CARL: CARL is Accelerated Reinforcement Learning *

Daniel Ahn Adrien Hadj-Chaib Josh Kimmel Hannah Nguyen

UCLA — CS 259 — June 2021

1 Background

Deep learning is a subset of machine learning and artificial intelligence that has become increasingly prominent in the last decade. As such, there has been a parallel increase in the efforts of architects to build specialized hardware to efficiently run the workloads that power deep learning.

A more nascent subset of machine learning is called reinforcement learning. This paradigm is centered around an agent within an environment. The agent can take some set of actions at any given time and is tasked with learning the best action to take given a current state. This is accomplished by reinforcement from the environment in the form of payoffs associated with taken and untaken actions. Reinforcement learning is not entirely disjoint from deep learning. In fact, it is very possible that some of the underlying decision-making mechanisms are deep neural networks (DNNs).

This more nascent machine learning paradigm gives rise to the opportunity to accelerate, just as we've seen with deep learning. Of course, acceleration of deep learning can very well be part of acceleration of a reinforcement learning system, but, as the DNNs are not the only underlying mechanism of decision making, they are also not the only mechanism that can benefit from acceleration.

In this report, we focus on a particular class of reinforcement learning that involves planning. The general problem being solved by this class is as follows:

- The agent exists in a state and can take one of a set of actions.
- The agent has some understanding of the world and thus the consequences of those actions, namely a hypothesis of the probability distribution of outcomes and rewards associated with potential actions.
- The agent, at each iteration, computes and takes the action with the highest expected reward, then updates its hypothesis about the world.

The probability distribution of outcomes and rewards associated with each potential action can be extrapolated `k` time steps into the future. This can effectively be represented as a tree where each node is an outcome associated with an action, and each edge has a weight according to the conditional probability of that outcome occurring. Leaf nodes are at most at depth `k` and have some terminal payout. The root node is the agent in its current state.

Under this formulation, assuming the hypothesis is indeed correct, a vital task of the agent is to analyze this tree to compute the action at time `0` that has the highest expected reward. This computation must occur frequently (at every time step) and, depending on tree structure, can be very

*https://en.wikipedia.org/wiki/Recursive_acronym

computationally expensive. Furthermore, at each iteration, other subsystems within the agent must evaluate prior decisions made by the agent and update its so-called hypothesis about the world accordingly (in real time). Thus, minimizing latency can make a huge difference in the system's ability to achieve an accurate understanding of the world. There also might be applications for this system that are extremely latency sensitive in the dimension of decision making (e.g. robotics).

Under the motivation of the problem statement described above, we have created an accelerated hardware implementation of the planning tree's evaluation and associated controller that can interface seamlessly with a traditional CPU. In doing so, the burden of a major component of heavy computation in the agent is lifted from the CPU, which can then devote itself to more energy efficient tasks. Alternatively, a lighter-weight CPU can be used in the system as it is not needed for the heavier computation involved.

We evaluate CARL against a pure software implementation of tree planning to demonstrate the performance gains that are possible with hardware acceleration. We also propose future work to both integrate the other necessary mechanisms (hardware-accelerated or not) into the system as well to allow for arbitrary scale and structure of planning trees. All code for the project is available at: `https://github.com/joshkimmel16/reinforce-accelerator`.

2 Planning Acceleration

This section details the microarchitecture and integration of the planner accelerator. At a high level, there are two primary modules of concern:

- Planning execution
- Planning controller

The execution module handles the actual computation of note assuming the tree is correctly configured. The controller offers an interface to software such that the tree can be properly configured for execution. Each subcomponent is discussed in detail to follow.

2.1 Planning Execution

This module is concerned with backpropagation through a provided tree to determine the optimal action and expected reward in the given time step. It has two primary input interfaces:

- Config Interface
- Data Interface

The config interface allows for high-level configuration of the tree. In this initial implementation, the only configurable parameter via this interface is the total number of nodes in the tree. The hardware imposes a strict cap of 1024 nodes, but processing can be sped up by specifying the actual number of nodes in the tree if it is less than 1024. Note that the hardware imposes a strict cap of 8 actions as well. As this number is not large, there is no need for a config input to control the true number of actions, but this will be introduced in future work.

