PPOAccel: A High-Throughput Acceleration Framework for Proximal Policy Optimization

Yuan Meng, Sanmukh Kuppannagari, Rajgopal Kannan, Viktor Prasanna

Abstract—Reinforcement Learning (RL) is a major branch of AI that enables agents to learn optimal decision making via interaction with the environment. Proximal Policy Optimization (PPO) is the state-of-the-art policy optimization based RL algorithm which achieves superior overall performance on various benchmarks. A PPO agent iteratively optimizes its policy - a function which chooses optimal actions approximated by a DNN, with each iteration consisting of two computationally intensive phases: Sample Generation - where agents inference on its policy and interact with the environment to collect data, and Model Update - where the policy is trained using the collected data. In this paper, we develop the first high-throughput PPO accelerator on CPU-FPGA heterogeneous platform. Our unified systolic-array based design accelerates both the inference and the training of the deep neural network used in a RL algorithm, and is generalizable to various MLP and CNN models across a wide range of RL applications. We develop novel optimizations to simultaneously reduce data access and computation latencies, specifically: (a) optimal data flow mapping to systolic array, (b) novel memory-blocked data layout to enable streaming stall-free data access in both forward and backward propagations, and, (c) a systolic array compute sharing technique to mitigate load imbalance in the training of two networks. We evaluate our design on widely used robotics and gaming benchmarks, achieving 1.4×–26× and 1.3×–2.7× improvements in throughput, respectively, when compared with state-of-the-art CPU/CPU-GPU implementations.

Index Terms—Reinforcement Learning, Hardware Accelerators, FPGA

1 Introduction

Reinforcement Learning (RL) is an area of Artificial Intelligence (AI) that constitutes a wide range of algorithms which span the Observe, Orient, Decide and Act phases of autonomous agents [1]. RL has found widespread success in the implementation of autonomous agents in domains such as self-driving cars [2], surveillance [3], wearable healthcare devices [4], etc.

An important class of RL consists of policy optimization [5] methods, where the policy is modeled as a function of probability distribution of actions conditional on states. The agent learns the policy by pushing up the probabilities of actions with high rewards. Proximal Policy Optimization (PPO) [6] is considered the state-of-the-art policy optimization method. PPO introduces a clipped objective that discourages the new policy from stepping far away from the old policy in each iteration. It achieves better sample complexity (number of samples required for convergence) and reliable performance [6].

Heterogeneous architectures consisting of CPUs and FPGAs have become popular in accelerating memory and compute intensive algorithms [7], [8]. Such platforms are integrated with abundant high-bandwidth memory and Interconnections such as CCIX [9] which provide coherent shared-memory access between the processor and accelerators. PPO consists of several computational kernels some of which (e.g. neural network propagation) can benefit from fine-grained parallelism of FPGAs, whereas others (e.g. Advantage and surrogate loss calculation) are better suited for CPUs.

PPO showcases better sample efficiency (the amount of environments steps required for an agent to reach a certain reward) compared with other policy optimization methods [6]. While this suggests that PPO requires fewer iterations given fixed amount of environmental steps in each iteration, each iteration is computationally intensive as it involves multiple epochs of neural network training [6]. Additionally, hardware acceleration of PPO is challenging as PPO has different data access patterns in forward and backward propagations requiring novel memory layout techniques to enable streaming data access. Moreover, different iterations needed for convergence in training the value and policy networks may lead to imbalanced workload in hardware.

This work is an extension of [10] in which we develop the first high-throughput PPO accelerator targeting CPU-FPGA heterogeneous platforms for applications that use Multi-Layer Perceptron (MLP) for policy representation, targeting both Sample Generation and Model Update phases. In this work, we generalize our design to support Convolutional Neural Networks (CNN) based policies. To accomplish this, we use the im2col algorithm and take advantage of the identical underlying computation kernel between MLP and CNN - General Matrix-Matrix Multiplication (GEMM). Additionally, we develop a novel technique to optimally map data flow to the systolic array to improve overall hardware utilization. Specifically, the major extended contributions of this work are:

- We identify the key computation and memory requirements of kernels in MLP- and CNN- based PPO, and map them onto CPU-FPGA platform.
- We design a unified systolic GEMM Core that accelerates inference and training of both MLP and CNN models.
- We propose a Layout Transformation Module (LTM) to convert input Tensor into Toeplitz matrix on-the-fly, efficiently mapping convolution operations onto the GEMM Core.
- We develop a design space exploration scheme that optimally maps the GEMM Core dataflow to improve effective

hardware utilization across all layers of the DNN.
- We implement the PPO accelerator on a heterogeneous platform consisting of state-of-the-art CPU and FPGA. The accelerator achieves up to 27.1× and 26× improvement in throughput compared with state-of-the-art CPU-only and CPU-GPU implementations, respectively.

2 Background & Motivation

2.1 Proximal Policy Optimization (PPO)

We consider the standard RL setting defined as a Markov Decision Process $\{\mathcal{S},\mathcal{A},\mathcal{R},P,\gamma,s_0\}$ that specifies a sequence of states $s_t \in \mathcal{S}$, actions $a_t \in \mathcal{A}$, rewards $r_t \in \mathcal{R}$, and initial state s_0 [11]. Starting from s_0, we draw observations at current state $s_t \in \mathcal{S}$, actions a_t from the policy $\pi(a_t|s_t)$, rewards r_t and the next state s_{t+1} from the environmental interaction. The objective is to find a policy π that maximizes the expected rewards. Deep RL models the agent's decision-making objectives (state-value, policy, etc.) with deep neural networks (DNN).

Typical implementations of PPO involve two DNN models, the policy network π_θ, and the value network V_ϕ. The value network approximates the value of a state using the expected future reward, $\hat{R}_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$. The objective of the value network is to minimize the error between its value prediction and actual $\hat{R}_t$ collected from the Sample Generation phase: $\mathcal{L}^{MSE} = \sum_{t\in[0\ldots T-1]} \left[V_\phi(s_t) - \hat{R}_t\right]^2$. The policy network represents a stochastic policy π_θ which returns a conditional probability distribution of actions given a state. The value network approximation is used in the objective of training the policy network to guide the policy to output actions that lead to "high-valued" states. To do this, an advantage function $\hat{A}$ is defined to indicate the advantage of a specific action a_t over average [6]: $\hat{A}_{\pi_\theta,t} = \hat{R}_t - V_\phi(s_t)$. PPO prevents large changes to the action distribution in a single update using a clipped objective function for policy [6]:

$$clip(\epsilon, \hat{A}_t) = \begin{cases} (1+\epsilon)\hat{A}_t & \hat{A}_t \geq 0 \\ (1-\epsilon)\hat{A}_t & \hat{A}_t < 0 \end{cases}$$
$$\mathcal{L} = \mathbb{E}_{\pi_{\theta_k}}\left[\min\left(\frac{\pi_\theta\left(a_t|s_t\right)}{\pi_{\theta_k}\left(a_t|s_t\right)}\hat{A}_t, clip\left(\epsilon, \hat{A}_t\right)\right)\right] \quad (1)$$

where π_{θ_k} is the old policy before the update and $0 \leq \epsilon \leq 1$ determines the update size. The function $g(\epsilon, \hat{A})$ modifies the objective by removing the incentive for moving the probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$ outside of the interval $[1-\epsilon, 1+\epsilon]$ [6].

In each iteration, PPO alternates between **(1) Sample Generation phase**: sample experiences through interaction with the environment, where each agent performs inference using its model and sequentially interacts with the environment to collect a fixed-length trajectory of $[s_t, \pi_{\theta_t}(s_t), a_t, r_t]$ and **(2) Model Update phase**: optimize the objective functions for policy and value networks by mini-batch Stochastic Gradient Update [12], using the experience datasets collected from the Sample Generation phase.

In the following sections, we consistently use the same set of hyper-parameters $[N, T, M, K]$ to characterize computation in one iteration: N is the number of parallel agents, T is the trajectory length of each agent, K denotes the number of training epochs and M is the mini-batch size. Note that in each training epoch, all $N \times T$ samples collected during Sample generation phase are covered by performing $\frac{N\times T}{M}$ rounds of mini-batch update.

2.2 Deep RL in Real-World Applications

Training Deep RL agents in the field for most real-world applications such as robotics, medical services etc. is infeasible due to the risk posed by unsafe policies as well as the large number of environmental interactions needed [7]. Therefore, such agents are trained using a simulation environment. Typically, multiple agents interact with multiple environments simulated in parallel threads to generate data (state, action, rewards) which is then used to train the neural network [13]. This tight coupling of Sample Generation with simulated environments and Model Update in each iteration is called the "Training in Simulation" process [7], which is the most time-consuming portion of Deep RL training, usually taking millions of iterations to complete [5], [6].

2.3 DNN Models for Deep RL

The DNN model used in a Deep RL algorithm is determined by the application domain. For robotics, simple MLPs are widely used [14]. For vision-based applications such as gaming, CNNs are used to extract higher level features from the input images [15]. MLPs and CNNs account for the vast majority of the applications in Deep RL [16] and are hence the focus of this work.

