Towards Hardware Accelerated Reinforcement Learning for Application-Specific Robotic Control

Shengjia Shao, Jason Tsai, Michal Mysior and Wayne Luk
Imperial College London
United Kingdom
{ss13412, jt2815, mjm215, w.luk}@imperial.ac.uk

Thomas Chau, Alexander Warren and Ben Jeppesen
Intel Corporation
United Kingdom
{thomas.chau, alexander.warren, ben.jeppesen}@intel.com

***Abstract*—Reinforcement Learning (RL) is an area of machine learning in which an agent interacts with the environment by making sequential decisions. The agent receives reward from the environment based on how good the decisions are and tries to find an optimal decision-making policy that maximises its long-term cumulative reward. This paper presents a novel approach which has shown promise in applying accelerated simulation of RL policy training to automating the control of a real robot arm for specific applications. The approach has two steps. First, design space exploration techniques are developed to enhance performance of an FPGA accelerator for RL policy training based on Trust Region Policy Optimisation (TRPO), which results in a 43% speed improvement over a previous FPGA implementation, while achieving 4.65 times speed up against deep learning libraries running on GPU and 19.29 times speed up against CPU. Second, the trained RL policy is transferred to a real robot arm. Our experiments show that the trained arm can successfully reach to and pick up predefined objects, demonstrating the feasibility of our approach.**

I. Introduction

Reinforcement Learning (RL) is a branch of machine learning concerning how to make sequential decisions. In the typical setting of RL, there is an agent interacting with the environment. In each time step t, the agent observes the state of the environment s_t and takes an action a_t, according to its decision-making policy π. The environment receives a_t and gives a scalar reward r_t to the agent. As the environment is affected by the agent's action, the environment state s will change to s_{t+1} in the next time step, then the agent will observe the new state s_{t+1}, take a new action a_{t+1}, and receive another reward r_{t+1}. This loop goes on and on. The agent aims to maximise the long-term cumulative reward gathered from the environment thorough trial and error. The agent gradually adjusts his decision-making policy π to achieve this goal.

Historically, a major driving force behind RL is game playing, in which the RL agent is the player and the reward is the score or the eventual win or lose. For example, Google's RL-based algorithm AlphaGo defeated human world champion Lee Sedol in 2016 [1]. Besides, as many real world problems are essentially sequential decision making problems, RL has been applied to other domains. In robotics, the state s is the robot's position, velocity, etc.; the policy π is the control logic; and the action a is the control signal sent to the robot's motors. Reward can be given for following the desired trajectory. RL has been applied to controlling robotic arms, autonomous vehicles, and humaniod robots, etc [2] [3] [4].

An important family of RL algorithms are Policy Gradient methods. Benefited from the capability to deal with continuous action space efficiently, Policy Gradient has been successful for robotic control [2]. The main requirement of Policy Gradient is that the agent's policy π must be differentiable. Currently, deep neural network is the most prevalent policy, which satisfies this requirement. Using θ to denote the parameters in the policy, such as the weights in the neural network, the policy π is parameterised as π_θ. Given an objective function $J(\pi_\theta)$, such as the expected cumulative reward, the gradient of the objective function with respect to the policy parameters is $\nabla_\theta J(\pi_\theta)$. Policy Gradient methods try to maximise $J(\pi_\theta)$ using gradient-based optimisation, i.e. $\Delta_\theta = \alpha \nabla_\theta J(\pi_\theta)$, where α is the step size. This process leads to an improved π_θ for higher reward.

There is published work on RL for robotic control based on simulated robotic benchmarks [3] [5]. This is possibly due to the difficulty and cost of obtaining a real robot and experimenting with it. Research efforts have been focusing on improving the RL algorithms, which are evaluated in simulated benchmarks [3] [5]. Currently, state-of-the-art Policy Gradient algorithms demonstrate solid performance in simulated robotic locomotion benchmarks [5]. While continuing improving the RL algorithms themselves such that the robot can walk more smoothly in simulation is beneficial, we choose a different perspective in this paper. Starting from Trust Region Policy Optimisation (TRPO), we select an advanced Policy Gradient algorithm that works well in simulation [3]. We study how to use this algorithm to control a real robot automatically, and how to use FPGA-based hardware acceleration to address the computational challenges that arise while we are doing so.

The robot we use is a robot arm with a gripper, and the task is to automatically reach and pick up an item from the table. We create a simulation model for the robot arm in software and build an integrated framework connecting physical robotic simulation and Reinforcement Learning library together. We train a neural network based RL agent in simulation, using the TRPO algorithm. As training is time consuming, we propose an FPGA-based hardware acceleration strategy for TRPO algorithm, with automated design space exploration. We implement the trained neural network on the development board to control the robot arm, and let the real system run in action. We adjust the hyper parameters of the training stage based on the robot arm's behaviour in both simulation and real

world tests. After adequate tuning, the robot arm can perform the task of reaching and picking up an item from the table successfully. Specifically, the major contributions of this paper are:

- A workflow of applying accelerated Reinforcement Learning to control a real robot: simulation-based RL training (TRPO algorithm), implementation on the real robot, and hyper parameter tuning.
- FPGA-based hardware acceleration of the TRPO algorithm, with automated design space exploration. Implementation on Stratix-V 5SGSD8 FPGA showed 4.65 times speed-up against Keras+Theano deep learning libraries running on Tesla C2070 GPU, and 19.29 times speed-up against the libraries running on i7-5930K CPU.
- Application of the proposed workflow to a real robot arm, using Reinforcement Learning to control the arm to reach and pick up an item automatically.

