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 π. 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 π 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 π_θ, where θ 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 α is the step size. This process leads to an improved π_θ 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 α 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 α based on Kullback-Leibler (KL) divergence [2]. KL is a statistical measure of the difference between two probability distributions. In TRPO, step size α is maximised provided that the KL between the old policy π_θ 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 framework, 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 $(\mathcal{S}, \mathcal{A}, P, r, \rho_0, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ the action space, $P : \mathcal{S}\times\mathcal{A}\times\mathcal{S} \rightarrow \mathbb{R}$ the transition probability distribution, $r : \mathcal{S} \rightarrow \mathbb{R}$ the reward function, $\rho_0 : \mathcal{S} \rightarrow \mathbb{R}$ the distribution of initial state s_0, and γ the discount factor. Let policy π_θ be a stochastic policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, which is a conditional distribution $\pi_\theta(a|s) = \mathbb{P}_\theta(\mathcal{A}_t = a|\mathcal{S}_t = s)$, where θ are the parameters.

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

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

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

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

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:

$$A_{\pi_\theta}(s, a) = Q_{\pi_\theta}(s, a) - V_{\pi_\theta}(s) \tag{3}$$

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

$$\begin{aligned} \max_{\theta} \quad & \mathbb{E}_{s\sim\rho_{\theta_{old}}, a\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s, a)\right] \\ \text{subject to} \quad & \mathbb{E}_{s\sim\rho_{\theta_{old}}}\left[D_{KL}(\pi_{\theta_{old}}(\cdot|s)||\pi_\theta(\cdot|s))\right] \leq \delta_{KL} \end{aligned} \tag{4}$$

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 allowed. KL is a measure of difference between two probability distributions P and Q:

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) log \frac{p(x)}{q(x)} dx \tag{5}$$

In each TRPO iteration, (4) is solved with 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 $\mathbf{z} = \mathbf{Ap}$. Matrix $\mathbf{A}$ is the Fisher-Information Matrix. Vector $\mathbf{z}$ 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 $\mathbf{A}$ is approximated as follows:

$$\mathbf{A} \approx \mathbf{H} = \frac{1}{N}\sum_{n=1}^{N} \frac{\partial^2}{\partial\theta_i \partial\theta_j} D_{KL}(\pi_{\theta_{old}}(\cdot|s_n)||\pi_\theta(\cdot|s_n)) \tag{6}$$

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

Algorithm 1 Conjugate Gradient Algorithm

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

procedure CONJUGATE GRADIENT($\mathbf{A}, \mathbf{b}$, MaxIter, Th)
  Initialise $\mathbf{p} = \mathbf{b}$, $\mathbf{r} = \mathbf{b}$, $\mathbf{x} = \mathbf{0}$, $\rho = \mathbf{r}^\intercal\mathbf{r}$, $iter = 0$
  while $\rho >$ Th **and** $iter <$ MaxIter **do**
    $\mathbf{z} \leftarrow \mathbf{Ap}$ ▷ Fisher-Vector Product (FVP)
    $v \leftarrow \mathbf{r}^\intercal\mathbf{r}/\mathbf{p}^\intercal\mathbf{z}$
    $\mathbf{x} \leftarrow \mathbf{x} + v\mathbf{p}$
    $\mathbf{r} \leftarrow \mathbf{r} - v\mathbf{z}$
    $\rho_{new} \leftarrow \mathbf{r}^\intercal\mathbf{r}$
    $\mathbf{p} \leftarrow \mathbf{r} + (\rho_{new}/\rho)\mathbf{p}$
    $\rho \leftarrow \rho_{new}$
    $iter \leftarrow iter + 1$
  end while
  return $\mathbf{x}$
end procedure