GEMM is the core operation in both MLP and CNN models (im2col method) allowing us to utilize most of the optimizations developed in [10] for CNN based Deep RL. However, CNNs impose some additional challenges: 1) They require additional format transformation from feature maps to input matrices based on im2col method. 2) The larger size of intermediate results induces off-chip data traffic. To address these challenges, we develop several CNN specific optimizations in this paper including a lightweight data transformation module to support format transformations along with double-buffering to overlap computation and communication (Section 3).

2.4 Limitations in PPO acceleration on GPU

While several frameworks deploy Deep RL on GPU for acceleration [17], [18], PPO computations suffer the following performance bottlenecks on these devices:

(a) Tight Coupling between Inference and Training [6], [10]: In PPO, the Sample Generation (requiring neural network inference) and Model Update (requiring training of neural network) have strict temporal interdependencies. Thus, neural network implementations which can perform high throughput training as well as low latency inferences are needed. GPUs are well suited for the former but less suited for the latter.

(b) Batch-size limitations: In DNN computations, smaller batch sizes lead to poorer utilization of massive GPU computing resources as the computations cannot hide memory access latency [19]. In theory, one can increase the batch size to improve throughput and parallelism (e.g. distributed RL [16], [20], [21]). In practice, however, large batch sizes adversely impact the quality of training [22], [23] in terms of more environmental steps required to achieve the target reward, prolonging the execution time of each iteration. On FPGA platform, we customize the accelerator to evenly minimize both computation and memory access latency.

(c) Kernel overheads: Each mini-batch training task requires all-reduce over the Streaming-Multiprocessors(SM) for gradient computation, which introduces synchronization overheads. Additionally, popular Deep Learning frameworks such as Tensorflow-GPU [24] often introduces kernel-launch overheads. These overheads become more obvious when the amounts of computation are

small. In PPO, repeated typically small-batch kernel launches lead to significant synchronization and kernel launch overheads which cannot be hidden. In comparison, FPGA accelerator introduces no kernel-launch overheads and negligible control overheads.

(d) High memory-access latency: Memory requirement of training even smaller MLP based Deep RL (2.75Mbits) is typically larger than the local memory of GPU cores which can be accessed with low latency. Thus, Deep RL on GPU relies on data accessed from higher levels of GPU memory-hierarchy which have higher latencies [25] (∼20 cycles:shared memory, ∼28 cycles:L1 cache, ∼200 cycles:L2 cache). We take advantage of FPGA's larger available on-chip memory (∼60Mbits BRAM) as well its programmability to implement optimal data layout to achieve ultra-low-latency access to DNN parameters.

When launching PPO traning tasks on a TITAN Xp GPU, we observe more than 80% synchronization and kernel launch overheads using small-sized MLP policies, and more than 55% various overheads using larger CNN policies. *The fine-grained parallelism, and large and configurable on-chip memories offered by state-of-the-art FPGA device make them a suitable candidate to address the above mentioned challenges and implement high performance accelerators for Deep RL.*

3 Framework Overview

PPO executes iteratively until certain convergence criteria is met. We partition the tasks in each PPO iteration such that the FPGA executes the computationally intensive DNN propagation operations which can benefit from fine grained parallelism. The computation of $\hat{A}_t, \hat{R}_t$ and objective are executed on CPU as they would occupy significant FPGA area and introduce non-trivial communication overhead if mapped on FPGA, while having lower computation intensity compared to the other operations (e.g., DNN propagation). Fig. 1 shows the task scheduling on the CPU-FPGA heterogeneous platform. The Host CPU communicates with

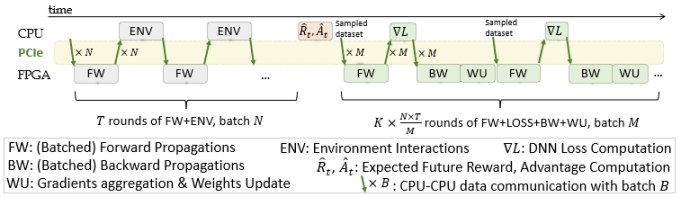


Fig. 1: CPU-FPGA Tasks Mapping and Scheduling

FPGA via PCIe interconnection. In Sample Generation phase, each inference request is initiated by N parallel agents on CPU by sending their states s_t to FPGA. After executing forward propagation (FW) using the states as inputs, N output actions are sent back to the host CPU for next-step environmental interactions. At the end of Sample Generation, the CPU computes $\hat{R}_t, \hat{A}_t$ for all $N \times T$ samples collected, and initiates a training request by sending M states (sampled from $N \times T$ samples) to the FPGA. The $\{$FW-LOSS-BW-WU$\}$ process is repeated to cover all $N \times T$ samples over K epochs, after which one iteration of the algorithm is complete and it repeats the next iteration.

3.1 Architecture Overview

As shown in Fig. 2, the accelerator consists of two Compute Units (CU), one for the value network and one for the policy network. Outputs for both networks are sent to the CPU (the output of policy

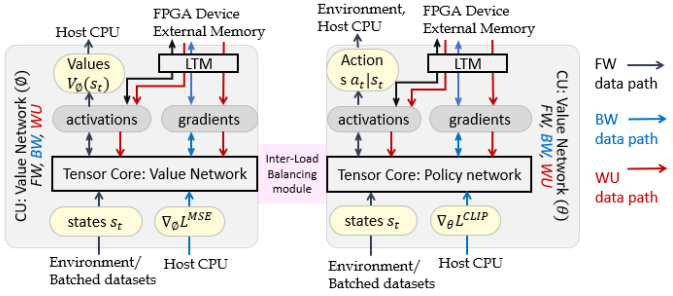


Fig. 2: Overview of Accelerator Design: CUs and their interactions with the Host/ Device External Memory

network is used to derive the agent's next-step action, and the output of value network is sent to CPU for computing the policy network Loss later in Model Update phase). CUs are re-used for both Sample Generation and Model Update phases without hardware reconfiguration. Sizes of DNNs used in RL applications are typically less than the on-chip memory resources of data-center FPGAs (∼ 1.8Mbits for the largest MLP typically used [6], [16] in Robotics benchmarks, and 54Mbits for CNN policy used in [6], [18]). Thus, we keep all the DNN weights on-chip and reuse them during all the inference and training processes. The memory requirement for DNN intermediate results (e.g. activations and gradients) is proportional to the inference/training batch sizes. While we demonstrate our generalized design assuming activations and gradients are stored in external memory as shown in Fig. 2, when possible (e.g. small MLP based policies or small batch sizes) we store all intermediate results on-chip to avoid unnecessary communication overhead. A Layout Transformation Modules (LTM) is integrated in each CU to support layout transformation in CNN.

3.2 Customized Compute Units

In this section, we show the detailed design for each component of the CU. Fig. 3 shows the detailed architecture of a single CU. It is composed of a 2D-systolic-array based GEMM Core to perform (1) FW, (2) BW and (3) gradients computation for WU; activation and activation derivative modules; and buffers to store network weights and intermediate results ($\boldsymbol{z}^l$ buffer for immediate matrix products, $\boldsymbol{a}^l$ buffers for FW generated activation results, and $\boldsymbol{\delta}^l$ for BW generated local gradients). The GEMM Core is composed of $P_{sys} \times P_{sys}$ Processing Elements (PE), each performing a MAC (Multiply-Accumulate) operation. The WU module is an array of adders that update the DNN weights using the gradients computed by the GEMM Core. The Hadaard Multiplication Module is used to calculate the Hadamard products of BW intermediate results and activation results in FW.

In the following paragraphs, we consider both fully-connected (FC) and CONV layers. The propagation rules of FC layers can be naturally expressed as GEMM operations. Similarly CONV layer propagations are commonly represented as GEMM using im2col method as follows: Denote a CONV layer using $(H_1, H_2, C_{in}, F_1, F_2, S, O_1, O_2, C_{out})$, where H_1, H_2, C_{in} are dimensions of the input feature maps, $F_1 \times F_2$ is the size of each kernel, S is the stride size, and O_1, O_2, C_{out} are dimensions of the output feature map. The C_{in} $F_1 \times F_2$ sized kernels corresponding to each output channel C_{out} are concatenated together into a row of the weight matrix of size $(C_{out}, C_{in} \times F_1 \times F_2)$,

and each group of C_{in} corresponding windows of input feature maps are stacked into a column of the input activation/gradient Toeplitz matrix of size $(C_{in} \times F_1 \times F_2, O_1 \times O_2)$. Now, the propagation of a FC layer (L^{l-1}, L^l), is mathematically equivalent to a CONV layer with $C_{in} = |L^{l-1}|$, $C_{out} = |L^l|$, $H_1 = H_2 = F_1 = F_2 = O_1 = O_2 = 1$. Therefore, we uniformly use the notations of dimensions for a CONV layer to represent the GEMM computations for both types of DNN layers.