The rest of this paper is organised as follows. Section II covers background. Section III details the proposed hardware acceleration of TRPO algorithm. Section IV presents a case study with the real robot arm. Finally, Section V presents the conclusion and suggests future work.

II. Background

A. Trust Region Policy Optimisation (TRPO)

In this paper, we use Trust Region Policy Optimisation (TRPO), an advanced RL algorithm, to train the RL agent to perform the robotic control task. Here we briefly review the key points of TRPO. Full details can be found in [3].

Consider an infinite-horizon discounted Markov Decision Process (MDP), defined by tuple $(\mathcal{S}, \mathcal{A}, P, r, \rho_0, \gamma)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ the set of actions, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the transition probability distribution, $r : \mathcal{S} \to \mathbb{R}$ is the reward function, $\rho_0 : \mathcal{S} \to \mathbb{R}$ is the distribution of initial state s_0, and $\gamma \in (0, 1)$ is the discount factor. Let policy π_θ be a stochastic policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which is a conditional distribution over actions given states $\pi_\theta(a|s) = \mathbb{P}_\theta(\mathcal{A}_t = a|\mathcal{S}_t = s)$. Here θ represents the policy parameters. Let $\eta(\pi_\theta)$ denote the expected discounted total reward by following policy π_θ:

$$\eta(\pi_\theta) = \mathbb{E}_{s_0, a_0, \dots}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t)\right] \tag{1}$$

where $s_0 \sim \rho_0(s_0)$, $a_t \sim \pi_\theta(a_t|s_t)$, $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$.

The state-value function V_π is the expected reward starting from state s_t and then following policy π_θ:

$$V_{\pi_\theta}(s_t) = \mathbb{E}_{a_t, s_{t+1}, \dots}\left[\sum_{l=0}^{\infty} \gamma^l r(s_{t+l})\right] \tag{2}$$

The action-value function Q_{π_θ} is the expected reward starting from state s_t, taking action a_t and then following policy π_θ:

$$Q_{\pi_\theta}(s_t, a_t) = \mathbb{E}_{s_{t+1}, a_{t+1}, \dots}\left[\sum_{l=0}^{\infty} \gamma^l r(s_{t+l})\right] \tag{3}$$

Algorithm 1 Conjugate Gradient Algorithm

Input: $\mathbf{A}$, $\mathbf{b}$, Maximum Iterations MaxIter, Threshold Th
Output: Solution to the linear equation $\mathbf{Ax} = \mathbf{b}$

1: **procedure** CONJUGATE GRADIENT($\mathbf{A}, \mathbf{b}$, MaxIter, Th)
2: Initialise $\mathbf{p} = \mathbf{b}$, $\mathbf{r} = \mathbf{b}$, $\mathbf{x} = \mathbf{0}$, $\rho = \mathbf{r}^\intercal\mathbf{r}$
3: Initialise $iter = 0$
4: **while** $\rho >$ Th **and** $iter <$ MaxIter **do**
5: $\mathbf{z} \leftarrow \mathbf{Ap}$ ▷ Fisher-Vector Product (FVP)
6: $v \leftarrow \mathbf{r}^\intercal\mathbf{r}/\mathbf{p}^\intercal\mathbf{z}$
7: $\mathbf{x} \leftarrow \mathbf{x} + v\mathbf{p}$
8: $\mathbf{r} \leftarrow \mathbf{r} - v\mathbf{z}$
9: $\rho_{new} \leftarrow \mathbf{r}^\intercal\mathbf{r}$
10: $\mathbf{p} \leftarrow \mathbf{r} + (\rho_{new}/\rho)\mathbf{p}$
11: $\rho \leftarrow \rho_{new}$
12: $iter \leftarrow iter + 1$
13: **end while**
14: **return** $\mathbf{x}$
15: **end procedure**

By subtracting V_{π_θ} from Q_{π_θ}, the advantage function A_{π_θ} indicates the advantage of a specific action a over average:

$$A_{\pi_\theta}(s, a) = Q_{\pi_\theta}(s, a) - V_{\pi_\theta}(s) \tag{4}$$

During each iteration, policy parameters θ will be updated. In TRPO, the new θ is chosen by solving the following constrained optimisation problem:

$$\begin{aligned} \max_\theta \quad & \mathbb{E}_{s\sim\rho_{\theta_{old}}, a\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s, a)\right] \\ \text{subject to} \quad & \mathbb{E}_{s\sim\rho_{\theta_{old}}}\left[D_{KL}(\pi_{\theta_{old}}(\cdot|s)||\pi_\theta(\cdot|s))\right] \le \delta_{KL} \end{aligned} \tag{5}$$

where $\rho_{\theta_{old}}$ is the discounted state-visitation frequencies induced by $\pi_{\theta_{old}}$, D_{KL} the Kullback-Leibler (KL) divergence, and δ_{KL} the maximum KL divergence allowed.

KL divergence is a measure of difference between two probability distributions P and Q, defined as follows:

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) log \frac{p(x)}{q(x)} dx \tag{6}$$

The optimisation problem (5) is solved in each TRPO iteration with the following two steps:

1. Compute a search direction via Conjugate Gradient (CG). The general framework of CG is shown in Algorithm 1.
2. Perform a line search in that direction, ensuring that the objective is improved without violating the KL constraint.