Equation (6) shows matrix $\mathbf{H}$ is the Hessian matrix of KL divergence with respect to θ, averaged over samples. Therefore, the dimension of $\mathbf{H}$ is the number of parameters in policy π_θ, which can be extremrly large. For instance, the π_θ for the Humanoid-v1 benchmark used in this work has 57634 paramaters, resulting in a Hessian with 3.3 billion numbers. Thus the explicit calculation and storage of $\mathbf{H}$ have to be avoided. The good news is that in the CG algorithm, there is no need to formulate $\mathbf{H}$ explicitly. All we need is the Fisher-Vector Product $\mathbf{z} = \mathbf{Ap} \approx \mathbf{Hp}$, 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]:

$$\mathbf{Hp} = \frac{1}{N}\sum_{n=1}^{N} \nabla_\theta \langle \nabla_\theta D_{KL_n} \cdot \mathbf{p} \rangle \tag{7}$$

Here, the KL divergence $D_{KL_n} = D_{KL}(\pi_{\theta_{old}}(\cdot|s_n)||\pi_\theta(\cdot|s_n))$ based on sample n is used as a loss function. Its gradient with respect to θ can be computed via standard back propagation. Then the dot product of the gradient and vector $\mathbf{p}$ 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 π_θ. 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 $\langle \nabla_\theta D_{KL}, \mathbf{p} \rangle$
4. Symbolic Differentiation $\frac{\partial}{\partial out_j} \langle \nabla_\theta D_{KL}, \mathbf{p} \rangle$
5. Symbolic Back Propagation, calculating $\nabla_\theta \langle \nabla_\theta D_{KL}, \mathbf{p} \rangle$

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 $\langle \nabla_\theta D_{KL}, \mathbf{p} \rangle$ with respect to each output value of the neural network (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. The gradient $\nabla_\theta D_{KL}$ calculated via back propagation and the subsequent dot product $\langle \nabla_\theta D_{KL}, \mathbf{p} \rangle$ are with respect to each parameter in the neural network θ_i. The dot product is a function of θ, $\mathbf{p}$, and sample s: $\langle \nabla_\theta D_{KL}, \mathbf{p} \rangle = f(\theta, \mathbf{p}, s)$. But in Step 4, we need to differentiate the dot product with respect to the final outputs of the neural network out_j, so that it can be back propagated in the next step:

$$\frac{\partial}{\partial out_j} \langle \nabla_\theta D_{KL}, \mathbf{p} \rangle = \frac{\partial}{\partial out_j} f(\theta, \mathbf{p}, s) \tag{8}$$

Note that out_j does not appear in the argument list of f. It is actually a function of sample s and network parameters θ: $out_j = out_j(\theta, s)$. Therefore, to differentiate the dot product with respect to out_j, we will need a change of variables.

In software it can be handled by the automatic symbolic math package without user effort, but in hardware it is generally infeasible for the circuit designer to manually derive all these equations for non-trivial problems. In hardware, it would be desirable to have a circuit that does the job to avoid manual calculation (section III.B), or to circumvent this obstacle (our proposed approach, section III.C-D) .

B. The Straightforward Approach

To carry out the change of variables in hardware, we need to make use of the chain rule of differentiation:

$$\frac{\partial \langle \nabla_\theta D_{KL}, \mathbf{p} \rangle}{\partial out_j} = \sum_i p_i \frac{\partial D_{KL}}{\partial \theta_i} \frac{\partial \theta_i}{\partial out_j} = \sum_i p_i \frac{\partial D_{KL}}{\partial \theta_i} \Big/ \frac{\partial out_j}{\partial \theta_i} \tag{9}$$

where p_i is the i^{th} item in vector $\mathbf{p}$, $\frac{\partial D_{KL}}{\partial \theta_i}$ is the i^{th} item in the gradient vector $\nabla_\theta D_{KL}$. $\frac{\partial out_j}{\partial \theta_i}$ can be calculated by back propagating an one-hot vector in which $out_j = 1$.