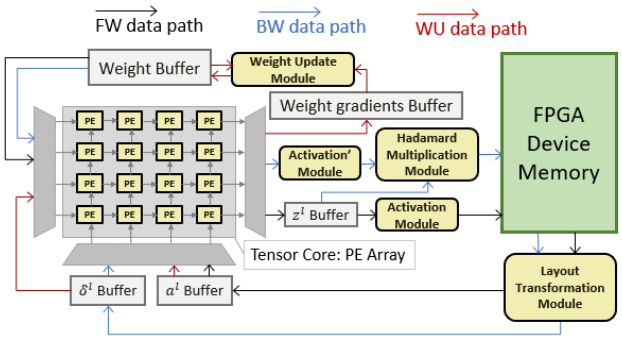


Fig. 3: CU architecture

In the Sample Generation phase, only FWs are performed and only the last layer results are collected and sent to CPU. Since N agents perform FW independetly in each time stamp, to fully utilize the resources of 2D systolic array and exploit parallelism between input vectors or feature maps, we 'vectorize' the N input state vectors/feature maps into a large matrix. According to feed forward rule (Note: For simplicity, in Eq. (2)(3)(4), the general $*$ operator is used to represent both direct matrix multiplication in a fully-connected MLP layer and im2col-GEMM in a CONV layer):

$$\text{FW}: \boldsymbol{z}^l = \boldsymbol{W}^l * \boldsymbol{a}^{l-1} \; ; \; \boldsymbol{a}^l = \sigma(\boldsymbol{z}^l) \tag{2}$$

In each FC/CONV layer, a piplined matrix-matrix multiplication is performed by the systolic array. The outputs of each hidden layer are buffered and sent to the GEMM Core as inputs of the next layer. The output of the final layer is sent to the CPU.

In the Model Update phase, we use mini-batch Stochastic Gradient Descent/Ascent (SGD/SGA) [26]. Each batched propagation is composed of M independent samples. In the FW, BW and weight gradient computation processes, we apply similar vectorization technique as that used in inference tasks. Error values are back-propagated to all hidden layers and the activation gradients of each hidden layer are calculated. In the case of CNN, local activation gradients $\boldsymbol{\delta}$ of layer l can be obtained by convolving the local gradients of the previous layer $(l-1)$ with its own convolution kernel:

$$\begin{aligned} &\text{BW}: \boldsymbol{\delta}^l = \sigma'\left(\boldsymbol{z}^l\right) \odot (\boldsymbol{\delta}^{l+1}\boldsymbol{W}^{l+1}) \\ &\odot: \text{Hadamard (element-wise) product [27];} \end{aligned} \tag{3}$$

During the backward propagations (i.e. transposed convolutions), the original kernel tensors are flipped and the number of input/output channels are switched. In case of MLP or FC layer, the weight matrices are transposed. Applying the basic chain rule, weight gradients Δw can be obtained by convolving the local gradients of the current layer with forward-pass activations of the last layer. The WU process is composed of computing weight gradients followed by updating the weights at a certain learning rate.

$$\begin{aligned} &\text{WU}: \Delta \boldsymbol{w} = \frac{\partial C}{\partial \boldsymbol{w}^L} = \boldsymbol{\delta}^l \boldsymbol{a}^{l-1} \\ &\boldsymbol{w}^l = -\alpha \Delta \boldsymbol{w} + \boldsymbol{w}^l \; ; \; \alpha\text{: Learning rate;} \end{aligned} \tag{4}$$

The WU can be performed after all the local gradient calculations (BW) are finished. In MLP networks or FC layers, the WU process aggregates all products between every neurons in adjacent layers, which is essentially another round of Matrix-Matrix Multiplication of local activations and local gradients; In CONV layers the WU process is essentially dilated convolution [28], which can be transformed into GEMM operation with im2col as well. Therefore we feed these data into the systolic array, with data flow path shown in red color in Fig. 3.

3.2.1 Layout Transformation Module (LTM)

When accelerating convolutional (CONV) layers using im2col [29] transformation from input 3D tensors to 2D Toeplitz data layout is required. However, the output (activation/gradient) feature maps of all hidden CONV layers are in spatial 3D tensor layout, which cannot be directly served as the input for the next layer requiring 2D Toeplitz layout. To avoid sending them to CPU for

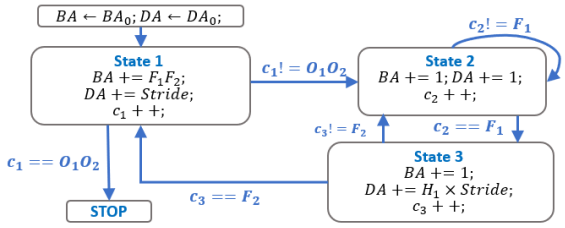


Fig. 4: LTM Control Logic

pre-processing between every layer which introduces significant PCIe data transfer, we introduce a lightweight layout transformation module (LTM) in the downstream of the memory controller and the upstream of the output buffer. As shown previously in Fig. 3, we write all outputs directly into the external memory in the original 3D tensor layout, then read them in the order of 2D Toeplitz layout layout through LTM's index conversion on-the-fly. Essentially, LTM generates the external memory addresses corresponding to each sliding window region in the flattened 3D tensor, and sends them to the memory controller. The data fetched by the memory controller into on-chip buffer are tiles of the converted 2D Toeplitz matrix. To realize this, we encode the feature map, filter and stride sizes in the control logic of the LTM (Fig. 4), such that it adapts to various CONV layer configurations. In the finite state machine shown in Fig. 4, state 2 loops inside each row (length F_1) of a sliding window, state 3 iterates all F_2 rows in a sliding window and state 1 steps over all overlapped sliding windows in the 3D Tensor. BA and DA stands for on-chip BRAM Address and external DDR Address, respectively. The process of state transition for fetching next matrix tiles, along with the communication time of fetching the next tiles, are overlapped with the computation using the current matrix tiles.

3.2.2 GEMM Core: Flexible Parallelism & Dataflows

3.2.2.1 Non-stationary v.s. Stationary CU Dataflows

Several dataflow schemes for systolic arrays have been proposed in the literature to perform GEMM [30]. Different dataflows

exploit different parallelism and therefore incur different trade-offs in the systolic array, based on input matrices' shapes. Our accelerator designs support the deployment of three dataflows summarized below and demonstrated in Fig. 5:

Non-stationary (NS) Dataflow: The input matrices $A(a \times b)$ and $B(b \times c)$ are partitioned along the common dimension b into tiles of size $b \times P_{sys}$. Each Processing Element (PE) computes a vector-product that contributes to one output element in C (in convolution, this is equivalent to computing parts of the convolution that will contribute to one output pixel). In each clock cycle each PE performs one multiply-accumulation (MAC) and shifts down the input and weight along two directions. Once all the MAC computations for an output element is finished, the final result is written to an output buffer and the PE immediately proceeds to work on a new pixel. Overall, the systolic array sized at $P_{sys} \times P_{sys}$ computes a $P_{sys} \times P_{sys}$ partition of the output matrix C in each tiling pass, and the tiling passes are repeated by feeding in all the tiles in the input matrices until all the block-partitions of the output matrix are finished.

Stationary Dataflow: We categorize stationary dataflows into Weight-Stationary (**WS**) and Input-Stationary (**IS**). In **WS**, in each pass, the PEs pre-load a (stationary) block of the weight/kernel matrix sized at $P_{sys} \times P_{sys}$ into their local registers. Then the input matrix, B, is fed as tiles of size $P_{sys} \times c$ into the systolic array horizontally in a pipelined fashion. In each clock cycle each PE performs a MAC operation, shifting the input to the next neighbor along horizontal direction and shifting the partial result down to accumulate the partial sums. Each of such $\frac{b}{P_{sys}}$ passes produces an intermediate accumulation result of a $P_{sys} \times c$-sized partition of the output matrix, accumulated at the bottom pf the systolic array using an accumulators connected with FIFOs of depth $P_{sys} + c$, after which one round ($\frac{b}{P_{sys}}$ passes) is complete, writing back final results of one $P_{sys} \times c$ partition of the output matrix. Such rounds are repeated until the weight matrix is completely covered. Note that although the WS-dataflow requires an array of accumulators at the bottom, it does not consume more accumulators than the NS-dataflow because the first row of PEs of WS-dataflow only perform multiplications. **IS** dataflow is the mirror of WS dataflow, where the input activation/gradient matrix is partitioned into (stationary) blocks to pre-load into the PEs, and the weight matrix is partitioned into tiles to be shifted through he systolic array.

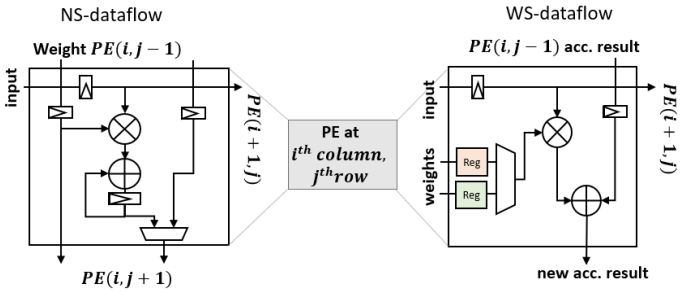


Fig. 5: NS/WS dataflow: PE

3.2.2.2 Conflict-Free Data Layout

While training (in Model Update phase), the weight matrices that need to be read in the BW passes are transpose of those in FW passes. If we store the matrix entirely in either row- or column- major order in P_{sys} BRAM blocks, we cannot ensure concurrent and consecutive accesses to the weight matrix by all PEs in both FW and BW passes. This is because multiple PEs requesting data from the same BRAM block in one of FW or BW processes will result in bank conflicts, costing delay proportional to $\frac{\sum \text{all layer sizes}}{P_{sys}}$ cycles to read these rows into consecutive streams before feeding into the systolic array. This issue happens in all the NS, IS and pre-loading process of WS dataflows, assuming a fixed dataflow for all taining phases.

