HEPPO: Hardware-Efficient Proximal Policy Optimization
A Universal Pipelined Architecture for Generalized Advantage Estimation

Hazem Taha and Ameer M. S. Abdelhadi
McMaster University
1280 Main St. W, Hamilton, Ontario L8S 4L8
{tahah,ameer}@mcmaster.ca

***Abstract*—This paper introduces HEPPO, an FPGA-based accelerator designed to optimize the Generalized Advantage Estimation (GAE) stage in Proximal Policy Optimization (PPO). Unlike previous approaches that focused on trajectory collection and actor-critic updates, HEPPO addresses GAE's computational demands with a parallel, pipelined architecture implemented on a single System-on-Chip (SoC). This design allows for the adaptation of various hardware accelerators tailored for different PPO phases. A key innovation is our strategic standardization technique, which combines dynamic reward standardization and block standardization for values, followed by 8-bit uniform quantization. This method stabilizes learning, enhances performance, and manages memory bottlenecks, achieving a 4x reduction in memory usage and a 1.5x increase in cumulative rewards. We propose a solution on a single SoC device with programmable logic and embedded processors, delivering throughput orders of magnitude higher than traditional CPU-GPU systems. Our single-chip solution minimizes communication latency and throughput bottlenecks, significantly boosting PPO training efficiency. Experimental results show a 30% increase in PPO speed and a substantial reduction in memory access time, underscoring HEPPO's potential for broad applicability in hardware-efficient reinforcement learning algorithms.**

I. Introduction

Reinforcement Learning (RL) is a subset of machine learning where agents learn best behaviors by interacting with the environment. RL agents do not receive correct input/output pairs like in supervised learning; they find strategies through trial and error, guided by rewards. This approach has been crucial in addressing intricate decision-making challenges in various fields such as robotics [1] and strategic games like chess and Go [2].

Proximal Policy Optimization (PPO) is a widely used reinforcement learning (RL) algorithm that optimizes policy directly through gradient ascent to maximize expected cumulative reward [3]. PPO enhances the stability of policy gradient methods by using a clipped objective function to ensure that policy updates are not excessively large, effectively addressing high gradient variance and instability. This approach eliminates the need for the computationally expensive second-order optimization step required by algorithms such as Trust Region Policy Optimization (TRPO), making PPO easier to implement and more computationally efficient [4]. By preventing large updates, PPO maintains robustness and improves training stability of TRPO [3]. At the core of PPO, it effectively balances the need for exploration and exploitation by preventing large policy updates, ensuring stable and efficient learning. This makes PPO a robust and practical choice for a wide range of RL tasks [3].

Algorithm 1: PPO Algorithm

Input : (1) initial policy parameters θ_0,
(2) initial value function parameters ϕ_0
Output : learned policy π_θ
for $k = 0, 1, 2, \ldots$ **do**
  Collect a set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$, based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \Big),$$

  // typically via stochastic gradient ascent with Adam [5]
  Fit the value function by regression on mean-squared error

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \Big(V_\phi(s_t) - \hat{R}_t \Big)^2,$$

  // typically via some gradient descent algorithm

In Algorithm 1, θ represents policy parameters, and ϕ represents value function parameters. The policy π_θ maps states to actions, while π_{θ_k} denotes the policy at the k-th iteration. Rewards-to-go $\hat{R}_t$ are the sum of future rewards from time step t, and advantage estimates $\hat{A}_t$ measure how much better an action is compared to the average action at a given state. The function $g(\epsilon, \hat{A}_t)$ clips the probability ratio to prevent large updates.

An essential component of PPO is the computation of advantage estimates. The advantage function, $A^\pi(s, a)$, measures how much better taking action a in state s compared to the average action taken in state s under policy π, namely,

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s), \tag{1}$$

where $Q^\pi(s, a)$ is the state-action value function, and $V^\pi(s)$ is the state value function.

To compute these advantage estimates effectively, Generalized Advantage Estimation (GAE) is used. GAE helps reduce the variance of the advantage estimates, leading to more stable and efficient policy updates [6].

A. Background on Generalized Advantage Estimation (GAE)

GAE addresses the variance-bias tradeoff in policy gradient methods for RL by using value functions to estimate the advantage function more accurately, at the cost of introducing some bias. The key idea is to use an exponentially-weighted

estimator of the advantage function, analogous to the TD(λ) method [6]. An estimation can be derived by utilizing the temporal-difference (TD) residual, δV_t,

$$\delta V_t = r_t + \gamma V(s_{t+1}) - V(s_t), \tag{2}$$

whereas the GAE is an exponentially-weighted average of k-step advantage estimators:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta V_{t+l}. \tag{3}$$

Alternatively, this can be computed sequentially, namely,

$$A_t^{GAE} = \delta_t + (\lambda\gamma) A_{t+1}^{GAE}. \tag{4}$$

GAE allows for direct computation of advantages, handling reward delays and noisy rewards more effectively, where Rewards-to-Go are defined as

$$\text{Rewards-to-Go} = V_t + \hat{A}_t^{GAE}. \tag{5}$$

GAE is used in PPO for policy updates as it provides low-variance and high-bias advantage estimates. This involves

1) Collecting trajectories using the current policy.
2) Computing the advantages and rewards-to-go for each state-action pair.
3) Using the computed advantages to update the policy using the PPO objective.

This combination allows PPO to leverage GAE's strengths, resulting in improved performance on complex RL tasks.

B. Challenges and Limitations in PPO Acceleration

Table I and Figure 1 highlight the time taken by different components of the PPO algorithm in both CPU-only and CPU-GPU systems. The profiling was conducted on a high-performance system with 32 Intel(R) Xeon(R) Silver 4216 CPU cores @ 2.10GHz and a Tesla V100-SXM2-32GB GPU, using the Humanoid environment by Gymnasium.