In each CG iteration, we will need to compute $\mathbf{z} = \mathbf{Ap}$ (line 5 of Algorithm 1). Vector $\mathbf{z}$ is called Fisher-Vector Product (FVP). As all other computations in the CG iteration have linear time complexity, the computation of FVP dominates the total computing time. $\mathbf{A}$ is the Fisher-Information Matrix (FIM), approximatly calculated using the training samples:

$$\mathbf{FIM} \approx \frac{1}{N}\sum_{n=1}^{N} \frac{\partial^2}{\partial\theta_i \partial\theta_j} D_{KL}(\pi_{\theta_{old}}(\cdot|s_n)||\pi_\theta(\cdot|s_n)) \tag{7}$$

where $n = 1, \cdots, N$ denotes each sample in the data set and i, j denote the parameters in policy π_θ.

B. Reinforcement Learning on FPGA

FPGA acceleration of Machine Learning has received great attention in the past few years, with much of the research work focusing on Convolutional Neural Networks (CNN).

Accelerating Reinforcement Learning on FPGA is a new direction. Su, Liu, Thomas and Cheung proposed an architecture for Q-Learning on FPGA [6]. Q-Learning is an important family of RL algorithms that tries to learn the action-value Q function (3), which has been successful in game playing.

For robotic control tasks, Policy Gradient methods are more suitable. We have explored FPGA acceleration of Policy Gradient algorithms, and developed a hardware architecture for Trust Region Policy Gradient (TRPO) [7]. Our previous work uses Pearlmutter Propagation to evaluate Fisher-Vector Product (FVP), the computational bottleneck of TRPO. With that architecture, the loop unrolling factor of each layer in the neural network needs to be optimised to achieve high efficiency, but such design space exploration is missing. We address this exploration next.

III. Hardware Acceleration of TRPO

In RL-based robotic control, the RL agent is the robot's controller. The agent's policy π is usually a Multi-Layer Perception (MLP) neural network to be trained by the RL algorithm. In our paper, we use TRPO to train the agent [3]. Our work is based on the hardware architecture for TRPO proposed in [7]. We focus on design space exploration, i.e. finding the optimal loop unrolling parameter for each neural network layer, which is the key to high performance.

A. Design Space Exploration Problem

The Design Space Exploration (DSE) problem of finding optimal loop unrolling parameters to accelerate TRPO on FPGA is challenging because: a) the solution space grows exponentially with respect to the number of layers; b) loop unrolling parameters of different layers affect each other.

Here we provide a systematic DSE approach. The general workflow is shown in Fig. 1. Given the neural network size and the FPGA specification, we can build performance model (the number of cycles processing one training sample) and resource model (the critical resource the design will consume for a given set of loop unrolling parameters).

For the case of TRPO, the critical resource is DSP block on the FPGA chip, as dense matrix-vector multiplication dominates the computation. We use P to denote the set of loop unrolling parameters, $LayerSize_i$ to denote the size of layer i, then the DSE problem is given as follows:

$$\begin{aligned} \min_{P} \quad & Cycle \\ \text{subject to} \quad & DSP \leq DSP_{FPGA} \\ & BW \leq BW_{mem} \\ & P_i \leq \text{LayerSize}_i \end{aligned} \tag{8}$$

This reflects three natural constraints: hardware resource constraint (DSP); memory bandwidth constraint (BW_{mem}); and problem specific constraint (unrolling factor of a given layer cannot exceed the size of that layer). We select P by solving this constrained optimisation problem.

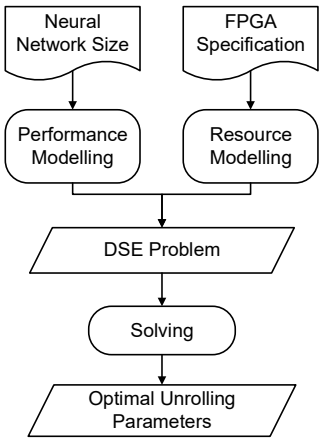

Fig. 1. Design Space Exploration (DSE) Procedure.

Our design space exploration differs from past papers accelerating CNN in the following ways:

- In RL we use an MLP network, rather than CNN
- Pearlmutter Propagation [7] [8] is used in training, rather than the conventional back propagation method
- We focus on training, rather than the inference stage

These differences lead to a different performance model, involving the number of cycles to process one training sample - the objective function to be minimised.

As the loop unrolling parameters must be integers, the design space exploration problem is a constrained integer programming problem. Currently, we write a python script to solve it via brute force. The 4 layer neural network used in our experiments takes less than 10 seconds to solve with our i7-4770 CPU. For larger problem sizes, standard integer programming software packages can be used.

B. Performance Model

When accelerating TRPO, we focus on the Fisher-Vector Product (FVP), which is the computational bottleneck [7]. To evaluate FVP, we need to carry out standard Forward Propagation, Pearlmutter Forward Propagation, and Pearlmutter Back Propagation for each layer in the neural network.

Standard Forward Propagation is given as follows:

$$x_i = \sum\nolimits_j w_{ji} y_j + b_i \tag{9a}$$

$$y_i = \sigma_i(x_i) \tag{9b}$$

where y_j are the inputs from the previous layer, w_{ji} and b_i are the weights and biases, x_i is the pre-activated value, $\sigma()$ is the activation function, and y_i is the output of this layer.

Let $E = E(y)$ be the loss function, the Pearlmutter Forward Propagation is given as follows:

$$\Re\{x_i\} = \sum\nolimits_j (w_{ji}\Re\{y_j\} + p_{ji} y_j) + p_i \tag{10a}$$

$$\Re\{y_i\} = \Re\{x_i\}\sigma_i'(x_i) \tag{10b}$$

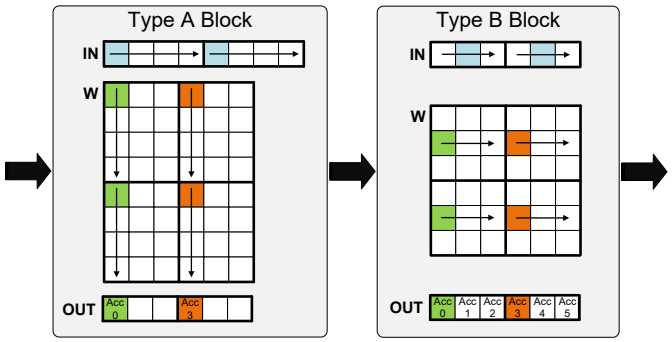

Fig. 2. Type A and Type B blocked matrix-vector multiplication. Both are 2×2 blocked for illustration purpose. In a Type A Block, the matrix is traversed in column major. Each inner loop generates two output elements, which are immediately passed onto the Type B Block. In a Type B Block, the matrix is traversed in row major. The current available input elements are multiplied with the corresponding matrix elements in several columns, generating several partial results for the corresponding outputs. These partial results are accumulated, and final results come out in the last inner loop.

where $\mathfrak{R}\{\cdot\}$ is the Pearlmutter differentiation operator [8], p_{ji} and p_i are the elements in $\mathbf{p}$ that correspond to w_{ji}, and b_i, respectively. $\mathbf{p}$ is the vector in the FVP equation $\mathbf{z} = \mathbf{A}\mathbf{p}$.

Standard Forward Propagation and Pearlmutter Forward Propagation can be merged together to form Combined Forward Propagation. Then, using the results calculated by the Combined Forward Propagation, we carry out the Pearlmutter Back Propagation [7]:

$$\mathfrak{R}\left\{\frac{\partial E}{\partial y_i}\right\} = e_i'(y_i)\mathfrak{R}\{y_i\} + \sum_j w_{ij}\mathfrak{R}\left\{\frac{\partial E}{\partial x_j}\right\} \tag{11a}$$

$$\mathfrak{R}\left\{\frac{\partial E}{\partial x_i}\right\} = \sigma_i'(x_i)\mathfrak{R}\left\{\frac{\partial E}{\partial y_i}\right\} \tag{11b}$$

$$\mathfrak{R}\left\{\frac{\partial E}{\partial w_{ij}}\right\} = y_i\mathfrak{R}\left\{\frac{\partial E}{\partial x_j}\right\} \tag{11c}$$

$$\mathfrak{R}\left\{\frac{\partial E}{\partial b_i}\right\} = \mathfrak{R}\left\{\frac{\partial E}{\partial x_i}\right\} \tag{11d}$$

While these equations look complicated, when we are evaluating them, each term, such as $\mathfrak{R}\left\{\frac{\partial E}{\partial y_i}\right\}$, is just a number. Therefore, most of the computations are essentially dense matrix-vector multiplication, which can be efficiently parallelised on FPGA via blocked matrix-vector multiplication.

We have two types of blocked matrix-vector multiplication, Type A and Type B, which are illustrated in Fig. 2 [7].

In a Type A block, the inner loop is the loop over input vector, and the weight matrix is traversed in a column major manner. An output item will be produced at the end of each inner loop. In a Type B block, the loop over input is the outer loop, and the weight matrix is traversed in row major. Partial results are accumulated, and final results come out in the last inner loop.

The two types of matrix-vector multiplication can be cascaded in an A-B-A-B manner to efficiently carry out forward and back propagation for a multi-layer neural network. The first layer uses Type A block, as it reads inputs from memory and it can read new values every cycle. The second layer's Type B block can start working as soon as the first batch of the results from the first layer becomes available. When the second layer outputs its results in the last inner loop, the third layer's Type A block can start. By arranging the layers in an A-B-A-B manner, the propagation between adjacent layers can be pipelined, which effectively reduces the number of cycles needed for computation.

We model the number of cycles the system needs, to be used in design space exploration. We will first look at the Type A and Type B blocks. Then we model the whole system.

For matrix-vector multiplication, we use $\mathsf{InLayerSize}$ and $\mathsf{OutLayerSize}$ to denote the dimension of input and output, respectively. Assume input dimension is blocked into $\mathsf{NumInBlocks}$ blocks, and output dimension $\mathsf{NumOutBlocks}$ blocks. Then the weight matrix is blocked into $\mathsf{NumInBlocks} \times \mathsf{NumOutBlocks}$ blocks. If stream padding is used, $\mathsf{InLayerSize}$ and $\mathsf{OutLayerSize}$ should be the padded dimension numbers.

The dimension in each rectangular block is given as follows:

$$\mathsf{InBlockDim} = \mathsf{InLayerSize}/\mathsf{NumInBlocks}$$
$$\mathsf{OutBlockDim} = \mathsf{OutLayerSize}/\mathsf{NumOutBlocks}$$

For Type A block, the number of cycles needed is:

$$Cycle_A = \mathsf{InBlockDim}_A \times \mathsf{OutBlockDim}_A \tag{12}$$

For Type B block, we have data dependencies as it needs to wait for the results from the Type A block in the previous layer. Also, as each inner loop calculates a partial result to be buffered and accumulated, we need to take computational latency into account. Let $\mathsf{InBlockDim}_{\mathsf{prev}}$ be the $\mathsf{InBlockDim}_A$ of the previous layer's Type A block and CL be the computational latency of calculating the partial result. Each inner loop in Type B block needs $InnerCycle_B$ cycles:

$$InnerCycle_B = max(\mathsf{InBlockDim}_{\mathsf{prev}}, \mathsf{OutBlockDim}_B, \mathsf{CL})$$