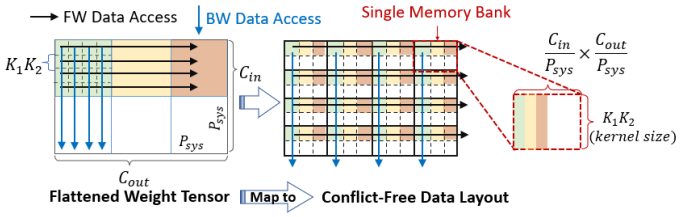


Fig. 6: Weights Data Layout

To address this issue, we design a new data layout to store on-chip parameters and intermediate results of neural networks in a memory-blocked fashion as illustrated in Fig. 6. We partition the input (weight) matrix into $\frac{C_{in}}{P_{sys1}} \times \frac{C_{out}}{P_{sys2}}$ blocks each sized at $P_{sys1} \times P_{sys2}$ along both dimensions (row and column), and each element $e_{i,j}$ with size $F_1 \times F_2$ in the block is stored in an individual memory block (e.g. BRAM, LUT) indexed with $Block_{i,j}$, such that we allocate $P_{sys1} \times P_{sys2}$ memory banks each storing $C_{in} \times C_{out}$. Fig. 6 shows such an example with $P_{sys1} = P_{sys2} = 4, C_{in} = 8, C_{out} = 12$. In each layer of forward propagation, in each pass, if data from the weight matrix is fed into systolic array in row-major order, P_{sys} data among different columns are accessed in parallel, and since these data are stored in different memory blocks, conflict-free parallel accesses can be achieved. Same is true for each pass of each layer in back propagation (data read in column major order), P_{sys} data can be accessed in one clock cycle along different rows as they are stored in separate memory blocks, and data can still be streamed in continuously without BRAM conflicts. Furthermore, to minimize bank consumption for very small weight matrices, we can store corresponding blocks of different weight matrices belonging to different layers into one memory block, such that a total of $P_{sys} \times P_{sys}$ memory blocks store all the weight matrices in the MLP instead of just one (Fig. 7).

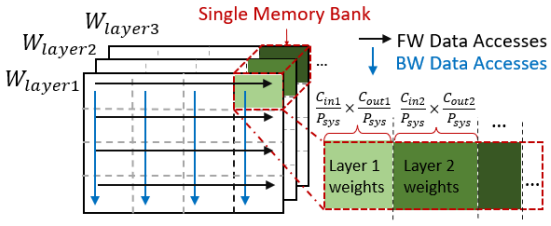


Fig. 7: Weight Matrix Storage

As shown in Fig. 3, The WU process also needs to take the transpose of a^l matrices, which are the intermediate results in each layer in the FW process of training, therefore both the weight buffer and a^l layers buffer are designed to support the blocked memory data layout. An additional benefit of this data layout is that matrix multiplications of consecutive layers in FW and BW passes can be performed without any stalls between them. This is enabled by stacking the data access pipelines of the two layers so

that as soon as the data of one layer is consumed, the data of the next layer starts feeding into the systolic array.

3.3 Load Balancing Module

In PPO, typically, the policy and value networks have separate parameters and are trained independently using two separate CUs. This might lead to load-imbalance between CUs due to potentially different number of epochs (and mini-batch sizes) required for training the two networks. This difference stems from the following: (1) Early stopping technique applied in PPO policy training when the new policy has large deviation from the old (Equation 1), and (2) different convergence rate between the networks. This may lead to the undesired situations where, when the training for one network is complete or stopped, the CU is completely idle waiting for the training of the other network to complete.

To address this issue, we develop a **Load Balancing Module** which off-loads the training of one network to both the CUs under such situations (Fig. 8). Both the CUs have access to the buffers storing the parameters of both the networks via multiplexers. When both networks are training, each CU accesses their own corresponding weight buffers. When one network finishes training, the CU is switched to access the weights corresponding to the other network, and the two CUs access the same weight buffer. The mini-batch inputs from CPU are evenly divided into two sub-mini-batches to feed into the two CUs concurrently. During FW and BW

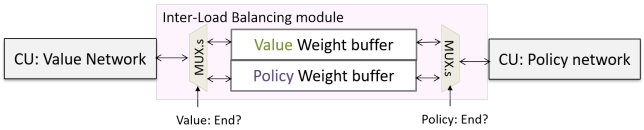
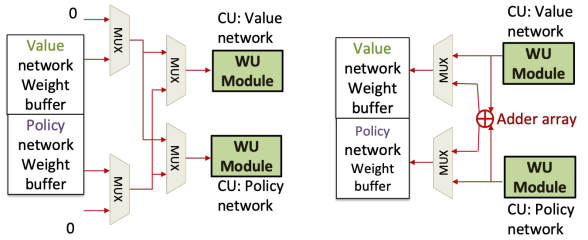


Fig. 8: Load Balance Module

processes, the PEs in systolic arrays of the two CUs simply read from the selected buffers. However, for WU, the mechanism is more complicated and requires the following additional logic: (i) At the output end, an adder array of length P_{sys} is needed to sum up the weight updates produced by each CU before writing into the weight buffer as both CUs will be updating the same memory addresses of the weight buffer in any given clock cycle (Fig. 9b), and (ii) at the input end, we allow only one of the CUs to read the weights while the other reads 0 to avoid adding the original weight twice (Fig. 9a).



(a) WU read operations (b) WU write operations

Fig. 9: Load Balance strategy for WU

4 Design Space Exploration

The dimensions of the systolic array as well as the choice of dataflow have a significant impact on the performance of GEMM operations of the FW/BW/WU processes. As the dimensions of GEMM operations vary widely across the layers, design space exploration to determine the optimal design choices — systolic array dimensions, and dataflow is critical to obtain high performance. We propose a Design Space Exploration (DSE) scheme to obtain optimal values of the design choices given the FPGA device specification, the DNN model and the hyper-parameters of the RL algorithm.

To achieve this, we first construct accurate performance models for each dataflow (Ψ) as follows: We define the GEMM dimension for each layer as a, b, c where $a \times b, b \times c$ are the sizes of weight and input matrices. Assuming a $P_{sys} \times P_{sys}$ systolic array, for each dataflow described in Section *3.2.2.1*, we model the cycles required for a single GEMM operation:

$$\mathbb{T}_{\underset{(a,b,c)}{\text{mm}}}(\Psi) = \begin{cases} \Psi = NS : \lceil \frac{a}{P_{sys}} \rceil \times \lceil \frac{c}{P_{sys}} \rceil \times b + 2P_{sys} - 1, \\ \Psi = WS : \lceil \frac{b}{P_{sys}} \rceil \times \lceil \frac{a}{P_{sys}} \rceil \times c + 2P_{sys} - 1, \\ \Psi = IS : \lceil \frac{b}{P_{sys}} \rceil \times \lceil \frac{c}{P_{sys}} \rceil \times a + 2P_{sys} - 1 \end{cases} \tag{5}$$

We use hyper-parameters N, M consistent with introduced in Section 2.1. We use notations for a DNN CONV/FC layer in accordance with Section 3.2, assuming a total of n layers in the DNN model. In Eq. 6, the expressions for $\mathbb{T}_{mm}$ replaces the general matrix dimensions (a, b, c) with the input Toeplitz matirx and weight matrix dimensions summarized in Sec. 3.2 (expressing CONV as GEMM): $(O_1 \times O_2, C_{in} \times F_1 \times F_2, C_{out})$. The ordering of these dimensions are different in FW (Conv), BW (Transposed Conv) and WU.

$$\begin{aligned} \mathbb{T}_{FW}(\mathcal{D}, \Psi) &= \sum_{i=1}^{i=n-1} \mathbb{T}_{mm}(\Psi)(\mathcal{D} * O_1 O_2^i, F_1 F_2^i * C_{in}^i, C_{out}^i) \\ \mathbb{T}_{BW}(M, \Psi) &= \sum_{i=n}^{i=2} \mathbb{T}_{mm}(\Psi)(M * H_1 H_2^i, F_1 F_2^i * C_{out}^i, C_{in}^i) \\ \mathbb{T}_{WU}(M, \Psi) &= \sum_{i=1}^{i=n-1} \mathbb{T}_{mm}(\Psi)(F_1 F_2^i * C_{in}^i, M * O_1 O_2^i, C_{out}^i) \end{aligned} \tag{6}$$

where $\mathcal{D}$ can be N or M, depending on Inference or Training phase. We further model the total execution time on FPGA for inferences in Sample Generation phase and the training tasks in Model Update phase:

$$\mathcal{T}_{inf}(\Psi) = \mathbb{T}_{FW}(N, \Psi) \times T + \mathcal{T}_{comm1} \text{ and} \tag{7}$$

$$\mathcal{T}_{train}(\Psi) = \mathcal{T}_{minibatch}(\Psi) \times K \times \frac{N \times T}{M} + \mathcal{T}_{comm2}, \text{ where}$$
$$\mathcal{T}_{minibatch}(\Psi) = \mathbb{T}_{FW}(M, \Psi) + \mathbb{T}_{BW}(M, \Psi) + \mathbb{T}_{WU}(M, \Psi) \tag{8}$$

T_{comm} can be modeled with $\frac{\text{data volume}}{BandWidth} + ovhd$, where $BandWidth$ is the theoretical streaming bandwidth of PCIe link or DDR access, and $ovhd$ is a small constant initialization latency for each bulk of data transfer. In practice, we use double buffering to hide as much DDR communication time as possible. T_{mm} required for each layer can be scaled with *Required BandWidth* divided by *Actual BandWidth* to account for the DDR communication time left un-hidden when the required bandwidth is greater than the available bandwidth. Note that while it is possible to have a layerwise-switching dataflow, this will lead to additional overheads in matrix transformations between layers. In this work, we limit to finding a single dataflow for all layers of FW (or

BW, WU) to avoid such overheads and leave the exploration of layerwise-switching dataflow for future.