The data interface allows for tree-specific parameters to be set on a per-node basis. As such, all data transactions must involve a node address. For the given node, the data transaction can specify any one of the following values:

- Parent address of the current node
- Reward of the current node (leaf node only)
- Action resulting in arrival at the current node
- Probability (weight) of action resulting in arrival at the current node
- Strategy of the given node (maximize vs. minimize expected reward)

Given the input interfaces above, it is clear that, subject to caps imposed by hardware, any arbitrary tree can be produced. Note that multiple connected instantiations of this module can handle arbitrarily large trees. Future work involves introducing equal scalability for the number of actions.

Once a tree has been fully configured, execution can occur. A reset signal allows for the computation to be restarted from the bottom of the tree at any point in time. The computation begins with the final leaf node (at index `num_nodes`-1) and proceeds as follows:

(1) If the current node's parent does not match the previous parent being processed, clear the action accumulation buffer. This accumulation buffer holds the current (partial or complete) expected rewards associated with each action for the current parent.

(2) Multiply the current node's reward by its probability and add this value to the accumulation buffer slot associated with the current node's action. Continue as such across all nodes (adjacent) whose parent is the same as the current node's parent.

(3) The next time the parent of the current node does not match the parent that was being processed, examine the accumulation buffer to find the action with the optimal expected reward. Set that parent's reward to this optimal value. If that parent is the root node, set the root node's action to the action that resulted in the optimal reward. Note that the optimal value depends on the current parent's strategy (minimize vs. maximize expected reward). Dynamic strategies are especially useful for modeling multi-player environments.

The module simply outputs the reward and action fields of the root node. To indicate that these values have (potentially) changed, the module also pulses a signal during the cycle in which the root node's values are updated.

2.2 Planning Controller

The controller acts as an interface between software and the execution module. As such, it must accomplish two primary tasks: (1) accept commands from software and (2) drive inputs to the execution unit/capture outputs and make them accessible to software.

2.2.1 Communication Protocol

The data exchange with software is implemented via memory-mapped I/O. In this implementation, we chose to use AXI as the physical mechanism for communication, but this is subject to change in future iterations. As a start, we have implemented a simple FIFO that emulates AXI. An immediate piece of future work is to integrate a proper AXI IP into the project.

Via the software interface to the controller, the processor writes commands to a command register in the controller as 64-bit values. Commands are generally of the format:

$value_{63-0} = command_{1-0} \,||\, data_{61-0}$

The `command` field is 2-bits long and can take on the following values:

- 00: Run computation
- 01: Set node data
- 10: Set config data

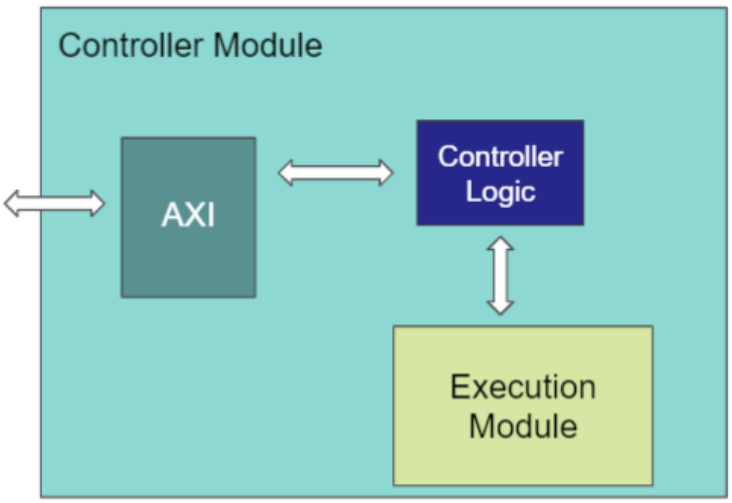
The `data` field is 62-bits long and is dynamic based on the provided `command`. When running a computation (00), no data is necessary and thus the field can be treated as all x.

When setting node data, the `data` field takes the following format:
$data_{61-0} = address_{9-0} \,||\, type_{1-0} \,||\, val_{49-0}$
`address` specifies which node should be updated, `type` specifies which field of that node is being updated (00: parent, 01: action and strategy, 10: reward, 11: weight) and `val` contains the value to update that field with.