The total number of cycles needed for Type B block is:

$$Cycle_B = InnerCycle_B \times \mathsf{InBlockDim}_B \tag{13}$$

When cascading A and Type B, the number of cycles is:

$$Cycle_{AB} = \mathsf{InBlockDim}_A + (\mathsf{InBlockDim}_B - 1) \times Cycle_B$$

The equation above places a lower bound on the cycles needed for each inner loop in Type B block, it would be a waste of resources to increase parallelism too high. Ideally, the optimal configuration should satisfy the following condition:

$$\mathsf{InBlockDim}_A \geq \mathsf{OutBlockDim}_B \geq \mathsf{CL} \tag{14}$$

This is the most efficient case since the latency of Type B block is hidden in that of Type A block. In this case, the total number of cycles needed for the two layers is:

$$Cycle_{AB} = \mathsf{InBlockDim}_A \times \mathsf{OutBlockDim}_A = Cycle_A \tag{15}$$

Longer cascaded A-B-A-B-... blocks can be modelled in the same way. The starting cycle of a block in layer #i is always the cycle that the first item from layer #i-1 becomes

available. In this way, we can derive $Cycle_{FP}$ and $Cycle_{BP}$, the number of cycles of the Combined Forward Propagation and that of the Pearlmutter Back Propagation, respectively. We overlap the Pearlmutter Back Propagation of sample #i with the Combined Forward Propagation of sample #i, as they are independent. Thus, the total number of cycles for processing one training sample is:

$$Cycle = max(Cycle_{FP}, Cycle_{BP}) \tag{16}$$

C. Evaluation

We use the `Humanoid-v1` benchmark from OpenAI Gym for evaluation [9]. The task is to control a humanoid robot to run, simulated by MuJoCo [10]. The observation space $\mathcal{S}$ is 376-dimensional (position, velocity, center of mass of each element, etc.), the action space $\mathcal{A}$ is 17-dimensional (torque control commands). This is the most complex MuJoCo benchmark in OpenAI Gym. It is also used in paper [5] that evaluates a wide range of RL algorithms, including the TRPO.

1) Design Space Exploration: We use an MLP with two hidden layers to control the robot, sized at 376 (input) - 128 - 64 - 17 (output). We use P_0, P_1, P_2, P_3 to denote the loop unrolling pamameters, from input layer to output layer. Following the calculating procedure presented in Section III.B, the performance model eq. (16) could be derived:

$$Cycle = \frac{\text{LayerSize}_0}{P_0} * \frac{\text{LayerSize}_1}{P_1} + \frac{\text{LayerSize}_2}{P_2} * \frac{\text{LayerSize}_3}{P_3}$$

To construct the DSE optimisation problem eq.(8), we also need a DSP usage model. We use fixed-point numbers with 23 fractional bits for computation. The number of integer bits are set based on the numerical range of each variable. After working out data types, we can model DSP usage accurately based on bit-width (how many DSPs each operation takes) and parallelism (how many operations). In our case, we have:

$$DSP = 4P_0P_1 + 8P_1P_2 + 7P_2P_3 + 19P_1 + 19P_2 + 2P_3$$

We use a Maxeler MAX4 system with Stratix-V 5SGSD8 FPGA, which has 1963 DSPs ($DSP_{FPGA} = 1963$). For our system and the TRPO architecture, the bandwidth constraint can be represented as a bit-width constraint, limiting the input layer's parallelism P_0. The input vector from the memory, which contains P_0 double precision numbers, 64-bit each, must not exceed the width of the bus, which is 3072-bit. Thus we have $BW = 64P_0$, $BW_{mem} = 3072$.

Substituting the above expressions for $Cycle$, DSP and BW constraints into the design space exploration problem eq.(8), the optimal loop unrolling factors P_0, P_1, P_2, P_3 can be obtained by solving it. The solution and the corresponding FPGA resource usage are reported in Table I.

2) Performance Evaluation: Given the training data, solving the search direction via Conjugate Gradient (CG) is the most time consuming part, and FVP dominates CG. The training data come from the MuJoCo simulation (50000 samples). We write a CG solver in C, running on a Xeon E5-2697V2 CPU, in which the FVP is calculated by a Stratix-V 5SGSD8 FPGA running at 200MHz. We evaluate our C-FPGA hybrid system against the Keras deep learning library with Theano backend running on a workstation with Tesla C2070 GPU and Core i7-5930K CPU for the task of computing TRPO search direction via CG. Table II shows the performance comparison.

TABLE I
PARALLELISM AND RESOURCE USAGE OF STRATIX-V 5SGSD8 FPGA

$[P_0, P_1, P_2, P_3]$	Logic Slices (ALM)	DSP	BRAM (M20K)
[24,10,4,9]	154658 / 262400	1818 / 1963	2339 / 2567

TABLE II
PERFORMANCE COMPARISON - HUMANOID-V1 BENCHMARK

	T_{CG} (T_{FVP}) CPU	T_{CG} (T_{FVP}) GPU	T_{CG} (T_{FVP}) FPGA
Time	12.016s (12.014s)	2.900s (2.896s)	0.623s (0.609s)
CG Speed	1.00×	4.14×	19.29×

TABLE III
THEORETICAL PERFORMANCE PREDICTION FOR HUMANOID-V1

Stratix-V Config. (DSP)	Cycles	Stratix-10 Config. (DSP)	Cycles
[24,10,4,9] (1818)	240	[47,16,8,17] (5474)	72

Here, T_{CG} (T_{FVP}) is the actual elapsed time of Conjugate Gradient (CG) with the elapsed time of FVP computation inside the brackets, measured in the experiments on CPU, GPU and C-FPGA hybrid. For CG, the proposed C-FPGA hybrid system achieves 4.65 times speed-up against the deep learning libraries running on Tesla C2070 GPU, and 19.29 times speed-up against them running on Core i7-5930K CPU.

