Customised Pearlmutter Propagation: A Hardware Architecture for Trust Region Policy Optimisation

Shengjia Shao and Wayne Luk
Department of Computing, Imperial College London
E-mail: {shengjia.shao12, w.luk}@imperial.ac.uk

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 to find an optimal policy that maximises the reward. Trust Region Policy Optimisation (TRPO) is a recent policy optimisation algorithm that achieves superior results in various RL benchmarks, but is computationally expensive. This paper proposes Customised Pearlmutter Propagation (CPP), a novel hardware architecture that accelerates TRPO on FPGA. We use the Pearlmutter Algorithm to address the key computational bottleneck of TRPO in a hardware efficient manner, avoiding symbolic differentiation with change of variables. Experimental evaluation using robotic locomotion benchmarks demonstrates that the proposed CPP architecture implemented on Stratix-V FPGA can achieve up to 20 times speed-up against 6-threaded Keras deep learning library with Theano backend running on a Core i7-5930K CPU.

I. INTRODUCTION

Reinforcement Learning (RL) is a branch of machine learning that addresses the sequential decision making problem of how an agent should take actions to maximise the cumulative reward gathered from 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 his policy $\pi$. The environment receives $a_t$ and gives a scalar reward $r_t$ to the agent. The environment state $s$ then changes to $s_{t+1}$, as it’s affected by the action. The agent’s task is to maximise his long-term cumulative reward by learning to behave optimally through trial and error.

As many real world problems are sequential decision making, RL is useful in various areas. In robotics, state $s$ is the robot’s position, velocity, etc.; policy $\pi$ is the control logic; and action $a$ is the control signal for motors; reward can be given for following the desired trajectory [1]. RL has also been successfully applied to game playing and finance.

An important class of RL algorithms are policy gradient methods. Assume we have a differentiable parameterised policy $\pi_\theta$, where $\theta$ denotes the policy parameters. Suppose we also have an objective function $J(\pi_\theta)$, such as the expected cumulative reward. Then the policy gradient is $\nabla_\theta J(\pi_\theta)$. Policy-gradient methods try to maximise $J(\pi_\theta)$ by gradient-based optimisation, i.e. $\Delta_\theta = \alpha \nabla_\theta J(\pi_\theta)$, where $\alpha$ is the step size. This process leads to an improved $\pi_\theta$ for higher reward.

Policy gradient methods are iterative algorithms. Each iteration is composed of gradient evaluation and parameter update. Usually at least hundreds of iterations are needed to achieve acceptable performance. To reduce the number of iterations, step size selection is critical. A trivial step size $\alpha$ will lead to too many iterations, while a step size too large may damage the policy rather than improving it.

Trust Region Policy Optimisation (TRPO) is a new policy gradient algorithm that selects step size $\alpha$ based on Kullback-Leibler (KL) divergence [2]. KL is a statistical measure of the difference between two probability distributions. In TRPO, step size $\alpha$ is maximised provided that the KL between the old policy $\pi_\theta$ and the new one $\pi_{\theta+\Delta_\theta}$ does not exceed a threshold. In practice, TRPO tends to give monotonic improvement with non-trivial step size, which helps it outperform previous policy gradient algorithms in a wide range of benchmarks like Cart-Pole, Mountain-Car, and MuJoCo robotic locomotion [3].

With TRPO, fewer iterations are needed, but each iteration becomes much more complicated - an optimisation problem with KL divergence as a constraint must be solved. This is carried out by a Conjugate Gradient (CG) solver. Inside the CG solver, the computation of Fisher-Vector Product (FVP) takes most of the time. Based on our profiling results running robotic locomotion benchmarks with Keras deep learning library, given the training data, CG takes around 80% of the computation time, and CG time is dominated by FVP computation.