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 out_j in the output layer:
   3.1 Back propagate out_j, calculating $\frac{\partial out_j}{\partial \theta_i}$ in (9)
   3.2 Evaluate (9) by mult-add to obtain $\frac{\partial \langle \nabla_\theta D_{KL}, \mathbf{p} \rangle}{\partial out_j}$
4. Standard Back Propagation, calculating $\nabla_\theta \langle \nabla_\theta D_{KL}, \mathbf{p} \rangle$

Note that the number of items in equation (9) is the number of parameters in policy π_θ, 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:

$$\mathbf{Hv} = \lim_{r \to 0} \frac{\nabla_{\mathbf{w}}(\mathbf{w} + r\mathbf{v}) - \nabla_{\mathbf{w}}(\mathbf{w})}{r} = \frac{\partial}{\partial r} \nabla_{\mathbf{w}}(\mathbf{w} + r\mathbf{v}) \Big|_{r=0} \tag{10}$$

We define the Pearlmutter differential operator $\Re\{\cdot\}$:

$$\Re\{f(\mathbf{w})\} = \frac{\partial}{\partial r} f(\mathbf{w} + r\mathbf{v}) \Big|_{r=0} \tag{11}$$

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:

$$\begin{aligned} x_i &= \sum\nolimits_j w_{ji} y_j + b_i \\ y_i &= \sigma_i(x_i) \end{aligned} \tag{12}$$

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, and the derivative of E with respect to the output value y_i is $e_i = \partial E / \partial y_i$. The standard back propagation is then given as follows:

$$\begin{aligned} \partial E/\partial y_i &= e_i(y_i) + \sum\nolimits_j w_{ij} (\partial E/\partial x_j) \\ \partial E/\partial x_i &= \sigma_i'(x_i)\ (\partial E/\partial y_i) \\ \partial E/\partial w_{ij} &= y_i\ (\partial E/\partial x_j) \\ \partial E/\partial b_i &= \partial E/\partial x_i \end{aligned} \tag{13}$$

By applying the $\Re\{\cdot\}$ operator to the standard forward propagation, we get the Pearlmutter Forward Propagation:

$$\begin{aligned} \Re\{x_i\} &= \sum\nolimits_j (w_{ji} \Re\{y_j\} + v_{ji} y_j) + v_i \\ \Re\{y_i\} &= \Re\{x_i\} \sigma_i'(x_i) \end{aligned} \tag{14}$$

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 $\Re\{\cdot\}$ operator to standard back propagation:

$$\begin{aligned}
\Re\left\{\frac{\partial E}{\partial y_i}\right\} &= e_i'(y_i)\Re\{y_i\} + \sum_j \left[w_{ij}\Re\left\{\frac{\partial E}{\partial x_j}\right\} + v_{ij}\frac{\partial E}{\partial x_j}\right] \\
\Re\left\{\frac{\partial E}{\partial x_i}\right\} &= \sigma_i'(x_i)\Re\left\{\frac{\partial E}{\partial y_i}\right\} + \Re\{x_i\}\sigma_i''(x_i)\frac{\partial E}{\partial y_i} \\
\Re\left\{\frac{\partial E}{\partial w_{ij}}\right\} &= y_i\Re\left\{\frac{\partial E}{\partial x_j}\right\} + \Re\{y_i\}\frac{\partial E}{\partial x_j} \\
\Re\left\{\frac{\partial E}{\partial b_i}\right\} &= \Re\left\{\frac{\partial E}{\partial x_i}\right\}
\end{aligned} \tag{15}$$

With the Pearlmutter forward and back propagation, the elements in the Hessian vector product $\mathbf{Hv}$ are just $\Re\{\partial E/\partial w_{ij}\}$ and $\Re\{\partial E/\partial b_i\}$. 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 θ_π maps observation state s to the mean vector μ of a m-dimensional diagonal Gaussian distribution $\pi_\theta(\cdot|s) = N(\mu, \sigma)$. The standard deviation is a stand-alone set of parameters $\sigma = diag(\sigma_1, \cdots, \sigma_m)$. 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 loss function E is the KL divergence, given by:

$$E = D_{KL}(\pi_{\theta_{old}}(\cdot|s)||\pi_\theta(\cdot|s)) \tag{16}$$

$$= \sum_{j=1}^{m}\left[ln(\frac{\sigma_j}{\hat{\sigma_j}}) + \frac{(\mu_j - \hat{\mu_j})^2 + \hat{\sigma_j}^2}{2\sigma_j^2}\right] - \frac{m}{2} \tag{17}$$

where $\hat{\mu_j}$ and $\hat{\sigma_j}$ are constants whose values are taken from μ_j and σ_j, respectively.