Beyond high speed, the FPGA solution also has high energy efficiency. When running the system on FPGA hardware, the FPGA power consumption is 22.5W, as reported by the Maxeler system monitoring tool. In contrast, the Tesla C2070 GPU consumes 238W. Taking both speed and power into account, FPGA is 49.19 times more energy efficient than GPU.

D. Discussion

Given the neural network size and the FPGA, we can obtain optimal loop unrolling factors by solving the DSE problem automatically in python. Multiple optimal solutions with the same number of cycles may exist, but some may be easier to place and route than others. By solving the DSE problem automatically, we find all non-obvious optimal solutions. We then try these optimal solutions and find one that can utilise 92.6% of the DSPs and successfully place and route. This helps us achieve a 43% performance gain against paper [7], which uses the same type of FPGA for the `Humanoid-v1` benchmark, but only utilises 69.7% DSPs available.

We can also use the model for planning. For example, assuming we use a Stratix-10 FPGA (5760 DSPs), we can reduce the number of cycles to 1/3 of that of Stratix-V, which is shown in Table III. As the system is bounded by computation, with around 3 times the number of DSPs in Stratix-10 we expect to triple the performance. Moreover, as Stratix-10 can run at a higher clock frequency than Stratix-V, the actual performance boost will be even higher.

IV. RL-based Robotic Control

In this section, we will first describe the general workflow of Reinforcement Learning based robotic control, then we present the robot arm case study to show how the workflow is applied to a real robot.

A. The General Workflow

When using Reinforcement Learning to control robots, the general workflow of the project can be summarised in Fig. 3.

1) Specifications: The starting point is the robot specification and the task specification. The robot specification is needed to build the simulation model. The task specification is needed to build the environment model, select the appropriate RL algorithm, and design the reward function.

2) Modelling and Hyperparameter Tuning: Based on the specifications, a simulation model of the robot and the environment can be built, which will be used in training.

The choice of training algorithm also depends on the task. As TRPO works well for a wide range of robotic locomotion tasks, we use it as the training algorithm [3].

The hyperparameters for training are also determined at this stage, such as the size of the neural network and the parameters for the training algorithm. In particular, an important hyper parameter is the reward function, which depends on the task. The initial values of hyper parameters can be set according to published results performing similar tasks. The hyper parameters need to be tuned if current values do not work well. Hyper parameter tuning is a trial and error process itself.

3) Training in Simulation: With a simulation model of the robot and a suitable RL algorithm, the RL agent (the controller of the robot) can be trained in simulation. Training in simulation is computationally intensive, but it has the following important advantages:

- Simulation is much faster and cheaper than operating the real robot.
- Training in simulation largely prevents the real robot from being damaged by an immature controller.
- The computational challenge can be addressed by hardware acceleration, such as the FPGA-based framework proposed in Section II.

4) Simulation Test: The trained RL-agent can be tested in simulation. If it controls the robot well in simulation then we can proceed to the next stage, otherwise we need to adjust the models and/or hyper parameters and re-train.

5) Implementation on the Real Robot: The trained RL-agent can be implemented on the real robot. The RL-agent used for robotic control is usually a neural network, which can be efficiently accelerated by FPGA. For non-trivial robotic control in real time, hardware acceleration is necessary.

6) Real World Test: The operation of the robot is evaluated in reality. If it works adequately then the project is finished; if it does not work well, which is not surprising since the simulation cannot capture every fine detail of the robot and the environment, we will need to adjust the model and/or the hyperparameters and start over. This is an iterative trial and error process.

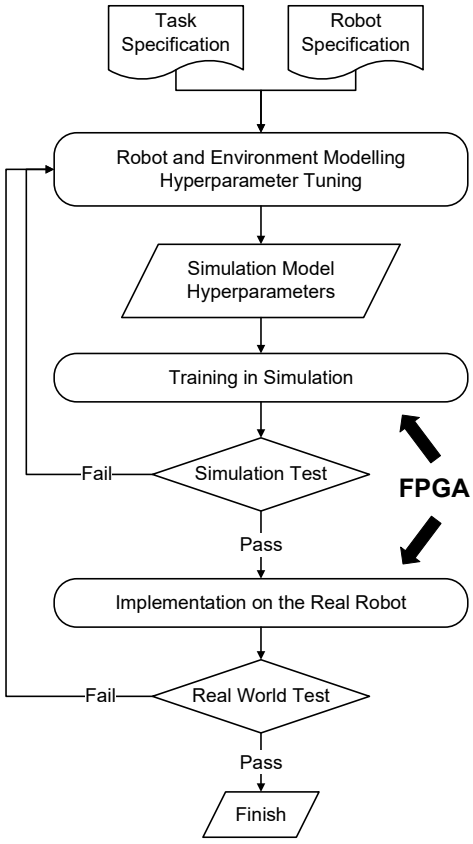

Fig. 3. RL-based Robotic Control Development Procedure. FPGA can assist both the simulation-based training (training stage) and the implementation of the trained controller (inference stage).

B. Task Specification and Robot Specification

We use the Lynxmotion robot arm with a gripper [11], shown in Fig. 4. Apart from the gripper, the robot arm has four joints: the base, the shoulder, the elbow and the wrist. Each servo is powered by a motor, controlled by Pulse-Width Modulation (PWM) signal. The robot arm is controlled by a development board with a MAX-10 FPGA and an Atom CPU. The task is to reach and pick up a whiteboard pen.