When setting config data, the `data` field takes the following format:
$data_{61-0} = type_{1-0} \,||\, val_{59-0}$
`type` specifies which config field is being updated (00: number of nodes) and `val` contains the value to update that field with.
This protocol allows software to dynamically change the tree structure and parameters as necessary based on the overall state in the system as a series of writes. When the controller receives the command to run the computation, it resets the execution unit and waits for the pulse indicating that the computation has finished. Upon receiving this pulse, it writes the output data to an output register (in the controller) and generates an interrupt. Software handles this interrupt by reading the contents of the output register in the controller to obtain the desired results. We are also exploring the notion of including the desired data in the interrupt or some similar tactic to improve performance.

2.3 Block Diagram

Below is a high-level block diagram of both the controller and execution unit.



2.4 Software Interface to the Planning Controller

We define a set of helper functions for software applications to communicate with the treeval controller. Implementing the communication protocol discussed in the previous section, the functions format and write the appropriate 64-bit command to the emulated AXI buffer for the following functionalities: reconfiguring the number of nodes in the tree, adding a node to the tree, setting the reward/action/weight of a node, starting the treeval computation, and retrieving a decoded result from the computation.

2.5 Synthesis Results

The following high-level resource utilization was reported from synthesis. It is briefly discussed below.

Instance	Module	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP Blocks
treeval_controller	(top)	36218	36218	0	0	33990	0	0	0
(treeval_controller)	(top)	90	90	0	0	620	0	0	0
axi	axi_fifo_dummy	144	144	0	0	326	0	0	0
execution_unit	treeval	35984	35984	0	0	33044	0	0	0

The LUT count is very high for execution module (treeval). Our hypothesis is that the node buffer is the culprit (1024x32 bit structure). An immediate next step is to (1) investigate this usage report further and (2) map the findings onto inefficiencies in the hardware. This is likely not too high effort, but due to time constraints, we have chosen to focus elsewhere for the scope of this initial report.

3 Software Implementation

3.1 Tree Specification

We used a JSON file as mean for the user to define a tree, the reason being that such files offer flexibility of specification and allow for nested structures. This allows for construction of a tree using a single JSON file, with the first node representing the root of the tree, each nested node representing the immediate children of the parent containing them. The structure of the JSON file mirrors that of the tree, also allowing for easy generation of trees programmatically for more expansive trees.

A JSON node contains:

- id: A marker, specifying which action this node belongs to
- val: Reward (positive or negative) for landing on this node from its parent
- weight: The probability of landing on this node from its parent, such that all nodes with the same id sum to 1
- children: An array of JSON nodes, each being an immediate child to the current node

3.2 Tree Generation

Generating the tree is done by loading the JSON file and parsing it in software similar to a Depth First Search. We extract the JSON node's data to construct the current node, then construct the children of the current node recursively. For the software implementation, we create a Node object, holding data and an array of pointers to its children. For the hardware implementation, we use the interface described in section 2.2.1 to create nodes objects directly on the hardware.

3.3 Tree Evaluation

Evaluating the tree consists in traversing the Node class tree in a Depth First Search pattern. For each node, we compute its expected value from its weight and value fields. We then back-propagate this value to the parent which will in turn merge all expected rewards of children from a same action to compute the expected value per action. The node then computes its own expected value using all its expected values per action. Once all back-propagation steps are performed, the root node contains the expected rewards one can obtain from playing the game.

4 Evaluation vs. Software Implementation

4.1 Analysis of Hardware Performance

Before addressing specific test cases, we decided that it is worth describing an analytical model of the hardware performance. This analytical model can be broken into two components: (1) performance of execution unit on a pre-configured tree and (2) performance of the controller to configure a tree. Each will be detailed in the sections to follow.

4.1.1 Performance of Execution Unit

For this analysis, we will assume that the pre-configured tree consists of `n` nodes. The first cycle is a reset cycle to begin the computation for the correct starting point. After this, every node other than the root node is processed. Processing at a given node takes 1-3 cycles, depending on the tree structure. If the current node is the first node encountered for a given parent, 1 cycle is spent clearing the action accumulation buffer. If the current node is the last node encountered for a given parent, 1 cycle is spent computing the optimal expected reward and action for that parent. Regardless, every node spends 1 cycle computing its contribution to its action's expected reward. As such, we can define an upper and lower bound on the number of cycles required for execution:
Upper: $3(n-1)+1 = 3n-2$
Lower: $(n-1)+1 = n$

4.1.2 Performance of Configuration

The analysis of the configuration by the controller must take part in two steps: (1) configuring the execution unit and (2) capturing inbound config messages from the AXI FIFO.