With this in mind, we are interested in whether FPGA-based custom computing can improve the computational efficiency of TRPO by accelerating the critical part. However, FVP is tricky to compute, and a straightforward implementation will not achieve high performance. This paper proposes Customised Pearlmutter Propagation (CPP), a novel architecture for FVP computation on FPGA, optimised using the Pearlmutter algorithm for efficiency. The major contributions are:

- A hardware architecture based on Customised Pearlmutter Propagation (CPP), which computes Fisher-Vector Product (FVP) efficiently on FPGA.
- Integration of CPP within the Conjugate Gradient solver to accelerate the key computational bottleneck of Trust Region Policy Gradient Optimisation (TRPO) algorithm.
- Implementation on Stratix-V FPGA and experimental evaluation with MuJoCo robotic locomotion benchmarks, achieving up to 20 times speed-up against Keras deep learning library with Theano backend on i7-5930K CPU.

The rest of this paper is organised as follows. Section II covers background. Section III details the CPP hardware architecture. Section IV presents experimental evaluation. Finally, Section V presents the conclusion and suggests future work.
II. BACKGROUND

A. Trust Region Policy Optimisation (TRPO)

Here we briefly review the key points of TRPO. Readers should refer to the TRPO paper [2] for full details.

Consider a Markov Decision Process (S, A, P, r, ρ₀, γ), where S is the state space, A the action space, P : S × A × S → R the transition probability distribution, r : S → R the reward function, ρ₀ : S → R the distribution of initial state s₀, and γ the discount factor. Let policy πθ be a stochastic policy πθ : S × A → [0,1], which is a conditional distribution πθ(a|s) = Pθ(A_t = a|S_t = s), where θ are the parameters.

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

\[ V_\pi(s_t) = E_{a_t,s_{t+1},...} \left[ \sum_{t=0}^{\infty} \gamma^t r(s_{t+t}) \right] \]  

The action-value function Qπθ is the expected reward starting from s_t, taking action a_t and then following policy πθ:

\[ Q_\pi(s_t, a_t) = E_{s_{t+1}, a_{t+1},...} \left[ \sum_{t=0}^{\infty} \gamma^t r(s_{t+t}) \right] \]  

By subtracting Vπθ from Qπθ, the advantage function Aπθ indicates the advantage of a specific action a over average:

\[ A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \]  

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

\[ \max_{\theta} \mathbb{E}_{s \sim p_{\text{old}}, a \sim \pi_{\theta, \text{old}}} \left[ \pi_{\theta}(a|s) \cdot A_{\theta, \text{old}}(s, a) \right] \]  

subject to  

\[ \mathbb{E}_{s \sim p_{\text{old}}} \left[ D_{KL}(\pi_{\theta, \text{old}}(\cdot|s)||\pi_{\theta}(\cdot|s)) \right] \leq \delta_{KL} \]  

where p_{old} is the discounted state-visitation frequencies induced by π_{\theta, old}. D_{KL} the Kullback-Leibler (KL) divergence, and δ_{KL} the maximum KL allowed. KL is a measure of difference between two probability distributions P and Q:

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

In each TRPO iteration, (4) is solved with two steps:

1. Compute a search direction via Conjugate Gradient (CG).
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 z = Ap. Matrix A is the Fisher-Information Matrix. Vector z is the Fisher-Vector Product (FVP). As all other computations in the CG iteration are either O(N) or O(1), FVP dominates the total computing time. Matrix A is approximated as follows:

\[ A \approx H = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial^2}{\partial \theta_i \partial \theta_j} D_{KL}(\pi_{\theta, \text{old}}(\cdot|s_n)||\pi_\theta(\cdot|s_n)) \]  

where n = 1, · · · , N denotes each sample in the data set and i, j denote the parameters in policy πθ.

Equation (6) shows matrix H is the Hessian matrix of KL divergence with respect to θ, averaged over samples. Therefore, the dimension of H is the number of parameters in policy πθ, which can be extremely large. For instance, the πθ for the Humanoid-v1 benchmark used in this work has 57634 parameters, resulting in a Hessian with 3.3 billion numbers. Thus the explicit calculation and storage of H have to be avoided. The good news is that in the CG algorithm, there is no need to formulate H explicitly. All we need is the Fisher-Vector Product z = Ap ≈ Hp, and it can be computed indirectly. This is the reason why CG is used in TRPO.