To tell the positions of the joints and the pen, we place geometric tags (pentagon, square, triangle) on them and use an Intel RealSense R200 3D Camera to track the geometric tags. The RealSense Camera is placed 60cm away from the robot arm at a fixed location. Raw data from the camera are processed by Intel LibRealSense software and OpenCV running on the Atom CPU on the development board to obtain the (X, Y, Z) coordinates, which are sent to the RL-based controller. Fig. 5 shows the geometric tags being tracked.

C. Simulation Environment

We use MuJoCo for physical simulation [10]. The simulation model of the robot arm created in MuJoCo is shown in Fig. 6. The MuJoCo software is connected to the OpenAI Gym, an RL library [9]. The OpenAI Gym runs on the top of

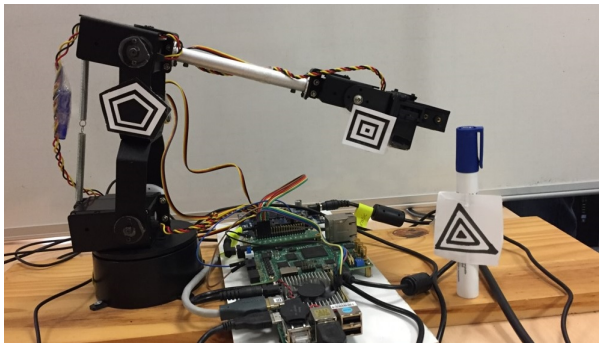

Fig. 4. System Configuration

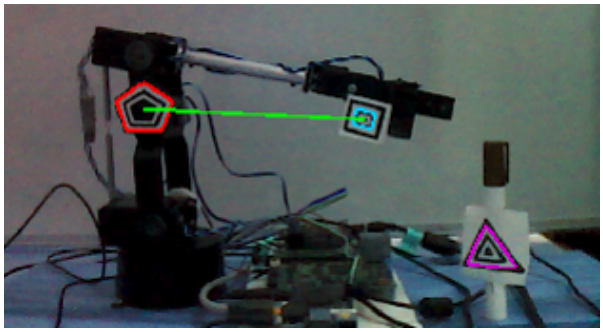

Fig. 5. Geometric tag tracking. We use Intel RealSense Camera to track the geometric tags so as to obtain (X, Y, Z) coordinates of the joints and the pen.

machine learning libraries Keras and Theano. We implement the TRPO algorithm in the OpenAI Gym.

OpenAI Gym, Keras and Theano are all python software. As our FPGA system does not have an FPGA-python interface currently, the training stage runs in software for now. An FPGA-python interface available in the future will enable effective system integration.

D. Hyperparameters

1) Reward Function: In RL, the agent tries to maximise the cumulative reward. Therefore, when using RL to control the robot, we must make sure that the only way to maximise the reward is to move along the desired trajectory.

For our robot arm, the task is to reach and pick up an item from the table. As the gripper hardware only has two modes, 'grasp' and 'release', its control is straightforward and is implemented with a few API function calls (not controlled by RL). As soon as the gripper reaches to the target item, the grasp function will be called to pick it up.

Therefore, the task for the RL agent is to move the robot arm so that the gripper reaches to the target item automatically. We design the reward function as follows:

$$R_t = -100 * d(\text{gripper}, \text{target})^2 - ||a||_2^2 \qquad (17)$$

where $d(\text{gripper}, \text{target})$ is the Euclidean distance between the gripper and the target item, and $||a||_2$ is the 2-norm of the gripper's acceleration.

Since $-100 * d(\text{griper}, \text{target})^2$ is negative unless the gripper reaches the target, it always punishes the RL agent. The closer to the target, the smaller the punishment, which encourages moving towards the target. Also, this 'always punishing' term encourages the RL agent to finish its task as quickly as possible to avoid future punishments. The second term $-||a||_2^2$ is a regularisation term to prevent vibration. It punishes the RL agent as long as the acceleration is non-zero, which encourages it to move the gripper at a constant speed. 100 is a factor balancing the two objectives.

The reward function is essentially describing the task in an RL manner, thus it mainly depends on the task but not the robot. The same reward function can be used to train other robots to reach and pick up the item, although each different robot requires its own MuJoCo simulation model for training.

2) Other Hyperparameters: We use an MLP neural network with two hidden layers, both sized at 16. The remaining hyperparameters all follow those in the TRPO paper [3].

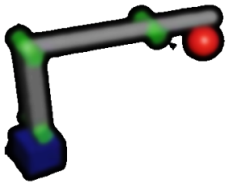

Fig. 6. Physical simulation model of the robot arm. The simulation model is created and simulated with the MuJoCo software [10]. The red ball represents the item to be grasped, which can be randomly positioned during simulation.

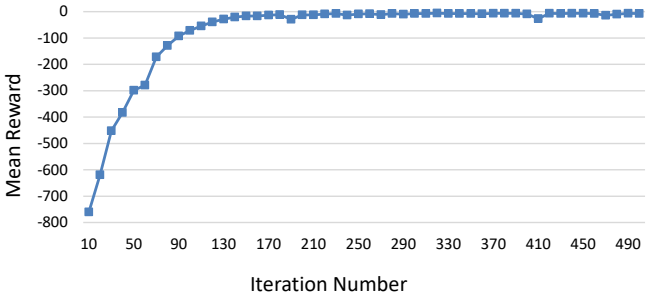

Fig. 7. Mean cumulative reward value vs. iteration number.

