11] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing," *Digit. Commun. Netw.*, vol. 5, no. 1, pp. 10–17, 2018.
[12] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for MEC," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2018, pp. 1–6.
[13] Y. Liu, H. Yu, S. Xie, and Y. Zhang, "Deep reinforcement learning for offloading and resource allocation in vehicle edge computing and networks," *IEEE Trans. Veh. Technol.*, vol. 68, no. 11, pp. 11158–11168, Nov. 2019.
[14] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. NIPS*, vol. 99, 1999, pp. 1057–1063.
[15] H. Lu, X. He, M. Du, X. Ruan, Y. Sun, and K. Wang, "Edge QoE: Computation offloading with deep reinforcement learning for Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9255–9265, Oct. 2020.
[16] G. Qu, H. Wu, R. Li, and P. Jiao, "DMRO: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 3, pp. 3448–3459, Sep. 2021.
[17] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 242–253, Jan. 2021.
[18] Y. Wang, M. Wang, B. Li, H. Li, and X. Li, "A many-core accelerator design for on-chip deep reinforcement learning," in *Proc. 39th Int. Conf. Comput.-Aided Design*, Nov. 2020, pp. 1–7.
[19] C. Kim, S. Kang, S. Choi, D. Shin, Y. Kim, and H.-J. Yoo, "An energy-efficient deep reinforcement learning accelerator with transposable PE array and experience compression," *IEEE Solid-State Circuits Lett.*, vol. 2, no. 11, pp. 228–231, Nov. 2019.
[20] C. Kim, S. Kang, D. Shin, S. Choi, Y. Kim, and H.-J. Yoo, "A 2.1TFLOPS/W mobile deep RL accelerator with transposable PE array and experience compression," in *IEEE ISSCC Dig. Tech. Papers*, Feb. 2019, pp. 136–138.
[21] S. Ruder, "An overview of gradient descent optimization algorithms," 2016, *arXiv:1609.04747*.
[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
[23] S. W. Golomb, *Shift Register Sequences: Secure and Limited-Access Code Generators, Efficiency Code Generators, Prescribed Property Generators, Mathematical Models*. Singapore: World Scientific, 2017.
[24] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI Conf. Artif. Intell.* 2016, vol. 30, no. 1, pp. 2094–2100.
[25] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 3, pp. 587–597, Mar. 2018.

GERASIMOS GEROGIANNIS received the Diploma degree (B.Eng. and M.Eng.) in electrical and computer engineering from the University of Patras, Patras, Greece, in 2021. He is currently pursuing the Ph.D. degree in electrical and computer engineering with the University of Illinois at Urbana–Champaign, Urbana, IL, USA.

He is currently a Research Assistant with the i-acoma Group, University of Illinois at Urbana–Champaign. His research interests include novel architectures for machine learning applications, machine learning methods in computer architecture, machine learning aided computer architecture design, applied machine learning, and hardware assisted edge computing.

MICHAEL BIRBAS received the Diploma and Ph.D. degrees in electrical and computer engineering from the University of Patras, Greece, in 1985 and 1991, respectively.

From 1992 to 1999, he was the Research and Development Manager and the Co-Founder of Synergy Systems, a high-tech start-up microelectronics-related company. From 1999 to 2002, he was a Research and Development Manager with GIGA Hellas S.A-an Intel company, where he was involved with the design of 10–40-Gbit/s Ser/Des components and system solution demonstrators for optical transport networks. From 2002 to 2006, he was with the Applied Electronics Laboratory Team, University of Patras, working in high-speed mixed integrated circuits for communication applications and in VLSI implementations of FEC algorithms. From 2006 to 2014, he was the Research and Development Manager and the Co-Founder of Analogies S.A., a high-tech start-up company focusing on the design of high-performance, multi gigabit, mixed signal/RF, and digital DSP silicon IP. Since 2014, he has been an Assistant Professor with the Department of Electrical and Computer Engineering, University of Patras, with his research activities focusing in smart grid applications, AI algorithms and accelerators, digital twins, and advanced embedded designs. He is the author/coauthor of several publications in international journals and conferences (more than 100) and the co-inventor of a number of patents (five of them registered at USPTO).

AIMILIOS LEFTHERIOTIS is currently pursuing the bachelor's degree with the Electrical and Computer Engineering Department, University of Patras.

His research interests include embedded systems, C/C++ programming, FPGA-based acceleration of scientific computations, hardware architecture, and high-level synthesis.

ELEFTHERIOS MYLONAS received the Diploma degree from the Electrical and Computer Engineering Department, University of Patras, in 2019. He is currently pursuing the Ph.D. degree with the University of Patras.

He holds research experience in the area of 5G communications for smart grid and industry 4.0 applications from participating in Horizon 2020 projects. His main research topic is special purpose edge processors for digital twin applications. His research interests include the next generation IoT, digital twins, cyber-physical systems, and intelligent edge/AI-enabled edge computing.

NIKOLAOS TZANIS received the Diploma degree in electrical and computer engineering from the University of Patras, Greece, in 2011, where he is currently pursuing the Ph.D. degree in the field of smart grid applications on embedded systems.

From 2015 to 2019, he worked as an Embedded Software Engineer with Intracom Telecom SA. He is also a Researcher at the Department of Research Technology and Development (DRTD), Greek TSO (IPTO). He holds research experience in the field of 5G communications for smart grid and industry 4.0 applications. His main research topic is hardware acceleration for real-time smart grid applications with emphasis in transient state estimation algorithms. His research interests include also cyber-physical systems, hardware assisted edge computing, and next generation industrial communication networks.

ALEXIOS BIRBAS received the M.S.E.E. and Ph.D. degrees from the University of Minnesota, Minneapolis, MN, USA, in 1986 and 1988, respectively. He has also held faculty positions at the Assistant Professor level at the University of Minnesota and at INPG, Grenoble, France. He has also co-founded two microelectronics related spin-off companies and has supervised 20 Ph.D. theses. He is currently a Professor of electronics with the Department of Electrical and Computer Engineering, University of Patras, and the Director of the Applied Electronics Laboratory, Patras Greece. He has also served as a consultant to the industry and has held industrial managerial positions. He has published over 150 papers in scientific journals and conference proceedings. His research interests include device electronics, optoelectronics, noise and fluctuation problems in electronics, RF and mixed signal high-speed circuit design and implementation of sensor read out circuits, smart grid electronics and transactive energy systems, smart metering, the 5G enabled IoT and Industry 4.0 real time applications, edge computing accelerators, and cyber physical systems.