Under the same assumptions as the prior section, each node other than the root requires 1 cycle per value being configured. These values are: parent, action/strategy, reward, weight. Thus, at most 4 cycles are required per node. However, it can be expected that many nodes are not leaf nodes and thus do not require a reward. In addition, 1 cycle is required to configure the number of nodes in the tree and 1 cycle is needed to set the strategy of the root node. In total, this offers upper and lower bounds of:
Upper: $4(n-1)+2 = 4n-2$
Lower: $3(n-1)+2 = 3n-1$

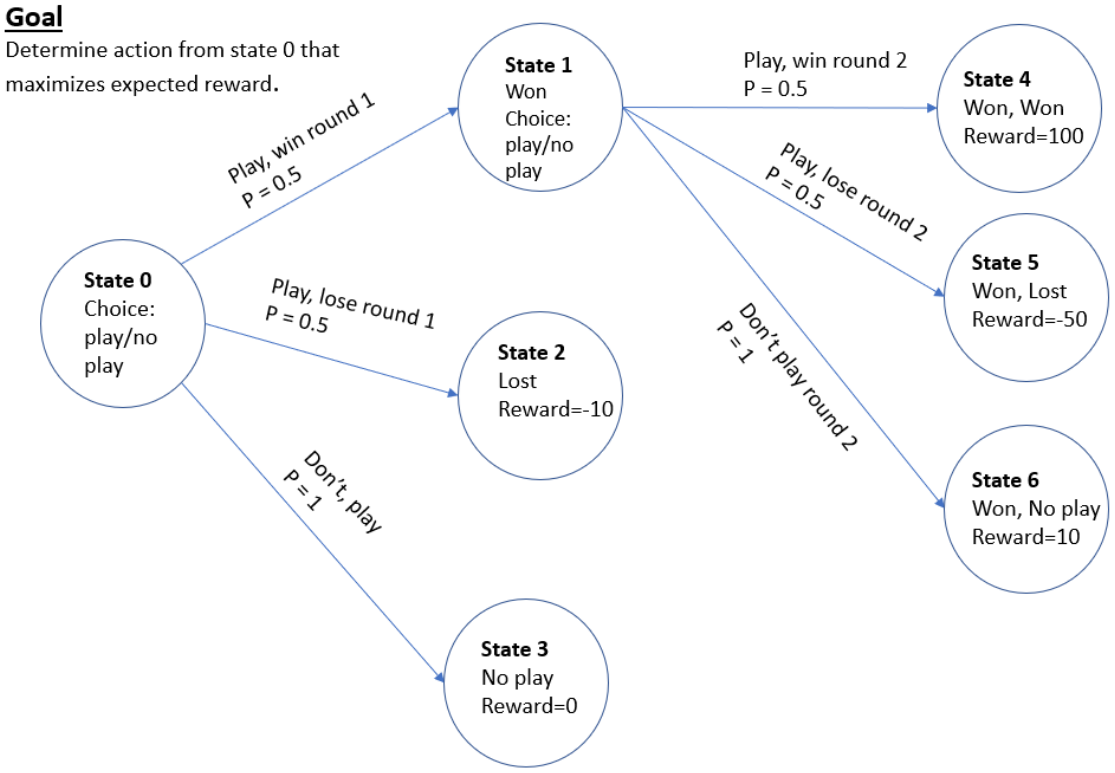
Next, we must analyze the message passing step from software. Every configuration command detailed above is sent as a message, which takes 1 cycle (at least, but we will ignore extra delay due to FIFO being full and/or software being busy). Once the message is in the FIFO, the following processing must occur: 1 cycle to capture the message from the FIFO and 1 cycle to process the message. Thus, the bounds above can be adjusted to:
Upper: $3(4n-2) = 12n-6$

Lower: $3(3n-1) = 9n-3$

Adding in the cycles required by the execution unit as well as an additional 2 cycles to push the final results as an outgoing message into the FIFO, we end up with the following upper and lower bounds:
Upper: $3n-2+12n-6+2 = 15n-6$
Lower: $n+9n-3+2 = 10n-1$

4.2 Example: Two Round Game

Here we illustrate an example planning tree that could be generated by an RL system solving a two round game. At the current time step, the game is in state 0, with two actions to choose from: play, or no play. The probability distribution of outcomes is used to generate a tree up to two time steps into the future. The leaf node with the highest expected reward can be identified and backtracked from to select the optimal action to take at the current time step.