The data reveals that the environment run and the GAE computation phase consume a significant portion of the processing time (47% and 30% respectively in CPU-GPU systems). Knowing that environments are typically high-level code compiled to run on commodity CPUs and are independent of the RL algorithm, it's not feasible to build custom hardware that can accelerate different environments.

In our work, we focused on exploiting the PPO characteristics to accelerate the algorithm itself. Developing custom hardware for the GAE phase and potentially other heavy components of PPO and executing these computations on an SoC would reduce the communication overhead between different systems like CPU, GPU, and DRAM. Replacing DRAM with on-chip memory for these operations would further decrease latency and improve data throughput, leading to significant performance gains in PPO training.

C. Related Work

This section discusses key contributions in hardware acceleration for RL, aligning with our work on HEPPO by addressing the computational demands of RL algorithms through innovative hardware designs, quantization techniques, and memory management strategies.

TABLE I: Time Profiling of PPO Iteration over Different Systems (as percentages of total time)

Phase	Sub-Phase	CPU-GPU	CPU Only
Trajectory Collection	DNN Inference	9.92%	10.46%
	Environment Run	46.58%	60.71%
	CPU-GPU Communication	0.85%	NA
	Storing Trajectories	5.73%	4.75%
GAE	GAE Memory Fetch	5.00%	3.49%
	GAE Computation	24.79%	11.23%
	GAE Memory Write	0.17%	0.32%
Network Update	Loss Calculation	5.21%	6.10%
	Backpropagation	1.77%	2.95%

Krishnan et al. introduced an RL training paradigm using 8-bit quantized actors to accelerate data collection without compromising learning convergence, achieving a 1.5× to 5.41× speedup and reducing carbon emissions by 1.9× to 3.76× compared to full-precision training [7]. Yang et al. employed fixed-point data types and arithmetic units for both training and inference, demonstrating a training throughput of 25293.3 inferences per second (IPS), 2.7 times higher than a CPU-GPU platform, and an energy efficiency of 2638.0 IPS/W, 15.4 times more energy-efficient than a GPU [8]. These studies highlight the effectiveness of quantization techniques in enhancing RL training speed and energy efficiency.

Meng et al. (2020) targeted the inference and training phases of the PPO algorithm on a CPU-FPGA heterogeneous platform, achieving throughput improvements of 2.1×–30.5× over CPU-only implementations and 2×–27.5× over CPU-GPU implementations [9]. Specific benchmarks showed a 23.5% increase in throughput for Hopper and a 21.2% increase for Humanoid with data layout optimization. Load balancing optimization led to improvements ranging from 9.3% to 28.3% in overall running average throughput.

Weng et al.'s EnvPool addresses the bottleneck of slow environment execution in RL training systems using a C++ thread pool-based executor engine, achieving 1 million frames per second for Atari environments and 3 million frames per second for MuJoCo environments on a NVIDIA DGX-A100 with 256 CPU cores [10]. Dalton et al.'s CuLE platform leverages GPU parallelization to run thousands of Atari game environments simultaneously, achieving up to 155 million frames per hour [11]. Liang et al. introduced a GPU-accelerated RL simulator using NVIDIA Flex, achieving substantial improvements in training complex RL tasks and offering significant scaling benefits with multi-GPUs [12]. These works underscore the critical role of efficient environment simulation in enhancing RL training performance.

To the best of our knowledge, we are the first to specifically target the optimization of the critical GAE step in PPO, which

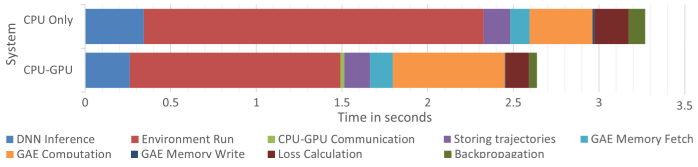

Fig. 1: Time Profiling of PPO Iteration over Different Systems

constitutes around 30% of the total processing time in CPU-GPU systems. This work addresses this gap by focusing on the computational demands of GAE, offering a significant contribution to the field of RL hardware acceleration.

D. Paper Contribution

Major contributions and innovations of this paper are

- Enabling the integration of multiple custom hardware components, memory, and CPU cores on a single system-on-chip (SoC) architecture, accommodating all phases of PPO from environment simulation to GAE computation. This reduces communication overhead and enhances data throughput and system performance.
- Introducing dynamic standardization for rewards and block standardization for values. This technique stabilizes learning, enhances training performance, and manages memory efficiently, reducing memory usage by 4x and increasing cumulative rewards by 1.5x.
- A parallel processing system that processes trajectories concurrently, employing a k-step lookahead approach for optimized advantage and rewards-to-go calculations. Our pipelined Processing Element (PE) can handle 300M elements per second, decimating the delay of GAE calculation and reducing PPO time by approximately 30%.
- A memory layout system that organizes rewards, values, advantages, and rewards-to-go on-chip for faster access. Using dual-ported Block RAM (BRAM) to implement a FILO storage mechanism, this system provides the required throughput each cycle, allowing overwriting of the same memory locations for efficient data handling.
- In-depth time profiling for the PPO algorithm revealing that GAE computation is a major contributor to processing time, accounting for 30% in CPU-GPU systems.

II. Algorithm Modification

We aim to achieve an optimally-reduced version of the PPO algorithm that can closely resemble the training behavior of the original algorithm, while allowing rescaling the input data to the GAE calculation phase. This guarantees that any computation done to the input data will be independent of the used environments and hyperparameters as all inputs are re-distributed evenly. To achieve our goal, several modifications have been proposed and investigated as follows.

A. Dynamic Standardization of Rewards

The motivation behind standardizing the rewards (and later values) is to have a consistent and predictable distribution in which we can perform quantization. Applying traditional standardization techniques has experimentally shown to cause training divergence. This is mainly because these methods independently alter the distribution of rewards within each training epoch, disrupting the relative differences in reward distributions between epochs and equalizing short-term and long-term rewards, misleading the training.