Differentiating the equation above, it follows that the first order derivative of E with respect to μ_j and σ_j are zero:

$$\partial E/\partial \mu_j = 0 \qquad \partial E/\partial \sigma_j = 0 \tag{18}$$

Note that μ_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:

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

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:

$$\begin{aligned}
\Re\{\partial E/\partial y_i\} &= e_i'(y_i)\Re\{y_i\} + \textstyle\sum_j w_{ij}\Re\{\partial E/\partial x_j\} \\
\Re\{\partial E/\partial x_i\} &= \sigma_i'(x_i)\Re\{\partial E/\partial y_i\} \\
\Re\{\partial E/\partial w_{ij}\} &= y_i\Re\{\partial E/\partial x_j\} \\
\Re\{\partial E/\partial b_i\} &= \Re\{\partial E/\partial x_i\}
\end{aligned} \tag{19}$$

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

$$\Re\{\partial E/\partial y_i\} = \Re\{\partial E/\partial \mu_i\} = \Re\{y_i\}/\sigma_i^2 \tag{20}$$

For the natural logarithm of standard derivation σ, we have:

$$\Re\{\partial E/\partial logstd_i\} = 2v_{logstd_i} \tag{21}$$

where v_{logstd_i} is the item in $\mathbf{v}$ 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

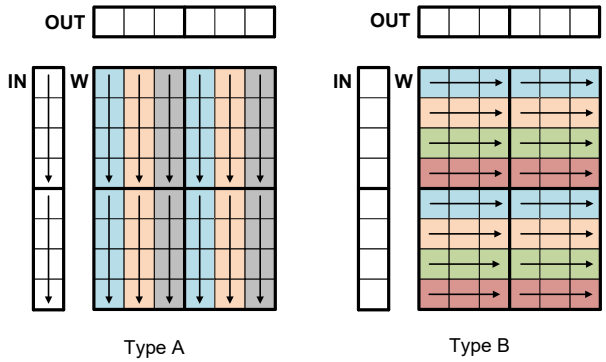

Fig. 1. Type A and Type B blocked matrix-vector multiplication. Both are 2×2 blocked for illustration purpose. In Type A, the loop over the input vector is the inner loop. In the first inner loop, the dot products of the input vector and the blue matrix columns are calculated, resulting in two output items (column 1 and 4). In the second inner loop, the dot products of the input and the orange matrix columns are calculated, and so on. In Type B, the loop over the input vector is the outer loop. In the first inner loop, the product of the 1st and 5th element in the input vector and their corresponding matrix items (blue) in all matrix columns are calculated, resulting in a partial result for the output. In the second inner loop, the product of the 2nd and 6th element in the input and their corresponding matrix items (orange) are calculated, resulting in another partial result for each output item. These partial results are accumulated, and final results come out in the last inner loop.

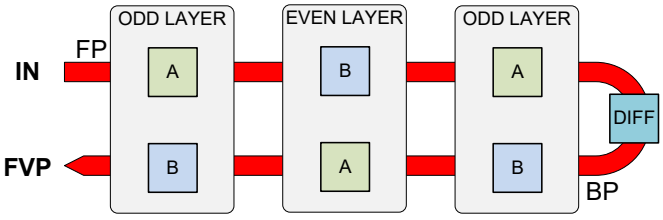

Fig. 2. Overall System Architecture. The Odd Layer has a Type A block for forward propagation and a Type B block for back propagation; the Even Layer has a Type B block for forward propagation and a Type A block for back propagation. There are buffers between adjacent layers which are omitted in this figure. The red U-shaped arrow indicates the data flow: the upper half is the Combined Forward Propagation (FP), the lower half is the Pearlmutter Back Propagation (BP). The DIFF module calculates $\Re\{\partial E/\partial y_i\}$ based on eq. (20), preparing for the Pearlmutter Back Propagation.

and back propagation for a multi-layer neural network. The first layer uses Type A block, as it can read new values from DRAM every cycle. The second layer's Type B block starts its fitst 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

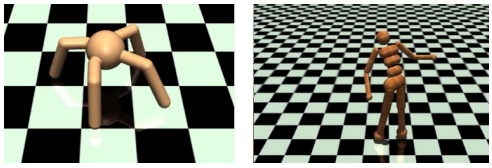

Fig. 3. Ant-v1 (left) and Humanoid-v1 (right) MuJoCo benchmarks [7].

layers. The DIFF block calculates $\Re\{\partial E/\partial y_i\}$ 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 $\mathcal{S}$ is 111-dimensional, the action space $\mathcal{A}$ 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 $\mathcal{S}$ is 376-dimensional, the action space $\mathcal{A}$ is 17-dimensional. We use a neural network sized at 376 (input) - 128 - 64 - 17 (output) for this problem, with $tanh()$ activation.

As mentioned bafore, 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.

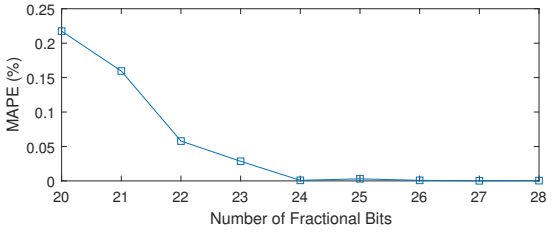

Fig. 4. Mean Absolute Percentage Error (MAPE) of the Conjugate Gradient result versus Number of Fractional Bits used in hardware. The result from FPGA is compared against that from Keras deep learning framework with Theano backend (double precision). Humanoid-v1 data set is used.

TABLE I
RESOURCE USAGE OF STRATIX-V 5SGSD8 FPGA

	Parallelism	Logic	Primary FF	DSP	BRAM
Ant-v1	[16,8,5,8]	133112	256505	1377	1846
Humanoid-v1	[48,4,8,2]	140238	269357	1368	1888
FPGA Total		262400	524800	1963	2567

For the CG solver, the maximum iterations $\mathrm{MaxIter} = 10$, and $\mathrm{CG}_{\mathrm{Damping}} = 0.1$. For TRPO, maximum KL divergence is $\delta_{KL} = 0.01$. These settings follow the TRPO paper [2].

B. Number Representation Optimisation

We use fixed-point numbers for computation. Fig. 4 shows the Mean Absolute Percentage Error (MAPE) of CG results vs. number of fractional bits, verified against double precision Keras software. The Humanoid-v1 data set is used. Judging from MAPE alone, it seems 24 fractional bits is optimal. But after taking account of other factors, we choose 23 fractional bits for our system. By analysing the value range, we find 4 integer bits are needed in forward propagation, and if we use 23 fractional bits, the total bit-width will be 27-bit. In the Stratix-V FPGA we used, a 27bit$\times$27bit multiplication can be carried out by one DSP, but for 28bit two DSPs are needed. Thus we sacrifice accuracy a little bit (0.028% MAPE) in exchange for a big reduction in resource usage, which enables higher parallelism. The same bit-width setting also works for Ant-v1, achieving 0.007% MAPE.

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 Kreas software with Theano backend.

TABLE II
PERFORMANCE COMPARISON

	Model	CG (FVP) FPGA	CG (FVP) Keras	Acc.
Ant-v1	0.158s	0.192s (0.182s)	3.976s (3.975s)	20.70
Humanoid-v1	0.823s	0.892s (0.852s)	12.016s (12.014s)	13.47

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.

REFERENCES

[1] 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.
[2] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust Region Policy Optimization," in *ICML*, 2015, pp. 1889–1897.
[3] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *ICML*, 2016, pp. 1329–1338.
[4] 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.
[5] B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural computation*, vol. 6, no. 1, pp. 147–160, 1994.
[6] A. Grama, *Introduction to parallel computing*. Pearson Education, 2003.
[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," *arXiv preprint arXiv:1606.01540*, 2016.