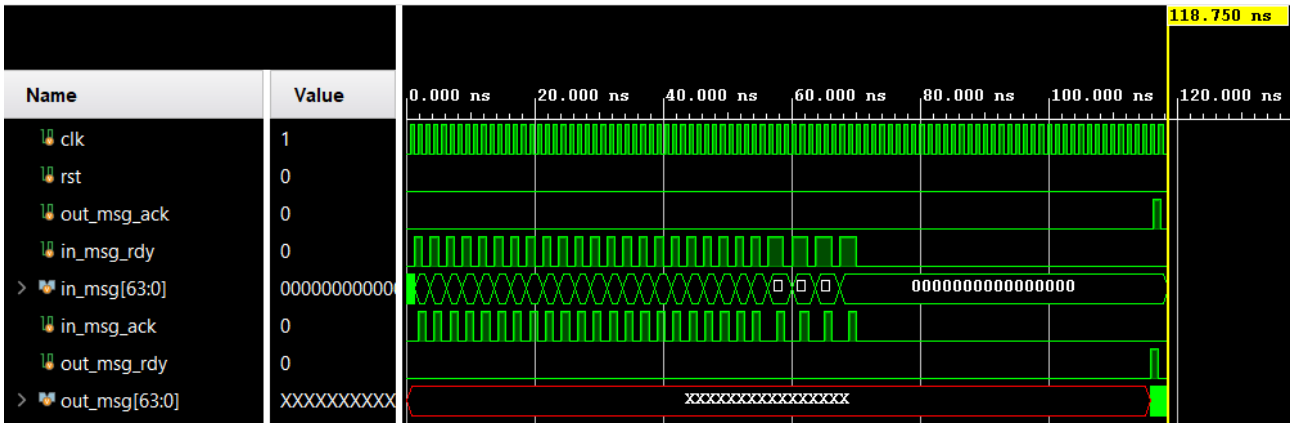
4.3 Example: Betting Game

In order to evaluate a more comprehensive tree, better able to showcase the performance difference between the software and hardware implementation, we implemented the following game. The player can choose whether or not to play. If the player does not play, it ends the game and the player keeps its accumulated gain. Playing involves flipping a biased coin whose probability distribution is unknown. If the player loses with probability $1-p$, the game ends and you lose 100 points. On the contrary, winning with probability p awards 100 points, granting the option to play another round. The subsequent round is similar in structure to the previous round, except that the probability distribution of the coin changes to $q \neq p$.

4.3.1 Hardware Performance: Simulation

Executing the simple two round example above entirely in hardware (i.e., the simulator generates the correct messages to send to the controller, not software) yields the following high-level performance:

Prediction execution cycles (upper bound): 99 cycles
Prediction execution cycles (lower bound): 69 cycles
System clock: 800MHz
Actual execution time: 119ns = 95 cycles



4.3.2 Software Performance

While it is difficult, if not impossible, to set a CPU clock speed, for the purposes of our experiment we have found it to fluctuate between 800-3600 MHz. To give a worst case evaluation of the hardware, we assume that our software runs at 800 MHz as well.

The software performance is predictably slower than that of the hardware. At 800 MHz, the tree evaluation takes an average of 5000 ns. We found that this is primarily due to the tree traversal. Therefore, the significant speedup of the hardware implementation can be attributed to its fast access to nodes in memory.

5 Future Work and Current Limitations

The subsections to follow discuss specific limitations and their corresponding items for future work. Some are shorter-term while others are much longer.