To solve this problem, a novel standardization technique has been developed and coined the name Dynamic Standardization. The idea is that at each training epoch, reward standardization shall be conducted while accounting for all previously attained rewards. As it will be computationally and memory intensive to store and reprocess all the rewards across training, a more efficient way is to store a running mean and running standard deviation that gets updated every epoch with the new reward.

To update the running mean with every new reward, we follow the equation

$$\text{RunningMean}_n = \text{RunningMean}_{n-1} + \frac{r_n - \text{RunningMean}_{n-1}}{n}, \quad (6)$$

where n is the total number of rewards processed so far, r_n is the n-th reward, and RunningMean_n is the running mean calculated up to the n-th reward.

As for the running standard deviation, inspired by Welford's algorithm [13] [14] for dynamically calculating variance over multiple iterations, the running variance for each new data point has been computed as follows.

1) Initialize M_0 and S_0 to 0.
2) For each new reward r_n

$$M_n = M_{n-1} + \frac{(r_n - M_{n-1})}{n}, \quad \text{and} \quad (7)$$

$$S_n = S_{n-1} + (r_n - M_{n-1}) \times (r_n - M_n). \quad (8)$$

3) The running standard deviation after n rewards is then

$$\text{RunningSTD}_n = \sqrt{\frac{S_n}{n}}, \quad (9)$$

where M_n is the running mean after n rewards and S_n is the cumulative value used for calculating variance.

B. Block Standardization of Values

Unlike rewards, the values are outputs of a trainable Neural Network (critic) that evolves differently over time and exhibits varying distributions. This observation is illustrated in Figure 2, which shows the distribution of values across a selected set of trajectories during training.

Dynamic standardization of values was unsuccessful as it affected the loss calculations. Instead, a more adaptable standardization method is required to handle these variations effectively while keeping a history of their original distribution to project them back in place. To address this, we propose a block standardization technique that quantizes values in batches. The steps involved in this process are as follows:

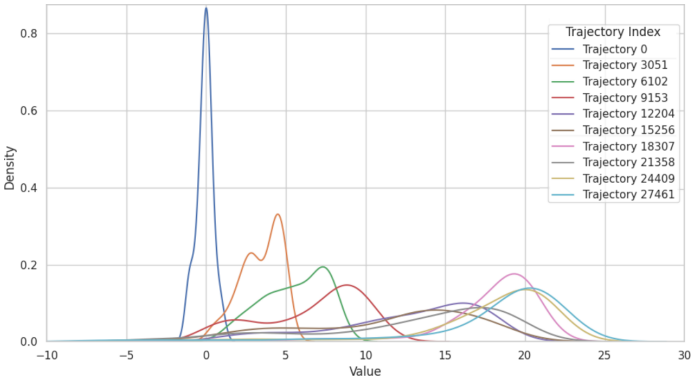

Fig. 2: Distribution of Value Across Collected Trajectories

1) **Batch Collection:** Collect a batch of values from multiple trajectories.
2) **Compute Statistics:** Calculate the mean (μ_v) and standard deviation (σ_v) for each batch.
3) **Standardization:** Scale values to have a mean of zero and a standard deviation of one by subtracting each the mean μ_v from each element in the block and then dividing by standard deviation σ_v.
4) **Uniform Quantization:** Quantize standardized values uniformly, storing them with μ_v and σ_v.
5) **Reconstruction:** De-quantize and convert values back to the original scale using the stored statistics.

This method leverages the similar distribution of trajectories collected at the same point in training, allowing for adaptive quantization based on the actual mean and standard deviation at that moment. The effectiveness of this method was validated through a series of experiments, demonstrating its robustness to shifts in training dynamics and its efficiency in utilizing memory bandwidth.

C. Quantization of Rewards and Values

Due to the memory bottlenecks discussed in Section IV, it is impractical to use 32-bit floating-point representation for each element in the Rewards and Values vectors. Instead, we adopt a quantization strategy tailored separately for rewards and values.

1) Quantization of Rewards: After applying dynamic standardization, rewards are centered around zero with a unit standard deviation. They are then uniformly quantized using n-bit codeword, mapping continuous values to discrete levels. During reconstruction, rewards are retrieved from memory, de-quantized, and used in their standardized form. Experimental testing showed that leaving the rewards in their standardized form enhances the cumulative rewards by around 50% as shown in section Section V.

2) Quantization of Values: Similar to the Quantization of Rewards, After applying block standardization, the values are uniformly quantized using n-bit codeword. During reconstruction, we also fetch and de-quantize the values. However, the main difference is that we have to do a final de-standardization step shifting the distribution back to its original form. This is done by multiplying the elements in v back by the stored standard deviation σ_v and then adding the mean μ_v.

III. HEPPO Architecture Details

The proposed architecture integrates the whole PPO pipeline in a single SoC, reducing latency and communication overhead compared to traditional CPU-GPU systems. Figure 3 illustrates the connections between the Processing System (PS), Programmable Logic (PL), and BRAM within the SoC.

A. Data Flow, Processing, and Efficiency

The PS access the BRAM for reading and writing via the AXI Interconnect, ensuring seamless data exchange with the custom logic in the PL. This integration keeps critical data on-chip, reducing the need to access external DRAM and enhancing throughput. Unlike traditional CPU-GPU systems which require frequent data transfers between the CPU, GPU, and DRAM, leading to higher latency and communication overhead. Detailed Processing Stages are as follows.

- *Data Preparation and Initiation:* Processed data is stored in BRAMs, and the PS communicates to the FPGA with an initiate signal.
- *Advantages and RTGs Calculation:* The FPGA fetches the data, performs de-quantization, calculates advantages and rewards-to-go (RTGs), writes back, and sends a completion signal back to the PS.
- *Actor-Critic Losses Calculation:* The PS retrieves the computed advantages and RTGs from the BRAMs and calculates actor-critic losses.
- *Back Propagation and Networks Update:* The PS sends the losses to the FPGA to perform backpropagation to update the neural networks. Updated network parameters are then used for subsequent iterations.

