FA3C: FPGA-Accelerated Deep Reinforcement Learning

Hyungmin Cho
Dept. of Computer Engineering
Hongik University
Seoul 04066, Korea
hcho@hongik.ac.kr

Pyeongseok Oh
Dept. of Computer Science and Engineering
Seoul National University
Seoul 08826, Korea
pyeongseok@aces.snu.ac.kr

Jiyoung Park
Dept. of Computer Science and Engineering
Seoul National University
Seoul 08826, Korea
jiyoung@aces.snu.ac.kr

Wookeun Jung
Dept. of Computer Science and Engineering
Seoul National University
Seoul 08826, Korea
wookeun@aces.snu.ac.kr

Jaejin Lee
Dept. of Computer Science and Engineering
Seoul National University
Seoul 08826, Korea
jaejin@snu.ac.kr

Abstract

Deep Reinforcement Learning (Deep RL) is applied to many areas where an agent learns how to interact with the environment to achieve a certain goal, such as video game plays and robot controls. Deep RL exploits a DNN to eliminate the need for handcrafted feature engineering that requires prior domain knowledge. The Asynchronous Advantage Actor-Critic (A3C) is one of the state-of-the-art Deep RL methods. In this paper, we present an FPGA-based A3C Deep RL platform, called FA3C. Traditionally, FPGA-based DNN accelerators have mainly focused on inference only by exploiting fixed-point arithmetic. Our platform targets both inference and training using single-precision floating-point arithmetic. We demonstrate the performance and energy efficiency of FA3C using multiple A3C agents that learn the control policies of six Atari 2600 games. Its performance is better than a high-end GPU-based platform (NVIDIA Tesla P100). FA3C achieves 27.9% better performance than that of a state-of-the-art GPU-based implementation. Moreover, the energy efficiency of FA3C is 1.62× better than that of the GPU-based implementation.

CCS Concepts • **Computer systems organization** → **Neural networks**; • **Computing methodologies** → *Multi-agent reinforcement learning*.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304058

Keywords reinforcement learning; deep neural networks; FPGA

ACM Reference Format:
Hyungmin Cho, Pyeongseok Oh, Jiyoung Park, Wookeun Jung, and Jaejin Lee. 2019. FA3C: FPGA-Accelerated Deep Reinforcement Learning . In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304058

1 Introduction

Reinforcement Learning (RL) is a class of machine learning. It optimizes the control of an agent (action policy) that interacts with the environment[28]. Deep Reinforcement Learning (*Deep RL*) eliminates the need for handcrafted feature engineering in RL and enables an end-to-end learning system. Using Deep Neural Networks (DNNs), Deep RL directly learns an action policy from high-dimensional sensory inputs.

Adoption of DNNs for RL means that the DNN parameters (*e.g.*, weights) should be trained using a large number of training samples similar to other DNN applications, such as convolutional neural networks and recurrent neural networks. However, unlike supervised learning, Deep RL training is based on the datasets generated while the training itself is happening. Because of this, performance improvement by exploiting massively-parallel computing resources in a GPU for a large number of training data samples may not be possible during Deep RL training.

The Asynchronous Advantage Actor-Critic (A3C) algorithm[16] and its variants[12, 29] are considered as state-of-the-art Deep RL algorithms. Moreover, it has been shown that A3C-based algorithms can be successfully used for continuous control domains[16] and transfer learning[8, 25].

However, a major hurdle of utilizing the A3C Deep RL algorithm is that its training process is not easily accelerated by GPUs. As we will discuss in Section 2.2, A3C training is performed with small computation batches resulting in low GPU utilization.

In this paper, we propose an FPGA-based A3C Deep RL platform, called FA3C. FPGA-based platforms typically deliver higher energy efficiency than CPU-only or GPU-based platforms. For example, Intel introduces a new platform that combines a server-class CPU and an FPGA into a single package to achieve higher energy efficiency at the data center scale[26]. In addition, Amazon has launched a new compute cloud service, EC2 F1 that is a compute instance with FPGAs to create custom hardware accelerations for applications[3]. However, existing FPGA-based DNN accelerators have focused on inference only[20].

As we will discuss in Section 3, an FPGA-based platform has several advantages over GPU-based platforms specifically for A3C: high energy efficiency, low execution latency even with frequent kernel launches, and customizable memory subsystems to handle limited off-chip data traffic. We demonstrate an efficient A3C Deep RL platform based on an FPGA. FA3C targets not only higher performance-per-Watt but also higher performance than GPU-based platforms. FA3C performs both inference *and* training on the FPGA. We present the design and implementation of our FPGA-based platform architecture that supports various types of computations in A3C DNN inference and training.

The design principles of FA3C are summarized as follows:

1. **Simple and generic PEs for all types of DNN layers.** In the training process, there are data dependences between adjacent layers. Thus, designing processing elements (PEs) for a specific layer or computation type would make them mostly idle in the training process. Pipelining is not an attractive solution because the batch size is small in A3C. Instead, a PE is designed to be utilized by multiple different types of computations. Unlike previous approaches, we propose a simple and generic processing element architecture that consists of a multiplier and an accumulator that support single-precision floating-point arithmetic. A compute unit (CU) in FA3C contains multiple such PEs. The PEs are utilized by different types of computations throughout the DNN layers in A3C.
2. **Two separate CUs for inference and training.** Although the FPGA off-chip DRAM bandwidth is a limiting factor of the training performance, not every layer of the DNN is bounded by the off-chip DRAM bandwidth. Since convolution layers have higher operational intensities (the amount of computation per off-chip data transfer), they have smaller off-chip data traffics than fully-connected layers. Based on this observation, we propose a CU pair where one is dedicated to inference tasks and the other to training tasks to make their workloads asymmetric. This makes FA3C more efficiently utilize the limited off-chip DRAM bandwidth. When FPGA resource allows, increasing the number of CU-pairs also increases parallelism.
3. **Two-level buffer hierarchy to maximize parallelism.** We propose a two-level buffer hierarchy that consists of on-chip buffers made of on-chip memory and line buffers made of registers. On-chip buffers effectively exploit the off-chip DRAM interface in the burst mode and cache data items from the DRAM. Line buffers prefetches and caches data items stored in different locations of an on-chip buffer through shifting, stitching, and scattering operations. Line buffers seamlessly feed operands to PEs to avoid stalls and to maximize parallelism.
4. **On-the-fly data layout changes using on-chip buffers.** Many FPGA-based DNN accelerators rely on compressed parameter representations, such as low precision values [20] or sparse matrix encoding [10] to reduce off-chip data traffic. This approach is applicable to inference-only platforms where the parameter compression is applied offline after training has been completed. However, an RL training process cannot rely on parameter compression because DNN parameters are continuously updated. Instead, we design data layouts for on-chip and off-chip memory to support the two-level buffer hierarchy. They efficiently serve different data usage patterns of different computational tasks and efficiently utilize the limited off-chip data bandwidth. We maintain only a single copy of the data in the off-chip DRAM. The layout is changed on-the-fly when the data are being loaded to on-chip buffers.

We demonstrate the performance and energy efficiency of FA3C by evaluating it with six Atari 2600 games. FA3C achieves higher computation performance than that of a platform with a high-end GPU (NVIDIA Tesla P100) that uses the highly-tuned NVIDIA cuDNN library. FA3C performs 27.9% better than the GPU-based A3C platform. Moreover, the performance-per-Watt of FA3C is 1.62× better than that of the GPU.

2 Background

In this section, we briefly review the deep reinforcement learning (RL) methods.

2.1 Deep Reinforcement Learning

In RL, an agent interacts with the environment over a discrete number of time steps (Figure 1). At each time step t, the agent receives the current state s_t from the environment and selects an action a_t from the action space A according to its policy π. After performing the selected action a_t in the environment, the agent receives a reward value r_t and the next state s_{t+1}.

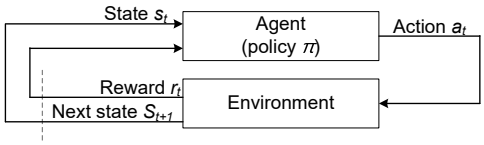


Figure 1. Standard RL setting.