Conventionally, FVP is computed in the following way [2]:

\[ Hp = \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \left( \nabla_\theta D_{KL_{\text{old}}}(\cdot|s_n)||\pi_\theta(\cdot|s_n) \right) \]  

where, the KL divergence \( D_{KL_{\text{old}}} = D_{KL}(\pi_{\theta, \text{old}}(\cdot|s_n)||\pi_\theta(\cdot|s_n)) \) based on sample n is used as a loss function. Its gradient with respect to θ can be computed via standard backpropagation. Then the dot product of the gradient and vector p is computed. Finally the dot product is back propagated again to obtain FVP. The symbolic differentiation tools provided by deep learning libraries can handle the computation without user effort.

B. Reinforcement Learning on FPGA

Much of the existing work accelerating deep learning on FPGA is focused on Convolutional Neural Networks (CNN). There is only one paper about accelerating reinforcement learning on FPGA, proposing an architecture for Deep Q-Learning [4]. Q-Learning tries to learn the action-value Q function (2). Unlike policy gradient, it does not have an explicit policy πθ. In contrast, action is selected by maximising the Q function, i.e. given the state s, select the action a that yields the maximum Q value \( Q(s, a) \). Q-Learning is effective for discrete action space. For continuous action spaces, maximising \( Q(s, a) \) can be difficult, and policy gradient methods tend to perform better. To our best knowledge, this work is the first paper to explore policy gradient methods on FPGA.
III. HARDWARE DESIGN

A. The Design Challenge

When using a deep learning library to compute FVP, the computation is based on equation (7). However, we cannot evaluate (7) in hardware in the same manner as software, and a straightforward approach will lead to poor efficiency, which is the biggest challenge of accelerating TRPO on FPGA.

In software, the deep learning library first builds a symbolic computational graph for FVP during the compilation process. Then it feeds samples through the computational graph during runtime, which actually calculates the FVP. The symbolic computational graph is built with the following 5 steps:

1. Symbolic Forward Propagation
2. Symbolic Back Propagation, evaluating gradient \( \nabla_{\theta} D_{KL} \)
3. Symbolic Dot Product \( \langle \nabla_{\theta} D_{KL}, p \rangle \)
4. Symbolic Differentiation \( \frac{\partial}{\partial \text{out}_j} \langle \nabla_{\theta} D_{KL}, p \rangle \)
5. Symbolic Back Propagation, calculating \( \nabla_{\theta} \langle \nabla_{\theta} D_{KL}, p \rangle \)

Step 1 is a preparation for Step 2, as back propagation needs the internal values computed during forward propagation. Step 4 is the symbolic differentiation of \( \langle \nabla_{\theta} D_{KL}, p \rangle \) with respect to each output value of the neural network (\( \text{out}_j \)), which are the inputs to the second round of back propagation (Step 5).

Step 1, 2, 3, 5 are all fine on hardware. The problem lies in Step 4, where we need to differentiate the dot product with respect to each parameter in the neural network (\( \text{out}_j \)). However, back propagation for computing Hessian vector products [5] as well as the computational graph is built with the following 5 steps:

1. Symbolic Forward Propagation
2. Symbolic Back Propagation, evaluating gradient \( \nabla_{\theta} D_{KL} \)
3. Symbolic Dot Product \( \langle \nabla_{\theta} D_{KL}, p \rangle \)
4. Symbolic Differentiation \( \frac{\partial}{\partial \text{out}_j} \langle \nabla_{\theta} D_{KL}, p \rangle \)
5. Symbolic Back Propagation, calculating \( \nabla_{\theta} \langle \nabla_{\theta} D_{KL}, p \rangle \)

The FVP for one sample can then be evaluated in hardware as follows. These steps need to be repeated for each sample.

1. Standard Forward Propagation of the sample
2. Standard Back Propagation, evaluating gradient \( \nabla_{\theta} D_{KL} \)
3. Repeat for each node \( \text{out}_j \) in the output layer:
   3.1 Back propagate \( \text{out}_j \), calculating \( \frac{\partial \text{out}_j}{\partial \text{in}_j} \) in (9)
   3.2 Evaluate (9) by mult-add to obtain \( \frac{\partial \langle \nabla_{\theta} D_{KL}, p \rangle}{\partial \text{out}_j} \)
4. Standard Back Propagation, calculating \( \nabla_{\theta} \langle \nabla_{\theta} D_{KL}, p \rangle \)

Note that the number of items in equation (9) is the number of parameters in policy \( \pi_\theta \), which will be large for a neural network based policy. Worse: equation (9) needs to be evaluated for the number of output nodes. Consequently, Step 3 becomes a huge overhead, using much more time than other steps combined. Hardware implementation based on this straightforward approach will be very inefficient.

C. Pearlmutter Propagation

Our proposed approach is based on the Pearlmutter algorithm, which is a special kind of forward propagation and back propagation for computing Hessian vector products [5].

From calculus, Hessian vector product can be derived as:

\[
\mathbf{Hv} = \lim_{r \to 0} \frac{\nabla_{\mathbf{w}} (\mathbf{w} + r\mathbf{v}) - \nabla_{\mathbf{w}} (\mathbf{w})}{r} = \frac{\partial}{\partial r} \nabla_{\mathbf{w}} (\mathbf{w} + r\mathbf{v})\bigg|_{r=0}
\]

We define the Pearlmutter differential operator \( \mathcal{R}\{\cdot\} \):

\[
\mathcal{R}\{f(\mathbf{w})\} = \frac{\partial}{\partial f} f(\mathbf{w} + r\mathbf{v})\bigg|_{r=0}
\]

where \( \mathbf{v} \) is a known constant.

Given a feed forward neural network, in which the standard forward propagation in a layer is as follows:

\[
x_i = \sum_j w_{ji}y_j + b_i \tag{12}
\]
\[
y_i = \sigma_i(x_i)
\]

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, \( \sigma(\cdot) \) is the activation function, and \( y_i \) is the output of this layer.

Let \( E = E(y) \) be the loss function, and the derivative of \( E \) with respect to the output value \( y_i \) is \( \varepsilon_i = \partial E / \partial y_i \). The standard back propagation is then given as follows:

\[
\frac{\partial E}{\partial y_i} = \varepsilon_i(y_i) + \sum_{j} w_{ji} \left( \frac{\partial E}{\partial x_j} \right)
\]

\[
\frac{\partial E}{\partial x_i} = \sigma'_i(x_i) \left( \frac{\partial E}{\partial y_i} \right)
\]

\[
\frac{\partial E}{\partial w_{ij}} = y_i \left( \frac{\partial E}{\partial x_j} \right)
\]

\[
\frac{\partial E}{\partial b_i} = \frac{\partial E}{\partial x_i}
\]

By applying the \( \mathcal{R}\{\cdot\} \) operator to the standard forward propagation, we get the Pearlmutter Forward Propagation:

\[
\mathcal{R}\{x_i\} = \sum_j (w_{ji} \mathcal{R}\{y_j\} + v_{ji}y_j) + v_i \tag{14}
\]

\[
\mathcal{R}\{y_i\} = \mathcal{R}\{x_i\} \sigma'_i(x_i)
\]

where \( v_{ji} \) and \( v_i \) are the elements in vector \( \mathbf{v} \) that correspond to \( w_{ji} \) and \( b_i \), respectively.
Similarly, we can derive the Pearlmutter Back Propagation by applying the $\mathfrak{R}\{\cdot\}$ operator to standard back propagation:

$\mathfrak{R}\left\{\frac{\partial E}{\partial y_i}\right\}=e_i^\prime(y_i)\mathfrak{R}\{y_i\} + \sum_j w_{ij} \mathfrak{R}\left\{\frac{\partial E}{\partial x_j}\right\} + v_{ij} \frac{\partial E}{\partial E}\mathfrak{R}\{y_i\}$

$\mathfrak{R}\left\{\frac{\partial E}{\partial x_i}\right\} = \sigma_i^\prime(x_i) \mathfrak{R}\left\{\frac{\partial E}{\partial y_i}\right\} + \mathfrak{R}\{x_i\} \sigma_i^\prime(x_i) \frac{\partial E}{\partial y_i}$

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

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

(15)

With the Pearlmutter forward and back propagation, the elements in the Hessian vector product $\mathbf{Hv}$ are just $\mathfrak{R}\{\partial E/\partial w_{ij}\}$ and $\mathfrak{R}\{\partial E/\partial b_i\}$. The particular propagation in the Pearlmutter algorithm eliminates the need for change of variables. The general procedure of computing a Hessian vector product with the Pearlmutter algorithm is as follows:

1. Standard Forward Propagation
2. Standard Back Propagation
3. Pearlmutter Forward Propagation
4. Pearlmutter Back Propagation

Step 1-3 compute the values to be used in Step 4. Although the Pearlmutter forward and back propagation look complicated, they still have quadratic time complexity, which is the same as that of standard forward and back propagation. Compared with the straightforward approach in Section III.B, the Pearlmutter approach is much more efficient. Moreover, the Pearlmutter forward and back propagation follow the regular pattern of matrix-vector multiplication, which is hardware efficient.

D. CPP: Customised Pearlmutter Propagation

Now we apply the Pearlmutter propagation to the specific problem of FVP calculation within TRPO context. This allows us to exploit problem specific features to simplify computation.

In TRPO, a neural network based policy $\pi_\theta$ maps observation state $s$ to the mean vector $\mu$ of a $m$-dimensional diagonal Gaussian distribution $\pi_\theta(\cdot|s) = N(\mu, \sigma)$. The standard deviation is a stand-alone set of parameters $\sigma = \text{diag}(\sigma_1, \cdots, \sigma_m)$. Their natural logarithm, $\logstd_i = \ln(\sigma_i)$, are part of the policy parameters, which are also trained in each iteration, but are unrelated to the neural network. For FVP computation, the log loss function $E$ is the KL divergence, given by:

$$E = D_{KL}(\pi_{\theta_{old}}(\cdot|s)\|\pi_\theta(\cdot|s)) = \sum_{j=1}^{m} \left[ \ln\left(\frac{\sigma_j}{\hat{\sigma}_j}\right) + \left(\frac{\mu_j - \hat{\mu}_j}{\sigma_j}\right)^2 + \frac{\sigma_j^2}{2}\right] - \frac{m}{2}$$

(16)

(17)

where $\mu_j$ and $\sigma_j$ are constants whose values are taken from $\mu_j$ and $\sigma_j$, respectively.

Differentiating the equation above, it follows that the first order derivative of $E$ with respect to $\mu_j$ and $\sigma_j$ are zero:

$$\frac{\partial E}{\partial \mu_j} = 0 \quad \frac{\partial E}{\partial \sigma_j} = 0$$

(18)

Note that $\mu_j$ is the final output values of the neural network. Then following the back propagation equations (13), it can be derived that all first order derivatives are zero:

$$\frac{\partial E}{\partial y_i} = 0 \quad \frac{\partial E}{\partial x_i} = 0 \quad \frac{\partial E}{\partial w_{ij}} = 0 \quad \frac{\partial E}{\partial b_i} = 0$$

The zero first order derivative is a problem-specific feature allowing architecture customisation. First, the standard back propagation is no longer needed since we already know the results are 0. Second, by substituting zeros into the Pearlmutter Back Propagation equations (15), we simplify them as follows:

$$\mathfrak{R}\left\{\frac{\partial E}{\partial y_i}\right\} = e_i^\prime(y_i)\mathfrak{R}\{y_i\} + \sum_j w_{ij} \mathfrak{R}\left\{\frac{\partial E}{\partial x_j}\right\}$$

$$\mathfrak{R}\left\{\frac{\partial E}{\partial x_i}\right\} = \sigma_i^\prime(x_i) \mathfrak{R}\left\{\frac{\partial E}{\partial y_i}\right\}$$

$$\mathfrak{R}\{\partial E/\partial w_{ij}\} = y_i \mathfrak{R}\{\partial E/\partial x_j\}$$

$$\mathfrak{R}\{\partial E/\partial b_i\} = \mathfrak{R}\{\partial E/\partial x_i\}$$

(19)

The starting point of Pearlmutter back propagation is the final output layer. As the neural network generates the mean vector $\mu$, we have $y_i = \text{out}_i = \mu_i$. For the neurons in this layer:

$$\mathfrak{R}\{\partial E/\partial y_i\} = \mathfrak{R}\{\partial E/\partial \mu_i\} = \mathfrak{R}\{y_i\}/\sigma_i^2$$

(20)

For the natural logarithm of standard derivation $\sigma$, we have:

$$\mathfrak{R}\{\partial E/\partial \logstd_i\} = 2v_{\logstd_i}$$

(21)

where $v_{\logstd_i}$ is the item in $v$ that corresponds to $\logstd_i$. With the proposed scheme above, the FVP for each sample can be computed in hardware in two steps, which we call the Customised Pearlmutter Propagation (CPP):

1. Combined Forward Propagation - eq. (12) and (14)
2. Pearlmutter Back Propagation - eq. (19)

The Pearlmutter Forward Propagation needs the result from the standard forward propagation, but they can be merged in order to be evaluated side-by-side, which formulates the Combined Forward Propagation in Step 1. This merge means each sample only needs to be fed into the hardware once, rather than twice, which halves the communication overhead. Compared against the straightforward approach in III.B, the proposed procedure CPP based on Pearlmutter propagation has superior efficiency.

E. The CPP System Architecture

From a computational perspective, both the Combined Forward Propagation and the Pearlmutter Back Propagation are essentially dense matrix-vector multiplication. They can be efficiently implemented in hardware via blocked matrix-vector multiplication [6].

1) Type A and Type B Blocks: We have two types of blocked matrix-vector product, A and B, illustrated in Fig.1.

In 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 Type B block, the loop over input is the outer loop, and the weight matrix is traversed in row major. Partial results are buffered, and final results will 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
B cascading scheme for efficient pipeline between adjacent layers for back propagation. This setting implements the A-B-A-B type B block for forward propagation and a type A block for back propagation; the Even Layer has a type A block for forward propagation and a type B block for back propagation; the Odd Layer has a type A block for forward propagation and a type B block for back propagation; the second inner loop. When the second layer outputs its results to the memory, the second layer’s type B block can start its first inner loop when the first item from the first layer becomes available. As soon as the second item from the first layer arrives, the second layer’s type B block can start its second inner loop. When the second layer outputs its results in its last inner loop, the third layer’s type A block can start. Therefore, by cascading in an A-B-A-B manner, the propagation between adjacent layers can be pipelined, reduces the number of cycles needed for computation.

2) Overall System Architecture: The overall system architecture is shown in Fig. 2. We have two types of layers. The Odd Layer has a type A block for forward propagation and a type B block for back propagation; the Even Layer has a type B block for forward propagation and a type A block for back propagation. This setting implements the A-B-A-B cascading scheme for efficient pipeline between adjacent types.

layers. The DIFF block calculates $\delta E/\partial y_j$ based on eq. (20), preparing for the Pearlmutter Back Propagation. As we need to compute the FVP averaged over the whole data set, the FVP for each sample is accumulated inside the hardware, and finally sent back to CPU to be averaged.