TABLE II: Decomposition of advantage estimates for different t values

t	$\hat{A}_t$		
T	$\hat{A}_T$	$=$	δ_T
$T-1$	$\hat{A}_{T-1}$	$=$	$C\delta_T + \delta_{T-1}$
		$=$	$C\hat{A}_T + \delta_{T-1}$
$T-2$	$\hat{A}_{T-2}$	$=$	$C^2\delta_T + C\delta_{T-1} + \delta_{T-2}$
		$=$	$C^2\hat{A}_T + C\delta_{T-1} + \delta_{T-2}$
$T-3$	$\hat{A}_{T-3}$	$=$	$C^3\delta_T + C^2\delta_{T-1} + C\delta_{T-2} + \delta_{T-3}$
		$=$	$C^2\hat{A}_{T-1} + C\delta_{T-2} + \delta_{T-3}$
		$=$	$C^3\hat{A}_T + C^2\delta_{T-1} + C\delta_{T-2} + \delta_{T-3}$

B. Advantage Estimate Decomposition and k-step Lookahead

In RL, the advantage estimate $A(t)$ is a critical component for policy improvement. Using Equation 4, we can decompose the advantage estimate calculation as shown in Table II.
where C is a constant defined as $\gamma \cdot \lambda$.

This decomposition shows how each advantage estimate depends on future values. An efficient way that developers use is to compute the estimates in revere order from $t = N$ to $t = 1$ to avoid recalculating the same terms multiple times, thus saving computational resources.

Single Cycle Implementation and Pipelining: The single-cycle GAE unit can be represented with potential pipelines highlighted in dashed green, as shown in Figure 4. In this implementation, various stages of the computation are pipelined to improve efficiency. However, when we attempt to pipeline the feedback loop (highlighted in red), it introduces bubbles into the system. These bubbles represent idle states where

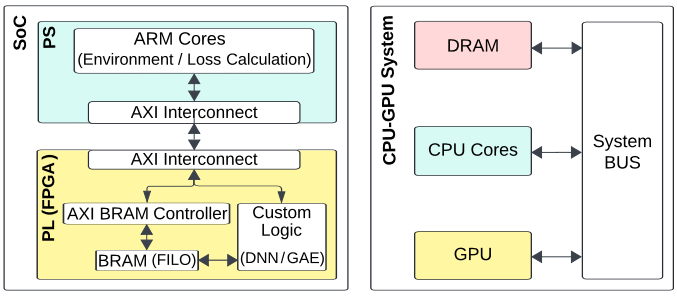

Fig. 3: Comparison of SoC Architecture and Traditional CPU-GPU System

the pipeline must wait for data from previous stages, severely reducing efficiency.

k-step Lookahead Solution: The k-step lookahead method addresses this inefficiency by introducing registers (delays) in the feedback loop. This approach can be explained through the modified advantage estimate calculations for 2-step lookahead

$$\hat{A}_t = C^2 \hat{A}_{t+2} + C\delta_{t+1} + \delta_t, \tag{10}$$

and for 3-step lookahead,

$$\hat{A}_t = C^3 \hat{A}_{t+3} + C^2\delta_{t+2} + C\delta_{t+1} + \delta_t. \tag{11}$$

Figure 4 illustrates the implementation of a 3-step lookahead. As shown, three registers are added to the feedback loop to apply 3-step lookahead transformation (highlighted in yellow). The added registers on the feedback loop can be moved inside the multiplier, enabling embedding pipelined multiplier through DSP blocks. This solution enables a fully pipelined processing, eliminating compute bubbles in the feedback loop. The following is a general equation for the k-step lookahead,

$$\hat{A}_t = C^k \hat{A}_{t+k} + \sum_{i=0}^{k-1} C^{(k-1)-i}\delta_{t+i},$$

facilitating the incorporation of additional registers on the feedback loop, and a more profound pipelining of the multiplier. Although the 2-step lookahead transformation is satisfactory for enabling our system to operate at the highest frequency and attain the peak performance, alternative systems, notably those with wider data formats, might necessitate more pipeline stages to operate at the maximum achievable frequency.

C. System Architectue

Figure 5 (a) shows the micro-architecture of HEPPO, which consists of Rewards Loaders (ReLs), Values Loaders (VaLs), and compute Processing Elements (PEs) forming a one-dimensional systolic array with N rows.

Parallelization: Rows in the systolic array run concurrently and independently, each processing distinct vectors from different agents assigned by a round-robin fashion. When one row finishes, it gets a new set of vectors. This parallel architecture enhances HEPPO's efficiency and scalability. While the BRAM stack memory enables substantial data transfers, a crossbar network ensures robust connections between ReLs, VaLs, and PEs to the BRAM stack memory.

Data flow: Each ReL reads element R_i from the rewards vector and sends it with index i and the signal Done to VaL. VaL fetches the corresponding i-th value V_i and sends R_i, V_i, i, and Done to the PEs. The PE calculates the Advantage Estimate (Adv) and Rewards-to-Go (RTG) and writes them back to the main memory at index i.

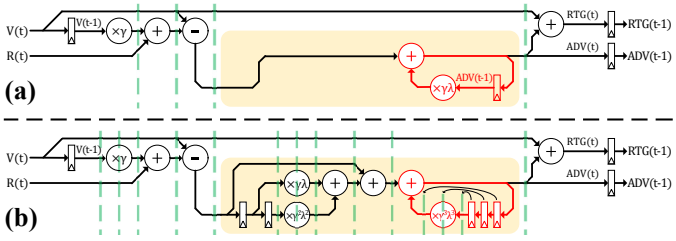

Fig. 4: Pipelining of the GAE unit: (a) possible pipelines (in dashed-green) are limited due to the logic loop (in red); (b) A 3-step Look-ahead (highlighted in orange) is applied to enable pipelining within the logic loop.