Overall, our DSE can be formulated as the following optimization problem:

$$\begin{aligned} &\underset{P_{sys},\Psi}{\text{minimize}}(\mathcal{T}_{inf}(\Psi) + \mathcal{T}_{train}(\Psi)) \\ &\text{subject to: } \mathbb{C}_{BRAM}, \mathbb{C}_{DSP} \end{aligned} \tag{9}$$

where $\mathbb{C}_{DSP} : P_{sys}^2 \times DSP_{PE} + DSP_{\text{WU, Mult. Modules}} < DSP_{FPGA}$ denotes DSP resource constraints on the systolic array, and $\mathbb{C}_{BRAM}$ denotes the available BRAM blocks to enable conflict-free data layout.

5 Experiments & Evaluations

5.1 Benchmark Experiments and Platforms Overview

We use two MuJoCo (Robotics) and an Atari (Gaming) benchmarks from OpenAI Gym [13], [31] - which are widely used to evaluate RL algorithms [14], [32], [33]. The Mujoco benchmarks' environmental states are represented in 1D vectors of positions/angles of robot joints, and the Atari benchmarks environmental states are compressed images of 4 successive game screenshots stored in 3D tensors. In Mujoco - Hopper and Humanoid, we apply PPO to learn the policy to control a robot to hop or run. We use the same structure for value and policy networks except for the output layer with $relu()$ activation. We use a neural network of size 11 (input) - 64 - 64 - 3 (output) for Hopper and a neural network of size 376 (input) - 128 - 64 - 17 (output) for Humanoid. For the Atari - Pong, we use a CNN with 3 CONV layers and 2 FC layers, which is consistent with those widely used in state-of-the-art Deep RL works [20], [34], [35].

We compare our CPU-FPGA design against two platforms: CPU-only and CPU-GPU. Our FPGA implementations follow the same semantics (synchronous trajectories and training) and use the same data type (float 32) as those on CPU and GPU baselines. We focus on improving the PPO iteration speed without changing the training quality (number of iterations required to achieve certain reward). We use Intel Xeon 5120 as the Host CPU for all three implementations. We use TITAN Xp GPU as the GPU and Xilinx Alveo U200 as the FPGA. For comparisons on Mujoco, we use two baseline implementations: (1) the OpenAI Spinningup implementation of PPO [17] which uses MPI [36] for parallel agents on CPU along with training using mini-batch SGD/SGA [26]; (2) the OpenAI Baselines implementation of PPO2 [18], which uses SubprocEnv library [37] for vectorized environmental steps and optimized mini-batch SGD/SGA for training. For comparisons on Atari, we use the OpenAI Baseline implementation only. All baseline implementations are based on TensorFlow [38].

For the FPGA performance evaluation of PPO on Mujoco, we implement the design consisting of two CUs for the two networks (value and policy) and the load balancing module on the FPGA. For PPO on Atari, consistent with the Baseline PPO2 implementations, the parameters are shared between the two networks along the CONV layers and the first fully-connected layer. A unified objective function for both network is minimized - $-c_1\mathcal{L}^{log} + \mathcal{L}^{MSE} +$ entropy, with value coefficient c_1 and an entropy term [21], [39].

5.2 DSE & FPGA Resource Utilizations

We first perform DSE as discussed in Section 4. As the typical hyper-parameters and DNN layer/channel sizes considered are powers of 2, we constrain P_{sys} to be powers of 2 to minimize the required zero-paddings on the systolic array(s). For all three dataflows, performance increases monotonically with P_{sys}, therefore the optimal systolic array sizes returned are the same. On Alveo board, we set $P_{sys} = 16$ for Mujoco benchmarks experiments and $P_{sys} = 32$ for Atari benchmark.

We implement the design consisting of two CUs, buffers (with the data layout optimization) and the load balancing module on the FPGA for Mujoco benchmarks. We use one large CU for CNN inference and training on Atari baselines without load balancing module (as the value and policy networks are shared). Using Vivado 2018.3 for synthesis, our implementations for Mujoco and Atari benchmarks sustain clock frequency of 285 MHz and 300MHz, respectively. Table 1 summarizes the resource usage breakdown on the FPGA. Note that the three dataflows have the same resource usage.

TABLE 1: FPGA Resource Utilizations: exp.1: Hopper; 2: Humanoid; 3: Pong

	exp.	M=64	M=128	M=256	M=512
BRAM	1	696(32%)	752(35%)	810(38%)	926(43%)
	2	638(30%)	926(43%)	1466(68%)	2160(100%)
	3	3745 (72%)			
DSP	1&2	3744 (55%)	*DSP* exp.3		4227 (62%)
Logic	1&2	501k (42%)	*Logic* exp.3		1061k (89%)
FlipFlop	1&2	708k (30%)	*FlipFlop* exp.3		1392k (59%)

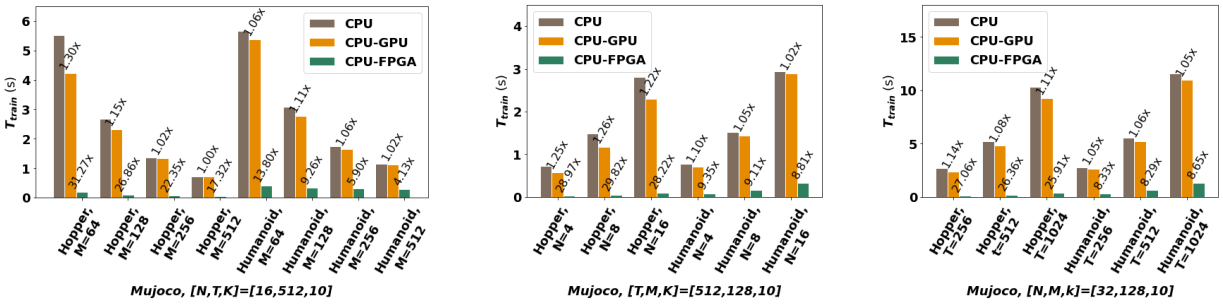
In Table 1, exp. 1,2 stand for experiments on Hopper and Humanoid with MLP policies, and exp.3 stands for experiment on Pong with CNN policy. The BRAM utilization for both Mujoco benchmarks increases with mini-batch size M and MLP size as we use on-chip memory entirely for the network propagations. The utilization of the remaining resources do not vary with different hyper-parameters for exp. 1& 2. For exp.3, we keep the CNN parameters on-chip and use fixed sized $\boldsymbol{z}^l, \boldsymbol{a}^l$ and $\boldsymbol{\delta}^l$ buffers, interfacing with DDR4 for streaming accesses to intermediate activation and gradient feature maps, which are implemented using double buffering technique to hide DRAM communication overheads.

5.3 Performance

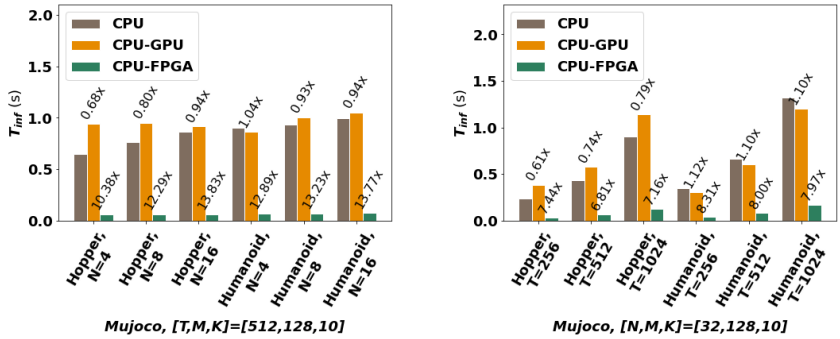
5.3.1 Inference & Training Execution Time, Throughput

In accordance with the existing works, we use the metric: Number of inferences processed per second (IPS) to measure the throughput performance. IPS is the ratio of the total number of samples collected in the Sample Generation phase ($N \times T$) to the total time taken in executing one complete PPO iteration.