In a video game playing scenario, the agent can be mapped to the game player. The state is the current screen pixels, the action is the player's control input to the game, and the reward is the received game score.

The goal of RL training is finding a policy that maximizes the sum of discounted rewards $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$ where T is the time step when the agent encounters the terminal state (*e.g.*, the end of a game episode), and $\gamma \in (0, 1]$ is a discount factor that reflects the importance of future rewards. An RL agent may directly learn the policy, or it may indirectly optimize the policy by learning a *value function* that evaluates how good each state and an action is.

Deep RL applications use DNNs to approximate the RL components (*i.e.*, the policy or value function) from high-dimensional inputs. For example, DQN[18] uses a DNN to approximate the value function on the current state (*Q-function*) and determines the next action based on the Q-function value from the DNN.

Computations in Deep RL training can be classified into two types of tasks: *inference* and *training*. The *inference task* generates datasets for training by interacting with the environment using the current DNN parameters while the *training task* updates the DNN parameters.

Table 1. DNN Layers Used in A3C for Atari 2600 Games

	Layer type	# of parameters	# of output features
0	Input	-	28K
1	Convolution (**Conv1**) (filter: 8 × 8, stride: 4)	4K	6K
2	ReLU activation		
3	Convolution (**Conv2**) (filter: 4 × 4, stride: 2)	8K	3K
4	ReLU activation		
5	Fully-connected (**FC3**)	664K	256
6	ReLU activation		
7	Fully-connected (**FC4**)	8K	32
8	Softmax (action) / Linear (value)		

2.2 Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic (A3C) is based on the actor-critic RL method [28] in which inference tasks use the same policy as training tasks. A3C maintains the coupling of inference and training tasks by executing those tasks alternately in each agent.

The DNN used by A3C has two types of outputs: a softmax layer for the action policy $\pi(a_t|s_t;\theta)$ and a linear layer for the value function $V(s_t;\theta)$ where θ is the current DNN parameters. Table 1 lists the DNN layers used in A3C for Atari 2600 games. Except for the last layer for the policy and value output, the DNN is the same as the DNN model used for DQN [17].

The update of A3C DNN parameters is performed by the policy gradient method, which is minimizing the following two objective functions through parameter update with train data samples (A3C additionally include an entropy regularization term in the policy objective function):

$$f_\pi(\theta) = -log\pi(a_t|s_t;\theta)(R_t - V(s_t;\theta)) \quad \text{and} \quad f_V(\theta) = (R_t - V(s_t;\theta))^2$$

where R_t is the sum of discounted rewards from step t, and it can be evaluated at a terminal state. Instead of waiting for the end of an episode to perform training, A3C uses *bootstrapping* to estimate R_t. Bootstrapping uses the value V of the state after k steps to estimate R_t as $\hat{R_t} = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta)$. In A3C, the number of steps to apply bootstrapping (*i.e.*, k) is between 1 and t_{max} (t_{max} is set to 5 in the original A3C publication). In computation point of view, this adds an extra DNN inference before the training task happens after t_{max} inference steps.

Figure 2 illustrates the training process of A3C. A3C has a shared global parameter set (*global* θ), and each agent maintains its own copy of the parameter set (*local* θ), a snapshot copy of the current global θ. Each agent separately performs inference and training tasks. During the training process, the global θ is continuously updated. A routine in the A3C training process mainly consists of t_{max} inference steps that use local θ and a training task that follows the inference tasks. In addition, an agent performs a *parameter sync* task that copies the current global θ as its local θ before it starts the inference tasks. Also, a bootstrapping inference is performed before the training task to calculate the objective functions. The input datasets used in the inference tasks form a training batch that contains t_{max} training datasets for the following training task. The training task calculates the gradients of the DNN parameters based on local θ. The obtained gradients are applied to global θ using the RMSProp optimizer[11].

2.3 DNN Computations

The DNN computations are categorized into three types: forward propagation (*FW*), backward propagation (*BW*), and gradient computation (*GC*). In a training task, FW, BW, and GC are performed in turn. An inference task is FW itself. FW is performed using input feature maps and the DNN parameters in each layer to produce output feature maps. Output feature maps become input feature maps to the following layer.

BW computes how much the input feature map values should be changed to optimize the objective function (i.e., *gradients* of the input feature maps). The gradients of the feature maps are not directly applied to the DNN parameters.

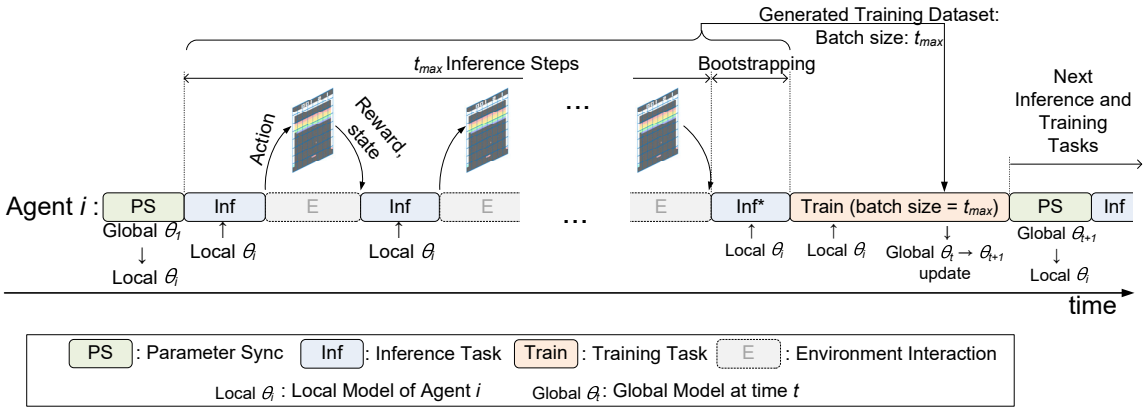


Figure 2. Computations performed in A3C.

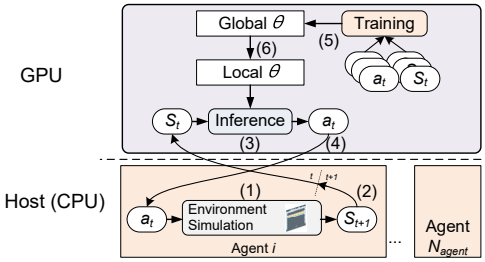


Figure 3. A3C on a platform with a CPU and a GPU.

Instead, they are required by GC to calculate the gradients of the DNN parameters. BW computation proceeds from the last layer to the first layer. In each layer, BW is performed using the DNN parameters of the layer and the gradients of the output feature maps.

The goal of GC is to compute the gradients of the DNN parameters. GC uses the input feature maps computed by FW and the gradients of the output feature maps computed by BW of the following layer.

3 Performance Bottlenecks in A3C

In this section, we describe the performance bottlenecks in A3C computation and advantages of an FPGA-based A3C platform over a GPU-based platform to resolve the bottlenecks. In general, FPGA-based platforms provide higher energy efficiency than GPU-based platforms.

3.1 A3C on a GPU-based Platform

Figure 3 illustrates how A3C can be implemented in a platform with a multicore CPU and a GPU. On the host (CPU) side, each agent is implemented as a thread. Each agent thread applies an action to its local environment, obtains the next state (1), and transfers the state to the GPU to offload the inference task (2). On the GPU side, the inference task is performed using the DNN parameters for the local model that resides in GPU memory (3). The result of the inference(*i.e.*, the action for the next state) is passed to the CPU (4). After T_{max} + 1 steps of inferences, a training task is executed on the GPU with the accumulated states and actions to update the global model (5). The global model is copied to the local model after the training task (6). Note that inference tasks and training tasks of all the agents are executed on a shared GPU.

3.2 Batch Size Limitation in A3C

For a given platform with a fixed amount of computing capacity and off-chip data bandwidth, the achievable DNN training performance is often limited by the amount of computation per off-chip data transfer (*operational intensity*). The operational intensity largely depends on the computation batch size (*the number of datasets per fetched parameter set*).

As shown in Figure 2 and Figure 3, an agent performs an inference task using the current environment state as the input to the DNN. Based on the inference result, the next action is selected and applied to the environment. Inevitably, we cannot create a large inference batch because the next state of the environment depends on the inference result on the previous state. Similarly, the batch size for a training task is also limited. After t_{max} + 1 inference steps, a training task is executed with a batch size of t_{max}, which is not large enough for efficiently utilizing GPU computing resources. However, the batch size of A3C training is tightly related to the training quality, and an arbitrary change may result in degraded quality. For example, to reach the game score of 200 points in Breakout in Atari 2600 games, A3C training requires about 35 million steps when t_{max} is 5 whereas it requires over 70 million steps when t_{max} is set to 32. This leads to a low computational efficiency on GPU platforms that are designed for high operational intensity.

3.3 Limited Off-chip Data Bandwidth

A3C training generates a large amount of off-chip data traffic. Table 2 summarizes the theoretical amount of off-chip data

Table 2. Off-chip Data Traffic in A3C Training

Task type	Data type	Load	Store
Parameter sync	Global θ	2,592KB×1	-
	Local θ	-	2,592KB×1
Inference task (batch size: 1)	Local θ	2,592KB×6	-
	Input data	110KB×6	-
Training task (batch size: 5)	Global θ	2,592KB×1	2,592KB×1
	RMS g	2,592KB×1	2,592KB×1
	Local θ	2,592KB×1	-
	Input data	110KB×5	-
Total		24,538KB	7,776KB

traffic for a training routine of an A3C agent. Since the memory subsystem in the GPU side is fixed, a new memory hierarchy specifically suitable for A3C cannot be implemented in a GPU-based platform. Moreover, it is not easy to apply data layout management techniques that maximize the off-chip memory bandwidth utilization in a GPU-based platform. Such data layout transformation comes with considerable execution time overhead in a GPU-based platform. We replicate the FA3C data layout management scheme using software in the GPU side, but the layout transformation task offsets the performance gain. However, it is easy to implement a customized memory subsystem specifically for A3C in an FPGA-based platform.

3.4 Kernel Launch Overhead

The performance of A3C heavily depends on the execution latency of small computation batches because an agent has to interact with its environment using the previous inference result. A typical GPU programming model, such as OpenCL and CUDA, incurs overhead to launch kernels. The kernel launch overhead is not trivial in GPU-based A3C because it frequently launches kernels that contain a relatively small amount of computation. When we measure the kernel launch overhead by comparing the execution times of A3C kernels implemented in CUDA with those of dummy kernels that contain no meaningful computation, the launch overhead accounts for more than 38% of the overall GPU kernel execution time. The kernel launch time is almost nonexistent (less than 0.02% in our measurement) in an FPGA-based platform.

4 FPGA-based Platform Design

In this section, we describe in detail the design of our FPGA-based A3C Deep RL platform.

4.1 FA3C Platform Architecture

Figure 4 shows the architecture of the FA3C platform. Each agent is implemented as a thread in the host (*i.e.*, CPU) to interact with the environment. For example, the environment for Atari 2600 games is the Arcade Learning Environment (ALE) framework[6]. DNN computation is performed by compute units (*CU*s) in the FPGA. A CU consists of multiple processing elements (*PE*s). Each CU is capable of executing

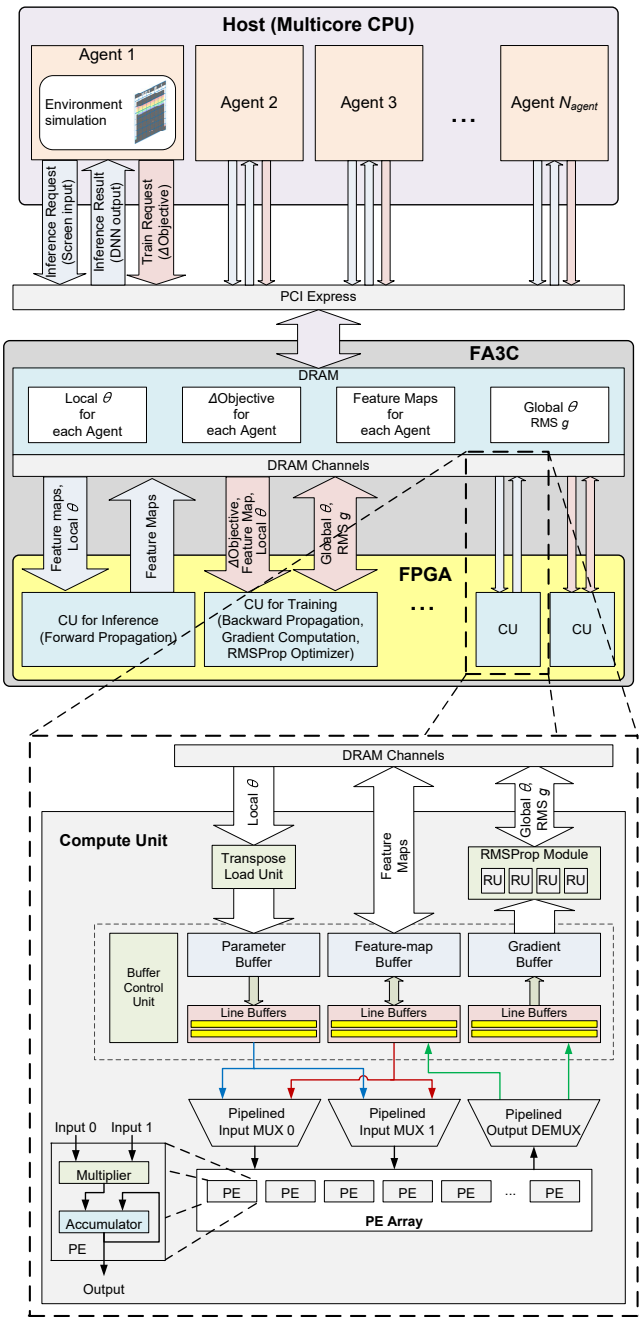


Figure 4. FA3C platform architecture.

either an inference or training task and serves one agent at a time. The FA3C architecture basically has a pair of CUs to execute inference and training requests from the multiple agents. However, depending on available FPGA resources and off-chip data channels, it may include more pairs of CUs to improve throughput.

An agent communicates with FA3C through PCI-E DMA operations. Its inference request is initiated by sending its current game screen input to FA3C. The inference request is

finished when the output of the last DNN layer in FA3C is transferred back to the agent after the forward propagation. The agent then calculates a softmax function to determine the action for the current time step. The softmax calculation is not off-loaded to FA3C because the amount of computation is relatively small, and it would require resource-consuming exponential functions in the FPGA.

DNN parameters reside in the off-chip DRAM of the FPGA side. They are loaded to FPGA on-chip buffers on demand. If there exist multiple off-chip DRAM channels, FA3C locates global parameters and local parameters in different memory channels.

For training, the agent calculates the objective function and sends the obtained gradients (ΔObjective) to FA3C. Similar to softmax, this computation is not off-loaded to the FPGA side. The DNN feature maps generated by an inference task are stored in the off-chip DRAM and reused by the associated training task to reduce computation overhead.

4.2 Processing Elements and Compute Units

As shown in Figure 4, a CU contains multiple PEs, multiple on-chip buffers, multiple line buffers, an RMSProp module, a transpose load unit (*TLU*), and a buffer control unit (*BCU*). An on-chip buffer consists of multiple rows of Block Memory[31] or Embedded Memory[2]. Each row is a one-dimensional word array. A line buffer is a one-dimensional word array made of registers. We describe the details of all CU components in this subsection but the buffers, TLU, and BCU. Their details will be explained in later subsections.

4.2.1 Processing elements.

The basic building block of a CU is a PE that consists of a pair of 32-bit single-precision multiplier and accumulator (Figure 4). We choose this configuration to support all three types of computations in the DNN (forward propagation, backward propagation, and gradient computation).

In other (inference-only) FPGA-based DNN implementations, the basic computation unit is an array of multipliers followed by an adder tree [24, 33] or the systolic array [13, 30]. However, neither of these units are suitable for efficient DNN training because depending on the layer configuration and computation type, the number of values that need to be accumulated (we call it *accumulation frequency*) to generate one element of the output is different.

For example, the accumulation frequency is $I \times K^2 + 1$ for the forward propagation of a convolution layer, where I is the number of input channels and $K \times K$ is the size of convolution filters. However, the accumulation frequency is the same as the batch size for computing a gradient value of a fully-connected layer. A fully-connected layer can be considered as a convolution layer with R,C, and K are all ones, where R and C are the numbers of rows and columns of the output feature map, respectively.

Since an adder tree or systolic array has to be designed for a specific accumulation frequency, they are inefficient for other values of accumulation frequency. Moreover, the systolic array is not fully utilized with an operation that is not a matrix-by-matrix multiplication, such as those appeared in fully-connected layers. However, in our PE structure in Figure 4, it is possible to control the accumulation frequency to support various types of operations.

4.2.2 Compute units.

Multiple PEs form a CU that is capable of performing either an inference or training task across all layers. A CU also contains on-chip buffers for feature maps, parameters, and gradients (Figure 4). Because a single PE is utilized by multiple input and output data types, the datapath to and from a PE are equipped with pipelined mux and demux to help meet the timing. Although this requires a large number of FPGA registers, FA3C is rather limited by on-chip memory and DSPs than registers. Thus, the pipelined datapath utilizes the abundant registers in the FPGA fabric.

As mentioned before, the basic configuration of FA3C uses two separate CUs, one dedicated to inference and the other to training to balance the off-chip data bandwidth between DNN layers. The performance of two CUs with N PEs is better than that of a single CU with $2N$ PEs because the off-chip data bandwidth can be more effectively utilized by the two CUs.

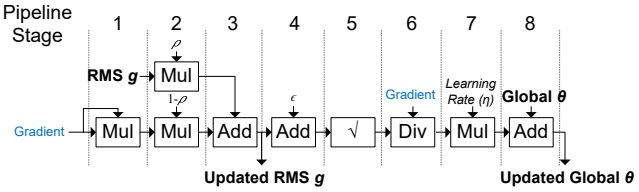


Figure 5. An RU computation pipeline. Values written in bold indicate off-chip traffic. ρ, ϵ, and η are constant RMSProp parameters.

4.2.3 The RMSProp module.

As shown in Figure 4, computed gradients are applied to the global model parameters (global θ) through a dedicated RMSProp module. The RMSProp module maintains a separate RMSProp parameter g for each global θ value[11]. Because the update is independently computed for each parameter value, the computation can be fully pipelined (Figure 5). We call such a computation unit inside the RMSProp module an *RU*.

FA3C increases parallelism with multiple RUs. On every clock cycle in the RMSProp computation, an RU reads two words and also produces two words (Figure 5). Thus, when the off-chip DRAM interface has a data width of 16 words,

four RUs are sufficient to fully exploit the off-chip DRAM bandwidth.

When on-chip gradient buffers are filled up with computed gradients, the RMSProp computation is triggered. The RMSProp module also contains multiple on-chip buffers to handle the update of θ and g parameters using double buffering. While the RUs are performing θ and g update on a buffer, the RMSProp module handles off-chip data traffic of the other buffer.

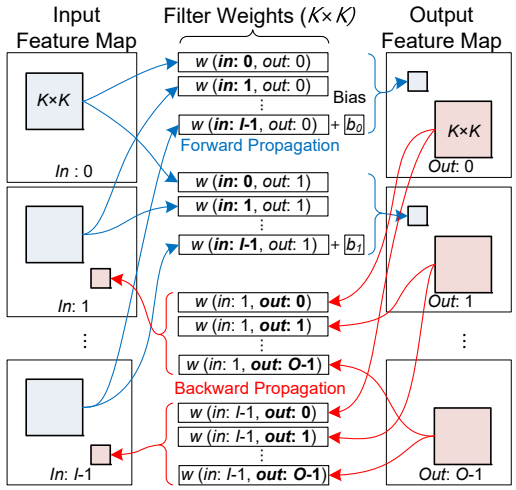


Figure 6. Computations in forward and backward propagation. $w(in{:}x, out{:}y)$ stands for the weights of a filter that is associated with input channel x and output channel y.

4.3 Data Layout: Feature Maps

The off-chip DRAM interface typically transfers sixteen single-precision floating point values (sixteen 32-bit words or 512 bits) at a time in the burst mode. Thus, each row in the two-dimensional feature map stored in the DRAM (for fully-connected layers, the feature map has only a single row) is aligned in 16-word granularity to simplify data transfer to the on-chip buffer. The width of a row in the on-chip feature map buffer is sixteen words so that we can fully exploit the DRAM bandwidth on every cycle during off-chip burst transfers. The internal fragmentation caused by the alignment may increase off-chip data traffic to store/load feature maps, but the increased amount of the off-chip data traffic is less than 1% of the total off-chip data traffic.

Output feature maps are computed via forward propagation (Figure 6) in inference stages using the agent's local model. The same feature maps are used again in the gradient computation for the same agent. FA3C does not recompute the output feature maps for the following training task because it would waste computation and off-chip data bandwidth to load model parameters. Instead, FA3C saves the output feature maps in the off-chip DRAM after each inference task and reload them in the successive training task.

The gradients of the feature maps computed during the backward propagation step also use the same on-chip feature map buffer because their dimensions are the same. FA3C schedules the backward propagation after the gradient computation in each layer to avoid an overlap in the buffer use.

4.4 Data Layout: DNN Parameters

Managing the data layout for DNN parameters is more complicated because a desired on-chip data layout could be different depending on different computation stages (Figure 6). In Figure 6, I and O are the numbers of input and output channels, respectively.

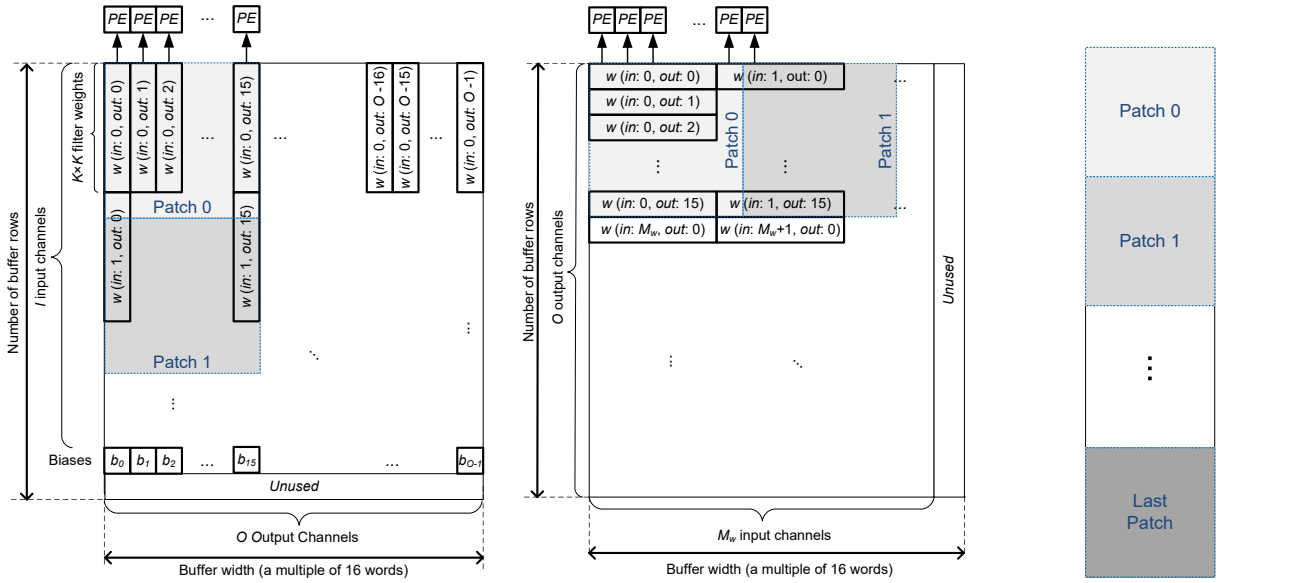
4.4.1 Layout for forward propagation.

During the forward propagation of a convolution layer, an output feature value is calculated using $K \times K$ parameters over I input channels (Figure 6). FA3C uses a special data layout in the on-chip parameter buffer and utilizes all the available PEs to compute multiple output feature values simultaneously. FA3C makes a PE in charge of an output value and consume the required parameter values in sequence over $I \times K^2$ cycles. Thus, only one parameter value out of the $I \times K^2$ parameter sequence is stored in each row of the on-chip parameter buffer, and a row contains O parameter values for each of the output channels. This layout, called *FW parameter layout*, is shown in Figure 7a.

4.4.2 Layout for backward propagation.

The backward propagation step calculates the gradients of feature map values. For simplicity, we assume the stride size (S) is one in this subsection. The gradient value of an input feature (*input gradient*) is calculated by accumulating the product of $K \times K$ gradient values over O output channels (*output gradients*) and corresponding parameter values (Figure 6).

Suppose that the parameter values are stored using the FW parameter layout for backward propagation. Parameters that can be accessed in a single clock cycle (*i.e.*, parameters in a single buffer row) all belong to the same input channel. Thus, all PEs calculate input gradients of a single input channel only. In this case, the computation throughput can be increased by calculating multiple input gradients of the same input channel simultaneously. However, a large amount of output gradients have to be accessed simultaneously because different input gradients in the same input channel are associated with different regions of the output feature maps. In the extreme, for fully-connected layers where each individual input feature map value is considered as a separate input channel, the computation throughput is severely limited. Since there is only one input gradient to produce for a given input channel, FA3C ends up utilizing only one PE at a time under the FW parameter layout.



(a) FW parameter layout in the buffer. **(b)** BW parameter layout in the buffer. **(c)** Parameter layout in the DRAM.

Figure 7. DNN parameter layouts in FA3C. $w(in{:}x, out{:}y)$ stands for the weights of a filter that is associated with input channel x and output channel y.

This issue can be resolved by making PEs access parameter values for multiple input channels at the same time. One of the layouts for this access pattern would be a layout where the index of output and input channels are switched from the FW parameter layout. To implement this layout, FA3C uses a simple hardware mechanism, transposition. It transposes the FW parameter layout for backward propagation when the parameters are loaded to the on-chip parameter buffer. The layout, called *BW parameter layout*, is shown in Figure 7b.

For simplicity, consider a case where $K^2 < O$. This covers most of widely-used DNNs including that used in A3C. In the BW parameter layout, each parameter buffer row contains $M_w = \left\lfloor \frac{O}{K^2} \right\rfloor$ filter weights of different input channels. This layout enables PEs to compute input gradients across multiple input channels simultaneously. Thus, FA3C can calculate multiple input gradients using the same set of output gradients. It also resolves the low PE utilization issue for fully-connected layers because of a large value of M_w (*e.g.*, the same as O) for fully-connected layers.

4.4.3 Layout in the DRAM and the TLU.

Even though the on-chip parameter buffer uses two different versions of parameter layouts (*i.e.*, FW and BW), FA3C maintains one copy of the parameters in the off-chip DRAM. FA3C partitions the parameter values into 16×16-word patches, represented by gray squares in Figure 7. These patches are stored contiguously in the DRAM in a manner shown in Figure 7c.

To load these patches in the BW parameter layout, the transpose load unit (TLU) transposes each patch using registers and shift operations. Then, the parameters in the transposed layout are placed in the parameter buffer to be used by the PEs. A CU has two instances of TLUs to improve performance. One of them fills in the parameter buffer while the other is preparing the transposed patch. To hide the latency from the DRAM interface, a TLU sends the read requests in advance before the PEs begin to use the parameters. The prefetched parameters are temporarily staged in an on-chip FIFO buffer before TLU performs the transformation.

Parameter buffers in FA3C also use double buffering to achieve performance improvement through computation and data access overlap.

4.4.4 Layout for gradient computation.

The gradient buffer uses the FW parameter layout to simplify the RMSProp parameter update process. Thus, the RMSProp module does not need to use the TLU when loading the global parameters from the DRAM.

4.5 Line Buffers and the Buffer Control Unit

A line buffer is a one-dimensional array of registers that prefetches and caches multiple elements in different locations of an on-chip buffer. It feeds operands to all PEs simultaneously to avoid stalls and to maximize parallelism. The BCU controls the behaviors of the line buffers. Each on-chip buffer may have multiple sets of line buffers to support double buffering and prefetching. The line buffer for the outputs

of PEs does not need double buffering because PEs produce output values only after its accumulation finishes.

The BCU supports a combination of the following line buffer management operations:

- **Shifting.** A PE may access data items in different locations in an on-chip buffer row. This may result in an excessive routing requirement because the row width can be very wide. Fortunately, the access patterns are highly regular, and FA3C employs a shifting mechanism to mitigate the routing efforts. For example, in forward propagation, a row of the input feature map is loaded to a line buffer and shifted left by one word each clock cycle. The BCU shifts the data items in the line buffer over K cycles (*i.e.*, the width of a convolution filter), and then replace the line buffer with the next feature map row.
- **Stitching.** The BCU restores a feature map row by combining multiple on-chip feature map buffer rows in a single line buffer. The stitched line buffer becomes a one-dimensional row that moves together when shifting happens. This operation is required when the feature map width is bigger than the on-chip buffer width (*i.e.*, 16 words).
- **Scattering.** If PEs write data items that are scattered over multiple on-chip buffer rows, they first write the data items to a line buffer, and then the BCU sends the data items in the line buffer to the on-chip buffer. For example, in the forward propagation, PEs produce output feature map items over multiple channels simultaneously. The BCU manages the data items using a line buffer to distribute them to separate on-chip buffer rows.

Table 3. Sizes of Line Buffers

Stage	PE port	On-chip buffer	Width	Number of line buffers
FW	Input 0	Inpute feature map	C_{in}	1
	Input 1	Parameter (FW parameter layout)	$min(N_{PE}, O)$	0
	Output	Output feature map	N_{PE}	1
GC	Input 0	Inpute feature map	C_{in}	K
	Input 1	Output feature map (gradient)	C_{out}	$M_{GC} = \left\lfloor \frac{N_{PE}}{K^2} \right\rfloor$
	Output	Gradient	N_{PE}	1
BW	Input 0	Parameter (BW parameter layout)	$min(N_{PE}, O)$	0
	Input 1	Output feature map (gradient)	C_{out}	$M_{BW} = \left\lfloor \frac{N_{PE}}{M_w \times C_{in}} \right\rfloor$ $(M_w = \left\lfloor \frac{O}{K^2} \right\rfloor)$
	Output	Input feature map (gradient)	N_{PE}	1

4.5.1 Sizes of line buffers.

Table 3 summarizes the sizes of line buffers for each computation type. C_{in} and C_{out} represent the widths of input and output feature maps, respectively. N_{PE} stands for the number of PEs in a CU.

For example, the forward propagation uses a line buffer with C_{in} elements for the input feature map. Since C_{in} can be larger than the width of the on-chip feature map buffer (16 words), the BCU gathers $\left\lceil \frac{C_{in}}{16} \right\rceil$ feature map buffer rows to compose a line buffer. For a word in the line buffer, O PEs that correspond to each output channel are assigned. The line buffer is shifted by one word on each cycle so that the PEs can access the required input feature value without changing the input port. If $N_{PE} > O$, $M_{FW} = \left\lfloor \frac{N_{PE}}{O} \right\rfloor$ data items in the line buffer are used simultaneously to increase parallelism. The parameter buffer does not require a line buffer because its on-chip buffer layout already matches the PE access pattern, but we may add a line buffer to improve timing. In this case, the width of the line buffer would be the same as the width of the on-chip parameter buffer, $min(N_{PE}, O)$. The output values from PEs are temporarily stored in a line buffer for the required data transformation by the BCU (*e.g.*, scattering) to match the target on-chip buffer layout.

The gradient computation loads K lines from the input feature map buffer to K line buffers so that the PEs can calculate the gradients of a $K \times K$ convolution filter in parallel. If $N_{PE} > K^2$, $M_{GC} = \left\lfloor \frac{N_{PE}}{K^2} \right\rfloor$ line buffers are loaded from the output feature map buffer to simultaneously access output gradients of multiple output channels to fully exploit available PEs. Similarly, the backward propagation uses multiple (M_{BW}) line buffers to maximize parallelism.

5 Evaluation

In this section, we evaluate the FA3C platform by comparing the training performance and energy efficiency with GPU-based A3C implementations. The A3C platforms are evaluated by training Atari 2600 games. We use six Atari games: Beam Rider, Breakout, Pong, Qbert, Seaquest, and Space Invaders.

Table 4. FPGA Resource Usage Breakdown on Xilinx VCU1525 UltraScale+ VU9P.

Component	Logic utilization	Registers	On-chip memory blocks	DSP blocks
PEs	188.8K	252.6K	0	2048
Parameter buffer	20.8K	1.7K	256	0
Gradient buffer	8.9K	0.6K	128	0
Feature-map buffer	9.2K	1.2K	192	0
BCU (line buffer)	72.1K	111.0K	0	0
RMSProp	53.4K	64.8K	216	288
Pipelined MUX	50.1K	50.1K	16	0
TLU	17.0K	35.1K	16	0
DDR-CU interconnect	83.3K	136.2K	263	0
DDR4 controller	86.3K	98.0K	102	12
PCI-E DMA	87.4K	124.4K	78	0
Total	677.3K (57.3%)	875.7K (37.0%)	1267 (40.6%)	2348 (34.3%)

5.1 Evaluation Environment

We implement FA3C using the Xilinx VCU1525 acceleration board. The platform has an UltraScale+ VU9P FPGA, which is the same FPGA used for the Amazon EC2 F1 FPGA instances. FA3C on VCU1525 has two sets of CUs, and each set contains one CU for inference tasks and another CU for training tasks. Each CU has 64 PEs each. Table 4 summarizes the resource usage breakdown on the Xilinx VU9P FPGA.

Table 5. Evaluation Platform

	System	
Host CPU	2× Xeon E5-2630 2.20 GHz	
Host memory	DDR4 128GB	
	FPGA	**GPU**
Model	Xilinx VCU1525 (UltraScale+ VU9P)	NVIDIA Tesla P100
Core clock speed	180MHz	1328MHz
Manufacturing process	16nm	16nm
External DRAM interface	DDR4	HBM2
Peak DRAM bandwidth	143 GB/sec	732 GB/sec
Host interface	PCI Express 3.0 x16	PCI Express 3.0 x16

In our evaluation, FA3C is compared with high-end GPU-based A3C implementations (Table 5). Both FPGA and GPU are evaluated using the same host system. In our evaluation, **FA3C** denotes our FPGA-based A3C Deep RL platform. **A3C-TF-CPU** is a CPU-based A3C Deep RL implementation. We use an open-source implementation of A3C that uses TensorFlow[1] as the DNN training framework[15]. **A3C-TF-GPU** is the same A3C implementation as A3C-TF-CPU, but the inference and training tasks are performed using GPU. **GA3C-TF** is also a TensorFlow-based Deep RL implementation for Atari games. It implements the GA3C algorithm on a GPU. It is a modified version of A3C. It combines all local models to a global model[5].

Because of ease of use, using a DNN framework, such as TensorFlow or PyTorch[23], to implement a DNN model is a common practice. However, for RL implementations, the framework may introduce non-trivial performance overhead because the DNN model has to frequently interact with the environment. Thus, we also evaluate the GPU performance without such overhead from the framework. In **A3C-cuDNN**, the training and inference tasks are performed by directly invoking the NVIDIA cuDNN library. cuDNN is a highly-optimized DNN computation library for NVIDIA GPUs[22]. DNN computation tasks, such as forward and backward propagation stages, are computed using cuDNN primitives. One exception is that the forward propagation tasks of the fully-connected layers are performed using cuBLAS[21] for better performance on smaller batches. CuBLAS is also a highly-tuned GPU library provided by NVIDIA.

We present the result from best-performing configuration parameters of each implementation.

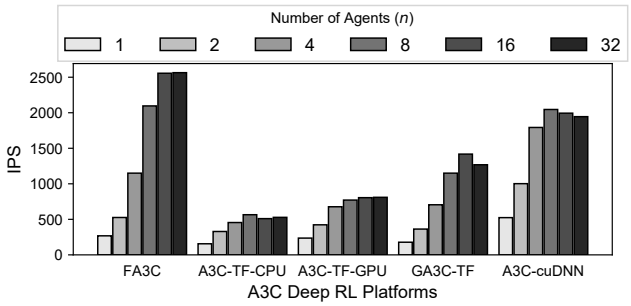


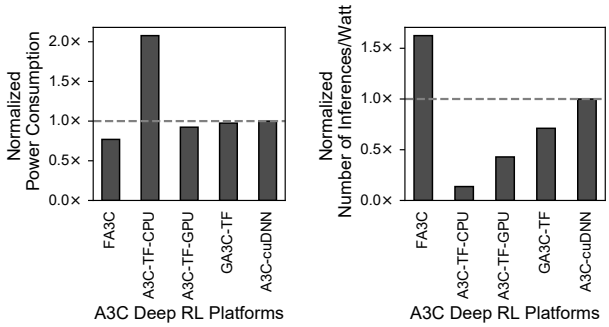
Figure 8. Performance of A3C Deep RL platforms.

5.2 Performance

We measure the performance of the Deep RL platforms using the number of inferences processed per second (*IPS*) across all agents. Because each A3C agent performs a training task every t_{max} inference requests, the IPS also accounts for the processed training tasks. For example, when t_{max} is 5 and the achieved IPS is 500, the Deep RL platform processes 500 inference tasks, 100 extra inferences for value bootstrapping, and 100 training tasks per second.

Figure 8 compares the IPS values between FA3C and other platforms. As the number of agents (n) grows, the platforms generally show higher IPS. The peak performance is observed when $n \geq 16$ on FA3C. When $n = 16$, which is the maximum number of agents used in the original A3C publication, the IPS of FA3C is higher than 2,550. It is 27.9% better than the best IPS of A3C-cuDNN.

On the same GPU-based TensorFlow framework, GA3C-TF has better performance than A3C-TF-GPU, but both of them are worse than FA3C or A3C-cuDNN. TensorFlow also utilizes cuDNN, but it may include additional overhead from the framework. Please note that the training method used by GA3C is different from A3C, which may result in different training quality.



(a) Power consumption **(b)** Performance per Watt

Figure 9. Power efficiency of A3C Deep RL platforms.

5.3 Energy Efficiency

The power consumption during the A3C training process not only consists of the power consumed by the GPU or FPGA computing platform. The host also consumes a significant amount of power for environment simulation. To compare the energy efficiency of each platform (including the energy overhead for the communication with the host), we compare the power consumption difference between the whole system and a dummy platform during the A3C training process. The dummy platform also runs the A3C agents, but it does not perform any DNN inference or training task. Instead, the A3C agents in the host simulate the environment with randomly-selected actions.

We use an external power meter that measures the actual power consumption (Watt) of the system, and we report the 10-minute power consumption average during the A3C computation. To accurately measure the power consumed by GPU or FPGA, the clock speed of the CPU cores are fixed to their maximum frequency during the measurement. Figure 9a shows the measured power consumption (the lower the better). The graph shows relative power consumption levels normalized to the power consumed by A3C-cuDNN, which has the best performance among the GPU platforms.

Each bar shows the power consumption difference between the whole system (*i.e.*, host+FPGA or host+GPU) and the dummy platform. The IPS of the dummy platform is set to match the target A3C platform. We observe that FA3C consumes 18 Watts, on average, for the A3C computation. It is a 30.0% reduction from the A3C-cuDNN average power consumption.

Figure 9b shows the energy efficiency measured in the number of inferences processed per Watt (the higher the better). The graph is also normalized to the energy efficiency level of A3C-cuDNN. We observe that FA3C achieves more than 142 inferences per Watt. This is 1.62× better than the energy efficiency of A3C-cuDNN.

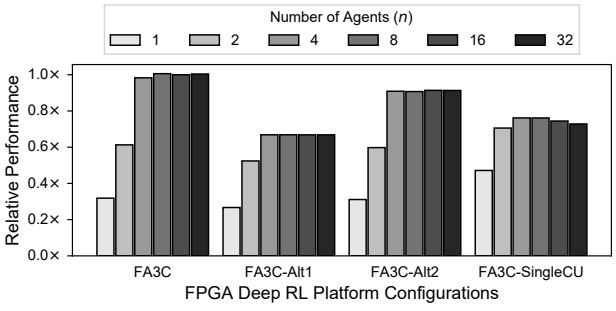


Figure 10. Performance of different FA3C configurations.

5.4 Comparison of FA3C Configurations

To evaluate the effectiveness of our FPGA design strategies, we compare different configurations of the FA3C platform. First, to show the effectiveness of our data layout mechanism, we compare FA3C with two alternative approaches. **FA3C-Alt1** is a configuration where all computation types use the same FW parameter layout. **FA3C-Alt2** generates both FW and BW parameter layouts in the DRAM when the global θ is updated by the RMSProp module. Also, we show the performance benefit of our dual CU design (*i.e.*, separate inference and training CUs) through a comparison with **FA3C-SingleCU**, where the PEs of the two CUs are combined into a single CU, and the single CU handles all inference and training requests.

Figure 10 compares the training performance of different FA3C platform configurations. The graphs show the relative performance normalized to the FA3C performance when $n = 16$ (the higher the better). The performance comparison was done using an FA3C implementation on the Altera Stratix V FPGA using one set of CUs (*i.e.*, one inference CU and one training CU. FA3C-SingleCU has one CU that handles both inference and training).

The achieved performance results from FA3C-Alt1 and FA3C-Alt2 show the performance benefits of our on-chip and off-chip parameter layout mechanisms. FA3C-Alt1 results in significant performance degradation (33% lower when $n = 16$) due to the underutilized PEs during backward propagation. Many PEs are left idle, especially for the fully-connected layers, because the required parameter values are not fetched at the rate required by the PEs. The training performance of FA3C-Alt2 is slightly lower than the FA3C performance because it has to write two versions of the same parameter set in different layouts to the DRAM. Note that it not only introduces extra off-chip traffic, but also requires additional FPGA resources to generate two sets of the parameters.

FA3C-SingleCU shows better performance than FA3C when there are a small number of agents. This is because all PEs can be flexibly utilized for all three different types of computations, where the amount of computation is not perfectly balanced between different types. However, as the number of agents grows, the achieved performance of FA3C gets better than that of FA3C-SingleCU. In FA3C with dual CUs, the off-chip data bandwidth is shared by the two different tasks running on the two CUs. A task with a low operational intensity can be jointly executed with another task with a high operational intensity in the other CU by effectively sharing the off-chip data bandwidth without hindering with each other. The platform benefits from the dual CU design when $n \geq 4$ where the platform is fully utilized.

5.5 Limited Optimization Opportunities for GPU-based Platforms

The data layout management of FA3C is accomplished by exploiting the highly flexible architecture of FPGA-based platforms. On GPU platforms, where the memory hierarchy is fixed, the obtainable performance gain from such a mechanism is rather limited. To demonstrate the limitation, we

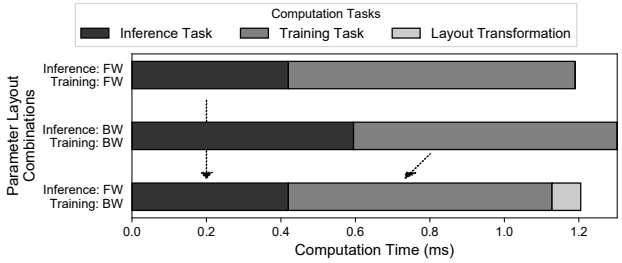


Figure 11. GPU computation time under different parameter layouts.

compare the GPU computation performance under different parameter layouts. For a flexible use of custom data layouts, we use our own A3C implementation written in OpenCL for a GPU. We highly tune the OpenCL implementation to closely match the computation speed of A3C-cuDNN within 12%.

Figure 11 compares the computation time of the inference and training tasks on the GPU under three different parameter layout combinations. To highlight the differences, Figure 11 shows the computation time for the fully-connected layers only, which accounts for the majority of the total parameter size. When both tasks use the same parameter layout, the task with mismatching parameter layout experiences performance degradation. For example, the inference task with the BW parameter layout is 41.7% slower than the case when it is with the FW parameter layout. Better performance can be achieved if we provide each computation task with the best matching parameter layout (*i.e.*, the lowermost bar in Figure 11). However, the GPU needs an extra kernel execution for the data layout transformation. This may offset the obtained performance gain. On FA3C, the dedicated TLU module performs the transformation to hide the performance impact.

5.6 Atari Game Training Results

We present the Atari game training results on our FA3C platform. The training results reported in the original A3C publication show the average scores from the best training runs with different learning rate per game. Also, the results are measured using a crafted initial game condition, called the *human starts evaluation metric*. Because the exact initial condition is not available publicly, it is difficult to reproduce exactly the same result. Instead, we compare the training result of FA3C with those from A3C-TF-GPU to show that our FA3C platform correctly trains the A3C DNNs. We use the initial learning rate of 7×10^{-4} that linearly scales down to zero over 100M inference steps. The number of agents is set to 16.

The training graphs in Figure 12 show the game scores obtained during the training process. The observed raw game scores are highly noisy because each game instance is assigned with a different random seed, and an agent randomly selects an action from the probability distribution over π. To show the trend of the training progress, the graphs show the moving average over 1,000 game scores. The horizontal axis of each graph is the number of processed inference steps. Depending on the training performance (*i.e.*, IPS), the actual wall-clock time to reach a certain training outcome can be different per platform.

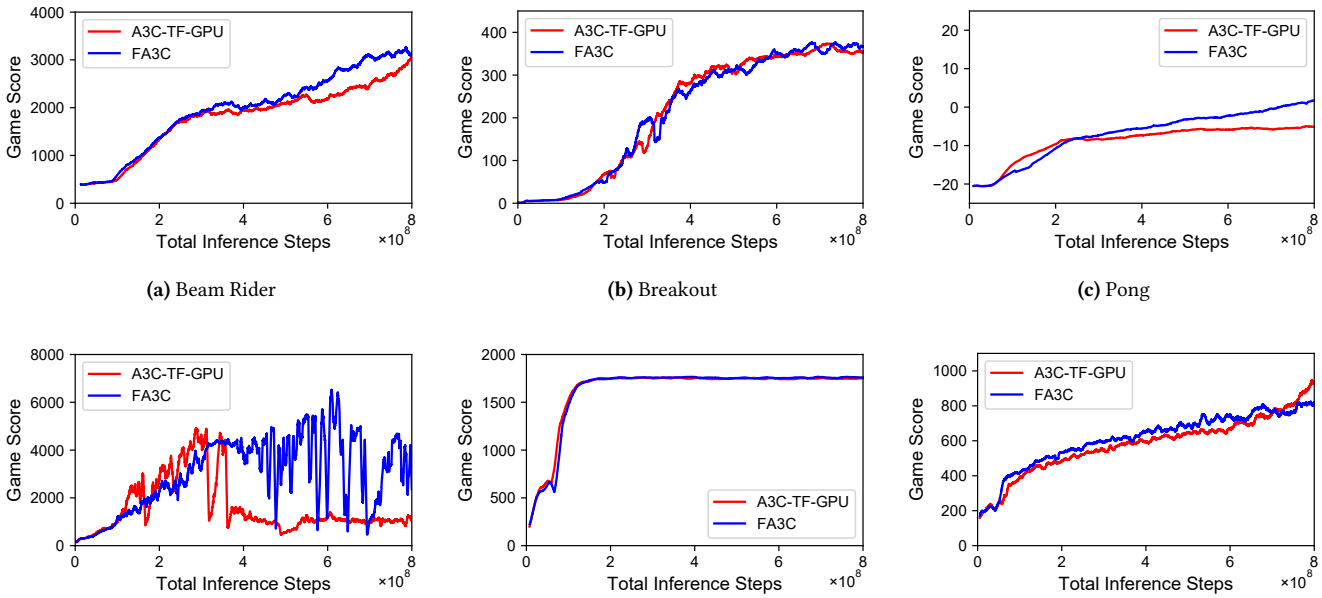
The results show that the FA3C platform has similar training trends to those of the GPU-based implementation, meaning that FA3C reaches a higher score earlier due to the better IPS value. The trend in Qbert is relatively unstable, but the same is true for the GPU-based implementation as well. The Deep RL training process is often unstable in some environments. It may require a rigorous hyperparameter tuning to successfully train the model. An efficient Deep RL platform plays a crucial role in such a process.

6 Related Work

The General Reinforcement Learning Architecture (Gorila) [19] distributes DQN training over multiple machines. Distributing DQN on 100 machines using Gorila can accelerate the training over 20 times, but A3C running on a CPU with 16 agents outperforms the Gorila framework.

To further accelerate A3C training using GPU platforms, two alternative methods [5, 7] were proposed. GA3C [5] tries to accelerate the A3C training by eliminating the local parameter set of the agents. GA3C maintains only one global parameter set, and all inference and training computations are performed using the single parameter set. This approach can combine multiple inference or training requests to form large computation batches. However, this approach can lead to unstable or slow learning because in GA3C, the model used for inference may be different from the model used for training. Similar to GA3C, PAAC [7] also enhances the GPU computation efficiency for A3C by maintaining only a single parameter set. Unlike GA3C, PAAC synchronizes all training computations; after each agent completes t_{max} steps, the agent has to wait until the update of the global parameter set is performed. The global parameter set is updated using the training data gathered from all agents. Since all training steps are synchronized, the performance may not scale to a larger number of agents.

Most of the existing FPGA-based DNN accelerators target inference computations only [4, 9, 24, 27, 32–34]. They are often based on high-level synthesis (HLS) or OpenCL for FPGAs [4, 33, 34] while some platforms use RTL-based implementations [9, 27]. TABLA [14] presents a framework that automatically generates FPGA accelerators for a variety of machine learning algorithms including the backpropagation stage of a neural network. However, its training performance



(a) Beam Rider **(b)** Breakout **(c)** Pong

(d) Qbert **(e)** Seaquest **(f)** Space Invaders

Figure 12. Atari game training results on FPGA and GPU platforms.

is comparable to GPU-based platforms only when the network is very small, with few tens of neurons.

7 Conclusions

Known GPU-based A3C platforms do not fully utilize the GPU's computing power because of the small batch size in A3C, limited off-chip data bandwidth, and frequent-kernel-launch overhead. This results in poor energy efficiency. To overcome the performance bottlenecks of the GPU-based platforms, we present the design and implementation of an FPGA-based A3C Deep Reinforcement Learning (Deep RL) platform, called FA3C.

We design a simple and generic processing element (PE) to serve various types of computations in DNN layers. A basic compute unit (CU) in FA3C contains multiple such PEs. A two-level on-chip buffer hierarchy in a CU prevents its PEs from stalling by seamlessly providing operands to the PEs. Data layouts in off-chip DRAM and on-chip buffers efficiently serve different data access patterns of different computational tasks. The buffer hierarchy and the data layouts together make a CU efficiently utilize the limited off-chip DRAM bandwidth. Only a single copy of the data is maintained in the DRAM. The layout is changed on-the-fly when the data are loaded to on-chip buffers. To effectively share the off-chip DRAM bandwidth, a pair of CUs is proposed with asymmetric loads. One of them is dedicated to inference, and the other is dedicated to training.

With these design principles, FA3C achieves both higher performance and better energy efficiency than a high-end GPU-based A3C platform. The performance we demonstrated in this work indicates that FPGA-based platforms can be exploited for DNN training not only for inference. Moreover, their high energy efficiency enables Deep RL to be applicable to applications in various areas including datacenters and embedded systems.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grants (No. NRF-2017R1C1B5017414 and No. NRF-2016M3C4A7952587), by the BK21 Plus program for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU, No. 21A20151113068) through NRF, and by the Institute for Information & communications Technology Promotion (IITP) grant (No. 2018-0-00581, CUDA Programming Environment for FPGA Clusters), all funded by the Ministry of Science and ICT (MSIT) of Korea. It was also supported in part by the Samsung Advanced Institute of Technology (SAIT) in Samsung Electronics Co., Ltd. ICT at Seoul National University provided research facilities for this study.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and*

Implementation (OSDI'16). USENIX Association, 265–283.

[2] Altera. 2018. Embedded Memory in Altera FPGAs. (2018). Retrieved Jan. 20, 2019 from https://www.intel.com/content/www/us/en/programmable/solutions/technology/memory/embedded.html

[3] Amazon Web Services, Inc. 2019. Amazon EC2 F1 Instances. (2019). Retrieved Jan. 20, 2019 from https://aws.amazon.com/ec2/instance-types/f1/

[4] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™ Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM Press, 55–64. https://doi.org/10.1145/3020078.3021738

[5] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. 2017. Reinforcement Learning thorugh Asynchronous Advantage Actor-Critic on a GPU. In *ICLR*.

[6] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2015. The Arcade Learning Environment: An Evaluation Platform for General Agents. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*. AAAI Press, 4148–4152.

[7] Alfredo V. Clemente, Humberto Nicolás Castejón Martínez, and Arjun Chandra. 2017. Efficient Parallel Methods for Deep Reinforcement Learning. *arXiv preprint* (2017). arXiv:1705.04862

[8] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, and Daan Wierstra. 2017. PathNet: Evolution Channels Gradient Descent in Super Neural Networks. *arXiv preprint* (2017). arXiv:1701.08734

[9] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. https://doi.org/10.1109/FCCM.2017.25

[10] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 243–254. https://doi.org/10.1109/ISCA.2016.30

[11] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. 2012. Overview of mini-batch gradient descent. (2012). Retrieved Jan. 20, 2019 from http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[12] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. 2016. Reinforcement Learning with Unsupervised Auxiliary Tasks. *arXiv preprint* (2016). arXiv:1611.05397

[13] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[14] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. TABLA: A unified template-based framework for accelerating statistical machine learning. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 14–26. https://doi.org/10.1109/hpca.2016.7446050

[15] Kosuke Miyoshi. 2016. Asynchronous deep reinforcement learning. (2016). Retrieved Aug. 7, 2018 from https://github.com/miyosuda/async_deep_reinforce

[16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML '16)*, Vol. 48. JMLR.org, 1928–1937.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari With Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533. https://doi.org/10.1038/nature14236

[19] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. 2015. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv preprint*. arXiv:1507.04296

[20] Eriko Nurvitadhi, Suchit Subhaschandra, Guy Boudoukh, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, and Duncan Moss. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM Press, 5–14. https://doi.org/10.1145/3020078.3021740

[21] NVIDIA. 2018. cuBLAS Dense Linear Algebra on GPUs. (2018). Retrieved Jan. 20, 2019 from https://developer.nvidia.com/cublas

[22] NVIDIA. 2018. NVIDIA cuDNN GPU Accelerated Deep Learning. (2018). Retrieved Jan. 20, 2019 from https://developer.nvidia.com/cudnn

[23] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS Workshop*.

[24] Jiantao Qiu, Sen Song, Yu Wang, Huazhong Yang, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, and Ningyi Xu. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM Press, 26–35. https://doi.org/10.1145/2847263.2847265

[25] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive Neural Networks. *arXiv preprint* (2016). arXiv:1606.04671

[26] Herman Schmit and Randy Huang. 2016. Dissecting Xeon + FPGA: Why the Integration of CPUs and FPGAs Makes a Power Difference for the Datacenter: Invited Paper. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED '16)*. ACM Press, 152–153. https://doi.org/10.1145/2934583.2953983

[27] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE, 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[28] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning : An Introduction*. MIT Press.

[29] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. 2016. Sample Efficient Actor-Critic with Experience Replay. *arXiv preprint*. arXiv:1611.01224

[30] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM Press, Article 29, 6 pages. https://doi.org/10.1145/3061639.3062207

[31] Xilinx. 2017. Block Memory Generator,. (2017). Retrieved Jan. 20, 2019 from https://www.xilinx.com/products/intellectual-property/block_memory_generator.html

[32] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD '16)*. ACM Press, Article 12, 8 pages. https://doi.org/10.1145/2966986.2967011

[33] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. ACM Press, 161–170. https://doi.org/10.1145/2684746.2689060

[34] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM Press, 25–34. https://doi.org/10.1145/3020078.3021698