The A-B-A-B cascading between adjacent layers is a fine-grained pipeline. We also implement a coarse-grained pipeline, overlapping the processing of adjacent training samples. This is because the FVP of different samples are independent. We begin the forward propagation of sample $i+1$ as soon as the forward propagation of sample $i$ finishes. Therefore, we are overlapping the back propagation of sample $i$ and the forward propagation of sample $i+1$, which further halves the number of cycles needed to compute FVP for the entire data set.

The system architecture is modular and parameterised. Multiple instances of Odd Layer and Even Layer modules can be instantiated according to the application. Also, type A and type B blocks are parameterised to support various matrix blocking schemes. Stream padding is implemented to handle the case in which a certain matrix dimension is not a multiple of the number of blocks in that dimension.

IV. EXPERIMENTAL EVALUATION

A. Benchmark Problems

In our experiment, we use two MuJoCo benchmarks from OpenAI Gym [7], Ant and Humanoid. They are also used in [3] that evaluates various RL algorithms, including TRPO. In both problems, the RL algorithm tries to learn how to control a robot to run from scratch, without any prior knowledge.

- Ant-v1: A robot with 13 rigid links and 8 actuated joints, shown in Fig. 3. The observation space $S$ is 111-dimensional, the action space $A$ is 8-dimensional. We use a neural network sized at 111 (input) - 64 - 32 - 8 (output) for this problem, with $\tanh(\_)$ activation.
- Humanoid-v1: A humanoid robot with many more links and joints, shown in Fig. 3. The observation space $S$ is 376-dimensional, the action space $A$ is 17-dimensional. We use a neural network sized at 376 (input) - 128 - 64 - 17 (output) for this problem, with $\tanh(\_)$ activation.

As mentioned before, solving the search direction via Conjugate Gradient (CG) is the most time consuming part of TRPO, and FVP computation dominates the CG time. We write a CG solver in C with FVP calculated by FPGA. The training data set comes from MuJoCo simulation, consisting of 50000 samples for each benchmark. We evaluate our C-FPGA hybrid system against the Keras deep learning library with Theano backend (double precision) running on CPU.
The result from FPGA is compared against that from Keras deep learning framework with Theano backend (double precision). Humanoid-v1 data set is used.

Fig. 4. Mean Absolute Percentage Error (MAPE) of the Conjugate Gradient result versus Number of Fractional Bits used in hardware. The result from FPGA is compared against that from Keras deep learning framework with Theano backend (double precision). Humanoid-v1 data set is used.

The 6-threaded Keras and Theano software run on a workstation with Core i7-5930K CPU (32nm, 6 cores, 2.5GHz).

C. Performance Evaluation

We compare the measured elapsed time for our C-FPGA hybrid system and the Keras deep learning framework with Theano backend to compute TRPO search direction via Conjugate Gradient algorithm, which is the most time consuming part. Table II shows the performance comparison.

The proposed hardware architecture is implemented on Maxeler’s MAX4 platform with a Stratix-V 5SGSD8 FPGA. FPGA clock frequency is 200MHz. The FPGA host computer has Intel Xeon E5-2640 CPU (32nm, 6 cores, 2.5GHz).

The 6-threaded Keras and Theano software run on a workstation with Core i7-5930K CPU (22nm, 6 cores, 3.5GHz).

Here, Model is the theoretical FVP computing time calculated by dividing the number of cycles to run by FPGA frequency. CG(FVP)FPGA and CG(FVP)Keras is the actual elapsed time for Conjugate Gradient with the elapsed time for FVP computation inside the brackets, measured in the experiments. Acc. is the C-FPGA speed-up of CG against 6-threaded Keras software with Theano backend.

The difference between model prediction and actual measured FVP computation time is due to FPGA API call latency. A higher speed-up is achieved for Ant-v1 benchmark, which has a smaller problem size. However, it is difficult to draw the big picture of how the system scales based on just two data points. We plan to explore the scalability in future work.