Table 2 shows the IPS for the CPU-FPGA, CPU-only and CPU-GPU implementations, using various benchmarks with different hyper-parameter combinations. To further visualize the trends in speedups of inference and training tasks without considering the environmental simulation times, we plotted the total inference & training execution time for each corresponding set of hyper-parameters in Fig. 10 (compared only with OpenAI Baselines [39] implementation which has better training performance) and 11. In each figure, the first row of subfigures show the total training (Model Update phase) execution time on their y-axes; The second row of subfigures show the total time of inferences in the entire Sample Generation phase. The labels $(a\times, b\times)$ above the CPU-GPU and CPU-FPGA bars indicate their amount of speedups of inference/training execution time over CPU-only implementations.

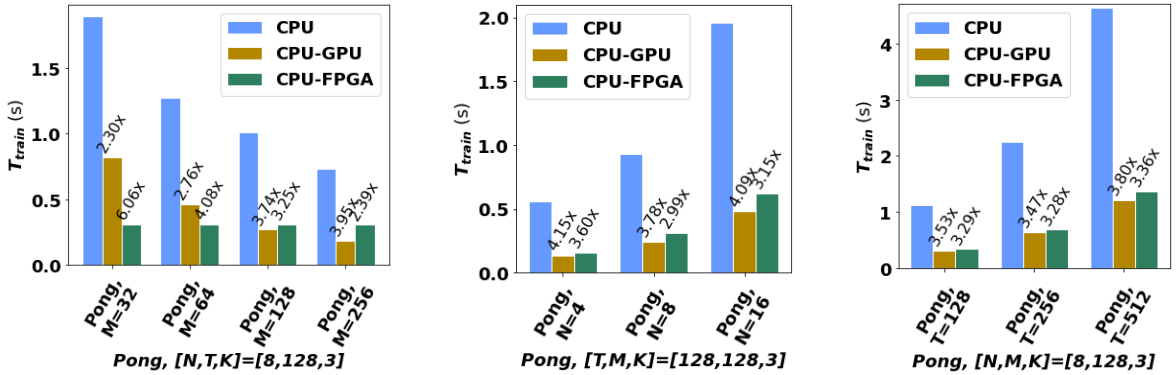


(a) Mujoco with MLP: scaling M (b) Mujoco with MLP: scaling N (c) Mujoco with MLP: scaling T

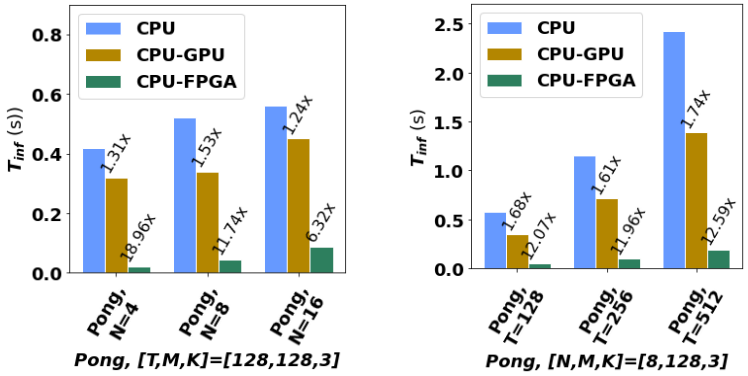


(d) Mujoco with MLP: scaling N (e) Mujoco with MLP: scaling T

Fig. 10: Mujoco benchmarks: Execution time comparison of our design with baselines varying N, M, T



(a) Pong with CNN: scaling M (b) Pong with CNN: scaling N (c) Pong with CNN: scaling T



(d) Pong with CNN: scaling N (e) Pong with CNN: scaling T

Fig. 11: Atari benchmark: Execution time comparison of our design with baselines varying N, M, T

TABLE 2: IPS Comparison of CPU-FPGA Design with CPU and CPU-GPU Baselines

Benchmarks & Baselines		Varying Training Batch Sizes M $[N, T, K] = [16, 512, 10]$				Varying Inference Batch Sizes N $[T, M, K] = [512, 512, 10]$			Varying Trajectory Lengths T $[N, M, K] = [32, 512, 10]$		
		$M = 64$	$M = 128$	$M = 256$	$M = 512$	$N = 4$	$N = 8$	$N = 16$	$T = 256$	$T = 512$	$T = 1024$
Hopper-SpinUp	C	509	1.15k	2.31k	2.45k	2.71k	2.78k	2.8k	2.64k	2.7k	2.73k
	C-G	539	1.27k	2.71k	2.82k	2.86k	2.9k	3.3k	2.83k	2.78k	2.96k
	C-F	13.8k (26x)	17.1k	17.7k	18.7k (6.6x)	7k (2.4x)	13.1k	21.4k (6.5x)	26.4k	28.6k	29.8k
Human-SpinUp	C	443	720	1.17k	2k	1.62k	1.82k	1.9k	2.1k	2.15k	2.13k
	C-G	478	780	1.28k	2.18k	1.73k	2.1k	2k	2.2k	2.26k	2.2k
	C-F	8.3k (17.4x)	9.2k	9.7k	10.1k (4.6x)	3.2k (1.8x)	5.6k	7.6k (3.8x)	8.24k	9.2k	9.48k
		$[N, T, K] = [16, 512, 10]$				$[T, M, K] = [512, 128, 10]$			$[N, M, K] = [32, 128, 10]$		
		$M = 64$	$M = 128$	$M = 256$	$M = 512$	$N = 4$	$N = 8$	$N = 16$	$T = 256$	$T = 512$	$T = 1024$
Hopper-Baselines	C	929	1.37k	1.76k	2k	646	964	1.39k	2k	2.11k	2.12k
	C-G	1.07k	1.43k	1.73k	2k	617	993	2.1k	2.2k	2.24k	2.26
	C-F	3.63k (3.4x)	2.72k	2.76k	2.78k (1.4x)	1.07k (1.7x)	1.91k	4.12k (1.9x)	6.2k	6.85k	6.8k
Human-Baselines	C	914	1.28k	1.59k	1.85k	437	724	1.12k	1.67k	1.64k	1.64k
	C-G	936	1.32k	1.63k	1.8k	447	725	1.16k	1.71k	1.69k	1.7k
	C-F	2.44k (2.6x)	2.52k	2.55k	2.57k (1.4x)	643 (1.44x)	1.18k	2.13k (1.8x)	3.73k	3.58k	3.79k
		$[N, T, K] = [4, 128, 3]$				$[T, M, K] = [128, 128, 3]$			$[N, M, K] = [8, 128, 3]$		
		$M = 32$	$M = 64$	$M = 128$	$M = 256$	$N = 4$	$N = 8$	$N = 16$	$T = 128$	$T = 256$	$T = 512$
Pong-Basline	C	326	406	453	512	365	541	687	507	501	488
	C-G	550	682	781	822	459	808	1.17k	797	768	776
	C-F	870 (1.6x)	870	871	870 (1.1x)	561 (1.2x)	904	1.49k (1.3x)	1058	908	998

⋆ SpinUp: Spinningup implementation; Human: Mujoco Humanoid Benchmark. ⋆ C: CPU; C-G: CPU-GPU; C-F: CPU-FPGA

For Mujoco experiments, we consistently observe higher IPS on Hopper than Humanoid, and our FPGA-based design achieves higher improvements against the baselines on Hopper than Humanoid (Table 2, all rows labeled Hopper- compared to Human- for the same implementation method and platforms). This is because: (a) The Humanoid environment costs longer sequential simulation time which leads to smaller room for speed-up after parallelization; (b) Larger network is required for Humanoid than Hopper. For comparisons with Spinningup implementations, we observe higher increase in IPS than comparing with Baselines implementations (Table 2, rows labeled -Baselines compared to -SpinUp). This is because: (1) PPO2 (Baselines) share parameters between policy and value networks) that results in ($\sim 2\times$) shorter training time than that in PPO (Spinningup) implementation which trains value and policy network separately; (2) the SubprocEnv wrapper used for $env.step()$ function in PPO2 (Baselines) implementation enables deploying multiple instances of environment in the same CPU core, while the PPO (Spinningup) implementation allow one simulation environment per CPU core to run in parallel asynchronously. As a result, each environmental (ENV) step of PPO2 (Baselines) is $\sim 8\times$ longer than ENV step in PPO (Spinningup), thus the room for FPGA speedup is smaller when comparing with PPO2 (Baselines) due to longer sequential ENV time. If the ENV time with SubprocEnv wrapper gets further optimized, we can get higher increase in IPS on FPGA compared with Baselines implementation, as evident from the amount of speedup on just the inference and training tasks as shown in Fig. 10.

In Fig. 10a, 11a and Table 2, column 1 for "Varying Training Batch Sizes M", we fix [N, T, K] and vary the training batch size M. The inference comparisons are not shown as changes in M only affect training. We observe that on both CPU and CPU-GPU, as the mini-batch size drops, the IPS drops too (Table 2, rows for "C" and "C-G" under column 1). This is due to an increase in the frequency of synchronizations & communications between threads/processors for weight gradient aggregation operations in WU (with frequency $\propto \frac{N \times T}{M}$), as discussed in Section 2.4. Our fine-grained pipelined design, where synchronization overheads are non-existent, does not suffer from such slowdowns (Table 2, rows for "C-F" under column 1). This is also evident from our performance model (Equation 8), where M contributes to both numerator and denominator equally.