IV. Data Layout

To enhance the efficiency of the Proximal Policy Optimization (PPO) algorithm, we propose a memory layout that organizes rewards, values, advantages, and rewards-to-go on-chip for faster access. This layout groups data from different trajectories with the same timestep into memory blocks, enabling simultaneous retrieval and processing. Additionally, it employs a First-In-Last-Out (FILO) storage mechanism to align with the backward iteration required for GAE calculations. This section details the memory organization, access patterns, and bandwidth considerations.

A. Memory Bandwidth Bottleneck

In a typical large RL setup with 64 trajectories and 1024 timesteps, both rewards and values are stored in 32-bit floating-point format. The required memory per timestep for 64 trajectories (128 elements) is 512 bytes.

For parallel processing, these 512 bytes need to be fetched from memory per clock cycle. Assuming a typical DDR4 3200 bandwidth of 25 GB/s and a clock frequency of 300 MHz, the available bandwidth per cycle is calculated as Bandwidth per cycle $= \frac{25\times10^9 \text{ bytes/s}}{300\times10^6 \text{ cycles/s}} = 83.3$ bytes/cycle.

This results in a shortfall of 428.7 bytes per cycle. Clearly, DRAM cannot supply enough data to sustain 64 parallel processing elements (PEs), severely limiting parallelization. To overcome this bottleneck, we store data in on-chip dual-port Block RAM (BRAM), which meets the required 512 bytes per cycle, ensuring high-throughput parallel processing.

1) Memory Block Layout: The memory layout is structured as 2D arrays, with dimensions representing timesteps and trajectories. Each memory block stores specific data (rewards, values, advantages, or rewards-to-go) indexed by timestep and trajectory. This layout enables parallel processing of different trajectories using the same fetched block which improves efficiency and lower memory accesses.

Figure 6 illustrates the dual-port Block RAM (BRAM) stack memory system, consisting of: $\mathbf{BRAM_0}$, which stores rewards $R_{i,j}$, and $\mathbf{BRAM_1}$, which stores values $V_{i,j}$, both from different trajectories i at the same timestep j.

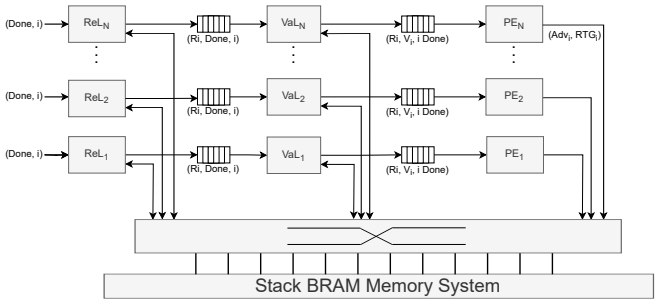

Fig. 5: The HEPPO architecture consisting of Rewards Loaders (ReLs), Values Loaders (VaLs), PEs, system crossbar, and BRAM stack memory

Algorithm 2: PPO/GAE Memory Layout and Processing

```
Initialization
    Initialize memory blocks RMB, VMB, AMB, RTGMB
Data Insertion
    foreach timestep t do
        foreach trajectory i do
            Push reward[i][t] into RMB[t][i]
            Push value[i][t] into VMB[t][i]
GAE Calculation and In-Place Update
    foreach timestep t, backward do
        foreach trajectory i do
            Retrieve reward from RMB[t][i]
            Retrieve value from VMB[t][i]
            Compute advantage and reward-to-go
            Store advantage in AMB[t + 1][i]
            Store reward-to-go in RTGMB[t + 1][i]
```

2) FILO Storage Mechanism: The FILO storage mechanism uses a stack-based structure to store rewards and values:

- *Push Operation*: Rewards and values are pushed onto the stack at each timestep.
- *Pop Operation*: Rewards and values are popped from the stack during GAE calculation, starting from the last timestep and iterating backward.

3) In-Place Updates and Dual-Port Memory: The system uses dual-port memory for simultaneous read and write operations, enabling in-place updates where advantages and rewards-to-go can overwrite the original rewards and values reducing memory usage by half.

Algorithm 2 is implemented to manage the FILO stack structure in BRAM, ensuring efficient data access patterns compatible with the hardware architecture.

This proposed design ensures fast data retrieval and processing that allows continuous feeding of data to the PEs, keeping them always busy.

V. Experimental Results

The results of our work are divided across multiple axes. In this section, we will discuss each axis and how they can affect the overall acceleration of the PPO algorithm.

A. Dynamic Rewards Standardization

It's important to note that, in most PPO implementations, the final calculated advantage vector is standardized to stabilize gradient updates and ensure smoother and more consistent training. This practice has become widely adopted due to its positive impact on training dynamics, as highlighted in various implementations and community discussions [15], [16].

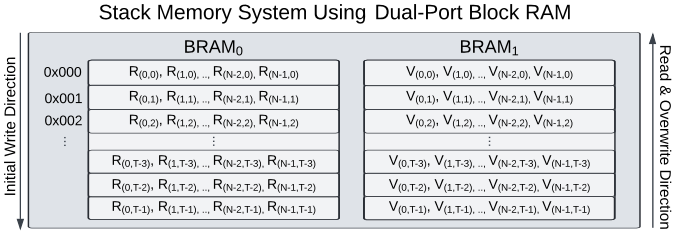

Fig. 6: Dual-Port Block RAM Stack Memory System

Figure 7 shows training outcomes in the Humanoid environment (Gymnasium toolkit). Our PPO version (with and without standardized advantages) achieved over 1.5x increase in cumulative rewards compared to the original PPO, continuing to improve after the original PPO plateaued. This improvement was consistent across MuJoCo and Atari environments, confirming our modification benefits both hardware and training.

B. Quantization of Rewards and Values

Optimal Quantization Size: Detailed investigation, shown in Figure 8 and Figure 9 shows that quantizing using 5 and 7 bits performed the poorest followed by 3, 4 which are near to the baseline (PPO + DS). Finally, quantizing with 6, 8 to 10 performed equally higher than the baseline. The reason why using 5 and 7 bits performed worse than 3 and 4 in some of the trials and better in others is most likely due to the inherent variance of the policy gradient algorithm and the probabilistic nature of RL. To avoid this unstable region, it was concluded through all trials that 8 bits and above can be seen as a threshold for a stable uniform quantization that archives high performance.

C. Summary of Rewards and Values Quantization Approaches

To summarize the effects of various quantization approaches on PPO performance, we conducted several experiments with their configurations shown in Table III.

- **Experiment 1**: Baseline PPO without quantization.
- **Experiment 2**: Dynamic standardization of rewards.
- **Experiment 3**: Both rewards and values are standardized and uniformly and quantized by block to 8-bit codewords.
- **Experiment 4**: Both rewards and values are standardized and uniformly and quantized by block, with rewards kept in standardized form throughout computations.
- **Experiment 5**: Dynamic standardization applied to rewards and block approach to values.

These experiments highlight the importance of dynamic standardization and appropriate quantization techniques in

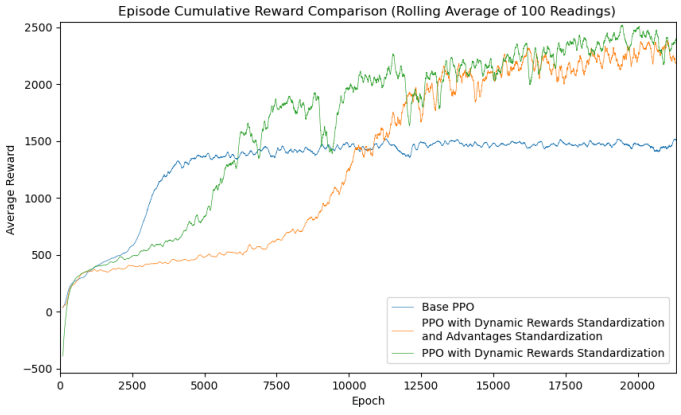

Fig. 7: Comparative Analysis of Cumulative Rewards Between Original PPO and Modified PPO with Dynamic Standardization

TABLE III: Overview of Experiment Attributes

Experiment Index	Standardization				Uniform Quant.	
	Rewards			Values		
	Dynamic Std.	Block Std./De-Std.	Block Std./No De-Std.	Block Std./De-Std.	Rewards	Values
1						
2	✓					
3		✓		✓	✓	✓
4			✓	✓	✓	✓
5	✓			✓	✓	✓

improving PPO training efficiency. Experiment 4 performance was poor, indicating that simply keeping rewards standardized does not enhance performance but keeping them in the dynamically standardized form does. Experiment 5, combining dynamic standardization for rewards and Block Quantization for values, showed the best performance, emphasizing the significance of the way rewards are standardized.

Figure 10 provides a comprehensive comparison for different PPO implementations, illustrating the performance impact of different quantization strategies and demonstrating how dynamic standardization and adaptive quantization methods can optimize PPO performance.

D. Hardware Implementation of HEPPO

A parameterized Verilog model of HEPPO's proposed pipelined architecture, as described in Subsection III-C, has been developed with a data width of 32 bits (after fetching and de-quantizing the elements). The AMD-Xilinx Zynq® UltraScale+™ MPSoC ZCU106 Evaluation Kit was chosen to host our implementation. This device integrates a quad-core Arm® Cortex™-A53 processing system (PS) and a dual-core Arm Cortex-R5 real-time processor, providing the necessary computing for running the environment. The FPGA fabric within the Programmable Logic (PL) provides extensive resources for custom logic implementation, including neural network inference and GAE computation.

For the DNN inference within the PL, we adapt the systolic array implementation introduced by Meng et al. (2020) [9]. Their design achieves a clock frequency of 285 MHz, whereas our overall system is designed to run at 300 MHz. As all design subsystems operate sequentially (processing does not overlap in time), it's advantageous to enable each subsystem to run at its highest frequency. While this creates multiple

Fig. 8: Uniform Quantization (3-6 bits) of Rewards

TABLE IV: Resource Utilization for a 2-step lookahead system

Resource	Total Usage (64 PEs)	Available	Utilization (%)
LUTs	12864	274080	4.69
FFs	54336	548160	9.91
DSPs	768	2520	30.48

clock domains, data synchronization is not required because all subsystems operate sequentially and communicate through BRAMs. However, control signals across domains, such as those indicating that processing has ended and data is ready, still need to be synchronized.

1) Area Utilization: Figure 11 illustrates the resource utilization percentages for 1-step, 2-step, and 3-step lookahead implementations per Processing Element (PE). As seen, there is a quadratic increase in resource usage with each increase in n. This figure highlights how the increase in lookahead steps (n) impacts the utilization of various resources (LUTs, FFs, and DSPs), demonstrating a clear quadratic trend. Based on our implementation, we found that $n > 1$ allows the system to operate at a maximum frequency of 300 MHz. This is due to the intensive pipelining and absence of pipeline stalls. Hence, a single PE is estimated to handle 300 million elements per second with the continuous data flow supported by the efficient design of the FILO BRAM memory system which ensures the required throughput every cycle. This is in contrast to a normal CPU-GPU system that suffers from DRAM memory access latency, buffering, and scheduling, all of which are a great bottleneck.

We choose to work with 2-step lookahead. The resource utilization in Table IV is estimated for 64 PEs based on

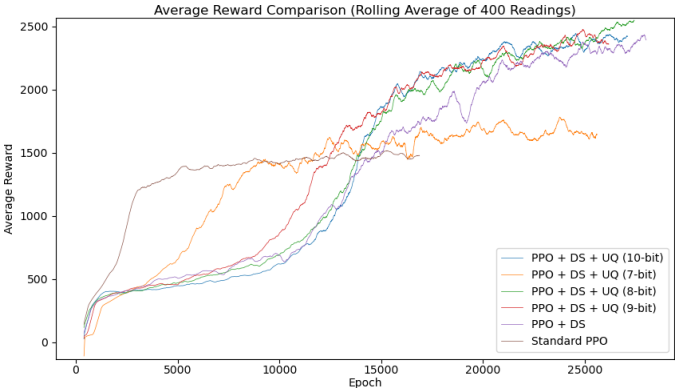

Fig. 9: Uniform Quantization (7-10 bits) of Rewards

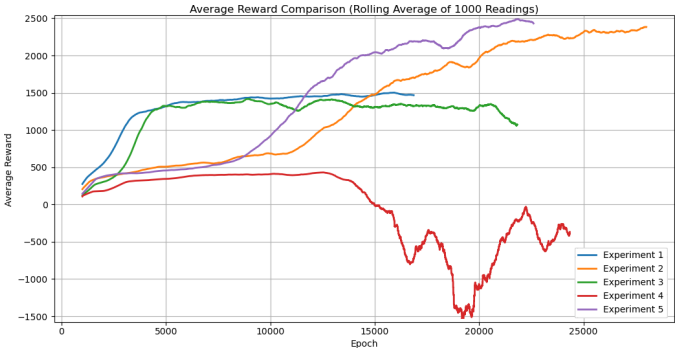

Fig. 10: Average Reward Comparison (Rolling Average of 1000 Readings). Refer to Table III for experiment details.

our single PE implementation. It shows that the ZCU106 Evaluation Kit can comfortably accommodate our design. The utilization percentages for LUTs, FFs, and DSPs are well within the available resources, with the most significant utilization being DSPs at 17.7%. This efficient usage ensures that the system can run at the desired frequency and handle the required throughput without encountering resource constraints.

2) Memory Utilization Requirements: For 64 trajectories and 1024 timesteps, with rewards and values overwritten by advantages and rewards-to-go, the memory required is 128 bytes per timestep, totaling 128 KB for 1024 timesteps.

BRAM Utilization: Each BRAM provides 36 Kb of storage, so storing 128 KB requires approximately 29 BRAM blocks (around 9% utilization).

Bandwidth Requirement: We read 64 rewards and 64 values (128 bytes) and write 64 advantages and 64 rewards-to-go (128 bytes) per clock cycle, requiring a total bandwidth of 256 bytes/cycle.

Each dual-port BRAM handles 4 bytes per port, per cycle. To meet the 256 bytes/cycle requirement, 57 BRAM ports are needed. Since each dual-port BRAM has 2 ports, this translates to 32 BRAM blocks (10% utilization) required to support both memory storage and bandwidth needs, ensuring efficient parallel processing.

3) System Estimated Speedup: We conducted a test on a standard GAE implementation [17] on a CPU-GPU system comprising 32 cores each is Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz, and a Tesla V100-SXM2-32GB GPU, it was concluded that this setup can handle $\approx$ 9000 elements per second. This is interpreted by the nature of this phase which processes trajectories of unequal sizes in reverse, this is traditionally achieved by iterating over one trajectory at a time not in batch form. However, in our implementation, we process a batch of 64 trajectories at a time in custom hardware made specifically for accelerating this phase. Hence, our system can theoretically handle around 2 million times faster than a traditional implementation, significantly reducing the time taken at the GAE stage and increasing the PPO speed by around 30%.

In addition, having our FILO memory on-chip with the CPU cores as well as the FPGA greatly reduces the memory access time of storing and fetching the trajectories which account for around 11.73% of the PPO time.

Finally, our proposed solution opens the door for a full PPO system implementation on-device, and it can adapt to any cutting-edge implementation of DNN in the PL. For this research paper, we adapt the systolic array implementation by Meng et al. (2020), leveraging the FPGA's capabilities to handle high-throughput neural network inference, backpropagation, and GAE computations. Their work is claimed to have achieved substantial performance improvements, ranging from 2.1× to 30.5× when compared to state-of-the-art CPU implementations and 2× to 27.5× when compared to CPU-GPU implementations.

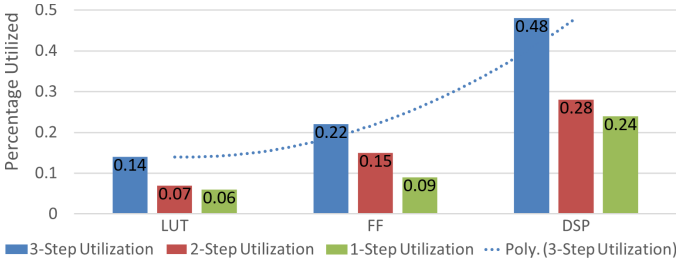

Fig. 11: Resource Utilization Percentage for n-Step Lookahead PE

VI. Conclusion and Future Work

We introduced HEPPO (Hardware-Efficient Proximal Policy Optimization), an FPGA-based implementation designed to accelerate the GAE stage of the PPO algorithm. HEPPO utilizes dynamic reward standardization and 8-bit uniform quantization, reducing memory usage by 4x and increasing cumulative rewards by 50%.

Our innovative memory layout system, using FILO storage and dual-port memory, efficiently handles rewards, values, and advantages, ensuring high throughput and compact data management. The ultra-pipelined Process Element (PE) unit, operating at 300 MHz, greatly enhances throughput and efficiency, outperforming conventional CPU-GPU systems and improving PPO training efficiency by an estimated 30%.

HEPPO leverages AMD-Xilinx Zynq UltraScale+ MPSoC's capabilities, integrating environment simulation, neural network inference, backpropagation, and GAE computation on a single SoC. This minimizes communication latency and optimizes data handling which originally accounted for around 11% of the training.

Further incremental optimizations of HEPPO's SoC are possible. Overclocking techniques [18] and bit-serial computation [19] in FPGAs can be employed to accelerate overall processing. To mitigate power consumption, multiple clock domains can be implemented for the ARM cores, the DNN, and the GAE calculations. High-performance clock-domain crossing (CDC) FIFOs can facilitate faster data transfers [20]–[24]. Additionally, hardware-efficient data compression methods, optimized for deep learning workloads, can be leveraged to minimize data transfers [25]–[27].

Future work should focus on optimizing custom hardware for other phases of the PPO algorithm, particularly in accelerating environment simulation, which consumes 47% of the training time. Investigating techniques for dynamic High-Level Synthesis of environments on FPGA and implementing loss calculation on FPGA could eliminate the need for CPU cores, significantly boosting the computational efficiency of the algorithm.

Acknowledgements

Many thanks to the anonymous reviewers for their valuable feedback and suggestions. This research would not have been possible without access to design tools and libraries provided by the Canadian Microelectronics Corporation (CMC) and access to the Compute Canada Database (CCDB) through the Digital Research Alliance of Canada. This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant program.

REFERENCES

[1] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, "Deep Reinforcement Learning for Robotic Manipulation—The State of the Art," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 319–326, 2018.

[2] D. Silver *et al.*, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv preprint arXiv:1712.01815*, 2017. [Online]. Available: http://arxiv.org/abs/1712.01815

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[4] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust Region Policy Optimization," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015, pp. 1889–1897.

[5] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, May 7-9 2015, conference Track Proceedings. [Online]. Available: http://arxiv.org/abs/1412.6980

[6] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," *arXiv preprint arXiv:1506.02438*, 2015. [Online]. Available: https://doi.org/10.48550/arXiv.1506.02438

[7] S. Krishnan, S. Chitlangia, M. Lam, Z. Wan, A. Faust, and V. J. Reddi, "QuaRL: Quantization for Fast and Environmentally Sustainable Reinforcement Learning," *arXiv preprint arXiv:1910.01055*, 2022. [Online]. Available: http://arxiv.org/abs/1910.01055

[8] J. Yang, S. Hong, and J.-Y. Kim, "FIXAR: A Fixed-Point Deep Reinforcement Learning Platform with Quantization-Aware Training and Adaptive Parallelism," in *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 259–264.

[9] Y. Meng, S. Kuppannagari, and V. Prasanna, "Accelerating Proximal Policy Optimization on CPU-FPGA Heterogeneous Platforms," in *Proceedings of the IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 19–27.

[10] J. Weng *et al.*, "EnvPool: A Highly Parallel Reinforcement Learning Environment Execution Engine," in *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS)*, New Orleans, LA, USA, 2022.

[11] S. Dalton and I. Frosio, "Accelerating Reinforcement Learning through GPU Atari Emulation," in *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS)*, Vancouver, BC, Canada, 2020.

[12] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox, "GPU-Accelerated Robotic Simulation for Distributed Reinforcement Learning," in *Proceedings of the 2nd Conference on Robot Learning (CoRL)*, Zürich, Switzerland, 2018.

[13] B. P. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962. [Online]. Available: https://www.jstor.org/stable/1266577

[14] D. E. Knuth, *The Art of Computer Programming, volume 2: Seminumerical Algorithms, 3rd edn.* Boston: Addison-Wesley, 1998.

[15] Z. Yang. (2021) *Justifying Advantage Normalization for PPO* [Online]. Available: https://github.com/DLR-RM/stable-baselines3/issues/485. [Accessed: 08 April 2024].

[16] mbcel. (2018) *Understanding Normalization of Advantage Function in PPO* [Online]. Available: https://github.com/openai/baselines/issues/544. [Accessed: 08 April 2024].

[17] E. Y. Yu. (2023) *Coding PPO from Scratch with PyTorch (Part 4/4)* [Online]. Available: https://medium.com/@z4xia/4e21f4a63e5c. [Accessed: 08 April 2024].

[18] A. Brant *et al.*, "Safe Overclocking of Tightly Coupled CGRAs and Processor Arrays using Razor," in *Proceedings of the IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013, pp. 37–44.

[19] A. M. Abdelhadi and L. Shannon, "Revisiting Deep Learning Parallelism: Fine-Grained Inference Engine Utilizing Online Arithmetic," in *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 383–386.

[20] A. M. S. Abdelhadi and H. Li, "Enabling Mixed-Timing NoCs for FPGAs: Reconfigurable Synthesizable Synchronization FIFOs," in *Proceedings of the 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 312–318.

[21] A. M. S. Abdelhadi and H. Li, "Reconfigurable Synthesizable Synchronization FIFOs," in *Proceedings of the IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 272–272.

[22] A. M. Abdelhadi, "Synthesizable Synchronization FIFOs Utilizing the Asynchronous Pulse-Based Handshake Protocol," in *Proceedings of the IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2020, pp. 1–7.

[23] A. M. Abdelhadi, "High-Throughput Synthesizable Synchronization FIFOs for Mixed-Timing NoCs," in *Proceedings of the 13th International Workshop on Network on Chip Architectures (NoCArc)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2020, pp. 1–6.

[24] A. M. S. Abdelhadi and M. R. Greenstreet, "Interleaved Architectures for High-Throughput Synthesizable Synchronization FIFOs," in *Proceedings of the 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2017, pp. 41–48.

[25] A. D. Lascorz *et al.*, "Atalanta: A Bit is Worth a "Thousand" Tensor Values," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA, 2024, p. 85–102.

[26] A. H. Zadeh *et al.*, "Mokey: Enabling Narrow Fixed-Point Inference for Out-of-the-Box Floating-Point Transformer Models," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, New York, NY, USA, 2022, p. 888–901.

[27] I. Edo Vivancos *et al.*, "Boveda: Building an On-Chip Deep Learning Memory Hierarchy Brick by Brick," in *Proceedings of Machine Learning and Systems (MLSys)*, vol. 3, 2021, pp. 1–20. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2021/file/12a304a31e42dfefa21c82431e849124-Paper.pdf