V. CONCLUSION AND FUTURE WORK

In this work, we propose a novel hardware architecture, Customised Pearlmutter Propagation (CPP), for accelerating Trust Region Policy Gradient (TRPO) on FPGA. The design addresses the key computational bottleneck of TRPO, which is the Fisher-Vector Product (FVP) computation inside the Conjugate Gradient (CG) solver. The proposed approach is based on the Pearlmutter algorithm, which enables an efficient hardware design, circumventing the key obstacle - symbolic differentiation with change of variables.

The proposed system is evaluated using two MuJoCo robotic locomotion benchmarks. Experimental results show that the proposed solution running on Stratix-V FPGA achieved up to 20 times speed-up against Keras deep learning framework with Theano backend running on i7-5930K CPU.

Future work includes further fine tuning of the CPP architecture, automating the development of optimised CPP implementations, as well as additional experimental evaluation to explore the scalability of the proposed approach.

Acknowledgements. The support of the Lee Family Scholarship, the EU Horizon 2020 Research and Innovation Programme under grant agreement number 671653 and the UK EPSRC (EP/L00058X/1, EP/L016796/1, EP/N031768/1 and EP/P010040/1) is gratefully acknowledged.

Table I

<table>
<thead>
<tr>
<th></th>
<th>Parallelism</th>
<th>Logic</th>
<th>Primary FF</th>
<th>DSP</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant-v1</td>
<td>[16,8,5,8]</td>
<td>133112</td>
<td>256505</td>
<td>1377</td>
<td>1846</td>
</tr>
<tr>
<td>Humanoid-v1</td>
<td>[48,4,8,2]</td>
<td>140258</td>
<td>269357</td>
<td>1368</td>
<td>1888</td>
</tr>
<tr>
<td>FPGA Total</td>
<td></td>
<td>262400</td>
<td>524800</td>
<td>1963</td>
<td>2567</td>
</tr>
</tbody>
</table>

A higher speed-up is achieved for Ant-v1 benchmark, which has a smaller problem size. However, it is difficult to draw the big picture of how the system scales based on just two data points. We plan to explore the scalability in future work.

V. CONCLUSION AND FUTURE WORK

In this work, we propose a novel hardware architecture, Customised Pearlmutter Propagation (CPP), for accelerating Trust Region Policy Gradient (TRPO) on FPGA. The design addresses the key computational bottleneck of TRPO, which is the Fisher-Vector Product (FVP) computation inside the Conjugate Gradient (CG) solver. The proposed approach is based on the Pearlmutter algorithm, which enables an efficient hardware design, circumventing the key obstacle - symbolic differentiation with change of variables.

The proposed system is evaluated using two MuJoCo robotic locomotion benchmarks. Experimental results show that the proposed solution running on Stratix-V FPGA achieved up to 20 times speed-up against Keras deep learning framework with Theano backend running on i7-5930K CPU.

Future work includes further fine tuning of the CPP architecture, automating the development of optimised CPP implementations, as well as additional experimental evaluation to explore the scalability of the proposed approach.

Acknowledgements. The support of the Lee Family Scholarship, the EU Horizon 2020 Research and Innovation Programme under grant agreement number 671653 and the UK EPSRC (EP/L00058X/1, EP/L016796/1, EP/N031768/1 and EP/P010040/1) is gratefully acknowledged.

Table II

<table>
<thead>
<tr>
<th>Model</th>
<th>CG (FVP) FPGA</th>
<th>CG (FVP) Keras</th>
<th>Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant-v1</td>
<td>0.158s</td>
<td>0.192s (0.182s)</td>
<td>3.976s (3.975s)</td>
</tr>
<tr>
<td>Humanoid-v1</td>
<td>0.823s</td>
<td>0.892s (0.852s)</td>
<td>12.016s (12.014s)</td>
</tr>
</tbody>
</table>

REFERENCES