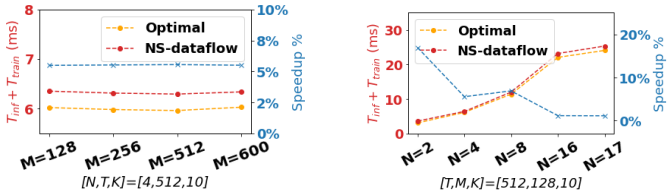
In Fig. 10b, 10d, 11b, 11d and Table 2, column 2 for "Varying Inference Batch Sizes N", we fix [M, T, K] and vary the inference batch size N. FPGA shows superior performance on both inference and training tasks on Mujoco benchmarks. On Atari benchamarks, although FPGA incurs slightly slower CNN training, this slow down is offset by the larger amount of speedup in inferences and we still observe higher IPS (Table 2, rows labeled Pong- under column 2). On Mujoco benchmarks, FPGA inference latencies do not show a significant increasing trend (Fig. 10d because the PCIe transfer request latency dominates compared to computation (Equation 7). In training, such latencies are incurred repeatedly as N increases, so both computation and communication times scale up. As shown in column 2 of Table 2, on all platforms, higher IPS is observed by increasing N due to increased parallelism. The highest performance improvement is observed at N=16 implying that our design scales equivalent or better than the baselines with increasing N.

In Fig. 10c, 10e, 11c, 11e and Table 2, column 3 for "Varying Trajectory Lengths T", we fix [N, M, K] and vary each agent's trajectory length T. Increasing T leads to increases in both number of inferences processed per iteration and total iteration time (both inference and training time are prolonged due to increasing T) , therefore we observe that throughput remains roughly the same on each platform with different values of T (Table 2 col. 10-12). Overall, we observe $1.8\times \sim 10.9\times$ improvements on CPU-FPGA platform over the baselines (not shown in table due to space limitation).

5.3.2 Flexible Dataflow Optimization

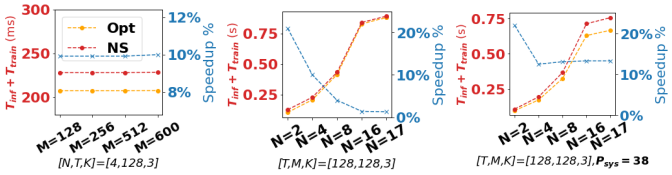
To evaluate the effectiveness of our flexible dataflow dse scheme, we consider a design [10] which performs DSE by fixing $\Psi = NS$, i.e. only considers NS dataflow as baseline. The total execution time of inferences and training tasks in one iteration are displayed for a Mujoco benchmark (Fig. 12) and an Atari benchamrk (Fig. 13) in red and orange plots. We also plot the speedup in percentage (blue plots). We observe a constant increase of performance in both benchmarks with varying M (Fig. 12a and 13a). When $N < P_{sys}$ we observe larger improvements on

smaller N (Fig. 12b and 13b). This is due to the fact that in FW, the NS dataflow parallelizes between the individual samples in an inference batch sized at N, as confirmed in the performance model in Section 4 (N is in a term in Equation 5, and b term is usually larger than P_{sys} as shown in Equation 6). The smaller N is, the less utilized the systolic array remains, giving more room of improvement for stationary dataflow which puts N in the temporal term and parallelizes along the two larger dimensions.



(a) Hopper, Scaling M (b) Hopper, Scaling N

Fig. 12: Dataflow optimization: Mujoco benchmark



(a) Pong, Scaling M (b) Pong, Scaling N (c) Pong, Scaling N

Fig. 13: Dataflow optimization: Atari benchmark

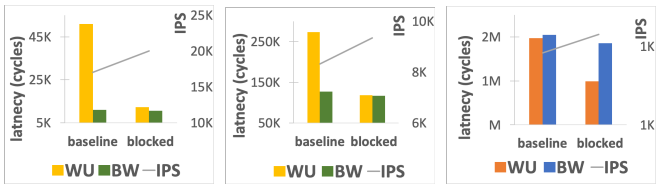
Also, as most of DNN layer/channel sizes and P_{sys} we consider are powers of 2, relatively small improvement is observed for large N. Higher improvement can be expected when the systolic array size or DNN layer sizes are not powers of 2, such that some GEMM dimensions do not evenly divide P_{sys}. As shown using a theoretical analysis with 38×38 systolic array assuming same clock rate (Fig. 13c), The performance difference from different dataflows is more obvious in Fig. 13c compared to Fig. 13b since parallelizing along different GEMM dimensions with a factor of 38 leads to higher variance in PE utilization. For any P_{sys}, we show that the optimal dataflow by our DSE always yields better or equivalent performance than fixed NS dataflow.

5.3.3 Data Layout Optimization

To evaluate the effectiveness of data layout optimization, we consider a baseline (no optimization) design where the weight and activation matrices are stored in P_{sys} BRAM blocks such that data can be streamed into systolic array in FW passes while extra cycles are required in the BW and WU passes. We measure the BW/WU latencies of one single mini-batch in clock cycles and the overall IPS for this case on Mujoco benchmarks for the set of hyper-parameters $[N, M, T, K]$ = [32, 512, 512, 10], and compare with our optimized design using blocked data layout, shown in Fig. 14. Our result shows 23.5% increase in overall throughput (IPS) for Hopper and 21.2% increase for Humanoid using memory blocked data layout (with Spinningup ENV time). On Atari benmark, Pong, our result on $[N, M, T, K]$ = [8, 128, 128, 3] shows 11.2% in IPS.

5.3.4 Inter-Load Balance Optimization

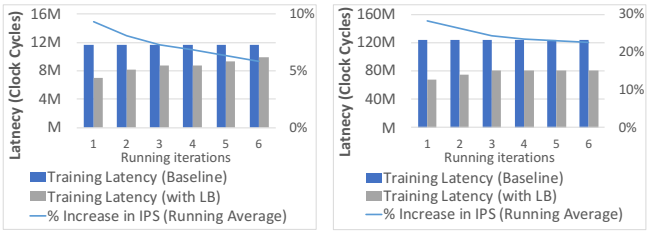
We compare implementations with and without load balance optimization using the early stopping trick used in the OpenAI Spinningup code [17] wherein load-imbalance occurs at the early iterations of PPO as KL divergence is prone to be larger (resulting



(a) Hopper (b) Humanoid (c) Pong

Fig. 14: Data layout optimization

in earlier stopping of policy training). We run the algorithm for the first 6 iterations and measure the training latencies in every iteration as well as the running average throughput, again with hyper-parameters $[N, M, T, K]$ = [32, 512, 512, 10]. The result in Fig. 15 shows 9.3% to 5.8% and 28.3% to 22.5% increase in overall running average throughput (IPS) in the first 6 iterations of PPO for Hopper and Humanoid, respectively.



(a) Hopper (b) Humanoid

Fig. 15: Load balance optimization

6 Related Works

Most of the existing works on accelerating RL on FPGAs focus on Q-Learning algorithms such as Deep Q Learning [40], [41] and Table based Q-Learning [42], [43]. Unlike policy optimization methods which directly learns the policy, Q-Learning algorithms learn the quality of state-action pairs. Due to the fundamentally different approach in learning, these techniques cannot be directly ported to policy optimization methods.

A few recent works have focused on accelerating policy optimization methods. A preliminary version of this work [10] focuses on accelerating PPO with MLP-based policies. A FPGA implementation of the A3C algorithm is presented in [23] which uses CNN policy targeting the application of Atari 2600 games. Unlike PPO, A3C is an asynchronous algorithm requiring relaxed dependency constraints between the inference and training, which imposes different hardware requirement than the synchronous semantics of PPO (e.g. asynchronous data transfers lead to communication overhead if trivially adopted to batched execution in PPO). In [33], an accelerator design is developed for Trust Region Policy Optimization (TRPO). The TRPO accelerator design does not exploit the advantage of parallel agents or batched training. In [7], a CPU-FPGA architecture is proposed for Deep Deterministic Policy Gradient (DDPG) that combines Deep Q-Learning with policy optimization by deploying four DNN models. Due to the intrinsically different inter-DNN-model dependencies in DDPG, their methodology cannot be trivially adopted to accelerate PPO.

Additionally, we summarize existing work on CNN acceleration using heterogeneous platforms. [44] proposed a CNN inference framework on embedded heterogeneous system-on-chip (SoC) architectures. CNN training accelerators on FPGA are proposed [45], [46]. These inference and training accelerators target general CNN architecture, however, none of them address

the unique model interactions in the context of mapping CNN-based RL on heterogeneous patforms.

To the best of our knowledge, our work is the first to propose a FPGA-based framework targeting PPO acceleration with both CNN and MLP policies.

7 Limitations and Future Directions

In this work, we developed a framework for accelerating PPO on CPU-FPGA platforms, and obtained consistent speedups across different benchmarks and DNN models. Nevertheless, we point out some limitations of the current work and future opportunities.

Limitations: For larger CNNs, due to the limited DDR-bandwidth of FPGA platforms, communication overheads on FPGAs may not be as efficiently hidden as in GPUs with higher global memory bandwidth. Additionally, when kernel computation is small and number of such kernel launch is frequent (small MLP and batch sizes), limited PCIe bandwidth may become a bottleneck due to frequent CPU-FPGA (same for CPU-GPU) communication. To address these limitations, in the future, we will integrate low-memory optimization for convolution kernels [47] and utilize new HBM technologies on FPGA [48].

Other Future Directions: One opportunity is to map our framework on a TPU which has fixed systolic dataflow for GEMM. A challenge in this context is realizing peripheral modules such as data allocation, Load-Balancing Module, and computation of $\hat{A}_t$ and $\hat{R}_t$ on CPU-TPU platform. While GEMM operations in DRL policies can be efficiently performed using hardware systolic arrays on FPGA or ASIC, recent advances in GPU architecture are also promising for accelerating Deep Learning applications. For example, GPU Tensor Cores are shown to bring $\sim 10\times$ performance gain compared to traditional CUDA Cores in deep learning training workloads. Another example is the recently-proposed GPU Software Systolic Array (SSA) Model [49] that improves GEMM kernels performance on GPUs by utilizing low-latency register exchange operations. A promising future direction is to port our framework on modern GPUs with each GEMM Core replaced with Tensor Cores (or SSA), and to perform a CPU-GPU task mapping (including layout transformation, objective computation, inter load-balancing) with careful manipulation of CPU/GPU thread-level synchronizations between the DNN models. This could lead to significant speedup in PPO Model Update phase compared to existing GPU baseline implementations. Another research direction that we will pursue is to utilize cache-coherent interconnect on a tightly-coupled CPU-FPGA compute node. This will reduce CPU-FPGA communication overhead and enable efficient parallelization over processor-FPGA supporting a wider range of DRL algorithms including asynchronous or off-policy algorithms. We will also extend our framework into an overlay with complete compiler tool-chain exposed to higher-level abstractions of DRL algorithms.

8 Conclusion

In this work, we accelerated PPO targeting CPU-FPGA heterogeneous platform. We developed novel optimizations and achieved significant speedup over CPU and CPU-GPU implementations. The optimizations developed have a broader applicability, and can be easily extended to more policy models such as LSTM which share similar GEMM kernels as MLP and CNN. While this paper focus on current state-of-the-art implementations of PPO only, we discovered future opportunities in deploying and optimizing DRL worloads in general on both CPU-FPGA and CPU-GPU heterogeneous platforms.

9 Acknowledgment

This work is supported by the U.S. National Science Foundation under grant CNS-2009057 and Xilinx.

References

[1] R. Sutton, M. Bowling, D. Schuurmans, and V. Bulitko, "Reinforcement learning and artificial intelligence," 2006.

[2] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[3] P. Remagnino, A. Shihab, and G. A. Jones, "Distributed intelligence for multi-camera visual surveillance," *Pattern recognition*, vol. 37, no. 4, pp. 675–689, 2004.

[4] P. Hamet and J. Tremblay, "Artificial intelligence in medicine," *Metabolism*, vol. 69, pp. S36–S40, 2017.

[5] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[7] C. Guo, W. Luk, S. Q. S. Loh, A. Warren, and J. Levine, "Customisable control policy learning for robotics," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 91–98.

[8] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A customizable matrix multiplication framework for the intel harpv2 xeon+ fpga platform: A deep learning case study," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 107–116.

[9] Ccix, 2017. [Online]. Available: https://www.ccixconsortium.com/

[10] Y. Meng, S. Kuppannagari, and V. Prasanna, "Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 19–27.

[11] W. S. Lovejoy, "A survey of algorithmic methods for partially observed markov decision processes," *Annals of Operations Research*, vol. 28, no. 1, pp. 47–65, 1991.

[12] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[13] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[14] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning*, 2016, pp. 1329–1338.

[15] R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana, "Deep reinforcement learning for general video game ai," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2018, pp. 1–8.

[16] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 279–291.

[17] Openai spinningup implementation of ppo, 2018. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html,https://github.com/openai/spinningup/tree/master/spinup/algos/ppo

[18] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," https://github.com/openai/baselines, 2017.

[19] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004, pp. 133–137.

[20] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," *arXiv preprint arXiv:1802.01561*, 2018.

[21] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," *arXiv preprint arXiv:1712.09381*, 2017.

[22] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski *et al.*, "What matters in on-policy reinforcement learning? a large-scale empirical study," *arXiv preprint arXiv:2006.05990*, 2020.

[23] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "Fa3c: Fpga-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 499–513.

[24] Tensorflow, 2015. [Online]. Available: https://github.com/tensorflow/tensorflow

[25] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," 2018.

[26] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[27] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, 2015, vol. 25.

[28] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[29] Y. Bi, G. Chen, Q. Deng, and Y. Wang, *Embedded Systems Technology: 15th National Conference, ESTC 2017, Shenyang, China, November 17-19, 2017, Revised Selected Papers*. Springer, 2018, vol. 857.

[30] J. Cong and J. Wang, "Polysa: Polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[31] Mujoco & atari gym environments, 2016. [Online]. Available: https://gym.openai.com/envs/

[32] E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba, "Benchmarking model-based reinforcement learning," *arXiv preprint arXiv:1907.02057*, 2019.

[33] S. Shao and W. Luk, "Customised pearlmutter propagation: A hardware architecture for trust region policy optimisation," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–6.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[35] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[36] W. Gropp, W. D. Gropp, E. Lusk, A. D. F. E. E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[37] Subprocvecenv: Vectorized environments, 2019. [Online]. Available: https://github.com/openai/baselines/blob/master/baselines/common/vec_env/subproc_vec_env.py

[38] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[39] Openai baselines implementation of ppo2, 2019. [Online]. Available: https://github.com/openai/baselines/tree/master/baselines/ppo2

[40] J. Su, J. Liu, D. B. Thomas, and P. Y. Cheung, "Neural network based reinforcement learning acceleration on fpga platforms," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 68–73, 2017.

[41] P. R. Gankidi and J. Thangavelautham, "Fpga architecture for deep learning and its application to planetary robotics," in *2017 IEEE Aerospace Conference*. IEEE, 2017, pp. 1–9.

[42] L. M. Da Silva, M. F. Torquato, and M. A. Fernandes, "Parallel implementation of reinforcement learning q-learning technique for fpga," *IEEE Access*, vol. 7, pp. 2782–2798, 2018.

[43] R. Rajat, Y. Meng, S. Kuppannagari, A. Srivastava, V. Prasanna, and R. Kannan, "Qtaccel: A generic fpga based design for q-table based reinforcement learning accelerators," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 323–323.

[44] G. Zhong, A. Dubey, C. Tan, and T. Mitra, "Synergy: An hw/sw framework for high throughput cnns on embedded heterogeneous soc," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 2, pp. 1–23, 2019.

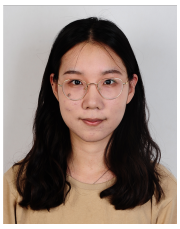
[45] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," in *2016 IEEE 27Th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2016, pp. 107–114.

[46] S. K. Venkataramanaiah, Y. Ma, S. Yin, E. Nurvithadhi, A. Dasu, Y. Cao, and J.-s. Seo, "Automatic compiler based fpga accelerator for cnn training," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 166–172.

[47] W. Zhang, M. Jiang, and G. Luo, "Evaluating low-memory gemms for convolutional neural network inference on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 28–32.

[48] Z. Wang, H. Huang, J. Zhang, and G. Alonso, "Shuhai: Benchmarking high bandwidth memory on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 111–119.

[49] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for gpu memory-bound kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–81.



Yuan Meng obtained her BS degree in electrical and computer engineering at Rensselaer Polytechnic Institute and is currently a Ph.D. student in Computer Engineering at the University of Southern California. She is recipient of Annenberg Fellowship at Ming Hsieh Department of Electrical and Computer Engineering. Her research interests include parallel computing, hardware acceleration, and machine learning.

Sanmukh R. Kuppannagari received the B.Tech. degree in Computer Science and Engineering from IIT-Guwahati in 2011 and the PhD degree in Computer Engineering from the University of Southern California (USC) in 2018. He is currently a PostDoctoral Scholar at the Department of Electrical and Computer Engineering, USC. His research interests include combinatorial optimization, parallel computing and hardware acceleration of AI algorithms.

Rajgopal Kannan obtained his B. Tech degree in Computer Science and Engineering from IIT-Bombay in 1991 and the Ph.D. in Computer Science from the University of Denver in 1996. He is currently a Research Adjunct Professor in Electrical Engineering at the University of Southern California. He was formerly a Professor in the Dept. of Computer Science at Louisiana State University (2000-2015). His academic research was funded by DARPA, NSF and DOE and he has published over 150 research papers with two patents awarded in the area of network optimization. His research interests are at the intersection of Graph Analytics, Machine Learning and Edge Computing - enabling application acceleration at the edge on low power devices.

Viktor K. Prasanna received the BS degree in electronics engineering from the Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University. He is Charles Lee Powell Chair in engineering in the Ming Hsieh Department of Electrical Engineering and professor of computer science with the University of Southern California. His research interests include high performance computing, parallel and distributed systems, reconfigurable computing. He serves as the director of the Center for Energy Informatics, USC. Currently, he is the editor-in-chief of the Journal of Parallel and Distributed Computing. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering cochair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He received the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University, and the W. Wallace McDowell Award from the IEEE Computer Society in 2015 for his contributions to reconfigurable computing.