E. Training in Simulation

We use TRPO to train the RL agent in simulation. Fig. 7 shows the mean cumulative reward we get in each iteration. According to our reward function (17), the maximum possible reward is 0. The mean cumulative reward jumps quickly to 0 within 200 iterations and stays there. Thus, with the correct setting, TRPO trains the RL agent effectively and efficiently.

F. Implementation and Test with the Real Robot Arm

We port the trained RL agent (neural network) to the robot arm controller and run the system in reality. After observing the real world behaviour, we adjust the hyper parameters to further improve performance. The main adjustment is adding the $-||a||_2^2$ term to the reward function eq. (17), which significantly reduces robot vibration. The final video can be viewed at https://www.youtube.com/watch?v=aLklVQa8tXM. A screenshot from the video is provided in Fig. 8.

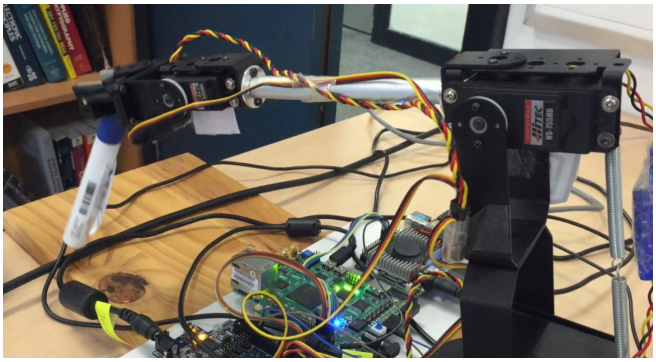

Fig. 8. Experiment with real robot arm. The trained RL agent (neural network) is implemented on the robot arm controller. The robot arm successfully reached to the pen and picked it up, automatically controlled by the neural network trained in simulation. This photo is a screenshot from the video, showing the final moment when the pen is picked up. The whole video can be viewed at https://www.youtube.com/watch?v=aLklVQa8tXM

Currently, the trained neural network is implemented in C and runs on the Atom CPU on the development board. This is because the robot arm is relatively simple so that a small neural network can control it well without much computational overhead. For a more complex robot performing non-trivial tasks, hardware acceleration of the controller will be necessary.

V. Conclusion and Future Work

In this paper, we explore FPGA-based acceleration for Reinforcement Learning (RL), targeting robotic control applications. The general workflow is to train the RL agent in simulation, then implement the trained RL agent on the controller of the real robot. Hyper parameters for training can be adjusted based on simulation and real world results.

We use MLP neural network as the RL agent and use Trust Region Policy Optimisation (TRPO) to train it. In the training stage, FPGA can be used to accelerate Fisher-Vector Product (FVP), the computational bottleneck of TRPO. Our contribution to the training stage is the automated design space exploration to obtain the optimal loop unrolling factors according to the neural network sizes and resource constraints, which is the key to high performance. The proposed system is evaluated using the MuJoCo robotic locomotion benchmark `Humanoid-v1`, showing that the proposed solution running on Stratix-V FPGA achieves up to 4.65 times speed-up against Keras+Theano deep learning library running on Tesla C2070 GPU, 19.29 times speed-up against them running on i7-5930K CPU, and 43% faster than a previous FPGA implementation.

We also apply the proposed workflow to a real robot arm with a gripper. The task is to control the robot arm to reach to and pick up an item from the table. We design the reward function to encourage the RL agent to move the gripper quickly and smoothly to the target item. We build a simulation model for the robot arm using MuJoCo software and train the RL agent with TRPO algorithm. The trained neural network implemented on the robot arm controller is able to successfully move the robot arm towards the target item and pick it up.

Future work is threefold. First is system integration. One solution is to implement the python machine learning libraries' equivalent in C and integrate with FPGA. Ideally, if an FPGA-python interface becomes available in the future, our FPGA design can be integrated with the python libraries directly for seamless hardware acceleration in the training stage.

Second is to explore the acceleration of physical simulation. TRPO's input data come from the physical simulation of the robot, currently carried out by MuJoCo, a single threaded software running on CPU only. Performance can be improved by accelerating the physical simulation on FPGA.

Third is to improve the training mechanism. In [4], multiple real robots are trained in parallel with knowledge sharing between them to accelerate the training process. A promising direction is to deploy similar strategy in our simulation based training to further increase parallelism for better efficiency. This could potentially lead to a multi-FPGA design.

The most exciting prospect of this research is its potential for improving robotic control by accelerating RL policy training. This paper explores this potential with a simple robot and a simple task; in the era of automation there is tremendous scope in exploring how the proposed approach can be applied to increase productivity, such as accelerating the training of industrial robots for future factory automation.

Acknowledgements

The support of UK EPSRC (EP/I012036/1, EP/L00058X/1, EP/L016796/1 and EP/N031768/1), the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, Altera, Intel, the Maxeler University Programme and the Lee Family Scholarship is gratefully acknowledged.

References

[1] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[2] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[3] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust Region Policy Optimization," in *ICML*, 2015, pp. 1889–1897.

[4] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3389–3396.

[5] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *ICML*, 2016, pp. 1329–1338.

[6] 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.

[7] 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)*, Sept 2017, pp. 1–6.

[8] B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural computation*, vol. 6, no. 1, pp. 147–160, 1994.

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

[10] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 5026–5033.

[11] B. P. Jeppesen, N. Roy, L. Moro, and F. Baronti, "An FPGA-based controller for collaborative robotics," in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, June 2017, pp. 1067–1072.