5.1 Action Scalability

As discussed earlier, the hardware currently supports a maximum of 8 distinct actions that can be taken from any node. The reason for this is that we wanted to impose a strict 1 cycle latency maximum on the step of computation that determines the optimal action by comparing all existing expected rewards. A large set of actions makes this increasingly difficult while maintaining a desirable cycle time. Future work surrounding this subject is to look into mechanisms of sorting the action buffer so as to reduce the amount of work that must be done in the final step of computation. This can allow for a much larger buffer without having to pay as large of a penalty in the context of sequential combinational logic.

5.2 Tree Scalability

As discussed earlier, the hardware currently supports a maximum of 1024 distinct nodes in the tree. There are two areas of work/research to improve upon this: (1) allow the controller to instantiate multiple execution units such that larger trees can be formed and (2) determine what the optimal number of nodes is to pack into a single execution module. Of course, there is some practical limit to the number of nodes even when instantiating multiple modules and/or increasing the maximum capacity. As such, we must also try to understand what a practical target is from the perspective of real-world applications.

5.3 Expected Reward Quantization

One performance optimization is to eliminate the need for divides and floating point multiplies by representing probabilities as base-128 integers and shifting appropriately. While this is much more energy-efficient, there are a few limitations: (1) we effectively perform integer division with expected rewards, meaning that expectations that differ only the decimal places will be treated as equal by the hardware and (2) we have limitations on the representable rewards and probabilities. (2) is likely not too much of an issue in practice as the software can likely normalize as necessary and doesn't need terribly high precision. (1) is more problematic, in that actions that inherently aren't equal can be treated as such. We are looking into efficient floating point operations to replace the existing functionality.

5.4 Integration of AXI IP

The initial implementation relies on AXI as the physical communication mechanism between the controller and software. We have a few action items on this front. (1) in the interest of time given various licensing constraints, we decided to implement a simple FIFO that emulates the AXI IP. An immediate next step is to integrate actual AXI IP's into the design or, if our FIFO emulation suits the desired performance, integrate the CPU side appropriately. In addition, it is worth researching whether or not this is indeed the most desirable communication mechanism - especially if the goal is to integrate into more commercial systems.

5.5 Integration Testing with CPU

Though many of the pieces of the system are in place, we did not have time to implement the full system in hardware. The main piece that is missing is a soft core that can be placed into an FPGA-based design that can run the necessary code to build trees and capture results. Our plan is to use MicroBlaze as this soft core, but due to various licensing constraints we were not able to get a full system working in time. The major purpose of this piece is to properly characterize the expected performance of the system, which as yet has only been approximated.

5.6 Integration of Other RL Subsystems

The reinforcement learning system described at the beginning of this paper consists of much more than just the planning accelerator. Two other major system components that could be accelerated are (1) the policy/world hypothesis generator, which is typically a fully-trained DNN and (2) the training mechanism for this DNN. We have discussed accelerators for both inference and training throughout the quarter and hope to propose efficient hardware based on prior research to integrate into our system. In addition, the software piece that unites these accelerators by computing real-world rewards and re-configuring the system accordingly would need to be implemented. An extension of our system software seems like a plausible approach to tackle this next step.

6 Conclusion

We propose CARL as a way to accelerate the evaluation of the planning tree in a reinforcement learning system. The planner accelerator comprises of an execution module and an interface to properly configure the tree. To show the improvements that such an accelerator can achieve, we implement a simple game in software. Our results show that CARL is able to evaluate a tree several orders of magnitude faster than a C++ software implementation is able to. We leave for future work increasing the scalability of distinct actions, the size of the tree, optimizing performance, and integrating the AXI IP in order to implement and test the full system in hardware.

7 Statement of Work

Daniel implemented the tree evaluation in software and the example use of the interface to the hardware accelerator. Adrien implemented the JSON parsing and tree generation in software. Josh implemented the planning execution module, the AXI dummy module in hardware, and ran simulations and synthesis for all hardware modules. Hannah implemented the planning controller module and the software interface to the accelerator. We collaborated on preparing for the presentation and producing this report.

References

[Kargar 2019] Kargar, Isaac: *RL Series-REINFORCE*. 2019. – URL `https://kargarisaac.medium.com/rl-series-reinforce-in-pytorch-98102f710c27`

[Xilinx 2012] Xilinx: *AXI Reference Guide*. 2012. – URL `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf`